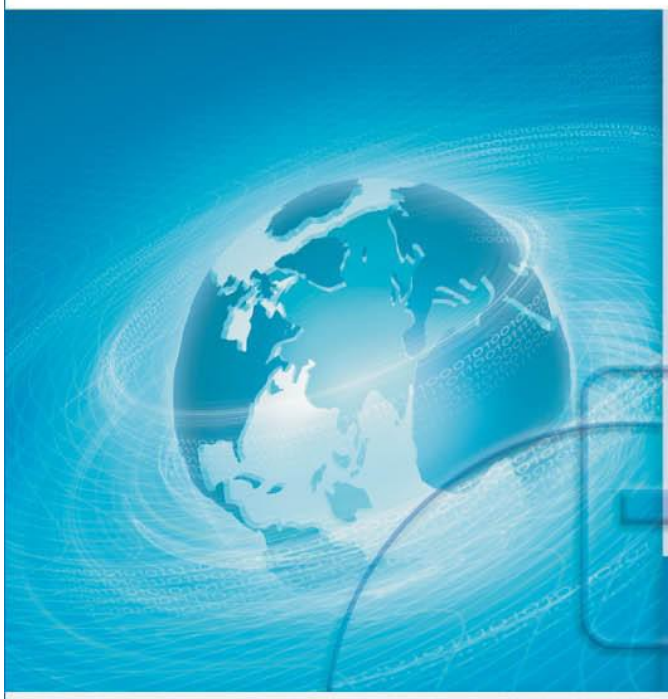
Microsoft

アプリケーション アーキテクチャ ガイド 2.0



patterns & practices

著作権

このドキュメントに記載されている情報 (URL 等のインターネット Web サイトに関する情報を含む) は、将来予告なしに変更することがあります。別途記載されていない場合、このソフトウェアおよび関連するドキュメントで使用している会社、組織、製品、ドメイン名、電子メール アドレス、ロゴ、人物、場所、出来事などの名称は架空のものです。実在する商品名、団体名、個人名などとは一切関係ありません。お客様ご自身の責任において、 適用されるすべての著作権関連法規に従ったご使用を願います。このドキュメントのいかなる部分も、米国 Microsoft Corporation の書面による許諾を受けることなく、その目的を問わず、どのような形態であっても、複製または譲渡、あるいは検索システムに格納または公開することは禁じられています。ここでいう形態とは、複写や記録など、電子的な、または物理的なすべての手段を含みます。

マイクロソフトは、このドキュメントに記載されている内容に関し、特許、特許申請、商標、著作権、またはその他の無体財産権を有する場合があります。別途マイクロソフトのライセンス契約上に明示の規定のない限り、このドキュメントはこれらの特許、商標、著作権、またはその他の無体財産権に関する権利をお客様に許諾するものではありません。

© 2009 Microsoft Corporation. All rights reserved.

Microsoft、MS-DOS、Windows、Windows NT、Windows Server、Active Directory、MSDN、Visual Basic、Visual C++、Visual C#、Visual Studio、および Win32 は、米国 Microsoft Corporation の米国およびその他の国における登録商標または商標です。

記載されている会社名、製品名には、各社の商標のものもあります。

序文 ～ S.Somasegar

私のチームでは、独自のテクノロジを使用してマイクロソフト製品を開発したり、ユーザーやパートナーと日常的に交流したりする中で、マイクロソフト テクノロジを使用したアプリケーション アーキテクチャ、設計パターン、および設計原理にベスト プラクティスを適用するための実用的なガイダンスを策定しました。このガイダンスは、開発者とソリューション アーキテクトの両方に有用です。このガイド『マイクロソフト アプリケーション アーキテクチャ ガイド』は、マイクロソフト社内で培ったノウハウ、社外の専門家、ユーザー、およびコミュニティから得たガイダンスを開発者とソリューション アーキテクトの皆さんと共有するために統合したものです。

このガイドは、ソリューション アーキテクトと開発者が、マイクロソフト プラットフォームで実行するより効果的なアプリケーションの設計と開発ができるようになることと、新しいプロジェクトの初期段階における重要な判断をサポートすることを目的としています。また、アーキテクトと開発者が既存のソリューションを強化するのに役立つ特殊なトピックに関するコンテンツも提供しています。このガイドには、25 人以上のマイクロソフト社外の専門家とユーザーによる寄稿とレビューが含まれています。

アーキテクチャ パターン、アーキテクチャの原理、品質特性、横断的関心事の観点からソリューションについて考えると、ソリューションを開発する際に役立つ、基準となるアプリケーション アーキテクチャや関連するテクノロジ、パターン、ガイダンス資産をすばやく決定できます。また、このガイドを使用して、アプリケーション アーキテクチャの重要な点を特定すると、シナリオに合わせて、そこを改良できます。

このガイドでは、一般的な種類のアプリケーションの参照アプリケーション アーキテクチャ、品質特性と横断的関心事のガイドライン、およびソリューション アーキテクチャの設計と改良に役立つ設計の手法のガイドラインを提供します (ここで言う一般的な種類のアプリケーションには、Web アプリケーション、リッチ クライアント アプリケーション、リッチ インターネット アプリケーション (RIA)、モバイル アプリケーション、サービス アプリケーションなどが含まれます)。

『マイクロソフト アプリケーション アーキテクチャ ガイド 第 2 版』が、適切なアーキテクチャ、適切なテクノロジ、および関連するパターンを選択して、効果的な設計を決定する一助となれば幸いです。

マイクロソフト

Developer Division、シニア バイス プレジデント

S. Somasegar

序文 ～ Scott Guthrie

アプリケーション アーキテクチャは難易度の高いトピックです。事実、アプリケーション アーキテクチャに関するさまざまな書籍、記事、およびホワイトペーパーからも、そのことが見て取れます。開発者やアーキテクトにとっても、マイクロソフト プラットフォームのアーキテクチャとベスト プラクティスの設計を理解することはたいへん困難です。このガイドの前身である『Application Architecture for .NET: Designing Applications and Services』では、このトピックをわかりやすく説明していましたが、この書籍が執筆されたのは 2002 年と少し前のことになってしまいました。

2002 年以降に誕生した多くのテクノロジについて扱うために、J. D. Meier、David Hill、および Microsoft patterns & practices チームは、アプリケーション アーキテクチャに関する新しいガイドを作成しました。このガイドでは、最新のベスト プラクティスとテクノロジに基づいた、マイクロソフト プラットフォームで実行するアプリケーションやサービスの設計に関する洞察力に富んだガイダンスが提供されています。その結果、ソリューション アーキテクトと開発者が、マイクロソフト プラットフォーム対応のアプリケーションを効果的に設計するのに役立つ『マイクロソフト アプリケーション アーキテクチャ ガイド第 2 版』が誕生しました。このガイドでは、.NET Framework とマイクロソフト プラットフォーム、これらで使用されている主要テクノロジと機能を概要します。また、プラットフォームに依存しない、パターン指向の、原理に基づいたガイダンスも提供しているので、堅牢な基盤に基づいてアプリケーションを設計するのに役立ちます。

このガイドは、アプリケーション アーキテクチャの構造を構成する多数の重要なアーキテクチャと設計の原理に基づいています。また、設計における重要な判断とその対処に関するガイドラインと、品質特性、横断的関心事、アプリケーション アーキテクチャを形成する機能についても説明します。アプリケーション アーキテクチャを形成する要素には、パフォーマンス、セキュリティ、スケーラビリティ、管理容易性、展開、通信などがあります。

このガイドでは、メタ レベルにおいて、ソリューション アーキテクトが考慮する必要があるティアとレイヤーの概要について解説しています。各ティアと各レイヤーについては、焦点、機能、一般的な設計パターン、およびテクノロジの視点から説明しています。その後、この情報を背景として、関連する原理、パターン、およびプラクティスへと展開します。最後に、正規のアプリケーションの規範を提示して、一般的なアプリケーションの種類について解説します。各規範については、対象のシナリオ、関連するテクノロジ、パターン、およびインフラストラクチャの観点から説明します。

このガイドの内容は、基本的に、マイクロソフト社内の専門家、マイクロソフト パートナー、ユーザー、およびコミュニティで蓄積されたノウハウに基づいています。そのため、マイクロソフト プラットフォームについての理解を深め、適切なアーキテクチャとテクノロジを選択して、実証済みの慣例と教訓を使用してアプリケーションを開発するのに役立ちます。

マイクロソフト
.NET 開発者プラットフォーム部門、コーポレート バイス プレジデント
Scott Guthrie

序文 ～ David Hill

冗談好きな開発者の間で流行っていた古いジョークに「アーキテクトとして見てもらうために必要なことは、技術的な質問を受けたときに "場合によります" と答えるだけだ」というものがありました。「ソリューションに認証と承認を実装するには、どの方法が最適ですか? - 場合によります」、「データ アクセス レイヤーを実装するにはどうすればよいですか? - 場合によります」、「ソリューションの UI には、どのテクノロジを使用すればよいですか? - 場合によります」、「どうすればアプリケーションにスケーラビリティを持たせることができますか? - 場合によります」という具合です。これで、このジョークの概要はおわかりいただけたと思います。
事実、言うまでもなく技術的な質問への回答は場合によって異なります。結局、すべてのソリューションは異なっていて、多くの要素が含まれており、技術的な要素もそうでない要素もあります。ソリューションの規模にかかわらず、このような要素は、ソリューションのアーキテクチャと設計に大きな影響を及ぼします。開発者とソリューション アーキテクトは、ソリューションを開発する際に使用するテクノロジとツールはもちろんのこと、ビジネス、エンド ユーザー、組織の IT 環境、管理のインフラストラクチャ、および経済環境によって課される、(しばしば矛盾する) 要件と制約のバランスを取る必要があります。
このような要件と制約は、新しい機会が生じたり、システムに新たな要求が課されたりするたびに進化しており、IT にかかわる生活のスパイスとなっています。ビジネス ルールの変化や新たなビジネス領域の出現は、新しいアプリケーションと既存のアプリケーションの両方に影響を及ぼします。ユーザーは時間の経過と共に、優れた、一貫性のある、密接に統合されたユーザー エクスペリエンスを期待するようになります。それを受けて、新たなコンプライアンスの要件が発生したり、コスト削減または可用性やスケーラビリティの向上を実現する新しい IT インフラス

トラクチャ テクノロジが誕生したりします。また、開発コストの削減や以前は実装が困難だったシナリオの実現を約束する、新しいテクノロジ、フレームワーク、およびツールも絶えずリリースされています。

これらをすべて理解しながら、効果的なソリューションを予算内でスケジュールどおりに実現することが、簡単な作業でないのは明らかです。そのためには、開発者とソリューション アーキテクトが、競合および重複する多くの要素 (技術的ではないものも含まれます) を考慮し、すべての要素間のバランスを取る必要があります。あまりにも多くの要素を考慮しようとすると、設計が過剰で複雑なソリューションになります。そのようなソリューションは、構築するのに時間がかかるにもかかわらず、耐久性と柔軟性を強化するという目的を果たしません。反対に少しの要素しか考慮しないと、制限された、柔軟性のない、粗製のソリューションとなり、そのようなソリューションは発展させるのが困難で、うまく拡張できません。つまり、多くの場合、開発者とソリューション アーキテクトは、"全体的なソリューション" と "部分的なソリューション" の中庸を取る必要があります。

これは、アプリケーション アーキテクチャの真髄を表しているのではないでしょうか。つまり、今日のツールとテクノロジを使用して、できるだけ多くのビジネス価値を生み出しながら、スケーラビリティ、柔軟性、保守容易性により現行の価値を最大限に高めるために、今日のビジネスによって課される要件や制約と将来かされることが予想される要件や制約の両方に注意を払う必要があるということです。開発者またはソリューション アーキテクトがアーキテクチャの原理とパターンについて熟知すると、ソリューションの全体的な成功の鍵を握る全体的な設計プロセスと設計に関する重要な問題を理解して考慮できるようになります。このような知識を習得することにより、多くの情報に基づいて決断を下したり、競合または重複する要件と制約の適切なバランスを取ったりすることが可能になります。また、ソリューションがビジネスの目標を達成または上回るだけでなく、コスト効率のよい、拡張性、保守容易性、および柔軟性のある方法で目標を達成するようにできます。

これが、開発者とソリューション アーキテクトの両者にかかわることにはお気付きでしょう。どちらも、このガイドで概要を説明している、アーキテクチャのパターンと原理を着実に理解することにより、大きなメリットを得られます。実装の詳細は、全体的な設計ほど重要ではないと考える方もいるかもしれませんが、個人的な経験から言うと、そうではありません。小さな判断も、時間の経過と共に蓄積されます。実装レベルの詳細は、全体的なソリューション アーキテクチャと、そのスケーラビリティ、保守容易性、柔軟性に大きな影響を及ぼすので、開発者とソリューション アーキテクチャは、これを着実に理解する必要があります。また、理解を共有することで、円滑なコミュニケーションが行われるというメリットもあります。

このガイドは、適切な判断を下して優れたソリューションを構築するうえで役立つ、アプリケーション アーキテクチャ、設計原理、および設計パターンの概要を提供することを目的としています。また、最初から最後まで順番に読んだり、参考文献として使用して、関連のあるセクションを直接参照したりできるような構成になっています。ガイドの前半では、一般的に適用できるアーキテクチャと設計原理に重点を置いて説明します。そのアーキテクチャと設計原理は、すべての種類のソリューションに適用されます。後半では、Web アプリケーション、リッチ クライアン

ト アプリケーション、モバイル アプリケーションなどの、一般的なアプリケーションの種類に重点を置いて、各種類の一般的なアーキテクチャや設計に関する重要な考慮事項について説明します。このガイドで取り上げるアーキテクチャは、お手持ちのソリューションには、直接対応しないかもしれませんが、個々の状況に合わせて使用および発展できるベースライン アーキテクチャに関する情報を提供しています。アーキテクチャの重要な要素を特定する方法についてのアドバイスを提供しているので、徐々にアーキテクチャを改良できます。

このガイドは、.NET Framework をベースとしたマイクロソフト プラットフォームで実行するソリューションを開発することに特化しているので、関連するテクノロジとツールの詳細情報を提供する記事とリソースを関連情報として紹介しています。ですが、基盤となる原理とパターンは、一般的にどのプラットフォームにも適用できるということがおわかりいただけると思います。このガイドは、アプリケーションのアーキテクチャと設計におけるすべての側面に関する完全で包括的な資料ではありません。このような資料を作成するには、ガイドの内容を大幅に増やすか、ガイドを何巻も作る必要があります。そのため、このガイドでは、重要なトピックの実用的な概要を、詳細なガイダンスや詳細な資料へのリンクと併せて提供しています。

アプリケーションのアーキテクチャと設計の分野は、動的で常に進化しています。これまでにソリューションの構築を成功に導いた基盤は、今後も当面は役立ちますが、テクノロジと新しい設計手法の両方において、革新の速度が衰えないことについても備える必要があります。マイクロソフト プラットフォームと .NET Framework、これらがサポートするテクノロジとシナリオの範囲は深く広大で、その程度は日々増しています。しかし、予測されるものが現実になるのを待つ必要はなく、今すぐ魅力的で有益なソリューションを構築することは可能です。ぜひ、このガイドをその作業にお役立てください。

patterns and practices

David Hill

2009 年 9 月

ガイドの概要

このガイドは、開発者とソリューション アーキテクトが、十分に試行されて信頼できるアーキテクチャ、設計原理、およびパターンを活用することによって、マイクロソフト プラットフォームと .NET Framework で実行する効果的で高品質のアプリケーションを少ないリスクですばやく構築できるようにすることを目的としています。

このガイドでは、優れたアプリケーション アーキテクチャとアプリケーション設計の堅固な基盤を提供する基になる原理とパターンの概要を紹介します。この基盤の概要を背景として、アプリケーションの機能をレイヤー、コンポーネント、およびサービスに分ける汎用的なガイダンスを提供します。さらに、ソリューションの重要な設計特性、重要な品質特性 (パフォーマンス、セキュリティ、スケーラビリティなど)、および横断的関心事 (キャッシュ、ログなど) の特定と対処についてのガイダンスも提供します。また、さらに一歩踏み込んで、Web アプリケーション、リッチ インターネット アプリケーション (RIA)、リッチ クライアント アプリケーション、サービス アプリケーション、モバイル アプリケーションなどの、一般的な種類のアプリケーションのアーキテクチャと設計に特化したガイダンスも提供します。

このガイドを読むと、次のことができるようになります。

- マイクロソフト プラットフォームで実行する正常なソリューションを開発するための基盤となるアーキテクチャと設計の原理およびパターンを理解する
- ソリューションのレイヤー、コンポーネント、およびサービスを設計する際に役立つ適切な戦略と設計パターンを特定する
- ソリューションの主要な設計上の判断事項を特定して対処する
- ソリューションの重要な品質特性と横断的関心事を特定して対処する
- ソリューションに適したテクノロジを選択する
- ソリューションのベースライン アーキテクチャ候補を作成する
- patterns & practices ソリューションの資産とソリューションを実装する際に役立つガイダンスを見つける

このガイドでは広範囲を扱っていますが、アプリケーションのアーキテクチャと設計の分野に関する完全で包括的な専門書ではなく、マイクロソフト プラットフォームと .NET Framework で実行するソリューションのアーキテクチャと設計に関する一般的な原理についての参考資料や実用的で便利な概要情報を提供する資料となることを目的としています。

そのため、このガイドでは、特定のシナリオに特化した具体的または正式なソリューション アーキテクチャは提示しません。その代わりに、優れたアーキテクチャと設計の基盤となる原理やパターンの簡潔な概要を提供し、ユーザーが直面するいくつかの重要な問題に関する推奨事項を提示します。

ガイドの大部分は、テクノロジに依存しない原理に基づいた内容となっているので、すべてのプラットフォームやテクノロジに適用できます。ただし、使用できるテクノロジ中から適切なものを選択したり、特定の状況でテクノロジを最大限に活用したりするうえで役立つことが予想される、マイクロソフト テクノロジと .NET Framework テクノロジに関する考慮事項を含めてあります。

対象読者

このガイドの主な対象読者は、マイクロソフト プラットフォームと .NET Framework で実行するアプリケーションの設計に関するガイダンスを必要としている開発者とソリューション アーキテクトです。
ですが、アプリケーションのアーキテクチャと設計に興味のある技術者の方、マイクロソフト プラットフォームまたは .NET Framework で実行する優れたアプリケーションの設計で採用されている基盤となるパターンや原理を知りたい技術者の方、またはマイクロソフト プラットフォームや .NET Framework になじみのない技術者の方にも役立つ情報を提供しています。

このガイドの使用方法

このガイドは、アプリケーションのアーキテクチャと設計について順を追って説明するチュートリアルではなく、それらの概要を提供する参考資料です。このガイドは次の 4 つのセクションで構成されており、各セクションは複数の章で構成されています。

- 1 つ目のセクション「ソフトウェアのアーキテクチャと設計」では、優れたアプリケーションのアーキテクチャと設計の基盤およびアーキテクチャを設計する際に推奨される手法を提示する基本原理とパターンの概要を提供します。アプリケーション アーキテクチャの基礎について学ぶ場合は、このセクションから読み始めます。2 つ目以降のセクションを読み進めると、レイヤー型の設計、コンポーネント、品質特性、横断的関心事、通信、展開、および一般的なアプリケーションの種類について理解できます。
- 2 つ目のセクション「設計の基礎」では、ソリューションのレイヤー、コンポーネント、サービスを設計する際に一般的に適用できるガイダンスと、品質特性および横断的関心事への対処に関するガイダンスを提供します。また、通信と展開に関するトピックについても取り上げます。アプリケーションのアーキテクチャと設計のレイヤー型の手法や特定のコンポーネントとサービスの設計について学ぶ場合は、このセクションから読み始めます。3 つ目以降のセクションを読み進めると、品質特性を考慮したり、物理的な展開の戦略を設計したりする方法について理解できます。
- 3 つ目のセクション「アプリケーションの原型 (アーキタイプ)」では、Web アプリケーション、RIA、リッチ クライアント アプリケーション、モバイル アプリケーション、サービス アプリケーションなどの、一般的な種類のアプリケーションのアーキテクチャと設計に関する具体的なガイダンスを提供します。アプリケーションのアーキテクチャと設計については多少なじみがあり、一般的な種類のアプリケーションのアーキテクチャと主な設計の特徴や各アプリケーションの種類についての具体的な

ガイダンスについて学ぶ場合は、このセクションから読み始めます。4 つ目のセクションまで読み進めると、新しい知識を習得したり、既存の知識を確認したりできます。

- 4 つ目のセクション「付録」では、マイクロソフト プラットフォーム テクノロジと .NET Framework テクノロジ、それらの機能の概要を提供します。このセクションでは、一般的な設計パターンの概要や、参考資料として追加のリソースと資料の情報も提供しています。.NET Framework になじみがない場合やマイクロソフト プラットフォームで使用できるテクノロジについて学ぶ場合は、このセクションから読み始めます。このセクションでは、.NET Framework やプラットフォームで提供されるサービスの概要を確認したり、主要なテクノロジのマトリックスについて理解したり、Enterprise Library や patterns & practices の設計パターンに関するライブラリなどの patterns & practices ソリューションの資産についての情報を得られます。

経験と目的に応じて、ニーズに最適なセクションを直接参照することもできます。また、ガイドを最初から最後まで読むと、マイクロソフト プラットフォームと .NET Framework の設計とアーキテクチャに関する広範囲に及ぶ概要を理解できるので、アーキテクチャと設計の手法を理解するのに役立ちます。このガイドは、アプリケーション開発ライフサイクルとアプリケーション開発プロセスに取り入れたり、トレーニング ツールとして使用したりすることもできます。

フィードバックとサポート

このガイドの正確性を徹底するために、あらゆる努力を重ねましたが、このガイドのトピックへのフィードバックがございましたら、ぜひお寄せください。推奨事項、有用性、およびユーザビリティに関する技術的な問題、記述や編集に関する問題に関するフィードバックをお送りください。さまざまな Web リソースに簡単にアクセスするには、オンライン版の参考文献 (http://www.microsoft.com/architectureguide、英語) を参照してください。
このガイドについてのフィードバックについては、Application Architecture Guide コミュニティ サイト (http://www.codeplex.com/AppArchGuide、英語) からお寄せください。

技術サポート

このガイドで言及しているマイクロソフト製品とマイクロソフト テクノロジに関する技術サポートは、マイクロソフト製品サポート サービス (PSS) で提供しています。製品のサポート情報については、マイクロソフト サポート オンラインの Web サイト (http://support.microsoft.com) を参照してください。

コミュニティとニュースグループによるサポート

MSDN 管理ニュースグループのサイト (http://msdn.microsoft.com/ja-jp/subscriptions/aa974230.aspx) では、コミュニティからサポートを受けたり、ガイドについて議論したりすることもできます。また、フィードバックもお寄せいただけます。

このガイドの執筆チーム

このガイドの執筆は、次の .NET アーキテクチャと開発の専門家が担当しました。

J.D. Meier

David Hill

Alex Homer

Jason Taylor

Prashant Bansode

Lonnie Wall

Rob Boucher Jr.

Akshay Bogawat

寄稿者とレビュー担当者

以下の寄稿者とレビュー担当者に感謝します。

テスト チーム: Rohit Sharma、Praveen Rangarajan

編集チーム: Dennis Rea

マイクロソフト社外の寄稿者とレビュー担当者: Adwait Ullal、Andy Eunson、Brian Sletten、Christian Weyer、David Guimbellot、David Ing、David Weller、David Sussman、Derek Greer、Eduardo Jezierski、Evan Hoff、Gajapathi Kannan、Jeremy D. Miller、John Kordyback、Keith Pleas、Kent Corley、Mark Baker、Paul Ballard、Peter Oehlert、Norman Headlam、Ryan Plant、Sam Gentile、Sidney G Pinney、Ted Neward、Udi Dahan、Oren Eini aka Ayende Rahien、Gregory Young

マイクロソフト社内の寄稿者とレビュー担当者: Ade Miller、Amit Chopra、Anna Liu、Anoop Gupta、Bob Brumfield、Brad Abrams、Brian Cawelti、Bhushan Nene、Burley Kawasaki、Carl Perry、Chris Keyser、Chris Tavares、Clint Edmonson、Dan Reagan、David Hill、Denny Dayton、Diego Dagum、Dmitri Martynov、Dmitri Ossipov、Don Smith、Dragos Manolescu、Elisa Flasko、Eric Fleck、Erwin van der Valk、Faisal Mohamood、Francis Cheung、Gary Lewis、Glenn Block、Gregory Leake、Ian Ellison-Taylor、Ilia Fortunov、J.R. Arredondo、John deVadoss、Joseph Hofstader、Kashinath TR、Koby Avital、Loke Uei Tan、Luke Nyswonger、Manish Prabhu、Meghan Perez、Mehran Nikoo、Michael Puleio、Mike Francis、Mike Walker、Mubarak Elamin、Nick Malik、Nobuyuki Akama、Ofer Ashkenazi、Pablo Castro、Pat Helland、Phil Haack、Rabi Satter、Reed Robison、Rob Tiffany、Ryno Rijnsburger、Scott Hanselman、Seema Ramchandani、Serena Yeoh、Simon Calvert、Srinath Vasireddy、Tom Hollander、Vijaya Janakiraman、Wojtek Kozaczynski

---

成功例

このガイドがお役に立った際には、ぜひお知らせください。ソリューションを構築する際に直面した問題の簡単な概要と、このガイドがどのように役に立ったかについて記述したメールを MyStory@Microsoft.com (英語のみ) までお送りください。

目次

第 I 部　ソフトウェアのアーキテクチャと設計

このセクションは、アーキテクチャと設計の基礎を理解するのに役立つ一連のトピックで構成されています。まず、ソフトウェア アーキテクチャとその重要性について説明します。要件や制約、ユーザー、ビジネス、アプリケーションを実行するシステムの共通部分など、考慮しなければならない一般的な問題についても取り上げます。その後、基本的な設計原則の定義、現在一般的に使用されているアーキテクチャのパターンとスタイルについて説明し、最後にアーキテクチャを設計する際に推奨する手法を紹介します。詳細については、次の章を参照してください。

- 第 1 章「ソフトウェア アーキテクチャとは」
- 第 2 章「ソフトウェア アーキテクチャの基本原理」
- 第 3 章「アーキテクチャのパターンとスタイル」
- 第 4 章「アーキテクチャと設計の手法」

1

ソフトウェア アーキテクチャとは

ソフトウェア アプリケーション アーキテクチャは、技術上および運用上のすべての要件を満たす構造化されたソリューションを定義して、パフォーマンス、セキュリティ、管理容易性など、一般的な品質特性を最適化するプロセスです。これには、さまざまな要因に基づいた判断を下す必要があります。この判断は、品質、パフォーマンス、管理容易性、およびアプリケーションの全体的な成功に大きな影響を及ぼす可能性があります。
Philippe Kruchten、Grady Booch、Kurt Bittner、Rich Reitman は、Mary Shaw と David Garlan の著書『Software Architecture: Perspectives on an Emerging Discipline』(1996 年) を基に、アーキテクチャの定義を導き出して改良しました。この定義は次のとおりです。

> "ソフトウェア アーキテクチャでは、ソフトウェア システムの構成に関する一連の重要な判断を網羅しています。これには、システムを構成する要素とインターフェイスの選択、要素間のコラボレーションとして指定される動作、このような構成と動作の要素のより大きなサブシステムに対する構成、この構成の指針となるアーキテクチャ スタイルが含まれます。また、機能性、ユーザビリティ、復元性、パフォーマンス、再利用性、理解できること、経済的な制約、テクノロジの制約、トレードオフ、および外観への配慮も必要です。"

Martin Fowler は、著書『エンタープライズ アプリケーション アーキテクチャ パターン』でアーキテクチャについて説明する際には、いくつかの頻出テーマについて概説しています。このテーマについては、次のように記述しています。

> “アーキテクチャは、システムを大きなレベルで分解したもので、決定事項の変更は困難です。システムには複数のアーキテクチャが存在し、アーキテクチャにとって重要な事項はシステムの運用中に変化します。要するに、重要な要素は、すべてアーキテクチャということになります。”
> http://www.pearsonhighered.com/educator/academic/product/0,3110,0321127420,00.html (英語)

Bass、Clements、Kazman 共著の『実践ソフトウェア アーキテクチャ』では、アーキテクチャを次のように定義しています。

> “プログラムやコンピューティング システムのソフトウェア アーキテクチャは、単一または複数のシステム構造で、ソフトウェアの要素、外部からアクセスできるソフトウェア要素のプロパティ、および要素間の関係から成ります。アーキテクチャは、ユーザー側のインターフェイスと関係があるもので、要素の個々の詳細 (内部の実装のみに関連する詳細) はアーキテクチャではありません。”
> http://www.aw-bc.com/catalog/academic/product/0,4096,0321154959,00.html (英語)

アーキテクチャが重要な理由

ソフトウェアは、他の複雑な構造と同様、堅固な基盤をベースに構築されている必要があります。主要なシナリオを検討せず、一般的な問題に対処するための設計を施さず、重要な判断がもらたす長期的な結果を理解しなければ、アプリケーションを危険にさらす可能性があります。最新のツールとプラットフォームは、アプリケーション開発の簡略化に役立ちますが、個々のシナリオや要件に基づいて、慎重にアプリケーションを設計する必要がなくなるわけではありません。アーキテクチャの設計が不適切だとソフトウェアも危険にさらされます。たとえば、ソフトウェアでは、動作が不安定になったり、既存または今後のビジネス要件をサポートできなかったり、運用環境での配置や管理が困難になります。

システムは、ユーザー、システム (IT インフラストラクチャ)、およびビジネスの目標に関する考慮事項に従って設計する必要があります。この各領域について、主要なシナリオを要約して、重要な品質特性 (信頼性、スケーラビリティなど) および満足度と不満足度が高い主要な領域を特定する必要があります。可能であれば、この各領域における成否を判断するメトリックを策定して検討します。

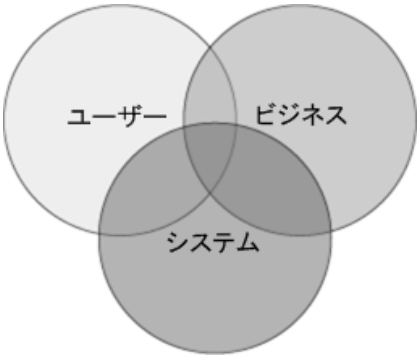


図 1

ユーザー、ビジネス、およびシステムの目標

多くの場合は、トレードオフが発生する可能性が高く、この 3 つの領域間で競合する要件のバランスを取る必要があります。たとえば、ソリューションの全体的なユーザー エクスペリエンスは、ビジネスと IT インフラストラクチャの機能である場合が非常に多く、このいずれかが変更されると、ユーザー エクスペリエンスに大きな影響を及ぼすことがあります。同様に、ユーザー エクスペリエンスの要件が変化すると、ビジネスと IT インフラストラクチャの要件に大きな影響を及ぼすことがあります。パフォーマンスは、ユーザーとビジネスの主要目標となる場合がありますが、システム管理者は、目標を満たすために必要なハードウェアに必ずしも投資できるとは限りません。このような場合には、目標の 80% 程度を実現するようにするとバランスが取れることがあります。

アーキテクチャでは、アプリケーションの主要な要素とコンポーネントが、アプリケーションの他の主要な要素やコンポーネントでどのように使用される (または、どのような相互作用があるのか) かに重点を置きます。設計時には、使用するデータ構造とアルゴリズムや各コンポーネントの実装の詳細が懸念事項になります。アーキテクチャと設計の懸念事項は、重複することがよくあります。アーキテクチャと設計を区別する厳格な規則を使用するよりも、この 2 つの領域を組み合わせて考えた方が理にかなっています。明らかにアーキテクチャを重視した判断を下す場合もあります。また、設計を重視した判断を下す場合もありますが、このような判断によりアーキテクチャを理解することもできます。

このガイドで説明するプロセスを実行して、このガイドに含まれる情報を使用すると、関連するすべての懸念事項に対処し、指定のインフラストラクチャに配置でき、本来の目標と目的を達成するアーキテクチャ ソリューションを構築できます。

ソフトウェア アーキテクチャについて考える際は、次の懸念事項を検討してください。

- ユーザーはどのようにアプリケーションを使用しますか。
- アプリケーションはどのように運用環境に配置されて管理されますか。
- セキュリティ、パフォーマンス、同時実行、国際化、構成など、アプリケーションの品質特性の要件は何ですか。
- どのようにしたら時間が経過しても柔軟性と保守容易性が確保されたアプリケーションを設計できますか。
- 現時点または配置後にアプリケーションに影響を及ぼす可能性があるアーキテクチャの動向は何ですか。

アーキテクチャの目的

アプリケーション アーキテクチャでは、ユース ケースを理解して、これをソフトウェアに実装する方法を特定することで、ビジネス要件と技術上の要件を結び付けることを目指しています。アーキテクチャの目的は、アプリケーションの構造に影響を及ぼす要件を特定することです。優れたアーキテクチャにより、技術的なソリューションの構築に伴うビジネス上のリスクは軽減します。また、優れた設計には十分な柔軟性があるので、時間の経過と共に起こり得る、ハードウェアとソフトウェアのテクノロジ、ユーザー シナリオや要件の動向に対処することができます。アーキテクトは、設計上の判断がもたらす全体的な効果、品質特性 (パフォーマンス、セキュリティなど) 間に内在するトレードオフ、およびユーザー、システム、ビジネスの要件に対処するために必要なトレードオフを考慮する必要があります。

優れたアーキテクチャの条件は次のとおりです。

- システムの構造は公開し、実装の詳細は非公開にします。
- すべてのユース ケースとシナリオを実現します。
- さまざまなステークホルダーの要求に対応します。
- 機能要件と品質要件の両方に対処します。

アーキテクチャの展望

重要なのは、現時点において、アーキテクチャに関する判断を方向付けている原動力を理解することです。これを理解すると、今後どのようにしてアーキテクチャに関する判断を下すかが変わってきます。このような原動力は、ユーザーの要求によって決定されます。また、より早く成果を上げること、さまざまな作業スタイルやワークフローに対するより良いサポート、およびソフトウェア設計の適応性の向上を求める、ビジネスの要求によっても決定されます。考慮する必要がある主な動向は次のとおりです。

- **ユーザーへの権限付与**: ユーザーへの権限付与をサポートする設計では、柔軟性があり、構成可能で、ユーザー エクスペリエンスに重点を置いています。アプリケーションを設計する際には、ユーザーの個人用設定とオプションの適切なレベルを考慮する必要があります。ユーザーが、このレベルを制御するのではなく、アプリケーションの操作方法を定義できるようにします。ただし、混乱を招く可能性がある不必要なオプションや設定を提供して、ユーザーの負担が高くならないようにします。主要なシナリオを理解して、そのレベルを可能な限り単純にします。これにより、ユーザーは簡単に必要な情報を見つけてアプリケーションを使用できます。

- **市場の成熟度**: 既存のプラットフォームとテクノロジを使用して、成熟した市場を活用します。適宜、高度なアプリケーションのフレームワークを基盤として使用すると、既存の再利用可能な機能を再作成することなく、アプリケーションの一意で有用な機能に注力できます。一般的な問題に関して多数の実証済みソリューションの資産を提供するパターンを使用します。
- **柔軟性のある設計**: 柔軟性のある設計では、ソリューションの再利用性と保守容易性を高めるために、疎結合を使用することが増えています。プラグ可能な設計では、配置後の拡張性を実現できます。また、SOA などのサービス指向の技法を利用することで、他のシステムとの相互運用性を提供することもできます。
- **今後の動向**: アーキテクチャを構築する際には、配置後に設計に影響を及ぼす可能性がある今後の動向について理解する必要があります。たとえば、リッチ UI やメディア、マッシュアップなどの複合モデル、増加するネットワーク帯域幅と可用性、増加するモバイル デバイスの使用、今もなお向上するハードウェアのパフォーマンス、コミュニティや個人による情報発信モデルへの関心、クラウド ベースのコンピューティングの高まり、リモート操作などの動向について検討します。

アーキテクチャ設計の原理

現時点でアーキテクチャについて考えるときには、設計は時間と共に進化するが、システムを完全に設計するために必要な情報を事前にすべて把握することはできないということを前提としています。通常、アプリケーションを実装する段階では、新しいことを習得したり、実際の要件に基づいて設計やテストを進めたりするにつれて、設計を発展させる必要があります。この発展が必要なことを念頭に置いてアーキテクチャを設計すると、設計プロセスに着手した時点で、完全には把握していなかった要件に、アーキテクチャを適応させることができます。
アーキテクチャを設計する際には、次の質問事項を検討します。

- 適切に理解しないと、最もリスクが高くなるアーキテクチャの基本的な部分は何ですか。
- 変更される可能性が最も高く、設計を後回しにしても影響が少ないのは、アーキテクチャのどの部分ですか。
- 主要な仮定は何ですか。また、それはどのようにテストしますか。
- どのような場合に設計のリファクタリングが必要となる可能性がありますか。

アーキテクチャを過剰に設計したり、確認できない仮定は立てないようにします。むしろ、後で変更を加えられるように選択肢は狭めないようにします。設計には早い段階で修正しなければならないものがあり、その再設計が必要に

なると、多額のコストがかかる可能性があります。このような領域は迅速に特定し、適切な判断を下すために必要な時間を投資します。

アーキテクチャの基本原理

アーキテクチャを設計する際には、次の基本原理を考慮します。

- **永続ではなく、変化を前提に構築する**: 時間が経過するにつれてアプリケーションにどのような変更が必要となる場合があるのか検討します。このような検討を行うことにより、新しい要件や課題に対処して、これをサポートできる柔軟性を組み込めます。
- **モデル化してリスクを分析および軽減する**: 必要に応じて、設計ツール、統一モデリング言語 (UML) などのモデリング システム、および視覚表現を使用して、要件とアーキテクチャや設計に関する判断を取り込んで、判断による影響を分析します。ただし、モデル化を進めすぎて、設計を簡単に反復して適用できなくならないように注意します。
- **コミュニケーション ツール/共同作業ツールとしてモデルと視覚表現を使用する**: 設計に関する情報、判断、設計に対して現在加えている変更を効率的な方法で伝達することは、優れたアーキテクチャを実現するうえで重要です。アーキテクチャのモデル、ビュー、およびその他の視覚表現を使用して、設計に関する情報をすべてのステークホルダーに効率的に伝達して共有し、設計に対する変更を迅速に伝達します。
- **主要な技術上の判断事項を特定する**: このガイドの情報を使用して、主要な技術上の判断事項と、間違いが起こりやすい領域を理解します。最初にこのような主要な判断を適切に下すために時間を投資すると、より柔軟性が高く、変更によって問題が発生する可能性が低い設計を実現できます。

アーキテクチャを改良するには、段階的で反復的な手法の使用を検討します。まず、ベースライン アーキテクチャの設計に着手して全体像を適切に理解してから、テストを反復的に実行しアーキテクチャを改良することで、アーキテクチャ候補を発展させます。最初からすべてを理解しようとしないことが重要です。まずは、要件と仮定に関する設計のテストを開始するために必要な設計をできる範囲で行います。何度かテストを実施しながら、設計の詳細を追加し、重要な判断を間違いなく下してから、詳細の検討に移ります。よくある落とし穴は、早すぎる段階で詳細の検討に入り、不適切な仮説を立てたり、アーキテクチャを効果的に評価できなかったことが原因で不適切な判断を下すというものです。アーキテクチャをテストする際には、次の質問事項を検討します。

- このアーキテクチャでは、どのような仮説を立てましたか。
- このアーキテクチャで満たしている明示的または黙示的な要件は何ですか。

- このアーキテクチャのアプローチに伴う主なリスクは何ですか。
- 主なリスクを軽減するために講じている対策は何ですか。
- このアーキテクチャは、どのような点でベースライン アーキテクチャや 1 つ前のアーキテクチャ候補よりも優れていますか。

ソフトウェア アーキテクチャ設計の基本原理に関する詳細については、第 2 章「ソフトウェア アーキテクチャの基本原理」を参照してください。
アーキテクチャに対する段階的で反復的なアプローチ、ベースライン アーキテクチャとアーキテクチャ候補、および設計の表現と伝達に関する詳細については、第 4 章「アーキテクチャと設計の手法」を参照してください。

関連情報

Len Bass、Paul Clements、Rick Kazman 著『実践ソフトウェア アーキテクチャ』(日刊工業新聞社、2005 年)
Martin Fowler 著『エンタープライズ アプリケーション アーキテクチャ パターン』(翔泳社、2005 年)

2

ソフトウェア アーキテクチャの基本原理

概要

この章では、ソフトウェア アーキテクチャの主要な設計原理とガイドラインを紹介します。ソフトウェア アーキテクチャは、システムの構成や構造だとされています。この “システム” とは、特定の機能または一連の機能を実行するコンポーネントのコレクションのことを表します。つまり、アーキテクチャでは、特定の機能をサポートするようにコンポーネントを構成することに重点を置いています。多くの場合、この機能の構成は、コンポーネントを "関連領域" にグループ化することで行います。図 1 に、コンポーネントを関連領域でグループ化した、一般的なアプリケーションのアーキテクチャの例を示します。

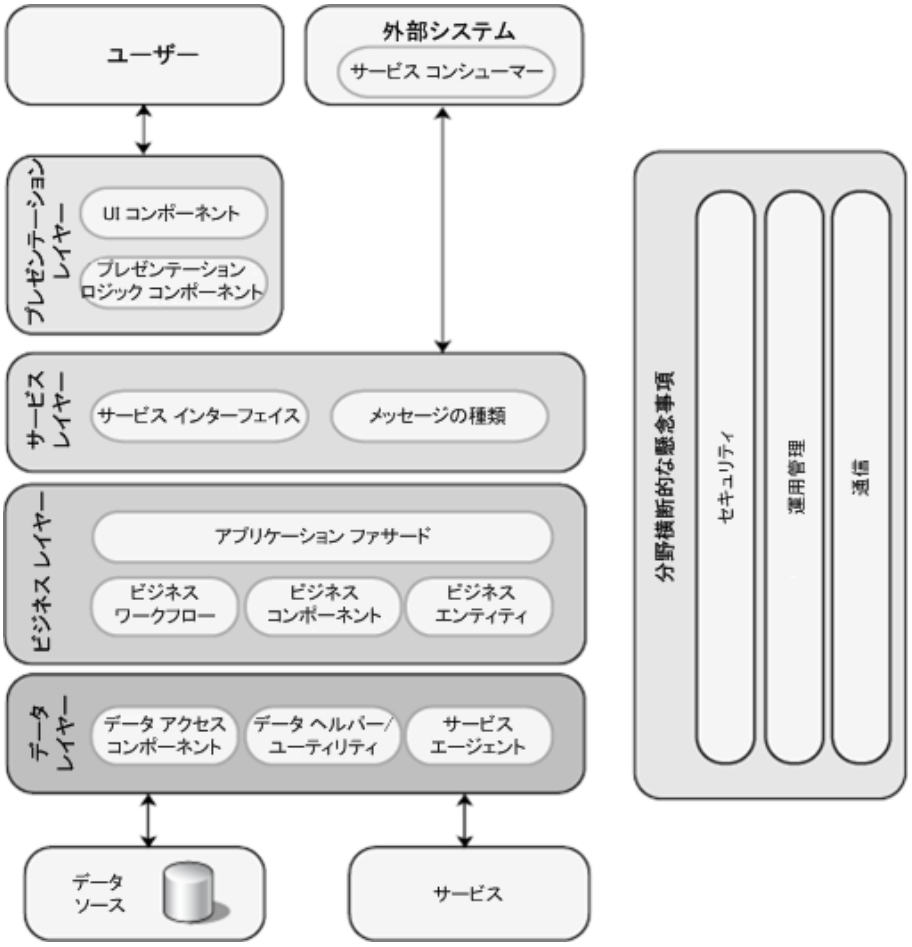


図 2

一般的なアプリケーションのアーキテクチャ

コンポーネントのグループ化に加えて、関連領域では、コンポーネント間の相互作用や異なるコンポーネントがどのように連携するのかについても重点を置いています。この章のガイドラインでは、アプリケーションのアーキテクチャを設計する際に考慮すべき、さまざまな関連領域を取り上げます。

設計の基本原理

設計を開始する際には、アーキテクチャの作成に役立つ基本原理に留意してください。これにより、実績のある原理に準じ、コストと保守の必要性を最小限に抑え、ユーザビリティと拡張性を実現するアーキテクチャを作成できます。基本原理は次のとおりです。

- **関心の分離**: アプリケーションを機能で分割し、機能の重複を可能な限り排除します。相互作用する部分を最小限に抑え、統合性を高めて結合性を抑えるのが重要です。ただし、不適切な境界で機能を分離すると、機能の内容に大きな重複がない場合でも、機能間で高い結合性と複雑さが生じます。
- **単一の役割の原理**: 各コンポーネントとモジュールで管理するのは、特定の機能や機能性、または統合された機能の集合体のみにとどめる必要があります。
- **重複排除の原則 (DRY)**: 機能を提供する場所は 1 か所に絞る必要があります。たとえば、アプリケーションの設計では、特定の機能は単一のコンポーネントに実装する必要があり、他のコンポーネントに実装する機能と重複しないようにします。
- **事前設計の最小化**: 必要な設計のみを行います。開発コストや設計段階のミスによるコストが非常に高ければ、事前に包括的な設計とテストが必要になる場合があります。それ以外の場合は (特にアジャイル開発では)、事前の大規模設計 (BDUF) を回避できます。アプリケーションの要件が明確でなかったり、設計が時間の経過と共に発展する可能性がある場合、早い段階で大規模設計に取り組まないようにします。この原理は、YAGNI (“今必要なことだけを行う”) と呼ばれることもあります。

アプリケーションやシステムを設計する際、ソフトウェア アーキテクトの使命は、設計を関連領域に分離して、アプリケーションやシステムが、なるべく複雑にならないようにすることです。たとえば、ユーザー インターフェイス (UI)、業務プロセス、およびデータ アクセスはすべて、異なる関連領域に存在します。関連領域に分離して設計したコンポーネントでは、その領域に的を絞って、他の関連領域のコードと混在しないようにする必要があります。たとえば、UI 処理コンポーネントには、データ ソースに直接アクセスするコードは含めず、データを取得するためには、ビジネス コンポーネントやデータ アクセス コンポーネントのいずれかを使用する必要があります。
ただし、アプリケーションへの投資については、コストや価値の面から判断する必要もあります。たとえば、UI データを結果セットにバインドできるように、構造を簡略化しなければならない場合があります。通常、機能の境界はビジネスの面からも考慮します。次のガイドラインの概要は、アプリケーションの設計、実装、配置、テスト、および管理の簡略化に影響を及ぼす可能性がある、さまざまな要因を検討する際に役立ちます。

設計に関する慣例

- **各レイヤーで設計パターンの一貫性を維持する**: 可能であれば、論理レイヤーでは、特定の操作に関するコンポーネントの設計で一貫性を維持する必要があります。たとえば、データベースのテーブルやビューへのゲートウェイとして機能するオブジェクトを作成するために、Table Data Gateway パターンの使用する場合は、Repository などの別のパターンを含めないようにします。というのも、Repository パターンでは、データにアクセスしたり、ビジネス エンティティを初期化したりするため

に異なるパラダイムを使用するからです。ただし、ビジネス トランザクションとレポートの作成機能を実行するアプリケーションなど、要件に大きなばらつきがあるレイヤーのタスクには、複数の異なるパターンを使用しなければならない場合があります。

- **アプリケーションの機能は重複しないようにする**: 特定の機能は単一のコンポーネントで提供する必要があります。同じ機能は他のコンポーネントで重複して提供しないようにします。このような構成により、コンポーネントを統合し、特定の機能や機能性が変更されても、簡単にコンポーネントを最適化できます。アプリケーションの機能に重複があると、変更を実装するのが困難になり、明瞭さが失われ、不整合が生じる可能性があります。
- **継承ではなく複合 (コンポジション) を使用する**: 機能を再利用する際には、可能な限り、継承ではなく複合を使用します。これは、継承を使用すると親クラスと子クラスの間の依存関係が強くなり、子クラスの再利用が制限されるためです。また、クラスを再利用すると継承階層も崩れるので、対応が非常に困難になります。
- **開発のコーディング スタイルと名前付け規則を確立する**: 組織でコーディング スタイルと名前付けの標準を確立しているかどうかを確認します。確立していない場合は、一般的な標準を確立する必要があります。標準を確立すると、一貫性のあるモデルが提供され、チームのメンバーは他のメンバーが記述したコードのレビューも容易に行えるようになるので、保守容易性の向上につながります。
- **開発時には自動化された QA（品質分析）の技法を使用して、システムの品質を維持する**: 開発中には、単体テストやその他の自動化された品質分析技法 (依存関係分析、スタティック コード分析など) を使用します。コンポーネントとサブシステムについては、明確な動作とパフォーマンス メトリックを定義し、開発時には自動 QA ツールを使用して、ローカルの設計や実装に関する判断がシステムの品質全体に悪影響を及ぼさないことを確認します。
- **アプリケーションの運用を考慮する**: IT インフラストラクチャで必要なメトリックと運用データを特定し、アプリケーションの配置と運用を確認します。アプリケーションのコンポーネントとサブシステムの個々の運用上の要件を明確に理解したうえで、これらを設計すると、全体的な配置と運用が大幅に簡略化されます。開発時には自動 QA ツールを使用して、アプリケーションのコンポーネントとサブシステムで適切な運用データが提供されていることを確認します。

アプリケーション レイヤー

- **関心事で分離する**: アプリケーションを機能で分割し、機能の重複を可能な限り排除します。この手法を使用する一番のメリットは、機能または機能性を、その他の機能や機能性から独立した状態で最適化できることです。また、ある機能でエラーが発生しても、他の機能でエラーが発生する原因にはならず、

相互に独立した状態で実行できます。さらに、この手法を使用すると、より簡単にアプリケーションを理解して設計することが可能で、相互に依存関係がある複雑なシステムの管理が容易になります。

- **レイヤー間の通信方法を明確にする**: アプリケーションのすべてのレイヤーが、相互に通信したり、他のすべてのレイヤーと依存関係を持てるようにすると、そのソリューションを理解して管理するのが困難になります。レイヤー間の依存関係とデータ フローについては明確な判断を下す必要があります。
- **抽象化を使用してレイヤー間の疎結合を実装する**: これは、インターフェイス コンポーネントを定義することで実現できます。たとえば、レイヤーのコンポーネントで認識できる形式に要求を変換する一般的な入出力の機能を持つファサードを定義できます。また、インターフェイス型や抽象型の基本クラスを使用して、インターフェイス コンポーネントで実装する必要がある共通のインターフェイスや抽象化 (依存関係の反転) を定義することもできます。
- **同じ論理レイヤーに種類の異なるコンポーネントを混在させない**: 関連領域を特定してから、各関連領域と関係のあるコンポーネントを論理レイヤーにグループ化します。たとえば、UI レイヤーには、ビジネス プロセスのコンポーネントを含めない代わりに、ユーザー入力やユーザー要求の処理に使用するコンポーネントを必ず含めるようにします。
- **レイヤーやコンポーネント内でデータ形式の一貫性を維持する**: データ形式が混在していると、アプリケーションの実装、拡張、および管理が困難になります。ただし、データの形式を変換する際には、変換コードを実装して、処理のオーバーヘッドを負う必要があります。

コンポーネント、モジュール、および機能

- **コンポーネントやオブジェクトは、他のコンポーネントやオブジェクト内部の詳細に依存しないようにする**: 各コンポーネントやオブジェクトでは、他のオブジェクトやコンポーネントのメソッドを呼び出す必要があります。また、このメソッドでは、要求の処理方法 (状況によっては、要求を適切なサブコンポーネントや他のコンポーネントにルーティングする方法) に関する情報を保持している必要があります。これにより、保守容易性と柔軟性の高いアプリケーションを開発できます。
- **コンポーネントに過剰な機能を持たせないようにする**: たとえば、UI 処理コンポーネントには、データ アクセス コードを含めたり、追加の機能を提供したりしないようにします。多くの場合、過剰な機能を提供するコンポーネントには、ログ記録、例外処理など、横断的な機能性が混在したビジネス機能を提供する多数の機能やプロパティが含まれています。このような設計では、エラーが発生しやすく、管理が難しくなります。単一の役割と関心の分離の原理を適用することで、このような問題を回避できます。

- **コンポーネント間の通信方法を理解する**: これには、アプリケーションでサポートしなければならない配置シナリオを理解する必要があります。すべてのコンポーネントが同じプロセスで実行されるかどうかと、(メッセージ ベースのインターフェイスを実装することによって) 物理境界やプロセス境界で通信がサポートされる必要があるかどうかを確認する必要があります。
- **アプリケーションのビジネス ロジックから抽象化された横断的なコードを可能な限り分離する**: 横断的なコードとは、セキュリティ、通信、または運用管理 (ログ記録、インストルメンテーションなど) に関連するコードのことです。これらの機能を実装したコードをビジネス ロジックと混在させると、拡張や管理が困難な設計となる可能性があります。分野横断的なコードを変更するには、分野横断的なコードに混在するすべてのビジネス ロジック コードに対応する必要があります。分野横断的な懸念事項の管理を可能にするフレームワークと技法 (アスペクト指向プログラミングなど) の使用を検討してください。
- **コンポーネントの明確なコントラクトを定義する**: コンポーネント、モジュール、および機能では、その使用方法と動作を明確に示すコントラクトやインターフェイスの仕様を定義する必要があります。コントラクトでは、コンポーネント、モジュール、または機能に内在する機能性に他のコンポーネントがどのようにアクセスできるのか、前提条件、後条件、副作用、例外、パフォーマンス特性などの要因から見た、この機能性の動作を規定する必要があります。

設計に関する主要な考慮事項

このガイドでは、決定すべき重要事項について説明します。これを理解することにより、アーキテクチャの設計に着手して、繰り返し開発しながら、すべての重要な要因を検討できます。これ以降のセクションで概要を説明する主要な決定事項は次のとおりです。

- アプリケーションの種類の決定
- 配置に関する方針の決定
- 適切なテクノロジの決定
- 品質特性の決定
- 横断的関心事の決定

設計プロセスの詳細については、第 4 章「アーキテクチャと設計の手法」を参照してください。

アプリケーションの種類の決定

適切なアプリケーションの種類を選択することは、アプリケーションを設計するプロセスにおいて重要です。この選択は、固有の要件とインフラストラクチャの制限事項に従って行います。多くのアプリケーションでは、複数の種類のクライアントをサポートする必要があり、複数の基本的な規範を使用する場合があります。このガイドでは、次の基本的なアプリケーションの種類について説明します。

- モバイル デバイス用に設計されたアプリケーション
- 主にクライアント PC で実行されるように設計されたリッチ クライアント アプリケーション (RIA)
- リッチ UI とメディア シナリオをサポートするインターネットから配置されるように設計されたリッチ インターネット アプリケーション (RIA)
- 疎結合コンポーネント間の通信をサポートするように設計されたサービス アプリケーション
- 常時接続シナリオにおいて、主にサーバーで実行されるように設計された Web アプリケーション

より専門的な種類のアプリケーションについての情報とガイドラインも提供します。これには、次のようなものが含まれます。

- ホストされているクラウド ベースのアプリケーションとサービス
- Microsoft Office とマイクロソフト サーバー テクノロジを統合する Office Business Application (OBA)
- ポータル形式でビジネスに必要な情報と機能を提供する SharePoint 基幹業務 (LOB) アプリケーション

アプリケーションの規範の詳細については、第 20 章「アプリケーションの種類の選択」を参照してください。

配置に関する方針の決定

アプリケーションは、さまざまな環境に配置されます。配置先の環境では、コンポーネントが物理的に異なるサーバーに分散していたり、ネットワーク プロトコル、ファイアウォール、ルーターの構成などに制限事項があったりします。一般的な配置パターンがいくつか存在しますが、各パターンでは、さまざまな分散シナリオと非分散シナリオに関するメリットおよび考慮事項を示しています。アプリケーションの要件と、ハードウェアでサポートできる適切なパターンや選択した配置オプションによって環境から課される制約とのバランスを取る必要があります。これらの要素は、アーキテクチャの設計に影響を及ぼします。

配置に関する問題の詳細については、第 19 章「物理層と配置」を参照してください。

適切なテクノロジの決定

アプリケーションで使用するテクノロジを選択する際に検討すべき重要な要素は、開発するアプリケーションの種類と、アプリケーションの配置トポロジおよびアーキテクチャのスタイルに採用するのが望ましいオプションです。また、テクノロジは、組織のポリシー、インフラストラクチャの制限事項、リソースのスキルなどに従って選択します。選択するテクノロジを決定する前には、このすべての要素を考慮して、選択したテクノロジの機能をアプリケーションの要件と比較する必要があります。

マイクロソフト プラットフォームで使用できるテクノロジの詳細については、付録 A「マイクロソフト アプリケーション プラットフォーム」を参照してください。

品質特性の決定

品質特性 (セキュリティ、パフォーマンス、ユーザビリティなど) を使用して、設計に関して解決すべき重要な問題を検討することができます。要件によって、このガイドで取り上げるすべての品質特性について考慮しなければならない場合もあれば、一部だけを考慮すればよい場合もあります。たとえば、アプリケーションの設計では、セキュリティとパフォーマンスを必ず考慮する必要がありますが、相互運用性やスケーラビリティに関してはすべての設計で考慮しなくてもかまいません。まず、要件と配置シナリオを理解すると、設計において重要な品質特性がわかります。品質特性では競合が発生する場合があることに注意してください。たとえば、セキュリティを重視すると、多くの場合、パフォーマンスやユーザビリティが低下します。

品質特性を考慮した設計を行う場合、次のガイドラインを検討してください。

- 品質特性は、システムの機能から切り離したシステムの特性です。
- 技術的な観点では、品質特性を実装するかどうかが、良いシステムと悪いシステムの分岐点となります。
- 品質特性には、実行時に測定されるものと検査でのみ推定可能なものの 2 種類があります。
- 品質特性間のトレードオフを分析します。

品質特性を検討する際には、次の事項を検討する必要があります。

- アプリケーションに必要な重要な品質特性は何ですか。これを設計プロセスの一環として特定します。
- 特定した品質特性に対応する際の重要な要件は何ですか。また、それは定量化できますか。
- 要件が満たされたことを確認する基準は何ですか。

品質特性の詳細については、第 16 章「品質特性」を参照してください。

横断的関心事の決定

横断的関心事は、設計の主要な領域で、アプリケーションの特定のレイヤーとは関連がありません。たとえば、次の事項に関しては、一元管理されるソリューションや共通のソリューションの実装を検討する必要があります。

- 各レイヤーで共通のストアにログを記録したり、後で結果を関連付けることができるように個別のストアにログを記録できるようにするログ記録のメカニズム
- 複数のレイヤー間で ID を渡して、リソースへのアクセスを許可する認証と承認のメカニズム
- 各レイヤー内で機能したり、システムの境界に例外が伝達されたときにはレイヤーをまたいで機能したりする例外管理のフレームワーク
- レイヤー間の通信で使用できる通信の手法
- プレゼンテーション レイヤー、ビジネス レイヤー、データ アクセス レイヤーでデータをキャッシュできる共通のキャッシュ インフラストラクチャ

次の一覧に、アプリケーションを設計する際に検討する必要がある主要な横断的関心事の一部を示します。

- **インストルメンテーションとログ記録**: すべてのビジネス クリティカルなイベントとシステム クリティカルなイベントをインストルメント化します。また、詳細情報をログに記録して、機密情報を除いた情報を使用して、システムでイベントを再作成します。
- **認証**: ユーザーを認証したり、レイヤー間で認証済み ID を渡す方法を決定します。
- **承認**: 各レイヤーと信頼境界内において、適切な粒度で承認が正常に行われるようにします。
- **例外管理**: 機能上、論理上、物理上の境界で例外をキャッチして、機密情報がエンド ユーザーに開示されないようにします。
- **通信**: 適切なプロトコルを選択し、ネットワーク経由の呼び出しを最小限に抑えて、ネットワーク上で渡される機密データを保護します。
- **キャッシュ**: 何をどこにキャッシュする必要があるのかを特定し、アプリケーションのパフォーマンスと応答性を向上します。キャッシュの設計時には Web ファームとアプリケーション ファームの問題を検討します。

横断的関心事の詳細については、第 17 章「横断的関心事」を参照してください。

3

アーキテクチャのパターンとスタイル

概要

この章では、現在アプリケーションで一般的に使用されている高度なパターンと原理について説明します。これらは "アーキテクチャ スタイル" と呼ばれ、クライアント/サーバー アーキテクチャ、レイヤー型アーキテクチャ、コンポーネント ベースのアーキテクチャ、メッセージ バス アーキテクチャ、サービス指向アーキテクチャ (SOA) などのパターンがあります。この章では、それぞれのスタイルに関して、概要、基本原理、主なメリット、およびアプリケーションに適したアーキテクチャ スタイルを選択するのに役立つ情報を提供します。スタイルでは、アプリケーションのさまざまな側面を定義していることを理解することが重要です。たとえば、展開パターンを定義するアーキテクチャ スタイルもあれば、構造と設計の問題を定義するものもあれば、通信要素を定義するものもあります。そのため、一般的なアプリケーションでは、通常、この章で説明する複数のスタイルを組み合わせて使用しています。

アーキテクチャ スタイルとは

アーキテクチャ スタイル (アーキテクチャ パターンと呼ばれることもあります) は、一連の原理で構成されたもので、システム群に抽象的なフレームワークを提供する大まかなパターンです。アーキテクチャ スタイルでは、頻繁に発生する問題の解決策を提供することで、分割を強化して、設計の再利用を促進します。アーキテクチャのスタイルとパターンは、アプリケーションを形成する一連の原理と見なすことができます。David Garlan と Mary Shaw は、アーキテクチャ スタイルを次のように定義しています。

> "構造上の構成パターンの観点では、アーキテクチャ スタイルはシステム群です。具体的に言うと、アーキテクチャ スタイルでは、そのスタイルのインスタンスで使用できるコンポーネントとコネクターの表現形式と、その組み合わせに関する一連の制約を決定します。この制約には、アーキテクチャを定義する際のトポロジ

に関する制約 (循環がないなど) が含まれます。その他の制約 (たとえば、実行セマンティクスに関連する制約など) が、スタイル定義に含まれている場合もあります。"
(詳細については、1994 年 1 月に公開された David Garlan および Mary Shaw 共著「An Introduction to Software Architecture」(CMU-CS-94-166、http://www.cs.cmu.edu/afs/cs/project/able/ftp/intro_softarch/intro_softarch.pdf、英語) を参照してください)

アーキテクチャ スタイルを理解すると、いくつかのメリットがあります。最大のメリットは、共通の言語を手に入れられることです。また、テクノロジにとらわれないコミュニケーションが可能になります。その結果、詳細に触れることなく、パターンと原理を含む概要レベルでのコミュニケーションが促進されます。たとえば、アーキテクチャ スタイルを使用すると、クライアント/サーバーと n 層の使用について話し合うことができます。アーキテクチャ スタイルは、それぞれの焦点となる主要な領域で分類できます。次の表に、焦点となる主要な領域と対応するアーキテクチャ スタイルを示します。

カテゴリ	アーキテクチャ スタイル
通信	サービス指向アーキテクチャ (SOA)、メッセージ バス
配置	クライアント/サーバー、n 層、3 層
ドメイン	ドメイン駆動設計
構造	コンポーネント ベース、オブジェクト指向、レイヤー型アーキテクチャ

主要なアーキテクチャ スタイルの概要

次の表に、この章で説明する一般的なアーキテクチャ スタイルを示します。この表では、各スタイルの簡単な説明も提示します。この章の後半では、各スタイルの詳細と、アプリケーションに適したアーキテクチャ スタイルを選択するのに役立つガイダンスを紹介します。

アーキテクチャ スタイル	説明
クライアント/サーバー	システムを 2 つのアプリケーションに分割し、クライアントがサーバーに要求を行います。多くの場合、サーバーは、ストアド プロシージャの形でアプリケーション ロジックを実装したデータベースです。
コンポーネント ベースの	アプリケーションの設計を、適切に定義された通信インターフェイスを公開する再利用可能な機能コンポーネントや論理コンポーネントに分解

アーキテクチャ	します。
ドメイン駆動設計	ビジネス ドメインのモデル化と、ビジネス ドメイン内のエンティティに基づいてビジネス オブジェクトを定義することに重点を置く、オブジェクト指向のアーキテクチャ スタイルです。
レイヤー型アーキテクチャ	アプリケーションの関心事を、積み重ねられたグループ (レイヤー) に分割します。
メッセージ バス	ソフトウェア システムを使用するように指定するアーキテクチャ スタイルです。このアーキテクチャ スタイルでは、1 つ以上の通信チャネルを使用してメッセージの送受信を行えるシステムを使用することを指定して、アプリケーションが相互に詳細を把握していなくても対話できるようにします。
n 層と 3 層	レイヤー型のスタイルと同じように、機能を個別のセグメントに分割しますが、各セグメントが物理的に別のコンピューター上に存在する層である点が異なります。
オブジェクト指向	アプリケーションやシステムの役割を再利用可能で自己完結型のオブジェクトに分割する設計パラダイムです (各オブジェクトには、そのオブジェクトに関連するデータと動作が含まれます)。
サービス指向アーキテクチャ (SOA)	コントラクトとメッセージを使用して、機能をサービスとして公開および使用するアプリケーションのことを指します。

アーキテクチャ スタイルを組み合わせる

ソフトウェア システムのアーキテクチャが 1 つのアーキテクチャ スタイルで構成されていることはほとんどなく、複数のアーキテクチャ スタイルを組み合わせて使用しています。たとえば、SOA の設計が、レイヤー型アーキテクチャの手法とオブジェクト指向のアーキテクチャ スタイルを使用して開発されたサービスで構成されている場合があります。

また、アーキテクチャ スタイルを組み合わせて使用すると、一般に公開される Web アプリケーションを構築している場合に、レイヤー型のアーキテクチャ スタイルを使用して関心事を効率的に分離できるので有益です。これにより、ビジネス ロジックとデータ アクセス ロジックから、プレゼンテーション ロジックが分離されます。組織のセキュリティ要件によっては、3 層の展開手法か 3 層以上の展開手法を使用して、アプリケーションを展開しなけ

ればならない場合があります。たとえば、プレゼンテーション層は、組織の内部ネットワークと外部ネットワークの間に位置する境界ネットワークに配置できます。プレゼンテーション層では、対話モデルを実現するために、Model-View-Controller (MVC) などの個別のプレゼンテーション パターン (レイヤー型の開発スタイルの一種) を使用することもできます。また、SOA のアーキテクチャ スタイルを使用して、Web サーバーとアプリケーション サーバー間で、メッセージ ベースの通信を実装することもできます。

デスクトップ アプリケーションを開発している場合は、サーバーで実行中のプログラムに要求を送信するクライアントが存在することがあります。この場合は、クライアント/サーバーのアーキテクチャ スタイルを使用してクライアントとサーバーを配置し、コンポーネント ベースのアーキテクチャ スタイルを使用して、設計を、適切な通信インターフェイスを公開する独立したコンポーネントに分割します。これらのコンポーネントにオブジェクト指向の設計手法を使用すると、再利用性、テスト容易性、および柔軟性が向上します。

選択するアーキテクチャ スタイルは、さまざまな要素の影響を受けます。たとえば、設計と実装を行う際の組織の設備能力、開発者の能力と経験、インフラストラクチャと組織の制約などがあります。これ以降のセクションでは、アプリケーションに適したスタイルを決定するのに役立つ情報を提供します。

クライアント/サーバーのアーキテクチャ スタイル

クライアント/サーバーのアーキテクチャ スタイルでは、独立したクライアントとサーバー システム、および接続しているネットワークを含む分散システムを定義しています。最も単純な形式のクライアント/サーバー システムでは、複数のクライアントが直接アクセスするサーバー アプリケーションを使用します。これは、2 層のアーキテクチャ スタイルと呼ばれます。

これまで、クライアント/サーバーのアーキテクチャは、多くのビジネス ロジックをストアド プロシージャの形式で含むデータベース サーバーや専用のファイル サーバーと通信するデスクトップ向けの GUI アプリケーションで使用されていました。ただし、多くの場合、このアーキテクチャ スタイルでは、クライアントとサーバーの関係を定義しています。この関係では、クライアントが (おそらく GUI を使用して) 1 つ以上の要求を開始し、応答を待機して、受け取った応答を処理します。サーバーでは、通常、ユーザーを承認してから、結果を生成するために必要な処理を実行します。また、情報をクライアントに伝達するために、さまざまなプロトコルとデータ形式を使用して応答を送信します。

現在、クライアント/サーバーのアーキテクチャ スタイルの例には、インターネットやイントラネット上で実行される Web ブラウザー ベースのプログラム、ネットワーク データ サービスにアクセスする Microsoft Windows® オペレーティング システム ベースのアプリケーション、リモート データ ストアにアクセスするアプリケーション

(電子メール プログラム、FTP クライアント、データベース クエリ ツールなど)、リモート システムを操作するツールやユーティリティ (システム管理ツール、ネットワーク監視ツーなど) があります。
これ以外には、次のようなものがあります。

- **Client-Queue-Client (クライアント キュー クライアント) システム**: この手法では、サーバー ベースのキューを通じて、クライアント間の通信を可能にします。クライアントは、単純にデータを格納するためのキューとして動作するサーバーからデータを読み取り、そのサーバーにデータを送信できます。これにより、クライアントでは、ファイルと情報を分散して同期することが可能になります。これは、"passive queue (パッシブ キュー)" のアーキテクチャと呼ばれることもあります。
- **ピア ツー ピア (P2P) アプリケーション**: Client-Queue-Client (クライアント キュー クライアント) システムをベースに開発された P2P のスタイルでは、クライアントとサーバーがそれぞれの役割を入れ替えて、複数のクライアント間でファイルや情報を分散および同期できます。要求に対する複数の応答、共有データ、リソースの探索、およびピアの削除からの復元を通じて、クライアント/サーバーのスタイルを拡張します。
- **アプリケーション サーバー**: シン クライアントがブラウザーや専用のクライアントによってインストールされたソフトウェアを通じてアクセスするアプリケーションとサービスを、サーバーがホストして実行する特殊なアーキテクチャ スタイルです。たとえば、このスタイルは、ターミナル サービスなどのフレームワークを使用して、サーバー上で実行されるアプリケーションを実行するクライアントに使用されます。

クライアント/サーバーのアーキテクチャ スタイルの主なメリットは次のとおりです。

- **セキュリティの強化**: すべてのデータがサーバーに格納されるので、一般的に、クライアント コンピューターにデータが格納される場合と比べてセキュリティの制御が強化されます。
- **一元管理されたデータ アクセス**: すべてのデータがサーバーに格納されるので、その他のアーキテクチャ スタイルに比べて、データへのアクセスとデータの更新が大幅に管理しやすくなります。
- **保守容易性**: ネットワークを通じて相互に認識しているサーバー間で、コンピューティング システムの役割と機能が分散されます。そのため、クライアントが、サーバーの修復、アップグレード、または再配置による影響を受けることはありません。

多数のクライアントをサポートするサーバー ベースのアプリケーションを構築する場合、Web ブラウザーを通じて公開される Web ベースのアプリケーションを開発する場合、組織のすべてのユーザーが使用するビジネス プロセスを実装する場合、または他のアプリケーションで使用されるサービスを開発する場合は、クライアント/サーバーのアーキテクチャ スタイルの使用を検討してください。また、データ ストレージ、バックアップ、および管理機能を一元化したり、アプリケーションでさまざまなクライアントの種類やデバイスをサポートしたりする必要がある場

合には、多くのネットワーク スタイルと同じように、クライアント/サーバーのアーキテクチャ スタイルが適しています。

ただし、従来の 2 層のクライアント/サーバーのアーキテクチャ スタイルには、いくつかの欠点があります。たとえば、アプリケーション データとビジネス ロジックがサーバー上で緊密に結合されることで、システムの拡張性やスケーラビリティに悪影響を及ぼしたり、中央サーバーに依存することで、システムの信頼性に悪影響を及ぼしたりする可能性があります。この問題に対処するために、クライアント/サーバーのアーキテクチャ スタイルは、より一般的な 3 層 (または、n 層) のアーキテクチャ スタイルへと進化を遂げました (3 層のアーキテクチャ スタイルについては、この章の後半で説明します)。このスタイルでは、2 層のクライアント/サーバーのアーキテクチャ固有のいくつかの欠点が克服され、新しいメリットが生まれました。

コンポーネント ベースのアーキテクチャ スタイル

コンポーネント ベースのアーキテクチャでは、システムの設計と開発に使用するソフトウェア エンジニアリングの手法を定義しています。このアーキテクチャでは、設計を、メソッド、イベント、およびプロパティを含む明確に定義された通信インターフェイスを公開する個別の機能コンポーネントや論理コンポーネントに分割することに重点を置いています。オブジェクト指向の設計原理よりも高度な抽象化が提供されるので、通信プロトコルや共有状態などの問題に重点が置かれることはありません。

コンポーネント ベースのスタイルの主要な原理は、次のようなコンポーネントを使用することにあります。

- **再利用可能**: 通常、コンポーネントは、さまざまなアプリケーションの各種シナリオで再利用できるように設計します。ただし、特定のタスクのために設計するものもあります。
- **置き換え可能**: コンポーネントは、他の同様のコンポーネントと簡単に置き換えられます。
- **コンテキストに特化しない**: コンポーネントは、さまざまな環境やコンテキストで動作するように設計します。状態データなど、特殊な情報は、コンポーネントに含めたり、コンポーネントによってアクセスされるのではなく、コンポーネントに渡す必要があります。
- **拡張可能**: 既存のコンポーネントを拡張して新たなコンポーネントを作成して、新しい動作を実現できます。
- **カプセル化**: コンポーネントでは、呼び出し元がコンポーネントの機能を使用できるようにしながら、内部のプロセス、内部変数、または状態に関する詳細を非公開にできるインターフェイスを公開できます。
- **独立**: コンポーネントは、他のコンポーネントへの依存を最小限に抑えるように設計します。そのため、他のコンポーネントやシステムに影響を及ぼすことなく、コンポーネントを適切な環境に配置できます。

アプリケーションで一般的に使用するコンポーネントの種類には、グリッドやボタンなどのユーザー インターフェイス コンポーネント (通称、コントロール) と、他のコンポーネントで使用される特定の機能のサブセットを公開するヘルパー コンポーネントやユーティリティ コンポーネントがあります。これ以外には、リソースを集中的に使用し、あまり頻繁にアクセスされず、Just-In-Time (JIT) の手法 (リモートや分散コンポーネントのシナリオで一般的な手法です) を使用してアクティブ化する必要があるコンポーネントや、メッセージのキュー、保存、および転送を使用してメソッド呼び出しを非同期に実行するキュー コンポーネントなどがあります。

コンポーネントを実行できる環境を提供するプラットフォームのメカニズムに依存するコンポーネントは、"コンポーネント アーキテクチャ" と呼ばれます。この例としては、Windows ではコンポーネント オブジェクト モデル (COM) や分散コンポーネント オブジェクト モデル (DCOM)、その他のプラットフォームでは Common Object Request Broker Architecture (CORBA) や Enterprise JavaBeans (EJB) があります。コンポーネント アーキテクチャでは、コンポーネントとそのインターフェイスの配置、コンポーネント間でのメッセージやコマンドの伝達、および (場合によっては) 状態保持の方法を管理します。

ただし、多くの場合、コンポーネントという用語は、構成要素や要素という、より基本的な意味で使用されます。Microsoft .NET Framework では、このようなコンポーネント ベースの手法を使用したアプリケーションの構築がサポートされています。たとえば、このガイドでは、ビジネス コンポーネントとデータ コンポーネント (一般的には .NET Framework アセンブリにコンパイルされるコード クラス) について説明しています。これらは .NET Framework ランタイムの管理下で実行され、このようなコンポーネントが各アセンブリ内に 1 つ以上存在する場合があります。

コンポーネント ベースのアーキテクチャ スタイルの主なメリットは、次のとおりです。

- **配置の容易性**: 互換性のある新しいバージョンが使用可能になったときに、他のコンポーネントやシステム全体に影響を及ぼすことなく、既存のバージョンと置き換えられます。
- **コストの削減**: サード パーティ製のコンポーネントを使用して、開発と保守にかかるコストを分散できます。
- **開発の容易性**: コンポーネントでは、定義された機能を提供するための既知のインターフェイスを実装するので、システムの他の部分に影響を及ぼすことなく開発を行えます。
- **再利用可能**: 再利用可能なコンポーネントを使用すると、いくつかのアプリケーションやシステム間で開発と保守にかかるコストを分散できます。
- **技術的な複雑さの軽減**: コンポーネントでは、コンポーネント コンテナーとそのサービスを使用して複雑さを軽減します。コンポーネント サービスには、コンポーネントのアクティブ化、有効期間の管理、メソッドのキュー、イベント、トランザクションなどがあります。

Dependency Injection パターンや Service Locator パターンなどの設計パターンは、コンポーネント間の依存関係を管理して、疎結合や再利用を促進するために使用できます。多くの場合、このようなパターンは、複数のアプリケーション間でコンポーネントを組み合わせたり再利用したりする、複合アプリケーションの開発に使用します。既に適切なコンポーネントがある場合やサード パーティのサプライヤーから適切なコンポーネントを入手できる場合、アプリケーションで (データをほとんど、またはまったく入力しないで) プロシージャ形式の機能がメインで実行される場合、または異なるコード言語で記述したコンポーネントを組み合わせる必要がある場合は、コンポーネント ベースのアーキテクチャ スタイルの使用を検討してください。また、個々のコンポーネントを簡単に置換および更新できる、プラグ可能なアーキテクチャや複合アーキテクチャを作成する場合も、このスタイルの使用を検討してください。

ドメイン駆動設計のアーキテクチャ スタイル

ドメイン駆動設計 (DDD) は、ビジネス ドメイン、その要素と動作、およびそれらの関係に基づいてソフトウェアを設計するオブジェクト指向の手法です。このアーキテクチャ スタイルの目的は、ビジネス ドメインの専門家がわかる共通言語でドメイン モデルを定義することによって、ビジネス ドメインで必要菜機能を実現するソフトウェア システムを構築することです。ドメイン モデルは、ソリューションを合理化できるフレームワークと見なせます。ドメイン駆動設計を適用するには、モデル化するビジネス ドメインについてよく理解したり、そのようなビジネス知識を習得することに精通している必要があります。多くの場合、開発チームは、ビジネス ドメインの専門家と連携してビジネス ドメインをモデル化します。アーキテクト、開発者、およびその技術の専門家はさまざまな経歴を持っており、多くの環境では、それぞれの目的、設計、および要件の定義する際に使用する表現が異なります。ただし、ドメイン駆動設計では、チーム全体が、ビジネス ドメインに重点を置く共通言語のみを使用するため、技術的な専門用語は使用しません。

ソフトウェアの中核にはドメイン モデル (この共通言語が直接投影されたもの) があるため、チームはドメイン モデルに関する共通言語を分析することによって、ソフトウェア内のギャップをすばやく特定できます。共通原語を作成することは、単にドメインの専門家から情報を入手して、それを適用するだけではありません。開発チーム内で発生するコミュニケーションの問題の原因の多くは、ドメインの共通言語の誤解だけでなく、ドメインの共通言語自体があいまいであることにもあります。ドメイン駆動設計のプロセスの目的は、使用する共通言語を実装することだけでなく、ドメインの共通言語を改善および改良することにもあります。つまり、モデルはドメインの共通言語を直接投影したものになるので、構築するソフトウェアにメリットがあります。

モデルを単純で役に立つ言語コンストラクトとして維持するためには、ドメイン モデルで、大規模な分離とカプセル化を実装する必要があることが一般的です。このため、ドメイン駆動設計に基づいて構築したシステムのコストは

比較的高くなることがあります。ドメイン駆動設計を使用すると、保守容易性など、多くの技術上のメリットはありますが、複雑なドメイン (モデルと言語のプロセスによって、複雑な情報を伝達したり、ドメインの共通理解を構築するうえで、明らかなメリットがもたらされる場合) にのみ適用することをお勧めします。
ドメイン駆動設計のスタイルの主なメリットは、次のとおりです。

- **コミュニケーション**: 開発チームの全メンバーが、ドメイン モデルとドメイン モデルで定義しているエンティティを使用して、技術的な専門用語を必要とすることなく、ビジネス ドメインの共通言語を使用して、ビジネスに関する知識と要件を伝達できます。
- **拡張可能**: 多くの場合、ドメイン モデルはモジュール形式で柔軟性があるため、条件や要件の変更に合わせて、簡単に更新および拡張できます。
- **テスト可能**: ドメイン モデル オブジェクトは疎結合でまとまりがあるため、簡単にテストできます。

ドメインが複雑で開発チーム内のコミュニケーションと理解を向上する必要がある場合、またはすべての関係者が理解できる共通言語でアプリケーションの設計を説明する必要がある場合は、DDD の使用を検討してください。また、その他の技法を使用して管理するのが難しい、大規模で複雑な企業データのシナリオにも、DDD が適しています。

ドメイン駆動設計の技法の詳細については、「Domain Driven Design （ドメイン駆動設計） Quickly 日本語版」(http://www.infoq.com/jp/minibooks/domain-driven-design-quickly) を参照してください。また、Eric Evans 著『Domain-Driven Design: Tackling Complexity in the Heart of Software』(Addison-Wesley、ISBN: 0-321-12521-5) と Jimmy Nilsson 著『Applying Domain-Driven Design and Patterns: With Examples in C# and NET』(Addison-Wesley、ISBN: 0-321-26820-2) も参照してください。

レイヤー型のアーキテクチャ スタイル

レイヤー型アーキテクチャでは、アプリケーション内の関連する機能を、垂直方向に積み重ねられた別個のレイヤーにグループ化することに重点を置いてます。各レイヤーの機能は、共通の役割や機能で関連付けられます。レイヤー間の通信は疎結合された状態で明示的に行われます。アプリケーションを適宜レイヤーにグループ化することにより、関心事が明確に区別され、その結果、柔軟性と保守容易性を得られます。
レイヤー型のアーキテクチャ スタイルは、inverted pyramid of reuse (再利用の逆ピラミッド) と表現され、各レイヤーでは、その直下にあるレイヤーの機能と抽象化を統合します。厳密なレイヤー型アーキテクチャでは、レイヤー内のコンポーネントは、同じレイヤー内のコンポーネントか、その直下のレイヤー内のコンポーネントとしか対話できません。緩やかなレイヤー型アーキテクチャでは、レイヤー内のコンポーネントは、同じレイヤー内のコンポーネントか、それ以下のレイヤー内のコンポーネントと対話できます。

アプリケーションのレイヤーは、同一の物理コンピューター (同じ層) に配置するか、別個のコンピューター (n 層) に分散させることが可能で、各レイヤーのコンポーネントは、明確に定義されたインターフェイスを通じて他のレイヤーのコンポーネントと通信します。たとえば、一般的な Web アプリケーションの設計は、プレゼンテーション レイヤー (UI に関連する機能)、ビジネス レイヤー (ビジネス ルールの処理)、データ レイヤー (データ アクセスに関連する機能で、その大半が高度なデータ アクセス フレームワークを使用して実装されている場合がほとんどです) で構成されます。n 層アプリケーションのアーキテクチャ スタイルの詳細については、この章の後半の「n 層と 3 層のアーキテクチャ スタイル」を参照してください。

レイヤー型のアーキテクチャ スタイルを使用する設計の一般的な原理は、次のとおりです。

- **抽象化**: レイヤー型アーキテクチャでは、システムの全体像を抽象化し、個々のレイヤーの役割や機能と、レイヤー間の関係を理解するのに必要な情報を提供します。
- **カプセル化**: データ型、メソッド、およびプロパティについて仮定する必要はありません。また、これらの特性はレイヤーの境界で公開されないので、設計時に実装する必要はありません。
- **機能レイヤーの明確な定義**: 各レイヤーの機能は明確に分離されます。上位レイヤー (プレゼンテーション レイヤーなど) では、下位レイヤー (ビジネス レイヤー、データ レイヤーなど) にコマンドを送信し、下位レイヤーで発生したイベントに応答することで、データはレイヤー間で上下に渡されます。
- **高い結合性**: 各レイヤーで機能の境界が明確に定義され、そのレイヤーのタスクと直接関連する機能が各レイヤーに含まれるため、レイヤー内で結合を最大化するのに役立ちます。
- **再利用可能**: 下位レイヤーは上位レイヤーに依存しないので、他のシナリオで再利用できます。
- **疎結合**: レイヤー間の通信は、抽象化とイベントに基づいて行われるので、レイヤー間の疎結合が実現されます。

レイヤー型アーキテクチャを採用したアプリケーションの例には、経理システムや顧客管理システムなどの基幹業務 (LOB) アプリケーション、企業の Web ベースのアプリケーションや Web サイト、ビジネス ロジックの実装に一元管理されたアプリケーション サーバーを使用する、企業のデスクトップ クライアントやスマート クライアントなどがあります。

レイヤー型のアーキテクチャ スタイルは、多数の設計パターンでサポートされています。たとえば、Separated Presentation パターンには、UI によるユーザーとの対話、プレゼンテーション ロジックとビジネス ロジック、およびユーザーが使用するアプリケーション データを処理する、さまざまなパターンが含まれます。また、このパターンでは、開発者が UI を動作させるためのコードを記述して、グラフィック デザイナーが UI 自体を作成することが可能です。このように機能を別個の役割に分割することで、個々の役割の動作をテストする機会が増加します。Separated Presentation パターンの主要な原理は、次のとおりです。

- **関心事の分離**: Separated Presentation パターンでは、UI 処理の関心事を別個の役割に分割しています。たとえば、MVC には、Model、View、および Controller という 3 つの役割があります。Model と View では、それぞれデータ (ビジネス ルールを含むドメイン モデル) と UI を表し、Controller では、要求の処理、モデルの操作、およびその他の操作を実行します。
- **イベント ベースの通知**: Model の変更によってデータを管理している場合は、一般的に Observer パターンを使用して、View に通知します。
- **イベント処理の委任**: Controller では、View の UI コントロールでトリガーされたイベントを処理します。

Separated Presentation パターンには、これ以外に、Passive View パターンや Supervising Presenter (Supervising Controller) パターンがあります。
レイヤー型のアーキテクチャ スタイルと Separated Presentation パターンの主なメリットは、次のとおりです。

- **抽象化**: レイヤーでは、抽象化レベルで変更を行うことが可能です。階層構造になっている各レイヤーで使用する抽象化のレベルは、上下に調整できます。
- **分離**: リスクを軽減し、システム全体への影響を最小限に抑えるために、テクノロジのアップグレードはレイヤーごとに行えます。
- **管理容易性**: 主な関心事を分離することで、依存関係を特定して、コードをより管理しやすい単位にまとめられます。
- **パフォーマンス**: レイヤーを複数の物理層に分散することで、スケーラビリティ、フォールト トレランス、およびパフォーマンスが向上します。
- **再利用性**: 役割によって再利用性を促進します。たとえば、MVC では、Controller とその他の対応している View を共に再利用して、同一のデータと機能に対して役割固有のビューやユーザーがカスタマイズしたビューを提供します。
- **テスト容易性**: 明確に定義されたレイヤーのインターフェイスとレイヤーのインターフェイスの実装を切り替えられることにより、テスト容易性が向上します。Separated Presentation パターンを使用すると、テスト中に Model、Controller、または View のような具体的なオブジェクトの動作を模倣するモック オブジェクトを構築できます。

他のアプリケーションでの再利用に適しているレイヤーがある場合、サービスのインターフェイスを通じて適切なビジネス プロセスを公開するアプリケーションがある場合、またはアプリケーションが複雑であるため、概要レベルの設計を分離して、各チームがさまざまな機能分野に集中できるようにする必要がある場合は、レイヤー型のアーキテクチャ スタイルの使用を検討してください。また、アプリケーションで複数の種類のクライアントやデバイスを

サポートする必要がある場合や、複雑で構成可能なビジネス ルールやビジネス プロセスを実装する必要がある場合にも、レイヤー型のアーキテクチャ スタイルが適しています。

テスト容易性を向上し、UI 機能の保守作業を簡略化する必要がある場合、または UI を機能させるロジック コードの開発と UI をデザインする作業を分ける必要がある場合は、Separated Presentation パターンの使用を検討してください。また、UI のデザインに要求を処理するコードが含まれておらず、ビジネス ロジックが実装されていない場合も、このパターンが適しています。

メッセージ バスのアーキテクチャ スタイル

メッセージ バス アーキテクチャでは、1 つ以上の通信チャネルを使用してメッセージを送受信できるソフトウェア システムを使用する原理を定義して、アプリケーションが相互に詳細を把握することなく対話できるようにしています。これは、共通のバスを通じてメッセージを (通常は非同期に) 渡すことによってアプリケーション間で通信を行うようなアプリケーションを設計するためのスタイルです。最も一般的なメッセージ バスのアーキテクチャの実装では、メッセージング ルーターか Publish/Subscribe パターンが使用され、多くの場合、メッセージ キューなどのメッセージング システムを使用して実装されます。多くの実装は、メッセージの送受信を行うための共通のスキーマと共有インフラストラクチャを使用して通信する個々のアプリケーションで構成されています。メッセージ バスでは、次の項目を処理する機能が提供されます。

- **メッセージ指向の通信**: アプリケーション間のすべての通信は、既知のスキーマを使用するメッセージに基づいて行われます。
- **複雑な処理ロジック**: 複雑な操作は、多段階モデルの一環として、特定のタスクをサポートする小さな操作を組み合わせることによって実行できます。
- **処理ロジックへの変更**: バスとの通信は共通のスキーマとコマンドに基づいているため、バスでアプリケーションを挿入または削除して、メッセージの処理に使用するロジックを変更できます。
- **さまざまな環境との統合**: 一般的な標準に基づいたメッセージ ベースの通信モデルを使用して、Microsoft .NET Framework や Java などの、異なる環境用に開発されたアプリケーションと通信できます。

メッセージ バスの設計は、長い間、複雑な処理規則をサポートするために使用されてきました。この設計では、アプリケーションをプロセスに組み込んだり、同じアプリケーションのいくつかのインスタンスをバスにアタッチすることでスケーラビリティを向上したりできる、プラグ可能なアーキテクチャが提供されます。メッセージ バスのスタイルには、次のような種類があります。

- **Enterprise Service Bus (ESB: エンタープライズ サービス バス)**: ESB では、メッセージ バスの設計に基づいて、バスにアタッチされているコンポーネントとバス間の通信にサービスを使用します。通常、ESB では、メッセージを別の形式に変換するサービスが提供されるので、互換性がないメッセージ形式を使用するクライアント間の通信が可能になります。
- **Internet Service Bus (ISB: インターネット サービス バス)**: エンタープライズ サービス バスと似ていますが、アプリケーションは社内ネットワークではなくクラウドでホストされる点が異なります。ISB の主要な概念は、Uniform Resource Identifier (URI) とポリシーを使用して、クラウドでホストしているアプリケーションとサービスのロジックのルーティングを制御することです。

メッセージ バスのアーキテクチャ スタイルの主なメリットは、次のとおりです。

- **拡張性**: 既存のアプリケーションに影響を及ぼすことなく、アプリケーションをバスに追加したり、アプリケーションをバスから削除したりできます。
- **複雑さの軽減**: 各アプリケーションではバスとの通信方法だけを認識していればよいので、アプリケーションの複雑さが軽減します。
- **柔軟性**: ルーティングを制御する構成やパラメーターを変更するだけで、複雑なプロセス (アプリケーション間の通信パターン) を構成している一連のアプリケーションを、ビジネスやユーザーの要件の変化に合わせて簡単に変更できます。
- **疎結合**: アプリケーションでメッセージ バスとの通信に適したインターフェイスを公開していれば、アプリケーション自体には依存しないため、同じインターフェイスを公開するアプリケーションを変更、更新、および置換できます。
- **スケーラビリティ**: 複数の要求を同時に処理するために、同じアプリケーションの複数のインスタンスをバスにアタッチできます。
- **アプリケーションの単純性**: メッセージバスを実装するとインフラストラクチャが複雑になりますが、各アプリケーションでは、他のアプリケーションへの複数の接続をサポートする必要はなく、メッセージ バスへの接続のみをサポートすれば良いというメリットがあります。

タスクを実行するためにアプリケーション間で相互運用が必要なアプリケーションがあるか、または複数のタスクを 1 つの操作に統合する必要がある場合は、メッセージ バスのアーキテクチャ スタイルの使用を検討してください。また、外部アプリケーションとの連携が必要なタスクや異なる環境でホストされるアプリケーションを実装する必要がある場合にも、このスタイルが適しています。

n 層と 3 層のアーキテクチャ スタイル

n 層と 3 層のアーキテクチャ スタイルでは、レイヤー型のスタイルとほぼ同じ方法で、機能をセグメントに分割することを定義していますが、階層化される各セグメントが物理的に別のコンピューターに配置される点が異なります。これらはコンポーネント指向の手法を通じて進化したもので、通常、通信には、メッセージ ベースの手法ではなくプラットフォーム固有の方法を使用します。

n 層アプリケーションのアーキテクチャは、アプリケーション、サービス コンポーネント、およびそれらの分散配置を機能で分割することを特徴とし、スケーラビリティ、可用性、管理容易性、およびリソースの使用率が向上します。各層は、その直上と直下にある層を除く、すべての層から完全に独立しています。n 番目の層で把握している必要があるのは、n + 1 番目の層からの要求を処理する方法、その要求を n - 1 番目の層に転送する方法 (その直下に層がある場合のみ)、および要求の結果の処理方法だけです。スケーラビリティを向上するために、層間の通信は一般的に非同期で行われます。

通常、n 層アーキテクチャには、少なくとも 3 つの別個の論理部分があり、それぞれ別の物理サーバーに配置されています。それぞれの部分では、特定の機能が実行されます。レイヤー型の設計手法では、レイヤーで公開される機能に 1 つ以上のサービスやアプリケーションが依存している場合、そのレイヤーは層に配置されます。

n 層と 3 層のアーキテクチャ スタイルは、セキュリティが重視される、一般的な財務関係の Web アプリケーションなどで使用されます。ビジネス レイヤーはファイアウォールの内側に配置する必要があります。これにより、プレゼンテーション レイヤーが境界ネットワークの別個の層に配置されます。このスタイルは、プレゼンテーション レイヤーがクライアント コンピューターに配置され、ビジネス レイヤーとデータ レイヤーが 1 つまたは複数のサーバー層に配置される一般的な接続型のリッチ クライアント アプリケーションでも使用されます。

n 層と 3 層のアーキテクチャ スタイルの主なメリットは、次のとおりです。

- **保守容易性**: それぞれの層が他の層から独立しているので、アプリケーション全体に影響を及ぼすことなく、更新や変更を適用できます。
- **スケーラビリティ**: 層はレイヤーの配置に基づいているため、アプリケーションは、かなり簡単にスケールアウトできます。
- **柔軟性**: 各層は個別に管理または拡張できるので、柔軟性が高くなります。
- **可用性**: アプリケーションでは、簡単に拡張できるコンポーネントを使用して、実現するシステムのモジュール式アーキテクチャを活用できるので、可用性が向上します。

アプリケーション内のレイヤーの処理要件が異なり、あるレイヤーの処理で、他のレイヤーの処理が遅延するほどの量のリソースが使用されたり、アプリケーション内のレイヤーのセキュリティ要件が異なる場合は、n 層と 3 層の

アーキテクチャ スタイルの使用を検討してください。たとえば、機密データは、ビジネス レイヤーやデータ レイヤーに格納しても問題ないが、プレゼンテーション レイヤーには格納しないようにする必要がある場合です。また、アプリケーション間でビジネス ロジックを共有できるようにしたり、必要な数のサーバーを各層に割り当てるのに十分なハードウェアがある場合も、n 層と 3 層のアーキテクチャ スタイルが適しています。
すべてのサーバーが社内ネットワーク内に配置されているイントラネット アプリケーションや、セキュリティ要件によって、一般に公開される Web サーバーやアプリケーション サーバーにビジネス ロジックを配置することが制限されていないインターネット アプリケーションを開発する場合は、3 層のアーキテクチャ スタイルの使用を検討してください。セキュリティ要件によって、ビジネス ロジックを境界ネットワークに配置できない場合、またはアプリケーションでリソースが大量に使用され、その機能を別のサーバーにオフロードする必要がある場合は、3 層以上のアーキテクチャ スタイル使用を検討してください。

オブジェクト指向のアーキテクチャ スタイル

オブジェクト指向アーキテクチャは、アプリケーションやシステムを再利用可能で自己完結型の個別のオブジェクトに分割する機能のグループをベースとした設計パラダイムです。各オブジェクトには、そのオブジェクトに関連するデータと動作が含まれます。オブジェクト指向の設計では、システムを、一連のルーチンや手続き型の命令ではなく、一連の連携するオブジェクトと見なします。オブジェクトは独立していて疎結合されており、メソッドの呼び出し、または他のオブジェクトのプロパティへのアクセスと、メッセージの送受信によって、インターフェイスを通じて通信します。オブジェクト指向のアーキテクチャ スタイルの主要な原理は次のとおりです。

- **抽象化**: 抽象化により、処理の基本的な機能を維持したまま、複雑な処理を減らして汎用的な処理にすることが可能です。たとえば、Get や Update のような単純なメソッドを使用するデータ アクセス処理をサポートする既知の定義を抽象インターフェイスにできます。また、構造化データを保持する 2 つの形式のマッピングに使用するメタデータも抽象化の一例です。
- **複合化**: オブジェクトは別のオブジェクトから組み立てることが可能で、内部オブジェクトを他のクラスに公開しないか、または単純なインターフェイスとして公開するかを選択できます。
- **継承**: オブジェクトは他のオブジェクトから継承することが可能で、基本オブジェクトの機能を使用するか、その機能をオーバーライドして新しい動作を実装することができます。また、基本オブジェクトへの変更は、そのオブジェクトを継承するオブジェクトにも自動的に反映されるので、メンテナンスや更新が容易になります。

- **カプセル化**: オブジェクトの機能はメソッド、プロパティ、およびイベントによってのみ公開され、状態や変数などの内部の詳細は、他のオブジェクトには公開されません。インターフェイスの互換性があれば、他のオブジェクトやコードに影響を及ぼすことなく、オブジェクトを簡単に更新および置換できます。
- **ポリモーフィズム (多態性)**: 既存のオブジェクトと置き換え可能な新しい型を実装することによって、アプリケーションの処理をサポートする基本型の動作をオーバーライドできます。
- **分離**: オブジェクトで実装し、コンシューマーが認識できる抽象インターフェイスを定義することによって、オブジェクトをコンシューマーから分離できます。これにより、インターフェイスのコンシューマーに影響を及ぼすことなく、別の実装を提供できます。

オブジェクト指向のスタイルは、一般的に、複雑な科学的な処理や財務処理をサポートするオブジェクト モデルを定義したり、ビジネス ドメインにおける実際の成果物 (顧客、注文など) を表すオブジェクトを定義したりする場合に使用します。通常、後者のプロセスは、オブジェクト指向のスタイルの原理を活用した、より専門的なドメイン駆動設計のスタイルを使用して実装します。詳細については、この章の前半で説明した「ドメイン駆動設計のアーキテクチャ スタイル」を参照してください。

オブジェクト指向のアーキテクチャ スタイルの主なメリットは、次のとおりです。

- **理解しやすい**: アプリケーションを実際のオブジェクトに密接にマップするため、理解しやすくなります。
- **再利用可能**: ポリモーフィズムと抽象化によって、再利用性が向上します。
- **テスト可能**: カプセル化によりテスト容易性が向上します。
- **拡張可能**: カプセル化、ポリモーフィズム、および抽象化により、オブジェクトで公開されるインターフェイスが、データの表現の変更による影響を受けません。これにより、他のオブジェクトとの通信や相互作用の機能が制限されることはありません。
- **高い結合性**: 関連するメソッドおよび機能のみをオブジェクト内に配置して、機能セットに合わせて異なるオブジェクトを使用することによって、高度なレベルの結合を実現できます。

実際のオブジェクトや操作に基づいてアプリケーションをモデル化する必要がある場合や、設計と運用の要件を満たす、適切なオブジェクトとクラスが既に存在している場合は、オブジェクト指向のアーキテクチャ スタイルの使用を検討してください。また、ロジックとデータの両方を再利用可能なコンポーネントにカプセル化する必要がある場合や、抽象化と動的な動作が必要な複雑なビジネス ロジックがある場合も、オブジェクト指向のスタイルが適しています。

サービス指向のアーキテクチャ スタイル

サービス指向アーキテクチャ (SOA) では、アプリケーションの機能を一連のサービスとして提供して、ソフトウェア サービスを使用するアプリケーションを作成できます。呼び出し、公開、および検出が可能な標準ベースのインターフェイスを使用するので、サービスは緩やかに結合されます。SOA のサービスでは、コンポーネントやオブジェクト ベースではなく、アプリケーションを対象とするインターフェイスを通じて、スキーマとメッセージ ベースのアプリケーションとの対話を実現することに重点を置いています。SOA のサービスは、コンポーネント ベースのサービス プロバイダーとして扱わないようにする必要があります。

SOA のスタイルでは、さまざまなプロトコルやデータ形式を使用して情報を伝達することで、ビジネス プロセスを相互運用可能なサービスにパッケージ化できます。クライアントとその他のサービスは、同じ層で実行されているローカル サービスにアクセスしたり、接続しているネットワーク経由でリモート サービスにアクセスすることが可能です。

SOA のアーキテクチャ スタイルの主要な原理は次のとおりです。

- **自律的なサービス**: 各サービスは、個別に管理、開発、配置、およびバージョン管理されます。
- **分散可能なサービス**: 必要な通信プロトコルがネットワークでサポートされていれば、サービスは、ローカルまたはリモートを問わず、ネットワーク上の任意の場所に配置できます。
- **疎結合されているサービス**: 各サービスは他のサービスから独立しているので、インターフェイスの互換性があれば、サービスを使用しているアプリケーションに支障をきたすことなく、サービスを置き換えたり更新したりできます。
- **クラスではなくスキーマとコントラクトを共有するサービス**: サービスが通信する際には、内部クラスではなくコントラクトとスキーマを共有します。
- **ポリシーに基づく互換性**: この場合のポリシーとは、トランスポート、プロトコル、セキュリティなどの機能の定義を指しています。

サービス指向アプリケーションは、情報の共有、多段階のプロセスの処理 (予約システム、オンライン ストアなど)、エクストラネットを経由した業界固有のデータやサービスの公開、複数のソースの情報を組み合わせるマッシュアップの作成などに使用されます。

SOA のアーキテクチャ スタイルの主なメリットは、次のとおりです。

- **ドメインの調整**: 標準インターフェイスを使用する共通のサービスを再利用することによって、ビジネスとテクノロジの機会を増加しながら、コストを削減できます。

- **抽象化**: サービスは自律しており、正式なコントラクトを通じてアクセスされるので、疎結合と抽象化が提供されます。
- **検出可能性**: サービスでは、他のアプリケーションやサービスがそのサービスを検出して、インターフェイスを自動的に特定できるようにするための記述を公開できます。
- **相互運用性**: プロトコルとデータ形式は業界標準に基づいているため、さまざまなプラットフォームでサービスのプロバイダーとコンシューマーを構築および配置できます。
- **合理化**: 多数のアプリケーションで機能を複製するためではなく、特定の機能を提供するために、サービスでは、きめ細かい設定を構成できます。これにより、機能の重複を排除できます。

再利用するのに適したサービスにアクセスできる場合、ホスティング企業が提供している適切なサービスを購入できる場合、1 つの UI がさまざまなサービスで構成されているアプリケーションを構築する必要がある場合、または Software plus Services (S+S: ソフトウェア プラス サービス)、Software as a Service (SaaS: サービスとしてのソフトウェア)、またはクラウド ベースのアプリケーションを作成する場合は、SOA のスタイルの使用を検討してください。また、アプリケーションのセグメント間でメッセージ ベースの通信をサポートして、プラットフォームから独立した形で機能を公開する必要がある場合、認証などの統合されたサービスを活用するか、ディレクトリ経由で検出可能で、インターフェイスに関する予備知識を持たないクライアントが使用できるサービスを公開する必要がある場合も、SOA のスタイルが適しています。

関連情報

Web リソースに簡単にアクセスするには、オンライン版の参考文献 (http://www.microsoft.com/architectureguide、英語) を参照してください。

Eric Evans 著『Domain-Driven Design: Tackling Complexity in the Heart of Software』(Addison-Wesley、2004 年)

Jimmy Nilsson 著『Applying Domain-Driven Design and Patterns: With Examples in C# and NET』(Addison-Wesley、2006 年)

アーキテクチャ スタイルの詳細については、以下のリソースを参照してください。

- ドメイン駆動設計の概要 (http://msdn.microsoft.com/ja-jp/magazine/dd419654.aspx)
- ドメイン駆動設計・開発の実践 (http://www.infoq.com/jp/articles/ddd-in-practice)
- Fear Those Tiers (http://msdn.microsoft.com/en-us/library/cc168629.aspx、英語)
- Layered Versus Client-Server (http://msdn.microsoft.com/en-us/library/bb421529.aspx、英語)

- Message Bus (http://msdn.microsoft.com/en-us/library/ms978583.aspx、英語)
- マイクロソフトのエンタープライズ サービス バス (ESB) に関するガイダンス (http://www.microsoft.com/biztalk/solutions/soa/esb.mspx、英語)
- Separated Presentation (http://martinfowler.com/eaaDev/SeparatedPresentation.html、英語)
- Services Fabric: Fine Fabrics for New-Era Systems (http://msdn.microsoft.com/en-us/library/cc168621.aspx、英語)

第Ⅱ部　設計の基礎

このセクションは、レイヤー型アーキテクチャの基礎を理解するのに役立つトピックで構成されています。プレゼンテーション レイヤー、ビジネス レイヤー、データ レイヤー、サービス レイヤーなど、多くのアプリケーションで一般的に使用されているレイヤーについての実用的なガイドを提供します。このセクションは、次の章で構成されています。

- 第 5 章「レイヤー型アプリケーションのガイドライン」
- 第 6 章「プレゼンテーション レイヤーのガイドライン」
- 第 7 章「ビジネス レイヤーのガイドライン」
- 第 8 章「データ レイヤーのガイドライン」
- 第 9 章「サービス レイヤーのガイドライン」

通常、各レイヤーは多数のコンポーネントで構成されています。各レイヤーのコンポーネントを設計する際には、設計の全体的な完成度に影響するさまざまな要素を考慮する必要があります。このセクションでは、一般的に発生する問題を回避し、ベスト プラクティスに従ってコンポーネントを設計するためのガイドを提供します。詳細については、次の章を参照してください。

- 第 10 章「コンポーネントのガイドライン」
- 第 11 章「プレゼンテーション レイヤーのコンポーネントの設計」
- 第 12 章「ビジネス レイヤーのコンポーネントの設計」
- 第 13 章「ビジネス エンティティの設計」
- 第 14 章「ワークフロー コンポーネントの設計」
- 第 15 章「データ レイヤー のコンポーネントの設計」

アプリケーション設計の全体的な品質と今後の成功は、アプリケーションの設計で、セキュリティ、再利用性、パフォーマンス、保守容易性など、さまざまな品質特性に、どの程度対応しているかによって決まります。また、アプリケーションには、例外処理、キャッシュ、ログ記録など、横断的関心事を解決する機能が備わっていることが多いのが実情です。このセクションでは、アプリケーションに横断的関心事を解決する機能がある場合の品質特性と設計についてのガイドを提供します。詳細については、次の章を参照してください。

- 第 16 章「品質特性」
- 第 17 章「横断的関心事」

適切な通信インフラストラクチャを設計することは、アプリケーション (特に分散アプリケーション) を適切に設計するうえで重要です。このセクションでは、通信要件を理解して、分離、セキュリティ、およびパフォーマンスを適切なレベルで提供するための設計を実装するのに役立つ情報を提供します。詳細については、第 18 章「通信とメッセージ」を参照してください。

また、アプリケーションの配置方法、実行時にアプリケーションをサポートする物理インフラストラクチャ、ネットワーク、その他の機器によって課される制限についても考慮する必要があります。このセクションの最後の章にあたる 19 章では、物理的な配置シナリオについて説明し、複数層の配置モデルを採用したときに生じるセキュリティなどの問題を取り上げます。詳細については、第 19 章「物理層と配置」を参照してください。

4

アーキテクチャと設計の手法

概要

この章では、使用する候補となるアーキテクチャについて考えたりその概要を作成したりするために使用できる反復手法について説明します。この章は、このガイドで説明する主要な決定事項 (品質特性、アーキテクチャ スタイル、アプリケーションの種類、テクノロジ、配置に関する決定など) を 1 つにまとめるのに役立ちます。

この手法は 5 つの主なステップの繰り返しで構成され、各ステップは、この章の残りの部分で説明する個々の考慮事項に基づいて分割されています。この反復手法は、ソリューション候補を作成するのに役立ちます。ステップを繰り返すことによって、こうした候補をさらに改良し、最終的に、アプリケーションに最適なアーキテクチャ設計を作成できます。また、プロセスの最後には、アーキテクチャをレビューして、その結果を関係者全員に伝達できます。

組織で採用しているソフトウェアの開発方法によっては、アーキテクチャをプロジェクトの存続期間中に何度も再検討する場合があります。この手法を使用して、スパイク プログラムの作成、プロトタイプの作成、および実際の開発の期間に学んだことを基に、アーキテクチャをさらに改良できます。

これは使用可能な手法の 1 つに過ぎないと認識しておくことも重要です。アーキテクチャを定義し、レビューし、伝達する、より正式な手法は他にも多数あります。そのいくつかについては、この章の最後で簡単に説明します。

基になる情報、得られるもの、および設計ステップ

設計の基になる情報は、アーキテクチャで対応する必要がある要件と制約を具体化するのに役立ちます。基になる情報として一般的なものは、ユース ケースと使用シナリオ、機能要件、機能以外の要件 (パフォーマンス、セキュリティ、信頼性などの品質特性を含む)、技術要件、配置環境、およびその他の制約です。設計プロセスでは、アーキテクチャの観点から重要なユース ケース、アーキテクチャについて特に注意が必要な問題、および設計プロセスで定義した要件と制約を満たすアーキテクチャ ソリューション候補の一覧を作成します。設計を徐々に改良して、最

終的にすべての要件を満たしすべての制約に従うようにするための一般的な手法は、図 1 に示す 5 つの主要な段階から成る反復手法です。

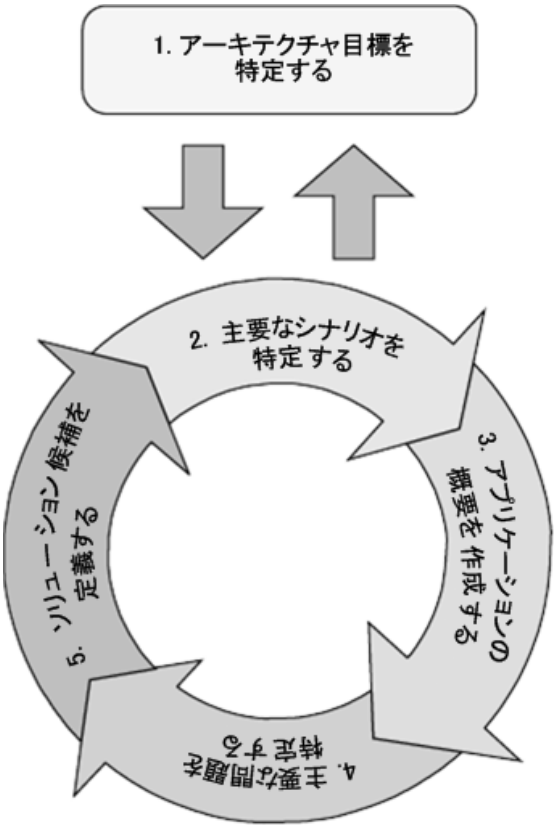


図 3

主要なアーキテクチャ設計作業の反復的なステップ

各ステップは次のとおりです。詳細については、後のセクションで説明します。

1. **アーキテクチャ目標を特定する**: 目標がはっきりしていると、アーキテクチャ、および設計における適切な問題の解決に的を絞ることができます。また、現在のフェーズが完了したことと、次のフェーズに進める状態になったことを適宜に判断できます。
2. **主要なシナリオを特定する**: 主要なシナリオを使用して、最も重要な問題に設計の的を絞り、アーキテクチャ候補の準備ができたら候補を評価します。
3. **アプリケーションの概要を作成する**: アプリケーションを運用する実際の環境と設計を結び付けるために、アプリケーションの種類、配置アーキテクチャ、アーキテクチャ スタイル、およびテクノロジを特定します。
4. **主要な問題を特定する**: 品質特性と分野横断的な懸念事項に基づいて、主要な問題を特定します。主要な問題とは、アプリケーションを設計する際に最も間違いが発生しやすい領域です。

5. **ソリューション候補を定義する**: アーキテクチャ設計の次のサイクルを開始する前に、ソリューションを発展および改善するアーキテクチャのスパイクやプロトタイプを作成し、主要なシナリオ、問題、および配置に関する制約に基づいて評価します。

このアーキテクチャ プロセスは、反復的で段階的な手法となる必要があります。最初のアーキテクチャ候補となるのは大まかな設計で、主要なシナリオ、要件、既知の制約、品質特性、およびアーキテクチャの枠組みに基づいてテストします。アーキテクチャ候補を改良していくにつれて、設計の詳細が見えてくるので、主要なシナリオ、アプリケーションの概要、および問題への取り組み方をさらに発展させられます。

> アーキテクチャの設計で反復手法を使用する場合、アプリケーションを垂直方向ではなく水平方向 (レイヤー) に区切って反復を行ってしまうことがよくあります。垂直方向で反復を行うには、ユーザーから見て 1 つの機能全体 (ユース ケース) を構成する複数のレイヤーをまたぐ機能について考える必要があります。垂直方向に反復を行わないと、ユーザーによる検証の前に、多くの機能を実装するという危険を冒すことになります。

1 回のサイクルでアーキテクチャを構築しようとするのではなく、 サイクルごとに詳細を追加していくようにします。詳細にこだわるのではなく、主要なステップに的を絞り、アーキテクチャと設計の基盤にできるフレームワークを構築します。この後のセクションでは、各ステップについてのガイドラインと情報を提供します。

アーキテクチャ目標を特定する

アーキテクチャ目標は、アーキテクチャと設計プロセスを具体化し、作業範囲を定め、作業が完了したことを判断するのに役立つ目的や制約となります。アーキテクチャ目標を特定する際には、次のキー ポイントを考慮します。

- **アーキテクチャの目的を最初に特定する**: アーキテクチャと設計の各フェーズに費やす期間の長さは、最初に特定した目的の内容によって異なります (たとえば、プロトタイプを構築しようとしているのか、考えられるパスをテストしようとしているのか、それとも、新しいアプリケーションの長期にわたるアーキテクチャ プロセスに着手しようとしているのかを定義します)。
- **アーキテクチャを使用するユーザーを特定する**: 設計は、他のアーキテクトが使用するのか、または開発者やテスター、運用スタッフ、および管理者が使用するのかを特定します。対象ユーザーのニーズや経験を考慮して、ユーザーにとって、より利用しやすい設計を作成します。

- **制約を特定する**: テクノロジの選択肢と制約、使用上の制約、および配置に関する制約を理解します。アプリケーション開発プロセスにおける後の段階で時間を無駄にしたり予想外の事態が発生したりすることがないように、最初の段階で制約を理解します。

目的と期間

アーキテクチャの大まかな目的に基づいて、それぞれの設計作業に費やす期間の長さを定めることができます。たとえば、プロトタイプを設計するには数日間しか必要ない場合があるのに対して、複雑なアプリケーションの包括的で非常に詳細なアーキテクチャを完成するには、数か月間かかる可能性があり、多数のサイクルに及ぶアーキテクチャと設計が必要になる場合があります。目標に対する理解は、各ステップに費やす期間と労力を判断したり、結果がどのようなものになるかを理解したり、アーキテクチャの目的と優先事項を明確に定義したりすることに使用します。考えられる目的には、次のようなものがあります。

- 包括的なアプリケーション設計を作成する
- プロトタイプを構築する
- 主要な技術的リスクを特定する
- 考えられる選択肢をテストする
- システムを理解するために共有のモデルを構築する

この中のどれを目的とするかによって、設計で重点を置く部分も、費やす期間の長さも変わってきます。たとえば、認証アーキテクチャの主要なリスクを特定する必要がある場合は、認証シナリオ、認証アーキテクチャの制約、および考えられる認証テクノロジの選択肢の特定に多くの時間と労力を費やすでしょう。しかし、アプリケーションの全体的なアーキテクチャを検討している早い段階では、認証は、対処したりソリューションに関するドキュメントに記載したりする多数の考慮事項の 1 つに過ぎません。
アーキテクチャ作業には、Web アプリケーションの注文処理 UI に関するフィードバックを得るためのプロトタイプを構築すること、位置データを検索結果にマップするさまざまな方法をテストすること、顧客の注文を追跡するアプリケーションを構築すること、セキュリティ レビューを実施するために、アプリケーションの認証と承認のアーキテクチャを設計することなどがあります。

主要なシナリオを特定する

アーキテクチャと設計において、"ユース ケース" とは、システムと 1 つ以上のアクター (ユーザーまたは別のシステム) との間の一連のやり取りを表現したものを指します。一方、"シナリオ" は、ユース ケースを初めから終わりまでたどるのではなく、ユーザーによるシステムとのやり取りを幅広く包括的に表現したものを指します。システムのアーキテクチャについて考える場合、目標となるのは、アーキテクチャに関する決断を下すのに役立ついくつかの主要なシナリオを特定することです。必要なのは、ユーザー、ビジネス、およびシステムの目標の間でバランスをとることです (第 1 章「ソフトウェア アーキテクチャとは」の図 1 参照)。
主要なシナリオとは、アプリケーションの成功にとって最も重要であると考えられるシナリオのことです。主要なシナリオは、次の条件の 1 つ以上を満たす任意のシナリオとして定義できます。

- 問題 (未知の重要な領域、またはリスクの大きい領域) を表している
- アーキテクチャの観点から重要なユース ケース (次のセクション参照) を示している
- 品質特性と機能の両方にかかわる部分を表している
- 品質特性間のトレードオフを表している

たとえば、ユーザー認証を伴うシナリオは、品質特性 (セキュリティ) と重要な機能 (ユーザーがシステムにログインする方法) の両方にかかわるので、主要なシナリオとなる可能性があります。また、馴染みのないテクノロジや新しいテクノロジを軸とするシナリオも、主要なシナリオとなる場合があります。

アーキテクチャの観点から重要なユース ケース

アーキテクチャの観点から重要なユース ケースは、設計の多くの側面に影響を及ぼします。このようなユース ケースは、アプリケーションの成功を決定付けるうえで特に重要です。このようなユース ケースは、配置するアプリケーションの受け入れにとって重要で、アーキテクチャを評価する際に有益なツールとなるには、設計内容を十分に取り込んでいる必要があります。アーキテクチャの観点から重要なユース ケースには、次のようなものがあります。

- **ビジネスクリティカルなユース ケース**: 他の機能と比べて、使用頻度が高いか、ユーザーや他の関係者にとって特に重要なユース ケースです。また、高いリスクを伴うユース ケースも、これに該当します。
- **影響力の大きいユース ケース**: 機能と品質特性の両方にかかわるか、アプリケーションのあらゆるレイヤーと層に影響を及ぼす分野横断的な懸念事項を表すユース ケースです。たとえば、セキュリティが重視される作成、読み取り、更新、および削除 (CRUD) 操作がこれに該当します。

アーキテクチャの観点からアプリケーションにとって重要なユース ケースを特定したら、アーキテクチャ候補の成否を評価する手段として使用できます。アーキテクチャ候補がより多くのユース ケースに対処したり、より効率的に既存のユース ケースに対処したりする場合は、通常、このアーキテクチャ候補がベースライン アーキテクチャよりも優れているということになります。ユース ケースの定義については、「What is use case?」(http://searchsoftwarequality.techtarget.com/sDefinition/0,,sid92_gci334062,00.html、英語) を参照してください。

優れたユース ケースでは、ユーザー、システム、およびビジネスという 3 つの観点からアーキテクチャをとらえています。このようなシナリオとユース ケースを使用して、設計をテストし、問題が存在する可能性がある箇所を特定します。ユース ケースとシナリオについて考える際には、次の事項を考慮します。

- プロジェクトの早い段階で、アーキテクチャのすべてのレイヤーを使用し、アーキテクチャの観点から重要なエンド ツー エンドのシナリオをサポートするアーキテクチャ候補を作成することにより、リスクを軽減します。
- アーキテクチャ モデルをガイドとして使用して、シナリオ、機能要件、技術要件、品質特性、および制約に合わせてアーキテクチャ、設計、およびコードを変更します。
- その時点で把握している情報に基づいてアーキテクチャ モデルを作成し、その後のプロセスやサイクルで対処する必要がある質問事項の一覧を作成します。
- アーキテクチャと設計に重要な変更を十分に加えたら、このような変更を反映および使用したユース ケースを作成することを検討します。

アプリケーションの概要を作成する

アプリケーションが完成した状態を示す概要を作成します。この概要は、アーキテクチャをより具体的なものにし、現実の制約や決定に結び付けるのに役立ちます。アプリケーションの概要を作成するステップは、次の作業で構成されています。

1. **アプリケーションの種類を決定する**: まず、構築するアプリケーションの種類を決定します (たとえば、モバイル アプリケーション、リッチ クライアント、リッチ インターネット アプリケーション、サービス、Web アプリケーションを構築しているのか、このような異なる種類のアプリケーションのいずれかを組み合わせたものを構築しているのかを決めます)。一般的なアプリケーションの規範の詳細については、第 20 章「アプリケーションの種類の選択」を参照してください。

2. **配置に関する制約を特定する**: アプリケーション アーキテクチャを設計する際には、組織のポリシーや流儀、およびアプリケーションを配置する予定のインフラストラクチャを考慮する必要があります。配置先の環境が固定されていたり柔軟性のないものだったりする場合、アプリケーション設計には、その環境に付随する制約を反映する必要があります。また、アプリケーション設計では、セキュリティや信頼性などのサービス品質 (QoS) 属性も考慮する必要があります。プロトコルの制約とネットワーク トポロジにより、設計のトレードオフが必要になる場合もあります。アプリケーション アーキテクチャとインフラストラクチャ アーキテクチャの間に存在する要件と制約を設計プロセスの早い段階で特定すると、適切な配置トポロジを選択して、アプリケーションと配置先インフラストラクチャとの間で発生する問題を解決できます。配置シナリオの詳細については、第 19 章「物理層と配置」を参照してください。
3. **重要なアーキテクチャ設計スタイルを特定する**: 設計で使用するアーキテクチャ スタイルを決定します。アーキテクチャ スタイルは、一連の原理で構成されており、 システム群に抽象的なフレームワークを提供する大まかなパターンと考えることができます。各スタイルでは、システムを組み立てるのに使用できるコンポーネントの種類、組み立てで使用するリレーションシップの種類、組み立て方に関する制約、および組み立て方の意味に関する仮定を指定する一連のルールを定義しています。アーキテクチャ スタイルは、頻繁に繰り返し発生する問題の解決策を提供することで、分割を強化し、設計の再利用性を促進します。一般的なアーキテクチャ スタイルには、サービス指向アーキテクチャ (SOA)、クライアント/サーバー、レイヤー型、メッセージ バス、およびドメイン駆動設計があります。アプリケーションでは、複数のスタイルを組み合わせて使用する場合がよくあります。現在一般的に使用されているアーキテクチャ スタイルの詳細については、第 3 章「アーキテクチャのパターンとスタイル」を参照してください。
4. **関連するテクノロジを決定する**: 最後に、アプリケーションの種類や他の制約に基づいて関連するテクノロジの選択肢を特定し、設計で使用するテクノロジを決定します。考慮すべき主要な要素は、開発するアプリケーションの種類と、どのようなアプリケーション配置トポロジやアーキテクチャ スタイルが望ましいかの 2 点です。選択するテクノロジは、組織のポリシー、インフラストラクチャの制限事項、リソースのスキルなどにも左右されます。次のセクションでは、それぞれの種類のアプリケーションで一般的に使用されるマイクロソフト テクノロジについて説明します。

関連するテクノロジ

設計で使用するテクノロジを選択する際には、どのテクノロジが、選択したアーキテクチャ スタイル、選択したアプリケーションの種類、およびアプリケーションで重要な品質特性をサポートするのに役立つかを検討します。たと

えば、マイクロソフト プラットフォームに関しては、次の一覧が、どのプレゼンテーション テクノロジ、実装テクノロジ、および通信テクノロジがそれぞれの種類のアプリケーションに最適かを理解するのに役立ちます。

- **モバイル アプリケーション**: .NET Compact Framework、ASP.NET for Mobile、Silverlight for Mobile などのプレゼンテーション レイヤー テクノロジを使用して、モバイル デバイス用のアプリケーションを開発できます。
- **リッチ クライアント アプリケーション**: Windows Presentation Foundation (WPF)、Windows フォーム、および XAML ブラウザー アプリケーション (XBAP) というプレゼンテーション レイヤー テクノロジを併用して、クライアント上に配置および実行されるリッチな UI を備えたアプリケーションを開発できます。
- **リッチ インターネット アプリケーション (RIA)**: Microsoft Silverlight™ ブラウザー プラグインを使用するか、Silverlight と AJAX を併用して、リッチな UI エクスペリエンスを Web ブラウザー内で展開できます。
- **Web アプリケーション**: ASP.NET Web フォーム、AJAX、Silverlight コントロール、ASP.NET MVC、および ASP.NET Dynamic Data を使用して、Web アプリケーションを開発できます。
- **サービス アプリケーション**: Windows Communication Foundation (WCF) と ASP.NET Web サービス (ASMX) を使用して、外部システムやサービス コンシューマーに機能を公開するサービスを開発できます。

各アプリケーションの種類で使用できるテクノロジの詳細については、このガイドの最後にある付録の以下のトピックを参照してください。

- 付録 A「マイクロソフト アプリケーション プラットフォーム」
- 付録 B「プレゼンテーション テクノロジ」
- 付録 C「データ アクセス テクノロジ」
- 付録 D「統合テクノロジ」
- 付録 E「ワークフロー テクノロジ」

アーキテクチャをホワイトボードに描く

アーキテクチャをホワイトボードに描けることは重要です。ホワイトボードに描いた内容を紙ベースで共有するにしろ、スライドで共有するにしろ、それ以外の形で共有するにしろ、重要なのは、主要な制約と決定を示して、議論の骨組みを作って議論を開始できるようにすることです。実際のところ、アーキテクチャをホワイトボードに描くこと

には 2 つの価値があります。アーキテクチャをホワイトボードに描けない場合は、アーキテクチャをよく理解していないということになります。一方、明確で簡潔な図をホワイトボードに描くことができれば、他の関係者がアーキテクチャを理解して、詳細をより簡単に伝達できます。

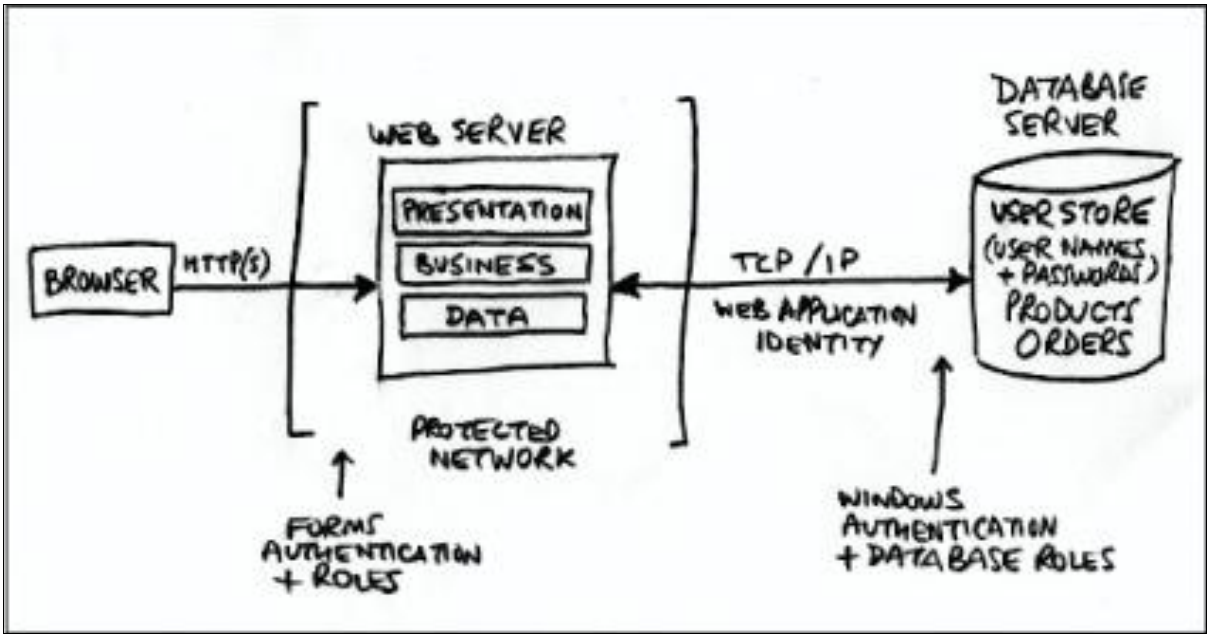


図 4

Web アプリケーションの大まかな設計を示すアーキテクチャが描かれたホワイトボードの例 (使用するプロトコルと認証方法を示しています)

主要な問題を特定する

最も間違いが発生しやすい領域を理解するために、アプリケーション アーキテクチャの問題を特定します。問題となる可能性があるのは、 新しいテクノロジの登場、重要なビジネス要件などです。たとえば、「別のサード パーティが提供するサービスに切り替えられるか」、「新しい種類のクライアントのサポートを追加できるか」、「請求書の作成に関連するビジネス ルールをすばやく変更できるか」、「X 用の新しいテクノロジに移行できるか」などの問題が考えられます。これらの要素は非常に大まかなものですが、一般的に、これらの要素 (およびリスクのある他の領域) は、実装期間における "品質特性" と "分野横断的な懸念事項" に対応します。

品質特性

品質特性とは、実行時の動作、システム設計、およびユーザー エクスペリエンスに影響を与える、アーキテクチャの全般的な特性です。アプリケーションが、求められる品質特性の組み合わせ (ユーザビリティ、パフォーマンス、

信頼性、セキュリティなど) をどの程度備えているかによって、設計の成否とソフトウェア アプリケーションの全体的な品質がわかります。こうした品質特性の要件を満たすようにアプリケーションを設計する際には、他の要件への影響を考慮し、複数の品質特性の間のトレードオフを分析する必要があります。各品質特性の重要性や優先度はシステムによって異なります。たとえば、基幹業務 (LOB) システムでは、相互運用性よりも、パフォーマンス、スケーラビリティ、セキュリティ、およびユーザビリティが重視されます。市販のアプリケーションでは、LOB アプリケーションよりも相互運用性がはるかに重視される可能性が高いでしょう。

品質特性は、レイヤーと層をまたいでアプリケーション全体に影響を与える可能性がある問題領域を表します。システム設計全体にかかわる属性もあれば、 実行時の問題、設計時の問題、またはユーザー中心の問題に固有の属性もあります。次の一覧は、品質特性についての考えを整理して、最も影響を受ける可能性が高いシナリオであるかを理解するのに役立ちます。

- **システム品質**: システム全体の全般的な品質 (サポート性、テスト容易性など)
- **実行時品質**: システムの品質のうち、実行時に直接表れるもの (可用性、相互運用性、管理容易性、パフォーマンス、信頼性、スケーラビリティ、セキュリティなど)
- **設計品質**: システムの設計を反映する品質 (概念的な整合性、柔軟性、保守容易性、再利用性など)
- **ユーザー品質**: システムのユーザビリティ

設計が適切な品質特性を実装するようにする方法の詳細については、第 16 章「品質特性」を参照してください。

分野横断的な懸念事項

分野横断的な懸念事項とは、設計の特性のうち、すべてのレイヤー、コンポーネント、および層にかかわる可能性のあるもののことです。これは、影響力の大きい設計上の間違いが最も発生しやすい領域でもあります。分野横断的な懸念事項の例を以下に示します。

- **認証と承認**: どのようにして、認証と承認の適切な戦略を選択し、レイヤーや層の間で ID をやり取りし、ユーザー ID を格納するか。
- **キャッシュ**: どのようにして、適切なキャッシュ テクノロジを選択して、キャッシュするデータ、データをキャッシュする場所、および適切な有効期限ポリシーを決定するか。
- **通信**: どのようにして、レイヤーや層の間の通信に使用する適切なプロトコルを選択し、レイヤー間の疎結合を設計し、非同期通信を実行し、機密データをやり取りするか。
- **構成管理**: 構成できるようにする必要がある情報、構成情報の格納場所と格納方法、機密の構成情報を保護する方法、およびファームやクラスターで構成情報を処理する方法をどのようにして決定するか。

- **例外管理**: どのようにして、例外を処理してログに記録し、必要に応じて通知を行うか。
- **ログ記録とインストルメンテーション**: ログに記録する情報、ログ記録を構成できるようにする方法、および必要なインストルメンテーションのレベルをどのようにして決定するか。
- **検証**: 検証を実行する場所と方法、長さ、範囲、形式、型に基づく検証に使用する手法、無効な入力値を制約および拒否する方法、悪意や危険性がある可能性を秘めた入力の不適切な部分を削除する方法、および検証ロジックを定義してアプリケーションの複数のレイヤーと層で使い回す方法を、どのようにして決定するか。

設計で分野横断的な懸念事項を適切に処理する方法の詳細については、第 17 章「分野横断的な懸念事項」を参照してください。

問題を軽減するための設計を行う

品質特性と分野横断的な懸念事項を設計要件との関連で分析することにより、特定の問題領域に的を絞ることができます。たとえば、セキュリティという品質特性は言うまでもなく設計においてきわめて重要な要素であり、アーキテクチャの多くのレベルや領域にかかわります。セキュリティに関連する分野横断的な懸念事項は、注意を払う必要がある特定の領域に関するガイダンスを提供します。分野横断的な懸念事項のカテゴリは、さらなる分析のためにアプリケーション アーキテクチャを分割したり、アプリケーションの脆弱性を特定するのに役立ちます。この手法を使用すると、セキュリティの側面を最適化する設計を作成できます。セキュリティに関する分野横断的な懸念事項を考察する際には、次のようなことを検討します。

- **監査とログ記録**: "だれがいつ何をしたか"、 "アプリケーションは正常に動作しているか" です。監査は、アプリケーションでセキュリティ関連のイベントがどのように記録されるかということで、 ログ記録は、アプリケーションの動作に関する情報がどのように公開されるかということです。
- **認証**: "あなたは何者か" ということです。認証とは、あるエンティティが別のエンティティを明確に特定する処理のことで、これは一般に、ユーザー名とパスワードなどの資格情報を使用して行います。
- **承認**: "どのような処理を実行できるか" です。承認は、アプリケーションがリソースへのアクセスや操作をどのように制御するかということです。
- **構成管理**: "アプリケーションはどのようなコンテキストで動作するか"、 "アプリケーションの接続先データベースはどれか"、 "アプリケーションはどのように管理されるか"、 "こうした設定はどのように保護されるか" です。構成管理は、アプリケーションがこうした動作や問題をどのように処理するかということです。

- **暗号化**: "機密をどのように処理するか (機密性)"、 "データやライブラリの改ざんをどのように防止するか (整合性)"、 "暗号化の面で強力なランダム値を作成するにはどうすればよいか" です。暗号化は、アプリケーションがどのように機密性と整合性を実現するかということです。
- **例外管理**: "アプリケーション内のメソッド呼び出しでエラーが発生した場合、アプリケーションではどのような処理を行い、 どの程度の情報を明らかにするか"、 "エンド ユーザーにわかりやすいエラー メッセージを返すか"、 "有益な例外情報を呼び出し元のコードに返すか"、 "エラー発生時にエンド ユーザーに有益で役立つ情報が提供されるか"、"アプリケーションは、管理者がエラーの根本原因分析を実行するのをサポートするか" です。例外管理は、アプリケーション内の例外をどのように処理するかということです。
- **入力とデータ検証**: "アプリケーションへの入力が有効かつ安全であることをどのようにして特定するか"、 "アプリケーションの入り口で入力を制約し、出口で出力をエンコードするか"、 "アプリケーションでは、データベースやファイル共有などのデータ ソースを信頼できるか" です。入力検証は、アプリケーションが処理を続行する前に、入力されたデータをどのようにフィルター処理、無効化、または拒否するかということです。
- **機密データ**: "アプリケーションでは機密データをどのように処理するか"、 "機密のユーザー データとアプリケーション データはアプリケーションで保護されるか" です。機密データは、アプリケーションが、メモリ内、ネットワーク上、または永続的なストア内で保護する必要があるデータをどのように処理するかということです。
- **セッション管理**: "アプリケーションはユーザー セッションをどのように処理および保護するか" です。セッションは、ユーザーとアプリケーションとの間の、関連のある一連のやり取りのことです。

こうした質問とその回答を使用してアプリケーションのセキュリティに関する重要な設計の決断を下すことができます。また、アーキテクチャ作業の一環としてこうした質問とその回答を文書化することができます。たとえば、図 3 は、標準的な Web アプリケーション アーキテクチャで特定されたセキュリティの問題を示しています。

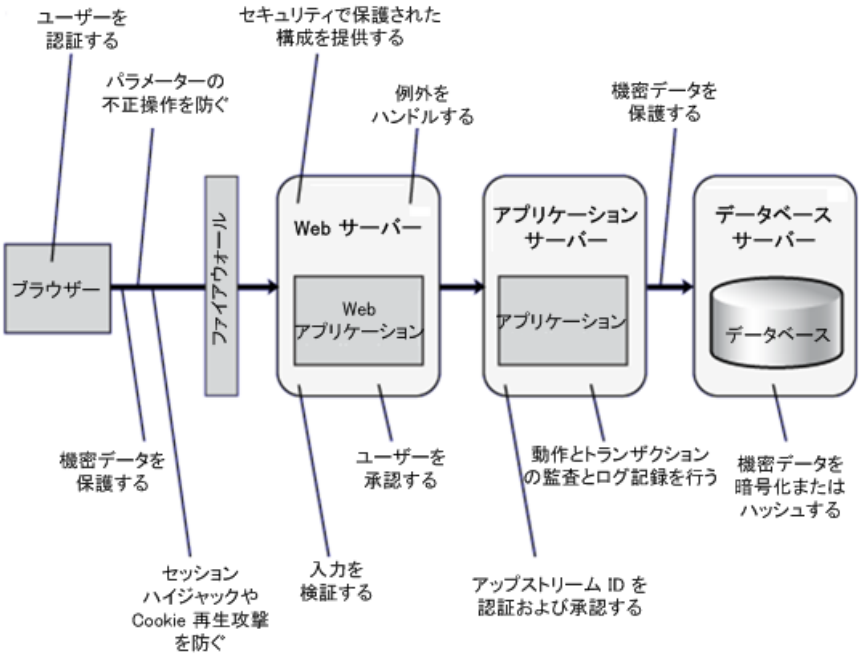


図 5

標準的な Web アプリケーション アーキテクチャで特定されたセキュリティの問題

ソリューション候補を定義する

主要な問題を特定したら、最初のベースライン アーキテクチャを作成し、アーキテクチャ候補を作成するための詳細に取り掛かることができます。その間に、アーキテクチャ スパイクを使用して、設計の特定の領域を調査したり、新しい概念を検証したりすることができます。その後、サイクルを反復的に実行して設計を改善する前に、特定した主要なシナリオや要件に基づいて、新しいアーキテクチャ候補を検証します。

> アジャイル プロセスに従って設計や開発を行っている場合は特にそうですが、サイクルにアーキテクチャと開発作業の両方が含まれていることが重要です。これにより、"事前の大規模設計" 手法を回避できます。

ベースライン アーキテクチャとアーキテクチャ候補

"ベースライン アーキテクチャ" は、既存のシステム (現在のシステムがどのようなものか) を表します。新しいアーキテクチャを作成している場合、最初のベースラインは、アーキテクチャ候補を構築する基盤となる最初の大まかなアーキテクチャ設計です。アーキテクチャ候補には、アプリケーションの種類、配置アーキテクチャ、アーキテクチャ スタイル、テクノロジの選択肢、品質特性、および分野横断的な懸念事項が含まれます。

設計を発展する際には、各段階で主要なリスクを理解して、それを軽減するように設計を変更すること、最も効果的かつ効率的な方法で設計情報を伝達すること、および柔軟性とリファクタリングを念頭に置いてアーキテクチャを構築することに努めてください。数回のサイクルや数個のアーキテクチャ候補を通じて、また、複数のアーキテクチャ スパイクを使用して、アーキテクチャに何度も変更を加えなければならない場合があります。アーキテクチャ候補が前のものより優れている場合は、新しいアーキテクチャ候補を作成およびテストする基盤となるベースラインとして使用できます。

この反復的で段階的な手法を使用することにより、大きなリスクを最初に取り除き、アーキテクチャを反復的に提供し、アーキテクチャ テストを使用して、新しく作成したベースラインが 1 つ前のベースラインよりも優れていることを証明することができます。アーキテクチャ スパイクをベースに作成した新しいアーキテクチャ候補をテストする際には、次の質問事項を検討します。

- このアーキテクチャは、新しいリスクをもたらすことなく成功するか
- このアーキテクチャは、前回のサイクルよりも多くの既知のリスクを軽減するか
- このアーキテクチャは、より多くの要件を満たすか
- このアーキテクチャでは、アーキテクチャの観点から重要なユース ケースが実現されるか
- このアーキテクチャでは、品質特性の懸念事項に対処しているか
- このアーキテクチャでは、より多くの分野横断的な懸念事項に対処しているか

アーキテクチャ スパイク

"アーキテクチャ スパイク" は、アプリケーションの設計やアーキテクチャ全体のほんの一部をテスト実装したものです。目的は、技術的な仮定を検証するためにソリューションの特定部分の技術的な側面を分析することと、設計や実装戦略の候補の中から適切なものを選択することですが、実装のタイムスケールを概算することを目的としている場合もあります。

アーキテクチャ スパイクは、アジャイル開発手法やエクストリーム プログラミング開発手法の一環として使用されることが多いですが、採用する開発手法にかかわらず、ソリューションの設計を改良して発展するための非常に効果的な方法となる可能性があります。アーキテクチャ スパイクを使用してソリューション全体の設計における主要部分に的を絞ることにより、重要な技術的な課題を解決し、ソリューションの設計における全体的なリスクを軽減して、不確定要素を削減できます。

次の作業

アーキテクチャのモデリング作業が完了したら、設計の改良、テストの計画、および他の関係者に設計内容を伝達する作業に着手できます。次のガイドラインを覚えておいてください。

- アーキテクチャ候補やアーキテクチャ テスト ケースを文書化する場合は、簡単に更新できるようにドキュメントを簡単なものにします。このようなドキュメントには、目標の詳細、アプリケーションの種類、配置トポロジ、主要なシナリオと要件、テクノロジ、品質特性、テストなどを含めます。
- 品質特性を使用して、設計や実装を具体的なものにします。たとえば、開発者は、特定したアーキテクチャ リスクに関連する禁止パターンを把握して、問題への対処に役立つ適切な実証済みのパターンを使用する必要があります。
- 入手した情報を、関係のあるチーム メンバーや他の関係者に伝達します。関係者には、アプリケーション開発チーム、テスト チーム、ネットワーク管理者、システム管理者などが含まれます。

アーキテクチャをレビューする

アプリケーションのアーキテクチャをレビューすることは、設計段階のミスによるコストを軽減しアーキテクチャの問題をできるだけ早く見つけて修正するために非常に重要なタスクです。アーキテクチャ レビューは、プロジェクトのコストとプロジェクトが失敗する可能性を軽減する実証済みのコスト効率のよい方法です。アーキテクチャを頻繁に (プロジェクトの主要なマイルストーンごと、また、アーキテクチャに他の重要な変更を加えた場合に) レビューします。レビューでの一般的な質問を念頭に置いてアーキテクチャを構築します。これは、アーキテクチャをより良いものにするためでもあり、レビューを実施するたびに必要となる時間を削減するためでもあります。
アーキテクチャ レビューの主な目的は、ベースライン アーキテクチャとアーキテクチャ候補の実現可能性を特定して、アーキテクチャで機能要件と品質特性が提案された技術的ソリューションと適切に結び付けられていることを確認することです。また、アーキテクチャ レビューは、問題を特定して、改善が必要な領域を認識するのにも役立ちます。

シナリオ ベースの評価

シナリオ ベースの評価は、アーキテクチャ設計をレビューする強力な方法です。シナリオ ベースの評価では、ビジネスの観点から最も重要であり、アーキテクチャに最も大きな影響を与えるシナリオに重点を置きます。次の一般的なレビュー手法のいずれかを使用することを検討してください。

- **Software Architecture Analysis Method (ソフトウェア アーキテクチャ分析法、SAAM)**: SAAM は、当初は更新性を評価するために設計されていましたが、後に、更新性、移植性、拡張性、統合性、対応する機能の範囲などの品質特性に関してアーキテクチャをレビューするために拡張されました。
- **アーキテクチャ トレードオフ分析法 (ATAM)**: ATAM は、SAAM を改良したバージョンで、アーキテクチャに関する決定を品質特性の要件に関してレビューしたり、このような決定が特定の品質目標をどの程度満たしているかに関してレビューしたりするのに役立ちます。
- **Active Design Review (積極的な設計レビュー、ADR)**: ADR は、未完成または発展段階のアーキテクチャに最適なレビュー手法です。主な違いは、全般的なレビューを実施するのではなく、1 回のレビューでは、一連の問題やアーキテクチャの個々のセクションに重点を置いてレビューが行われることです。
- **Active Reviews of Intermediate Designs (中間設計の積極的なレビュー、ARID)**: ARID は、一連の問題に重点を置いて発展段階のアーキテクチャをレビューするという ADR の手法と、品質特性に重点を置いてシナリオ ベースのレビューを行うという ATAM や SAAM の手法を組み合わせたものです。
- **費用便益分析法 (CBAM)**: CBAM では、アーキテクチャに関する決定がコスト、便益、およびスケジュールに与える影響に重点を置きます。
- **Architecture Level Modifiability Analysis (アーキテクチャ レベルの更新性分析、ALMA)**: ALMA では、企業情報システム (BIS) のアーキテクチャの更新性を評価します。
- **Family Architecture Assessment Method (ファミリ アーキテクチャ評価法、FAAM)**: FAAM では、情報システム ファミリ アーキテクチャの相互運用性と拡張性を評価します。

アーキテクチャ設計を分析およびレビューする手法の詳細については、『Evaluating Software Architectures: Methods and Case Studies (SEI Series in Software Engineering)』(Paul Clements、Rick Kazman、Mark Klein 共著、Addison-Wesley Professional、ISBN-10: 020170482X、ISBN-13: 978-0201704822) を参照してください。

アーキテクチャ設計を表現および伝達する

設計内容を伝達することは、設計が適切に実装されるようにすることと同様に、アーキテクチャ レビューでも重要です。そのため、アーキテクチャ設計は、開発チーム、システム管理者やシステム オペレーター、事業主、および他の関係者を含む関係者全員に伝達する必要があります。

アーキテクチャ ビューについて考える方法の 1 つは、重要な決定のマップとして考えるというものです。ここで言うマップは地図のことではなく、アーキテクチャを共有して伝達するのに役立つ抽象的な図のことです。アーキテクチャを関係者に説明するためのよく知られた方法はいくつかあります。その一部を以下に示します。

- **4+1**: この手法では、完全なアーキテクチャに関する 5 つのビューを使用します。そのうち 4 つは、アーキテクチャを異なる観点から表現するものです。その 4 つは、論理ビュー (オブジェクト モデルなど)、プロセス ビュー (同時実行と同期の側面など)、物理ビュー (ソフトウェア レイヤーと機能を分散ハードウェア インフラストラクチャにマップしたもの)、および配置ビューです。5 つ目のビューは、ソフトウェアのシナリオとユース ケースを示します。詳細については、「Architectural Blueprints—The “4+1” View Model of Software Architecture」(http://www.cs.ubc.ca/~gregor/teaching/papers/4+1view-architecture.pdf、英語) を参照してください。
- **アジャイル モデリング**: この手法は、表現より内容を重視する考え方に基づいています。そのため、この方法で作成したモデルは単純で理解しやいながらも、十分な正確さと一貫性を兼ね備えています。ドキュメントが単純なため、関係者が成果物のモデリングに積極的に参加できます。詳細については、『Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process』(Scott Ambler 著、J. Wiley、ISBN-10: 0471202827、ISBN-13: 978-0471202820) を参照してください。
- **IEEE 1471**: IEEE 1471 は、正式には ANSI/IEEE 1471-2000 と呼ばれる標準の略称です。これを使用すると、アーキテクチャの記述内容が強化されます。具体的には、コンテキスト、ビュー、およびビューポイントに特定の意味が与えられます。詳細については、「IEEE Std 1471-2000 IEEE Recommended Practice for Architectural Description of Software-Intensive Systems - Description」(http://standards.ieee.org/reading/ieee/std_public/description/se/1471-2000_desc.html、英語) を参照してください。
- **統一モデリング言語 (UML)**: この手法では、システム モデルに関する 3 つのビューを使用します。その 3 つとは、機能要件ビュー (ユーザーの視点から見た、システムの機能要件 (ユース ケースなど))、静的な構造ビュー (オブジェクト、属性、リレーションシップ、および操作 (クラス ダイアグラムなど))、および動的な動作ビュー (オブジェクト間のコラボレーションやオブジェクトの内部状態の変化 (シーケンス図、アクティビティ図、ステート図など)) です。詳細については、『UML モデリングのエッセンス第 3 版』(Martin Fowler 著、翔泳社、ISBN-10: 4798107956、ISBN-13: 978-4798107950) を参照してください。

関連情報

Scott Ambler 著 『Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process』(J. Wiley、2002 年)

Paul Clements、Rick Kazman、Mark Klein 共著 『Evaluating Software Architectures: Methods and Case Studies (SEI Series in Software Engineering)』(Addison-Wesley Professional、2001 年)

Martin Fowler 著 『UML モデリングのエッセンス第 3 版』(翔泳社、2005 年)

5

レイヤー型アプリケーションのガイドライン

概要

この章では、アプリケーションの全体的な構造を、相互に通信したり、他のクライアントやアプリケーションと通信したりする個別のレイヤーにコンポーネントを論理的にグループ化することによって説明します。レイヤーは、コンポーネントや機能の論理的な区分に関心があり、コンポーネントの物理的な場所については考慮しません。レイヤーは異なる層に配置できますが、同じ層に配置される場合もあります。この章では、アプリケーションを別個の論理的な区分に分割する方法、アプリケーションに適した機能レイアウトを選択する方法、およびアプリケーションで複数の種類のクライアントをサポートする方法について紹介します。また、レイヤーでロジックを公開するために使用できるサービスについても概説します。

> レイヤーと層 (“ティア”) の違いを理解するのは重要です。"レイヤー" はアプリケーションの機能とコンポーネントを論理的にグループ化したものです。一方、“層” は個別のサーバー、コンピューター、ネットワーク、または遠隔地に機能とコンポーネントを物理的に分散したものです。レイヤーと層には同じ一連の名前 (プレゼンテーション、ビジネス、サービス、およびデータ) が使用されますが、物理的な分離を意味するのは層 (“ティア”) だけであることを覚えておいてください。一般的には、1 台の物理コンピューター (同じ層) に複数のレイヤーを配置します。層という用語は、2 層、3 層、n 層など物理的な分散パターンを示すと考えることができます。物理層と配置に関する詳細については、第 19 章「物理層と配置」を参照してください。

論理レイヤー型の設計

設計は、論理的にグループ化したソフトウェア コンポーネントに分割できます。これは設計しているアプリケーションの種類にかかわらず、ユーザー インターフェイスあっても、サービスを公開することのみを目的としたサービス アプリケーション (アプリケーションのサービス レイヤーとは異なります) であっても関係ありません。このように論理的にグループ化したものをレイヤーと呼びます。レイヤーは、コンポーネントで実行されるさまざまな種類のタスクを区別したり、コンポーネントの再利用性をサポートする設計を簡単に作成したりするのに役立ちます。各論理レイヤーは、サブレイヤーでグループ化され、独立した多数の種類のコンポーネントで構成されており、各サブレイヤーでは固有のタスクを実行しています。

多くのソリューションに存在する汎用的な種類のコンポーネントを特定することで、アプリケーションやサービスのマップを作成して、このマップを設計の青写真として使用できます。アプリケーションを別個の役割や機能を持つレイヤーに分割することで、コードの保守容易性を最大限に高め、さまざまな方法で配置した場合にアプリケーションの動作を最適化したり、テクノロジや設計を決定する必要がある場所を明確に線引きします。

プレゼンテーション レイヤー、ビジネス レイヤー、およびデータ レイヤー

最も大まかで抽象的なレベルで見ると、システムの論理アーキテクチャ ビューは、レイヤーでグループ化した一連の連携するコンポーネントだと考えられます。図 1 は、これらのレイヤーや、ユーザー、他のアプリケーション (アプリケーションのビジネス レイヤーに実装されているサービスを呼び出します)、データ ソース (データへのアクセスを提供するリレーショナル データベース、Web サービスなど)、および外部サービスやリモート サービス (アプリケーションで使用されます) との関係を簡略化して大まかに表現したものです。

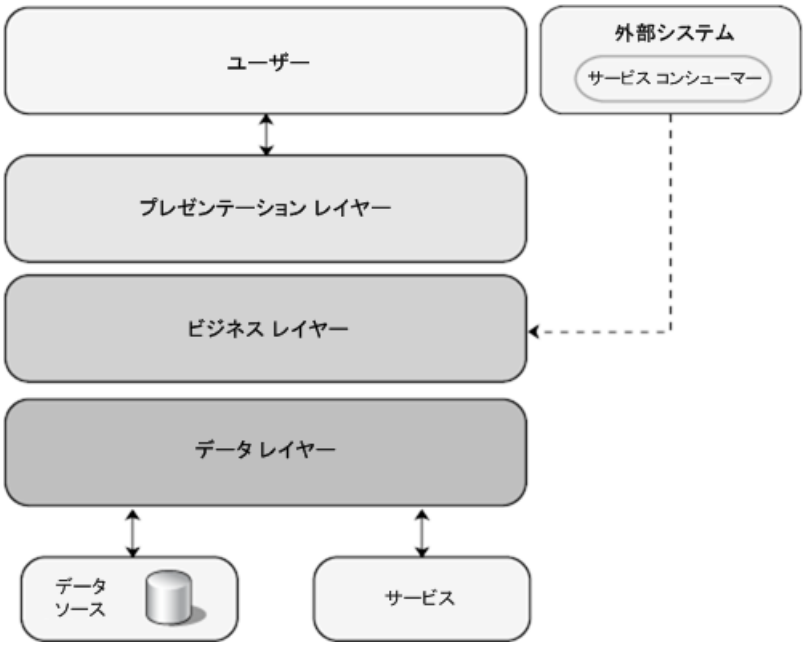


図 6 レイヤー型システムの論理アーキテクチャ ビュー

これらのレイヤーは同じ物理層に配置される場合と別個の物理層に配置される場合があります。レイヤーが別個の層に配置されたり、物理的な境界で分離されている場合、設計では、このビューを考慮する必要があります。詳細については、この章の後半の「レイヤー型構造の設計手順」を参照してください。

図 1 のように、アプリケーションは多数の基本的なレイヤーで構成されています。この一般的な 3 つのレイヤーで構成される設計には、次のレイヤーが含まれます。

- **プレゼンテーション レイヤー**: このレイヤーには、ユーザーとシステムのやりとりを管理するユーザー指向の機能が含まれています。また、通常、ビジネス レイヤーにカプセル化される主要なビジネス ロジックへの共通の橋渡しの役割をするコンポーネントで構成されています。プレゼンテーション レイヤーの設計に関する詳細については、第 6 章「プレゼンテーション レイヤーのガイドライン」を参照してください。プレゼンテーション レイヤーのコンポーネントの設計に関する詳細については、第 11 章「プレゼンテーション レイヤーのコンポーネントの設計」を参照してください。
- **ビジネス レイヤー**: このレイヤーでは、システムの主要機能を実装し、関連するビジネス ロジックをカプセル化します。通常、複数のコンポーネントで構成されており、その一部では他の呼び出し元で使用できるサービス インターフェイスを公開している場合があります。ビジネス レイヤーの設計に関する詳細については、第 7 章「ビジネス レイヤーのガイドライン」を参照してください。ビジネス レイヤーのコンポーネントの設計に関する詳細については、第 12 章「ビジネス レイヤーのコンポーネントの設計」を参照してください。

- **データ レイヤー**: このレイヤーでは、システムの境界内でホストされているデータ、サービスを通じてアクセスできるネットワークに接続している他のシステムで公開されているデータへのアクセスを提供します。データ レイヤーでは、ビジネス レイヤーのコンポーネントで使用できるジェネリック インターフェイスを公開しています。データ レイヤーの設計に関する詳細については、第 8 章「データ レイヤーのガイドライン」を参照してください。データ レイヤー コンポーネントの設計に関する詳細については、第 15 章「データ レイヤー コンポーネントの設計」を参照してください。

サービスとレイヤー

大まかな観点から見ると、サービス ベースのソリューションは複数のサービスで構成され、各サービスはメッセージを渡すことで他のサービスと通信していると見なすことができます。概念的な観点では、サービスはソリューションのコンポーネントと見なすことができます。ただし、各サービスは、内部的には他のアプリケーションと同様にソフトウェア コンポーネントで構成されています。また、このコンポーネントは、プレゼンテーション レイヤー、ビジネス レイヤー、およびデータ レイヤーに論理的にグループ化できます。他のアプリケーションでは、サービスがどのように実装されているかを知らなくても、サービスを使用することができます。前のセクション「ソフトウェアのアーキテクチャと設計」で説明したレイヤー型の設計の原理は、サービス ベースのソリューションにも同様に適応されます。

サービス レイヤー

アプリケーションで、他のアプリケーションにサービスを提供したり、クライアントを直接サポートする機能を実装する必要がある場合、図 2 のようにアプリケーションのビジネス機能を公開するサービス レイヤーを使用するのが一般的な手法です。サービス レイヤーでは、クライアントが異なるチャネルを使用してアプリケーションにアクセスできるよう、効率的に別のビューを提供します。

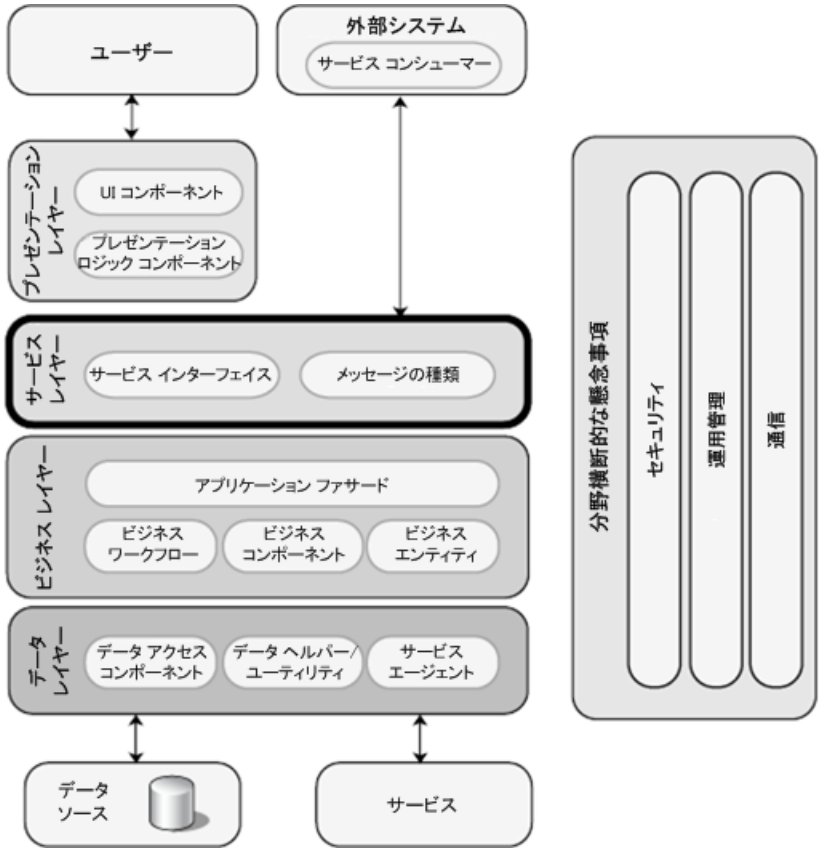


図 2 アプリケーションに含まれるサービス レイヤー

このシナリオでは、ユーザーはプレゼンテーション レイヤー経由でアプリケーションにアクセスできます。プレゼンテーション レイヤーでは、ビジネス レイヤーのコンポーネントと直接通信したり、通信方式で機能を構成する必要がある場合はビジネス レイヤーのアプリケーション ファサード経由で通信します。一方、外部クライアントと他のシステムでは、サービス インターフェイス経由でビジネス レイヤーと通信して、アプリケーションにアクセスして、その機能を使用できます。この構成により、アプリケーションでは、複数の種類のクライアントに対してより優れたサポートを提供することができ、アプリケーション間で機能の再利用と高度な構成が促進されます。

プレゼンテーション レイヤーでは、サービス レイヤーを経由してビジネス レイヤーと通信することがあります。ただし、これは絶対要件ではありません。アプリケーションを物理的に配置したときに、プレゼンテーション レイヤーとビジネス レイヤーが同じ層に配置されていると、2 つのレイヤーは直接通信できる場合があります。サービス レイヤーの設計に関する詳細については、第 9 章「サービス レイヤーのガイドライン」を参照してください。レイヤー間の通信に関する詳細については、第 18 章「通信とメッセージ」を参照してください。

レイヤー型構造の設計手順

アプリケーションの設計に着手する際には、まず、ごく大まかなレベルの抽象化に重点を置いて、機能をレイヤーでグループ化します。次に、設計するアプリケーションの種類に基づいて、各レイヤーにパブリック インターフェイスを定義する必要があります。レイヤーとインターフェイスを定義したら、アプリケーションの配置方法を決定する必要があります。最後に、アプリケーションのレイヤーと層の間の通信で使用する通信プロトコルを選択します。構造とインターフェイスは時間の経過と共に変化する可能性がありますが、特にアジャイル開発を使用している場合は、この手順に従うことで、プロセスの開始段階で重要な側面を考慮できます。一般的な設計手順は次のとおりです。

- 手順 1 - レイヤー型に関する方針を選択する
- 手順 2 - 必要なレイヤーを決定する
- 手順 3 - レイヤーとコンポーネントの分散方法を決定する
- 手順 4 - レイヤーをまとめる必要があるかを判断する
- 手順 5 - レイヤー間の通信規則を決定する
- 手順 6 - 横断的関心事を特定する
- 手順 7 - レイヤー間のインターフェイスを定義する
- 手順 8 - 配置に関する方針を選択する
- 手順 9 - 通信プロトコルを選択する

手順 1 - レイヤー型に関する方針を選択する

レイヤー型とは、アプリケーションのコンポーネントを別個の役割や機能を持つグループに論理的に分離したものです。レイヤー型の手法を使用すると、アプリケーションの保守容易性を向上したり、必要に応じてアプリケーションを簡単に拡張してパフォーマンスを向上することができます。関連のある機能をレイヤーでグループ化する方法は数多くあります。ただし、アプリケーションを極端に少ないレイヤーまたは多いレイヤーに分割すると不要に複雑さが増し、全体的なパフォーマンス、保守容易性、および柔軟性が低下する可能性があります。アプリケーションに適したレイヤーの粒度を決定することは、レイヤー型に関する方針を決定するうえで重要な最初の手順です。

また、レイヤー型を実装する目的が、単に機能を論理的に分離するためなのか、または可能であれば物理的に分離するためなのかを検討する必要があります。レイヤーの境界を越えると、ローカルのパフォーマンスでオーバーヘッドが生じます。物理的に離れているコンポーネントの境界を越える場合は特にそうです。ただし、アプリケーションのスケーラビリティと柔軟性を全体的に向上することで、このパフォーマンスのオーバーヘッドを大幅に軽減できます。

また、レイヤー型を使用することで、隣接したレイヤーに影響を及ぼすことなく、各レイヤーのパフォーマンスを容易に最適化できます。
論理レイヤー型では、通信するアプリケーション レイヤーは、同じ層に配置され同じプロセス内で動作します。これにより、コンポーネントのインターフェイス経由の直接的な呼び出しなど、より高度なパフォーマンスの通信メカニズムを使用することができます。ただし、今後も論理レイヤー型のメリットを維持して柔軟性を確保するためには、カプセル化とレイヤー間の疎結合を保つ必要があることに注意する必要があります。

別個の層 (異なる物理コンピューター) に配置されたレイヤーでは、接続しているネットワーク経由で隣接したレイヤーと通信します。そのため、選択した設計では、通信の待ち時間を考慮し、レイヤー間の疎結合を保つ適切な通信メカニズムをサポートする必要があります。
また、別個の層に配置する可能性が高いアプリケーション レイヤーと、同じ層に配置する可能性が高いアプリケーション レイヤーを決定するのも、レイヤー型に関する方針において重要な部分です。柔軟性を維持するには、レイヤー間の通信は、必ず疎結合にします。これにより、レイヤーが同じ層に配置されている場合はより高度なパフォーマンスを実現して、必要に応じてレイヤーを複数の層に配置することができます。

レイヤー型の手法を使用すると、複雑さが増し、初期開発時間が長くなる可能性がありますが、適切に実装すると、アプリケーションの保守容易性、拡張性、および柔軟性が大幅に向上します。ただし、再利用性と疎結合のトレードオフを考慮する必要があります。これは、パフォーマンスや複雑さに変化があった場合に生じるものです。アプリケーションで採用するレイヤー型とレイヤー間の通信方法を慎重に検討すると、パフォーマンスと柔軟性の間で適切なバランスをとることができます。通常、レイヤーを使用しない緊密に結合している設計から得られる可能性があるわずかなパフォーマンスの向上は、レイヤー型の設計で得られる柔軟性と保守容易性の向上と比較すると微々たるものです。

一般的なレイヤーの種類の詳細と必要なレイヤーの決定に関するガイダンスについては、この章の前半のセクション「論理レイヤー型の設計」を参照してください。

手順 2 - 必要なレイヤーを決定する

関連のある機能をレイヤーでグループ化する方法は数多くあります。ビジネス アプリケーションにおける最も一般的な手法は、プレゼンテーション、サービス、ビジネス、およびデータ アクセスの機能を別個のレイヤーに分離することです。また、アプリケーションによっては、レポート レイヤー、管理レイヤー、またはインフラストラクチャ レイヤーを導入しているものもあります。
レイヤーの追加は慎重に行います。レイヤーでアプリケーションの保守容易性、スケーラビリティ、または柔軟性を明らかに向上できるような関連コンポーネントを論理的にグループ化しない場合、そのレイヤーは追加しないように

します。たとえば、アプリケーションでサービスを公開しない場合は、別個のサービス レイヤーは必要なく、プレゼンテーション レイヤー、ビジネス レイヤー、およびデータ アクセス レイヤーのみを配置できます。

手順 3 - レイヤーおよびコンポーネントの分散方法を決定する

異なる複数の物理層にレイヤーとコンポーネントを配置するのは、必要な場合にとどめる必要があります。分散配置を実装する一般的な理由は、セキュリティ ポリシー、物理的な制約、共有のビジネス ロジック、およびスケーラビリティです。

- Web アプリケーションでは、プレゼンテーション レイヤーの複数のコンポーネントが同時にビジネス レイヤーのコンポーネントにアクセスする場合、ビジネス レイヤーとプレゼンテーション レイヤーのコンポーネントを同じ物理層に配置することを検討します。セキュリティ上の制限で、この 2 つのコンポーネント間に信頼境界が必要でない限り、この配置により、パフォーマンスが最大限に高まり、運用管理が容易になります。
- デスクトップ上で UI プロセスを使用するリッチ クライアント アプリケーションでは、セキュリティ上の理由と運用管理を容易にするため、ビジネス レイヤーのコンポーネントを物理的に個別のビジネス層に配置する場合があります。
- ビジネス エンティティは、それを使用するコードと同じの物理層に配置します。つまり、ビジネス エンティティは複数の場所に配置する場合があります。たとえば、ロジックでビジネス エンティティを使用したり参照する物理的に別のプレゼンテーション層やデータ層にコピーを配置することがあります。サービス エージェント コンポーネントは、セキュリティ制限でコンポーネント間に信頼境界が必要でない限り、コンポーネントを呼び出すコードと同じ層に配置します。
- 類似した読み込みや I/O の特性を持つ非同期ビジネス コンポーネント、ワークフロー コンポーネント、およびサービスは、個別の物理層に配置することを検討します。このように配置すると、インフラストラクチャを微調整してパフォーマンスとスケーラビリティを最大限に高められます。

手順 4 - レイヤーをまとめる必要があるかを判断する

場合によっては、レイヤーをまとめる (緩和する) 方が理にかなっています。たとえば、ごく少数のビジネス ルールが適用されるアプリケーションや主に検証のためにルールを使用するアプリケーションでは、単一のレイヤーにビジネス ロジックとプレゼンテーション ロジックの両方が実装される場合があります。Web サービスからデータを取得して、そのデータを表示するアプリケーションでは、単に Web サービスの情報を直接プレゼンテーション レイ

ヤーに追加して、直接 Web サービスのデータを使用する方が理にかなっています。この場合、データ アクセス レイヤーとプレゼンテーション レイヤーを論理的に組み合わせています。
レイヤーをまとめる方が理にかなっている場合があります。ただし、基本的に機能はレイヤーでグループ化する必要があります。場合によっては、プロキシの機能を提供したり、多くの機能を提供しなくてもカプセル化や疎結合を実現するパススルー レイヤーとして機能するレイヤーもあります。ただし、機能を分離すると、後で設計に含まれる他のレイヤーにほとんど、またはまったく影響を及ぼすことなく機能を拡張できるというメリットがあります。

手順 5 - レイヤー間の通信規則を決定する

レイヤー型に関する方針では、レイヤー間の通信方法についての規則を定義する必要があります。通信規則を定義する主な理由は、依存関係を最小限に抑えて循環参照を削除するためです。たとえば、2 つのレイヤーがもう一方のレイヤーのコンポーネントへの依存関係を持っている場合、循環する依存関係が発生します。このため、次の手法のいずれかを使用して、レイヤー間では一方向の通信のみを許可するという一般的な規則に従うことをお勧めします。

- **トップダウン方式の通信**: 上位レイヤーは下位レイヤーと通信できますが、下位レイヤーは上位レイヤーと通信しないようにします。この規則により、レイヤー間の循環する依存関係を避けられます。イベントを使用すると依存関係なしに、下位レイヤーで発生した変化が上位レイヤーのコンポーネントで認識されるようになります。
- **厳密な通信**: 各レイヤーが通信できるのは、直下のレイヤーのみです。この規則によって、各レイヤーでは直下のレイヤーのみを把握するという懸念事項の分離が強化されます。この規則には、レイヤーのインターフェイスを変更した場合、直上のレイヤーしか影響を受けないというメリットがあります。時間の経過と共に新しい機能を導入して変化するが、その変化による影響を最小限に抑える必要があるアプリケーションを設計したり、複数の物理層に分散される可能性があるアプリケーションを設計する場合は、この手法の使用を検討します。
- **厳密でない通信**: 上位レイヤーでは、間にあるレイヤーをバイパスして、下位レイヤーと直接通信できます。これによりパフォーマンスが向上しますが、依存関係が拡大します。つまり、下位レイヤーを変更すると複数の上位レイヤーに影響が及びます。複数の物理層に分散しないアプリケーション (スタンドアロンのリッチ クライアント アプリケーションなど) を設計する場合や、複数のレイヤーに影響を及ぼす変更を簡単に管理できる小さなアプリケーションを設計する場合は、この手法の使用を検討します。

手順 6 - 横断的関心事を特定する

レイヤーを定義したら、レイヤーをまたぐ機能を特定する必要があります。このような機能は “横断的関心事” と表されることが多く、ログ記録、キャッシュ、検証、認証、および例外管理が含まれます。アプリケーションにおいて横断的関心事を特定するのは重要なことで、可能な場合は、この懸念事項を管理するコンポーネントは別個に設計します。この手法を使用すると、再利用性と保守容易性が向上します。

横断的関心事のコードは、各レイヤーのコンポーネントのコードと混在させないようにします。コードを分離することで、レイヤーとそのコンポーネントでは、ログ記録、キャッシュ、認証などの操作を実行する必要がある場合に、横断的関心事のコンポーネントを呼び出すだけでよくなります。レイヤーをまたいで機能を使用できる必要があるのと同じように、レイヤーが別個の物理層に配置されている場合でも、横断的関心事のコンポーネントは、すべてのレイヤーからアクセスできるように配置する必要があります。

横断的関心事の機能はさまざまな手法を使用して処理できます。たとえば、Patterns & Practices Enterprise Library などの一般的なライブラリや、メタデータを使用して横断的関心事のコードをコンパイル済みの出力に直接挿入するアスペクト指向プログラミング (AOP) のメソッドがあります。横断的関心事の詳細については、第 17 章「横断的関心事」を参照してください。

手順 7 - レイヤー間のインターフェイスを定義する

レイヤーのインターフェイスを定義する際の主な目的は、レイヤー間の疎結合を強化することです。つまり、レイヤーでは、他のレイヤーが依存する可能性がある内部の詳細を公開しないようにします。そのため、レイヤーのインターフェイスは、レイヤーに含まれるコンポーネントの詳細を非公開にするパブリック インターフェイスを提供することで、依存関係を最小限に抑えるように設計する必要があります。このように詳細を公開しないことは “抽象化” と呼ばれており、さまざまな方法で実装できます。レイヤーのインターフェイスは、次の設計手法を使用して定義できます。

- **抽象インターフェイス**: これは抽象型基本クラスを定義するか、具象クラスの型定義として機能するコード インターフェイス クラスを定義することで実現できます。この型では、レイヤーのすべてのコンシューマーがレイヤーとの通信に使用する一般的なインターフェイスを定義します。また、抽象インターフェイスを実装したテスト オブジェクト (モック オブジェクトと呼ばれる場合もあります) を使用できるので、この手法を使用するとテスト容易性が向上します。
- **一般的な設計の型**: 多くの設計パターンでは、インターフェイスをさまざまなレイヤーに定義する具体的なオブジェクト型を定義します。このようなオブジェクト型では、レイヤーに関する詳細を公開しない抽象化の機能を提供します。たとえば、Table Data Gateway パターンでは、データベースのテーブ

ルを表し、データと通信するために必要な SQL クエリを実装するオブジェクト型を定義しています。オブジェクトのコンシューマーは、SQL クエリに関する知識はありません。また、オブジェクトがデータベースに接続してコマンドを実行する方法の詳細も知りません。多くの設計パターンは抽象インターフェイスに基づいていますが、具象クラスに基づいているものもあります。Table Data Gateway パターンなど、適切なパターンの大半は、この点について十分な裏付けが行われています。インターフェイスをレイヤーに迅速かつ簡単に実装する場合またはレイヤーのインターフェイスに設計パターンを実装する場合は、一般的な設計の型の使用を検討します。

- **依存関係の逆転**: これは抽象インターフェイスが、レイヤー以外の場所で定義されているか、レイヤーに依存していないプログラミング スタイルです。レイヤーは相互に依存するのではなく、一般的なインターフェイスに依存します。Dependency Injection パターンは、依存関係の逆転の一般的な実装です。コンテナーでは、依存関係の挿入を使用して、他のコンポーネントが依存している可能性があるコンポーネントの配置方法を指定するマッピングを定義します。また、そのコンテナーでは、このような依存関係のあるコンポーネントを自動的に作成して挿入できます。依存関係の逆転の手法では、コードではなく構成によって依存関係を構築するので、柔軟性があり、プラグ可能な設計の実装が可能です。また、具体的なテスト クラスを設計のさまざまなレイヤーに簡単に挿入できるので、テスト容易性が最大限まで高まります。
- **メッセージ ベース**: メソッドを呼び出したり、オブジェクトのプロパティにアクセスしたりすることで、他のレイヤーのコンポーネントと直接通信するのではなく、メッセージ ベースの通信を使用して、インターフェイスを実装しレイヤー間の通信を提供できます。物理境界とプロセス境界をまたぐ通信をサポートするメッセージング ソリューションには、Windows Communication Foundation、Web サービス、Microsoft Message Queuing などがあります。ただし、抽象インターフェイスを、通信のデータ構造の定義に使用する一般的な種類のメッセージと組み合わせることもできます。メッセージ ベースのインターフェイスとの主な違いは、通信のすべての詳細をカプセル化する共通の構造をレイヤー間の通信で使用することです。この構造では、操作、データ スキーマ、エラー コントラクト、セキュリティ情報など、レイヤー間の通信に関連する多くの構造を定義できます。Web アプリケーションを実装して、プレゼンテーション レイヤーとビジネス レイヤー間のインターフェイスを定義している場合、複数の種類のクライアントをサポートしなければならないアプリケーション レイヤーがある場合、または物理境界とプロセス境界をまたぐ通信をサポートする場合は、メッセージ ベースの手法の使用を検討します。また、共通の構造を使用して通信を形式化する場合や、ステータス情報をメッセージで伝達するステートレスなインターフェイスと通信する場合も、メッセージ ベースの手法の使用を検討します。

Web アプリケーションのプレゼンテーション レイヤーとビジネス ロジック レイヤーの間の通信を実装するには、メッセージ ベースのインターフェイスの使用をお勧めします。ビジネス レイヤーで呼び出し間の状態を保持しない場合 (つまり、プレゼンテーション レイヤーとビジネス レイヤーの間の呼び出しがそれぞれ新しいコンテキストを表す場合)、要求と併せてコンテキスト情報を渡し、プレゼンテーション レイヤーで例外とエラー ハンドルに関する一般的なモデルを提供できます。

手順 8 - 配置に関する方針を選択する

多くのソリューションで採用されているアプリケーションの配置構造には、いくつかの一般的なパターンがあります。アプリケーションに最適な配置ソリューションを決定する際には、まず、一般的なパターンを確認します。さまざまなパターンを十分に理解したら、次にシナリオ、要件、およびセキュリティ上の制約を検討して最も適切なパターンを 1 つまたは複数選択します。配置パターンに関する詳細については、第 19 章「物理層と配置」を参照してください。

手順 9 - 通信プロトコルを選択する

設計のレイヤーまたは層の間で行われる通信に使用する物理的なプロトコルは、アプリケーションのパフォーマンス、セキュリティ、および信頼性において重要な役割を果たします。通信プロトコルの選択は、分散配置の検討よりも重要です。コンポーネントが同じ物理層に配置されると、多くの場合、コンポーネント間の直接的な通信をあてにできます。ただし、多くのシナリオで行われるように、コンポーネントとレイヤーを物理的に別のサーバーとクライアント コンピューターに配置する場合、このようなレイヤーのコンポーネント間で、効率的かつ正確に通信する方法を検討する必要があります。通信プロトコルと通信テクノロジの詳細については、第 18 章「通信とメッセージ」を参照してください。

6

プレゼンテーション レイヤーのガイドライン

概要

この章では、アプリケーションのプレゼンテーション レイヤーの設計に関する主要なガイドラインを示します。このガイドラインは、一般的なレイヤー型アプリケーション アーキテクチャにプレゼンテーション レイヤーにおける位置付け、プレゼンテーション レイヤーに通常含まれるコンポーネント、およびプレゼンテーション レイヤーの設計時に直面する主な問題について理解するのに役立ちます。また、設計、推奨される設計手順、関連する設計パターン、およびテクノロジの選択肢に関するガイドラインも紹介します。

プレゼンテーション レイヤーには、ユーザー インターフェイスを実装および表示して、ユーザーの操作を管理するコンポーネントが含まれています。また、ユーザーが入力および表示するためのコントロールやユーザーの操作を構成するコンポーネントも含まれています。図 1 は、一般的なアプリケーション アーキテクチャにおけるプレゼンテーション レイヤーの位置付けを示しています。

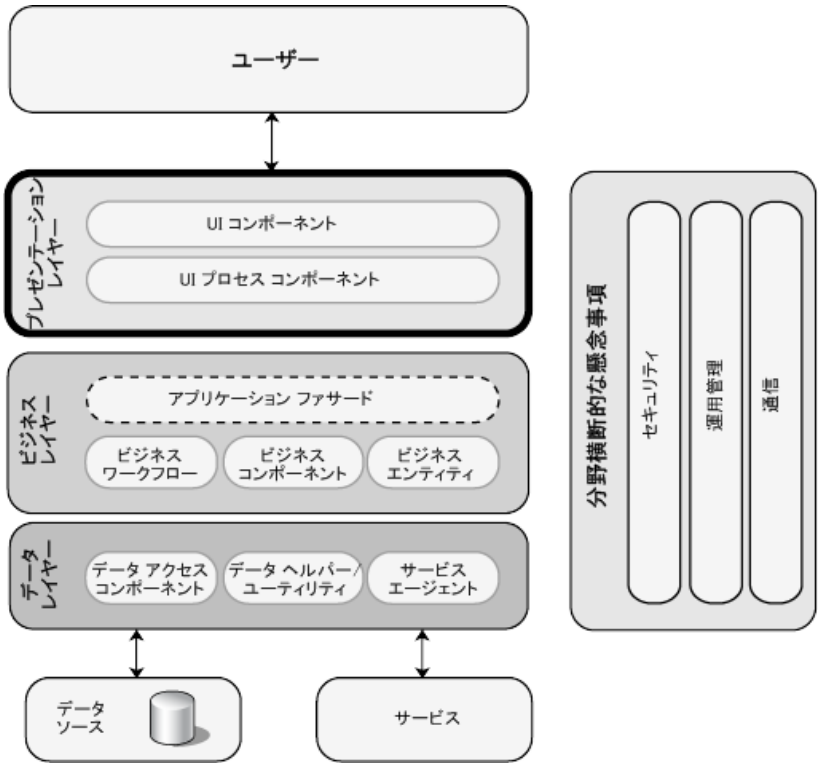


図 7

一般的なアプリケーションのプレゼンテーション レイヤーとそこに含まれる場合があるコンポーネント

通常、プレゼンテーション レイヤーには、次のコンポーネントが含まれています。

- **ユーザー インターフェイス コンポーネント**: ユーザーに情報を表示したり、ユーザーからの入力を受け付けたりするのに使用するアプリケーションの視覚的要素です。
- **プレゼンテーション ロジック コンポーネント**: プレゼンテーション ロジックとは、特定のユーザー インターフェイスの実装に依存しない形で、アプリケーションの論理的な動作と構造を定義するアプリケーション コードのことです。Separated Presentation パターンを実装すると、プレゼンテーション ロジック コンポーネントには Presenter、Presentation Model、および ViewModel コンポーネントが含まれる場合があります。また、プレゼンテーション レイヤーには、ビジネス レイヤーのデータをカプセル化する Presentation Layer Model コンポーネントや、ビジネス ロジックとビジネス データをプレゼンテーション レイヤーで簡単に使用できる形式でカプセル化した Presentation Entity コンポーネントも含まれる場合があります。

プレゼンテーション レイヤーで一般的に使用されるコンポーネントの詳細については、第 10 章「コンポーネントのガイドライン」を参照してください。プレゼンテーション レイヤーのコンポーネントの設計に関する詳細については、第 11 章「プレゼンテーション レイヤーのコンポーネントの設計」を参照してください。

設計に関する一般的な考慮事項

プレゼンテーション レイヤーを設計する際に考慮する必要がある主要な要素はいくつかあります。設計がアプリケーションの要件を満たして、ベスト プラクティスに従うようにするには、次の原理を使用します。

- **適切なアプリケーションの種類を選択する**: 選択するアプリケーションの種類は、プレゼンテーション レイヤーの選択肢に多大な影響を及ぼします。リッチ (スマート) クライアント、Web クライアント、またはリッチ インターネット アプリケーション (RIA) を実装するかどうかを決定します。これは、アプリケーションの要件と、組織やインフラストラクチャの制約に基づいて決定します。主要なアプリケーションの規範と、そのメリットおよびデメリットの詳細については、第 20 章「アプリケーションの種類の選択」を参照してください。
- **適切な UI テクノロジを選択する**: アプリケーションの種類によって、プレゼンテーション レイヤーの開発に使用できるテクノロジは異なります。それぞれのテクノロジには、適切なプレゼンテーション レイヤーの設計の成否に影響する可能性がある明確なメリットがあります。それぞれのアプリケーションの種類で使用できるテクノロジの詳細については、付録 B 「プレゼンテーション テクノロジ」を参照してください。
- **関連するパターンを使用する**: この章の最後で紹介するプレゼンテーション レイヤーのパターンで、一般的なプレゼンテーションの問題に対する実証済みの解決策を確認します。すべてのパターンがすべてのアプリケーションの規範に同じように適用されるわけではないことに注意してください。ただし、基になるアプリケーション ロジックからプレゼンテーション固有の懸念事項を分離する汎用的な Separated Presentation パターンは、すべてのアプリケーションの種類に適用されます。MVC、MVP、Supervising Presenter などのパターンは、リッチ クライアント アプリケーションと RIA のプレゼンテーション レイヤーの設計でよく使用されます。Web アプリケーションでは、Model-View-Controller (MVC) パターンと Model-View-Presenter (MVP) パターンの変化形を使用できます。
- **懸念事項の分離を設計する**: レンダリング、表示、およびユーザーの操作に重点を置く、専用の UI コンポーネントを使用します。ユーザー操作が複雑な場合や単体テストを行う必要がある場合は、ユーザー操作の処理の管理に専用のプレゼンテーション ロジック コンポーネントを使用することを検討します。また、UI とプレゼンテーション ロジック コンポーネントで簡単に使用できる形式でビジネス ロ

ジックとデータを表す、専用のプレゼンテーション エンティティを使用することも検討します。プレゼンテーション エンティティは、ビジネス レイヤーでビジネス エンティティが使用されるのとほぼ同じ方法で使用できるので、ビジネス レイヤーのビジネス ロジックとビジネス データがプレゼンテーション レイヤー内にカプセル化されます。使用できるプレゼンテーション レイヤーのコンポーネントの種類に関する詳細については、第 11 章「プレゼンテーション レイヤーのコンポーネントの設計」を参照してください。

- **ヒューマン インターフェイスのガイドラインを考慮する**: プレゼンテーション レイヤーの設計時に、アクセシビリティ、ローカリゼーション、ユーザビリティなどの要素を含む、組織の UI 設計のガイドラインを実装します。
- 既定の UI のガイドラインで、選択するクライアントの種類とテクノロジに基づいて対話性、ユーザビリティ、システムの互換性、準拠、および関連する UI の設計パターンを確認し、適用可能なガイドラインをアプリケーションの設計と要件に適用します。
- **ユーザー主導の設計原理を遵守する**: プレゼンテーション レイヤーを設計する前に、ユーザーについて理解します。アンケート、ユーザビリティに関する調査、およびインタビューを実施して、ユーザーの要件に合う最適なプレゼンテーションの設計を決定します。

設計に関する具体的な問題

設計を策定する際に考慮する必要がある一般的な問題がいくつかあります。これらの問題は、設計の特定の領域に分類できます。次のセクションでは、最も頻繁にミスが発生する一般的な領域に関するガイドラインを示します。

- キャッシュ
- 通信
- 構成
- 例外管理
- ナビゲーション
- ユーザー エクスペリエンス
- ユーザー インターフェイス
- 検証

キャッシュ

キャッシュは、アプリケーションのパフォーマンスと UI の応答性を向上するのに使用できる最適なメカニズムの 1 つです。プレゼンテーション レイヤーでデータ キャッシュを使用すると、データの検索処理を最適化して、ネットワークのラウンド トリップを回避し、負荷の高い処理や繰り返し行われる処理の結果を格納して、同じ処理が不要に繰り返されないようにすることができます。キャッシュの方針を設計する際には、次のガイドラインを考慮します。

- キャッシュに適した場所 (メモリ内やディスク上など) を選択します。アプリケーションを Web ファームに展開する場合は、同期する必要があるローカル キャッシュは使用しないようにします。一般的に、Web ファームやアプリケーション ファームに展開する場合は、トランザクション リソース マネージャー (Microsoft SQL Server® など) や、分散キャッシュがサポートされる製品 (Danga Interactive の Memcached テクノロジやマイクロソフトの Velocity のキャッシュ メカニズムなど) の使用を検討します。ただし、サーバー間の差異が重大ではない場合やデータの変化が緩やかな場合は、メモリ内のキャッシュの使用が適しています。
- メモリ内のキャッシュを使用する場合は、すぐに使用できる形式でデータをキャッシュすることを検討します。たとえば、データベースのデータをそのままキャッシュするのではなく、具体的なオブジェク

トを使用します。ただし、データが頻繁に変更される場合は、キャッシュのコストが、データを再作成または取得するコストより高くなることがあるので、揮発性データはキャッシュしないようにします。

- 機密データをキャッシュする場合は、必ず暗号化します。
- キャッシュ内のデータは削除されている可能性があるので、キャッシュ内にデータが存在していることを当てにしないようにします。また、キャッシュ データは、古いデータになっている可能性があることについても考慮します。たとえば、ビジネス トランザクションを実行する場合は、キャッシュ データを使用するのではなく、最新のデータを取得してトランザクションに適用することをお勧めします。
- キャッシュ データに対する承認の権限を考慮します。さまざまな役割を持ったユーザーがデータにアクセスする可能性がある場合は、適切な承認を適用できるデータのみをキャッシュします。
- 複数のスレッドを使用している場合は、キャッシュへのすべてのアクセスがスレッド セーフになるようにします。

キャッシュの技法に関する詳細については、第 17 章「分野横断的な懸念事項」を参照してください。

通信

実行時間の長い要求では、ユーザーの応答性だけでなく、コードの保守容易性とテスト容易性を考慮に入れて対処します。要求処理を設計する際には、次のガイドラインを考慮します。

- 非同期操作やワーカー スレッドを使用して、Windows フォームと WPF アプリケーションで実行時間が長い操作の UI がブロックされないように考慮します。ASP.NET では、AJAX を使用して非同期要求を実行することを考慮します。実行時間の長い操作の進行状況をユーザーに通知します。また、ユーザーが実行時間の長い操作をキャンセルできるようにすることについても考慮します。
- UI の処理ロジックとレンダリング ロジックが混在しないようにします。
- Web サービスの呼び出しやデータベースのクエリなど、リモートのソースやレイヤーに対して負荷の高い呼び出しを行う場合は、これらの呼び出しを chatty (小さな多数の要求) か chunky (大きな 1 つの要求) のどちらにするのが適切かを検討します。ユーザーがタスクを完了するために大量のデータを必要とする場合は、データを表示して、処理を開始するのに必要なデータだけを取得し、その後、バックグラウンド スレッドで残りのデータを徐々に取得するか、ユーザーが要求するたびに追加のデータを取得するかを検討します (この手法は、データのページング、UI の仮想化などで使用されます)。ユーザーがタスクを完了するために呼び出しが完了するのを待機する必要がない場合は、より大きい chunky な呼び出しの使用を検討します。

構成

プレゼンテーション レイヤーで、独立したモジュールと実行時に構成されるビューを使用した場合に、アプリケーションの開発と管理が容易になるかどうかを検討します。UI の構成パターンでは、実行時にビューとプレゼンテーションのレイアウトを作成することがサポートされています。また、このようなパターンは、コードとライブラリの依存関係を最小限に抑えるのにも役立ちます。このパターンを使用しない場合、依存関係が変化すると、モジュールの再コンパイルと再展開が強制的に実行されます。構成パターンは、プレゼンテーション ロジックとビューの共有、再利用、および置換を実装するのに役立ちます。UI の構成の方針を設計する際には、次のガイドラインを考慮します。

- コンポーネント間の依存関係を回避します。たとえば、可能であれば抽象パターンを使用して、保守容易性に関する問題を回避します。実行時の依存関係の挿入をサポートするパターンの使用も検討します。
- プレースホルダーを伴うテンプレートの作成を検討します。たとえば、Template View パターンを使用して動的な Web ページを構成し、再利用性と一貫性を確保します。
- 再利用可能なモジュールからビューを構成することを検討します。たとえば、Composite View パターンを使用して、モジュール形式のアトミックなコンポーネントからビューを構築します。簡単に追加できる別個のモジュールを使用して、アプリケーションを分離することを検討します。
- 実行時に動的に生成されたレイアウトは、読み込みと管理が難しいことがあるので、使用する場合は注意が必要です。実行時にビューとプレゼンテーションの動的なレイアウトおよび挿入をサポートする、パターンとサード パーティ製のライブラリを確認します。
- プレゼンテーション レイヤーのコンポーネント間で通信する場合は、Publish/Subscribe などの疎結合された通信パターンの使用を検討します。このパターンを使用すると、コンポーネント間の結合が緩やかになり、テスト容易性と柔軟性が向上します。

例外管理

アプリケーションに実装する一元化された例外管理のメカニズムを設計して、予期しない例外 (ローカルで回復できない例外) を一貫した方法でキャッチして管理します。レイヤーや層の境界を越えて伝達される例外と信頼境界を越える例外には特に注意する必要があります。例外管理の方針を設計する際には、次のガイドラインを考慮します。

- ユーザーには、わかりやすいエラー メッセージを提供して、アプリケーションで発生したエラーを通知します。ただし、エラー ページ、エラー メッセージ、ログ ファイル、および監査ファイルでは機密データを公開しないようにします。可能な場合は、アプリケーションで一貫性のある状態を維持します。それができない場合は、アプリケーションを終了することを検討します。
- 他の場所 (グローバル エラー ハンドラー内など) でキャッチされない例外をキャッチして、例外発生後にリソースと状態がクリーンアップされるようにします。グローバルなエラー ページやエラー メッセージを表示するグローバル例外ハンドラーは、ハンドルされない例外に対処するのに役立ちます。通常、ハンドルされない例外は、システムが一貫性のない状態であることを示すため、システムをシャットダウンしなければならない場合があります。
- システム例外とビジネス エラーを区別します。ビジネス エラーの場合は、わかりやすいエラー メッセージを表示して、ユーザーが操作を再試行できるようにします。システム例外の場合は、サービスやデータベースの障害などの問題によって例外が発生しているのかを確認して、わかりやすいエラー メッセージを表示し、トラブルシューティングで使用できるようにエラー メッセージをログに記録します。
- ハンドルできる例外のみをキャッチして、必要でない場合はカスタムの例外を使用しないようにします。アプリケーション ロジックのフローを制御するために、例外を使用しないでください。

例外管理の技法に関する詳細については、第 17 章「分野横断的な懸念事項」を参照してください。

ナビゲーション

ナビゲーションの方針を設計する際には、ユーザーが画面やページ間を簡単に移動できるようにして、ナビゲーションをプレゼンテーションと UI 処理から分離します。アプリケーション全体でナビゲーションのリンクとコントロールを同じように表示すると、ユーザーが混乱したり、複雑だと感じることが少なくなります。ナビゲーションの方針を設計する際には、次のガイドラインを考慮します。

- ツール バーとメニューをデザインする際には、UI で提供される機能をユーザーが見つけやすくなるように考慮します。
- ウィザードを使用して、フォーム間のナビゲーションを予測可能な方法で実装し、必要に応じて、セッション間のナビゲーションの状態を保存する方法を決定することを検討します。
- ナビゲーションのイベント ハンドラーのロジックが重複しないようにして、ハードコーディングされたナビゲーション パスの使用をできる限り回避します。Command パターンを使用して、複数のソースからの一般的な操作を処理することを検討します。

ユーザー エクスペリエンス

アプリケーションの優劣は、ユーザー エクスペリエンスの良し悪しで決まります。実際のパフォーマンスよりも、ユーザーが感じるパフォーマンスの方が重要です。そのため、ユーザー操作のパターンに関するユーザーの期待値を管理して知ることが重要です。たとえば、ページの読み込みが完了するのにかかる時間を提示すれば、ユーザーはページが読み込まれるまで長い間待機してもかまわないかもしれません。この場合は、この待機時間がユーザー操作の妨げになることはありません。その他の状況では、(一部の UI 操作では 1 秒に満たない短時間であっても) 短時間の遅延により、アプリケーションが応答しないように感じる場合もあります。ユーザビリティに関する調査、アンケート、およびインタビューを実施して、ユーザーがアプリケーションに要求および期待するものを理解し、この結果を念頭に置いて効果的な UI を実現する設計を検討します。ユーザー エクスペリエンスを設計する際には、次のガイドラインを考慮します。

- 過剰な機能を備えたインターフェイスや複雑なインターフェイスを設計しないようにします。主要なユーザー シナリオにおけるアプリケーションの操作をわかりやすくしたり、UI の重要な変更 (状態の変化など) にユーザーの注意を引き付けるために色やちょっとしたアニメーションを使用することを検討します。
- 機密データを公開することなく、役に立つ有益なエラー メッセージを提供します。
- 完了するまでに長時間かかる可能性のある処理が、ユーザーの操作をブロックしないようにします。少なくとも、処理の進捗状況を提示して、ユーザーが処理をキャンセルできるようにするかどうかを検討します。
- 構成 (必要に応じてパーソナル化) を通じて、UI に柔軟性を持たせて、カスタマイズできるようにすることで、ユーザーに権限を与えることを検討します。
- 初期設計の主要な要件ではありませんが、ローカリゼーションとグローバリゼーションのサポート方法を検討します。設計が完了してからローカリゼーションとグローバリゼーションのサポートを追加すると、大量の手直しとリファクタリングが必要になる可能性があります。

ユーザー インターフェイス

データ入力とデータ検証の要件をサポートする、適切なユーザー インターフェイスを設計します。ユーザビリティを最大限に高めるには、組織で定義された既定のガイドラインと、業界の既定のユーザビリティに関するガイドライ

ンに従います。業界のガイドラインは、入力の設計とメカニズムに関する、ユーザーを対象とした長年にわたる研究に基づいて策定されています。ユーザー インターフェイスのレイアウトの方針を選択する場合は、レイアウトを構築するデザイナー チームが存在するのか、開発チームが UI を作成するのかを検討します。デザイナーが UI を作成する場合は、コードや開発に特化したツールを使用する必要がないレイアウトの手法を選択します。ユーザー インターフェイスを設計する際には、次のガイドラインを考慮します。

- MVP などの Separated Presentation パターンを使用して、レイアウトの設計をインターフェイスの処理から分離することを検討します。テンプレートを使用すると、すべての UI 画面で共通の外観と動作が提供され、UI のすべての要素で共通の外観と動作が保持されるので、アクセシビリティや使いやすさを最大限に高めることができます。複雑なレイアウトは使用しないようにします。
- データ収集タスクではフォーム ベースの入力コントロールの使用を、テキストや描画ドキュメントなど、より自由な形式の入力の収集ではドキュメント ベースの入力メカニズムの使用を、順序立てられているデータ収集タスクやワークフロー ベースのデータ収集タスクではウィザード ベースの手法の使用を検討します。
- アプリケーションをローカライズする場合は、ハードコーディングされた文字列の使用と、テキストとレイアウトの情報 (右から左に読む言語をサポートするなど) を入手するために外部リソースを使用することは避けます。
- 設計のアクセシビリティを考慮します。入力の方針を設計する際には、障害のあるユーザーについて考慮する必要があります。たとえば、目が不自由なユーザーのために読み上げソフトウェアを実装したり、視力が弱いユーザーのためにテキストや画像を拡大したりします。また、ポインティング デバイスを操作できないユーザーのために、(可能であれば) キーボードのみのシナリオをサポートします。
- さまざまな画面サイズと画面解像度を考慮し、さまざまなデバイスと入力の種類 (モバイル デバイス、タッチ スクリーン、ペンとインクなど) を考慮します。たとえば、タッチ スクリーンの入力では、マウスとキーボードによる入力のみを想定してデザインされた UI よりもボタンの間隔が広い、より大きなボタンを使用することが一般的です。Web アプリケーションを構築する場合は、レイアウトにカスケード スタイル シート (CSS) を使用することを検討します。これにより、レンダリングのパフォーマンスと保守容易性が向上します。

検証

入力とデータの検証を効率的に行う方針を設計することは、アプリケーションのセキュリティや適切な動作を実現するうえで重要です。ユーザーの入力だけでなく、プレゼンテーション レイヤーのビジネス ルールに関する検証規則も決定します。入力とデータの検証の方針を設計する際には、次のガイドラインを考慮します。

- 入力の検証はプレゼンテーション レイヤーで処理され、ビジネス ルールの検証はビジネス レイヤーで処理される必要があります。ただし、ビジネス レイヤーとプレゼンテーション レイヤーが物理的に離れている場合は、ユーザビリティと応答性を向上するために、ビジネス ルールの検証ロジックをプレゼンテーション レイヤーに反映することをお勧めします。これは、メタデータを使用するか、両方のレイヤーに存在する検証規則の共通のコンポーネントを使用して実現できます。
- 悪意のある入力を制限、拒否、および一部削除する検証の方針を設計します。検証を実装する際に役立つ設計パターンとサード パーティ製のライブラリを確認します。検証に適しているビジネス ルール (トランザクションの制限など) を特定し、包括的な検証を実装して、これらのルールが危険にさらされないようにします。
- 検証エラーを正しくハンドルして、エラー メッセージで機密情報が公開されないようにします。また、悪意のある操作の検出に役立てるために、検証で発生したエラーをログに記録するようにします。

検証の技法に関する詳細については、第 17 章「分野横断的な懸念事項」を参照してください。

テクノロジに関する考慮事項

マイクロソフト プラットフォームに関しては、プレゼンテーション レイヤーに適した実装テクノロジを選択する際に、次のガイドラインが役立ちます。また、このガイドラインでは、特定の種類のアプリケーションとテクノロジで役に立つ、一般的なパターンに関する情報も提供します。

モバイル アプリケーション

モバイル アプリケーションを設計する際には、次のガイドラインを考慮します。

- Microsoft Windows ベースのさまざまなデバイスで実行され、常時接続、不定期に接続、オフライン状態で実行できる、完全な機能を備えたアプリケーションを構築する場合は、.NET Compact Framework の使用を検討します。
- さまざまなモバイル デバイスをサポートするか、ワイヤレス アプリケーション プロトコル (WAP)、コンパクト HTML (cHTML)、または同様のレンダリング形式が必要な接続型アプリケーションを構築する場合は、ASP.NET for Mobile の使用を検討します。

リッチ クライアント アプリケーション

リッチ クライアント アプリケーションを設計する際には、次のガイドラインを考慮します。

- リッチ メディアとリッチ グラフィックス対応したアプリケーションを構築する必要がある場合は、Windows Presentation Foundation (WPF) の使用を検討します。
- Web サーバーからダウンロードして Windows クライアントで実行するアプリケーションを構築する必要がある場合は、XAML ブラウザー アプリケーション (XBAP) の使用を検討します。
- ドキュメント ベースのアプリケーションやレポート作成に使用するアプリケーションを構築する必要がある場合は、Microsoft Office Business Application (OBA) を設計することを検討します。
- さまざまなサード パーティ製のコントロールや迅速なアプリケーション開発をサポートするツールを活用する必要がある場合は、Windows フォームの使用を検討します。Windows フォームを使用して複合アプリケーションを設計する場合は、patterns & practices の Smart Client Software Factory の使用を検討します。
- WPF を使用してアプリケーションを構築する場合は、次の事項を考慮します。
    - 複合アプリケーションでは、patterns & practices の Composite Client Application Guidance の使用を検討します。
    - Presentation Model (Model-View-ViewModel) パターンの使用を検討します。これは、(View が従来のように開発者によって作成されるのではなく、デザイナーによって作成される) 現在の UI 開発プラットフォームに合わせてカスタマイズされた、Model-View-Controller (MVC) の変化形です。これは、ユーザー コントロールに DataTemplate を実装して、デザイナーが細かく制御できるようにすることで実現できます。また、WPF のコマンドを使用して、View と Presenter や ViewModel 間で通信することも検討します。

リッチ インターネット アプリケーション

リッチ インターネット アプリケーション (RIA) を設計する際には、次のガイドラインを考慮します。

- 複数のプラットフォームを対象とし、多数のグラフィックスを使用し、リッチ メディアとプレゼンテーションの機能をサポートするブラウザー ベースの接続型アプリケーションを構築する必要がある場合は、Silverlight の使用を検討します。
- Silverlight を使用してアプリケーションを構築する場合は、次の項目を考慮します。
    - この章の前半で説明したように、Presentation Model (Model-View-ViewModel) パターンの使用を検討します。
    - 持続性を持ちながら変化する必要があるアプリケーションを設計する場合は、patterns & practices の Composite Client Application Guidance の使用を検討します。

Web アプリケーション

Web アプリケーションを設計する際には、次のガイドラインを考慮します。

- Web ブラウザーや専用のユーザー エージェント経由でアクセスするアプリケーションを構築する必要がある場合は、ASP.NET の使用を検討します。
- ASP.NET を使用してアプリケーションを構築する場合は、次の事項を考慮します。
    - マスター ページを使用して、すべてのページで使用する一貫した UI の開発と実装を簡略化することを考慮します。
    - ページの再読み込み回数を少なくして、対話性とバックグラウンド処理を強化するには、AJAX と ASP.NET Web フォームの併用を検討します。
    - 独立したリッチ メディア コンテンツと対話性を含める必要がある場合は、Silverlight コントロールと ASP.NET の併用を検討します。
    - アプリケーションのテスト容易性を向上したり、アプリケーションのユーザー インターフェイスとビジネス ロジックをより明確に分離したりする必要がある場合は、ASP.NET の MVC

Framework の使用を検討します。このフレームワークでは、Web アプリケーション開発で Model-View-Controller ベースの手法がサポートされます。

---

patterns & practices の Smart Client Software Factory と Composite Client Application Guidance の詳細については、この章の後半にある「patterns & practices のサービス」を参照してください。

パフォーマンスに関する考慮事項

プレゼンテーション レイヤーのパフォーマンスを最大限に高めるには、次のガイダンスを考慮します。

- プレゼンテーション レイヤーを注意深く設計して、豊富な機能と応答性を備えたユーザー エクスペリエンスを実現するのに必要な機能が含まれるようにします。たとえば、プレゼンテーション レイヤーでは、層間で通信を行うことなく、ユーザーの入力をすぐに検証できるようにします。この処理を実現するには、メタデータや共有コンポーネントを使用して、ビジネス レイヤーのデータ検証規則がプレゼンテーション レイヤーに反映されることが必要な場合があります。
- アプリケーションのプレゼンテーション レイヤーとビジネス レイヤーまたはサービス レイヤー間のやりとりは、非同期に行われる必要があります。このようにすると、長い待ち時間や断続的な接続が、アプリケーションのユーザビリティや応答性に悪影響を及ぼす可能性が低くなります。
- ユーザーに表示するデータは、プレゼンテーション レイヤーにキャッシュすることを検討します。たとえば、株価情報に表示された履歴情報をキャッシュできます。
- 一般的に、ユーザーの数が限られている場合や、データの合計サイズが比較的小さい場合を除き、セッション データを保持したり、ユーザー固有のデータをキャッシュしたりしないようにします。ただし、ユーザーがしばらく操作を続行することが多い場合に、ユーザー固有のデータを一時的にキャッシュすることは、適切な技法です。セッション データやユーザー固有のデータを格納またはキャッシュする場合は、Web ファームやアプリケーション ファーム内で発生するアフィニティの問題に注意してください。
- クエリを実行して情報を要求する場合は、常にデータ ページングを使用します。際限のない量のデータを返す可能性があるクエリには依存せず、表示するデータの量に適したデータ ページのサイズを使用します。クライアント側のページングは、やむを得ない場合にのみ使用します。
- ASP.NET でビュー ステートを使用すると、各ラウンド トリップに含まれるデータの量が増加し、アプリケーションのパフォーマンスが低下する可能性があるため、これを使用する場合には注意します。

必要に応じて、読み取り専用のセッションを使用するようにページを構成するか、セッションを一切保持しないようにページを構成することを検討します。

プレゼンテーション レイヤーの設計手順

次の手順は、アプリケーションのプレゼンテーション レイヤーの設計に関する推奨プロセスです。この手法を使用すると、アーキテクチャを開発する際に、関連するすべての要素を考慮できます。手順は次のとおりです。

1. **クライアントの種類を特定する**: 要件を満たし、組織のインフラストラクチャと開発の制約に準拠するクライアントの種類を選択します。たとえば、ユーザーがモバイル デバイスを携帯していて、ネットワークに断続的に接続する場合は、モバイル クライアントが最適な選択肢です。適切なクライアントの種類を選択するのに役立つ情報については、第 20 章「アプリケーションの種類の選択」を参照してください。
2. **プレゼンテーション レイヤーのテクノロジを選択する**: UI とプレゼンテーション レイヤーの大まかな機能を特定し、これらの要件を満たして、選択したクライアントの種類で使用できる UI テクノロジを選択します。この時点で、使用可能なテクノロジの中に適しているものがない場合は、クライアントの種類を再検討することをお勧めします。それぞれのアプリケーションの種類で使用できるテクノロジの詳細については、付録 B 「プレゼンテーション テクノロジ」を参照してください。
3. **ユーザー インターフェイスを設計する**: UI をモジュール形式にする必要があるかどうかを検討して、プレゼンテーション レイヤー内で懸念事項の分離を強化する方法を決定します。Presentation Model、MVC、MVP などの Separated Presentation パターンの使用を検討します。この章の前半の「構成」、「ナビゲーション」、「ユーザー エクスペリエンス」、および「ユーザー インターフェイス」で紹介したガイドラインを使用して、要件を満たす適切な UI を設計します。設計で使用する可能性があるコンポーネントの種類の詳細については、第 11 章「プレゼンテーション レイヤーのコンポーネントの設計」を参照してください。
4. **データ検証の方針を決定する**: データ検証の技法を使用して、信頼されない入力からシステムを保護します。また、例外処理とログ記録の適切な方針も決定します。検証、例外処理、およびログ記録に関する適切な方針の実装の詳細については、第 17 章「分野横断的な懸念事項」を参照してください。
5. **ビジネス ロジックの方針を決定する**: ビジネス ロジックを取り除いて、これをプレゼンテーション レイヤーのコードから分離します。これにより、アプリケーションの保守容易性が向上し、プレゼンテー

ション レイヤーに影響を及ぼすことなく、ビジネス ロジックを簡単に変更できるようになります。選択する技法は、アプリケーションの複雑さによって決まります。この一般的な手法を次に示します。

- **UI の検証**: ユーザーの入力を検証するためだけにビジネス ロジックを使用する単純なアプリケーションでは、ビジネス ロジックを UI コンポーネントに配置することが可能です。ただし、UI コンポーネント内で、検証に関係のないビジネス ロジックと混在しないように注意する必要があります。
- **ビジネス プロセス コンポーネント**: 複雑なアプリケーション、トランザクションをサポートするアプリケーション、または UI の検証以外にも基本的なビジネス ロジックを含むアプリケーションでは、UI コンポーネントで使用する別個のコンポーネントにビジネス ロジックを配置することを検討します。
- **ドメイン モデル**: ビジネス ロジックを複数のアプリケーションで共有する複雑なエンタープライズ アプリケーションでは、ビジネス コンポーネントを別個の論理レイヤーに分離することを検討します。このようにすると、ビジネス レイヤーを別個の物理層に配置できるので、スケーラビリティが向上し、他のアプリケーションによる再利用がサポートされます。
- **ルール エンジン**: 複雑な検証、処理のオーケストレーション、およびドメイン ロジックをサポートする必要があるアプリケーションでは、ビジネス ロジックを Microsoft BizTalk® Server などのルール エンジンに配置することを検討します。

6. **他のレイヤーとの通信に関する方針を決定する**: アプリケーションに、データ アクセス レイヤー、ビジネス レイヤーなど、複数のレイヤーが含まれる場合は、プレゼンテーション レイヤーとその他のレイヤーとの通信に関する方針を決定します。独立したビジネス レイヤーが存在している場合、プレゼンテーション レイヤーではビジネス レイヤーと通信します。ビジネス レイヤーが存在しない場合、プレゼンテーション レイヤーではデータ アクセス レイヤーと直接通信します。その他の層にアクセスするには、次の技法を使用します。

   - **直接的なメソッド呼び出し**: 通信するレイヤーがプレゼンテーション レイヤーと同じ物理層にある場合は、メソッドを直接呼び出せます。
   - **Web サービス**: データ アクセスやビジネス ロジックを他のアプリケーションと共有する必要がある場合、ビジネス レイヤーやデータ アクセス レイヤーがプレゼンテーション レイヤーと別の層に配置されている場合、または分離を重視する場合は、Web サービス インターフェイスを使用します。ビジネス ロジックやデータ アクセス ロジックが、イントラネット内のプレゼンテーション レイヤーで使用される場合は、WCF で TCP プロトコルを使用することを

考慮します。ビジネス ロジックやデータ アクセス ロジックが、インターネット内のプレゼンテーション レイヤーで使用される場合は、WCF で HTTP プロトコルを使用することを考慮します。ビジネス ロジックやデータ アクセス ロジックで実行時間が長い呼び出しが行われる場合は、WCF とメッセージ キューを使用する非同期通信の採用を検討します。

適切な通信方法の実装に関する詳細については、第 18 章「通信とメッセージ」を参照してください。

関連する設計パターン

次の表に示すようにプレゼンテーション レイヤーの主要なパターンはカテゴリに分類されます。各カテゴリの設計を行う際には、これらのパターンの使用を検討します。

カテゴリ	**関連するパターン**
キャッシュ	**Cache Dependency**: 外部からの情報を使用して、キャッシュに格納されているデータの状態を判断します。 **Page Cache**: 頻繁にアクセスされ、変更の頻度が低く、構築に大量のシステム リソースを消費する動的な Web ページの応答時間が向上します。
構成とレイアウト	**Composite View**: 個々のビューを組み合わせて、複合的に表します。 **Presentation Model** (Model-View-ViewModel) パターン: 現在の UI 開発プラットフォーム (View が従来のように開発者によって作成されるのではなく、デザイナーによって作成されます) に合わせてカスタマイズされた Model-View-Controller (MVC) の変化形です。 **Template View**: 共通のテンプレート ビューを実装し、このテンプレート ビューを使用してビューを取得または構成します。 **Transform View**: プレゼンテーション層に渡されたデータを HTML に変換して、UI で表示します。 **Two-Step View**: モデル データを、具体的な形式を指定しない論理表現に変換してから、その論理表現を変換して、必要な実際の形式を追加します。
例外管理	**Exception Shielding**: 例外が発生したときに、サービスの内部実装に関する情報が公開されないようにします。
ナビゲーション	**Application Controller**: 画面のナビゲーションを処理する一元化された場所です。

	Front Controller: 単一のハンドラー オブジェクトを通じて、すべての要求を渡すことで要求処理を統合する、Web のみに対応したパターンです。このハンドラー オブジェクトは、実行時にデコレータを使用して変更できます。 **Page Controller**: 要求からの入力を受け取り、その入力を特定のページや Web サイトの操作で処理します。 **Command**: 要求処理を、共通の実行インターフェイスを備えた別個のコマンド オブジェクトにカプセル化します。
ユーザー エクスペリエンス	**Asynchronous Callback**: 実行時間が長いタスクを、バックグラウンドで実行される独立したスレッドで実行し、タスクが完了したときにスレッドがコールバックする機能を提供します。 **Chain of Responsibility**: 複数のオブジェクトが要求を処理できるようにすることで、要求の送信者と受信者が一対一で結合されないようにします。

Page Cache パターンの詳細については、「Microsoft .NET を使用したエンタープライズ ソリューション パターン」(http://msdn.microsoft.com/ja-jp/library/ms998469.aspx) を参照してください。

Application Controller、Front Controller、Page Controller、Template View、Transform View、および Two-Step View の各パターンの詳細については、Martin Fowler 著 『エンタープライズ アプリケーション アーキテクチャ パターン』 (翔泳社、2005 年) を参照するか、http://martinfowler.com/eaaCatalog (英語) を参照してください。

Composite View パターンと Presentation Model パターンの詳細については、「Patterns in the Composite Application Library」(http://msdn.microsoft.com/en-us/library/dd458924.aspx、英語) を参照してください。

Chain of Responsibility パターンの詳細については、「開放/閉鎖原則」(http://msdn.microsoft.com/ja-jp/magazine/cc546578.aspx) を参照してください。

Command パターンの詳細については、Erich Gamma、Richard Helm、Ralph Johnson、John Vlissides 共著 『オブジェクト指向における再利用のためのデザイン パターン』(ソフトバンク クリエイティブ、1999 年) の第 5 章「振る舞いに関するパターン」を参照してください。

Asynchronous Callback パターンの詳細については、「Creating a Simplified Asynchronous Call Pattern for Windows Forms Applications」(http://msdn.microsoft.com/en-us/library/ms996483.aspx、英語) を参照してください。

Exception Shielding パターンと Entity Translator パターンの詳細については、「Useful Patterns for Services」(http://msdn.microsoft.com/en-us/library/cc304800.aspx、英語) を参照してください。

patterns & practices のサービス

マイクロソフトの patterns & practices グループが提供している関連サービスの詳細については、次のリソースを参照してください。

- Composite Client Application Guidance (http://msdn.microsoft.com/en-us/library/cc707819.aspx、英語)
- Smart Client Software Factory (http://msdn.microsoft.com/en-us/library/aa480482.aspx、英語)
- Web Client Software Factory (http://msdn.microsoft.com/en-us/library/bb264518.aspx、英語)

関連情報

Web リソースに簡単にアクセスするには、http://www.microsoft.com/architectureguide (英語) でオンライン版の参考文献を参照してください。

- Choosing the Right Presentation Layer Architecture (http://msdn.microsoft.com/en-us/library/aa480039.aspx、英語)
- 分散メモリ オブジェクト キャッシュ システムの memcached (http://www.danga.com/memcached/、英語)
- Microsoft Inductive User Interface Guidelines (http://msdn.microsoft.com/en-us/library/ms997506.aspx、英語)
- マイクロソフト プロジェクト コード名 Velocity (http://msdn.microsoft.com/ja-jp/data/cc655792.aspx)
- User Interface Text Guidelines (http://msdn.microsoft.com/en-us/library/bb158574.aspx、英語)
- Web クライアントの設計と実装に関するガイドライン (http://msdn.microsoft.com/ja-jp/library/ms978631.aspx)
- Web プレゼンテーションのパターン (http://msdn.microsoft.com/ja-jp/library/ms998516.aspx)

7

ビジネス レイヤーのガイドライン

概要

この章では、アプリケーションのビジネス レイヤーの設計に関する主要なガイドラインを示します。このガイドラインは、一般的なレイヤー型アプリケーションのアーキテクチャにおけるビジネス レイヤーの位置付け、ビジネス レイヤーに通常含まれるコンポーネント、およびビジネス レイヤーの設計時に発生する主要な問題について理解するのに役立ちます。また、設計、推奨される設計手順、関連する設計パターン、およびテクノロジの選択肢に関するガイドラインも紹介します。図 1 は、一般的なアプリケーション アーキテクチャにおけるビジネス レイヤーの位置付けを示しています。

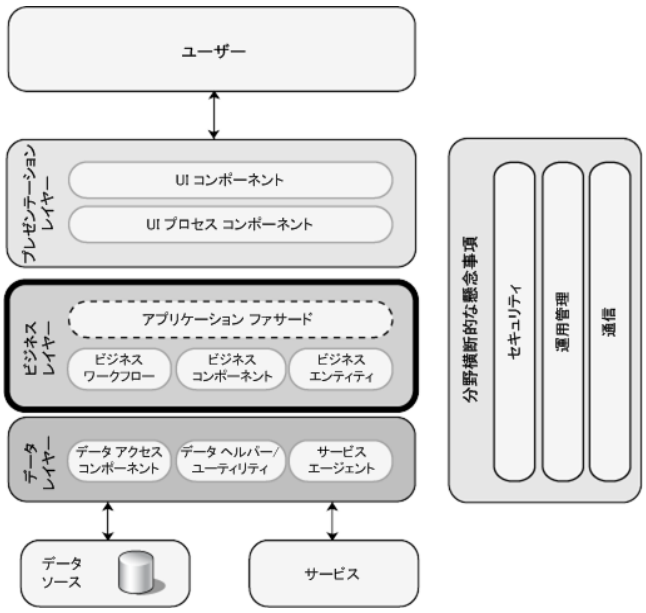


図 8

一般的なアプリケーションのビジネス レイヤーとそこに含まれる場合があるコンポーネント

通常、ビジネス レイヤーには、次のコンポーネントが含まれます。

- **アプリケーション ファサード**: 一般的に、このオプション コンポーネントでは、複数のビジネス操作をビジネス ロジックが使用しやすくなる単一の操作に統合することによって、簡略化されたインターフェイスがビジネス ロジック コンポーネントに提供されます。外部の呼び出し元では、ビジネス コンポーネントの詳細と、コンポーネント間の関係を把握する必要がないので、依存関係が軽減されます。
- **ビジネス ロジック コンポーネント**: ビジネス ロジックは、アプリケーション データの取得、処理、変換、および管理、ビジネス ルールとビジネス ポリシーの適用、そしてデータの一貫性と妥当性の確保に関連するアプリケーション ロジックです。再利用の機会を最大限に高めるには、ビジネス ロジック コンポーネントには、ユース ケースまたはユーザー ストーリー固有の動作やアプリケーション ロジックを含めないようにします。ビジネス ロジック コンポーネントは、さらに次の 2 つのカテゴリに分類できます。
    - **ビジネス ワークフロー コンポーネント**: UI コンポーネントで、ユーザーから必要なデータを収集してビジネス レイヤーに渡したら、アプリケーションでは、そのデータを使用してビジネス プロセスを実行できます。多くのビジネス プロセスには、正しい順序で実行する必要がある複数の手順が含まれており、ビジネス プロセス間ではオーケストレーション経由で通信が行われる場合があります。ビジネス ワークフロー コンポーネントでは、実行時間が長い多段階のビジネス プロセスが定義および調整され、このコンポーネントはビジネス プロセス管理ツールを使用して実装できます。また、操作をインスタンス化して実行するビジネス プロセス コンポーネントと連動しています。ビジネス ワークフロー コンポーネントの詳細については、第 14 章「ワークフロー コンポーネントの設計」を参照してください。
    - **ビジネス エンティティ コンポーネント**: ビジネス エンティティ (一般的に言うビジネス オブジェクト) では、顧客や注文など、アプリケーション内で実際の要素を表すのに必要なビジネス ロジックとビジネス データをカプセル化します。ビジネス エンティティでは、データ値がプロパティに格納および公開され、アプリケーションで使用するビジネス データを保持して管理し、ビジネス データと関連機能へのプログラムによるステートフルなアクセスが提供されます。また、一貫性を確保して、ビジネス ルールと動作を実装するために、エンティティに含まれるデータを検証して、ビジネス ロジックをカプセル化します。ビジネス エンティティ コンポーネントの詳細については、第 13 章「ビジネス エンティティの設計」を参照してください。

ビジネス レイヤーで一般的に使用されるコンポーネントの詳細については、第 10 章「コンポーネントのガイドライン」を参照してください。
ビジネス レイヤーのコンポーネントの設計に関する詳細については、第 12 章「ビジネス レイヤーのコンポーネントの設計」を参照してください。

設計に関する一般的な考慮事項

ビジネス レイヤーを設計する際、ソフトウェア アーキテクトの目標は、タスクを関連領域に分離して、ソフトウェアがなるべく複雑にならないようにすることです。たとえば、ビジネス ルール、ビジネス ワークフロー、およびビジネス エンティティを処理するロジックは、すべて別個の関連領域を表しています。関連領域に分離して設計したコンポーネントでは、その領域に的を絞って、他の関連領域に関係のあるコードを含めないようにします。ビジネス レイヤーを設計する際には、次のガイドラインを考慮します。

- **別個のビジネス レイヤーが必要かどうか判断する**: 可能な場合は、別個のビジネス レイヤーを使用して、アプリケーションの保守容易性を高めることをお勧めします。ただし、(データ検証以外の) ビジネス ルールがほとんどないか、まったくないアプリケーションは例外とします。
- **ビジネス レイヤーの機能とコンシューマーを特定する**: これを特定すると、ビジネス レイヤーで実行するタスクや、ビジネス レイヤーを公開する方法を決定するのに役立ちます。ビジネス レイヤーでは、複雑なビジネス ルールの処理、データの変換、ポリシーの適用、および検証を実行します。プレゼンテーション レイヤーと外部アプリケーションでビジネス レイヤーが使用される場合、ビジネス レイヤーをサービス経由で公開します。
- **ビジネス レイヤーに異なる種類のコンポーネントが混在しないようにする**: ビジネス レイヤーは、プレゼンテーションとデータにアクセスするコードがビジネス ロジック コードと混在するのを回避し、ビジネス ロジックからプレゼンテーション ロジックとデータ アクセス ロジックを分離して、ビジネス機能のテストを簡略化するために使用します。また、ビジネス ロジックの共通機能を一元化して、再利用を促進する目的にも使用します。
- **リモートのビジネス レイヤーにアクセスする際のラウンド トリップの回数を減らす**: ビジネス レイヤーが、通信する必要のあるレイヤーとクライアントから分離された物理層に存在する場合、メッセージ ベースのリモート アプリケーション ファサードか、細かい操作を少ない数の大まかな操作に統合するサービス レイヤーを実装することを検討します。また、ネットワーク経由で転送されるデータには、データ転送オブジェクト (DTO) のような、大まかなパッケージを使用することを検討します。

- **レイヤー間の密結合を避ける**: ビジネス レイヤーのインターフェイスを作成する際には、抽象化の原理を使用して、結合を最小限に抑えます。抽象化の手法には、パブリック オブジェクト インターフェイス、一般的なインターフェイスの定義、抽象型基本クラス、またはメッセージを使用するものがあります。Web アプリケーションの場合は、プレゼンテーション レイヤーとビジネス レイヤー間でメッセージ ベースのインターフェイスを使用することを検討します。詳細については、第 5 章「レイヤー型アプリケーションのガイドライン」を参照してください。

設計に関する具体的な問題

設計を策定する際に考慮する必要がある一般的な問題がいくつかあります。これらの問題は、設計の具体的な領域に分類できます。次のセクションでは、最も頻繁にミスが発生する一般的な領域に関するガイドラインを示します。

- 認証
- 承認
- キャッシュ
- 結合
- 例外管理
- ログ記録、監査、およびインストルメンテーション
- 検証

認証

ビジネス レイヤー用の効果的な認証方針を設計することは、アプリケーションのセキュリティと信頼性の両方において重要です。正しく設計しないと、スプーフィング攻撃、辞書攻撃、セッション ハイジャックなどの攻撃に対して、アプリケーションが脆弱になる可能性があります。認証の方針を設計する際には、次のガイドラインを考慮します。

- 信頼境界内の同じ層にあるプレゼンテーション レイヤーまたはサービス レイヤーのみでビジネス レイヤーが使用される場合、ビジネス レイヤーに認証機能を実装しないようにします。最初の呼び出し元の ID ベースの承認または認証が必要な場合のみ、呼び出し元の ID をビジネス レイヤーに渡します。

- ビジネス レイヤーが、複数のアプリケーションの別個のユーザー ストアで使用される場合、シングル サインオンのメカニズムの実装を検討します。また、カスタム認証のメカニズムを使用することは避けて、できる限り組み込みのプラットフォームのメカニズムを使用します。
- プレゼンテーション レイヤーとビジネス レイヤーが同じコンピューターに配置されており、最初の呼び出し元のアクセス制御リスト (ACL) のアクセス許可に基づいてリソースにアクセスする必要がある場合は、偽装の使用を検討します。また、プレゼンテーション レイヤーとビジネス レイヤーが別個のコンピューターに配置されており、最初の呼び出し元の ACL のアクセス許可に基づいてリソースにアクセスする必要がある場合は、デリゲートの使用を検討します。ただし、デリゲートは、多くの環境でサポートされていないので、これを使用するのは、リソースの使用が増加した場合に限ります。セキュリティ要件に違反しなければ、境界でユーザーを認証することと、下位レイヤーの呼び出しに信頼されたサブシステムの手法を使用することを検討します。または、(特にサービス ベースのアプリケーションでは) クレーム ベースのセキュリティ手法を使用することを検討します。この手法を使用すると、統合された ID のメカニズムを利用して、ターゲット システムでユーザーの要求を認証できるようになります。

承認

ビジネス レイヤー用の効果的な承認方針を設計することは、アプリケーションのセキュリティと信頼性の両方において重要です。正しく設計しないと、情報漏えい、データの改ざん、および特権の昇格に対して、アプリケーションが脆弱になる可能性があります。承認の方針を設計する際には、次のガイドラインを考慮します。

- ID、アカウント グループ、役割、またはその他のコンテキスト情報に基づいて、呼び出し元を承認することで、リソースを保護します。役割に関しては、できる限り粒度を小さくして、アクセス許可の組み合わせの数を減らすことを検討します。
- ビジネスにおける決定には役割ベースの承認を使用し、システムの監査にはリソース ベースの承認を使用し、ID、役割、許可、権利、その他の要素などの情報の組み合わせに基づく、統合された承認をサポートする必要がある場合にはクレーム ベースの承認を使用することを検討します。
- パフォーマンスとスケーリングの機会に大きな影響を及ぼす場合があるので、偽装とデリゲートの使用はできる限り避けます。一般的に、直接呼び出すよりも、呼び出し時にクライアントを偽装するほうが負荷が高くなります。
- 同じコンポーネントに承認コードとビジネス処理コードが混在しないようにします。

- 一般的に、アプリケーションでは承認を実施することが普及しているので、承認のインフラストラクチャによるパフォーマンスのオーバーヘッドが大きくならないようにします。

キャッシュ

ビジネス レイヤー用の適切なキャッシュ方針を設計することは、アプリケーションのパフォーマンスと応答性の両方において重要です。キャッシュを使用すると、参照データの検索処理を最適化し、ネットワークのラウンド トリップを回避して、同じ処理が不要に何度も実行されることを回避できます。キャッシュの方針の設計では、作業の一環として、キャッシュ データを読み込むタイミングと方法を判断する必要があります。クライアント側で遅延が発生するのを回避するには、キャッシュ データを非同期で読み込むか、バッチ処理を使用します。キャッシュの方針を設計する際には、次のガイドラインを考慮します。

- ビジネス レイヤーで定期的に再利用される静的データをキャッシュすることを検討します (ただし、揮発性データはキャッシュしないようにします)。また、データベースから迅速かつ効率的に取得できないデータもキャッシュすることを検討します。ただし、処理が遅くなる可能性があるため、大量のデータはキャッシュしないようにし、 必要最低限のデータのみをキャッシュします。
- ビジネス レイヤーに使用できる形式のデータをキャッシュすることを検討します。
- 機密データをキャッシュすることはできる限り避けます。キャッシュする必要がある場合は、キャッシュ内の機密データを保護するメカニズムを設計します。
- Web ファームでの展開が、ビジネス レイヤーのキャッシュ ソリューションの設計に及ぼす影響を考慮します。同じクライアントからの要求を処理できるサーバーがファームに複数台ある場合、キャッシュ ソリューションでは、キャッシュ データの同期をサポートする必要があります。

キャッシュの技法に関する詳細については、第 17 章「分野横断的な懸念事項」を参照してください。

結合

ビジネス レイヤーのコンポーネントを設計する際には、レイヤーが密接に結合されるようにして、レイヤー間の疎結合を実装します。この設計により、アプリケーションのスケーラビリティが向上します。結合を設計する際には、次のガイドラインを考慮します。

- 循環する依存関係を避けます。ビジネス レイヤーでは、下位レイヤー (データ アクセス レイヤー) に関する情報のみが必要であり、上位レイヤー (プレゼンテーション レイヤーやビジネス レイヤーに直接アクセスする外部のアプリケーション) に関する情報は必要ありません。

- 抽象化を使用して、疎結合されたインターフェイスを実装します。インターフェイス コンポーネント、一般的なインターフェイスの定義、または共有の抽象化を使用することで、これを実現できます。それらにおいては、具体的なコンポーネントが、他の具体的なコンポーネントではなく抽象化に依存しています (依存関係の逆転の原則)。詳細については、第 5 章「レイヤー型アプリケーションのガイドライン」で、レイヤー型構造を設計する手順を参照してください。
- 動的な動作で疎結合が必要でない限り、ビジネス レイヤー内は密結合になるように設計します。
- 結合性が高くなるように設計します。コンポーネントには、そのコンポーネントに関連する機能のみを含めます。また、ビジネス コンポーネントでは、データ アクセス ロジックとビジネス ロジックが混在しないようにします。
- メッセージ ベースのインターフェイスを使用してビジネス コンポーネントを公開することを検討し、結合を緩和して、必要な場合は別個の物理層にコンポーネントを配置できるようにします。

例外管理

ビジネス レイヤー用の効果的な例外管理のソリューションを設計することは、アプリケーションのセキュリティと信頼性の両方において重要です。正しく設計しないと、アプリケーションがサービス拒否 (DoS) 攻撃に対して脆弱になり、アプリケーションの機密情報や重要な情報が開示される可能性があります。例外の発生とハンドルは負荷の高い操作なので、例外管理を設計する際には、パフォーマンスへの影響を考慮する必要があります。例外管理の方針を設計する際に考慮する必要があるガイドラインは、次のとおりです。

- 例外は、ハンドルできる内部例外のみをキャッチするか、情報を追加する必要がある場合にキャッチします。たとえば、null 値を変換する際に発生するデータ変換の例外をキャッチします。ビジネス ロジックやアプリケーションのフローを制御するために、例外を使用しないでください。
- 適切な例外の伝達に関する方針を設計します。たとえば、例外は、次のレイヤーに渡す前に、必要に応じてログに記録して変換できる境界のレイヤーに伝播するようにします。エラーとフォールトの根本原因を分析するときに、異なるレイヤーで発生した関連する例外を関連付けられるように、コンテキスト ID を含めることを検討します。
- 他の場所 (グローバル エラー ハンドラー内など) でキャッチされない例外をキャッチして、例外発生後にリソースと状態がクリーンアップされるようにします。
- 重大なエラーと例外のログ記録および通知に関する適切な方針を設計して、例外に関する十分な詳細情報をログに記録し、機密情報は開示しないようにします。

例外管理の技法に関する詳細については、第 17 章「分野横断的な懸念事項」を参照してください。

ログ記録、監査、およびインストルメンテーション

ビジネス レイヤー用のログ記録、監査、およびインストルメンテーションの優れたソリューションを設計することは、アプリケーションのセキュリティと信頼性の両方において重要です。正しく設計しないと、否認の脅威に対してアプリケーションが脆弱になり、悪意のある操作を実行したことをハッカーが否定できるようになります。合法的な処置において行われた不正行為を証明するのに、ログ記録ファイルが必要となる場合もあります。一般的に、監査は、ログ記録情報がリソースへのアクセスと同じタイミングで生成され、リソースにアクセスした同じルーチンで生成された場合に、最も確実だと見なされます。インストルメンテーションは、パフォーマンス カウンターとパフォーマンス イベントを使用して実装できます。システム監視ツールでは、このインストルメンテーションか他のアクセス ポイントを使用して、アプリケーションの状態、パフォーマンス、および正常性についての情報を管理者に提供できます。ログ記録とインストルメンテーションの方針を設計する際には、次のガイドラインを考慮します。

- ビジネス レイヤーのログ記録、監査、およびインストルメンテーションを一元化します。また、patterns & practices の Enterprise Library、Apache Logging Services の log4net、Jarosław Kowalski が提供している NLog などのサード パーティ製のソリューションを使用して、例外のハンドルとログ記録の機能を実装することを検討します。
- ビジネス コンポーネントに、システムクリティカルなイベントとビジネスクリティカルなイベントのインストルメンテーションを含めます。
- ログ ファイルには、ビジネスに関する機密情報を格納しないようにします。
- ログ記録でエラーが発生しても、ビジネス レイヤーの正常機能に影響しないようにします。
- ビジネス レイヤーに実装する機能へのすべてのアクセスを監査およびログ記録することを検討します。

検証

ビジネス レイヤー用の効果的な検証のソリューションを設計することは、アプリケーションのユーザビリティと信頼性の両方において重要です。正しく設計しないと、アプリケーションがデータの不整合やビジネス ルールの違反に対して脆弱になり、ユーザー エクスペリエンスが低下します。また、クロス サイト スクリプト攻撃や、SQL インジェクション攻撃、バッファのオーバーフロー、その他の種類の入力による攻撃などのセキュリティの問題に対してアプリケーションが脆弱になる場合があります。有効な入力と悪意のある入力に関する包括的な定義はありません。

また、悪用されるリスクは、アプリケーションで入力をどのように使用するかに左右されます。検証の方針を設計する際には、次のガイドラインを考慮します。

- 入力の検証がプレゼンテーション レイヤーで行われる場合も、ビジネス レイヤーでは、すべての入力とメソッドのパラメーターを検証します。
- 検証の手法を一元化して、テスト容易性と再利用性を最大限に高めます。
- ユーザーの入力を制限、拒否、および一部削除します。つまり、すべてのユーザーの入力が悪意のある入力だと仮定します。また、入力されたデータの長さ、範囲、形式、型を検証します。

配置に関する考慮事項

ビジネス レイヤーを配置する際には、運用環境のパフォーマンスとセキュリティ上の問題を考慮する必要があります。ビジネス レイヤーを配置する際には、次のガイドラインを考慮します。

- スケーラビリティやセキュリティの要件に違反しなければ、アプリケーションのパフォーマンスを最大限に高めるために、ビジネス レイヤーをプレゼンテーション レイヤーと同じ物理層に配置することを検討します。
- リモート ビジネス レイヤーをサポートする必要がある場合は、TCP プロトコルを使用して、アプリケーションのパフォーマンスを向上することを検討します。
- インターネット プロトコル セキュリティ (IPSec) を使用して、物理層間で渡されるデータを保護することを検討します。
- Secure Sockets Layer (SSL) 暗号化を使用して、ビジネス レイヤーのコンポーネントで行うリモート Web サービスの呼び出しを保護します。

ビジネス レイヤーの設計手順

ビジネス レイヤーを設計する際には、ビジネス コンポーネント、ビジネス エンティティ、ビジネス ワークフロー コンポーネントなどの、レイヤーの主な構成の設計要件も考慮する必要があります。このセクションでは、ビジネス レイヤー自体を設計する際に必要となる作業について簡単に説明します。ビジネス レイヤーを設計する際には、次の主な手順を実行します。

1. **ビジネス レイヤーの大まかな設計を作成する**: プレゼンテーション レイヤー、サービス レイヤー、その他のアプリケーションなどのビジネス レイヤーのコンシューマーを特定します。これを特定すると、ビジネス レイヤーを公開する方法を決定するのに役立ちます。次に、ビジネス レイヤーのセキュリティ要件、検証の要件、検証の方針を決定します。この章の前半の「設計に関する具体的な問題」のセクションで紹介したガイドラインを使用して、大まかな設計を作成する際には、関連するすべての要素を考慮するようにします。
2. **ビジネス コンポーネントを設計する**: アプリケーションを設計および導入する際に使用できるビジネス コンポーネントには、ビジネス プロセス コンポーネント、ユーティリティ コンポーネント、ヘルパー コンポーネントなど、複数の種類があります。アプリケーションの設計、トランザクションの要件、および処理の規則のさまざまな側面が、ビジネス コンポーネントで採用する設計に影響を及ぼします。詳細については、第 12 章「ビジネス レイヤーのコンポーネントの設計」を参照してください。
3. **ビジネス エンティティ コンポーネントを設計する**: ビジネス エンティティには、アプリケーションで使用するビジネス データが含まれており、それを管理するのに使用します。また、ビジネス エンティティでは、エンティティに含まれるデータの検証を行う必要があります。さらに、そのデータにアクセスして、データの初期化に使用するプロパティと操作を提供する必要もあります。詳細については、第 13 章「ビジネス エンティティの設計」を参照してください。
4. **ビジネス ワークフロー コンポーネントを設計する**: タスクを規定の順序で完了する必要があるシナリオ (各手順が完了してから次の手順に進む必要があります)、またはユーザーによる調整が必要なシナリオが多数あります。これらの要件は、主要なワークフロー シナリオにマップできます。また、要件と規則が、ワークフロー コンポーネントを実装するオプションに、どのような影響を及ぼすかを理解する必要があります。詳細については、第 14 章「ワークフロー コンポーネントの設計」を参照してください。

アプリケーションでコンポーネントを設計および使用する方法の詳細については、第 10 章「コンポーネントのガイドライン」を参照してください。

関連する設計パターン

次の表に示すように、主要なパターンはカテゴリに分類されます。各カテゴリの設計を行う際には、これらのパターンの使用を検討します。

カテゴリ	関連するパターン
ビジネス コンポーネント	**Application Façade**: 動作を一元化および集約して均一のサービス レイヤーを提供します。 **Chain of Responsibility**: 複数のオブジェクトで要求を処理できるようにして、要求の送信者と受信者が一対一で結合されないようにします。 **Command**: 要求処理を、共通の実行インターフェイスを備えた別個のコマンド オブジェクトにカプセル化します。
ビジネス エンティティ	**Domain Model**: ドメイン内のエンティティとエンティティ間の関係を表す一連のビジネス オブジェクトです。 **Entity Translator**: 要求ではデータ コントラクトをビジネス エンティティに変換し、応答では逆の変換を行うオブジェクトです。 **Table Module**: データベースのテーブルまたはビュー内のすべての行のビジネス ロジックを処理する、1 つのコンポーネントです。
ワークフロー	**Data-Driven Workflow**: ワークフローまたはシステムのデータの値に基づいてシーケンスが決定されるタスクを含むワークフローです。 **Human Workflow**: ユーザーが手動で実行するタスクを含むワークフローです。 **Sequential Workflow**: 先行するタスクが完了してから次のタスクが開始されるというシーケンスに従うタスクを含むワークフローです。 **State-Driven Workflow**: システムの状態によってシーケンスが決定されるタスクを含むワークフローです。

Façade パターンの詳細については、Erich Gamma、Richard Helm、Ralph Johnson、John Vlissides 共著『オブジェクト指向における再利用のためのデザイン パターン』(ソフトバンク クリエイティブ、1999 年) の第 4 章「構造に関するパターン」を参照してください。

Chain of Responsibility パターンの詳細については、「開放/閉鎖原則」(http://msdn.microsoft.com/ja-jp/magazine/cc546578.aspx) を参照してください。

Command パターンの詳細については、Erich Gamma、Richard Helm、Ralph Johnson、John Vlissides 共著『オブジェクト指向における再利用のためのデザイン パターン』(ソフトバンク クリエイティブ、1999 年) の第 5 章「振る舞いに関するパターン」を参照してください。

Entity Translator パターンの詳細については、「Useful Patterns for Services」(http://msdn.microsoft.com/en-us/library/cc304800.aspx、英語) を参照してください。
データ ドリブン ワークフロー、ヒューマン ワークフロー、シーケンシャル ワークフロー、およびステート ドリブン ワークフローの詳細については、「Windows Workflow Foundation の概要」(http://msdn.microsoft.com/ja-jp/library/ms734631.aspx) と「Workflow Patterns」(http://www.workflowpatterns.com/、英語) を参照してください。

patterns & practices のサービス

マイクロソフトの patterns & practices グループが提供している関連サービスの詳細については、次のリソースを参照してください。

- Enterprise Library (http://msdn.microsoft.com/en-us/library/cc467894.aspx、英語)
- Unity Application Block (依存関係の挿入のメカニズム) (http://msdn.microsoft.com/en-us/library/dd203101.aspx、英語)

関連情報

Web リソースに簡単にアクセスするには、http://www.microsoft.com/architectureguide (英語) でオンライン版の参考文献を参照してください。
ビジネス レイヤーの統合に関する詳細については、「Integration Patterns」(http://msdn.microsoft.com/en-us/library/ms978729.aspx、英語) を参照してください。
Apache Logging Services の log4net の詳細については、http://logging.apache.org/log4net/index.html (英語) を参照してください。
Jarosław Kowalski が提供している NLog の詳細については、http://www.nlog-project.org/introduction.html (英語) を参照してください。

8

データ レイヤーのガイドライン

概要

この章では、アプリケーションのデータ レイヤーの設計に関する主要なガイドラインを示します。このガイドラインは、一般的なレイヤー型アプリケーション アーキテクチャにおけるデータ レイヤーの位置付け、データ レイヤーに通常含まれるコンポーネント、およびデータ レイヤーの設計時に発生する主要な問題について理解するのに役立ちます。また、設計、推奨される設計手順、関連する設計パターン、およびテクノロジの選択肢に関するガイドラインも紹介します。図 1 は、一般的なアプリケーション アーキテクチャにおけるデータ レイヤーの位置付けを示しています。

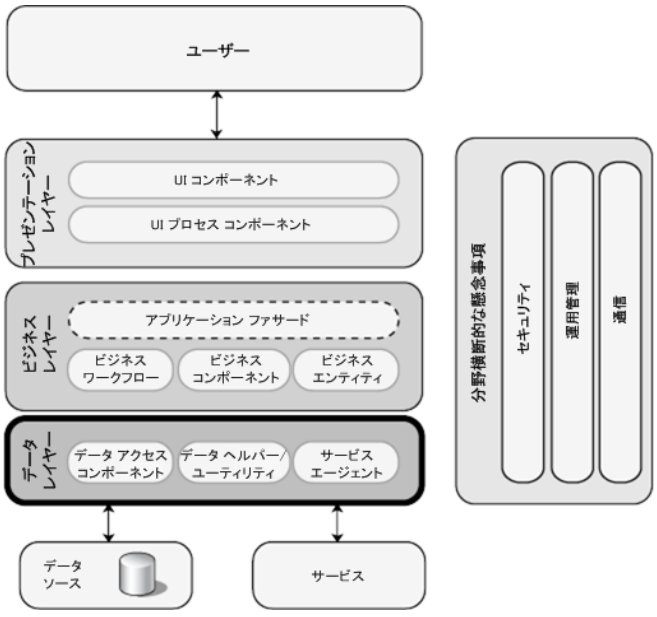


図 9

一般的なアプリケーションのデータ レイヤーとそこに含まれる場合があるコンポーネント

データ レイヤーには、次のコンポーネントが含まれる場合があります。

- **データ アクセス コンポーネント**: このコンポーネントは、基になるデータ ストアにアクセスするために必要なロジックを抽象化します。また、アプリケーションの構成と保守をより簡単に行えるようにするために、共通のデータ アクセス機能を一元化します。開発者が共通のデータ アクセス ロジックを特定して、独立した再利用可能なヘルパーやユーティリティ データ アクセス コンポーネントに実装する必要があるデータ アクセス フレームワークもあれば、 このようなコンポーネントを自動的に実装し、開発者が記述しなければならないデータ アクセス コードの量を削減するデータ アクセス フレームワークもあります (多くのオブジェクト/リレーショナル マッピング (O/RM) フレームワークなどは、これに該当します)。
- **サービス エージェント**: ビジネス コンポーネントが外部のサービスで提供されるデータにアクセスする必要がある場合、そのサービスとの通信を管理するためのコードを実装しなければならないことがあります。サービス エージェントは、アプリケーションでサービスを呼び出すためのさまざまな要求を分離するデータ アクセス コンポーネントを実装しています。また、キャッシュ、オフライン サポート、サービスが公開するデータの形式とアプリケーションで必要な形式との基本的なマッピングなどの追加サービスを提供する場合もあります。

データ レイヤーで一般的に使用されるコンポーネントの詳細については、第 10 章「コンポーネントのガイドライン」を参照してください。データ アクセス コンポーネントの作成に関する詳細については、第 15 章「データ コンポーネントの設計」を参照してください。

設計に関する一般的な考慮事項

データ アクセス レイヤーは、アプリケーションの要件を満たし、効率的かつ安全に機能し、ビジネス要件の変化に応じて容易に保守および拡張できる必要があります。データ レイヤーを設計する際には、次の一般的な設計ガイドラインを考慮します。

- **適切なデータ アクセス テクノロジを選択する**: 選択するデータ アクセス テクノロジは、処理する必要があるデータの種類と、アプリケーションでデータをどのように操作するのかによって決まります。適切なテクノロジは、シナリオによって異なります。このガイドの最後にある付録 C「データ アクセス テクノロジ」では、こうした選択肢を提示し、それぞれのデータ アクセス テクノロジのメリットと問題点を列挙しています。

- **抽象化を使用して、データ アクセス レイヤーへの疎結合されたインターフェイスを実装する**: これは、インターフェイス コンポーネント (レイヤーのコンポーネントで理解できる形式に要求を変換する既知の入出力を設定したゲートウェイなど) を定義することで実現できます。また、インターフェイス型や抽象基本クラスを使用して、インターフェイス コンポーネントで実装する必要がある共有の抽象化を定義することもできます。レイヤーの抽象化に関する詳細については、第 5 章「レイヤー型アプリケーションのガイドライン」を参照してください。
- **データ アクセス機能をデータ アクセス レイヤー内にカプセル化する**: データ アクセス レイヤーでは、データ ソース アクセスの詳細を隠ぺいする必要があります。また、接続の管理、クエリの生成、およびアプリケーション エンティティのデータ ソース構造へのマッピングを行う必要もあります。データ アクセス レイヤーのコンシューマーは、カスタム オブジェクト、TypedDataSet、XML などのアプリケーション エンティティを使用して抽象インターフェイスを通じて通信します。ただし、コンシューマーは、データ アクセス レイヤーの内部の詳細を把握しないようにする必要があります。このようにして問題を分離すると、アプリケーションの開発と保守に役立ちます。
- **アプリケーション エンティティをデータ ソース構造にマップする方法を決定する**: アプリケーションで使用するエンティティの種類は、エンティティをデータ ソース構造にマップする方法を決定するうえで主要な要因です。一般的な設計手法では、Domain Model パターンや Table Module パターン、またはオブジェクト/リレーショナル マッピング (O/RM) フレームワークを使用したりしますが、その他の形式を使用してビジネス エンティティを実装することもできます。データ ソースから読み込んだデータをビジネス エンティティやデータ構造に設定し、アプリケーションのビジネス レイヤーやプレゼンテーション レイヤーでこうしたビジネス エンティティやデータ構造を使用できるようにするための方針を特定する必要があります。Domain Model パターンや Table Module パターンの詳細については、この章の後半にある「関連する設計パターン」を参照してください。ビジネス エンティティとデータ形式の詳細については、第 13 章「ビジネス エンティティの設計」を参照してください。
- **データ構造の統合を検討する**: サービスを通じてデータを公開する場合は、データを統合された構造にまとめるためにデータ転送オブジェクト (DTO) を使用することを検討します。また、DTO は、粒度の粗い操作を促進する一方で、異なる境界レイヤー間でデータを移動するための構造を提供します。集計操作においては、DTO では複数のビジネス エンティティのデータを保持することができます。Table Data Gateway パターンや Active Record パターンを使用する場合は、データの表現に DataTable の使用を検討することをお勧めします。
- **接続の管理方法を決定する**: 一般に、データ アクセス レイヤーでは、アプリケーションで必要な、すべてのデータ ソースへの接続を作成して管理する必要があります。組織のセキュリティ要件に従うた

めに、接続情報を格納および保護するための適切な方法を選択する必要があります。これには、構成ファイルのセクションを暗号化する、構成情報の格納先をサーバーに限定するなどの方法が考えられます。詳細については、第 15 章「データ コンポーネントの設計」を参照してください。

- **データ例外のハンドル方法を決定する**: データ アクセス レイヤーでは、データ ソースや CRUD (作成、読み取り、更新、および削除) 操作に関連するすべての例外をキャッチし、(少なくとも最初に) ハンドルする必要があります。データ自体に関する例外、データ ソース アクセス エラー、およびタイムアウト エラーは、このレイヤーでハンドルし、エラーがアプリケーションの応答性や機能に影響を及ぼす場合にのみ他のレイヤーに渡す必要があります。
- **セキュリティ リスクを考慮する**: データ アクセス レイヤーでは、データを盗んだり破損させたりする攻撃を防ぎ、データ ソースへのアクセスに使用されるメカニズムを保護する必要があります。たとえば、データ ソースの情報が公開されないようにエラーや例外の情報の一部を削除したり、最小限の特権を持つアカウントを使用して、アプリケーションが要求する操作を実行するのに必要な特権のみを与えるようにします。データ ソース自体で特権を制限できる場合でも、データ ソースだけでなくデータ アクセス レイヤーにもセキュリティを実装する必要があります。SQL インジェクション攻撃を成功させないために、データベースには、パラメーター化されたクエリを通じてアクセスする必要があります。ユーザー入力データから動的クエリを作成するために、文字列の連結を使用しないでください。
- **ラウンド トリップの回数を減らす**: 複数のコマンドを 1 つのデータベース操作にまとめることを検討します。
- **パフォーマンスとスケーラビリティの目標を考慮する**: 設計時には、データ アクセス レイヤーのスケーラビリティとパフォーマンスの目標を考慮に入れる必要があります。たとえば、インターネット ベースの商取引アプリケーションを設計する場合は、データ レイヤーのパフォーマンスがアプリケーションのボトルネックになる可能性があります。データ レイヤーのパフォーマンスが重要な場合は、プロファイル機能を使用して、負荷の高いデータ操作を理解し、そのような操作を削減または取り除きます。

設計に関する具体的な問題

設計を策定する際に考慮する必要がある一般的な問題がいくつかあります。これらの問題は、設計の具体的な領域に分類できます。次のセクションでは、最も頻繁にミスが発生する一般的な領域に関するガイドラインを示します。

- バッチ処理

- バイナリ ラージ オブジェクト
- 接続
- データ形式
- 例外管理
- オブジェクト リレーショナル マッピング
- クエリ
- ストアド プロシージャ
- ストアド プロシージャと動的 SQL のどちらを使用するか
- トランザクション
- 検証
- XML

バッチ処理

データベース コマンドをバッチ処理すると、データ レイヤーのパフォーマンスが向上する場合があります。データベース実行環境への要求はオーバーヘッドを伴います。バッチ処理を行うと、スループットが向上し待ち時間が短くなるため、全体的なオーバーヘッドが削減される場合があります。データベースでは、類似するクエリのクエリ実行プランをキャッシュして、それを再利用できるので、類似するクエリをバッチ処理すると、パフォーマンスが向上する場合があります。バッチ処理を設計する際には、次のガイドラインを考慮します。

- データベースへのラウンド トリップの回数を減らして、ネットワーク トラフィックを最小限に抑えるために、バッチ コマンドを使用することを検討します。ただし、最大限のメリットを得るには、類似するクエリのみをバッチ処理するようにします。類似していないクエリや任意のクエリをバッチ処理しても、類似するクエリをバッチ処理した場合と同じレベルのオーバーヘッドの削減は得られません。
- 複数のデータ セットを読み込んだり、コピーしたりする場合は、バッチ コマンドと DataReader を使用すること検討します。ただし、大量のファイル ベースのデータをデータベースに読み込む場合は、バッチ コマンドと DataReader ではなく、データベース一括コピー ユーティリティを使用することを検討します。
- データベース リソースをロックする実行時間の長いバッチ コマンドでトランザクションを実行することは避けます。

バイナリ ラージ オブジェクト

データを単一のストリームとして格納および取得する場合、そのデータはバイナリ ラージ オブジェクト (BLOB) だと考えることができます。BLOB 自体が構造を持っている場合もありますが、この構造は、BLOB の格納先であるデータベースや、データベースの読み取りや書き込みを行うデータ レイヤーには公開されません。データベースには、BLOB データまたは BLOB データへのポインターを格納できます。BLOB データは、データベースに直接格納されない場合は、通常、ファイル システムに格納されます。BLOB は一般に画像データの格納に使用しますが、オブジェクトのバイナリ表現を格納するためにも使用できます。BLOB に関する設計を行う際には、次のガイドラインを考慮します。

- BLOB データをデータベースに格納する必要があるかどうかを検討します。最新のデータベースは以前と比べて BLOB データの処理が大幅に強化されており、列に適切なデータ型を選択できます。また、関連するメタデータの保守容易性、バージョン管理、操作、および格納先が提供されます。ただし、BLOB データをディスクに格納して、データベースにはデータへのリンクのみを格納する方が実用的かどうかについても検討します。
- サーバー間での大きなバイナリ オブジェクトの同期を簡略化するために、BLOB を使用することを検討します。
- BLOB データを検索する必要があるかどうかを検討します。検索する必要がある場合は、BLOB データを解析するのではなく、他の検索可能なデータベース フィールドを作成し、作成したフィールドにデータを設定します。
- BLOB を取得する場合は、ビジネス レイヤー内またはプレゼンテーション レイヤー内で操作できるように BLOB を適切な型にキャストします。

接続

データ ソースへの接続は、データ レイヤーの重要な部分です。すべてのデータ ソース接続は、データ レイヤーで管理する必要があります。接続の管理と作成には、データ レイヤーとデータ ソースの両方の貴重なリソースを使用します。パフォーマンスとセキュリティを最大限に高めるために、データ レイヤーの接続に関する設計を行う際には、次のガイドラインを考慮します。

- 一般に、接続はできるだけ後で開き、できるだけ早く閉じます。接続を必要以上に長い時間開いたままにすることは避けます。
- できる限り、1 つの接続でトランザクションを実行します。

- "信頼されたサブシステム" セキュリティ モデルを使用し、偽装や個々の ID の使用をできるだけ避けることにより、接続プールを活用します。
- セキュリティ上の理由から、接続情報の格納にシステム/ユーザー データ ソース名 (DSN) を使用することは避けます。
- データ ソースへの接続が失われたり、タイムアウトしたりした場合に対処するための再試行ロジックを設計する必要があるかどうかを検討します。ただし、根本的な原因がリソースの競合のような問題にある場合は、操作を再試行すると、問題が悪化し、スケーリングの問題に発展することがあります。詳細については、第 15 章「データ コンポーネントの設計」を参照してください。

データ形式

適切なデータ形式を選択すると、他のアプリケーションとの相互運用性がもたらされ、異なるプロセスや物理コンピューターの間でのシリアル化された通信が容易になります。データ形式とシリアル化は、ビジネス レイヤーでアプリケーションの状態を格納および取得できるようにするうえでも重要です。データ形式を設計する際には、次のガイドラインを考慮します。

- 他のシステムやプラットフォームとの相互運用性が必要な場合または時間と共に変化する可能性のあるデータ構造を処理する場合は、XML の使用を検討します。
- 単純な CRUD ベースのアプリケーションでの非接続型のシナリオでは、DataSet を使用することを検討します。
- 物理的な境界を越えてデータを転送する必要がある場合は、シリアル化と相互運用性の要件を検討します。たとえば、カスタム ビジネス オブジェクトをシリアル化する方法、こうしたオブジェクトをデータ転送オブジェクト (DTO) に変換する必要がある場合に変換する方法、およびデータを受け取るレイヤーが許容できる形式を検討します。

データ形式の詳細については、第 15 章「データ コンポーネントの設計」を参照してください。アプリケーションでコンポーネントを設計および使用する方法の詳細については、第 10 章「コンポーネントのガイドライン」を参照してください。

例外管理

データ レイヤーで例外が一貫した方法でキャッチおよびスローされるように、一元化された例外管理の方針を設計します。可能であれば、例外ハンドル ロジックは、アプリケーションの分野横断的な懸念事項を実装するコンポーネントに集中して実装します。信頼境界を越えて伝播する例外、および他のレイヤーや層に伝播する例外には、特に注意する必要があります。ハンドルされていない例外が原因でアプリケーションの信頼性の問題が発生したり、機密のアプリケーション情報が公開されたりすることがないように、ハンドルされていない例外を考慮した設計を行います。例外管理の方針を設計する際には、次のガイドラインを考慮します。

- データ アクセス レイヤーでキャッチおよびハンドルする必要がある例外を特定します。たとえば、デッドロックと接続の問題は、多くの場合、データ レイヤー内で解決できます。しかし、同時実行違反など、解決するにはユーザーへの通知が必要な例外もあります。
- 適切な例外の伝達に関する方針を設計します。たとえば、例外は、次のレイヤーに渡す前に、必要に応じてログに記録して変換できる境界のレイヤーに伝播するようにします。エラーとフォールトの根本原因を分析するときに、異なるレイヤーで発生した関連する例外を関連付けられるように、コンテキスト ID を含めることを検討します。
- データ ソース エラーやタイムアウトが発生した操作の再試行処理を実装することを検討します (ただし、操作を再試行しても問題ない場合に限ります)。
- 他の場所 (グローバル エラー ハンドラー内など) でキャッチされない例外をキャッチして、例外発生後にリソースと状態がクリーンアップされるようにします。
- 重大なエラーと例外のログ記録および通知に関する適切な方針を設計して、例外に関する十分な詳細情報をログに記録し、機密情報は開示しないようにします。

オブジェクト リレーショナル マッピング

オブジェクト指向 (OO) アプリケーションを設計する場合は、OO モデルとリレーショナル モデルの間のインピーダンス不整合と、この 2 つのモデル間の変換を困難にする可能性のある要因を考慮します。たとえば、OO 設計におけるカプセル化 (フィールドの隠ぺい) は、データベースにおけるプロパティのパブリックな特性と矛盾する場合があります。その他のインピーダンス不整合には、データ型の違い、構造の違い、トランザクションの違い、データの操作方法の違いなどがあります。このような不整合を処理する一般的な方法は、データ アクセス用の設計パターン (Repository など) とオブジェクト/リレーショナル マッピング (O/RM) ツールを使用することです。多くの場合、ドメイン駆動設計手法 (ドメイン内のオブジェクトに基づくエンティティのモデリングをベースとした手法) が適切な選択肢です。ドメイン駆動設計の詳細については、第 3 章「アーキテクチャのパターンとスタイル」と第 13 章「ビジネス エンティティの設計」を参照してください。

オブジェクト リレーショナル マッピングに関する設計を行う際には、次のガイドラインを考慮します。

- ドメイン エンティティとデータベースの間にオブジェクト/リレーショナル マッピング (O/RM) レイヤーを提供するフレームワークを使用することを検討します。これには、必要なカスタム コードの量を大幅に削減できる最新の O/RM ソリューションを使用できます。
- データベース スキーマを完全に制御できる新しい環境で作業する場合は、O/RM ツールを使用して、定義したオブジェクト モデルをサポートするスキーマを生成し、データベースとドメイン エンティティの間のマッピングを提供できます。
- 既存のデータベース スキーマを使用しなければならない既存の環境で作業する場合は、O/RM ツールをドメイン モデルと既存のリレーショナル モデルの間のマッピングに使用できます。
- 小規模なアプリケーションを設計する場合や O/RM ツールを持っていない場合は、Repository などの一般的なデータ アクセス パターンを実装します。Repository パターンでは、リポジトリ オブジェクトを使用して、ドメイン エンティティをメモリ内にあるかのように処理することができます。
- Web アプリケーションやサービスを設計する場合は、エンティティをグループ化し、必要なデータのみを使用してドメイン エンティティを部分的に読み込むオプション (通常、この処理は遅延読み込みと呼ばれます) をサポートします。これにより、アプリケーションでは、ステートレスな操作をサポートするために必要なより高い負荷を処理したり、初期化されたドメイン モデルをメモリ内のユーザーごとに保持することを回避して、リソースの使用を制限したりすることができます。

クエリ

クエリは、データ レイヤーにおける主要なデータ操作であり、 アプリケーションからの要求をデータベースに対する CRUD 操作に変換するメカニズムです。クエリは非常に重要なので、データベースのパフォーマンスとスループットを最大限に高めるように最適化する必要があります。データ レイヤーでクエリを使用する場合は、次のガイドラインを考慮します。

- セキュリティの問題を軽減し SQL インジェクション攻撃が成功する可能性を減らすために、パラメーター化された SQL クエリと型指定されたパラメーターを使用します。ユーザー入力データから動的クエリを作成するために、文字列の連結を使用しないでください。
- オブジェクトを使用してクエリを作成することを検討します。たとえば、Query Object パターンを実装するか、ADO.NET によって提供されるパラメーター化されたクエリのサポートを使用します。データベース内のデータ スキーマをクエリ実行用に最適化することも検討します。

- 動的 SQL クエリを作成する場合は、ビジネス処理ロジックが SQL ステートメントの生成に使用するロジックと混在しないようにします。これらのロジックが混在すると、保守やデバッグが非常に困難なコードになる場合があります。

ストアド プロシージャ

以前は、ストアド プロシージャの方が動的 SQL ステートメントよりもパフォーマンスが優れていました。しかし、最新のデータベース エンジンを使用すると、ストアド プロシージャと (パラメーター化されたクエリを使用した) 動的 SQL ステートメントのパフォーマンスは一般にほぼ同じです。ストアド プロシージャの使用を検討する場合は、抽象化、保守容易性、および環境について考慮する必要があります。このセクションでは、ストアド プロシージャを使用するアプリケーションの設計に役立つガイドラインを提供します。ストアド プロシージャと動的 SQL ステートメントのどちらを使用するかを判断するためのガイダンスについては、次のセクションを参照してください。
ストアド プロシージャのセキュリティとパフォーマンスの観点から見た主要なガイドラインは、型指定されたパラメーターを使用することと、ストアド プロシージャ内では動的 SQL を使用しないことです。パラメーターの使用は、クエリ プランをゼロから再構築するのではなくキャッシュされたクエリ プランを使用する要因の 1 つです。パラメーターの型や数が変化すると、新しいクエリ実行プランが生成されますが、パフォーマンスが低下する場合があります。ストアド プロシージャを設計する際には、次のガイドラインを考慮します。

- プロシージャへの入力パラメーターと単一の値を返す出力パラメーターに、型指定されたパラメーターを使用します。一覧や表形式のデータを渡すために、XML パラメーターやテーブル値パラメーターを使用することを検討します。ストアド プロシージャ内で表示用にデータを書式設定するのではなく、適切な型を返して、プレゼンテーション レイヤーで書式設定を実行します。
- ストアド プロシージャ内で動的 SQL を生成する必要がある場合は、パラメーター変数やデータベース変数を使用します。ただし、ストアド プロシージャ内で動的 SQL を使用するとパフォーマンス、セキュリティ、および保守容易性に影響する場合があることを覚えておいてください。
- データの処理中に一時テーブルを作成することは避けます。一時テーブルを使用する必要がある場合は、ディスク上ではなくメモリ内に作成することを検討します。
- 適切なエラー ハンドル設計を実装し、アプリケーション コードでハンドルできるエラーを返します。

ストアド プロシージャと動的 SQL のどちらを使用するか

ストアド プロシージャと動的 SQL のどちらを使用するかを判断する際の主な焦点は、データベースのストアド プロシージャに実装された SQL を使用するか、コードで動的に生成された SQL ステートメントを使用するかです。ストアド プロシージャと動的 SQL のどちらを使用するかを判断する際には、抽象化の要件、保守容易性、および環境の制約を考慮する必要があります。また、多くの場合、どちらを使用するかは、開発者の好みやスキルにも左右されます。

ストアド プロシージャの主なメリットは、データベースに抽象化レイヤーが提供されることです。これにより、データベース スキーマを変更した場合のアプリケーション コードへの影響を最小限に抑えられます。また、ストアド プロシージャの方が動的 SQL よりもセキュリティの実装と管理が容易です。というのも、ストアド プロシージャを使用すると、ストアド プロシージャ以外のすべてへのアクセスを制限したり、ほとんどのデータベースでサポートされているきめ細かいセキュリティ機能を利用したりすることができるからです (ただし、セキュリティ機能により、接続プールの利用が制限される可能性があることに注意してください)。

動的 SQL ステートメントの主なメリットは、ストアド プロシージャよりも柔軟であることが多く、より迅速に開発を進められることです。多くのオブジェクト/リレーショナル マッピング (O/RM) フレームワークでは、動的クエリが自動的に生成されるので、開発者が記述する必要のあるコードの量が大幅に削減されます。

ストアド プロシージャと動的 SQL クエリのどちらを使用するかを判断する際には、次のガイドラインを考慮します。

- 1 つのクライアントと少数のビジネス ルールを使用する小規模なアプリケーションでは、多くの場合、動的 SQL を使用するのが最適です。
- 複数のクライアントを使用する大規模なアプリケーションの場合は、必要な抽象化を実現する方法を検討します。その抽象化を実装する場所を決定します。どこに実装するか (ストアド プロシージャという形でデータベースに実装するか、データ アクセス パターンまたは O/RM 製品という形でアプリケーションのデータ レイヤーに実装するか) を決定します。
- 大量のデータを扱う操作に関しては、ストアド プロシージャを使用した方が、よりデータの近くで操作を実行できるので、パフォーマンスが向上する場合があります。
- データベース スキーマを変更した場合のアプリケーション コードへの影響を最小限に抑えるには、ストアド プロシージャを使用してデータベースへのアクセスを提供することを検討することをお勧めします。これは、スキーマを正規化または最適化する際に、アプリケーション コードへの変更を分離して最小限に抑えるのに役立ちます。ストアド プロシージャの入出力への変更はアプリケーション コードに影響を及ぼすことがありますが、多くの場合、このような変更はストアド プロシージャにアクセスする特定のコンポーネントに分離できます。オブジェクト/リレーショナル マッピング (O/RM) フレ

ームワークも、スキーマが更新された場合にアプリケーション コードの変更を分離して最小限に抑えるのに役立ちます。

- 動的 SQL クエリの使用を検討する場合は、データベース スキーマへの変更がアプリケーションに及ぼす影響を理解する必要があります。そのため、ビジネス レイヤーは、ビジネス コンポーネントをデータベース クエリの実行から分離するように実装する必要があります。Query Object や Repository などのいくつかのパターンを使用して、この分離を実現できます。オブジェクト/リレーショナル マッピング (O/RM) フレームワークは、ビジネス コンポーネントとデータベース クエリの実行を完全に分離するのに役立ちます。
- アプリケーションの開発を行うチームを考慮します。チーム メンバーがデータベース プログラミングに馴染みがない場合は、開発メンバーに馴染みのあるツールやパターンの使用を検討します。
- デバッグのサポートを考慮します。動的 SQL の方が、アプリケーション開発者にとってデバッグを行うのが容易です。

トランザクション

トランザクションとは、要求に対応して、データベースの整合性を確保するために 1 つの atomic 単位として処理される一連の情報と関連する操作のやり取りです。トランザクションは、すべての情報と操作の処理が完了して、データベースへの変更が確定した場合にのみ完了したと見なされます。トランザクションでは、エラーが発生した場合に元に戻す (ロールバックする) というデータベース操作をサポートしています。これは、データベース内のデータの整合性を維持するのに役立ちます。

適切な同時実行モデルを特定して、トランザクションの管理方法を決定することは重要です。同時実行に関しては、オプティミスティック モデルとペシミスティック モデルのどちらかを選択できます。オプティミスティック同時実行制御では、データにロックは適用されず、データを更新するには、データが最後に取得したときから変更されていないことをコードでチェックする必要があります。通常、このチェックはタイムスタンプに基づいて行います。ペシミスティック同時実行制御では、データはロックされ、ロックが解放されるまで別の操作でデータを更新することはできません。

トランザクションを設計する際には、次のガイドラインを考慮します。

- 再試行と構成が可能になるトランザクション境界を検討し、トランザクションは必要な場合にのみ有効にします。単純なクエリには明示的なトランザクションは必要ない場合がありますが、データベースの既定のトランザクション コミット動作と分離レベル動作を把握するようにします。既定では、

Microsoft SQL Server® データベースでは、SQL ステートメントが別個のトランザクションとして実行されます (オートコミット トランザクション モード)。

- ロックが適用される時間をできる限り短くするために、トランザクションはできるだけ短くします。実行時間の長いトランザクションや共有データへのアクセスでロックを使用するのは避けます。共有データへのアクセス中にデータをロックすると、他のコードによるデータへのアクセスをブロックする可能性があります。排他的ロックを使用すると競合やデッドロックが発生する可能性があるので、排他的ロックの使用は避けます。
- データへの変更が他の操作で使用するデータに反映されるタイミングと方法を定義する適切な分離レベルを使用します。データの一貫性と競合との間にはトレードオフが存在します。分離レベルを高くすると、データの一貫性は高まりますが、全体的な同時実行性は低下します。分離レベルを低くすると、競合が減ることによりパフォーマンスは向上しますが、一貫性は低下します。
- System.Transactions 名前空間のクラスを使用している場合は、System.Transactions 名前空間の TransactionScope オブジェクトで提供される暗黙モデルの使用を検討します。暗黙のトランザクションは手動 (明示的な) トランザクションほど高速ではありませんが、簡単に作成できます。また、暗黙のトランザクションを使用すると、柔軟で保守が容易な中間層ソリューションを実現できます。手動トランザクションを使用する場合は、ストアド プロシージャ内にトランザクションを実装することを検討します。
- コミットやロールバックを使用できない場合や実行時間の長いトランザクションを使用する場合は、トランザクションの操作が失敗した場合にデータ ストアを前の状態に戻すための補正のための方法を実装します。
- データベースに対して複数のクエリを実行する必要がある場合は、複数のアクティブな結果セット (MARS) の使用を検討します。MARS を使用すると、複数の順方向専用かつ読み取り専用の結果セットのサポートが提供され、同じ接続を使用して複数のクエリを実行できるようになります。MARS は、大量のトランザクションを使用する同時実行アプリケーションで役立つ場合があります。

検証

入力とデータの検証に関する効果的な方針を設計することは、アプリケーションのセキュリティを確保するうえで重要です。他のレイヤーやサード パーティ製のコンポーネントから受け取ったデータ、およびデータベースやデータ ストアから受け取ったデータの検証規則を決定します。信頼境界を越えるデータを検証できるように、信頼境界を理解します。検証の方針を設計する際には、次のガイドラインを考慮します。

- データ レイヤーが呼び出し元から受け取ったすべてのデータを検証します。NULL 値を適切に処理し、無効な文字を除外します。
- 検証を設計する際には、データの用途を考慮します。たとえば、動的 SQL の作成に使用するユーザー入力については、SQL インジェクション攻撃に使用される文字やパターンが含まれていないかどうかを確認する必要があります。
- 検証に失敗した場合は、必要な情報を提供するエラー メッセージを返します。

検証の技法に関する詳細については、第 17 章「分野横断的な懸念事項」を参照してください。

XML

拡張マークアップ言語 (XML) は、相互運用性を確保したり、データベースの外でデータ構造を維持したりするのに役立ちます。パフォーマンス上の理由から、膨大な量のデータに XML を使用する場合は注意が必要です。大量のデータを XML 形式で処理する必要がある場合は、(データの値が要素の値として格納されて、サイズが大きくなる) 要素ベースのスキーマではなく、(データの値が属性として格納される) 属性ベースのスキーマを使用します。XML の使用に関する設計を行う際には、次のガイドラインを考慮します。

- XML 形式のデータへのアクセスには (特に大量の XML データがある場合は)、 XML リーダーや XML ライターを使用することを検討します。リレーショナル データベースとやり取りする必要がある場合は、この機能をサポートするオブジェクト (ADO.NET の DataSet など) の使用を検討します。XML リーダーや XML ライターでの空白やコメントの処理に関しては、一般的な設定を使用します。
- 形式を定義し、XML 形式で格納および転送するデータの検証に、XML スキーマを使用することを検討します。XML スキーマ内の複雑なデータ パラメーターには、カスタム バリデーターを使用することを検討します。ただし、検証を行うとパフォーマンスが低下することを覚えておいてください。
- パフォーマンスを最大限に高めるために、XML データはデータベースの型指定された列に格納します (ただし、このような列が存在する場合に限ります) 。XML データを定期的にクエリする場合は、インデックスを設定します (ただし、データベースでインデックスがサポートされている場合に限ります)。

テクノロジに関する考慮事項

次のガイドラインは、設計するアプリケーションの種類とそのアプリケーションの要件に応じて、適切な実装テクノロジと技法を選択するのに役立ちます。

- クエリとパラメーターの基本的なサポートが必要な場合は、ADO.NET オブジェクトを直接使用することを検討します。
- 複雑なデータ アクセス シナリオをサポートしたり、データ アクセス コードを簡略化したりする必要がある場合は、Enterprise Library の Data Access Application Block を使用することを検討します。Enterprise Library の詳細については、付録 F「patterns & practices の Enterprise Library」を参照してください。
- 基になるデータベースのデータ モデルに基づくページを使用するデータ ドリブン Web アプリケーションを構築する場合は、ASP.NET Dynamic Data を使用することを検討します。
- XML 形式のデータを操作する場合は、System.Xml 名前空間のクラスとこれに従属する名前空間 LINQ to XML (XLinq) のクラスの使用を検討します。
- ASP.NET を使用してユーザー インターフェイスを作成する場合は、レンダリングのパフォーマンスを最大限に高めるために、DataReader を使用してデータにアクセスすることを検討します。DataReader は、各行のデータを迅速に処理する読み取り専用かつ順方向専用の操作に最適です。
- SQL Server にアクセスする場合は、パフォーマンスを最大限に高めるために、ADO.NET の SqlClient 名前空間のクラスを使用することを検討します。
- SQL Server 2008 にアクセスする場合は、BLOB データを格納したり、BLOB データにアクセスする際の柔軟性を高めるために、FILESTREAM を使用することを検討します。
- Domain Model パターンに基づいてオブジェクト指向のビジネス レイヤーを設計する場合は、オブジェクト/リレーショナル マッピング (O/RM) フレームワーク (ADO.NET Entity Framework やオープン ソースの NHibernate フレームワークなど) を使用することを検討します (詳細については、この章の最後の「関連情報」を参照してください)。

データ アクセス テクノロジの選択に関するガイダンスについては、第 15 章「データ コンポーネントの設計」を参照してください。マイクロソフト プラットフォームで使用できるデータ アクセス テクノロジについては、付録 C「データ アクセス テクノロジ」を参照してください。

パフォーマンスに関する考慮事項

パフォーマンスは、データ レイヤー設計とデータベース設計の両方の影響を受けます。データのスループットが最大限になるようにシステムを調整する場合は、両方の設計を考慮する必要があります。パフォーマンスに関する設計を行う際には、次のガイドラインを考慮します。

- 接続プールを使用し、擬似ロード シナリオの実行によって得られた結果に基づいてパフォーマンスを調整します。
- データ クエリの分離レベルを調整することを検討します。高いスループットが必要なアプリケーションを構築する場合、特殊なデータ操作は、トランザクションの残りの部分よりも低い分離レベルで実行できます。複数の分離レベルを併用するとデータの一貫性が低下する可能性があるので、この方法を使用するかどうかはケース バイ ケースで慎重に分析する必要があります。
- データベース サーバーへのラウンド トリップの回数を減らすために、コマンドのバッチ処理を検討します。
- データベース データのロックによるコストを削減するために、不揮発性データには、オプティミスティック同時実行制御を使用することを検討します。これにより、データベースの行のロックによるオーバーヘッド (ロック中は接続を開いたままにしておかなければならないことを含む) を回避できます。
- DataReader を使用する場合は、パフォーマンスを向上するために序数参照を使用します。

セキュリティに関する考慮事項

データ レイヤーでは、データを盗んだり破損させたりする攻撃からデータベースを保護して、 データ ソースへのアクセスは必要最低限にとどめるように制御する必要があります。また、データ ソースへのアクセスに使用されるメカニズムを保護する必要もあります。セキュリティに関する設計を行う際には、次のガイドラインを考慮します。

- SQL Server を使用する場合は、信頼されたサブシステム モデルを実装して、Windows 認証を使用することを検討します。信頼されたサブシステム モデルについては、第 19 章「物理層と配置」を参照してください。
- システム/ユーザー データ ソース名 (DSN) を使用するのではなく、構成ファイルで接続文字列を暗号化します。
- パスワードを格納する際には、暗号化されたバージョンのパスワードではなく、salt 処理されたハッシュを使用します。

- 呼び出し元が監査のためにデータ レイヤーに ID 情報を送信するようにします。
- セキュリティの問題を軽減し SQL インジェクション攻撃が成功する可能性を減らすために、パラメーター化された SQL クエリと型指定されたパラメーターを使用します。ユーザー入力データから動的クエリを作成するために、文字列の連結を使用しないでください。

配置に関する考慮事項

データ レイヤーを配置する場合、ソフトウェア アーキテクトの目標は、運用環境におけるパフォーマンスとセキュリティの問題を考慮することです。データ レイヤーを配置する際には、次のガイドラインを考慮します。

- スケーラビリティやセキュリティの要件に違反しない場合は、アプリケーションのパフォーマンスを向上するために、データ アクセス レイヤーをビジネス レイヤーと同じ層に配置します。
- リモート データ アクセス レイヤーをサポートする必要がある場合は、パフォーマンスを向上するために TCP プロトコルの使用を検討します。
- データ アクセス レイヤーは、データベースとは別のサーバーに配置することを検討します。データベース サーバーの物理的な特性は、データベース サーバーの役割に合わせて最適化されているため、データ レイヤーに最適な動作特性に適合することはめったにありません。この 2 つを同じ物理層に配置すると、アプリケーションのパフォーマンスが低下する可能性は非常に高くなります。

データ レイヤーの設計手順

適切な手法でデータ レイヤーを設計すると、開発期間が短縮され、アプリケーション配置後のデータ レイヤーの保守が楽になります。このセクションでは、データ レイヤーの効果的な設計手法の概要を簡単に説明します。データ レイヤーを設計する際には、次の主な手順を実行します。

1. **データ アクセス レイヤーの全体的な設計を作成する**: 新しい環境と既存の環境のどちらで作業するのかを決めて、データ ソースの制約を特定し、関連する制約を特定します。また、新規開発が必要な場合は、現状のデータ ソースと共存させる方法を検討します。
    - データ ソースに関連する作業が行われていない新しい環境のシナリオでは、データ ソースで使用するスキーマを完全に制御できます。制約はデータ ソース自体に基づいたものになります。

- 既存環境のシナリオでは、データ ソース スキーマを制御することはできません。また、データベース、既存のコンポーネントとのやり取りに使用するゲートウェイ コンポーネントなど、あらゆるものがデータ ソースとなる可能性があります。既存の業務の複雑さと制約を理解する必要があります。たとえば、既存のスキーマの変更を妨げる、定義済みの運用データ ストアなどの制約が存在するかどうかを判断する必要があります。ただし、通常、既存のスキーマに新しいテーブルやビューを追加することは可能です。また、Web サービスを使用してデータ レイヤーとやり取りするのか、ゲートウェイ コンポーネントを使用してレガシ アプリケーションとやり取りするのかも判断します。前者の場合、実行できる操作は、Web サービス コントラクトで定義されている操作に制限され、後者の場合は、ゲートウェイ コンポーネントによって公開されるインターフェイスで定義されている操作に制限されます。

2. **必要なエンティティの種類を選択する**: データ アクセス コンポーネントでは、エンティティを処理します。エンティティはアプリケーションで使用するデータの格納と管理に使用するので、必要なすべてのデータ検証コードをエンティティに含めることを検討する必要があります。選択したデータの種類と形式によって、相互運用性とシリアル化の要件への対応方法が異なるので、ビジネス エンティティに適したデータの種類と形式を選択することも重要です。使用するエンティティの種類の選択に関するガイダンスと、ビジネス コンポーネントやデータ コンポーネントで一般的に使用されるエンティティの種類については、第 13 章「ビジネス エンティティの設計」を参照してください。適切なデータ形式を選択および実装する際には、次のガイドラインを考慮します。

    - 単純な CRUD ベースのアプリケーションで非接続型のシナリオをサポートする必要がある場合は、DataSet または DataTable を個別に使用することを検討します。最も一般的な方法は、ADO.NET プロバイダーを使用することです。この方法は、ADO.NET プロバイダーを既に使用している既存のアプリケーションに最適です。新しいアプリケーションを開発している場合は、LINQ to DataSet を使用して、LINQ クエリで DataSet にデータを設定できます。
    - 他のアプリケーションがデータ アクセス レイヤーにアクセスし、他のシステムやプラットフォームとの相互運用性を確保する必要がある場合は、XML 形式を使用します。
    - アプリケーションの保守容易性が重要な場合は、カスタム ビジネス エンティティを使用します。これには、エンティティをデータベース操作にマップする追加のコードが必要になりますが、オブジェクト/リレーショナル マッピング (O/RM) ソリューションを使用すると、必要なカスタム コードの量を削減できます。より高い柔軟性が必要な場合は、ADO.NET Entity

Framework または別の O/RM フレームワーク (オープン ソースの NHibernate フレームワークなど) を選択します。

- 基本的な機能を提供して一般的なタスクをカプセル化している基本クラスから派生させて、エンティティを実装します。ただし、無関係の操作で基本クラスをオーバーロードしないように注意してください。これを行うと、基本クラスから派生するエンティティのまとまりが低下し、保守容易性とパフォーマンスの問題が発生する場合があります。
- データベース操作にはデータ アクセス ロジック コンポーネントを使用するようにエンティティを設計します。すべてのデータ アクセス ポリシーと関連するビジネス ロジックの実装を一元化します。たとえば、ビジネス エンティティが SQL Server データベースに直接アクセスする場合、ビジネス エンティティを使用するクライアントに配置されたアプリケーションでは、SQL Server への接続とログオンのアクセス許可が必要になります。

3. **データ アクセス テクノロジを選択する**: データ アクセス ロジックに必要な機能を特定し、その要件を満たすテクノロジを選択します。マイクロソフト プラットフォームで使用できるデータ アクセス テクノロジの種類については、付録 C「データ アクセス テクノロジ」を参照してください。
4. **データ アクセス コンポーネントを設計する**: アクセスするデータ ソースを列挙して、各データ ソースへのアクセス方法を決定します。データ アクセス コンポーネントの開発と保守を簡略化するために、ヘルパー コンポーネントを使用する必要があるのか、または使用するのが望ましいかを判断します。最後に、関連する設計パターンを特定します。たとえば、Table Data Gateway、Query Object、Repository などのパターンを使用することを検討します。詳細については、第 15 章「データ コンポーネントの設計」を参照してください。
5. **サービス エージェントを設計する**: 適切なツールを使用してサービス参照を追加します。サービス参照を追加すると、プロキシと、サービスのデータ コントラクトを表すデータ クラスが生成されます。次に、アプリケーションでサービスをどのように使用するかを決定します。ほとんどのアプリケーションでは、サービス エージェントが提供する機能やデータには、データ アクセス コンポーネント経由でアクセスする必要があります。これにより、データ ソースの種類にかかわらず、一貫性のあるインターフェイスが提供されます。小規模なアプリケーションでは、ビジネス レイヤー (場合によってはプレゼンテーション レイヤー) がサービス エージェントに直接アクセスすることがあります。

関連する設計パターン

次の表に示すように、主要なパターンはカテゴリに分類されます。各カテゴリの設計を行う際には、これらのパターンの使用を検討します。

カテゴリ	関連するパターン
全般	**Active Record**: ドメイン エンティティのデータ アクセス オブジェクトを含みます。 **Data Mapper**: オブジェクトとデータベース構造の独立性を維持しながら、データを構造間で移動するのに使用される、オブジェクトとデータベース構造との間のマッピング レイヤーを実装します。 **Data Transfer Object**: プロセス間で転送されるデータを格納して、必要なメソッド呼び出しの回数を削減するオブジェクトです。 **Domain Model**: ドメイン内のエンティティとエンティティ間の関係を表す一連のビジネス オブジェクトです。 **Query Object**: データベース クエリを表すオブジェクトです。 **Repository**: ドメイン エンティティを処理するデータ ソースのメモリ内表現です。 **Row Data Gateway**: データ ソース内の 1 つのレコードへのゲートウェイとして機能するオブジェクトです。 **Table Data Gateway**: データ ソース内のテーブルやビューへのゲートウェイとして機能し、選択、挿入、更新、および削除のクエリをすべて一元化するオブジェクトです。 **Table Module**: データベースのテーブルまたはビュー内のすべての行のビジネス ロジックを処理する、1 つのコンポーネントです。
バッチ処理	**Parallel Processing**: 合計処理時間を最小限に抑えるために、複数のバッチ ジョブを並列実行します。 **Partitioning**: 複数の大規模なバッチ ジョブを同時実行できるように分割します。
トランザクション	**Capture Transaction Details**: 当該のトランザクションに属するすべてのテーブルに加えられた変更を記録するために、トリガーやシャドウ テーブルなどのデータベース オブジェクトを作成します。 **Coarse Grained Lock**: 1 回のロックで、関連する一連のオブジェクトをロックします。 **Implicit Lock**: 共有リソースにアクセスするコードの代わりにフレームワーク コードを使用してロックを行います。 **Optimistic Offline Lock**: あるセッションで加えられた変更が別のセッションで加えられた変更と競合しないようにします。

	Pessimistic Offline Lock: データを使用する前にトランザクションでデータをロックすることにより、競合を回避します。 **Transaction Script**: 各トランザクションのビジネス ロジックを 1 つのプロシージャにまとめて、データベースを直接または軽量なデータベース ラッパー経由で呼び出します。

Domain Model、Table Module、Coarse Grained Lock、Implicit Lock、Transaction Script、Active Record、Data Mapper、Data Transfer Object、Optimistic Offline Lock、Pessimistic Offline Lock、Query Object、Repository、Row Data Gateway、および Table Data Gateway の各パターンの詳細については、『エンタープライズ アプリケーション アーキテクチャ パターン』(Martin Fowler 著、翔泳社、2005 年) を参照してください。または、http://martinfowler.com/eaaCatalog/ (英語) を参照してください。
Capture Transaction Details パターンの詳細については、「Data Patterns」(http://msdn.microsoft.com/en-us/library/ms998446.aspx、英語) を参照してください。

関連情報

データ アクセスについての一般的なガイドラインや情報に関する Web リソースに、より簡単にアクセスするには、http://www.microsoft.com/architectureguide (英語) でオンライン版の参考文献を参照してください。

- .NET Data Access Architecture Guide (http://msdn.microsoft.com/en-us/library/ms978510.aspx、英語)
- Transaction Control (http://msdn.microsoft.com/en-us/library/ms978457.aspx、英語)
- Data Patterns (http://msdn.microsoft.com/en-us/library/ms998446.aspx、英語)
- データ層コンポーネントの設計と層間のデータの受け渡し (http://msdn.microsoft.com/ja-jp/library/dd297667.aspx)
- Handling BLOBs (http://msdn.microsoft.com/en-us/library/ms978510.aspx#daag_handlingblobs、英語)
- Stored Procedures vs. Direct SQL (http://msdn.microsoft.com/en-us/library/ms978510.aspx、英語)
- NHibernate Forge コミュニティ サイト (http://nhforge.org/Default.aspx、英語)

9

サービス レイヤーのガイドライン

概要

サービスを通じてアプリケーションの機能を提供する場合は、サービスの機能を別個のサービス レイヤーに分離することが重要です。この章は、アプリケーション アーキテクチャにおけるサービス レイヤーの位置付けについて理解し、サービス レイヤーの設計手順を学ぶのに役立ちます。また、サービス レイヤーの設計時に発生する一般的な問題に関するガイダンスを提供し、サービス レイヤーを設計する際に使用する主要なパターンとテクノロジに関する主要な考慮事項についても説明します。

サービス レイヤーでは、サービス インターフェイスとデータ コントラクト (メッセージの種類) を定義して実装します。覚えておく必要がある重要な概念の 1 つは、サービスでは、内部の処理やアプリケーション内で使用するビジネス エンティティの詳細を公開してはならないということです。特に、ビジネス レイヤーのエンティティが、データ コントラクトに必要以上に影響を及ぼさないようにする必要があります。サービス レイヤーでは、ビジネス レイヤーのエンティティとデータ コントラクトの間でデータ形式を変換するトランスレーター コンポーネントを提供する必要があります。

図 1 は、アプリケーションの設計全体におけるサービス レイヤーの位置付けを示しています。

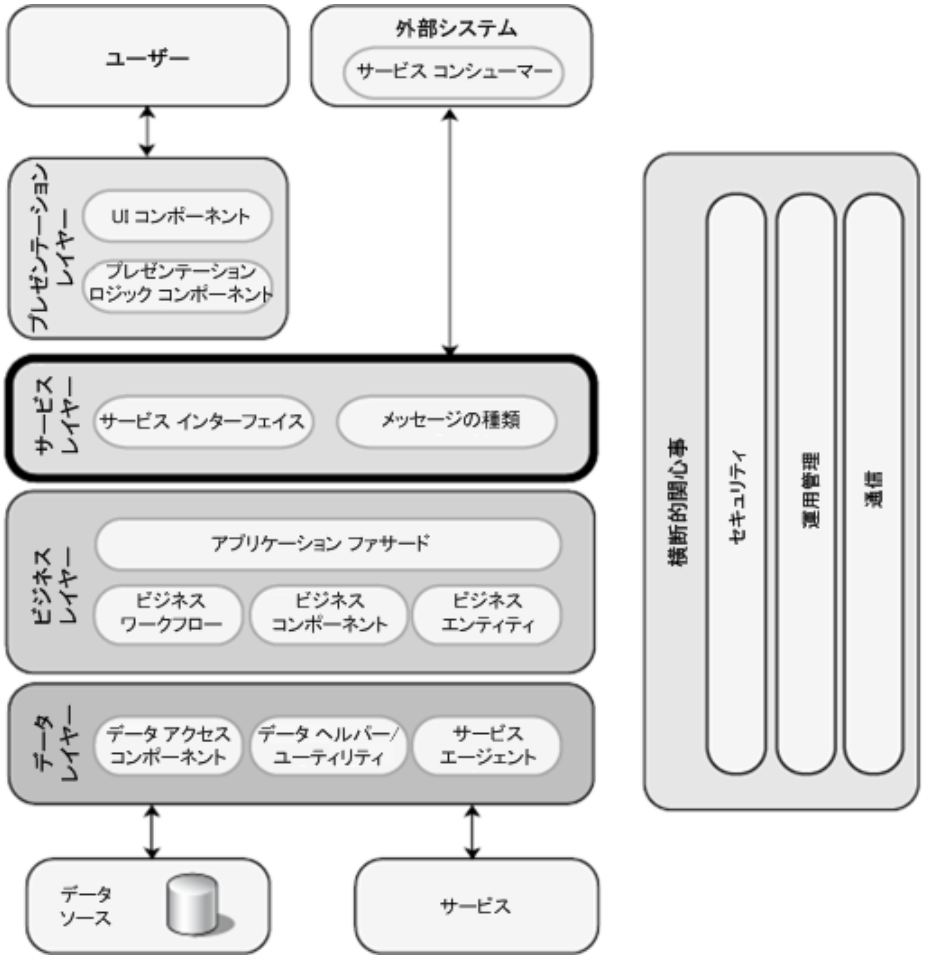


図 10

一般的なアプリケーションのサービス レイヤーとそこに含まれる場合があるコンポーネント

通常、サービス レイヤーには、次のコンポーネントが含まれています。

- **サービス インターフェイス**: サービスでは、すべての受信メッセージが送信されるサービス インターフェイスを公開します。サービス インターフェイスは、アプリケーションに実装されているビジネス ロジック (通常、ビジネス レイヤーのロジック) を潜在的なコンシューマーに公開するファサードと見なすことができます。
- **メッセージの種類**: サービス レイヤーでデータをやり取りする際、データ構造は、さまざまな種類の操作をサポートするメッセージ構造でラップされています。通常、サービス レイヤーには、データ型とメッセージ内で使用されるデータ型を定義するコントラクトも含まれます。

サービス レイヤーで一般的に使用するコンポーネントの詳細については、第 10 章「コンポーネントのガイドライン」を参照してください。サービス インターフェイスの設計に関する詳細については、第 18 章「通信とメッセージ」を参照してください。

設計に関する考慮事項

サービス レイヤーを設計する際に考慮しなければならない要素はたくさんあります。このような設計に関する考慮事項の多くは、レイヤー型アーキテクチャに関する実証済みの慣例と関連があります。しかし、サービスに関しては、メッセージ関連の要素も考慮する必要があります。主な考慮事項は、サービスで使用するのがメッセージ ベースの通信 (通常、ネットワーク経由) であることと、この通信はプロセス内での直接の通信よりも低速で、多くの場合、サービスとコンシューマーの間の通信が非同期になることです。さらに、確実な配信メカニズムが使用されていない場合、サービスとコントラクトの間でやり取りされるメッセージは、ルーティングされたり、変更されたり、送信時とは異なる順序で配信されたり、場合によっては失われたりすることがあります。このような考慮事項に対処するには、非決定的なメッセージング動作から成る設計が必要になります。サービス レイヤーを設計する際には、次のガイドラインを考慮します。

- **コンポーネントではなくアプリケーションを対象とするようにサービスを設計する**: サービスの操作は、粒度が粗く、アプリケーションの操作に焦点を合わせたものにする必要があります。粒度が細かすぎると、パフォーマンスやスケーラビリティの問題が発生する場合があります。しかし、サービスでは、膨大なデータを無制限に返さないようにする必要があります。たとえば、大量の人口統計データを返す可能性のあるサービスでは、1 回の呼び出しですべてのデータを返すのではなく、データの適切なサイズのサブセットを返す操作を提供する必要があります。サブセットのサイズは、サービスとそのコンシューマーに適したサイズにする必要があります。
- **拡張性を考慮し、クライアントの身元が明らかだと仮定せずにサービスとデータ コントラクトを設計する**: データ コントラクトは、(可能であれば) サービスのコンシューマーに影響を及ぼすことなく拡張できるように設計する必要があります。しかし、過度に複雑になることを避けたり、下位互換性のない変更を管理したりするために、既存のバージョンと並行して動作する新しいバージョンのサービス インターフェイスを作成しなければならないことがあります。クライアントや提供するサービスをクライアントがどのように使用する予定であるかに関して、仮定を立てるべきではありません。
- **サービス コントラクトのためだけの設計を行う**: サービス レイヤーでは、サービス コントラクトで指定されている機能のみを実装および提供する必要があり、内部の実装やサービスの詳細は、外部のコンシューマーに公開するべきではありません。また、サービスで実装する新機能を含むようにサービス

コントラクトを変更する必要があり、新しい操作と型が既存のコントラクトとの下位互換性を持たない場合は、コントラクトをバージョン管理することを検討します。サービスで公開される新しい操作を新しいバージョンのサービス コントラクトで定義し、新しいスキーマ型を新しいバージョンのデータ コントラクトで定義します。メッセージ コントラクトの設計に関する詳細ついては、第 18 章「通信とメッセージ」を参照してください。

- **サービス レイヤーに関する関心事をインフラストラクチャに関する関心事から分離する**: 横断的関心事を管理するためのコードは、サービス レイヤーのサービス ロジック コードと混在しないようにする必要があります。コードが混在すると、拡張や保守が困難な実装になる場合があります。一般に、横断的関心事を管理するコードは別個のコンポーネントに実装し、こうしたコンポーネントにはビジネス レイヤーのコンポーネントからアクセスする必要があります。
- **エンティティを標準的な要素で構成する**: 可能であれば、標準的な要素を使用して、サービスで使用する複雑な型やデータ転送オブジェクトを構成します。
- **無効な要求を受信する可能性を想定して設計する**: サービスで受信するメッセージがすべて有効であるとは限りません。検証ロジックを実装して、すべての入力を値、範囲、および型に基づいてチェックし、すべての無効なデータを拒否したり、すべての無効なデータの不適切な部分を削除したりします。検証の詳細については、第 17 章「横断的関心事」を参照してください。
- **重複するメッセージをサービスで検出および管理できるようにする (べき等性)**: サービスを設計する際には、Idempotent Receiver や Replay Protection などの既知のパターンを実装して、重複するメッセージが処理されないようにしたり、その重複する処理が結果に影響を及ぼさないようにします。
- **バラバラの順序で到着するメッセージをサービスで管理できるようにする (交換性)**: メッセージがバラバラの順序で到着する可能性がある場合は、メッセージを格納してから適切な順序で処理する設計を実装します。

設計に関する具体的な問題

サービス レイヤーの設計を策定する際に考慮する必要がある一般的な問題がいくつかあります。これらの問題は、設計の具体的な領域に分類できます。次のセクションでは、最も頻繁にミスが発生する各カテゴリに関するガイドラインを示します。

- 認証
- 承認

- 通信
- 例外管理
- メッセージング チャネル
- メッセージの構築
- メッセージ エンドポイント
- メッセージの保護
- メッセージのルーティング
- メッセージの変換
- サービス インターフェイス
- 検証

メッセージ プロトコル、非同期通信、相互運用性、パフォーマンス、およびテクノロジの選択肢の詳細については、第 18 章「通信とメッセージ」を参照してください。

認証

認証は、サービス コンシューマーの身元を特定するために使用します。サービス レイヤー用の効果的な認証方針を設計することは、アプリケーションのセキュリティと信頼性の両方において重要です。適切な認証方針を設計しないと、スプーフィング攻撃、辞書攻撃、セッション ハイジャックなどの攻撃に対して、アプリケーションが脆弱になる可能性があります。認証の方針を設計する際には、次のガイドラインを考慮します。

- (可能であれば、基盤となるプラットフォームの機能を活用して) ユーザーを安全に認証するための適切なメカニズムを特定し、認証を適用する必要がある信頼境界を特定します。
- 異なる信頼設定を使用するとサービス コードの実行にどのような影響があるかを考慮します。
- 基本認証を使用する場合、または資格情報がプレーンテキストとして渡される場合は、Secure Sockets Layer (SSL) などのセキュリティで保護されたプロトコルを使用するようにします。WS* 標準 (Web Services Security、Web Services Trust、および Web Services Secure Conversation) でサポートされるメッセージ レベルのセキュリティ メカニズムを SOAP メッセージで使用することを検討します。

承認

承認は、認証されたサービス コンシューマーがアクセスできるリソースや操作を特定するために使用します。サービス レイヤー用の効果的な承認方針を設計することは、アプリケーションのセキュリティと信頼性の両方において重要です。適切な承認方針を設計しないと、情報漏えい、データの改ざん、および特権の昇格に対して、アプリケーションが脆弱になる可能性があります。承認方針では、通常、粒度の粗い操作や作業 (こうした操作や作業を実行するのに必要なリソースではなく) を表現する必要があります。承認の方針を設計する際には、次のガイドラインを考慮します。

- ユーザー、グループ、および役割ごとに、リソースに適切なアクセス許可を設定します。最も権限の低い適切なアカウントでサービスを実行します。
- 承認方針の有効性と管理容易性を維持するため、可能であれば、粒度の細かい承認は避けます。
- Windows 認証を使用する場合は、URL 承認やファイルの承認を使用します。
- 必要に応じて、宣言型の主要なアクセス許可の要求を使用して、一般に公開されているメソッドへのアクセスを制限します。

通信

サービスの通信方針を設計する場合は、サービスでサポートしなければならない配置シナリオに基づいてプロトコルを選択する必要があります。通信の方針を設計する際には、次のガイドラインを考慮します。

- 通信要件を分析し、"要求 - 応答" と "二重通信" のどちらが必要で、メッセージ通信を一方向と双方向のどちらにする必要があるかを判断します。また、非同期呼び出しを行う必要があるかどうかも判断します。
- 信頼できない通信や断続的な通信の処理方法を決定します。サービス エージェントを実装したり、キューにメッセージを格納する信頼できるシステム (メッセージ キューなど) を使用したりする方法が考えられます。
- サービスを閉じたネットワーク内に配置する場合は、通信効率を最大限に高めるために伝送制御プロトコル (TCP) の使用を検討します。サービスを一般に公開されるネットワークに配置する場合は、ハイパーテキスト転送プロトコル (HTTP) の使用を検討します。
- 柔軟性を最大限に高めるために、動的 URL 動作を構成されたエンドポイントで使用することを検討します。たとえば、可能であれば、エンドポイントの URL をハードコーディングするのではなく、構成

や Universal Description, Discovery, and Integration (UDDI) などのディレクトリ サービスを使用します。

- メッセージのエンドポイント アドレスを検証し、メッセージに含まれる機密データを保護します。

例外管理

サービス レイヤー用の効果的な例外管理方針を設計することは、アプリケーションのセキュリティと信頼性の両方において重要です。正しく設計しないと、アプリケーションがサービス拒否 (DoS) 攻撃に対して脆弱になったり、機密情報や重要な情報が開示されたりする可能性があります。例外の発生とハンドルは負荷の高い操作なので、例外管理を設計する際には、パフォーマンスへの影響を考慮する必要があります。良い方法は、例外管理とログ記録の一元化されたメカニズムを設計し、システム管理者を支援するために、インストルメンテーションと一元的な監視をサポートするアクセス ポイントの提供を検討することです。例外管理の方針を設計する際には、次のガイドラインを考慮します。

- ハンドルできる例外のみをキャッチし、例外が発生した場合にどのようにしてメッセージの整合性を維持するかを検討します。ハンドルされていない例外を適切にハンドルするようにします。また、ビジネス ロジックの制御に例外を使用することは避けます。
- SOAP エラー要素またはカスタムの拡張を使用して、例外の詳細を呼び出し元に返します。
- 例外はログに記録しますが、例外メッセージやログ ファイルで機密情報を開示しないようにします。

例外管理の技法に関する詳細については、第 17 章「横断的関心事」を参照してください。

メッセージング チャネル

サービスとコンシューマーの間の通信は、チャネルを通じたデータの送信で成り立っています。ほとんどの場合は、選択したサービス インフラストラクチャ (Windows Communication Foundation (WCF) など) で提供されるチャネルを使用します。選択したインフラストラクチャでサポートされているパターンを理解し、サービスのコンシューマーとの通信に適したチャネルを特定する必要があります。メッセージ チャネルを設計する際には、次のガイドラインを考慮します。

- メッセージング チャネルに関連するパターン (Channel Adapter、Message Bus、Messaging Bridge など) を特定し、シナリオに適したパターンを選択します。また、要件に合った適切なサービス インフラストラクチャを選択するようにします。
- 必要に応じて、エンドポイント間でデータを傍受して検査する方法を決定します。

- 例外の条件をチャネルで処理するようにします。
- メッセージングをサポートしていないクライアントにアクセスを提供する方法を検討します。

メッセージの構築

サービスとコンシューマーの間でデータをやり取りする際、データはメッセージ内にラップされている必要があります。このメッセージの形式は、サポートする必要がある操作の種類によって異なります。操作の例には、ドキュメントのやり取り、コマンドの実行、イベントの生成などがあります。メッセージ構築の方針を設計する際には、次のガイドラインを考慮します。

- メッセージ構築に関連するパターン (Command Message、Document Message、Event Message、Request-Reply など) を特定し、シナリオに適したパターンを選択します。
- 膨大な量のデータをより小さなチャンクに分割し、それを順に送信します。
- 低速なメッセージ配信チャネルを使用する場合は、時間的制約のあるメッセージに有効期限情報を含めることを検討します。サービスでは、期限切れのメッセージを無視するようにします。

メッセージ エンドポイント

メッセージ エンドポイントは、アプリケーションがサービスとの通信に使用する接続を表します。サービス インターフェイスの実装がメッセージ エンドポイントを表します。サービスの実装を設計する際には、重複するメッセージや無効なメッセージがサービスに送信される可能性を考慮する必要があります。メッセージ エンドポイントを設計する際には、次のガイドラインを考慮します。

- メッセージ エンドポイントに関連するパターン (Messaging Gateway、Messaging Mapper、Competing Consumer、Message Dispatcher など) を特定し、シナリオに適したパターンを選択します。
- すべてのメッセージを受け入れるべきか、それとも特定のメッセージを処理するためのフィルターを実装する必要があるかを判断します。
- メッセージ インターフェイスのべき等性を考慮して設計を行います。べき等性とは、同じコンシューマーから重複するメッセージを受信する可能性があるが、その中の 1 つだけを処理する必要がある状況を指します。つまり、べき等性を考慮したエンドポイントでは、1 つのメッセージだけが処理され、重複するメッセージはすべて無視されることを保証します。

- メッセージ インターフェイスの交換性を考慮して設計を行います。交換性は、メッセージを受信する順序に関係しています。場合によっては、適切な順序で処理できるように、受信メッセージを格納する必要があります。
- 非接続型のシナリオを考慮して設計を行います。たとえば、後で配信できるようにメッセージをキャッシュまたは格納して、確実な配信をサポートすることが必要な場合があります。接続が切断されている間は、エンドポイントをサブスクライブしないようにする必要があります。

メッセージの保護

サービスとコンシューマーの間で機密データを転送する場合は、メッセージの保護を考慮した設計を行う必要があります。トランスポート レイヤーの保護 (IPSec、SSL など) またはメッセージ ベースの保護 (暗号化、デジタル署名など) を使用できます。メッセージの保護を設計する際には、次のガイドラインを考慮します。

- ほとんどの場合、メッセージ ベースのセキュリティ技法を使用してメッセージのコンテンツを保護することを検討する必要があります。メッセージ ベースのセキュリティは、メッセージに含まれる機密データを暗号化して保護するのに役立ち、デジタル署名は、メッセージの否認と改ざんを防ぐのに役立ちます。ただし、セキュリティ対策 1 つ 1 つがパフォーマンスに影響を与えることに注意してください。
- サービスとコンシューマーの間の通信が中間デバイス (他のサーバーやルーターなど) 経由でルーティングされない場合は、トランスポート レイヤーのセキュリティ (IPSec や SSL など) を使用できます。しかし、メッセージが中間デバイスを経由する場合は、必ず、メッセージ ベースのセキュリティを使用します。トランスポート レイヤーのセキュリティを使用すると、メッセージは、経由する中間デバイスで復号化されてから暗号化されるので、これはセキュリティ リスクとなります。
- セキュリティを最大限に高めるために、設計では、トランスポート レイヤーのセキュリティとメッセージ ベースのセキュリティの両方を使用することを検討します。トランスポート レイヤーのセキュリティは、メッセージ ベースのセキュリティを使用して暗号化できないヘッダー情報を、保護するのに役立ちます。

メッセージのルーティング

メッセージ ルーターは、サービス コンシューマーをサービスの実装から分離するために使用します。使用する可能性のあるルーターの主な種類は、単純なルーター、複合ルーター、およびパターン ベースのルーターの 3 つです。

単純なルーターでは、1 つのルーターを使用してメッセージの最終的な送信先を特定します。複合ルーターでは、複数の単純なルーターを組み合わせて、より複雑なメッセージ フローを処理します。単純なメッセージ ルーターに基づいてさまざまなルーティング スタイルを表現するには、アーキテクチャ パターンを使用します。メッセージのルーティングを設計する際には、次のガイドラインを考慮します。

- メッセージのルーティングに関連するパターン (Aggregator、Content-Based Router、Dynamic Router、Message Filter など) を特定し、シナリオに適したパターンを選択します。
- 連続性のある複数のメッセージがコンシューマーから送信される場合、ルーターでは、このすべてのメッセージが、必要な順序で同じエンドポイントに配信されるようにする必要があります (交換性)。
- メッセージ ルーターでは、メッセージをどのようにルーティングするかを決定するために、メッセージ内の情報を検査する場合があります。そのため、ルーターがこの情報にアクセスできるようにする必要があります。また、ヘッダーにルーティング情報を追加することが必要な場合もあります。メッセージを暗号化する場合、メッセージのルーティングに必要な情報は、暗号化されていないヘッダーに含めるようにする必要があります。

メッセージの変換

サービスとコンシューマーの間でメッセージをやり取りする際には、メッセージをコンシューマーが認識できる形式に変換しなければならない場合がよくあります。アダプターを使用して、メッセージをサポートしていないクライアントにメッセージ チャネルへのアクセスを提供することができます。また、トランスレーターを使用して、メッセージのデータを、各コンシューマーが認識できる形式に変換することもできます。メッセージの変換を設計する際には、次のガイドラインを考慮します。

- 変換を実行する要件と実行場所を決定します。変換のパフォーマンス オーバーヘッドを考慮に入れ、実行する変換の回数を最小限に抑えるようにします。
- メッセージの変換に関連するパターン (Canonical Data Mapper、Envelope Wrapper、Normalizer など) を特定します。ただし、Canonical Data Mapper モデルは必要な場合にのみ使用します。
- メタデータを使用してメッセージ形式を定義します。
- メタデータの格納に外部リポジトリを使用することを検討します。

サービス インターフェイス

サービス インターフェイスは、サービスで公開されるコントラクトを表します。コントラクトでは、サービスでサポートする操作、それに関連するパラメーターとデータ転送オブジェクトを定義します。サービス インターフェイスを設計する場合は、越える必要がある境界とサービスにアクセスするコンシューマーの種類を検討する必要があります。たとえば、サービスの操作は、粒度が粗く、アプリケーションを対象としたものにする必要があります。サービス インターフェイスの設計で発生しがちな最大のミスの 1 つは、サービスを操作の粒度が細かいコンポーネントとして扱うことです。これは、物理的な境界やプロセス境界を越えた呼び出しを何度も行わなければならない設計になります。このような呼び出しを何度も行うと、パフォーマンスの低下や待ち時間の増加につながる可能性があります。サービス インターフェイスを設計する際には、次のガイドラインを考慮します。

- 粒度の粗いインターフェイスを使用して、要求をバッチ処理し、ネットワーク経由の呼び出しの回数を最小限に抑えることを検討します。
- ビジネス ロジックの変更がインターフェイスに影響を及ぼすようなサービス インターフェイスの設計は避けます。ただし、ビジネス要件が変化した場合、これは避けられないことがあります。
- ビジネス ルールをサービス インターフェイスに実装することは避けます。
- さまざまな種類のクライアントとの互換性を最大限に高めるために、パラメーターには標準的な形式を使用することを検討します。インターフェイス設計では、クライアントがどのようにサービスを使用するかに関して仮定を立てないでください。
- サービス インターフェイスのバージョン管理を実装するためにオブジェクトの継承を使用することは避けます。
- 開発とテスト以外では、すべてのサービスに関して、トレースとデバッグ モードのコンパイルを無効にします。

検証

サービス レイヤーを保護するには、このレイヤーで受信するすべての要求を検証する必要があります。そうしないと、悪意のある攻撃に対しても、無効な入力によって発生したエラーに対しても、アプリケーションが脆弱になる可能性があります。有効な入力と悪意のある入力に関する包括的な定義はありません。また、悪用されるリスクは、アプリケーションで入力をどのように使用するかに左右されます。検証の方針を設計する際には、次のガイドラインを考慮します。

- 検証の手法を一元化して、テスト容易性と再利用性を最大限に高めることを検討します。

- すべてのメッセージ コンテンツ (パラメーターを含む) に対して、制約、拒否、および不適な部分の削除を行います。また、長さ、範囲、形式、型を検証します。
- メッセージの検証にスキーマを使用することを検討します。スキーマを使用した検証については、「Message Validation」(http://msdn.microsoft.com/en-us/library/cc949065.aspx、英語) と「Input/Data Validation」(http://msdn.microsoft.com/en-us/library/cc949061.aspx、英語) を参照してください。

REST と SOAP

Representational State Transfer (REST) と SOAP は、2 つの異なるサービス実装スタイルを表します。厳密に言うと、REST は、HTTP プロトコルで適切に機能する単純な動詞を使用して構築されたアーキテクチャ パターンです。REST のアーキテクチャ原理は HTTP 以外のプロトコルを使用して適用することもできますが、実際には、REST の実装は HTTP と組み合わせて使用されます。SOAP は XML ベースのメッセージング プロトコルで、HTTP を含むあらゆる通信プロトコルと組み合わせて使用できます。

この 2 つの手法の主な違いは、サービスの状態を管理する方法です。サービスの状態は、アプリケーションやセッションの状態と考えるのではなく、アプリケーションが実行中に経るさまざまな状態と考えます。SOAP では、さまざまな状態間の遷移を 1 つのサービス エンドポイントとの通信を通じて実現することが可能で、このエンドポイントによって多くの操作やメッセージの種類がカプセル化されて、多くの操作やメッセージの種類へのアクセスが提供されることがあります。

REST では、可能な操作が限られており、そのような操作は、URI (HTTP アドレス) で表現され、URI (HTTP アドレス) でアドレス指定できるリソースに適用されます。メッセージでは、リソースの現在の状態または要求された状態をとらえます。REST は Web アプリケーションと適切に連携するので、XML 以外の MIME の種類やストリーミング コンテンツに対する HTTP のサポートをサービス要求から利用できます。REST リソース間を移動するサービス コンシューマーは、ユーザーが Web ページ間を移動し Web ページを操作するのと同じような方法で URI を操作します。

REST と SOAP はどちらもほとんどのサービス実装で使用することができますが、一般に公開されているサービスの場合または未知のコンシューマーがサービスにアクセスできる場合には、REST の手法の方が適していることが多いです。SOAP は、さまざまな手続き型の通信 (アプリケーションのレイヤー間のインターフェイスなど) を実装するのにずっと適しています。SOAP では、HTTP のみに限定されません。SOAP で利用することができる WS-* 標準は、標準を提供することにより、メッセージングに関する一般的な問題 (セキュリティ、トランザクション、アド

レス指定、信頼性など) に対処する相互運用可能な方法を提供します。REST でも同様の機能を提供することができますが、現時点では、このような領域の標準はいくつかしか存在しないので、多くの場合、カスタムのメカニズムを作成する必要があります。

一般に、SOAP ベースの通信を設計する際には、ステートレスな REST 通信を設計する場合と同じ原理を使用できます。どちらの手法でも、動詞を使用してデータ (ペイロード) をやり取りします。SOAP の場合、使用できる動詞に制約はなく、こうした動詞はサービス エンドポイントで定義されています。REST の場合、使用できる動詞は、HTTP プロトコルを再現する事前設定された動詞に限定されます。REST と SOAP のどちらを選択するかを決める際には、次のガイドラインを考慮します。

- SOAP は、抽象的なレイヤーを構築するための基盤とすることができる基本的なメッセージング フレームワークを提供するプロトコルで、XML 形式のメッセージを使用してネットワーク経由で呼び出しや応答をやり取りする RPC フレームワークとしてよく使用されます。
- SOAP では内部のプロトコル実装を通じてセキュリティやアドレス指定などの問題に対処しますが、SOAP を使用するには SOAP スタックを使用できる必要があります。
- REST は、他のプロトコル (JavaScript Object Notation (JSON)、Atom Publishing Protocol、カスタムの Plain Old XML (POX) 形式など) を利用できる技法です。
- REST では、アプリケーションとデータを単なるサービス エンドポイントとしてではなくステート マシンとして公開します。REST を使用すると、GET や PUT などの標準的な HTTP 呼び出しを使用してシステムの状態を照会および変更できます。REST は本質的にステートレスです。つまり、サーバーではセッション状態のデータが格納されないので、クライアントからサーバーに送信される個々の要求それぞれに、要求を理解するために必要な情報すべてが含まれていなければなりません。

REST 関連の設計に関する考慮事項

REST は分散システム向けのアーキテクチャ スタイルで、システムをリソースに分割して複雑さを軽減するように設計されています。あるリソースでサポートされるリソースと操作は、一連の URI (HTTP アドレス) として表現され公開されます。REST リソースを設計する際には、次のガイドラインを考慮します。

- クライアントが使用できるリソースを特定して分類します。
- リソースの表現手法を選択します。REST の URI の開始位置に意味のある名前を使用し、特定のリソース インスタンスに一意識別子を使用することをお勧めします。たとえば、http://www.contoso.com/employee/ は従業員 (employee) の URI の開始位置を表します。

http://www.contoso.com/employee/smithah01 では、従業員 ID を使用して特定の従業員が示されています。

- 異なるリソース用に複数の表現をサポートする必要があるかどうかを判断します。たとえば、リソースで XML 形式、Atom 形式、または JSON 形式をサポートする必要があるかどうかを判断し、それをリソース要求に含めることができます。リソースは両方の形式で公開できます (たとえば、http://www.contoso.com/example.atom と http://www.contoso.com/example.json で公開できます)。
- 異なるリソース用に複数のビューをサポートする必要があるかどうかを判断します。たとえば、リソースで GET 操作と POST 操作をサポートする必要があるか、GET 操作のみをサポートする必要があるかを判断します。可能であれば POST 操作の乱用を避け、操作を URI に含めることを避けます。
- ユーザー セッション状態の管理をサービスに実装したり、状態の管理にハイパーテキストを使用したりする (Web ページで非表示コントロールを使用するなど) ことは避けます。たとえば、ユーザーが要求を送信する (ショッピング カートへの商品の追加など) 際には、データベースなどの永続的な状態ストアにデータを格納します。

SOAP 関連の設計に関する考慮事項

SOAP は、サービスのメッセージ レイヤーを実装するために使用するメッセージ ベースのプロトコルです。メッセージは、ヘッダーと本文を含むエンベロープで構成されています。ヘッダーは、サービスで実行される操作の範囲外にある情報を提供するために使用できます。たとえば、ヘッダーには、セキュリティ、トランザクション、またはルーティングに関する情報が含まれる場合があります。本文には、サービスの実装に使用するコントラクトが XML スキーマの形で含まれます。SOAP メッセージを設計する際には、次のガイドラインを考慮します。

- フォールトやエラーのハンドル方法と、適切なエラー情報をクライアントに返す方法を決定します。詳細については、「Exception Handling in Service Oriented Applications」(http://msdn.microsoft.com/en-us/library/cc304819.aspx、英語) を参照してください。
- サービスで実行できる操作のスキーマ、サービス要求で渡されるデータ構造、およびサービス要求から返される可能性のあるエラーやフォールトを定義します。
- サービスに適したセキュリティ モデルを選択します。詳細については、「Improving Web Services Security: Scenarios and Implementation Guidance for WCF」(http://msdn.microsoft.com/en-us/library/cc949034.aspx、英語) を参照してください。

- メッセージ スキーマでは、複雑な型の使用を避けます。相互運用性を最大限に高めるために、単純な型のみを使用するようにします。

REST と SOAP の詳細については、第 25 章「サービス アプリケーションの定義」を参照してください。

テクノロジに関する考慮事項

次のガイドラインは、サービス レイヤーを実装するための適切なテクノロジを選択するのに役立ちます。

- 簡略化のために ASP.NET Web サービス (ASMX) を使用することを検討します (ただし、Microsoft インターネット インフォメーション サービス (IIS) を実行している Web サーバーを使用できる場合に限ります)。
- 信頼できるセッションとトランザクション、アクティビティのトレース、メッセージのログ記録、パフォーマンス カウンター、複数のトランスポート プロトコルのサポートなどの高度な機能が必要な場合は、WCF サービスの使用を検討します。
- サービスで ASMX を使用することを決めていて、メッセージ ベースのセキュリティとバイナリ データ転送が必要な場合は、Web サービス拡張 (WSE) の使用を検討します。ただし、一般に、WSE 機能が必要な場合は、WCF への移行を検討する必要があります。
- サービスで WCF を使用することを決めた場合は、次のガイドラインを考慮します。
    - WCF 以外のクライアントや Windows 以外のクライアントとの相互運用性が必要な場合は、SOAP 仕様に基づく HTTP トランスポートを使用することを検討します。
    - イントラネット クライアントをサポートする必要がある場合は、TCP プロトコルとバイナリ メッセージ エンコードをトランスポート セキュリティと Windows 認証と共に使用することを検討します。
    - 同じコンピューター上の WCF クライアントをサポートする必要がある場合は、名前付きパイプ プロトコルとバイナリ メッセージ エンコードを使用することを検討します。
    - 暗黙のメッセージ ラッパーではなく明示的なメッセージ ラッパーを使用するサービス コントラクトを定義することを検討します。これにより、メッセージ コントラクトを操作の入出力として定義することが可能になります。つまり、サービス コントラクトに影響を及ぼすことなく、メッセージ コントラクトに含まれるデータ コントラクトを拡張できます。

メッセージ テクノロジの選択肢に関する詳細については、第 18 章「通信とメッセージ」を参照してください。

配置に関する考慮事項

サービス レイヤーは、アプリケーションの他のレイヤーと同じティアに配置することも、パフォーマンスと分離の要件上必要であれば、別のティアに配置することもできます。ただし、ほとんどの場合、ビジネス機能を公開する際のパフォーマンスへの影響を最小限に抑えるために、サービス レイヤーはビジネス レイヤーと同じ物理ティアに配置されます。サービス レイヤーを配置する際には、次のガイドラインを考慮します。

- 運用環境のパフォーマンスやセキュリティの問題により不可能な場合を除き、アプリケーションのパフォーマンスを向上させるために、サービス レイヤーはビジネス レイヤーと同じティアに配置します。
- サービスがサービス コンシューマーと同じ物理ティアに配置されている場合は、名前付きパイプや共有メモリ プロトコルの使用を検討します。
- ローカル ネットワーク内の他のアプリケーションのみがサービスにアクセスする場合は、通信に TCP を使用することを検討します。
- サービスがインターネットで一般に公開されている場合は、トランスポート プロトコルに HTTP を使用します。

配置パターンの詳細については、第 19 章「物理ティアと配置」を参照してください。

サービス レイヤーの設計手順

サービス レイヤーを設計する際には、まず、サービスで公開する予定のコントラクトで構成されたサービス インターフェイスを定義します。これは一般に、コントラクト ファーストの設計と呼ばれます。サービス インターフェイスを定義したら、次は、サービスの実装を定義します。サービスの実装は、データ コントラクトをビジネス エンティティに変換したり、ビジネス レイヤーと通信したりするために使用します。サービス レイヤーを設計する際には、次の基本的な手順を使用します。

1. メッセージに使用するスキーマを表すデータ コントラクトとメッセージ コントラクトを定義する
2. サービスでサポートする操作を表すサービス コントラクトを定義する
3. サービスのコンシューマーにエラー情報を返すエラー コントラクトを定義する
4. ビジネス エンティティとデータ コントラクトとの間の変換を行う変換オブジェクトを設計する
5. ビジネス レイヤーとの通信に使用する抽象化手法を設計する

---

patterns & practices の Web Service Software Factory: Modeling Edition (Service Factory とも呼ばれます) などの設計ツールを使用して Web サービスを生成できます。これは複数のリソースを統合したもので、既知のアーキテクチャ パターンや設計パターンに準拠した Web サービスを一貫した方法で迅速に構築するのに役立つように設計されています。詳細については、「Web Service Software Factory: Modeling Edition」(http://msdn.microsoft.com/en-us/library/cc487895.aspx、英語) を参照してください。
メッセージ コントラクトの設計とコントラクト ファーストの設計については、第 18 章「通信とメッセージ」を参照してください。レイヤー型アーキテクチャでの抽象化については、第 5 章「レイヤー型アプリケーションのガイドライン」を参照してください。

関連する設計パターン

次の表に示すように、主要なパターンはカテゴリに分類されます。各カテゴリの設計を行う際には、これらのパターンの使用を検討します。

カテゴリ	関連するパターン
通信	**Duplex**: サービスとクライアントの両方が相互に独立してメッセージを送信する双方向のメッセージ通信です。One-Way パターンと Request-Reply パターンのどちらが使用されるかは関係ありません。 **Fire and Forget**: 応答が期待されない場合に使用される、一方向のメッセージ通信メカニズムです。 **Reliable Sessions**: 送信元と送信先との間のエンド ツー エンドの信頼できるメッセージ送信です。エンドポイント間にある中間デバイスの数や種類は関係ありません。 **Request Response**: クライアントが送信したメッセージすべてに対して応答を受信することを期待する、双方向のメッセージ通信メカニズムです。
メッセージング チャネル	**Channel Adapter**: アプリケーションの API やデータにアクセスして、このデータに基づいてチャネルでメッセージを発行したり、メッセージを受信して、アプリケーション内で機能を呼び出したりすることができるコンポーネントです。 **Message Bus**: アプリケーションどうしを結び付けるミドルウェアを、アプリケーションがメッセージングを使用して連携できるようにする通信バスとして構成します。 **Messaging Bridge**: メッセージング システムどうしを結び付け、システム間のメッセージを複製するコンポーネントです。

	Point-to-Point Channel: 特定のメッセージを単独の受信者が受信するようにするように、メッセージをポイント ツー ポイント チャネルで送信します。 **Publish-Subscribe Channel**: 受信者の身元を知ることなく、メッセージの受信に関与するアプリケーションのみにメッセージを送信するためのメカニズムを作成します。
メッセージの構築	**Command Message**: コマンドをサポートするために使用するメッセージ構造です。 **Document Message**: アプリケーション間でドキュメントやデータ構造を信頼できる方法で転送するために使用する構造です。 **Event Message**: アプリケーション間の信頼できる非同期のイベント通知を提供する構造です。 **Request-Reply**: 要求と応答の送信に別個のチャネルを使用します。
メッセージ エンドポイント	**Competing Consumer**: 1 つのメッセージ キューに複数のコンシューマーを設定し、これらのコンシューマーがメッセージを処理する権利を奪い合うようにします。これにより、メッセージング クライアントは複数のメッセージを同時に処理できるようになります。 **Durable Subscriber**: 非接続型のシナリオでは、メッセージが確実に配信されるようにするため、メッセージは、保存されて、メッセージ チャネルへの接続時にクライアントからアクセスできるようになります。 **Idempotent Receiver**: サービスがメッセージを 1 回しか処理しないようにします。 **Message Dispatcher**: 複数のコンシューマーにメッセージを送信するコンポーネントです。 **Messaging Gateway**: メッセージ ベースの呼び出しを、残りのアプリケーション コードから分離するために 1 つのインターフェイスにカプセル化します。 **Messaging Mapper**: 受信メッセージの要求をビジネス オブジェクトに変換します。また、逆に、ビジネス オブジェクトを応答メッセージに変換する処理も行います。 **Polling Consumer**: メッセージの有無を確認するために定期的にチャネルをチェックするサービス コンシューマーです。 **Selective Consumer**: サービス コンシューマーでは、フィルターを使用して、特定の条件を満たすメッセージを受信します。 **Service Activator**: 非同期要求を受信してビジネス コンポーネントの操作を呼

	び出すサービスです。 **Transactional Client**: サービスと通信する際にトランザクションを実装できるクライアントです。
メッセージの保護	**Data Confidentiality**: メッセージ ベースの暗号化を使用してメッセージに含まれる機密データを保護します。 **Data Integrity**: メッセージが伝送中に改ざんされないようにします。 **Data Origin Authentication**: 高度な形態のデータ整合性を維持するために、メッセージの送信元を検証します。 **Exception Shielding**: 例外が発生したときに、サービスの内部実装に関する情報が公開されないようにします。 **Federation**: 複数のサービスやコンシューマーに分散した情報の統合ビューです。 **Replay Protection**: 攻撃者がメッセージを傍受して何度も実行するのを防ぐことにより、メッセージのべき等性を維持します。 **Validation**: メッセージのコンテンツとメッセージに含まれる値をチェックして、不適切な形式のコンテンツや悪意のあるコンテンツからサービスを保護します。
メッセージの ルーティング	**Aggregator**: 個々の関連するメッセージを収集および格納し、こうしたメッセージを結合し、結合によってできた 1 つのメッセージをさらに処理するために出力チャネルに発行するフィルターです。 **Content-Based Router**: メッセージの内容 (フィールドや指定されたフィールド値の有無など) に基づいて、各メッセージを適切なコンシューマーにルーティングします。 **Dynamic Router**: コンシューマーによって指定された条件や規則を評価してから、メッセージをコンシューマーに動的にルーティングするコンポーネントです。 **Message Broker (Hub and Spoke)**: 複数のアプリケーションと通信して複数の送信元からメッセージを受信し、適切な送信先を特定し、メッセージを適切なチャネルにルーティングする中核のコンポーネントです。 **Message Filter**: 一連の基準に基づいて望ましくないと判断されたメッセージがチャネル経由でコンシューマーに転送されるのを防ぎます。 **Process Manager**: ワークフローに含まれる複数の手順を通じてメッセージのルーティングを可能にするコンポーネントです。

メッセージの変換	**Canonical Data Mapper**: 共通のデータ形式を使用して、種類の異なる 2 つのデータ形式間の変換を実行します。 **Claim Check**: 必要に応じて永続的なストアからデータを取得します。 **Content Enricher**: メッセージに不足している情報を、外部データ ソースから取得して加えます。 **Content Filter**: メッセージから機密データを取り除きます。また、メッセージから不要なデータを取り除くことでネットワーク トラフィックを最小限に抑えます。 **Envelope Wrapper**: メッセージの保護、ルーティング、認証などに使用するヘッダー情報を含む、メッセージのラッパーです。 **Normalizer**: 組織でさまざまな形式のデータを使用する場合に、データを共通の交換形式に変換します。
REST	**Behavior**: 操作を実行するリソースに適用します。通常、このようなリソースでは、独自の状態を保持しておらず、POST 操作しかサポートしていません。 **Container**: エンティティ パターンを基盤として、入れ子になったリソースを動的に追加/更新する手段を提供します。 **Entity**: 読み取りは GET 操作で行えますが、変更は PUT 操作と DELETE 操作でしか行えないリソースです。 **Store**: PUT 操作でエントリを作成および更新できるようにします。 **Transaction**: トランザクション操作をサポートするリソースです。
サービス インターフェイス	**Façade**: 一連の操作に統一されたインターフェイスを実装します。これにより、簡略化されたインターフェイスが提供され、システム間の結合が緩和されます。 **Remote Façade**: 粒度の細かい操作に粒度の粗いインターフェイスを提供して、リモート サブシステムの一連の操作または処理への大まかな統一されたインターフェイスを作成します。これにより、サブシステムが使いやすくなり、またネットワーク経由の呼び出しが最小限に抑えられます。 **Service Interface**: 他のシステムがサービスとの通信に使用できる、プログラマティック インターフェイスです。
SOAP	**Data Contract**: サービス要求で渡されるデータ構造を定義するスキーマです。 **Fault Contracts**: サービス要求から返される可能性のあるエラーやフォールトを定義するスキーマです。 **Service Contract**: サービスが実行できる操作を定義するスキーマです。

Duplex パターンと Request Response パターンの詳細については、「サービス コントラクトの設計」(http://msdn.microsoft.com/ja-jp/library/ms733070.aspx) を参照してください。

Request-Reply パターンの詳細については、「Request-Reply」(http://www.eaipatterns.com/RequestReply.html、英語) を参照してください。

Command Message パターン、Document Message パターン、Event Message パターン、Durable Subscriber パターン、Idempotent Receiver パターン、Polling Consumer パターン、および Transactional Client パターンの詳細については、「SOA のメッセージングパターン (パート 1)」(http://msdn.microsoft.com/ja-jp/library/cc947720.aspx) を参照してください。

Data Confidentiality パターンと Data Origin Authentication パターンの詳細については、「Chapter 2: Message Protection Patterns」(http://msdn.microsoft.com/en-us/library/aa480573.aspx、英語) を参照してください。

Replay Detection パターン、Exception Shielding パターン、および Validation パターンの詳細については、「Chapter 5: Service Boundary Protection Patterns」(http://msdn.microsoft.com/en-us/library/aa480597.aspx、英語) を参照してください。

Claim Check パターン、Content Enricher パターン、Content Filter パターン、および Envelope Wrapper パターンの詳細については、「SOA のメッセージングパターン (パート 2)」(http://msdn.microsoft.com/ja-jp/library/cc947734.aspx) を参照してください。

Remote Façade パターンの詳細については、「P of EAA: Remote Facade」(http://martinfowler.com/eaaCatalog/remoteFacade.html、英語) を参照してください。

REST パターン (Behavior パターン、Container パターン、Entity パターンなど) の詳細については、「REST Patterns」(http://wiki.developer.mindtouch.com/REST/REST_Patterns、英語) を参照してください。

Aggregator パターン、Content-Based Router パターン、Publish-Subscribe Channel パターン、Message Bus パターン、および Point-to-Point Channel パターンの詳細については、「SOA のメッセージングパターン (パート 1)」(http://msdn.microsoft.com/ja-jp/library/cc947720.aspx) を参照してください。

関連情報

Web リソースに簡単にアクセスするには、http://www.microsoft.com/architectureguide (英語) でオンライン版の参考文献を参照してください。

- Microsoft .NET を使用したエンタープライズ ソリューション パターン
  (http://msdn.microsoft.com/ja-jp/library/ms998469.aspx)
- Web Service Security
  (http://msdn.microsoft.com/en-us/library/aa480545.aspx、英語)
- Improving Web Services Security: Scenarios and Implementation Guidance for WCF
  (http://www.codeplex.com/WCFSecurityGuide、英語)
- WS-* Specifications
  (http://www.soaspecs.com/ws-atomictransaction.php、英語)

10

コンポーネントのガイドライン

概要

コンポーネントでは、他の機能から切り離して配布およびインストールできる、特定の単位に含まれる一連の機能を分離する方法を提供します。この章では、コンポーネントの作成に関する一般的なガイドラインをいくつか紹介します。また、このガイドの他の章で取り上げているレイヤー型の手法で設計したアプリケーションの各レイヤーで一般的に使用するコンポーネントの種類についても説明します。ただし、コンポーネントを構築する手法は、アプリケーションの構造にかかわらず、全般的に適用できます。

コンポーネントの設計に関する一般的なガイドライン

アプリケーションで使用するコンポーネントを設計する際には、次の一般的なガイドラインを考慮します。

- **堅固なデザイン原理をコンポーネントのクラスに適用する**: 堅固な原理の概要は次のとおりです。
    - **単一の役割の原理**: 各クラスの役割は 1 つに絞る必要があります。
    - **開放/閉鎖原則**: クラスは、変更しなくても拡張できる必要があります。
    - **リスコフの置換原則**: サブタイプは基本型と置換できる必要があります。
    - **インターフェイス分離の原則**: クラス インターフェイスは、クライアント専用に設計して、粒度を細かくする必要があります。クラスでは、インターフェイスの要件が異なるさまざまなクライアントに別個のインターフェイスを公開する必要があります。
    - **依存関係の逆転の原則**: クラス間の依存関係を抽象化して、最初に低レベルのモジュールの設計をしなくてもトップダウン方式で設計できるようにする必要があります。抽象化は詳細に依存しないようにする必要がありますが、詳細は抽象化に依存する必要があります。

- **コンポーネントの結合性が高くなるように設計する**: 無関係の機能を追加したり、機能を混在させたりしてコンポーネントに過剰な機能を持たせないようにします。たとえば、ビジネス コンポーネントでは、データ アクセス ロジックとビジネス ロジックが混在しないようにします。機能がまとまっている場合は、レイヤーが物理的に離れていても、複数のコンポーネントを含むアセンブリを作成して、アプリケーションの適切なレイヤーにコンポーネントをインストールできます。
- **コンポーネントが他のコンポーネントの内部の詳細に依存しないようにする**: 各コンポーネントやオブジェクトでは、他のオブジェクトやコンポーネントのメソッドを呼び出す必要があります。また、このメソッドでは、要求の処理方法 (状況によっては、要求を適切なサブコンポーネントや他のコンポーネントにルーティングする方法) に関する情報を保持している必要があります。これにより、保守容易性と柔軟性の高いアプリケーションを開発できます。
- **コンポーネント間の通信方法を理解する**: これには、アプリケーションでサポートしなければならない配置シナリオを理解する必要があります。物理境界間やプロセス境界間の通信がサポートされる必要があるか、またはすべてのコンポーネントが同じプロセスで実行されるかどうかを判断することが必要です。
- **アプリケーション固有のロジックから抽象化された横断的なコードを維持する:** 横断的なコードとは、セキュリティ、通信、または運用管理 (ログ記録、インストルメンテーションなど) に関連するコードのことです。これらの機能を実装したコードをコンポーネントのロジックと混在させると、拡張や管理が困難な設計となる可能性があります。
- **コンポーネント ベースのアーキテクチャ スタイルの基本原理を適用する**: アーキテクチャの基本原理では、コンポーネントが再利用可能、置き換え可能、拡張可能で、カプセル化されていること、独立していること、およびコンテキストに特化していないことが必要であるとしています。コンポーネント ベースのアーキテクチャ スタイルに関する詳細については、第 3 章「アーキテクチャのパターンとスタイル」を参照してください。

レイヤー型アーキテクチャ アプリケーションのコンポーネントの分散

アプリケーションの各レイヤーには、レイヤーに機能を実装する一連のコンポーネントが含まれています。このようなコンポーネントは、再利用性を高めて、保守を簡略化するために、まとまりがあり疎結合されている必要があります。図 1 は、各レイヤーで一般的に使用されるコンポーネントの種類を示しています。

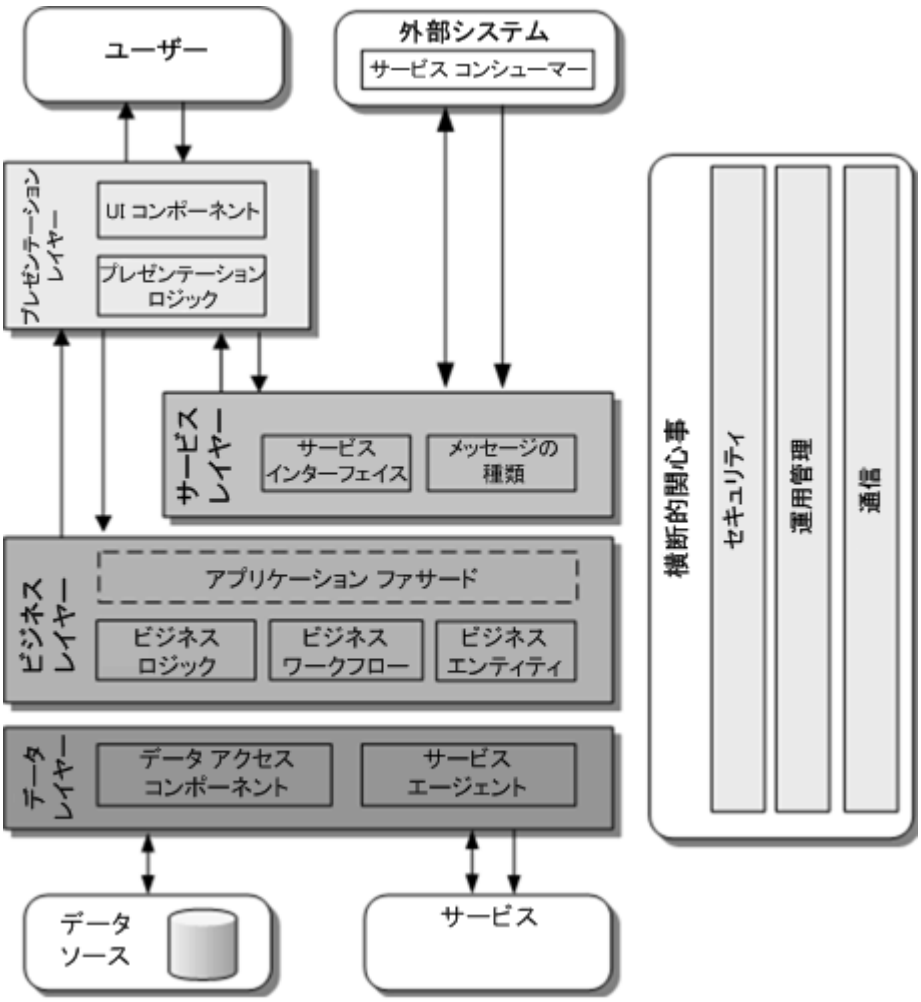


図 1

各レイヤーで一般的に使用されるコンポーネントの種類

次のセクションでは、図 1 で示したコンポーネントについて説明します。

プレゼンテーション レイヤーのコンポーネント

プレゼンテーション レイヤーのコンポーネントでは、ユーザーがアプリケーションを操作するために必要な機能を実装します。プレゼンテーション レイヤーで一般的に使用されるコンポーネントの種類には、次のようなものがあります。

- **ユーザー インターフェイス コンポーネント**: アプリケーションのユーザー インターフェイスは、ユーザー インターフェイス (UI) コンポーネントにカプセル化されています。このコンポーネントは、ユーザーに情報を表示したり、ユーザーからの入力を受け付けたりするのに使用するアプリケーションの視

覚的要素です。Separated Presentation パターンの実装で機能するように設計された UI コンポーネントは、View と呼ばれることもあります。多くの場合、UI コンポーネント特有の役割は、アプリケーションの基になるデータとロジックを表すインターフェイスを最適な方法でユーザーに提示することと、ユーザー入力ジェスチャを解釈して、基になるデータとアプリケーションの状態に入力が及ぼす影響を定義するプレゼンテーション ロジック コンポーネントにジェスチャを転送することです。また、UI コンポーネントには、ユーザー インターフェイスの実装に固有のロジックが含まれる場合もあります。ただし、アプリケーション ロジックが含まれていると、保守容易性と再利用性に影響を及ぼし、単体テストが行いにくくなるため、一般的に、UI コンポーネントにはアプリケーション ロジックをできる限り含めないようにする必要があります。

- **プレゼンテーション ロジック コンポーネント**: プレゼンテーション ロジックとは、特定のユーザー インターフェイスの実装に依存しない形で、アプリケーションの論理的な動作と構造を定義するアプリケーション コードのことです。プレゼンテーション ロジック コンポーネントは、アプリケーションのユース ケース (またはユーザー ストーリー) を実装することと、ユーザーが基になるアプリケーション ロジックとアプリケーションの状態を UI から独立した方法で操作できるようにすることに主にかかわっています。また、UI コンポーネントで簡単に使用できる形式で、ビジネス レイヤーのデータを整理することにもかかわっています。たとえば、複数のソースからデータを収集したり、より簡単に表示できるようにデータを変換したりします。プレゼンテーション ロジック コンポーネントは、さらに次の 2 つのカテゴリに分類できます。
    - **Presenter、Controller、Presentation Model、および ViewModel コンポーネント**: Separated Presentation パターンを実装するときに使用します。この種類のコンポーネントでは、多くの場合、プレゼンテーション レイヤーのプレゼンテーション ロジックをカプセル化します。再利用の機会とテスト容易性を最大限に高めるには、コンポーネントが特定の UI クラス、UI 要素、または UI コントロールに特化しないようにします。
    - **プレゼンテーション エンティティ コンポーネント**: このコンポーネントでは、ビジネス ロジックとビジネス データをカプセル化して、プレゼンテーション レイヤーの UI コンポーネントやプレゼンテーション ロジック コンポーネントで使いやすくします。たとえば、データ型を変換したり、いくつかのソースからデータを収集したりします。プレゼンテーション エンティティ コンポーネントは、プレゼンテーション ティアで直接使用されるビジネス レイヤーのビジネス エンティティである場合もあります。また、ビジネス エンティティ コンポーネントのサブセットを表し、アプリケーションのプレゼンテーション レイヤーをサポートするように設計されているものもあります。プレゼンテーション エンティティは、プレゼンテーション レイヤーにおけるデータの一

貫性と妥当性を確保するのに役立ちます。Separated Presentation パターンでは、このコンポーネントがモデルと呼ばれることもあります。

プレゼンテーション レイヤーの設計に関する詳細については、第 6 章「プレゼンテーション レイヤーのガイドライン」を参照してください。プレゼンテーション レイヤーのコンポーネントの設計に関する詳細については、第 11 章「プレゼンテーション レイヤーのコンポーネントの設計」を参照してください。

サービス レイヤーのコンポーネント

アプリケーションでは、クライアントやその他の使用しているシステムと通信するためにサービス レイヤーを公開することができます。サービス レイヤーのコンポーネントでは、通信チャネルでアプリケーションとメッセージをやり取りすることによって、他のクライアントやアプリケーションが、アプリケーションのビジネス ロジックにアクセスできるようにしたり、アプリケーションの機能を活用したりする方法を提供します。サービス レイヤーで一般的に使用されるコンポーネントの種類には、次のようなものがあります。

- **サービス インターフェイス**: サービスでは、すべての受信メッセージの送信先となるサービス インターフェイスを公開します。コントラクトは、サービスで特定のビジネス タスクが実行できるようにサービスとやり取りする必要がある一連のメッセージの定義で構成されています。サービス インターフェイスは、アプリケーションに実装されているビジネス ロジック (通常、ビジネス レイヤーのロジック) を潜在的なコンシューマーに公開するファサードと見なすことができます。
- **メッセージの種類**: サービス レイヤーでデータをやり取りする際、データ構造は、さまざまな種類の操作をサポートするメッセージ構造でラップされています。たとえば、ユーザーは Command Message、Document Message などの種類のメッセージを受け取ります。これらの種類のメッセージは、サービス コンシューマーとサービス プロバイダー間の通信のメッセージ コントラクトです。通常、サービス レイヤーでも、データ型とメッセージで使用されているデータ型を定義するデータ コントラクトを公開します。また、メッセージの種類に含まれているデータ型を内部のデータ型と分離します。内部のデータ型が外部のコンシューマーに公開されると、インターフェイスのバージョン管理に関する問題を引き起こすことがありますが、分離により、内部のデータ型が外部のコンシューマーに公開されることを防げます。

通信とメッセージに関する詳細については、第 18 章「通信とメッセージ」を参照してください。

ビジネス レイヤーのコンポーネント

ビジネス レイヤーのコンポーネントでは、システムの主要な機能を実装し、関連するビジネス ロジックをカプセル化します。ビジネス レイヤーで一般的に使用されるコンポーネントの種類には、次のようなものがあります。

- **アプリケーション ファサード**: 一般的に、このオプション コンポーネントでは、複数のビジネス操作をビジネス ロジックが使用しやすくなる単一のビジネス操作に統合することによって、簡略化されたインターフェイスをビジネス ロジック コンポーネントに提供します。また、外部の呼び出し元では、ビジネス コンポーネントの詳細と、コンポーネント間の関係を把握する必要がないので、依存関係が軽減されます。
- **ビジネス ロジック コンポーネント**: ビジネス ロジックは、アプリケーション データの取得、処理、変換、および管理、ビジネス ルールとビジネス ポリシーの適用、そしてデータの一貫性と妥当性の確保に関連するアプリケーション ロジックです。再利用の機会を最大限に高めるには、ビジネス ロジック コンポーネントには、ユース ケースまたはユーザー ストーリー固有の動作やアプリケーション ロジックを含めないようにします。ビジネス ロジック コンポーネントは、さらに次の 2 つのカテゴリに分類できます。
    - **ビジネス ワークフロー コンポーネント**: UI コンポーネントで、ユーザーからデータを収集してビジネス レイヤーに渡したら、アプリケーションでは、そのデータを使用してビジネス プロセスを実行できます。多くのビジネス プロセスには、正しい順序で実行する必要がある複数の手順が含まれており、ビジネス プロセス間ではオーケストレーション経由で通信が行われる場合があります。ビジネス ワークフロー コンポーネントでは、実行時間が長い多段階のビジネス プロセスが定義および調整され、このコンポーネントはビジネス プロセス管理ツールを使用して実装できます。また、ワークフロー コンポーネントで操作をインスタンス化して実行するビジネス プロセス コンポーネントと連携します。ビジネス ワークフロー コンポーネントの詳細については、第 14 章「ワークフロー コンポーネントの設計」を参照してください。
    - **ビジネス エンティティ コンポーネント**: ビジネス エンティティ (一般的に言うビジネス オブジェクト) では、顧客や注文など、アプリケーション内で実際の要素を表すのに必要なビジネス ロジックとビジネス データをカプセル化します。ビジネス エンティティでは、データ値をプロパティで格納および公開し、アプリケーションで使用するビジネス データを保持して管理し、ビジネス データと関連機能へのプログラムによるステートフルなアクセスが提供されます。また、一貫性を確保して、ビジネス ルールと動作を実装には、エンティティに含まれるデータを検証して、

ビジネス ロジックをカプセル化します。ビジネス エンティティ コンポーネントの詳細については、第 13 章「ビジネス エンティティの設計」を参照してください。

多くの場合、ビジネス エンティティは、ビジネス レイヤーとデータ レイヤーの両方のコンポーネントとサービスからアクセスできなければなりません。たとえば、ビジネス エンティティはデータ ソースにマップして、ビジネス コンポーネントからアクセスすることができます。レイヤーが同じ物理ティアに配置されている場合は、直接参照によってビジネス エンティティを共有できます。ただし、その際にも、ビジネス ロジックはデータ アクセス ロジックから分離する必要があります。そのためには、ビジネス エンティティを、ビジネス サービス アセンブリとデータ サービス アセンブリの両方で共有できる別個のアセンブリに移動します。これは、Dependency Inversion パターンを使用するのと同じことです。このパターンでは、ビジネス レイヤーとデータ レイヤーの両方が、共有のコントラクトとしてビジネス エンティティに依存するように、ビジネス レイヤーとデータ レイヤーからビジネス エンティティを分離しています。

ビジネス レイヤーの設計に関する詳細については、第 7 章「ビジネス レイヤーのガイドライン」を参照してください。ビジネス レイヤーのコンポーネントの設計に関する詳細については、第 12 章「ビジネス レイヤーのコンポーネントの設計」を参照してください。ビジネス エンティティ コンポーネントの設計に関する詳細については、第 13 章「ビジネス エンティティの設計」を参照してください。ワークフロー コンポーネントの設計に関する詳細については、第 14 章「ワークフロー コンポーネントの設計」を参照してください。

データ レイヤーのコンポーネント

データ レイヤーのコンポーネントでは、システムの境界内でホストされているデータと他のネットワークに対応したシステムで公開されているデータへのアクセスを提供します。データ レイヤーで一般的に使用されるコンポーネントの種類には、次のようなものがあります。

- **データ アクセス コンポーネント**: このコンポーネントでは、基になるデータ ストアにアクセスするために必要なロジックを抽象化します。多くのデータ アクセスのタスクには、独立した再利用可能なヘルパー コンポーネントとして抽出および実装できる共通のロジックか、フレームワークによる適切なサポートが必要です。これにより、データ アクセス コンポーネントの複雑さが軽減し、ロジックを一元化できるので、保守が容易になります。どのコンポーネントにも特化していない、データ レイヤーのコンポーネント間に共通のその他のタスクは、別個のユーティリティ コンポーネントとして実装できます。ヘルパー コンポーネントとユーティリティ コンポーネントは、多くの場合、他のアプリケーションで簡単に再利用できるように、ライブラリまたはフレームワークにカプセル化されます。

- **サービス エージェント**: ビジネス コンポーネントで外部サービスの機能を使用する必要がある場合は、そのサービスとの通信の動作を管理するコードを実装しなければならないことがあります。サービス エージェントでは、アプリケーションからさまざまなサービスを呼び出すという特質を分離します。また、キャッシュ、オフライン サポート、サービスが公開するデータの形式とアプリケーションが要求する形式との基本的なマッピングなどの追加サービスを提供する場合もあります。

データ レイヤーの設計に関する詳細については、第 8 章「データ レイヤーのガイドライン」を参照してください。データ コンポーネントの設計に関する詳細については、第 15 章「データ コンポーネントの設計」を参照してください。

横断的なコンポーネント

アプリケーションのコードで実行されるタスクの多くは、複数のレイヤーで必要になります。横断的なコンポーネントでは、すべてのレイヤーのコンポーネントからアクセスできる特殊な種類の機能を実装します。一般的な種類の横断的なコンポーネントには、次のようなものがあります。

- **セキュリティを実装するコンポーネント**: 認証、承認、検証などを実行するコンポーネントです。
- **運用管理タスクを実装するコンポーネント**: 例外ハンドル ポリシー、ログ記録、パフォーマンス カウンター、構成、トレースなどの機能を実装するコンポーネントです。
- **通信を実装するコンポーネント**: 他のサービスやアプリケーションと通信するコンポーネントなどです。

横断的関心事の管理に関する詳細については、第 17 章「横断的関心事」を参照してください。

関連する設計パターン

次の表に示すように、コンポーネントの主要なパターンはカテゴリに分類されます。各カテゴリの設計を行う際には、これらのパターンの使用を検討します。

カテゴリ	関連するパターン
ビジネス コンポーネント	**Application Façade**: 動作を一元化および集約して均一のサービス レイヤーを提供します。 **Chain of Responsibility**: 複数のオブジェクトで要求を処理できるようにし

	て、要求の送信者と受信者が一対一で結合されないようにします。 **Command**: 要求処理を、共通の実行インターフェイスを備えた別個のコマンド オブジェクトにカプセル化します。
ビジネス エンティティ	**Domain Model**: ドメイン内のエンティティとエンティティ間の関係を表す一連のビジネス オブジェクトです。 **Entity Translator**: 要求ではデータ コントラクトをビジネス エンティティに変換し、応答では逆の変換を行うオブジェクトです。 **Table Module**: データベースのテーブルまたはビュー内のすべての行のビジネス ロジックを処理する、1 つのコンポーネントです。
プレゼンテーション エンティティ	**Entity Translator**: 要求ではメッセージ データ型をビジネス型に変換し、応答では逆の変換を行うオブジェクトです。
プレゼンテーション ロジック	**Application Controller**: 画面のナビゲーションとアプリケーションのフローを処理するための一元化された場所を実装します。 **Model-View-Controller**: UI コードを、Model (データ)、View (インターフェイス)、Controller (処理ロジック) という 3 つの別個の単位に、View に重点を置きながら分離します。このパターンには、View が Model と通信する方法を定義する、Passive View および Supervising Presenter という 2 種類のパターンがあります。 **Model-View-ViewModel**: Command パターンを使用して View から ViewModel に通信する、Presentation Model パターンの一種です。 **Model-View-Presenter**: ユーザーの入力の処理、Presenter オブジェクトへの制御の転送という View の役割を維持しながら、要求処理を 3 つの別個の役割に分離します。 **Passive View**: コントローラーでユーザーの入力を処理できるようにして、View の更新という役割を維持しながら、View の役割を最小限に抑えます。 **Presentation Model**: View のロジックと状態をすべて View の外部に移動して、データ バインドとテンプレートを通じて View を表示します。 **Supervising Presenter** (または Supervising Controller): Model-View-Controller パターンの一種で、コントローラーが複雑なロジックを処理する (具体的には View 間の調整をする) 一方で、View 固有の簡単なロジックは View で処理されます。
サービス インターフェイス	**Façade**: 一連の操作に統一されたインターフェイスを実装します。これにより、

	簡略化されたインターフェイスが提供され、システム間の結合が緩和されます。 **Remote Façade**: 粒度の細かい操作に粒度の粗いインターフェイスを提供して、リモート サブシステムの一連の操作または処理への大まかな統一されたインターフェイスを作成します。これにより、サブシステムが使いやすくなり、ネットワーク経由の呼び出しを最小限に抑えられます。 **Service Interface**: 他のシステムがサービスとの通信に使用できる、プログラマティック インターフェイスです。
ワークフロー	**Data-Driven Workflow**: ワークフローまたはシステムのデータの値に基づいてシーケンスが決定されるタスクを含むワークフローです。 **Human Workflow**: ユーザーが手動で実行するタスクを含むワークフローです。 **Sequential Workflow**: 先行するタスクが完了してから次のタスクが開始されるというシーケンスに従うタスクを含むワークフローです。 **State-Driven Workflow**: システムの状態によってシーケンスが決定されるタスクを含むワークフローです。

Façade パターンの詳細については、Erich Gamma、Richard Helm、Ralph Johnson、John Vlissides 共著『オブジェクト指向における再利用のためのデザイン パターン』(ソフトバンク クリエイティブ、1999 年) の第 4 章「構造に関するパターン」を参照してください。

Chain of Responsibility パターンの詳細については、「開放/閉鎖原則」(http://msdn.microsoft.com/ja-jp/magazine/cc546578.aspx) を参照してください。

Command パターンの詳細については、Erich Gamma、Richard Helm、Ralph Johnson、John Vlissides 共著『オブジェクト指向における再利用のためのデザイン パターン』(ソフトバンク クリエイティブ、1999 年) の第 5 章「振る舞いに関するパターン」を参照してください。

Entity Translator パターンの詳細については、「Useful Patterns for Services」(http://msdn.microsoft.com/en-us/library/cc304800.aspx、英語) を参照してください。

データ ドリブン ワークフロー、ヒューマン ワークフロー、シーケンシャル ワークフロー、およびステート ドリブン ワークフローの詳細については、「Windows Workflow Foundation の概要」(http://msdn.microsoft.com/ja-jp/library/ms734631.aspx) と「Workflow Patterns」(http://www.workflowpatterns.com/、英語) を参照してください。

Application Controller と Model-View-Controller (MVC) の各パターンの詳細については、Martin Fowler 著『エンタープライズ アプリケーション アーキテクチャ パターン』 (翔泳社、2005 年) を参照するか、http://martinfowler.com/eaaCatalog (英語) を参照してください。

Supervising Presenter パターンと Presentation Model パターンの詳細については、「Patterns in the Composite Application Library」(http://msdn.microsoft.com/en-us/library/dd458924.aspx、英語) を参照してください。

Remote Façade パターンの詳細については、「P of EAA: Remote Facade」(http://martinfowler.com/eaaCatalog/remoteFacade.html、英語) を参照してください。

patterns & practices のサービス

マイクロソフトの patterns & practices グループが提供している関連サービスの詳細については、次のリソースを参照してください。

- Composite Client Application Guidance (http://msdn.microsoft.com/en-us/library/cc707819.aspx、英語)
- Enterprise Library (http://msdn.microsoft.com/en-us/library/cc467894.aspx、英語)
- Smart Client Software Factory (http://msdn.microsoft.com/en-us/library/aa480482.aspx、英語)
- Unity Application Block (依存関係の挿入のメカニズム) (http://msdn.microsoft.com/en-us/library/dd203101.aspx、英語)
- Web Client Software Factory (http://msdn.microsoft.com/en-us/library/bb264518.aspx、英語)

関連情報

Web リソースに簡単にアクセスするには、http://www.microsoft.com/architectureguide (英語) でオンライン版の参考文献を参照してください。

- Integration Patterns
  (http://msdn.microsoft.com/en-us/library/ms978729.aspx、英語)
- Robert C. Martin、Micah Martin 共著『Agile Principles, Patterns, and Practices in C#』(Prentice Hall、2006 年)
- User Interface Control Guidelines
  (http://msdn.microsoft.com/en-us/library/bb158574.aspx、英語)

11

プレゼンテーション レイヤーのコンポーネントの設計

概要

この章では、プレゼンテーション レイヤーに含まれるユーザー インターフェイス コンポーネントとプレゼンテーション ロジック コンポーネントを設計する際に従う必要がある手順について説明します。プレゼンテーション レイヤー自体の設計にかかわる段階もあれば、構築するコンポーネントの種類に重点を置いている段階もあります。まず、UI の要件を理解し、適切なテクノロジを選択する必要があります。適切なテクノロジを選択したら、プレゼンテーション ロジックとデータを UI コントロールにバインドする方法を決定できます。また、UI でのエラー ハンドルと検証に関する要件も理解する必要があります。この章の以降のセクションでは、プレゼンテーション レイヤーのコンポーネントを設計する手順について詳しく説明します。

手順 1 – UI の要件を理解する

UI の要件を理解することは、UI の種類、および UI の実装に使用するテクノロジとコントロールの種類を判断するうえで重要です。UI の要件は、アプリケーションでサポートする機能とユーザーの期待値によって決まります。最初に、アプリケーションのユーザーを特定し、ユーザーがアプリケーションを使用する際に達成することを期待する目標とタスクを理解します。タスクや操作の順序には特に注意する必要があります。ユーザーが期待しているのは構造化され手順に従って作業するユーザー エクスペリエンスなのか、それとも複数のタスクを同時に開始できるアドホックな構造化されていないエクスペリエンスなのかを判断します。また、この作業の一環として、ユーザーが必要とする情報と、その期待している形式も判断する必要があります。ユーザーがアプリケーションを操作する環境に

ついて理解するために、フィールド調査を実施することもできます。また、現在のユーザー エクスペリエンスのレベルを考慮し、これを UI で必要なユーザー エクスペリエンスと比較して、ユーザー エクスペリエンスが論理的で直感的なものになるようにします。これらの要素を考慮することで、ユーザー中心の設計を作成できます。

テクノロジの選択に大きな影響を与える要因の 1 つが、UI に必要な機能です。豊富な機能やユーザー操作を UI で公開する必要があるかどうか、UI が高い応答性を備えている必要があるかどうか、または UI でグラフィックやアニメーションをサポートする必要があるかどうかを特定します。また、ローカリゼーションの観点から、日付、時刻、通貨などのデータに関して、データ型、形式、およびプレゼンテーション形式の要件についても考慮します。さらに、ユーザーが実行時にレイアウトやスタイルを変更できるようにするなど、アプリケーションのパーソナル化に関する要件を特定します。

UI を直感的で使いやすくするには、インターフェイスのレイアウト方法や構成方法、およびユーザーがアプリケーションの UI を操作する方法を検討します。これは、適切なコントロールと UI テクノロジを選択するのに役立ちます。サポートする必要がある物理的な表示の要件 (画面のサイズ、解像度など) を理解して、アクセシビリティの要件 (テキストやボタンの拡大、インク入力、その他の特殊機能など) を判断します。関連する情報を UI のセクションにグループ化する方法、インターフェイスの競合やあいまいさを回避する方法、および重要な要素を強調する方法を決定します。ナビゲーションのコントロール、検索機能、明確な名前が付けられたセクション、サイト マップなどの機能を必要に応じて使用して、ユーザーがアプリケーションで情報をすばやく簡単に見つけられるようにする方法を特定します。

手順 2 – 必要な UI の種類を決定する

UI の要件に基づいて、アプリケーションで使用する UI の種類を決定できます。UI にはさまざまな種類があり、それぞれの種類に長所と短所があります。複数の種類の UI を使用して UI の要件を満たせると判明することは、よくあります。また、すべての UI 要件を完全に満たす UI の種類が 1 つもないこともあります。この場合は、ビジネス ロジックの共有セットに加えて、別の種類の UI を作成することを検討します。たとえば、コール センター アプリケーションを作成し、このアプリケーションで顧客向けセルフ ヘルプ機能の一部を Web やモバイル デバイスで公開する場合があります。

モバイル アプリケーション: シン クライアント アプリケーションまたはリッチ クライアント アプリケーションとして開発できます。リッチ クライアント モバイル アプリケーションでは、非接続型のシナリオや不定期に接続するシナリオをサポートできます。Web またはシン クライアント モバイル アプリケーションでは、接続型のシナリオだけをサポートします。モバイル アプリケーションの設計時には、デバイスのリソースも制約事項になることがあります。

リッチ クライアント アプリケーション: 通常は、さまざまなコントロールを使用してデータを表示するグラフィカル ユーザー インターフェイスを備えた、スタンドアロン アプリケーションまたはネットワーク アプリケーションです。ローカル ユーザーが使用できるようにデスクトップやラップトップ コンピューターに配置されます。リッチ クライアント アプリケーションはクライアント コンピューターで実行されるので、非接続型のシナリオと不定期に接続するシナリオに適しています。リッチ クライアント UI が適しているのは、UI で豊富な機能や豊富なユーザー操作をサポートしたり、非常に動的で応答性の高いユーザー エクスペリエンスを提供したりする必要がある場合です。また、アプリケーションが接続型と非接続型の両方のシナリオで動作したり、クライアント コンピューターのローカルのシステム リソースを使用したり、または同じコンピューター上の他のアプリケーションと統合したりする必要がある場合にも適しています。

リッチ インターネット アプリケーション (RIA): 通常は、ブラウザー内で実行される機能豊富なグラフィカル ユーザー インターフェイスを備えた Web アプリケーションです。一般的に、RIA は接続型のシナリオに使用します。RIA が適しているのは、UI で応答性の高い動的なユーザー エクスペリエンスをサポートしたりストリーミング メディアを使用する必要がある場合と、UI からさまざまなデバイスやプラットフォームに幅広くアクセスできる必要がある場合です。RIA ではクライアント コンピューターの処理能力を利用できますが、他のローカル システム リソース (Web カメラなど) や他のクライアント アプリケーション (Microsoft Office など) と簡単にやり取りすることはできません。

Web アプリケーション: 接続型のシナリオをサポートし、さまざまなオペレーティング システムやプラットフォームで実行している多種多様なブラウザーをサポートできます。Web アプリケーションが適しているのは、UI が標準に準拠し、最大限に幅広いデバイスとプラットフォームにアクセスでき、接続型のシナリオのみで機能する必要がある場合です。また、コンテンツが Web 検索エンジンで検索できるアプリケーションにも適しています。

コンソール ベースのアプリケーション: 代替テキストのみのユーザー エクスペリエンスが提供され、通常はコマンド シェル、PowerShell などのコマンド シェルで実行されます。コンソール ベースのアプリケーションは管理作業や開発作業に最適な形態なので、レイヤー型アプリケーションの設計に含まれることはほとんどありません。

手順 3 – UI のテクノロジを選択する

UI コンポーネントの UI の種類を特定したら、適切なテクノロジを選択する必要があります。一般に、選択するテクノロジは、手順 2 で選択した UI の種類によって異なります。次のセクションでは、UI の種類ごとに適切なテクノロジの一部を紹介します。

モバイル クライアントのユーザー インターフェイスは、次のプレゼンテーション テクノロジを使用して実装できます。

- **Microsoft .NET Compact Framework**: モバイル デバイス専用に設計された Microsoft .NET Framework のサブセットです。常時ネットワーク接続が利用できない可能性のあるデバイスで実行する必要があるモバイル アプリケーションには、このテクノロジを使用します。
- **ASP.NET for Mobile**: モバイル デバイス専用に設計された、ASP.NET のサブセットです。ASP.NET for Mobile アプリケーションは、インターネット インフォメーション サービス (IIS) サーバーでホストできます。幅広いデバイスとプラットフォームをサポートする必要があり、常時ネットワーク接続を利用できるモバイル Web アプリケーションには、このテクノロジを使用します。
- **Silverlight for Mobile**: この Silverlight クライアントのサブセットでは、Silverlight プラグインがモバイル デバイスにインストールされている必要があります。既存の Silverlight アプリケーションをモバイル デバイスに移植するために、または他のテクノロジを使用して作成できる UI よりも機能が豊富な UI を作成する必要がある場合は、このテクノロジを使用します (現時点ではこのテクノロジのリリースは発表されていますが、リリースされていません)。

リッチ クライアントのユーザー インターフェイスは、次のプレゼンテーション テクノロジを使用して実装できます。

- **Windows Presentation Foundation (WPF)**: WPF アプリケーションでは、より高度なグラフィック機能がサポートされます。たとえば、2D と 3D のグラフィックス、画面解像度への依存からの解放、高度なドキュメントと文字体裁のサポート、タイムラインを含むアニメーション、ストリーミング配信されるオーディオとビデオ、ベクター ベースのグラフィックなどです。WPF では、Extensible Application Markup Language (XAML) を使用して、UI、データ バインド、およびイベントを定義します。WPF には、高度なデータ バインド機能とテンプレート機能も用意されています。WPF アプリケーションでは、UI の視覚的要素と基になる制御ロジックが分離しているので、開発者とデザイナーの連携をサポートします。開発者はビジネス ロジックに集中して、デザイナーは外観に集中できるようになります。リッチ メディア ベースの対話型ユーザー インターフェイスを作成する場合は、このテクノロジを使用します。
- **Windows フォーム**: Windows Forms は、.NET Framework 1.0 から同梱されており、基幹業務スタイルのアプリケーションに最適です。Windows Presentation Foundation (WPF) を使用できる場合でも、Windows フォームの技術的専門知識があるときまたはアプリケーションのリッチ メディアや操作の要件が特にないときは、Windows フォームは UI 設計に最適です。
- **WPF ユーザー コントロールを使用する Windows フォーム**: WPF コントロールで提供される、より強力な UI 機能を使用できるようになります。WPF を既存の Windows フォーム アプリケーションに

追加したり、場合によっては、徐々に完全な WPF 実装を適用する手段として追加することもできます。この手法を使用して、リッチ メディアと対話の機能を既存のアプリケーションに追加します。ただし、WPF コントロールが最適な状態で動作するには、高性能なクライアント コンピューターが必要になる場合が多いことに注意してください。

- **Windows フォーム ユーザー コントロールを使用する WPF**: WPF では提供されない機能を備えた Windows フォーム コントロールによって、WPF を補完できるようになります。WindowsFormsIntegration アセンブリで提供される WindowsFormsHost コントロールを使用して、UI に Windows フォーム コントロールを追加できます。Windows フォーム コントロールを WPF の UI で使用する必要がある場合は、この手法を使用します。ただし、重複するコントロール、インターフェイスのフォーカス、およびさまざまなテクノロジで使用されるレンダリング技法に関する制約と問題があることに注意してください。
- **WPF を使用する XAML ブラウザー アプリケーション (XBAP)**: このテクノロジでは、Windows で実行されている Microsoft Internet Explorer または Mozilla Firefox で、サンドボックス化された WPF アプリケーションをホストします。Silverlight とは異なり、完全な WPF フレームワークを活用できますが、部分的に信頼されたサンドボックスからシステム リソースへのアクセスはいくらか制限されています。XBAP では、クライアント デスクトップに Windows Vista がインストールされているか、.NET Framework 3.5 と XBAP ブラウザー プラグインの両方がインストールされている必要があります。XBAP が適しているのは、Web に配置する既存の WPF アプリケーションがある場合、または Silverlight では使用できない WPF の豊富な視覚表現と UI の機能を活用する場合です。

リッチ インターネット アプリケーションのユーザー インターフェイスは、次のプレゼンテーション テクノロジを使用して実装できます。

- **Silverlight**: さまざまなプラットフォームやブラウザーで動作する、ブラウザー用に最適化された WPF のサブセットです。XBAP に比べて Silverlight は軽量で、短時間でインストールできます。Silverlight は、リソースの使用量が少なく、さまざまなプラットフォームでサポートされるので、高品質な WPF グラフィックのサポートが不要なグラフィカル アプリケーションや、アプリケーションをクライアントにインストールする必要がないグラフィカル アプリケーションに適しています。
- **AJAX を使用する Silverlight**: Silverlight では、Asynchronous JavaScript and XML (AJAX) がネイティブにサポートされ、そのオブジェクト モデルが Web ページに含まれる JavaScript に公開されています。この機能を使用すると、Web ページのコンポーネントと Silverlight アプリケーションの間で通信できるようになります。

Web アプリケーションのユーザー インターフェイスは、次のプレゼンテーション テクノロジを使用して実装できます。

- **ASP.NET の Web フォーム**: これは、.NET Web アプリケーション向けの UI を設計および実装する基本的なテクノロジです。ASP.NET の Web フォーム アプリケーションは Web サーバーにのみインストールする必要があり、クライアント デスクトップにコンポーネントをインストールする必要はありません。このセクションで説明している AJAX、Silverlight、MVC、または Dynamic Data の追加機能を必要としない Web アプリケーションには、このテクノロジを使用します。
- **AJAX を使用する ASP.NET の Web フォーム**: サーバーとクライアント間の要求を非同期に処理して応答性を向上し、より優れたユーザー エクスペリエンスを提供し、サーバーへのポスト バック数を削減するには、AJAX と ASP.NET の Web フォームを併用します。AJAX は、.NET Framework 3.5 以降で ASP.NET に不可欠な要素です。
- **Silverlight を使用する ASP.NET の Web フォーム**: 既存の ASP.NET アプリケーションがあれば、Silverlight コントロールを使用して豊富な視覚表現と UI の機能を備えたユーザー エクスペリエンスを提供して、Silverlight アプリケーションをゼロから作成することを回避できます。これは、Silverlight のリッチ メディア コンテンツを既存の Web アプリケーションに追加するのに適した手法です。Silverlight コントロールとそのコンテナーである Web ページの間では、JavaScript を使用してクライアント上で通信できます。
- **ASP.NET の MVC**: このテクノロジを使用すると、ASP.NET を使用して Model-View-Controller (MVC) パターンに基づくアプリケーションを構築できます。テスト駆動開発をサポートし、UI 処理と UI レンダリングの間における懸念事項を明確に分離する必要がある場合は、このテクノロジを使用します。この手法を使用すると、整った HTML を作成して、プレゼンテーション情報とロジック コードが混在しないようにすることができます。
- **ASP.NET の Dynamic Data**: このテクノロジを使用すると、統合言語クエリ (LINQ) to Entities データ モデルを利用するデータ ドリブン ASP.NET アプリケーションを作成できます。基幹業務 (LOB) スタイルの、単純なスキャフォールディングに基づくデータ ドリブン アプリケーションの迅速な開発モデルが必要な場合は、ASP.NET の Dynamic Data が適しています。

コンソール ベースのユーザー インターフェイスは、次のプレゼンテーション テクノロジを使用して実装できます。

- **コンソール アプリケーション**: コマンド シェルから実行でき、標準的な出力コンソールとエラー コンソールに出力を生成できるテキストのみのアプリケーションです。このようなアプリケーションは、多くの場合、呼び出し時のすべての入力を受け取って無人で実行するよう構築されています。
- **PowerShell コマンドレット**: PowerShell は、システムとアプリケーションの管理作業を包括的に制御して自動化するための、コマンド ライン シェル兼スクリプト環境です。コマンドレットは、PowerShell 言語といっそう緊密に統合されたエクスペリエンスが提供されるアプリケーション固有の PowerShell 環境の拡張機能です。

ここまでのセクションで説明したテクノロジの詳細については、付録 B「プレゼンテーション テクノロジ」を参照してください。

手順 4 – プレゼンテーション レイヤーのコンポーネントを設計する

UI に適した実装テクノロジを選択したら、次は UI コンポーネントとプレゼンテーション ロジック コンポーネントを設計します。プレゼンテーション レイヤーでは、次の種類のコンポーネントを使用できます。

- ユーザー インターフェイス コンポーネント
- プレゼンテーション ロジック コンポーネント
- Presentation Model コンポーネント

これらのコンポーネントでは、プレゼンテーション レイヤーにおける懸念事項の分離をサポートします。また、多くの場合はこれらのコンポーネントを使用して、UI 処理を Model、View、および Controller/Presenter の 3 つの異なる役割に分割することで MVP (Model-View-Presenter)、MVC (Model-View-Controller) などの Separated Presentation パターンを実装します。プレゼンテーション レイヤーの懸念事項をこのように分離すると、保守容易性とテスト容易性が向上し、再利用の機会が拡大します。Dependency Injection などの抽象パターンを使用することでも、プレゼンテーション ロジックを簡単にテストできるようになります。
コンポーネントの設計に関する一般的な考慮事項とアプリケーションのすべてのレイヤーで一般的に使用するコンポーネントの詳細については、第 10 章「コンポーネントのガイドライン」を参照してください。

ユーザー インターフェイス コンポーネント

UI コンポーネントは、ユーザーに情報を表示したり、ユーザーからの入力を受け付けたりするアプリケーションの視覚的要素です。Separated Presentation パターンでは、一般に View と呼ばれます。UI コンポーネントを設計する際には、次のガイドラインを考慮します。

- ページやウィンドウを独立したユーザー コントロールに分割して、ユーザー コントロールがなるべく複雑にならないようにし、再利用できるようにすることを検討します。適切な UI コンポーネントを選択し、UI で使用するコントロールのデータ バインド機能を利用します。
- ユーザー コントロールやページの継承階層を回避して、コードを再利用できるようにします。継承よりも構成を優先し、再利用可能なプレゼンテーション ロジック コンポーネントを作成することを検討します。
- 特殊な表示やデータの収集で必要にならない限り、カスタム コントロールを作成しないようにします。標準的なコントロールでは UI 要件を満たすことができない場合は、独自のカスタム コントロールの作成を決断する前に、コントロールのツールキットを購入することを検討します。カスタム コントロールを作成する必要がある場合は、できる限り、新しいコントロールを作成せずに既存のコントロールを拡張します。既存のコントロールを拡張する際には、既存のコントロールを継承するのではなく、動作をコントロールにアタッチすることを検討します。また、開発者がカスタム コントロールを簡単に使用できるように、デザイナーにカスタム コントロールのサポートを実装することを検討します。

プレゼンテーション ロジック コンポーネント

プレゼンテーション ロジック コンポーネントでは、視覚表現以外のユーザー インターフェイスの側面を処理します。この処理には、多くの場合、検証、ユーザー操作への応答、UI コンポーネント間の通信、およびユーザー操作のオーケストレーションが含まれます。プレゼンテーション ロジック コンポーネントは、必ずしも必要なものではありません。プレゼンテーション レイヤーで UI コンポーネントから分離する必要がある大規模な処理を実行する場合か、プレゼンテーション ロジックの単体テストの機会を拡大する必要がある場合にのみ、プレゼンテーション ロジック コンポーネントを作成します。プレゼンテーション ロジック コンポーネントを設計する際には、次のガイドラインを考慮します。

- UI で複雑な処理を実行したり、他のレイヤーと通信したりする必要がある場合は、プレゼンテーション ロジック コンポーネントを使用して、この処理を UI コンポーネントから分離することを検討します。

- プレゼンテーション ロジック コンポーネントを使用して、UI に関連する (ただしその UI 固有ではない) 状態を格納します。入力とデータ検証以外のビジネス ロジックおよびビジネス ルールは、プレゼンテーション ロジック コンポーネントに実装しないようにします。また、レンダリング ロジックや UI 固有のロジックもプレゼンテーション ロジック コンポーネントに実装しないようにします。
- プレゼンテーション ロジック コンポーネントを使用して、アプリケーションが障害やエラーから回復した後にユーザー インターフェイスで一貫性のある状態が保たれるようにします。
- UI で複雑なワークフローをサポートする必要がある場合は、ワークフロー システム (Windows Workflow Foundation など) またはアプリケーションのビジネス レイヤーのカスタム メカニズムを使用する個別のワークフロー コンポーネントを作成することを検討します。詳細については、第 14 章「ワークフロー コンポーネントの設計」を参照してください。

Presentation Model コンポーネント

Presentation Model コンポーネントでは、プレゼンテーション レイヤーの UI コンポーネントとプレゼンテーション ロジック コンポーネントで簡単に使用できる形式でビジネス レイヤーのデータを表します。一般に、このコンポーネントはデータを表すので、そのデータを収集するために、データ アクセス コンポーネントや場合によってはビジネス レイヤーのコンポーネントを使用します。このコンポーネントでビジネス ロジックをカプセル化している場合、通常、このコンポーネントはプレゼンテーション エンティティと呼ばれます。Presentation Model コンポーネントでは、たとえば、複数のソースからのデータの集計、より簡単に表示するためのデータ転送、および検証ロジックの実装を行ったり、プレゼンテーション レイヤー内のビジネス ロジックと状態を表せるようにしたりします。これらのコンポーネントは、一般に MVP、MVC などの Separated Presentation パターンを実装するのに使用します。Presentation Model コンポーネントを設計する際には、次のガイドラインを考慮します。

- Presentation Model コンポーネントが必要かどうかを判断します。通常、表示するデータや形式がプレゼンテーション レイヤーに固有の場合や、MVP、MVC などの Separated Presentation パターンを使用している場合は、プレゼンテーション レイヤーで、このコンポーネントを使用することがあります。
- データバインドされたコントロールを操作している場合は、UI コントロールに簡単にバインドできる適切な Presentation Model コンポーネントを設計または選択します。カスタムのオブジェクト、コレクション、またはデータ セットを Presentation Model コンポーネントの形式として使用する場合は、これらのものにデータ バインドをサポートする適切なインターフェイスとイベントが実装されていることを確認します。

- プレゼンテーション レイヤーでデータ検証を実行する場合は、検証用コードを Presentation Model コンポーネントに追加することを検討します。ただし、一元化された監視用コードやコード ライブラリを利用できる方法についても検討します。
- データがネットワーク経由で渡される場合やクライアント上のディスクに保存される場合は、Presentation Model コンポーネントに渡すデータのシリアル化に関する要件を考慮します。

また、Presentation Model コンポーネントとプレゼンテーション エンティティに適したデータ型を選択する必要があります。選択するデータ型は、アプリケーションの要件によって決定され、インフラストラクチャと開発技能による制約を受けます。まず、プレゼンテーション レイヤーのデータの形式を選択して、コンポーネントでビジネス ロジックと状態のカプセル化も実行するかどうかを決定する必要があります。次に、ユーザー インターフェイスでデータを表現する方法を決定する必要があります。プレゼンテーション レイヤーのデータに使用する一般的なデータ形式は、次のとおりです。

- **カスタム クラス**: ビジネス エンティティに直接マップする複雑なオブジェクトとしてデータを表現する場合は、カスタム クラスを使用します。たとえば、注文データを表すカスタムの Order オブジェクトを作成できます。また、カスタム クラスを使用して、ビジネス ロジックと状態のカプセル化、プレゼンテーション レイヤーの検証、およびカスタム プロパティの実装を行うこともできます。
- **配列とコレクション**: ある特定の 1 列の値を使用するリスト ボックス、ドロップダウン ボックスなどのコントロールにデータをバインドする場合は、配列またはコレクションを使用します。
- **DataSet と DataTable**: グリッド、リスト ボックス、ドロップダウン リストなど、データバインドされたコントロールで単一のテーブルに基づいたデータを操作する場合は、DataSet または DataTable を使用します。
- **TypedDataset**: データベースの変更による矛盾を避けるためにビジネス エンティティとの密結合が必要な場合は、TypedDataSet を使用します。
- **XML**: この形式が便利なのは、Web クライアントを操作する場合です。Web クライアントでは、データを Web ページに埋め込んだり、Web サービスや HTTP 要求経由で取得したりすることができます。ツリー ビューやグリッドなどのコントロールを使用する場合は、XML が適しています。XML は、保存、シリアル化、および通信チャネル経由でのやり取りも簡単です。
- **DataReader**: 常時ネットワーク接続が利用できる状態で、読み取り専用かつ順方向専用でデータにアクセスする場合は、DataReader を使用してデータを取得します。DataReader では、データベースのデータを順次処理したり、大量のデータを取得したりする効果的な方法が提供されます。ただし、ロジックがデータベースのスキーマに非常に緊密に結び付けられるので、通常はお勧めしません。

プレゼンテーション エンティティ

Presentation Model コンポーネントでは、可能な限り、ビジネス レイヤーのデータと、ビジネス ロジックおよび動作の両方をカプセル化する必要があります。このようにすると、プレゼンテーション レイヤーにおけるデータの一貫性と妥当性を確保して、ユーザー エクスペリエンスを向上できます。

Presentation Model コンポーネントがビジネス レイヤーのビジネス エンティティになり、プレゼンテーション レイヤーで直接使用されることがあります。また、Presentation Model コンポーネントがビジネス エンティティ コンポーネントのサブセットを表し、アプリケーションのプレゼンテーション レイヤーをサポートするよう明確に設計されている場合もあります。たとえば、データを UI コンポーネントやプレゼンテーション ロジック コンポーネントでより簡単に使用できる形式でコンポーネントに格納できます。このようなコンポーネントは、プレゼンテーション エンティティと呼ばれることがあります。

リッチ クライアント アプリケーションでは一般的なシナリオですが、ビジネス レイヤーとプレゼンテーション レイヤーの両方がクライアント上にある場合、通常はビジネス エンティティをビジネス レイヤーから直接使用します。しかし、ビジネス レイヤーで公開されているビジネス エンティティの形式や動作と異なる方法でビジネス データを格納または操作する必要がある場合は、プレゼンテーション エンティティの使用を検討することがあります。

ビジネス レイヤーがプレゼンテーション レイヤーとは別のティアに存在する場合、プレゼンテーション ティアでビジネス エンティティを使用できます。そのためには、ネットワーク経由でデータ転送オブジェクトを使用してデータをシリアル化し、プレゼンテーション ティアでビジネス エンティティのインスタンスとして復元します。また、必要な形式と動作がビジネス エンティティの形式と動作と異なる場合は、データをプレゼンテーション エンティティとして復元できます。図 1 に、このシナリオを示します。

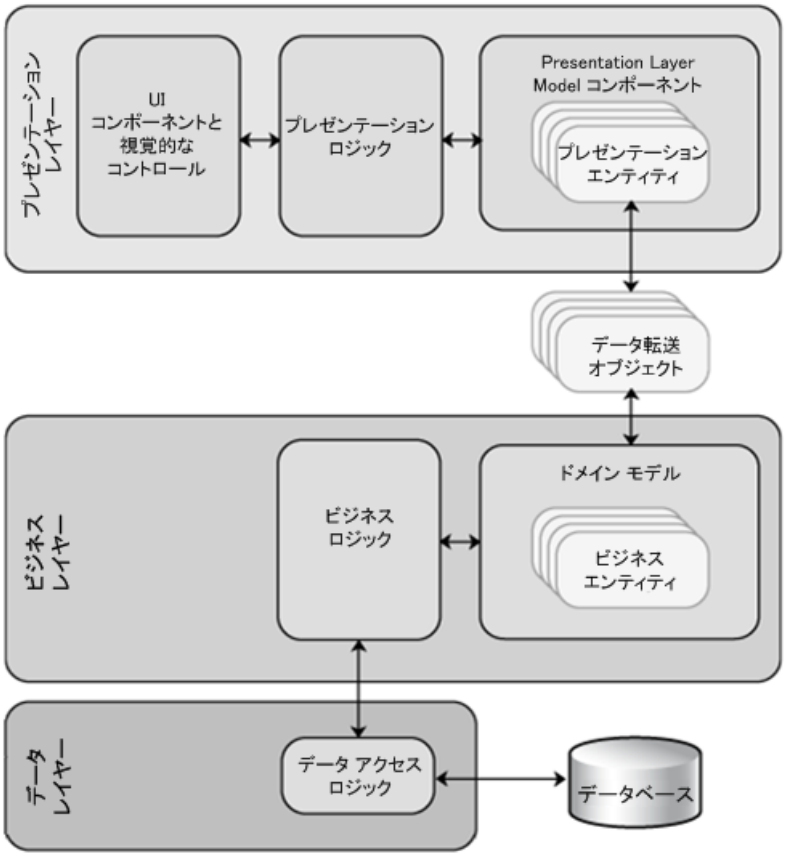


図 11

プレゼンテーション レイヤーとビジネス レイヤーが別の物理ティアに存在する場合に
役立つことがある Presentation Model コンポーネントとプレゼンテーション エンティティ

手順 5 – バインドの要件を決定する

データ バインドでは、ユーザー インターフェイスのコントロールと、アプリケーションのデータ コンポーネントやロジック コンポーネントとの間にリンクを作成する方法を提供します。データ バインドを使用すると、データベースのデータを配列やコレクションなど他の構造のデータと同様に表示および操作できます。データ バインドは、バインディング ターゲット (通常はユーザー インターフェイス コントロール) とバインディング ソース (通常はデータ構造コンポーネント、モデル コンポーネント、またはプレゼンテーション ロジック コンポーネント) の間の橋渡しの役割を果たします。

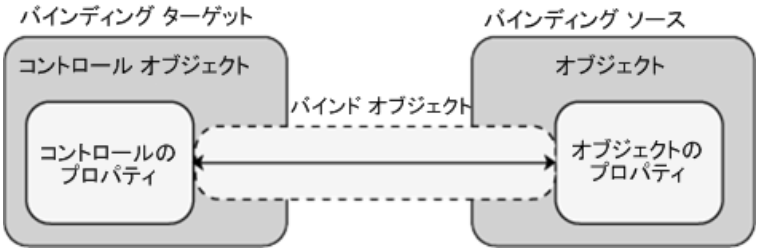


図 12

データ バインドで使用するオブジェクト

図 2 に示すように、通常、データ バインドには 4 つの要素があります。これらの要素が連携することで、バインディング ソースで公開されている値でバインドされたコントロールのプロパティが更新されます。データバインドされたコントロールとは、オブジェクトのコレクションにバインドされた DataGrid コントロールなど、データ ソースにバインドされたコントロールです。データ バインドは、Separated Presentation パターンでよく使用され、UI コンポーネント (View) を Presenter や Controller (プレゼンテーション ロジック コンポーネント) にバインドしたり、Presentation Layer Model コンポーネントやエンティティ コンポーネントにバインドしたりします。データ バインドのサポートとその実装は、UI テクノロジによって異なります。一般的に、ほとんどの UI テクノロジでは、コントロールをオブジェクトやオブジェクトの一覧にバインドできます。ただし、データ バインド テクノロジによっては、データ バインドを完全にサポートするために特定のインターフェイスやイベントをデータ ソースに実装する必要があります。たとえば、WPF では INotifyPropertyChanged を実装する必要があり、Windows フォームでは IBindingList を実装する必要があります。Separated Presentation パターンを使用している場合は、プレゼンテーション ロジック コンポーネントとデータ コンポーネントで、UI コントロールを簡単にコンポーネントにデータバインドするために必要なインターフェイスやイベントがサポートされるようにする必要があります。通常、使用できるバインドの一般的な種類は次の 2 つです。

- **一方向バインド**: バインド元プロパティが変更されると自動的にバインド先プロパティが更新されますが、バインド先プロパティが変更されてもバインド元プロパティには変更が反映されません。バインドされるコントロールが暗黙的に読み取り専用の場合は、この種類のバインドが適しています。たとえば、株価情報は一方向バインドです。バインド先プロパティの変更を監視する必要がない場合は、一方向バインドを使用すると不要なオーバーヘッドを回避できます。
- **両方向のバインド**: バインド元プロパティとバインド先プロパティのいずれかが変更されると、もう一方が自動的に更新されます。編集可能なフォームなど、完全な対話型の UI のシナリオには、この種類のバインドが適しています。Windows フォーム、ASP.NET、および WPF の多くの編集可能なコント

ロールでは両方向のバインドがサポートされているので、データ ソースと UI コントロールの変更が双方に反映されます。

手順 6 – エラー ハンドルの方針を決定する

UI コンポーネントはアプリケーションの外部境界なので、アプリケーションの安定性を最大限に高め、優れたユーザー エクスペリエンスを提供するには、適切なエラー ハンドルの方針を設計する必要があります。エラー ハンドルの方針を設計する際には、次のオプションを考慮します。

- **一元化された例外ハンドルの方針を設計する**: 例外とエラー ハンドルは横断的関心事なので、機能を一元化する別個のコンポーネントを使用して実装し、アプリケーションのさまざまなレイヤーから機能にアクセスできるようにする必要があります。また、このようにすると保守が容易になり、再利用性が高まります。
- **例外をログに記録する**: システムの境界でエラーをログに記録して、アプリケーションを使用する組織がエラーを検出して診断できるようにすることは、不可欠です。これはプレゼンテーション レイヤーのコンポーネントにおいて重要ですが、クライアント コンピューターで実行するコードで実現するのは困難な場合があります。個人を特定できる情報 (PII) やセキュリティに関する情報をログに記録する方法に注意し、ログのサイズと場所にも留意してください。
- **わかりやすいメッセージを表示する**: この方針では、エラーの理由を示して、ユーザーがエラーを修正する方法を説明する、わかりやすいメッセージを表示します。たとえば、データ検証のエラーでは、エラーが発生しているデータとデータが無効な理由がはっきりわかるようにする必要があります。メッセージでは、ユーザーがデータを修正したり有効なデータを入力したりする方法を示すこともできます。
- **再試行できるようにする**: この方針では、エラーの理由を説明してユーザーに操作の再試行を求める、わかりやすいメッセージを表示します。この方針は、リソースが使用できない場合やネットワークのタイムアウトなど、一時的で例外的な状況によってエラーが発生した場合に役立ちます。
- **汎用的なメッセージを表示する**: アプリケーションで予期しない例外が発生した場合は、エラーの詳細をログ記録する必要がありますが、ユーザーに表示するのは汎用的なメッセージのみにする必要があります。サポート グループに連絡する際に伝えられるように一意のエラー コードをユーザーに提示することを検討します。この方針は、予期しない例外に対処する際に役立ちます。一般的に、データの破損やセキュリティ リスクを防止するために、予期しない例外が発生した場合にはアプリケーションを終了することをお勧めします。

例外ハンドルの技法に関する詳細については、第 17 章「横断的関心事」を参照してください。Enterprise Library (例外ハンドルの方針を実装するために役立つ機能が付属) の詳細については、付録 F「patterns & practices の Enterprise Library」を参照してください。

手順 7 - 検証の方針を決定する

入力の検証に関する効果的な方針を設計すると、不要で悪意のあるデータをフィルター処理して、アプリケーションを脆弱性から保護できます。通常、入力の検証はプレゼンテーション レイヤーで実行され、ビジネス ルールの検証はビジネス レイヤーのコンポーネントで実行される必要があります。検証の方針を設計する際には、まず検証が必要なすべてのデータ入力を特定します。たとえば、Web クライアントからの、フォームのフィールド、パラメーター (GET と POST データ、クエリ文字列など)、非表示フィールド、およびビュー ステートへの入力は、すべて検証する必要があります。一般に、信頼されない入力元から渡されたすべてのデータは、検証する必要があります。
クライアント側コンポーネントとサーバー側コンポーネントの両方を備えたアプリケーション (RIA、アプリケーション サーバーのサービスを呼び出すリッチ クライアント アプリケーションなど) の場合、クライアントで実行するすべての検証に加えて、サーバーで検証を実行する必要があります。ただし、ユーザビリティとパフォーマンス上の理由から、検証の一部をクライアントで重複して実行できます。クライアントで検証を実行すると、ユーザーが無効なデータを入力した場合にすぐに通知できるので便利です。このため時間と帯域幅を節約できますが、悪意のある攻撃者は、クライアントに実装されている検証をバイパスする可能性がある点に注意してください。
検証するデータを特定したら、各データについて検証の技法を決定します。最も一般的な検証の技法は次のとおりです。

- **既知の適切な値の受け入れ** (許可リスト、肯定的検証): 一致条件を満たすデータのみを受け入れ、他のすべての値を拒否します。
- **既知の不適切な値の拒否** (禁止リスト、否定的検証): 既知の文字や値が含まれないデータを受け入れます。
- **一部削除**: 入力を安全にするために、既知の不適切な文字や値を削除または変換します。

一般的に、拒否する必要がある無効または悪意のあると思われる値をすべて特定しようとするのではなく、既知の適切な値のみを受け付ける (許可リストを使用する) ことをお勧めします。既知の適切な値の完全な一覧を定義できない場合は、二次的な防御策として、既知の不適切な値の部分的な一覧と一部削除、またはその両方を使用して検証を補完できます。

各プレゼンテーション テクノロジでは、異なる手法を使用して検証し、検証の問題をユーザーにレポートします。たとえば、WPF ではコンバーターと検証規則オブジェクト (多くの場合は、XAML を使用して接続されます) が使用され、Windows フォームでは検証とバインドのイベントが提供されます。
検証の技法に関する詳細については、第 17 章「横断的関心事」を参照してください。Enterprise Library (サーバー側とクライアント側の両方のオブジェクトとデータの検証に役立つ機能が付属) については、付録 F「patterns & practices の Enterprise Library」を参照してください。

patterns & practices のサービス

マイクロソフトの patterns & practices グループが提供している関連サービスの詳細については、次のリソースを参照してください。

- **Composite Client Application Guidance for WPF** (デスクトップと Silverlight の両方を対象): モジュール形式のアプリケーションの作成を簡略化します。詳細については、「Composite Client Application Guidance」(http://msdn.microsoft.com/en-us/library/cc707819.aspx、英語) を参照してください。
- **Enterprise Library**: 横断的関心事に対処する一連のアプリケーション ブロックを提供しています。詳細については、「Enterprise Library」(http://msdn.microsoft.com/en-us/library/cc467894.aspx、英語) を参照してください。
- **ソフトウェア ファクトリ**: スマート クライアント、WPF アプリケーション、Web サービスなど、特定の種類のアプリケーションを迅速に開発できます。詳細については、「patterns & practices: Catalog by Application Type」(http://msdn.microsoft.com/en-us/practices/bb969054.aspx、英語) を参照してください。
- **Unity Application Block** (エンタープライズ シナリオと Silverlight シナリオの両方を対象): 依存関係の挿入、サービスの場所、および制御の反転を実装する機能を提供します。詳細については、「Unity Application Block」(http://msdn.microsoft.com/en-us/library/dd203101.aspx、英語) を参照してください。

関連情報

Web リソースに簡単にアクセスするには、http://www.microsoft.com/architectureguide (英語) でオンライン版の参考文献を参照してください。

- Web クライアントの設計と実装に関するガイドライン (http://msdn.microsoft.com/ja-jp/library/ms978618.aspx)
- データ バインディングの概要 (http://msdn.microsoft.com/ja-jp/library/ms752347.aspx)
- 例外のデザインのガイドライン (http://msdn.microsoft.com/ja-jp/library/ms229014(VS.80).aspx)

12

ビジネス レイヤーのコンポーネントの設計

概要

ビジネス レイヤーのコンポーネントの設計は重要なタスクです。この設計を適切に行わないと、管理や拡張が困難なコードになる可能性があります。アプリケーションを設計および実装する際に使用できるビジネス レイヤーのコンポーネントには、複数の種類があります。このようなコンポーネントには、ビジネス ロジック コンポーネント、ビジネス エンティティ、ビジネス プロセス コンポーネントやビジネス ワークフロー コンポーネント、ユーティリティ コンポーネントやヘルパー コンポーネントなどがあります。この章では、まず、アプリケーションの設計で一般的に使用される、さまざまな種類のビジネス コンポーネントについて概説します。その中でも特に、ビジネス ロジック コンポーネントに主眼を置きます。また、アプリケーションの設計、トランザクションの要件、および処理の規則のさまざまな側面が、採用する設計にどのような影響を及ぼすかについても取り上げます。要件を理解したら、その要件をサポートする設計パターンに重点を置いて説明します。

手順 1 - アプリケーションで使用するビジネス コンポーネントを特定する

ビジネス レイヤーにはさまざまな種類のコンポーネントがあり、これを作成または使用してビジネス ロジックを処理する必要があります。この手順では、これらのコンポーネントを特定する方法を理解して、アプリケーションに必要なコンポーネントを特定することを目的としています。必要なコンポーネントの種類を判断するには、次のガイドラインを参考にしてください。

- ビジネス ロジックとアプリケーションの状態をカプセル化するには、ビジネス ロジック コンポーネントの使用を検討します。ビジネス ロジックは、アプリケーションのビジネス ルールと動作の実装と、データ検証などのプロセス全体における一貫性の維持に関するアプリケーション ロジックです。ビジネス ロジック コンポーネントは、簡単にテストできて、アプリケーションのプレゼンテーション レイヤーとデータ アクセス レイヤーから独立するように設計されている必要があります。
- ドメイン モデリングの手法の一環として、ビジネス エンティティの使用を検討します。これにより、アプリケーションで操作する必要があるビジネス ドメインの実際のビジネス エンティティ (製品、注文など) を表すコンポーネントに、ビジネス ロジックと状態をカプセル化します。ビジネス エンティティの詳細については、第 13 章「ビジネス エンティティの設計」を参照してください。
- 特定の順序で実行される多段階のプロセスをアプリケーションでサポートする必要がある場合、複数のビジネス ロジック コンポーネント間の通信が必要なビジネス ルールをアプリケーションで使用する場合、またはアプリケーションの進化や要件の変化に合わせてワークフローを更新してアプリケーションの動作を変更する場合は、ビジネス ワークフロー コンポーネントの使用を検討します。また、ビジネス ルールに基づいた動的な動作をアプリケーションで実装する必要がある場合にも、ビジネス ワークフロー コンポーネントの使用を検討します。この場合、ルールをルール エンジンに格納することを検討します。また、ワークフロー コンポーネントの実装には、Windows Workflow Foundation の使用を検討します。外部リソースに依存する複数の手順をアプリケーションで処理する必要がある場合や、アプリケーションに実行時間の長いトランザクションとして実行される必要があるプロセスが含まれている場合は、BizTalk Server などを使用してサーバー環境の統合を検討します。ワークフロー コンポーネントの詳細については、第 14 章「ワークフロー コンポーネントの設計」を参照してください。統合サービスの詳細については、付録 D「統合テクノロジ」を参照してください。

手順 2 - ビジネス コンポーネントに関する重要な決断を下す

作成するアプリケーションの全体的な設計と種類は、要求の処理に使用するビジネス コンポーネントに影響を及ぼします。たとえば、通常、Web アプリケーションのビジネス コンポーネントでは、メッセージ ベースの要求に対応し、Windows フォーム アプリケーションでは、イベント ベースの要求を使用してビジネス コンポーネントと直接通信します。また、さまざまな種類のアプリケーションを使用する際には、他にも考慮すべき要素がありますが、すべての種類のアプリケーションに共通している要素と、特定のアプリケーションの種類に特有の要素があります。ビジネス コンポーネントに関して決断を下す必要がある重要事項は次のとおりです。

- **場所**: ビジネス コンポーネントは、クライアントとアプリケーション サーバーのどちらに配置しますか。それとも、クライアントとサーバーの両方に配置しますか。スタンドアロンのリッチ クライアント アプリケーションやリッチ インターネット アプリケーション (RIA) の場合、パフォーマンスを向上する必要がある場合、またはビジネス エンティティにドメイン モデルの設計を使用する場合は、一部または全部のビジネス コンポーネントをクライアントに配置することを検討します。共通のビジネス ロジックを使用して複数の種類のクライアントをサポートする必要がある場合、クライアントからアクセスできないリソースにビジネス コンポーネントでアクセスする必要がある場合、またはセキュリティ上の理由から管理および保護されたサーバー環境のコンポーネントを保護するには、一部または全部のビジネス コンポーネントをアプリケーション サーバーに配置することを検討します。
- **結合**: プレゼンテーション レイヤーのコンポーネントは、ビジネス レイヤーのコンポーネントとどのように通信しますか。プレゼンテーション レイヤーのコンポーネントでビジネス コンポーネントを把握している場合は、蜜結合を使用する必要がありますか。また、ビジネス コンポーネントの詳細を非公開にするために抽象化を使用する場合は、疎結合を使用する必要がありますか。クライアントに両方のコンポーネントが配置されているリッチ クライアント アプリケーションまたは RIA では、簡略化のために、プレゼンテーション コンポーネントとビジネス コンポーネント間で蜜結合の使用を検討することをお勧めします。ただし、プレゼンテーション コンポーネントとビジネス コンポーネント間で疎結合を使用すると、テスト容易性と柔軟性がするというメリットがあります。アプリケーション サーバーや Web サーバーにビジネス コンポーネントが配置されているリッチ クライアント アプリケーションまたは RIA の場合は、可能な限り相互作用を疎結合にするようにサービス インターフェイスを設計できます。
- **通信**: ビジネス コンポーネントがプレゼンテーション コンポーネントと同じティアに配置されている場合、イベントとメソッドによるコンポーネント ベースの通信の使用を検討します。この方法を採用すると、パフォーマンスが最大限に高まります。ただし、ビジネス コンポーネントが Web サーバーとは別個の物理ティアに配置されている場合、プレゼンテーション レイヤーとビジネス レイヤー間で疎結合を使用する Web アプリケーションを設計する場合、またはリッチ クライアント アプリケーションや RIA の場合は、サービス インターフェイスを実装して、プレゼンテーション レイヤーとビジネス コンポーネント間でメッセージ ベースの通信を使用することを検討します。また、アプリケーション サーバーや Web サーバーと不定期に接続するリッチ クライアント アプリケーションや RIA の場合は、接続時にクライアントが再同期できるようにサービス インターフェイスを慎重に設計する必要があります。

メッセージ ベースの通信を使用する場合、重複する要求を管理する方法とメッセージを確実に配信する方法を検討します。サービス アプリケーション、Microsoft Message Queuing などのメッセージング システムを使用するメッセージ ベースのアプリケーション、またはプロセスの実行時間が長いためにユーザーが同じ操作を何度も実行することがある Web アプリケーションを設計する場合は、”べき等性” (重複する要求を無視する機能) が重要となります。Microsoft Message Queuing などのメッセージング システムを使用するメッセージ ベースのアプリケーション、クライアントとサービス間でメッセージ ルーターを使用するサービス、またはクライアントが応答を待機せずにメッセージを送信する "お任せ (Fire and Forget)" 操作をサポートするサービスを設計する場合は、"保証された配信" が重要となります。また、処理の待機中に格納されることがあるキャッシュ メッセージが古くなっている可能性も考慮します。

手順 3 - 適切なトランザクションのサポートを選択する

ビジネス コンポーネントでは、ビジネス レイヤーで必要になる可能性があるトランザクションを調整および管理します。ただし、まずはトランザクションのサポートが必要かどうかを判断します。トランザクションを使用して、1 つ以上のリソース マネージャー (データベース、メッセージ キューなど) に対して実行される一連の操作が、他のトランザクションから独立した 1 つの単位として完了するようにします。一連の操作に含まれる 1 つの操作でエラーが発生した場合、他の操作をすべてロール バックして、システムを一貫性のある状態に保つ必要があります。たとえば、複数のビジネス ロジック コンポーネントを使用して、3 つのテーブルを更新する操作を行うとします。更新時に 1 つの操作でエラーが発生し、残り 2 つの操作が成功した場合、データ ソースは一貫性のない状態になります。つまり、この状態では、他の操作で使用する際に無効なデータが存在することになります。トランザクションを実装する際に、使用できるオプションは次のとおりです。

- System.Transactions 名前空間では、ビジネス ロジック コンポーネントを使用してトランザクションを開始および管理します。Lightweight Transaction Manager (LTM) と共に .NET Framework 2.0 に導入された System.Transactions 名前空間は、永続的でないリソース マネージャーまたは単一の永続的リソース マネージャーに対応しています。この手法では、TransactionScope クラスを使用してプログラムを明示的に記述する必要があり、複数の永続的リソース マネージャーがトランザクションに参加している場合は、トランザクション スコープを拡大して、分散トランザクション コーディネーター (DTC) を使用できます。トランザクションのサポートが必要な新しいアプリケーションを作成したり、複数の永続的でないリソース マネージャーをまたぐトランザクションが存在する場合は、System.Transactions 名前空間の使用を検討します。

- WCF トランザクションは .NET Framework 3.0 で導入され、System.Transactions の機能を基盤としています。そのため、このトランザクションでは、TransactionScopeRequired、TransactionAutoComplete、TransactionFlow など、さまざまな属性やプロパティを使用して、実装されているトランザクションを管理するための宣言型の手法が提供されます。WCF サービスとの通信でトランザクションをサポートする必要がある場合は、WCF トランザクションの使用を検討します。ただし、トランザクションを管理するコードを使用するのではなく、宣言的なトランザクションを定義する必要があるかどうかを検討します。
- ADO.NET トランザクションは、.NET Framework 1.0 で導入された機能ですが、トランザクションの開始と管理にビジネス ロジック コンポーネントを使用する必要があります。これには、明示的なプログラミング モデルが使用されており、開発者は非分散トランザクションをコードで管理する必要があります。既に ADO.NET トランザクションを使用しているアプリケーションを拡張する場合または ADO.NET プロバイダーを使用してデータベースにアクセスし、トランザクションが 1 つのリソースに限定されている場合は、ADO.NET トランザクションの使用を検討します。また、ADO.NET 2.0 以降では、この一覧の前半で説明した System.Transactions 名前空間の機能を使用して、分散トランザクションをサポートします。
- ストアド プロシージャに組み込むことができるデータベース トランザクションを使用して、トランザクションを管理します。データベース トランザクションを使用すると、ビジネス プロセスの設計が簡略化される場合があります。トランザクションがビジネス ロジック コンポーネントによって開始される場合、データベース トランザクションはビジネス コンポーネントによって作成されるトランザクションに参加します。トランザクションで管理する必要があるすべての変更をカプセル化するストアド プロシージャを作成している場合や、同じストアド プロシージャを使用する複数のアプリケーションが存在し、そのストアド プロシージャ内でトランザクション要件をカプセル化できる場合は、データベース トランザクションの使用を検討します。

分散トランザクションを使用するシステムでは、サブシステム間の結合性が高まる可能性があることを考慮します。リモート システムを対象とするトランザクションでは、ネットワーク トラフィックが増大するため、パフォーマンスに影響を及ぼす可能性が高くなります。トランザクションは負荷が高い処理なので迅速に実行する必要があります。迅速に実行されないと、リソースが必要以上に長い間ロックされ、タイムアウトやデッドロックが発生する原因になる可能性があります。

トランザクションに参加すると、外部サービスが内部リソースをロックできるため、トランザクションに参加するサービスは、確実に信頼できるサービスに限定します。サービスを呼び出してビジネス プロセスを実行する場合、この呼び出しをまたぐアトミックのトランザクションは、可能な限り作成しないようにします。

手順 4 - ビジネス ルールの処理方法を特定する

ビジネス ルールの管理は、アプリケーションを設計するうえで非常に困難な側面の 1 つです。一般的に、ビジネス ルールはビジネス レイヤーで管理する必要があります。ですが、正確にはビジネス レイヤーのどの部分でコンポーネントを管理するのでしょうか。これには、ビジネス ロジック コンポーネント、ビジネス ワークフロー コンポーネント、およびビジネス ルール エンジンを使用したり、ドメイン モデルの設計をモデルにカプセル化したルールと組み合わせて使用したりします。ビジネス ルールの処理については、次のオプションを検討します。

- ビジネス ロジック コンポーネントは、コンポーネントの実装に使用した設計パターンに応じて、単純なルールまたは非常に複雑なルールの処理に使用できます。ドメイン モデルの設計をビジネス エンティティに実装しない場合、またはビジネス ルールを含む外部ソースを使用する場合は、Web アプリケーションやサービスのタスクまたはドキュメント指向の操作に、ビジネス ロジック コンポーネントの使用を検討します。
- ビジネス ルールをビジネス エンティティから分離する場合、使用するビジネス エンティティでビジネス ルールのカプセル化がサポートされていない場合、または複数のビジネス エンティティ間での通信を調整するビジネス ロジックをカプセル化する必要がある場合は、ワークフロー コンポーネントを使用します。
- ビジネス ルール エンジンでは、開発の知識を持たないユーザーがルールを確立および変更できるようにする方法を提供します。ただし、アプリケーションが複雑になり、追加のオーバーヘッドが発生するので、必要な場合にのみ使用します。つまり、ビジネス ルール エンジンは、アプリケーションに関連するさまざまな要素に基づいてルールを調整する必要がある場合にのみ使用します。定期的に変更する必要がある不安定なビジネス ルールの場合、カスタマイズをサポートして柔軟性を持たせる場合、またはビジネス ユーザーがルールを管理および更新できるようにする場合は、ビジネス ルール エンジンの使用を検討します。ユーザーが変更できるルールのみを公開し、ビジネス ロジックの適切な動作を実現するうえで重要なルールを未承認ユーザーが変更できないようにします。
- ドメイン モデルの設計は、ビジネス エンティティのビジネス ルールをカプセル化するために使用されます。ただし、ドメイン モデルは、適切に設計するのが困難な場合があり、特定の見解やコンテキストに重点を置く傾向があります。ビジネス ロジックの一部がクライアントに配置されていたり、ドメイン モデルのエンティティがメモリ内で初期化および格納されるリッチ クライアント アプリケーションや RIA の場合、または Web アプリケーションやサービス アプリケーションと関連のあるセッション状態で管理できるドメイン モデルの場合、ドメイン モデルにルールをカプセル化することを検討し

ます。ドメイン モデルの一部をクライアントに配置する場合は、ドメイン モデルをサーバーに反映してルールと動作を適用し、セキュリティと保守容易性を確保する必要があります。

手順 5 - 要件を満たすパターンを特定する

動作に関するパターンは、運用システムの動作を観察し、繰り返し行われるプロセスを確認したうえで作成されています。通常、ビジネス コンポーネントで使用する可能性があるパターンは、動作に関する設計パターンです。"動作に関する設計パターン" とは、設計の段階でアプリケーションの動作に重点を置いたパターンです。さまざまな種類のアプリケーションや、アプリケーションの設計のさまざまなレイヤーで発生するパターンを特定して定義しています。定義されているパターンをすべて学習するのは現実的ではありませんが、さまざまな種類のパターンの理解を深めることで、シナリオを確認してパターンで表せる動作を特定することは可能です。次の表に、ビジネス コンポーネントで一般的に使用するパターンを示します。

パターン	推奨事項
Adapter	互換性がないインターフェイスを持つクラスを同時に使用できます。開発者は、このパターンを使用すると、既存のクラスに別の実装を提供するポリモーフィックな一連のクラスを実装できます。
Command	さまざまなコンポーネントに対して同じコマンドを実行するために使用する、メニュー、ツール バー、およびキーボード ショートカットによる対話機能を備えたリッチ クライアント アプリケーションでの使用を推奨します。また、コマンドを実装するために Supervising Presenter パターンと組み合わせて使用することもできます。
Chain of Responsibility	複数の要求ハンドラーをチェーンして、各ハンドラーで要求を確認して、処理するか、チェーンの次のハンドラーに渡されるようにします。If、then、else ステートメントに代わるパターンで、複雑なビジネス ルールを処理する機能を備えています。
Decorator	実行時にオブジェクトの動作を拡張し、要求を実行する際に行う操作を追加または変更します。Decorator クラスで実装される一般的なインターフェイスが必要です。これを同時にチェーンすることで複雑なビジネス ルールを処理できます。
Dependency Injection	個別のクラスを使用して、オブジェクトのメンバー (フィールドやプロパティ) を作成および設定します。これにより、通常、実行時に構成ファイルの内容に基づいた依存関係が作成されます。構成ファイルでは、オブジェクト型のマッピングまたは登録を指定するコンテナーを定義します。また、アプリケーション コードでもオブジェクトのマッピングまたは登録を定義できます。このパターンでは、動作を変更して複雑なビジネス

	ルールを実装する柔軟性のある手法を提供します。
Façade	複数のビジネス ロジック コンポーネントから結果を統合する大まかな操作を提供します。通常、メッセージ ベースのインターフェイスのリモート ファサードとしてビジネス レイヤーに実装され、プレゼンテーション レイヤーとビジネス レイヤー間の疎結合を提供するために使用されます。
Factory	具象型を指定せずオブジェクト インスタンスを作成します。一般的なインターフェイスを実装するオブジェクトまたは一般的な基本クラスを拡張するオブジェクトが必要です。
Transaction Script	必要最低限のビジネス ルールを使用した基本的な CRUD 操作での使用を推奨します。また、Transaction Script のコンポーネントでもトランザクションが開始されます。つまり、コンポーネントによって実行されるすべての操作は、作業の atomic 単位である必要があります。ビジネス ロジック コンポーネントでは、このパターンを使用して、他のビジネス コンポーネントやデータ コンポーネントと通信して、操作を完了します。

上記の表では、ビジネス コンポーネントで使用される一般的なパターンの多くについて説明しましたが、ビジネス コンポーネントと関連のあるパターンは他にも数多く存在します。パターンを選択するときには、そのパターンが、シナリオに適合し、アプリケーションが必要以上に複雑にならないことを見極めるのが重要です。

関連情報

Web リソースに簡単にアクセスするには、http://www.microsoft.com/architectureguide (英語) でオンライン版の参考文献を参照してください。

- ビジネス コンポーネントの設計に関する詳細については、「.NET のアプリケーション アーキテクチャ: アプリケーションとサービスの設計」(http://msdn.microsoft.com/ja-jp/library/ms954595.aspx) を参照してください。
- ビジネス レイヤーとコンポーネントのパフォーマンスに関する詳細については、次のリソースを参照してください。
    - .NET アプリケーションのパフォーマンスとスケーラビリティのためのアーキテクチャと設計のレビュー (http://msdn.microsoft.com/ja-jp/library/ms998544.aspx)

    - アプリケーション パフォーマンスのための設計ガイドライン (http://msdn.microsoft.com/ja-jp/library/ms998541.aspx)

- ビジネス コンポーネントにトランザクションを実装する方法の詳細については、次のリソースを参照してください。
    - .NET Framework 2.0 の System.Transactions について (http://msdn.microsoft.com/ja-jp/library/ms973865.aspx)
    - トランザクション (http://msdn.microsoft.com/ja-jp/library/ms730266.aspx)
    - トランザクション処理 (http://msdn.microsoft.com/ja-jp/library/w97s6fw4.aspx)
- ビジネス コンポーネントにワークフローを実装する方法の詳細については、次のリソースを参照してください。
    - Windows Workflow Foundation ルール エンジンの紹介 (http://msdn.microsoft.com/ja-jp/library/aa480193.aspx)
    - Windows Workflow Foundation (http://msdn.microsoft.com/ja-jp/library/ms735967.aspx)

13

ビジネス エンティティの設計

概要

ビジネス エンティティにはデータ値が格納され、データ値はプロパティによって公開されます。つまり、ビジネス エンティティにはアプリケーションで使用するビジネス データが格納および管理されます。また、ビジネス データや関連する機能へのプログラムによるステートフルなアクセスが提供されます。さらに、一貫性を確保して、ビジネス ルールと動作を実装するために、エンティティに含まれるデータを検証して、ビジネス ロジックをカプセル化しています。そのため、適切なビジネス エンティティを設計または選択することは、ビジネス レイヤーのパフォーマンスと効率を最大限に高めるうえで非常に重要です。
この章では、ビジネス エンティティ コンポーネントの設計について説明します。まず、さまざまなデータ形式と、データがアプリケーションで使用される方法を確認します。次に、選択したデータ形式によって、設計にビジネス ルールを実装する方法がどのように決まるかを説明します。最後に、カスタム オブジェクトの設計オプションを紹介し、さまざまなデータ形式でシリアル化をサポートする方法について説明します。

コンポーネントの設計に関する一般的な考慮事項とアプリケーションの各レイヤーで一般的に使用されるコンポーネントの詳細については、第 10 章「コンポーネントのガイドライン」を参照してください。

手順 1 – 表現を選択する

この手順では、シナリオに適した表現を選択できるように、ビジネス エンティティを表現するさまざまな方法について説明し、各方法のメリットとデメリットを確認します。最も一般的な形式のオプションは次のとおりです。

- **カスタム ビジネス オブジェクト**: システムのエンティティを表す共通言語ランタイム (CLR) オブジェクトです。このオブジェクトは、ADO.NET Entity Framework (EF) や NHibernate などのオブジェ

クト/リレーショナル マッピング (O/RM) テクノロジを使用して作成できます (詳細については、この章の最後の「関連情報」を参照してください)。また、手動で作成することもできます。複雑なビジネス ルールや動作を関連データと共にカプセル化する必要がある場合は、カスタム ビジネス オブジェクトが適しています。AppDomain 境界、プロセス境界、または物理的な境界を越えてカスタム ビジネス オブジェクトにアクセスする必要がある場合は、データ転送オブジェクト (DTO) 経由でアクセスできるサービス レイヤーとカスタム ビジネス オブジェクトを更新または編集する操作を実装できます。

- **DataSet** または **DataTable**: DataSet は、通常、実際のデータベース スキーマに緊密にマップされたメモリ内データベースの形式を取ります。一般に、DataSet は、O/RM マッピング メカニズムを使用しておらず、アプリケーション ロジックのデータがデータベース スキーマに緊密にマップされるデータ指向のアプリケーションを構築している場合にのみ使用されます。ビジネス ロジックやビジネス ルールをカプセル化するために DataSet を拡張することはできません。DataSet は XML にシリアル化できますが、プロセス境界やサービス境界を越えて公開しないようにする必要があります。
- **XML**: 構造化データを含む標準ベースの形式です。通常、XML は、プレゼンテーション レイヤーで必要になるか、ロジックがコンテンツのスキーマに基づいてコンテンツを処理する場合にのみ、ビジネス エンティティを表現するために使用します。たとえば、XML ドキュメント内の既知のノードに基づいてメッセージがルーティングされるメッセージ ルーティング システムが、これに該当します。XML の使用と操作では、大量のメモリが使用される可能性があることに注意してください。

手順 2 – ビジネス エンティティの設計を選択する

ビジネス エンティティを表現するのにカスタム オブジェクトが最適だと判断したら、次はカスタム オブジェクトを設計します。カスタム オブジェクトの設計手法は、使用する予定のオブジェクトの種類によって異なります。たとえば、ドメイン モデル エンティティの場合はビジネス ドメインを詳しく分析する必要がありますが、テーブル モジュール エンティティの場合はデータベース スキーマを理解する必要があります。ビジネス オブジェクトを使用する際の一般的な設計手法には、次のようなものがあります。

- **ドメイン モデル**: オブジェクト指向の設計パターンです。ドメイン モデル設計の目的は、ビジネス ドメイン内の実際のエンティティを表現するビジネス オブジェクトを定義することです。ドメイン モデル設計を使用する場合、ビジネス エンティティまたはドメイン エンティティには、動作と構造の両方が含まれます。つまり、ビジネス ルールとリレーションシップは、ドメイン モデルにカプセル化されます。ドメイン モデル設計では、ビジネス ドメインを詳しく分析する必要があり、通常、ほとんどの

データベースで使用されるリレーショナル モデルにマップされることはありません。ビジネス ドメインに関係する複雑なビジネス ルールがある場合、リッチ クライアントを設計しており、ドメイン モデルを初期化してメモリに格納できる場合、または要求ごとにドメイン モデルを初期化する必要があるステートレスなビジネス レイヤーを操作していない場合は、ドメイン モデル設計の使用を検討します。ドメイン モデルとドメイン駆動設計の詳細については、この章後半の「ドメイン駆動設計」を参照してください。

- **テーブル モジュール**: オブジェクト指向の設計パターンです。テーブル モジュール設計の目的は、データベース内のテーブルまたはビューに基づいてエンティティを定義することです。通常、データベースへのアクセスとテーブル モジュール エンティティへのデータの設定に使用される操作は、エンティティ内にカプセル化されます。ただし、データ アクセス コンポーネントを使用して、データベース操作を実行したり、テーブル モジュール エンティティにデータを設定したりすることもできます。データベース内のテーブルまたはビューがアプリケーションで使用するビジネス エンティティを厳密に表現している場合、またはビジネス ロジックと操作が単一のテーブルまたはビューに関係している場合は、テーブル モジュール設計の使用を検討します。
- **カスタム XML オブジェクト**: アプリケーション コード内で操作できる、シリアル化解除された XML データを表します。このオブジェクトのインスタンスは、クラスのプロパティを XML 構造体の要素と属性にマップする属性を使用して定義されたクラスから作成します。Microsoft .NET Framework には、XML データをオブジェクトにシリアル化解除したり、オブジェクトを XML データにシリアル化したりするのに使用できるコンポーネントが用意されています。使用しているデータが XML 形式である場合 (XML ファイルや、結果セットとして XML を返すデータベース操作など)、XML 以外のデータ ソースから XML データを生成する必要がある場合、または読み取り専用のドキュメント ベースのデータを操作している場合は、カスタム XML オブジェクトの使用を検討します。

カスタム オブジェクトを使用する場合、すべてのビジネス エンティティが同じ設計に沿っている必要はありません。たとえば、ルールが複雑なアプリケーションのある部分では、ドメイン モデル設計が必要になることがありますが、それ以外の部分では、必要に応じて XML オブジェクト、テーブル モジュール設計、またはドメイン オブジェクトを使用できます。

手順 3 – シリアル化のサポートを決定する

境界を越えてビジネス エンティティを転送する方法を決定する必要があります。ほとんどの場合、AppDomain 境界、プロセス境界、サービス インターフェイス境界などの物理的な境界を越えてデータを渡すには、データをシリアル化する必要があります。また、論理的な境界を越えてデータを渡すときにデータをシリアル化することもできます。ただし、この場合はパフォーマンスへの影響に留意する必要があります。ビジネス エンティティを転送する際には、次のオプションを検討します。

- **必要な場合にのみシリアル化可能なビジネス エンティティを直接公開する**: 同じ物理ティアにある、アプリケーションの別のレイヤーでビジネス エンティティを使用している場合、シリアル化によりビジネス エンティティを直接公開する方法が最も簡単です。ただし、この方法には、ビジネス エンティティとその実装のコンシューマー間に依存関係が生じるというデメリットがあります。そのため、一般的に、ビジネス エンティティのコンシューマーを直接管理できて、物理ティア間でのビジネス エンティティへのリモート アクセスが不要でない限り、このオプションの使用はお勧めしません。
- **ビジネス エンティティをシリアル化可能なデータ転送オブジェクトに変換する**: データのコンシューマーとビジネス レイヤーの内部実装を分離するには、ビジネス エンティティを特殊なシリアル化可能なデータ転送オブジェクトに変換することを検討します。データ転送オブジェクト (DTO) は、境界を越えて転送するために複数のデータ構造体を単一の構造体にパッケージ化する設計パターンです。データ転送オブジェクトは、ビジネス エンティティの複数のコンシューマーが異なるデータ表現またはモデル (たとえば、プレゼンテーション ティア) を使用している場合にも役立ちます。この手法を使用すると、データのコンシューマーに影響を及ぼすことなくビジネス レイヤーの内部実装を変更できるようになり、インターフェイスのバージョン管理が容易になります。この手法は、外部クライアントでデータが使用される場合に適しています。
- **XML を直接公開する**: 場合によっては、ビジネス エンティティを XML としてシリアル化して公開することがあります。.NET Framework では、XML データの広範なシリアル化がサポートされています。ほとんどの場合、XML へのシリアル化は、ビジネス エンティティの属性で制御されます。

サービス インターフェイスのデータ スキーマに関する詳細については、第 9 章「サービス レイヤーのガイドライン」を参照してください。レイヤーとティアとの通信に関する詳細については、第 18 章「通信とメッセージ」を参照してください。

ドメイン駆動設計

ドメイン駆動設計 (DDD) は、ビジネス ドメイン、その要素と動作、およびそれらの関係に基づいてソフトウェアを設計するオブジェクト指向の手法です。このアーキテクチャ スタイルの目的は、ビジネス ドメインの専門家がわかる共通言語でドメイン モデルを定義することによって、ビジネス ドメインで必要な機能を実現するソフトウェア システムを構築することです。ドメイン モデルは、ソリューションを合理化できるフレームワークと見なせます。

ドメイン駆動設計を適用するには、モデル化するビジネス ドメインについてよく理解したり、そのようなビジネス知識を習得することに精通している必要があります。多くの場合、開発チームは、ビジネス ドメインの専門家と連携してビジネス ドメインをモデル化します。アーキテクト、開発者、およびその技術の専門家はさまざまな経歴を持っており、多くの環境では、それぞれの目的、設計、および要件を定義する際に使用する表現が異なります。ただし、ドメイン駆動設計では、チーム全体が、ビジネス ドメインに重点を置く共通言語のみを使用するため、技術的な専門用語は使用しません。

ソフトウェアの中核にはドメイン モデル (この共通言語が直接投影されたもの) があるため、チームはドメイン モデルに関する共通言語を分析することによって、ソフトウェア内のギャップをすばやく特定できます。共通言語を作成することは、単にドメインの専門家から情報を入手して、それを適用するだけではありません。開発チーム内で発生するコミュニケーションの問題の原因の多くは、ドメインの共通言語の誤解だけでなく、ドメインの共通言語自体があいまいであることにもあります。ドメイン駆動設計のプロセスの目的は、使用する共通言語を実装することだけでなく、ドメインの共通言語を改善および改良することにもあります。つまり、モデルはドメインの共通言語を直接投影したものになるので、構築するソフトウェアにメリットがあります。

ドメイン モデルは、エンティティ、値オブジェクト、集計ルート、レポジトリ、およびドメイン サービスを使用して表現され、境界コンテキストと呼ばれる、大まかな責任が割り当てられている領域に分類されます。

"エンティティ" とは、ソフトウェアの状態が変化しても変わらない一意の ID を持つ、ドメイン モデルのオブジェクトです。エンティティは状態と動作の両方をカプセル化します。Customer オブジェクトは、エンティティの一例です。このオブジェクトは、特定の顧客に関する状態を表して保持し、その顧客に対して実行できる操作を実装します。

"値オブジェクト" とは、ドメインの特定の側面を表すために使用されるドメインのオブジェクトです。値オブジェクトに一意の ID はありませんが、不変のオブジェクトです。値オブジェクトの例としては、Transaction Amount (トランザクション金額) や Customer Address (顧客の住所) があります。

"集計ルート" とは、論理的に関連する子エンティティや値オブジェクトをグループ化し、これらへのアクセスを制御して、その間の相互作用を調整するエンティティです。

"レポジトリ" は、通常、オブジェクト/リレーショナル マッピング (O/RM) フレームワークを使用して、集計ルートを取得および格納します。
"ドメイン サービス" は、操作、アクション、またはビジネス プロセスを表し、ドメイン モデルの他のオブジェクトを参照する機能を提供します。特定のライフサイクルや ID を持つオブジェクトに、機能やドメインのある部分をマップできないことがあります。このような機能は、ドメイン サービスとして宣言できます。電子商取引ドメインのカタログ価格設定サービスは、ドメイン サービスの一例です。

モデルを単純で役に立つ言語コンストラクトとして維持するためには、ドメイン モデルで、大規模な分離とカプセル化を実装する必要があることが一般的です。このため、ドメイン駆動設計に基づいて構築したシステムのコストは比較的高くなることがあります。ドメイン駆動設計を使用すると、保守容易性など、多くの技術上のメリットはありますが、複雑なドメイン (モデルと言語のプロセスによって、複雑な情報を伝達したり、ドメインの共通理解を構築するうえで、明らかなメリットがもたらされる場合) にのみ適用することをお勧めします。

ドメイン駆動設計の技法の詳細については、「Domain Driven Design (ドメイン駆動設計) Quickly 日本語版」(http://www.infoq.com/jp/minibooks/domain-driven-design-quickly) を参照してください。また、Eric Evans 著『Domain-Driven Design: Tackling Complexity in the Heart of Software』(Addison-Wesley、ISBN: 0-321-12521-5) と Jimmy Nilsson 著『Applying Domain-Driven Design and Patterns: With Examples in C# and NET』(Addison-Wesley、ISBN: 0-321-26820-2) も参照してください。

関連情報

Web リソースに簡単にアクセスするには、http://www.microsoft.com/architectureguide (英語) でオンライン版の参考文献を参照してください。

- ビジネス エンティティの設計パターンに関する詳細については、「Enterprise Solution Patterns Using Microsoft .NET」(http://msdn.microsoft.com/en-us/library/ms998469.aspx、英語) を参照してください。
- ビジネス エンティティの設計に関する詳細については、「Integration Patterns」(http://msdn.microsoft.com/en-us/library/ms978729.aspx、英語) を参照してください。
- ドメイン駆動設計の詳細については、次のリソースを参照してください。
    - ドメイン駆動設計の概要 (http://msdn.microsoft.com/ja-jp/magazine/dd419654.aspx)

    - ドメイン駆動設計・開発の実践
      (http://www.infoq.com/jp/articles/ddd-in-practice)

- ビジネス レイヤーの設計パターンに関する詳細については、「Service Orientation Patterns」(http://msdn.microsoft.com/en-us/library/aa532436.aspx、英語) を参照してください。
- ADO.NET Entity Framework の詳細については、「The ADO.NET Entity Framework Overview」(http://msdn.microsoft.com/en-us/library/aa697427(VS.80).aspx、英語) を参照してください。
- Microsoft Dynamics を使用したビジネス エンティティ設計の詳細については、「Business Entities」(http://msdn.microsoft.com/en-us/library/ms940455.aspx、英語) を参照してください。
- Microsoft Dynamics を使用したビジネス エンティティ モデリングの詳細については、「Modeling Entities」(http://msdn.microsoft.com/en-us/library/aa475207.aspx、英語) を参照してください。
- Office Business Applications (OBA) とビジネス エンティティの併用の詳細については、「Building Office Business Applications」(http://msdn.microsoft.com/en-us/library/bb266337.aspx、英語) を参照してください。
- オープン ソースである NHibernate フレームワークの詳細については、「NHibernate Forge」(http://nhforge.org/Default.aspx、英語) を参照してください。

14

ワークフロー コンポーネントの設計

概要

ユーザーのタスクを規定の順序で完了する必要がある (各手順が完了してから次の手順に進む必要がある) シナリオまたはユーザーのタスクが基になる一連のビジネス ルールを満たす必要があるシナリオは、多数あります。ワークフロー コンポーネントを使用すると、タスクをカプセル化したり、タスクを完了するのに必要な手順を調整したりすることができます。また、ワークフロー コンポーネントでは、処理対象の情報 (ユーザーやビジネス プロセスを定義する動的なビジネス ルールによって入力されるデータなど) に依存するタスクもサポートできます。
この章では、さまざまなシナリオを考察し、ワークフロー コンポーネントの設計方法についてのガイダンスを提供します。まず、実際のシナリオと主要なワークフロー シナリオの対応を確認して、アプリケーションに適したワークフロー スタイルを特定できるようにします。次に、要件と規則が、ワークフロー コンポーネントの実装で使用できるオプションに、どのような影響を及ぼすかを考察します。最後に、使用可能なさまざまなオプションをサポートするワークフロー コンポーネントの設計についてのガイダンスを提供します。

コンポーネントの設計に関する一般的な考慮事項とアプリケーションの各レイヤーで一般的に使用されるコンポーネントの詳細については、第 10 章「コンポーネントのガイドライン」を参照してください。

手順 1 – シナリオを使用してワークフロー スタイルを特定する

ワークフロー スタイルには、シーケンシャル、ステート マシン、およびデータ ドリブンという 3 つの基本的な種類があります。シーケンシャル ワークフローの場合、タスクは完了するまでに決められた一連の手順の間を遷移します。ステート マシン ワークフローの場合、アクティビティは、一連の状態と状態の遷移を引き起こすイベントとして定義されます。データ ドリブン ワークフローの場合、アクティビティはデータに関連する情報に基づいて実行

されます。そのため、ワークフロー コンポーネントを設計する際には、まず、サポートする必要があるワークフローのスタイルを理解する必要があります。次に、この 3 つの基本的なワークフロー スタイルを使用する場合についてのガイドを示します。

- **シーケンシャル ワークフロー スタイル**: このスタイルでは、ワークフローによってアクティビティのシーケンスが制御され、次に実行する手順が決定されます。シーケンシャル ワークフローには条件分岐やループを含めることもできますが、その遷移は予測可能です。あるタスクを完了するために一連の規定の手順を実行する必要がある場合や、システム管理、ビジネス間のオーケストレーション、ビジネス ルールの処理などのシナリオには、シーケンシャル ワークフローの使用を検討します。
- **ステート マシン ワークフロー スタイル**: このスタイルでは、ワークフローが一定の状態になり、イベントの発生を待機してから別の状態に遷移します。イベント駆動型のシナリオ向けに設計されたワークフロー、ユーザー インターフェイスのページ フロー (ウィザードのインターフェイスなど)、または注文に含まれるデータに基づいて手順とプロセスが適用される注文処理システムが必要な場合は、ステート マシン ワークフローの使用を検討します。
- **データ ドリブン ワークフロー スタイル**: このスタイルでは、ドキュメントに含まれる情報によって、ワークフローで実行するアクティビティが決定されます。このスタイルは、ドキュメント承認プロセスなどのタスクに適しています。

手順 2 - オーサリング モードを選択する

ワークフローは、コード、マークアップ言語、またはコードとマークアップの両方の組み合わせを使用して作成できます。使用する手法は、ソリューションに求められるオーサリング モードの要件によって異なります。選択するオーサリング モードは、アプリケーションをパッケージ化および配布する方法にも影響します。選択できるオプションは次のとおりです。

- **コードのみ**: このオプションは、時間が経過してもワークフローがあまり変化しない場合、マークアップでは簡単に表現できない複雑なビジネス ルールがある場合、ビジュアル デザイナーを使用したマークアップの作成よりもマネージ コードの作成に開発チームが慣れている場合、またはマークアップのオプションでは実現できない新しいワークフローの種類を作成する場合に使用します。コードのみのワークフローは、ソース コード管理システムにも簡単に統合できます。

- **コード分離**: このオプションは、ビジネス コンポーネントにカプセル化された複雑なビジネス ルールがある場合、またはユーザーや管理者がワークフロー デザイナーを使用してワークフローの一部の側面を変更できるようにする場合に使用します。
- **マークアップ**: このオプションは、時間の経過と共にワークフローが頻繁に変化する場合、ワークフローに関連するビジネス ルールをマークアップ言語で簡単に表現できる場合、新しいワークフローの種類を作成する必要がない場合、およびワークフロー モデルで参照しているワークフローの種類を再構築しなくてもモデルを更新できる柔軟性が必要な場合に使用します。

手順 3 - ルールの処理方法を決定する

この時点では、ワークフロー スタイルを特定して、ワークフローの作成に使用するオーサリング モードを決定する作業が完了しています。次の手順は、ワークフローでビジネス ルールを処理する方法を決定することです。ルールを処理する方法は、ビジネス ルールの複雑さ、永続性、および管理の要件に基づいて決定します。ワークフロー コンポーネントのビジネス ルールの処理については、次の要素を検討します。

- **ルールが複雑な場合**: コードのみ、またはコード分離オーサリング モードの使用を検討します。ビジネス コンポーネントを使用してルールを実装およびカプセル化できるので、ワークフローで実行を調整できます。
- **ルールが永続的ではない場合**: 単純なルールやデータ ドリブン ルールには、マークアップ オーサリング モードの使用を検討します。ただし、ルールが外部システム (ビジネス ルール エンジンなど) で管理されている場合は、コードのみ、またはコード分離オーサリング モードの使用を検討します。
- **ビジネス ユーザー、管理者、またはアナリストがルールを管理する場合**: ビジュアル デザイナーなどのルール編集機能を提供するマークアップ オーサリング モードを使用するソリューション、またはドメイン固有言語 (DSL) をサポートするソリューションの使用を検討します。ただし、ルールが外部システム (ビジネス ルール エンジンなど) で管理されている場合は、コード分離オーサリング モードの使用を検討します。

手順 4 – ワークフロー ソリューションを選択する

ワークフローに関するワークフロー スタイル、オーサリング モード、およびルール処理の要件を理解したので、次はワークフロー ソリューションを選択します。各ソリューションで提供される機能を基準にソリューションを選択します。マイクロソフト プラットフォームで使用できるテクノロジには、次のようなものがあります。

- **Windows Workflow Foundation (WF)**: シーケンシャル ワークフロー、ステート マシン ワークフロー、およびデータ ドリブン ワークフローを作成するための開発者中心のソリューションが提供されます。また、コードのみ、コード分離、およびマークアップ オーサリング モードがサポートされます。デザイナーは、Visual Studio 2005 では拡張機能によりサポートされ、Visual Studio 2008 以降では直接サポートされます。WF には、安全で信頼できるトランザクションによるデータ交換のためのプロトコル機能、アクティビティの追跡、およびさまざまなトランスポートとエンコードのオプションが用意され、システムのシャットダウンと再起動後も継続して実行される実行時間の長いワークフローがサポートされます。
- **ワークフロー サービス**: Windows Communication Foundation (WCF) と Windows Workflow Foundation (WF) が統合されることで、ワークフロー向けの WCF ベースのサービスが提供されます。Microsoft .NET Framework 3.5 以降、WCF は、サービスとして公開されるワークフローをサポートし、ワークフロー内からサービスを呼び出す機能を提供するように拡張されました。また、Microsoft Visual Studio 2008 には、ワークフロー サービスをサポートする新しいテンプレートとツールが用意されています。
- **Microsoft Office SharePoint Services (MOSS)**: WF に基づくワークフローのサポートを提供するコンテンツ管理とコラボレーションのプラットフォームで、 Microsoft Office SharePoint® サーバーに関連するヒューマン ワークフローとコラボレーション向けのソリューションが提供されます。Web インターフェイスを使用して SharePoint リスト アイテムに関連する承認ベースのワークフローを定義したり、SharePoint Designer または Visual Studio の Windows ワークフロー デザイナーを使用して条件付きワークフローやデータ ドリブン ワークフローを定義したりすることができます。ワークフローをカスタマイズするには、Visual Studio で WF オブジェクト モデルを使用できます。ただし、MOSS の使用が適しているのは、ビジネス レイヤーが単一の SharePoint サイトに限定され、他のサイトに存在する情報にアクセスする必要がない場合だけです。
- **BizTalk Server**: シーケンシャル、ステート マシン、およびデータ ドリブンの各ワークフローがサポートされ、コード分離とマークアップ オーサリング モードもサポートされます。電子データ交換 (EDI) 形式、XML 形式、またはその両方を使用して企業間の電子文書交換関係を有効にします。

BizTalk Server には、実行時間の長い疎結合されたビジネス プロセスや、信頼できるストア アンド フォワード メッセージング機能を備えたワークフローを設計して実行するための、強力なオーケストレーション機能が備わっています。BizTalk Server では、アダプターを通じて異なるアプリケーションやシステムが統合され、ビジネス ルール エンジンとビジネス アクティビティ監視が提供されます。マイクロソフト以外のシステムとのやり取り、EDI の実行、または Enterprise Service Bus (ESB) パターンの実装が必要な場合は、BizTalk ESB Toolkit の使用を検討してください。

手順 5 – ワークフローをサポートするようにビジネス コンポーネントを設計する

一般に、別個のコンポーネント内で実行される多段階のプロセスや実行時間が長いプロセスを伴うワークフローを実装する必要があるので、適切な例外を公開してワークフロー内のすべての障害状態をハンドルできるようにする必要があります。ビジネス ワークフローを設計する際には、応答が不要なメソッド呼び出しの使用を検討するか、十分な応答時間を確保する必要があります。一連の規定の手順を順序に従って同期した状態を保ちながらコンポーネントで処理を実行する必要がある場合は、パイプライン パターンの使用を検討します。また、プロセスの手順を非同期かつ順不同に実行できる場合は、イベント パターンの使用を検討します。
次のセクションでは、マイクロソフト プラットフォームで使用できるテクノロジを使用してワークフローを設計する方法を理解するのに役立つ情報を提供します。

Windows Workflow Foundation

Windows Workflow Foundation (WF) を使用して設計するビジネス コンポーネントには、カスタム ワークフロー、アクティビティ、状態オブジェクト、カスタム サービスなどがあります。必要なコンポーネントは、選択したワークフロー スタイルとオーサリング モードによって異なります。次に、WF を使用して 3 種類の基本的なワークフロー、カスタム サービス、およびワークフロー マークアップを作成する際の手法を示します。

- シーケンシャル ワークフローを設計する場合、Activity クラスを定義するか既存の Activity クラスを使用し (コードのみおよびコード分離)、カスタム ワークフロー クラスを定義し (コードのみ)、ワークフロー コンポーネントと連携するビジネス プロセス コンポーネントを定義します (コードのみ)。
- ステート マシン ワークフローを設計する場合、プロセスのさまざまな状態を表すために使用する状態クラスを定義し (コードのみおよびコード分離)、状態変化をトリガーするイベントを使用するか既存の

イベントを使用し (コードのみおよびコード分離)、状態遷移を管理する Activity クラスを定義するか既存の Activity クラスを使用し (コードのみおよびコード分離)、カスタム ワークフロー クラスを定義し (コードのみ)、ワークフロー コンポーネントと連携するビジネス プロセス コンポーネントを定義します (コードのみ)。

- データ ドリブン ワークフローを設計する場合、Activity クラスを定義するか既存の Activity クラスを使用し (コードのみおよびコード分離)、データ プロバイダーと通信する Condition クラスを定義するか既存の Condition クラスを使用し (コードのみおよびコード分離)、カスタム ワークフロー クラスを定義し (コードのみ)、ワークフロー コンポーネントと連携するビジネス プロセス コンポーネントを定義します (コードのみ)。
- カスタム サービスを設計する場合、サービスと通信する Activity クラスを定義するか既存の Activity クラスを使用し、必要な操作をサポートするサービス インターフェイスを定義し、実証済みの慣例を使用してサービスを設計し、サービスの適切なホスト (IIS、Windows プロセス アクティブ化サービス (WAS)、または WorkflowServiceHost) を選択します。
- ワークフロー マークアップを設計する場合、Visual Studio のデザイナー (Visual Studio 2005 では拡張機能として使用でき、Visual Studio 2008 以降では組み込みの機能です) または SharePoint Designer を使用して、SharePoint リストに基づくワークフローを作成できます。また、サード パーティ製のデザイナーを使用して、サード パーティ製品に関連するマークアップを使用したり、適切な XAML 構文を使用してコードを手作業で記述したりすることもできます。

BizTalk Server

BizTalk Server では、コード分離オーサリング モードとマークアップ オーサリング モードのどちらかをサポートできます。BizTalk Server を使用する場合、BizTalk Server オーケストレーションで使用されるワークフロー コンポーネントを設計しなければならないことがあります。ワークフロー コンポーネントには、たとえば、アダプターやコネクターがあります。また、ワークフローで必要なオーケストレーションを提供するサービスの作成、または BizTalk Server のワークフローからの要求を処理するビジネス コンポーネントの設計が必要になることもあります。カスタム コンポーネントを作成せずに BizTalk Server を使用することもできます (マークアップ オーサリング モードを使用している場合です)。つまり、単純な操作しか必要ない場合は、BizTalk Server のメッセージ変換機能や関数定義機能を利用できます。次に、BizTalk Server を使用してワークフローを作成する際の手法を示します。

- BizTalk Server のワークフロー コンポーネントを設計する場合、適切なインターフェイスを実装するクラスを定義してから、そのクラスを COM に登録します。

- BizTalk Server のビジネス コンポーネントを設計する場合、必要な操作をサポートするクラスを定義します。必要に応じてオーケストレーションによって呼び出されるビジネス コンポーネント内でアトミックなトランザクションを開始できます。また、実証済みの慣例を使用して必要な操作をサポートするビジネス レイヤーを設計する必要があります。
- カスタム サービスを設計する場合、サービスと通信できる BizTalk Server のクラスを作成するか既存のクラスを使用し、必要な操作をサポートするサービス インターフェイスを定義し、実証済みの慣例を使用してサービスを設計し、サービスに適したホスト (IIS または WAS) を選択します。

図 1 に、このすべてのコンポーネントが連携して BizTalk Server のワークフローをサポートするしくみを示します。

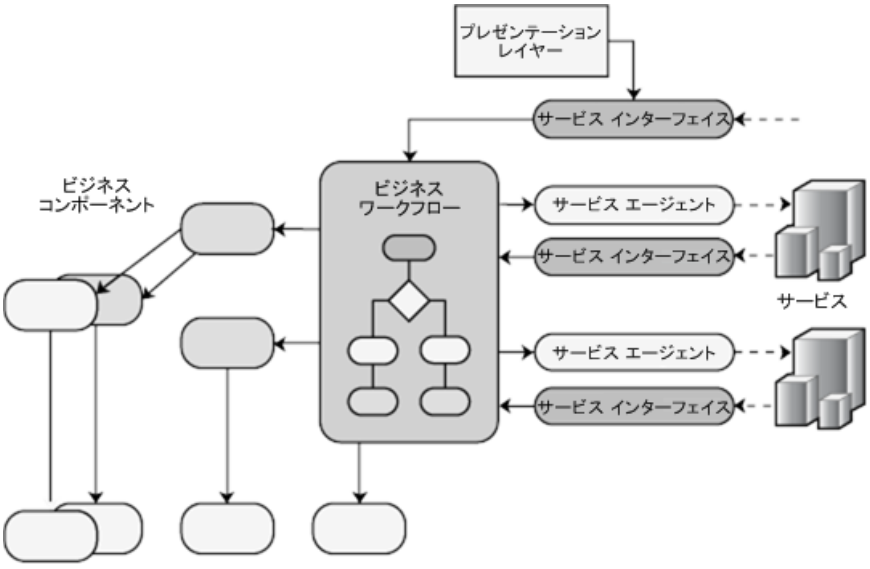


図 13

連携して BizTalk Server のワークフローをサポートするコンポーネント

BizTalk Server と ESB の併用

Microsoft Enterprise Service Bus (ESB) Toolkit は、接続型のサービス指向アプリケーションの構築に重点を置いた機能によって BizTalk Server を拡張します。ESB Toolkit は、メッセージング環境をサポートおよび実装するコンポーネントで構成され、メッセージ ベースのエンタープライズ アプリケーションを容易に構築できるようにします。ESB Toolkit には、次のコンポーネントが用意されています。

- **ESB Web Services**: このサービス群では、Microsoft ESB Toolkit の主要な機能が提供されます。提供されるサービスは、次のとおりです。

    - Itinerary on-ramp Web service: 外部のメッセージを受け取り、メッセージを処理するために送信します。
    - Resolver Web service: 外部のアプリケーションが Resolver Framework を呼び出して、Resolver Framework でサポートされる解決メカニズム (ビジネス ルール ポリシー、UDDI 登録、静的呼び出し、WS-MetadataExchange インターフェイス、メッセージの内容など) に基づいて ESB エンドポイントを参照できるようにします。
    - Transformation Web service: メッセージの内容を変換してビジネス要件を満たす機能を提供します。変換は、受信メッセージに対して直接、または BizTalk Server の MessageBox データベースから取得したメッセージに対して実行されます。
    - Exception Handling Web service: 外部ソースの例外メッセージを受け付け、ESB Exception Management Framework に公開します。その後、例外プロセッサ パイプラインによって例外メッセージが正常化および追跡されて、ESB Management Portal に公開されます。
    - UDDI Web service: アプリケーションやユーザーが、サービス名、ビジネス プロバイダー、またはビジネス カテゴリに基づいてエンドポイントを参照できるようにします。また、アプリケーションやユーザーが UDDI レポジトリに格納されたビジネス プロバイダー、サービス、およびカテゴリを操作できるようにします。
    - BizTalk Operations Web service: BizTalk Server ホスト、オーケストレーション、アプリケーション、および状態についての情報を公開します。

- **ESB Management Portal**: 例外とエラーの追跡、メッセージの再送信、警告と通知、UDDI 統合、レポートと分析、構成機能などの機能を提供します。
- **ESB Pipeline Interop Components**: BizTalk Server パイプラインで使用するための Java Messaging Service (JMS) と名前空間のコンポーネントが用意されています。
- **Exception Management Framework**: BizTalk Server メッセージング サブシステムとオーケストレーション サブシステムの両方から例外をキャプチャして、エラー メッセージを生成できます。
- **ESB Resolver and Adapter Provider Framework**: エンドポイントと変換の要件を動的に解決したり、メッセージをルーティングしたりするための、プラグ可能で構成可能なアーキテクチャを実装します。
- **Itinerary Processing**: このメカニズムは、複数のサービス呼び出し、またはルーティングや変換の要求を動的に記述、送信、および実行するための手軽な機能を提供します。

- **ESB サンプル アプリケーション**: Microsoft ESB Toolkit の使用方法を説明し、ESB Toolkit で提供される機能を SOA アプリケーションや ESB アプリケーションで活用する方法を紹介します。

Windows Workflow Foundation と BizTalk Server の併用

Windows Workflow Foundation (WF) または BizTalk Server 単独では、実装が必要なワークフローを完全にサポートできない場合が多数考えられます。このような状況に直面した場合、1 つのアプリケーションで両方のワークフロー ソリューションの適切な機能を利用できることがよくあります。BizTalk Server ルール エンジンと通信するコードのみのオーサリング モードの WF コンポーネントを使用してビジネス ルール ワークフローを実装する場合、BizTalk Server オーケストレーションから呼び出す必要がある既存の WF ワークフローが存在する場合、BizTalk Server オーケストレーションを実行する必要がある SharePoint ワークフローを作成している場合、または WF ワークフローを異種システムやレガシ システムと統合する必要がある場合に、WF と BizTalk Server の併用を検討します。

関連情報

ワークフロー テクノロジに関する Web リソースに簡単にアクセスするには、http://www.microsoft.com/architectureguide (英語) でオンライン版の参考文献を参照してください。

- Windows Workflow Foundation プログラミング入門 (http://msdn.microsoft.com/ja-jp/library/ms734696.aspx)
- Microsoft BizTalk ESB Toolkit (http://msdn.microsoft.com/en-us/library/dd897973.aspx、英語)

15

データ レイヤーのコンポーネントの設計

概要

データ レイヤーのコンポーネントでは、システムの境界内にホストされているデータと、ネットワークに接続している他のシステムで公開されているデータへのアクセスを提供します。データ レイヤーには、システムの境界内にホストされているデータにアクセスする機能を提供するデータ アクセス コンポーネントや、その他のバックエンド システムで公開されているデータに Web サービスを通じてアクセスする機能を提供するサービス エージェント コンポーネントなどが含まれます。また、ヘルパー関数やユーティリティを提供するコンポーネントが含まれることもあります。

この章では、データ コンポーネントを設計する基本的な手順を理解するのに役立つ情報を提供します。まず、アクセスするデータに関連する制約を特定します。制約を特定すると、適切なデータ アクセス テクノロジを選択するのに役立ちます。次に、マッピングの方針を選択して、データ アクセスの手法を決定します。この時点で、使用するビジネス エンティティとエンティティの形式も特定します。その後、データ アクセス コンポーネントをデータ ソースに接続する方法を決定できます。最後に、データ ソースの例外を管理するためのエラー ハンドル戦略を決定します。

手順 1 – データ アクセス テクノロジを選択する

適切なデータ アクセス テクノロジを選択する際には、使用するデータの種類と、そのデータをアプリケーション内で操作する方法について考慮する必要があります。適切なテクノロジは、シナリオによって異なります。次のガイドラインを使用して、アプリケーションのシナリオを使用可能なデータ アクセス テクノロジ ソリューションと対応付けます。

- **ADO.NET Entity Framework:** データ モデルを作成してリレーショナル データベースにマップする場合、継承を使用して 1 つのクラスを複数のテーブルにマップする場合、Microsoft SQL Server 製品ファミリ以外のリレーショナル ストアにクエリする場合は、ADO.NET Entity Framework (EF) の使用を検討します。柔軟性のあるスキーマを使用してリレーショナル モデルにマップする必要があるオブジェクト モデルが存在し、オブジェクト モデルからマッピング スキーマを分離する柔軟性が必要な場合は、EF が適しています。EF を使用する際には、次の機能の使用も検討します。
    - **LINQ to Entities**: 厳密に型指定されたエンティティにクエリを実行する必要がある場合や、LINQ 構文を使用してリレーショナル データにクエリを実行する必要がある場合は、LINQ to Entities の使用を検討します。
- **ADO.NET Data Services フレームワーク**: ADO.NET Data Services は EF をベースに構築されているので、REST インターフェイスを通じてエンティティ モデルの一部を公開することができます。RIA や n ティアのリッチ クライアント アプリケーションを開発していて、リソースを中心としたサービス インターフェイスを通じてデータにアクセスする必要がある場合には、ADO.NET Data Services フレームワークの使用を検討します。
- **ADO.NET Core**: 低レベルの API を使用して、アプリケーションでデータ アクセスを完全に制御する必要がある場合、ADO.NET プロバイダーへの既存の投資を活用する必要がある場合、データベースへのアクセスに従来のデータ アクセス ロジックを使用する場合は、ADO.NET Core の使用を検討します。また、他のデータ アクセス テクノロジで提供される追加の機能が不要な場合や、ネットワークに接続されていない状態でのデータ アクセス エクスペリエンスをサポートする必要がある場合は、ADO.NET Core が適しています。
- **Sync Services for ADO.NET**: 不定期に接続するシナリオをサポートする必要があるアプリケーションを設計する場合や、データベース間のコラボレーションが必要な場合は、Sync Services for ADO.NET の使用を検討します。
- **LINQ to XML**: アプリケーションで XML データを使用していて、LINQ 構文を使用してクエリを実行する必要がある場合は、LINQ to XML の使用を検討します。

マイクロソフト プラットフォームで使用できるデータ アクセス テクノロジの詳細については、付録 C「データ アクセス テクノロジ」を参照してください。

手順 2 – データ ストアからビジネス オブジェクトを取得して格納する方法を選択する

データ ソースの要件を特定したら、次はビジネス オブジェクトやビジネス エンティティをデータ ストアから読み込み、データ ストアに格納する際の方針を選択します。一般的に、オブジェクト指向のデータ モデルとリレーショナル データ ストアの間にはインピーダンス不整合が存在するので、この 2 つのデータ間の変換は困難な場合があります。この不整合に対処するための手法はいくつかありますが、これらの手法ではデータ型、構造、トランザクションの技法、データの操作方法などが異なります。最も一般的なのは、オブジェクト/リレーショナル マッピング (O/RM) ツールと O/RM フレームワークを使用する手法です。アプリケーションで使用するエンティティの種類は、エンティティをデータ ソース構造にマップする方法を決定する主な要因です。次のガイドラインは、データ ストアからビジネス オブジェクトを取得して格納する方法を選択するのに役立ちます。

- ドメイン エンティティとデータベース間でデータを変換する O/RM フレームワークの使用を検討します。データベース スキーマを完全に制御できる新しい環境で作業する場合は、O/RM ツールを使用して、オブジェクト モデルをサポートするスキーマを生成し、データベースとドメイン エンティティの間にマッピングを提供できます。既存のデータベース スキーマを使用しなければならない既存の環境で作業する場合は、O/RM ツールをドメイン モデルとリレーショナル モデルの間のマッピングに使用できます。
- OO 設計に関連する一般的なパターンは、ドメイン モデルです。これは、ドメイン内のオブジェクトに基づくエンティティのモデリングをベースとした手法です。ドメイン駆動設計の技法の詳細については、第 13 章「ビジネス エンティティの設計」を参照してください。
- エンティティを正しくグループ化して、高度なレベルの結合を実現します。場合によっては、ドメイン モデル内にオブジェクトを追加して、その関連するエンティティを集計ルートにグループ化しなければならないことがあります。
- Web アプリケーションや Web サービスを使用する場合は、エンティティをグループ化して、必要なデータのみが含まれるドメイン エンティティを部分的に読み込むオプションを提供します。このオプションにより、メモリ内で初期化されたドメイン モデルをユーザーごとに保持することを回避してリソースの使用を最小限に抑えて、アプリケーションではより高い負荷を処理できるようになります。

手順 3 – データ ソースへの接続方法を決定する

データ アクセス コンポーネントがデータ ソースにマップされるしくみがわかったら、次は、データ ソースに接続して、ユーザー資格情報を保護して、トランザクションを実行する方法を特定する必要があります。次のセクションのガイドラインは、適切な手法を選択するのに役立ちます。

- 接続
- 接続プール
- トランザクションと同時実行

接続

データ ソースへの接続は、データ レイヤーの重要な部分です。データ レイヤーでは、データ アクセス インフラストラクチャを使用して、すべてのデータ ソース接続を調整します。接続の管理と作成には、データ レイヤーとデータ ソースの両方の貴重なリソースを使用します。データ ソースに接続する適切な技法を設計するには、次のガイドラインを使用します。

- データ ソースへの接続はできるだけ後で開き、できるだけ早く閉じるようにします。このようにすることで、リソースがロックされる時間をできる限り短くして、他のプロセスでリソースをもっと自由に使用できるようになります。不揮発性データがある場合は、オプティミスティック同時実行制御を使用して、データベース データのロックによるコストを削減します。これにより、データベースの行のロックによるオーバーヘッド (ロック中は接続を開いたままにしておかなければならないことを含む) を回避できます。
- できる限り、1 つの接続でトランザクションを実行します。このようにすると、分散トランザクション コーディネーターのサービスを要求することなく、ADO.NET のトランザクション機能を使用できます。
- 接続プールを使用し、擬似ロード シナリオの実行によって得られた結果に基づいてパフォーマンスを調整します。データ クエリの接続の分離レベルを調整することを検討します。高いスループットが必要なアプリケーションを構築する場合、特殊なデータ操作は、トランザクションの残りの部分よりも低い分離レベルで実行できます。複数の分離レベルを併用するとデータの一貫性が低下する可能性があるので、この方法を使用するかどうかはケース バイ ケースで慎重に分析する必要があります。
- セキュリティ上の理由から、接続情報の格納にシステム/ユーザー データ ソース名 (DSN) を使用することは避けます。

- データ ソースへの接続が失われたり、タイムアウトしたりした場合に対処するための再試行ロジックを設計します。
- 可能であれば、データベース サーバーへのラウンド トリップの回数を減らすために、コマンドをバッチ処理してからデータベースに対して実行します。

もう 1 つの重要な考慮事項は、データ ソースへのアクセスに関連するセキュリティ要件です。つまり、データ ソースでデータ アクセス コンポーネントを認証する方法や、認証の要件について考慮する必要があります。データ ソースに接続するための安全な手法を設計するには、次のガイドラインを使用します。

- SQL Server 認証ではなく Windows 認証を使用します。Microsoft SQL Server を使用している場合は、信頼関係のあるサブシステムの認証に Windows 認証を使用することを検討します。
- SQL Server 認証を使用する場合は、カスタム アカウントと強力なパスワードを使用し、データベース ロールを使用して SQL Server における各アカウントの許可を制限し、接続文字列の格納に使用するファイルに ACL を追加し、構成ファイルに含まれる接続文字列を暗号化するようにします。
- データベースで最小限の特権を持つアカウントを使用して、呼び出し元がデータ レイヤーに ID 情報を送信するように要請して、アクセスを監査するようにします。
- データベースには、ユーザーの検証に使用する (プレーンテキストまたは暗号化された) パスワードを格納しないようにします。代わりに、salt 値 (ハッシュ関数への入力の 1 つとして使用されるランダムなビット値) を使用するパスワード ハッシュを格納します。
- SQL ステートメントを使用してデータ ソースにアクセスする場合は、信頼境界を理解し、SQL インジェクション攻撃を防ぐために、文字列の連結ではなく、パラメーター化された手法を使用してクエリを作成します。
- ネットワーク経由で SQL Server と送受信する機密データを保護します。Windows 認証では資格情報は保護されますが、アプリケーション データは保護されないことに注意が必要です。IPSec や SSL を使用して、チャネルのデータを保護します。

接続プール

接続プールを使用すると、アプリケーションでは、プールから接続を再利用したり、適切な接続がない場合には新しい接続を作成してプールに追加したりできます。アプリケーションが接続を終了すると、接続はプールに解放されますが、基になる接続は開いたままの状態が維持されます。つまり、ADO.NET では、接続が確立されるたびに、新しい接続を作成して、データ ソースに対して接続を開く必要はありません。プールにある開かれた接続ではリソース

が消費されますが、プールされた適切な接続を使用できる場合には、データ アクセスの遅延が減少し、アプリケーションはさらに効率的に実行されます。接続プールに影響する他の問題には、次のようなものがあります。

- 接続プールの有効性を最大限に高めるには、"信頼関係のあるサブシステム" セキュリティ モデルの使用を検討し、偽装をできるだけ避けます。使用する資格情報の数を最小限に抑えることにより、プールされた既存の接続の再利用率を高めて、接続プールのオーバーフローの変化を少なくできます。すべての呼び出しで異なる資格情報を使用する場合、ADO.NET では毎回新しい接続を作成する必要があります。
- 長い間開いたままになっている接続は、サーバーのリソースを保持し続けることになります。このような状況が発生する一般的な原因は、(たとえば、接続を明示的に閉じないで、接続が範囲外になるまで破棄しないことにより) 接続が早く開かれ、遅くまで閉じられないことにあります。
- 接続が開いている間だけ有効な DataReader オブジェクトを使用する場合は、接続が長い間開かれたままになる可能性があります。

トランザクションと同時実行

アプリケーションにビジネスクリティカルな操作が含まれる場合は、操作をトランザクションにラップすることを検討します。トランザクションでは、関連する操作を 1 つの atomic 単位としてデータベースで実行して、データベースの整合性を確保できます。トランザクションは、すべての情報と操作の処理が完了して、データベースへの変更が確定した場合にのみ完了したと見なされます。トランザクションでは、エラーが発生した場合に元に戻す (ロールバックする) というデータベース操作をサポートしています。これは、データベース内のデータの整合性を維持するのに役立ちます。次のガイダンスは、トランザクションを設計するのに役立ちます。

- 1 つのデータ ソースにアクセスする場合は、できるだけ接続ベースのトランザクションを使用します。手動 (明示的な) トランザクションを使用する場合は、ストアド プロシージャにトランザクションを実装することを検討します。トランザクションを使用できない場合は、データ ストアを前の状態に戻すための、補正操作を実装します。
- 実行時間の長いアトミックなトランザクションを使用する場合は、長時間データをロックしたままにしないようにします。このようなシナリオでは、補正のためのロックを使用します。トランザクションが完了するまでに時間がかかる場合は、完了時にクライアントにコールバックする非同期トランザクションの使用を検討します。また、大量のトランザクションを使用する同時実行アプリケーションでは、複数のアクティブな結果セットを使用して、潜在的なデッドロックの問題を回避することを検討します。

- 同時接続ユーザーによるデータの競合が発生する可能性が低い場合 (ユーザーが主にデータを追加したり別の行を編集したりする場合など) は、データ アクセス時にオプティミスティック ロックを使用して、適用された最後の更新を有効にすることを検討します。同時接続ユーザーによるデータの競合が発生する可能性が高い場合 (ユーザーが同じ行を編集する可能性がある場合など) は、データ アクセス時にペシミスティック ロックを使用して、最新のバージョンのみに更新を適用できるようにすることを検討します。また、アプリケーションで静的データにアクセスする場合や、スレッドを使用して非同期操作を実行する場合にも、同時実行の問題について考慮する必要があります。静的データは本質的にスレッド セーフではないので、あるスレッドでデータを変更すると、同じデータを使用している他のスレッドに影響を及ぼします。
- ロックが適用される時間をできる限り短くして、同時実行性を向上するために、トランザクションはできるだけ短くします。ただし、1 つの操作を実行するのに複数の呼び出しが必要な場合は、短く単純なトランザクションによって、chatty な (小さな要求が多い) インターフェイスになる可能性があることを考慮する必要があります。
- 適切な分離レベルを使用します。データの一貫性と競合との間にはトレードオフが存在します。分離レベルを高くすると、データの一貫性は高まりますが、全体的な同時実行性は低下します。分離レベルを低くすると、競合が減ることによりパフォーマンスは向上しますが、一貫性は低下します。

一般的に、トランザクションのサポートは、次の 3 種類の中から選択できます。

- .NET Framework の一部として提供される System.Transactions 名前空間のクラスでは、暗黙および明示的なトランザクション サポートの両方が提供されます。トランザクションのサポートが必要な新しいアプリケーションを開発している場合や、複数の永続的でないリソース マネージャーをまたぐトランザクションが存在する場合は、System.Transactions 名前空間の使用を検討します。ほとんどのトランザクションにお勧めの手法は、System.Transactions 名前空間の TransactionScope オブジェクトで提供される暗黙モデルを使用することです。暗黙のトランザクションは手動 (明示的な) トランザクションほど高速ではありませんが、簡単に作成できます。また、暗黙のトランザクションを使用すると、柔軟で保守が容易な中間ティア ソリューションを実現できます。トランザクションで暗黙モデルを使用しない場合は、System.Transactions 名前空間の Transaction クラスを使用して手動トランザクションを実装します。
- ADO.NET トランザクションは、1 つのデータベース接続に基づいたものになります。これは、1 つのデータ ストアでクライアントがトランザクションを制御する際の、最も効率的な手法です。既に ADO.NET トランザクションを使用しているアプリケーションを拡張する場合、データベースへのアク

セスに ADO.NET プロバイダーを使用しており、トランザクションが 1 つのデータベースに限定されている場合、または .NET Framework 2.0 がサポートされていない環境にアプリケーションを展開する場合は、ADO.NET トランザクションの使用を検討します。トランザクションで実行する操作の開始、コミット、およびロールバックには、ADO.NET コマンドを使用します。

- T-SQL (データベース) トランザクションは、データベースで実行されるコマンドで制御されます。このトランザクションは、トランザクションを全面的にデータベースが管理している、単一のデータ ストアでサーバーによって制御されるトランザクションとして使用すると最も効率的に機能します。トランザクションで管理する必要があるすべての変更をカプセル化するストアド プロシージャを開発する場合や、同じようなストアド プロシージャを使用する複数のアプリケーションが存在し、そのストアド プロシージャ内でトランザクション要件をカプセル化できる場合は、データベース トランザクションの使用を検討します。

手順 4 – データ ソース エラーのハンドルに関する方針を決定する

この手順では、データ ソース エラーをハンドルするための全体的な方針を設計します。データ ソースに関連するすべての例外は、データ アクセス レイヤーでキャッチする必要があります。データ自体に関する例外、データ ソース アクセス エラー、およびタイムアウト エラーは、このレイヤーでハンドルし、エラーがアプリケーションの応答性や機能に影響を及ぼす場合にのみ他のレイヤーに渡す必要があります。次のセクションのガイドラインは、適切な手法を選択するのに役立ちます。

- 例外
- 再試行ロジック
- タイムアウト

例外

一元化された例外管理の方針を使用することで、一貫した方法で例外をハンドルできるようになります。例外ハンドルは横断的関心事なので、レイヤー間で共有できる別個のコンポーネントにロジックを実装することを検討します。信頼境界を越えて伝播する例外、および他のレイヤーやティアに伝播する例外には、特に注意する必要があります。また、ハンドルされていない例外が原因でアプリケーションで信頼性の問題が発生したり、アプリケーションの機密

情報が公開されたりすることがないように、ハンドルされていない例外を考慮した設計を行います。例外管理の方針を設計する際には、次の手法が役立ちます。

- データ アクセス レイヤーでキャッチおよびハンドルする必要がある例外を特定します。多くのデッドロック、接続の問題、およびオプティミスティック同時実行制御のチェックは、データ レイヤーで解決できます。
- データ ソース エラーやタイムアウトが発生した操作の再試行処理を実装することを検討します (ただし、操作を再試行しても問題ない場合に限ります)。
- 適切な例外の伝達に関する方針を設計します。たとえば、例外は、次のレイヤーに渡す前に、必要に応じてログに記録して変換できる境界のレイヤーに伝播するようにします。
- 重大なエラーと例外のログ記録および通知に関する適切な方針を設計して、機密情報を開示しないようにします。
- patterns & practices の Enterprise Library などの既存のフレームワークを使用して、一貫した例外ハンドルと例外管理の方針を実装することを検討します。

再試行ロジック

サーバーやデータベースがフェールオーバーしたときになどに発生するエラーをハンドルする再試行ロジックを設計します。再試行ロジックでは、データベースに接続している間や、データベースに対してコマンド (クエリやトランザクション) を実行している間に発生するすべてのエラーをキャッチする必要があります。このようなエラーが発生する原因は複数あります。エラーが発生した場合、データ コンポーネントでは、すべての既存の接続を閉じて新しい接続を作成することで接続を再確立し、失敗したコマンドを必要に応じて再実行する必要があります。また、このプロセスは、一定回数のみ再試行し、それでもエラーになる場合は、エラーの例外を返すようにします。クエリや要求など、その後のすべての処理の再試行が非同期で実行されるようにして、アプリケーションが応答しない状態になることを避けます。

タイムアウト

接続タイムアウトとコマンド タイムアウトの適切な値を特定することは非常に重要です。接続タイムアウトやコマンド タイムアウトの値を、クライアントのタイムアウト (Web アプリケーションの場合は、ブラウザーや Web サーバーの要求のタイムアウト) の値よりも高く設定すると、データベース接続が開かれる前にクライアントの要求がタイムアウトする可能性があります。また、低い値を設定すると、エラー ハンドラーによって再試行ロジックが呼び出されます。トランザクションの実行中にタイムアウトが発生した場合、接続プールが有効になっていると接続を

閉じた後も、データベース リソースがロックされたままになることがあります。このような場合は、接続を閉じるときに、接続を破棄してプールに返されないようにする必要があります。接続を破棄すると、トランザクションがロールバックされ、データベース リソースが解放されます。

手順 5 – サービス エージェントのオブジェクトを設計する (省略可能)

サービス エージェントは、外部サービスとの通信のセマンティクスを管理し、アプリケーションからさまざまなサービスを呼び出すという特質を分離して、サービスで公開されているデータの形式とアプリケーションで必要な形式との間の基本的なマッピングを行うなどの追加サービスを提供するオブジェクトです。また、キャッシュ、オフライン サポート、または断続的な接続のサポートも実装することがあります。サービス エージェントのオブジェクトを設計するには、次の手順を実行します。

1. 適切なツールを使用してサービス参照を追加します。サービス参照を追加すると、プロキシと、サービスのデータ コントラクトを表すデータ クラスが生成されます。
2. アプリケーションでサービスをどのように使用するかを決定します。ほとんどのアプリケーションでは、サービス エージェントはビジネス レイヤーとリモート サービス間の抽象化レイヤーとして機能し、データの形式にかかわらず一貫したインターフェイスを提供することが可能です。小規模なアプリケーションでは、プレゼンテーション レイヤーからサービス エージェントに直接アクセスすることがあります。

関連情報

データ アクセスについての一般的なガイドラインや情報に関する Web リソースに、より簡単にアクセスするには、http://www.microsoft.com/architectureguide (英語) でオンライン版の参考文献を参照してください。

- .NET Data Access Architecture Guide (http://msdn.microsoft.com/en-us/library/ms978510.aspx、英語)
- Data Patterns (http://msdn.microsoft.com/en-us/library/ms998446.aspx、英語)
- データ層コンポーネントの設計と層間のデータの受け渡し (http://msdn.microsoft.com/ja-jp/library/dd297667.aspx)

16

品質特性

概要

品質特性は、実行時の動作、システム設計、およびユーザー エクスペリエンスに影響を与える全般的な要素です。品質特性は、レイヤーとティアを横断してアプリケーション全体に影響を与える可能性のある関心事を表します。この特性の中には、システム設計全体にかかわるものもあれば、実行時の問題、設計時の問題、またはユーザー中心の問題に固有のものもあります。アプリケーションが、求められる品質特性の組み合わせ (ユーザビリティ、パフォーマンス、信頼性、セキュリティなど) をどの程度備えているかによって、設計の成否とソフトウェア アプリケーションの全体的な品質がわかります。

品質特性のいずれかの要件を満たすようにアプリケーションを設計する際は、他の要件への影響の可能性を考慮し、複数の品質特性間のトレードオフを分析する必要があります。各品質特性の重要性や優先度はシステムによって異なります。たとえば、単一用途のパッケージ製品として販売されているアプリケーションでは、相互運用性は、基幹業務 (LOB) システムほど重要ではありません。

この章では、アプリケーションの設計時に考慮する必要がある品質特性について解説します。この章を最大限に活用するには、まず、次のセクションの表を使用して、品質特性がシステムとアプリケーションの品質要因にどのように対応しているかを理解し、各品質特性の説明を確認します。その後、品質特性ごとに主要なガイドラインを提示しているセクションを使用して、その特性が設計に及ぼす影響を理解し、このような問題に対処するために判断が必要な事項を特定します。この章ではすべての品質特性を紹介しているわけではなく、アーキテクチャに関して適切な検討を行う出発点となる情報を提供していることに注意してください。

一般的な品質特性

次の表に、この章で紹介する品質特性の概要を示します。この表では、設計品質、実行時の品質、システム品質、およびユーザー品質に関連する 4 つの具体的な分野に特性を分類しています。この表は、アプリケーション設計における各品質特性の意味を理解するのに役立ててください。

カテゴリ	品質特性	説明
設計品質	概念的整合性	概念的整合性は、設計全体の一貫性と整合性で定義します。この特性には、コンポーネントやモジュールの設計方法、コーディング方法や変数の命名などの要素が含まれます。
	保守容易性	保守容易性は、システムをある程度簡単に変更できることを表す能力です。機能の追加や削除、エラーの修正、および新しいビジネス要件への対応を行うと、このような変更によってコンポーネント、サービス、機能、およびインターフェイスに影響が及ぶことがあります。
	再利用性	再利用性とは、コンポーネントとサブシステムが他のアプリケーションやシナリオでの使用に対応できることを表す能力です。再利用性により、コンポーネントの重複や実装にかかる時間を最小限に抑えられます。
実行時品質	可用性	可用性は、システムが機能および稼動している時間の割合で定義します。この特性は、事前に定義された期間に対するシステムの総ダウンタイムの割合で評価できます。可用性は、システム エラー、インフラストラクチャの問題、悪意のある攻撃、およびシステム負荷の影響を受けます。
	相互運用性	相互運用性は、1 つまたは複数のシステムが、外部で開発され実行されている他の外部システムと通信したり情報をやり取りして、正常に動作できる能力です。相互運用性のあるシステムを利用すると、社内と社外のどちらでも情報を簡単にやり取りして再利用できます。
	管理容易性	管理容易性は、システム管理者がどれほど簡単にアプリケーションを管理できるかを表す能力です。通常、管理容易性を実現するには、シ

		ステムの監視に使用したりデバッグやパフォーマンス チューニングを行ったりするために公開される、十分かつ便利なインストルメンテーションを使用します。
	パフォーマンス	パフォーマンスは、一定期間に任意の処理を実行するシステムの応答性を表します。パフォーマンスは、待ち時間やスループットに基づいて評価できます。待ち時間は、任意のイベントへの応答にかかる時間です。スループットは、一定時間内に発生するイベント数です。
	信頼性	信頼性は、時間が経過してもシステムを運用し続けられる能力です。信頼性は、一定の期間内にシステムが目的の機能を実行できない状態にならない確率で評価されます。
	スケーラビリティ	スケーラビリティは、システムがパフォーマンスに影響を与えずに負荷の増大に対応できる能力または簡単に拡張できる能力です。
	セキュリティ	セキュリティは、設計した用途を逸脱した悪意のある操作や意図しない操作を防いだり、情報の漏えいや損失を防いだりするシステムの能力です。セキュリティで保護されたシステムは、資産を保護し、情報の不当な変更を阻止することを目的としています。
システム品質	サポート容易性	サポート容易性は、システムが正常に動作しないときに、問題の特定と解決に役立つ情報をシステムが提供する能力です。
	テスト容易性	テスト容易性は、システムとそのコンポーネントでどの程度簡単に、テスト条件を作成し、条件が満たされているかどうか判断するためにそのテストを実施できるかを示す指標です。テスト容易性が優れていると、システムの問題をタイミング良く効果的に分離できます。
ユーザー品質	ユーザビリティ	ユーザビリティは、アプリケーションがユーザーの要求を満たしている程度を表す能力です。ユーザビリティの高いアプリケーションでは、操作が直感的で、ローカリゼーションとグローバリゼーションが容易に行え、障害のあるユーザーがアクセスしやすく、全体的なユーザー エクスペリエンスが高くなります。

以降のセクションでは、各品質特性について詳しく説明し、特性ごとに、主な論点と判断が必要な事項についてのガイダンスを提供します。

- 可用性
- 概念的整合性
- 相互運用性
- 保守容易性
- 管理容易性
- パフォーマンス
- 信頼性
- 再利用性
- スケーラビリティ
- セキュリティ
- サポート容易性
- テスト容易性
- ユーザー エクスペリエンス/ユーザビリティ

可用性

可用性は、システムが機能および稼動している時間の割合で定義します。この特性は、事前に定義された期間に対するシステムの総ダウンタイムの割合で評価できます。可用性は、システム エラー、インフラストラクチャの問題、悪意のある攻撃、およびシステム負荷の影響を受けます。可用性の主な論点は次のとおりです。

- データベース サーバー、アプリケーション サーバーなどの物理ティアで、エラーが発生したり応答しなくなったりし、システム全体の障害につながることがある。システムの物理ティアでフェールオーバーをサポートする設計を検討します。たとえば、Web サーバーのネットワーク負荷分散機能を使用して、負荷を分散し、障害が発生しているサーバーに要求が送信されることを回避します。また、RAID メカニズムを使用して、ディスク エラー発生時のシステム障害を軽減することを検討します。地震や竜巻などの自然災害に備えて、フェールオーバーする地理的に離れた冗長なサイトが必要かどうかも検討します。
- 承認されたユーザーがシステムにアクセスできなくなるサービス拒否 (DoS) 攻撃によって、システムが (多くの場合は必要な処理時間や、ネットワーク構成とネットワークの混雑が原因で) 大量の負荷をすぐに処理できないと、操作が中断されることがある。DoS 攻撃によるシステムの中断を最小限に抑えるには、

攻撃にさらされる領域を縮小し、悪意のある動作を認識します。また、アプリケーションのインストルメンテーションを使用して意図しない動作を公開し、包括的なデータ検証機能を実装します。Circuit Breaker パターンや Bulkhead パターンを使用して、システムの復元性を向上することを検討します。

- リソースの不適切な使用によって可用性が低下することがある。たとえば、リソースを早く取得したことが原因で長期間リソースが保持されていると、リソース スタベーションが発生し、ユーザーからの新しい要求を処理できなくなります。
- アプリケーションの問題によって、システム全体に及ぶ障害が発生することがある。適切な例外ハンドルを設計して、回復が困難なアプリケーション エラーを削減します。
- セキュリティ修正プログラム、ユーザー アプリケーションのアップグレードなど、頻繁な更新によって、システムの可用性が低下することがある。実行時のアップグレードに関する設計方法を検討します。
- ネットワーク障害によってアプリケーションが使用できなくなることがある。不安定なネットワーク接続の処理方法を検討します。たとえば、不定期に接続を確立する機能を備えたクライアントを設計します。
- アプリケーション内の信頼境界を考慮します。また、サブシステムでなんらかのアクセス制御やファイアウォールの機能、および包括的なデータ検証機能を採用して、復元性と可用性を向上します。

概念的整合性

概念的整合性は、設計全体の一貫性と整合性で定義します。この特性には、コンポーネントやモジュールの設計方法、コーディング方法や変数の命名などの要素が含まれます。整合性のあるシステムでは、どの要素が全体的な設計に従っているかわかるので、システムが管理しやすくなります。一方、概念的整合性がないシステムでは、インターフェイスの変更、モジュールの頻繁な廃止、およびタスクの実行方法に関する一貫性の欠如による影響を絶えず受けることになります。概念的整合性の主な論点は次のとおりです。

- 設計にさまざまな関心領域が混在している。関心領域を特定し、論理的なプレゼンテーション レイヤー、ビジネス レイヤー、データ レイヤー、およびサービス レイヤーに適宜グループ化することを検討します。
- 開発プロセスに一貫性がないか、開発プロセスが適切に管理されていない。アプリケーション ライフサイクル管理 (ALM) の評価の実施を検討して、十分に試行された開発ツールと手法を使用します。
- アプリケーション ライフサイクルにかかわるグループ間で、コラボレーションとコミュニケーションが行われていない。開発プロセスのワークフロー、コミュニケーション、およびコラボレーションを容易にするツールを統合した開発プロセスを確立することを検討します。

- 設計とコーディングに関する標準がない。設計とコーディングに関する標準に関する公式のガイドラインを確立し、コードのレビューを開発プロセスに組み込んで、ガイドラインへの準拠を確認することを検討します。
- 既存の (レガシー) システムが原因で、新しいプラットフォームやパラダイムへのリファクタリングと移行のどちらもできないことがある。レガシー テクノロジを使用しない移行パスの作成方法、および外部依存関係からアプリケーションを切り離す方法を検討します。たとえば、Gateway 設計パターンを実装して、レガシー システムと統合します。

相互運用性

相互運用性は、1 つまたは複数のシステムが、外部で開発され実行されている他の外部システムと通信したり情報をやり取りして、正常に動作できる能力です。相互運用性のあるシステムを利用すると、社内と社外のどちらでも情報を簡単にやり取りして再利用できます。通信プロトコル、インターフェイス、およびデータ形式は、相互運用性の主要な考慮事項です。また、標準化も、相互運用性のあるシステムを設計するうえで考慮すべき重要な側面です。相互運用性の主な論点は次のとおりです。

- 異なるデータ形式を使用する外部システムやレガシー システムと相互作用している。複数のシステムを相互運用しながら、個別にシステムを発展したり、置き換えられるようにすることを検討します。たとえば、アダプターを備えたオーケストレーションを使用して、外部システムやレガシー システムに接続して、システム間でデータをやり取りする際にデータを変換します。または、正準的なデータ モデルを使用して、多種多様なデータ形式の操作を制御します。
- 境界があいまいになっているため、あるシステムの成果物が別のシステムに拡散している。 サービス インターフェイス、マッピング レイヤー、またはその両方を使用して、システムを分離する方法を検討します。たとえば、XML ベースのインターフェイスまたは標準的な型を使用するサービスを公開して、他のシステムとの相互運用性をサポートします。まとまりがあり、疎結合されるようにコンポーネントを設計して、柔軟性を最大限に高めて、置換と再利用性を促進します。
- 標準に準拠していない。作業しているドメインの正式な標準や事実上の標準に留意し、新しい専用の標準を作成するのではなく、既存の標準を使用することを検討します。

保守容易性

保守容易性は、システムをある程度簡単に変更できることを表す能力です。アプリケーションの機能を追加または削除して、エラーを修正したり、新しいビジネス要件に対応したりすると、このような変更によってコンポーネント、サービス、機能、およびインターフェイスに影響が及ぶことがあります。保守容易性は、エラーが発生したり、アップグレードのためにシステムを運用環境から削除した後で、システムを通常の運用状態に復元するのにかかる時間にも影響します。システムの保守容易性が向上すると、可用性が高まり、実行時の問題による影響が軽減されます。アプリケーションの保守容易性は、アプリケーションの全体的な品質特性によって異なりますが、保守容易性に直接影響を与える主な論点は多数あります。

- コンポーネントとレイヤーの間に必要以上の依存関係が存在したり、具象クラスと不適切に結合していると、置換、更新、および変更を簡単に実行できなくなる。また、具象クラスへの変更がシステム全体に影響することがある。システムの UI、ビジネス プロセス、およびデータ アクセス機能を明確に区別する、明確に定義されたレイヤー (関心領域) として、システムを設計することを検討します。具象クラスではなく抽象化 (抽象クラスやインターフェイス) を使用して、レイヤー間の依存関係を実装し、コンポーネントとレイヤー間の依存関係を最小限に抑えることを検討します。
- 直接的な通信を使用すると、コンポーネントやレイヤーの物理的な展開を変更できない。適切な通信モデル、通信形式、および通信プロトコルを選択します。アップグレードと保守が簡単で、テストする機会が増えるようプラグ可能なアーキテクチャを設計することを検討します。そのためには、プラグイン モジュールやアダプターを使用して柔軟性と拡張性を最大限に高められるインターフェイスを作成します。
- 認証や承認などの機能のカスタム実装に依存していると、再利用できなくなり、保守の妨げになる。この問題を回避するには、できる限り組み込みのプラットフォーム機能を使用します。
- コンポーネントとセグメントのロジック コードが結合されていないと、コンポーネントやセグメントの保守と置換が困難になり、他のコンポーネントへの不要な依存関係が発生する。まとまりがあり、疎結合されるようにコンポーネントを設計して、柔軟性を最大限に高めて、置換と再利用性を促進します。
- コード ベースが大規模、管理しにくい、不安定、または複雑すぎる。また、回帰が必要なためにリファクタリングに手間がかかる。システムの UI、ビジネス プロセス、およびデータ アクセス機能を明確に区別する、明確に定義されたレイヤー (関心領域) として、システムを設計することを検討します。ビジネス プロセスや動的なビジネス ルールへの変更を管理する方法を検討します。ビジネス プロセスが頻繁に変更される場合は、変更の管理にビジネス ワークフローのエンジンを使用します。ビジネス ルールの値だけが変更されることが多い場合は、ビジネス コンポーネントを使用してルールを実装することを検討しま

す。ビジネス意思決定のルールも変更されることが多い場合は、ビジネス ルール エンジンなどの外部ソースを使用します。

- 既存のコードに、自動化された回帰テスト スイートがない。システムの構築時に、テストを自動化する機能を実装します。テストを自動化することで、システムの機能が検証され、システムのさまざまな部分の機能や各部分の連携方法についてのドキュメントが作成されます。
- ドキュメントがないと、使用、管理、および将来のアップグレードの妨げになることがある。少なくとも、アプリケーションの全体的な構造について説明したドキュメントを用意します。

管理容易性

管理容易性は、システム管理者がどれほど簡単にアプリケーションを管理できるかを表す能力です。通常、管理容易性を実現するには、システムの監視に使用したりデバッグやパフォーマンス チューニングを行ったりするために公開される、十分かつ便利なインストルメンテーションを使用します。システムの監視に使用したりデバッグやパフォーマンス チューニングを行ったりするための、十分かつ便利なインストルメンテーションを公開して、管理しやすいアプリケーションを設計します。管理容易性の主な論点は次のとおりです。

- 正常性の監視、トレース、および診断に関する情報がない。アプリケーションのパフォーマンスに影響を及ぼす可能性がある重大な状態変化を定義する正常性モデルの作成を検討し、このモデルを使用して、管理インストルメンテーションの要件を指定します。イベントやパフォーマンス カウンターなど、状態の変化を検出するインストルメンテーションを実装し、イベント ログ、トレース ファイル、Windows Management Instrumentation (WMI) などの標準的なシステムを通じてこのような変更を公開します。エラーや状態の変化に関する十分な情報を取得してレポートし、正確な監視、デバッグ、および管理を可能にします。また、管理者が自分の監視している環境でアプリケーションの管理に使用できる管理パックの作成を検討します。
- 実行時に構成を変更する柔軟性がない。インフラストラクチャや展開の変更など、運用環境の要件に基づいてシステムの動作を変更できるようにする方法を検討します。
- トラブルシューティング ツールがない。トラブルシューティングに使用できるシステムの状態スナップショットを作成するコードを含めたり、運用と機能に関する詳細なレポートを生成するカスタム インストルメンテーションを含めたりすることを検討します。要求の詳細、モジュールの出力、他のシステムやサービスへのモジュールの呼び出しなど、保守やデバッグに役立つ可能性がある情報をログに記録して監査することを検討します。

パフォーマンス

パフォーマンスは、一定期間に任意の処理を実行するシステムの応答性を表します。パフォーマンスは、待ち時間やスループットに基づいて評価できます。待ち時間は、任意のイベントへの応答にかかる時間です。スループットは、一定時間内に発生するイベント数です。アプリケーションのパフォーマンスは、そのスケーラビリティに直接影響し、スケーラビリティがないとパフォーマンスに影響することがあります。アプリケーションのパフォーマンスが向上すると、共有リソースで競合が発生する可能性が低下するので、多くの場合スケーラビリティが向上します。システムパフォーマンスに影響を及ぼす要因には、特定の操作の要求、要求に対するシステムの応答などがあります。パフォーマンスの主な論点は次のとおりです。

- クライアントの応答が遅くなり、スループットが低下し、サーバー リソースが過剰に使用される。適切な方法でアプリケーションを構築し、十分なリソースがあるシステムに展開します。プロセスやティアの境界を越えて通信する必要がある場合は、最少限の呼び出し回数 (できれば 1 回) で特定のタスクを実行できる粒度の粗いインターフェイスの使用を検討します。また、非同期通信の使用を検討します。
- メモリの使用量が増加すると、パフォーマンスが低下し、過剰なキャッシュ ミス (要求されたデータがキャッシュで見つからないこと) が発生し、データ ストアへのアクセスが増加する。十分かつ適切なキャッシュの方針を設計します。
- データベース サーバーの処理量が増加すると、スループットが低下する。効果的な種類のトランザクション、ロック、スレッド、およびキューの手法を選択します。効率的なクエリを使用して、パフォーマンスへの影響を最小限に抑え、一部のデータだけを表示する場合はすべてのデータを取得しないようにします。効率的なデータベース処理を設計しないと、データベース サーバーで不要な負荷が発生し、パフォーマンスの目標を達成できず、予算内で対応できないことがあります。
- 使用するネットワーク帯域幅が増加すると、応答時間が遅延しクライアント/サーバー システムの負荷が増大する。適切なリモート通信メカニズムを使用して、ティアの間でパフォーマンスの高い通信を設計します。境界を越える遷移数の削減を図り、ネットワーク経由で送信されるデータ量を最小限に抑えます。バッチ処理によりネットワーク経由の呼び出し回数を削減します。

信頼性

信頼性は、時間が経過しても期待どおりにシステムを運用し続けられる能力です。信頼性は、指定した期間内にシステムで障害が発生せず、目的の機能が実行される確率で評価されます。信頼性の主な論点は次のとおりです。

- システムがクラッシュしたり、応答しなくなる。エラーを検出して自動的にフェールオーバーを開始するか、予備またはバックアップ システムに負荷をリダイレクトする方法を特定します。また、既存のシステムに対する要求の失敗が一定数検出されたときに代替システムを使用するコードの実装を検討します。
- 出力に一貫性がない。イベントやパフォーマンス カウンターなど、パフォーマンスの低下や外部システムに送信された要求のエラーを検出するインストルメンテーションを実装し、イベント ログ、トレース ファイル、WMI などの標準的なシステムを通じて情報を公開します。他のシステムやサービスへの呼び出しに関するパフォーマンスと監査情報をログに記録します。
- システム、ネットワーク、データベースなどが使用できないという外部要因により、システムで障害が発生する。不安定な外部システム、通信エラー、およびトランザクション エラーのハンドル方法を特定します。システムをオフラインにしても保留中のリクエストをキューに格納できる方法を検討します。ストア アンド フォワード システム、またはキャッシュされたメッセージ ベースの通信システムを実装して、ターゲット システムにアクセスできないときに要求を保存し、システムがオンラインになったらメッセージを再生できるようにします。Windows メッセージ キューまたは BizTalk Server を使用して、信頼できる 1 回限りの配信を非同期要求で実行することを検討します。

再利用性

再利用性は、新しい機能を追加するために、コンポーネントをほとんどまたはまったく変更せずに他のコンポーネントやシナリオで使用できることを表す能力です。再利用性により、コンポーネントの重複や実装にかかる時間を最小限に抑えられます。小規模で再利用可能なコンポーネントを構築して大規模なシステムで使用するには、まず、さまざまなコンポーネントに共通の特性を特定する必要があります。再利用性の主な論点は次のとおりです。

- 同じ結果をもたらす別個のコードがさまざまな場所で使用されている。たとえば、複数のコンポーネントで類似するロジックが重複していたり、複数のレイヤーやサブシステムで類似するロジックが重複している。アプリケーションの設計を確認して共通の機能を洗い出し、この機能を再利用可能な別個のコンポーネントに実装します。アプリケーションの設計を確認して、検証、ログ記録、認証など、横断的関心事を洗い出し、これらの機能を別個のコンポーネントとして実装します。
- わずかに異なるタスクの実装で複数の類似したメソッドが使用されている。パラメーターを使用して 1 つのメソッドの動作が変化するようにします。
- 複数のシステム間、またはアプリケーション内の異なるサブシステム間で、別のシステムの機能を共有したり再利用するのではなく、同じ機能を実装する複数のシステムが使用されている。他のレイヤーやシステムで使用できるサービス インターフェイスを通じて、コンポーネント、レイヤー、およびサブシステ

ムの機能を公開することを検討します。さまざまなプラットフォームでアクセスして認識できる、プラットフォームに依存しないデータ型やデータ構造を使用します。

スケーラビリティ

スケーラビリティは、システムがパフォーマンスに影響を与えずに負荷の増大に対応できる能力または簡単に拡張できる能力です。スケーラビリティは、垂直方向の拡張 (スケールアップ) と水平方向の拡張 (スケールアウト) の 2 つの方法で向上できます。垂直方向に拡張するには、1 つのシステムに、CPU、メモリ、ディスクなどのリソースを追加します。水平方向に拡張するには、アプリケーションを実行して負荷を分散しているファームに、コンピューターを追加します。スケーラビリティの主な論点は次のとおりです。

- 増加する負荷にアプリケーションが対応できない。スケーラビリティに関するレイヤーとティアの設計方法と、この設計がアプリケーションとデータベースを必要に応じてスケールアップまたはスケールアウトする機能に及ぼす影響を検討します。複数の論理レイヤーを同じ物理ティアに配置して、負荷の分散機能とフェールオーバー機能を最大限に高めると同時に、必要なサーバー数を削減できます。複数のデータベース サーバーにデータを分配して、スケールアップの機会を最大限に拡大し、データ サブセットを柔軟に配置できるようにすることを検討します。サーバー アフィニティを軽減するため、ステートフルなコンポーネントやサブシステムを使用することは可能な限り避けます。
- 応答の遅延や、処理の完了に時間がかかることがユーザーに影響を及ぼす。トラフィックと負荷の急激な増加に対応する方法を検討します。既定のサービスの負荷を超えたり、既存のシステムに対する保留中の要求が多数検出されたときには、別のシステムを使用するコードの実装を検討します。
- システムで、過剰な処理をキューに格納して、負荷が低下したときに処理することができない。ストアアンド フォワード システム、またはキャッシュされたメッセージ ベースの通信システムを実装して、ターゲット システムにアクセスできないときに要求を保存し、システムがオンラインになったらメッセージを再生できるようにします。

セキュリティ

セキュリティは、設計した用途を逸脱した悪意のある操作や意図しない操作がシステムに影響を及ぼす可能性を減らしたり、情報の漏えいや損失を防いだりするシステムの能力です。セキュリティを強化すると、攻撃が成功してシステムの運用に被害が発生する可能性が低くなるので、システムの信頼性が向上します。システムをセキュリティで保護すると、資産が保護され、情報が不当にアクセスまたは変更されなくなります。システムのセキュリティに影響を

及ぼす要素は、機密性、整合性、および可用性です。システムのセキュリティ保護に使用する機能は、認証、暗号化、監査、およびログ記録です。セキュリティの主な論点は次のとおりです。

- ユーザー ID がスプーフィングされる。認証と承認を使用して、ユーザー ID のスプーフィングを防止します。信頼境界を特定し、信頼境界を越えるユーザーを認証して承認します。
- SQL インジェクション、クロスサイト スクリプトなどの悪意のある入力によって損害が発生する。制限、拒否、および削除の原理を使用して、すべての入力の長さ、範囲、形式、および型を検証し、このような損害から保護します。ユーザーに表示するすべての出力をエンコードします。
- データが改ざんされる。サイトを匿名ユーザー、識別されたユーザー、および認証されたユーザーに分割し、アプリケーションのインストルメンテーションを使用して、監視可能な動作をログに記録して、公開します。また、セキュリティで保護されたトランスポート チャネルを使用し、ネットワーク経由で送信される機密データを暗号化して署名します。
- ユーザー操作が否認される。インストルメンテーションを使用して、アプリケーションの重要な動作に関するすべてのユーザー操作を監査してログに記録します。
- 情報が漏えいしたり機密データが失われる。アプリケーションを設計する際には、あらゆる側面において、機密システムやアプリケーション情報がアクセスされたり公開されたりしないように考慮します。
- サービス拒否 (DoS) 攻撃によってサービスが中断する。セッションのタイムアウトを短縮し、DoS 攻撃を検出して被害を軽減するコードやハードウェアを実装します。

サポート容易性

サポート容易性は、システムが正常に動作しないときに、問題の特定と解決に役立つ情報をシステムが提供する能力です。サポート容易性の主な論点は次のとおりです。

- 診断情報がない。システムの動作とパフォーマンスを監視する方法を検討します。Microsoft System Center などのシステム監視アプリケーションの使用を検討します。
- トラブルシューティング ツールがない。トラブルシューティングに使用できるシステムの状態スナップショットを作成するコードを含めたり、運用と機能に関する詳細なレポートを生成するカスタム インストルメンテーションを含めたりすることを検討します。要求の詳細、モジュールの出力、他のシステムやサービスへのモジュールの呼び出しなど、保守やデバッグに役立つ可能性がある情報をログに記録して監査することを検討します。

- トレース機能がない。トレースをコードでサポートするための共通コンポーネント (アスペクト指向プログラミング (AOP) または Dependency Injection など) を使用します。エラーのトラブルシューティングを行えるように、Web アプリケーションのトレースを有効にします。
- 正常性の監視が行われていない。アプリケーションのパフォーマンスに影響を及ぼす可能性がある重大な状態変化を定義する正常性モデルの作成を検討し、このモデルを使用して、管理インストルメンテーションの要件を指定します。イベントやパフォーマンス カウンターなど、状態の変化を検出するインストルメンテーションを実装し、イベント ログ、トレース ファイル、Windows Management Instrumentation (WMI) などの標準的なシステムを通じてこのような変更を公開します。エラーや状態の変化に関する十分な情報を取得してレポートし、正確な監視、デバッグ、および管理を可能にします。また、管理者が自分の監視している環境でアプリケーションの管理に使用できる管理パックの作成を検討します。

テスト容易性

テスト容易性は、システムとそのコンポーネントでどの程度簡単に、テスト条件を作成して、条件が満たされているかどうか判断するためにそのテストを実施できるかを示す指標です。テスト容易性が優れていると、システムの問題をタイミング良く効果的に分離できます。テスト容易性の主な論点は次のとおりです。

- 処理の順列が多い複雑なアプリケーションが一貫した方法でテストされない。これは、アプリケーション全体が密接に結合するように設計されていると、自動テストやきめ細かいテストを実行できないことに起因している場合があります。システムをモジュール形式で設計して、テストをサポートします。インストルメンテーションを提供したり、テスト用の探査や、出力をデバッグするメカニズム、および入力を簡単に指定できる方法を実装したりします。統合性が高く結合性が低いコンポーネントを設計して、システムの他のコンポーネントから分離した状態でコンポーネントをテストできるようにします。
- テスト計画がない。開発ライフ サイクルの早い段階でテストを開始します。テスト中はモック オブジェクトを使用し、単純で構造化されたテスト ソリューションを構築します。
- 手動テストと自動テストの両方に対するテスト カバレッジが適切ではない。ユーザー操作のテストを自動化する方法、およびテストとコード カバレッジを最大化する方法を検討します。
- 入力と出力に整合性がなく、同じ入力に対する出力が異なり、既知の入力パターンをすべて指定しても、出力によって出力ドメインが完全に網羅されない。システムの入出力の指定と解釈を容易にして、テストケースを構築しやすくする方法を検討します。

ユーザー エクスペリエンス/ユーザビリティ

アプリケーションのインターフェイスは、ユーザーを念頭に置いて設計することで、直感的に使用でき、ローカリゼーションとグローバリゼーションが可能で、障害のあるユーザーがアクセスしやすく、全体的なユーザー エクスペリエンスが優れたものにする必要があります。ユーザー エクスペリエンスとユーザビリティの主な論点は次のとおりです。

- タスクに必要な操作数が多すぎる (クリックの回数が多すぎる)。使いやすさを重視して、画面と入力のフロー、およびユーザーの操作のパターンを設計します。
- 多段階のインターフェイスで手順のフローが適切ではない。多段階の操作を簡略化するのが適切な場合は、ワークフローの組み込みを検討します。
- データ要素とコントロールが適切にグループ化されていない。適切なコントロールの種類 (オプション グループ、チェック ボックスなど) を選択し、許容されている UI の設計パターンを使用してコントロールとコンテンツをレイアウトします。
- ユーザーへのフィードバック (特にエラーと例外のフィードバック) が不十分で、アプリケーションが応答しない。Web ページに Asynchronous JavaScript and XML (AJAX) など、ユーザーの対話性を最大限に高めるテクノロジと技法を実装し、クライアント側で入力内容の検証機能を実装することを検討します。バックグラウンド タスク、およびコントロールを配置したり実行時間の長いタスクを実行したりするタスクに、非同期の技法を使用します。

関連情報

品質特性の実装と監査に関する Web リソースに簡単にアクセスするには、
http://www.microsoft.com/architectureguide (英語) でオンライン版の参考文献を参照してください。

- Implementing System-Quality Attributes (http://msdn.microsoft.com/en-us/library/bb402962.aspx、英語)
- Software Architecture in the New Economy (http://msdn.microsoft.com/en-us/library/cc168642.aspx、英語)
- Quality-Attribute Auditing: The What, Why, and How (http://msdn.microsoft.com/en-us/library/bb508961.aspx、英語).
- Michael Feathers 著『レガシーコード改善ガイド』(翔泳社、2009 年)

- Kyle Baley、Donald Belcham 著『Brownfield Application Development in .NET』(Manning Publications Co、2008 年)
- Michael Nygard 著『Release It! 本番用ソフトウェア製品の設計とデプロイのために』(オーム社、2009 年)

17

横断的関心事

概要

多くのアプリケーションの設計には、複数のレイヤーとティアにまたがる共通機能が含まれます。通常、このような共通機能では、認証、承認、キャッシュ、通信、例外管理、ログ記録とインストルメンテーション、検証などの操作をサポートします。このような機能は、アプリケーション全体に影響を及ぼすことから、一般に横断的関心事と呼ばれ、できる限りコード内で 1 か所にまとめる必要があります。たとえば、ログ エントリを生成し、アプリケーション ログに書き込むコードが、複数のレイヤーとティアに分散されている場合に、これらの関心事に関する要件が変更になると (ログの記録先が変更された場合など)、システム全体に分散されている関連コードを更新する必要があります。ログ記録のコードを 1 か所まとめておけば、コードの一部分のみを変更するだけで、動作を変更できます。この章では、アプリケーションで横断的関心事が果たす役割を理解し、アプリケーション内で横断的関心事が使用される領域を特定するための情報を提供します。また、横断的関心事を設計する際の一般的な問題について説明します。横断的関心事の機能はいくつかの手法を使用して処理できます。たとえば、patterns & practices の Enterprise Library などの一般的なライブラリや、メタデータを使用して横断的関心事のコードをコンパイル済みの出力や実行時中に直接挿入するアスペクト指向プログラミング (AOP) のメソッドがあります。

設計に関する一般的な考慮事項

次のガイドラインは、横断的関心事を管理する際の重要なポイントを理解するうえで役立ちます。

- 各レイヤーで必要な機能を検証し、その機能を共通コンポーネント (場合によってはアプリケーションの各レイヤー固有の要件に基づいて構成する汎用コンポーネント) に抽象化できる状況を検討します。このようなコンポーネントは、他のアプリケーションでも再利用できる可能性が高くなります。

- アプリケーションのコンポーネントとレイヤーの物理的な分散状態によっては、複数の物理レイヤーに横断的関心事のコンポーネントをインストールしなければならないことがあります。ただし、その場合でも、再利用性と開発にかかる時間とコストは削減されるので、メリットを得られます。
- Dependency Injection パターンを使用して、構成情報に基づいて、横断的関心事のコンポーネントをアプリケーションに挿入することを検討します。このようにすると、アプリケーションの再コンパイルや再展開を必要としないで、各セクションで使用する横断的関心事のコンポーネントを容易に変更できます。patterns & practices の Unity Library では、Dependency Injection パターンの包括的なサポートを提供します。その他の一般的な Dependency Injection のライブラリには、StructureMap、Ninject、Castle Windsor などがあります (詳細については、この章の最後の「関連情報」を参照してください)。
- 柔軟に構成して、開発時間を短縮できるサードパーティ製のコンポーネント ライブラリを使用することを検討します。patterns & practices の Enterprise Library がその一例で、このライブラリには、キャッシュ、例外管理、認証と承認、ログ記録、検証、および暗号化機能を実装するのに役立つアプリケーション ブロックが含まれています。また、ポリシーの挿入を実装するメカニズムと、依存関係の挿入のコンテナーを実装する、さまざまな横断的関心事のソリューションを実装しやすくするメカニズムも含まれています。Enterprise Library の詳細については、付録 F「patterns & practices の Enterprise Library」を参照してください。Castle Project でも、共通ライブラリが提供されています (詳細については、この章の最後の「関連情報」を参照してください)。
- コードで明示に呼び出すのではなく、アスペクト指向プログラミング (AOP) の技法を使用して、横断的関心事をアプリケーションに実装することを検討します。patterns & practices の Unity メカニズムと Enterprise Library の Policy Injection Application Block では、この手法がサポートされています。その他の例としては、Castle Windsor や PostSharp があります (詳細については、この章の最後の「関連情報」を参照してください)。

設計に関する具体的な問題

これ以降のセクションでは、アーキテクチャを開発する際に考慮すべき主な領域について説明し、各領域で一般的に発生する問題を回避するためのガイドラインを提供します。

- 認証

- 承認
- キャッシュ
- 通信
- 構成管理
- 例外管理
- ログ記録とインストルメンテーション
- 状態管理
- 検証

認証

適切な認証方針を設計することは、アプリケーションのセキュリティと信頼性の両方において重要です。適切な認証方針を設計して実装しないと、スプーフィング攻撃、辞書攻撃、セッション ハイジャックなどの攻撃に対して、アプリケーションが脆弱になる可能性があります。認証の方針を設計する際には、次のガイドラインを考慮します。

- 信頼境界を特定して、信頼境界を越えるユーザーと呼び出しを認証します。呼び出しがクライアントだけでなく、サーバーでも認証される必要がある (相互認証) 可能性を考慮します。
- 強力なパスワードまたはパスワード フレーズの使用を強制します。
- アプリケーション内に複数のシステムがあるか、ユーザーが同じ資格情報を使用して複数のアプリケーションにアクセスできるようにする必要がある場合は、シングル サインオン方針の利用を検討します。
- プレーンテキストの状態でパスワードをネットワーク経由で送らないようにします。また、データベースやデータ ソースにプレーンテキストでパスワードを格納しないようにします。データベースやデータ ソースには、パスワードのハッシュを格納します。

認証方針の設計と認証方針を実装する技法の詳細については、この章の最後の「関連情報」を参照してください。

承認

適切な承認方針を設計することは、アプリケーションのセキュリティと信頼性の両方において重要です。適切な承認方針を設計して実装しないと、情報漏えい、データの改ざん、および特権の昇格に対して、アプリケーションが脆弱になる可能性があります。承認の方針を設計する際には、次のガイドラインを考慮します。

- 信頼境界を特定して、信頼境界を越えるユーザーと呼び出し元を承認します。

- ID、グループ、または役割に基づいて、呼び出し元を承認することで、リソースを保護します。可能であれば、使用する役割の数を制限することで、粒度を最小限に抑えます。
- ビジネスにおける決定には役割ベースの承認を使用することを検討します。役割ベースの承認は、ユーザーをグループ (役割) に分割し、個々のユーザーではなく各役割にアクセス許可を設定するために使用します。このような承認を行うと、管理対象が多数のユーザーではなく、それよりも少数の役割になるので、管理が容易になります。
- システムの監査にはリソース ベースの承認を使用することを検討します。リソース ベースの承認では、リソース自体にアクセス許可を設定します。たとえば、Windows リソースのアクセス制御リスト (ACL) では、元の呼び出し元の ID に基づいてリソースへのアクセス権を決定します。WCF でリソース ベースの承認を使用する場合は、クライアントまたはプレゼンテーション レイヤー、WCF サービス レイヤー経由で、リソースにアクセスするビジネス ロジック コードに元の呼び出し元を偽装する必要があります。
- ID、役割、許可、権利などの要素の情報の組み合わせに基づく、統合された承認をサポートする必要がある場合は、クレーム ベースの承認を使用することを検討します。クレーム ベースの認証を使用すると、承認と認証のメカニズムから承認の規則を分離しやすくする追加の抽象化のレイヤーが提供されます。たとえば、証明書またはユーザー名とパスワードの資格情報を使用してユーザーを認証し、そのクレーム セットをサービスに渡して、リソースへのアクセスを決定できます。

承認方針の設計、および承認方針の実装技法の詳細については、この章の最後の「関連情報」を参照してください。

キャッシュ

キャッシュを使用すると、アプリケーションのパフォーマンスと応答性が向上します。ただし、キャッシュ方針が適切に設計されていないと、パフォーマンスと応答性が損なわれる可能性があります。キャッシュは、データ検索処理の最適化、ネットワーク ラウンド トリップの回避、および不要に重複する処理の回避に使用します。キャッシュを実装するときには、キャッシュ データを読み込むタイミングと、有効期限が切れたキャッシュ データを削除する方法とタイミングを決める必要があります。クライアント側での遅延を回避するには、非同期またはバッチ処理を使用して、よく使用されるデータをキャッシュにあらかじめ読み込みます。キャッシュの方針を設計する際には、次のガイドラインを考慮します。

- 適切なキャッシュ先を選択します。アプリケーションが Web ファームに配置される場合は、同期する必要があるローカル キャッシュは使用せず、トランザクション リソース マネージャー (Microsoft® SQL Server® など) または分散キャッシュがサポートされる製品 (Danga Interactive の

Memcached や Velocity というコード名のマイクロソフトのプロジェクトなど) の使用を検討します (詳細については、この章の最後の「関連情報」を参照してください)。

- メモリ内のキャッシュを使用する場合は、すぐに使用できる形式でデータをキャッシュすることを検討します。たとえば、データベースのデータをそのままキャッシュするのではなく、特定のオブジェクトを使用します。また、メモリ内キャッシュの実装には、Microsoft Velocity を使用することを検討します。
- 揮発性データはキャッシュしないようにし、機密データは暗号化しない限りキャッシュしないようにします。
- キャッシュ内のデータは削除されている可能性があるので、キャッシュ内にデータが存在していることをあてにしないようにします。ソースから項目を再度読み込むなど、キャッシュ エラーを処理するメカニズムを実装します。
- 複数のスレッドからキャッシュにアクセスする場合は、特に注意します。複数のスレッドを使用している場合は、一貫性が保たれるように、キャッシュへのすべてのアクセスがスレッド セーフになるようにします。

キャッシュ方針の設計の詳細については、この章の「キャッシュの設計手順」を参照してください。

通信

通信は、複数のレイヤーやティアにまたがるコンポーネント間のやり取りに関する領域です。選択するメカニズムは、アプリケーションでサポートする必要がある配置シナリオによって決まります。通信メカニズムを設計する際には、次のガイドラインを考慮します。

- 物理境界またはプロセス境界を越える通信ではメッセージ ベースの通信を使用し、プロセス内 (論理境界しか越えない) 通信ではオブジェクト ベースの通信を使用することを検討します。ラウンド トリップを削減し、物理境界とプロセス境界を越える通信のパフォーマンスを向上するには、通信回数は少なく、1 回の通信でやり取りされる情報量は多い、粒度の粗い (chunky な) インターフェイスを設計します。ただし、必要に応じて、粒度の細かい (chatty な) インターフェイスを公開してプロセス内呼び出しで使用できるようにすることと、これらの呼び出しを粒度の粗いファサードでラップして、物理境界やプロセス境界を越えるファサードにアクセスするプロセスで使用できるようにすることを検討します。
- メッセージが特定の順序で受信される必要がなく、相互に依存していない場合は、処理や UI スレッドがブロックされないように、非同期通信を使用することを検討します。

- Microsoft Message Queuing を使用して、システムまたはネットワークの中断やエラー発生時に後で配信できるように、メッセージをキューに登録することを検討します。メッセージ キューでは、トランザクション処理されたメッセージ配信を実行することが可能で、信頼できる 1 回限りの配信がサポートされます。
- インターネット通信には HTTP、イントラネット通信には TCP など、適切な転送プロトコルを選択します。適切なメッセージ交換パターン、接続ベースの通信にするかコネクションレス通信にするか、信頼性の保証レベル (サービス レベル アグリーメントなど)、および認証メカニズムを決定する方法を検討します。
- 暗号化、デジタル証明書、およびチャネル セキュリティ機能を使用して、通信中のメッセージと機密データを確実に保護します。

通信方針の設計に関する詳細については、第 18 章「通信とメッセージ」を参照してください。

構成管理

適切な構成管理メカニズムを設計することは、アプリケーションのセキュリティと柔軟性の両方において重要です。適切に設計しないと、アプリケーションがさまざまな攻撃に対して脆弱になる可能性があり、アプリケーションの管理オーバーヘッドの発生にもつながります。構成管理を設計する際には、次のガイドラインを考慮します。

- 外部から構成できる必要がある設定を慎重に検討します。各構成設定に実際のビジネス ニーズがあることを確認し、その要件を満たすために最小限必要な構成オプションを提供します。外部から構成できる設定を必要以上に提供すると、複雑なシステムになり、不適切な構成のためにセキュリティ侵害や誤動作が発生しやすくなる可能性があります。
- 構成情報を一元的に格納して、起動時にダウンロードまたはユーザーに適用するかどうかを決定します (たとえば、Active Directory グループ ポリシーを使用して、この動作を実現します)。構成情報へのアクセスを制限する方法を検討します。最小特権のプロセスとサービス アカウントの使用を検討します。また、構成ストアに格納する機密情報は暗号化します。
- 構成アイテムの適用先 (ユーザー、アプリケーション設定、または環境設定) に基づいて構成アイテムを論理セクションに分類します。このように分類すると、ユーザー グループごとに、または複数の環境用に異なる設定をサポートする必要がある場合に、構成を分けやすくなります。
- アプリケーションに複数のティアがある場合は、構成アイテムを論理セクションに分類します。サーバー アプリケーションが Web ファームで実行される場合は、共通の構成と、アプリケーションが実行さ

れるコンピューターに固有の構成を決定します。そのうえで、各セクションに適した構成ストアを選択します。

- 構成情報を編集する管理 UI を別個に用意します。

例外管理

適切な例外管理方針を設計することは、アプリケーションのセキュリティと信頼性の両方において重要です。適切に設計しないと、アプリケーションで発生した問題の診断と解決が非常に困難になります。また、アプリケーションがサービス拒否攻撃 (DoS) に対して脆弱になる可能性もあり、機密情報や重要な情報が漏えいする可能性があります。例外の生成と処理は負荷の高いプロセスなので、パフォーマンスの問題も考慮して設計することが重要です。適切な手法は、アプリケーションで使用する一元化された例外管理メカニズムを設計することです。また、Microsoft System Center などのエンタープライズ レベルの監視システムをサポートできるように、例外管理システム内のアクセスポイント (WMI イベントなど) を提供することを検討します。例外管理の方針を設計する際には、次のガイドラインを考慮します。

- 例外をラップまたは置換する適切な例外の伝達方針を設計するか、必要に応じてさらに情報を追加します。たとえば、例外の境界レイヤーへの伝播を許可し、次のレイヤーに渡される前に、必要に応じて境界レイヤーでログ記録や変換が実行されるようにします。エラーとフォールトの根本原因を分析しやすくするため、異なるレイヤーで発生した関連する例外を関連付けられるように、コンテキスト ID を含めることを検討します。また、ハンドルされていない例外にも対応するような設計にする必要があります。ハンドルできないか、情報を追加する必要がない限り、内部例外はキャッチしないようにします。また、例外を使用してアプリケーション フローを制御しないようにします。
- エラーが発生しても、アプリケーションが不安定な状態にならず、例外によってアプリケーションから機密情報やプロセスの詳細が漏えいすることがないようにします。適切に回復することを保証できない場合は、不明 (おそらく不適切) な状態でアプリケーションを実行し続けるのではなく、ハンドルされていない例外が発生した場合はアプリケーションが停止できるようにします。
- 重大なエラーと例外についてのログ記録と通知に関する適切な方針を設計して、例外に関する十分な詳細情報をログに記録し、サポート スタッフがシナリオを再現できるようにします。ただし、例外メッセージやログ ファイルで機密情報が開示されないように留意します。

例外管理方針の設計の詳細については、この章の「例外管理の設計手順」を参照してください。

ログ記録とインストルメンテーション

適切なログ記録とインストルメンテーションの方針を設計することは、アプリケーションのセキュリティと信頼性の両方において重要です。適切に設計しないと、ユーザーが自身の操作を否定する否認の脅威に対してアプリケーションが脆弱になる可能性があり、不正行為を証明するために訴訟においてログ ファイルが必要になる可能性があります。アプリケーションの複数のレイヤーにまたがる動作を監査してログに記録します。これは、疑わしい動作を検出し、深刻な攻撃の初期の兆候を捉えるうえで有効です。通常、監査は、監査情報がリソースへのアクセスと同じタイミングで生成され、リソースにアクセスした同じルーチンで生成された場合に、最も確実だと見なされます。インストルメンテーションは、パフォーマンス カウンターとパフォーマンス イベントを使用して実装することが可能で、アプリケーションの状態、パフォーマンス、および正常性についての情報を管理者に提供します。ログ記録とインストルメンテーションの方針を設計する際には、次のガイドラインを考慮します。

- システム クリティカルなイベントとビジネス クリティカルなイベントをキャプチャする、一元的なログ記録とインストルメンテーションのメカニズムを設計します。ログ記録とインストルメンテーションの粒度が細かくなりすぎないようにします。ただし、追加情報を取得してデバッグに役立つように、実行時に構成できる追加のログ記録とインストルメンテーションを利用することも検討します。
- 安全なログ ファイル管理ポリシーを作成します。ログ ファイルには機密情報を保存しないようにして、ログ ファイルが不当にアクセスされないように保護します。アプリケーション レイヤーをまたいで監査およびログ記録のデータにアクセスする方法とデータを渡す方法を検討します。ログ記録エラーは抑制しながら、適切にハンドルします。
- ログ シンク (トレース リスナー) を構成できるようにして、配置環境の要件に合わせて実行時に変更できるようにすることを検討します。patterns & practices の Enterprise Library などのライブラリは、アプリケーションにログ記録とインストルメンテーションを実装する場合に便利です。その他によく利用される Dependency Injection のライブラリには、NLog や log4net があります (詳細については、この章の最後の「関連情報」を参照してください)。

ログ記録とインストルメンテーションの詳細については、この章の「例外管理の設計手順」を参照してください。

状態管理

状態管理は、プロセスのコンポーネント、操作、またはステップの状態を表すデータの維持に関する領域です。状態データの維持には、さまざまな形式とストアを使用できます。状態管理メカニズムの設計は、アプリケーションのパ

フォーマンスに影響を及ぼすことがあります。わずかな状態情報しか維持しない場合でも、パフォーマンスとアプリケーションのスケーラビリティに悪影響を及ぼす可能性があるので、 必要なデータのみを維持し、状態の管理に利用できるオプションを理解する必要があります。状態管理のメカニズムを設計する際には、次のガイドラインを考慮します。

- 状態管理では、できる限り無駄を省き、状態の維持に必要な最小限のデータを保持します。
- プロセス境界やネットワーク境界をまたいで状態データを維持する必要がある場合は、状態データがシリアル化されるようにします。
- 適切な状態ストアを選択します。プロセスとメモリ内に状態を保存すると、最高のパフォーマンスを得られますが、この技法は、プロセスやシステムを再起動した場合に状態を維持する必要がない場合にしか利用できません。プロセスやシステムの再起動後も、状態が維持されるようにするには、ローカル ディスクまたはローカルの SQL Server で状態を維持します。アプリケーションで状態が非常に重要な場合や、複数のコンピューター間で状態を共有する場合は、SQL Server など集中管理された場所に状態情報を保存することを検討します。

検証

効果的な検証メカニズムを設計することは、アプリケーションのユーザビリティと信頼性の両方において重要です。適切に設計しないと、アプリケーションがデータの不整合やビジネス ルールの違反に対して脆弱になり、ユーザー エクスペリエンスが低下します。また、入力が適切に検証されないと、クロス サイト スクリプト攻撃、SQL インジェクション攻撃、バッファのオーバーフロー、その他の種類の入力による攻撃などのセキュリティの問題に対してアプリケーションが脆弱になる場合があります。残念ながら、有効な入力と悪意のある入力とを区別できる標準的な定義はありません。また、アプリケーションで入力を使用する方法も、脆弱性の悪用に関するリスクに影響します。検証のメカニズムを設計する際には、次のガイドラインを考慮します。

- できる限り、無効な入力の条件を定義するのではなく、受け入れられる入力を具体的に定義した許可一覧を使用するように検証システムを設計します。後で、禁止一覧の範囲を狭めるよりも、許可一覧の範囲を広げる方がはるかに簡単です。
- セキュリティ チェックについては、クライアント側の検証だけに頼らないようにします。クライアント側の検証は、ユーザーにフィードバックを返し、ユーザー エクスペリエンスを向上するために使用します。クライアント側の検証は容易にバイパスできるため、正しくない入力や悪意のある入力のチェックには、必ずサーバー側の検証を使用します。

- 検証手法を再利用できる場合は、別個のコンポーネントにまとめるか、patterns & practices が提供している Enterprise Library の Validation Block など、サードパーティ製のライブラリを使用することを検討します。このようにすると、アプリケーションのレイヤーとティア全体で一貫性のある検証メカニズムを適用できます。
- 必ずユーザーの入力を制限、拒否、および一部削除します。つまり、すべてのユーザーの入力が悪意のある入力だと仮定します。信頼境界を特定し、信頼境界を越えるすべての入力データについて、長さ、形式、型、および範囲を検証します。

検証方針の設計に関する詳細については、この章の「入力とデータ検証の設計手順」を参照してください。

キャッシュの設計手順

キャッシュは、パフォーマンスを最大限に高めるうえで、きわめて重要な役割を果たします。ただし、不適切な技法を適用するとパフォーマンスが低下する可能性があるため、適切なキャッシュ方針を設計することが重要です。次の手順は、アプリケーションに適切なキャッシュ方針を設計するうえで役立ちます。

手順 1 – キャッシュするデータを決定する

アプリケーション設計の一環として、キャッシュに適したデータを決定することが重要です。アプリケーションの各レイヤーで、キャッシュするデータの一覧を作成します。次の種類のデータをキャッシュすることを検討します。

- **アプリケーション全体に関係するデータ**: アプリケーションの全ユーザーに適用される、比較的静的なデータをキャッシュすることを検討します。たとえば、製品一覧や製品情報です。
- **比較的静的なデータ**: 完全に静的であるか、頻繁に変更されないデータをキャッシュすることを検討します。たとえば、定数、構成やデータベースから読み取られる固定値です。
- **比較的静的な Web ページ**: Web ページの出力または頻繁に変更されない Web ページのセクションをキャッシュすることを検討します。
- **ストアド プロシージャのパラメーターとクエリ結果**: 頻繁に使用されるクエリ パラメーターとクエリ結果をキャッシュすることを検討します。

手順 2 – データのキャッシュ先を決定する

データのキャッシュ先を決める場合、通常、キャッシュの物理的な場所と論理的な場所の 2 つについて検討します。

物理的な場所は、メモリ内、またはファイルかデータベースを使用したディスク上の場所になります。メモリ内キャッシュは、ASP.NET のキャッシュ メカニズム、Enterprise Library の Caching Application Block、または分散メモリ内キャッシュ メカニズム (Velocity というコード名のマイクロソフトのプロジェクトや Danga Interactive の Memcached テクノロジ) を使用して実行できます。メモリ内キャッシュは、データがアプリケーションで頻繁に使用される場合、キャッシュされるデータの揮発性が比較的強く、頻繁に再取得する必要がある場合、およびキャッシュされるデータの量が比較的少ない場合に有効な選択肢です。ファイル システム ベースのキャッシュやデータベース キャッシュは、元のストアからデータを取得するよりも、キャッシュ ストアのデータにアクセスした方が効率的な場合、キャッシュされるデータの揮発性が比較的弱い場合、およびデータを再取得するサービスが常時利用できるとは限らない場合に有効な選択肢です。ディスク ベースの手法は、キャッシュするデータの量が比較的多い場合や、プロセスやコンピューターの再起動後もキャッシュされたデータが維持される必要がある場合にも適しています。

キャッシュの論理的な場所は、アプリケーション ロジック内の場所を表します。処理とネットワーク ラウンド トリップを最小限に抑え、アプリケーションのパフォーマンスと応答性を最大限に高めるには、データが使用される場所からできる限り近い場所にデータをキャッシュすることが重要です。キャッシュ データの論理的な場所を決める際には、次のガイドラインを考慮します。

- データがページやユーザー固有のもので、機密情報を含まず、軽量である場合は、クライアント側にキャッシュすることを検討します。
- クライアントによって頻繁に要求される比較的静的なページがある場合や、一定の頻度でページが更新される場合、または結果が Web サービスから返される場合は、プロキシ サーバーまたは (Web アプリケーションの) Web サーバーにキャッシュすることを検討します。また、頻繁に変更されない HTTP パラメーターに基づいて異なる出力を生成できるページがある場合にも、この手法の使用を検討します。出力の範囲が狭い場合に、これは特に有効です。
- 比較的静的なページ出力がある場合、少数のユーザーのユーザー設定に関する少量のデータがある場合、または作成に高い負荷がかかる UI コントロールがある場合は、プレゼンテーション レイヤーにデータをキャッシュすることを検討します。また、製品一覧や製品情報など、ユーザーに対して表示する必要があって、作成に高い負荷がかかるデータがある場合も、この手法の使用を検討します。
- サービス、ビジネス プロセス、またはワークフローの状態を維持する必要がある場合、または比較的静的なデータがプレゼンテーション レイヤーからの要求の処理に必要で、このデータの作成に高い負荷がかかる場合は、ビジネス レイヤーにデータをキャッシュすることを検討します。
- コレクション内で頻繁に呼び出されるストアド プロシージャに入力パラメーターがある場合、または頻繁に実行されるクエリから返される少量の生データがある場合は、データ レイヤーにデータをキャ

ッシュすることを検討します。また、型指定されたデータセットのスキーマも、データ レイヤーにキャッシュすることを検討します。

- 結果セットを取得するために膨大なクエリ処理を必要とするデータについては、データベース内の別のテーブルにキャッシュすることを検討します。これは、パフォーマンスを向上するために、ページング メカニズムを実装して表示するデータのセクションを読み取るという、キャッシュするデータの量が膨大な場合にも適しています。

手順 3 – キャッシュするデータの形式を決定する

キャッシュする必要があるデータを決定し、データをキャッシュする場所を決めたら、次はキャッシュされたデータの形式を決める必要があります。データをキャッシュするときには、目的の用途に最適な形式で保存し、追加または繰り返しの処理や変換が不要になるようにします。このようなことを考慮してキャッシュされたデータは、メモリ内キャッシュを使用してデータをキャッシュしなければならない場合、複数のプロセスやコンピューターでキャッシュを共有する必要がない場合、キャッシュされたデータを複数のメモリの場所の間で転送する必要がない場合、および DataSet、DataTable、Web ページなど生データをキャッシュする必要がない場合に有効です。

キャッシュされたデータを保存または転送する必要がある場合は、シリアル化の要件を検討します。キャッシュされたデータのシリアル化は、ディスク ベースのキャッシュにデータをキャッシュするか、別個のサーバーまたは SQL Server データベースにセッション状態を保存する場合に有効です。この手法は、複数のプロセスやコンピューターでキャッシュを共有する場合、複数のメモリの場所の間でキャッシュを転送する場合、またはカスタム オブジェクトをキャッシュする場合にも有効です。データのシリアル化には、XML シリアライザーまたはバイナリ シリアライザーを使用できます。XML シリアライザーは、相互運用性が重要な関心事である場合に、有効な選択肢です。パフォーマンスが重要な関心事である場合は、バイナリ シリアライザーの使用を検討します。

手順 4 – 適切なキャッシュ管理方針を決定する

適切なキャッシュの有効期限とキャッシュのフラッシュ ポリシーを決定する必要があります。有効期限とフラッシュは、キャッシュ ストアからキャッシュされたデータを削除することに関係があります。フラッシュでは、使用頻度の高い項目をキャッシュする領域を空けるために、有効なキャッシュ項目を削除する場合があります。一方、有効期限では、無効で有効期限が切れた項目を削除します。ただし、すべてのキャッシュ実装で、これらのオプションがすべて利用できるわけではないため、基盤のキャッシュ システムの機能を確認する必要があります。

キャッシュ内のデータと項目の妥当性を維持するため、キャッシュの有効期限に関する方針を設計します。キャッシュの有効期限ポリシーを決める際には、次の情報を参考に時間ベースの有効期限と通知ベースの有効期限の両方について検討します。

- 時間ベースの有効期限ポリシーでは、キャッシュされたデータは、相対的または絶対的な間隔に基づいて有効期限切れまたは無効になります。これは、キャッシュ データが揮発性データである場合、キャッシュされたデータが定期的に更新される場合、またはキャッシュされたデータが特定の時間や間隔においてのみ有効である場合に、有効な選択肢です。時間ベースの有効期限ポリシーを使用する場合は、絶対時間での有効期限ポリシーか相対時間での有効期限ポリシーを選択できます。絶対時間での有効期限ポリシーでは、キャッシュされたデータの有効期限が切れる時刻を指定して、キャッシュされたデータのライフタイムを定義できます。一方、相対時間での有効期限ポリシーでは、キャッシュされたデータが最後にアクセスされた時間から有効期限が切れる時間までの間隔を指定して、キャッシュされたデータのライフタイムを定義できます。
- 通知ベースの有効期限ポリシーでは、キャッシュされたデータが内部または外部ソースからの通知に基づいて有効期限切れまたは無効になります。これは、非揮発性のキャッシュ データを扱っている場合、キャッシュされたデータが一定の間隔で更新されない場合、またはデータが外部または内部のシステムで変更されない限り有効な場合に、有効な選択肢です。一般的な通知ソースは、ディスク ファイルの書き込み、WMI イベント、SQL Server の依存関係の通知、およびビジネス ロジックの操作です。通知により、依存関係のあるキャッシュ項目が有効期限切れまたは無効にされます。

キャッシュのフラッシュ方針は、記憶域、メモリなどのリソースが効率よく使用されるように設計します。キャッシュのフラッシュ方針を決める際には、次の情報を参考に明示的なフラッシュまたは清掃を選択します。

- **明示的なフラッシュ**: 項目がフラッシュして削除するタイミングを決める必要があります。これは、破損したか古いキャッシュ データを削除するシナリオをサポートする必要がある場合、清掃をサポートしないカスタム ストアを使用している場合、またはディスク ベースのキャッシュを使用している場合に有効な選択肢です。
- **清掃**: 項目を清掃する条件とヒューリスティックを決める必要があります。これは、システム リソースが不足してきたときに自動的に清掃を有効にする場合、ほとんど使用されないか重要でない項目をキャッシュから自動的に削除する場合、またはメモリ ベースのキャッシュを使用している場合に、有効な選択肢です。

一般的に使用される清掃のヒューリスティックは、次のとおりです。

- **最近、最も使われていないもの**: このアルゴリズムでは、使用されていない期間が最も長い項目が清掃されます。
- **最も使われていないもの**: このアルゴリズムでは、読み込まれてからの使用頻度が最も低い項目が清掃されます。
- **優先度**: このアルゴリズムでは、キャッシュされた項目に優先順位を付けて、キャッシュの清掃時に最も優先順位が高い項目を維持するようにキャッシュに指示します。

手順 5 – キャッシュ データを読み込む方法を決定する

適切なキャッシュの読み込み方法を選択することは、アプリケーションのパフォーマンスと応答性を最大限に高めるうえで役立ちます。キャッシュにデータを設定する方法を決める際には、アプリケーションの起動時やキャッシュの初期読み込み時に利用する必要があるデータの量と、アプリケーションの起動時間とパフォーマンスへの影響を考慮します。たとえば、アプリケーションの起動時にはにキャッシュにデータを事前に読み込むことも、必要になった場合にのみデータを取得してキャッシュすることもできます。アプリケーションの起動時にデータをキャッシュに読み込むと、アプリケーションの応答性は向上しますが、起動時間が長くなります。一方、データの初回アクセス時にデータをキャッシュに読み込む場合、アプリケーションの起動時間は短くなりますが、初期の応答性は低下します。キャッシュへのデータ設定に関する方針を設計する際には、次の情報を参考にプロアクティブ読み込みまたはリアクティブ読み込みを使用します。

- アプリケーションの起動時に必要なデータをすべて取得し、アプリケーションの実行中にデータをキャッシュする場合は、プロアクティブ読み込みを選択します。プロアクティブ読み込みは、キャッシュされたデータが比較的静的であるか、更新頻度、ライフタイム、およびサイズが把握できている場合に有効な選択肢です。データのサイズがわからない場合、データをすべて読み込むとシステム リソースが枯渇する可能性があります。また、キャッシュ データのソースが低速なデータベースであるか、データが低速なネットワーク経由または信頼性の低い Web サービスから取得される場合にも有効です。
- アプリケーションから要求されたときにデータを取得し、それ以降の要求で使用できるようにデータをキャッシュする場合は、リアクティブ読み込みを選択します。リアクティブ読み込みは、キャッシュ データの揮発性が比較的高い場合、キャッシュ データのライフタイムが不明な場合、キャッシュ データの量が多い場合、およびキャッシュ データのソースの信頼性と応答性が高い場合に有効な選択肢です。

例外管理の設計手順

例外管理の方針が堅牢で適切に設計されていると、アプリケーションの設計を簡略化して、セキュリティと管理性を向上できます。また、アプリケーションが開発しやすくなり、開発にかかる時間とコストを削減できます。次の手順は、アプリケーションに適切な例外管理方針を設計するうえで役立ちます。

手順 1 – ハンドルする例外を特定する

アプリケーションの例外管理を設計する際には、ハンドルする例外を特定することが重要です。アクセス許可がないシステム リソースにアクセスしたユーザーによって生成された例外、ディスク、CPU、またはメモリの問題に起因するシステム エラーなど、システム例外やアプリケーション例外をハンドルする必要があります。また、ハンドルするビジネス例外も特定する必要があります。たとえば、ビジネス ルール違反などのアクションによって発生した例外が、これに該当します。

手順 2 – 例外の検出方針を決定する

アプリケーション全体で一貫性して構造化例外処理が使用されるように、設計で規定する必要があります。これにより、一貫性がない状態になる可能性が低い、より堅牢なアプリケーションを作成できます。構造化例外処理では、コード内で発生したエラーを検出し、エラーに適宜対応するために、try、catch、および finally ブロックを使用して例外を管理する手段を提供します。

例外の検出について検討する際には、ログに記録する例外の詳細情報を収集する場合、例外に付加情報を追加する場合、コード ブロック内で使用されたリソースをクリーンアップする場合、または例外から回復するための操作を再試行する場合にのみ、例外をキャッチすることが重要です。これらの作業を実行する必要がない場合は、例外をキャッチせずにコール スタック内を伝播できるようにします。

手順 3 – 例外の伝播方針を決定する

以下の例外の伝播方針について検討します。アプリケーションは、各コンテキストの要件に応じて、これらの方針のいずれかまたはすべてを組み合わせて使用できます (使用する必要があります)。

- **例外の伝播を許可する**: この方針は、ログに記録する例外の詳細情報の収集、例外への関連情報の追加、使用されたリソースのコード ブロック内でのクリーンアップ、または例外からの回復操作の再試行が必要ない場合に有効です。例外がコード スタック内を伝播できるようにします。

- **例外をキャッチして再スローする**: この方針では、例外をキャッチし、その他の処理を実行して、例外を再度スローします。通常、この手法では、例外情報はそのまま維持されます。この方針は、リソースをクリーンアップする必要がある場合、例外情報をログに記録する必要がある場合、またはエラーから回復を試みる必要がある場合に有効です。
- **例外をキャッチし、ラップし、スローする**: この方針では、汎用例外をキャッチし、リソースをクリーンアップするか他の関連処理を実行して、例外に対応します。エラーから回復できない場合は、例外を呼び出し元との関連性がより高い別の例外でラップし、この新しい例外をスローして、コード スタック内の上位のコードで処理できるようにします。この方針は、例外の関連性を維持する場合や例外をハンドルするコードに追加情報を提供する場合、またはその両方を行う場合に有効です。
- **例外をキャッチして破棄する**: これは推奨される方針ではありませんが、特定のシナリオでは適している場合があります。例外をキャッチして、通常どおりアプリケーションの実行を続けます。必要に応じて、例外をログに記録して、リソースのクリーンアップを実行できます。この方針は、ログがいっぱいになったときに生成される例外など、ユーザーの操作に影響しないシステム例外に有効な場合があります。

手順 4 – カスタムの例外方針を決定する

カスタム例外を設計する必要があるのか、標準の .NET Framework の例外の種類だけで対応できるのかを検討します。例外階層または .NET Framework に適切な例外がある場合は、カスタム例外は使用しません。ただし、条件ロジックを使用しないように、アプリケーションが特定の例外を検出して処理する必要がある場合、または特定の要件に合わせて追加情報を含める必要がある場合は、カスタム例外を使用します。

カスタム例外クラスを作成する必要がある場合は、クラス名の最後に必ず Exception という単語を付けます。また、シリアル化コンストラクターも含め、カスタム例外クラス用の標準のコンストラクターを実装します。これは、標準の例外メカニズムと統合するうえで重要です。カスタム例外を実装するには、適切でより汎用的な例外から派生させて、要件に合うように例外を特殊化します。

一般に、例外管理方針を設計する場合は、例外階層を作成して、その中でカスタム例外を整理します。これは、問題をすばやく分析したり追跡するのに役立ちます。カスタム例外では、例外が発生したレイヤー、例外が発生した可能性があるコンポーネント、および発生した例外の種類 (セキュリティ、システム、またはビジネス例外) についての情報を提供する必要があります。

アプリケーションの例外階層は、アプリケーション コードのどこからでも参照できる単一のアセンブリにまとめて保存することを検討します。このようにすると、一元的な例外クラスの管理と配置が可能になります。また、境界を

またいで例外をマーシャリングする方法についても検討します。.NET Framework の Exception クラスでは、シリアル化をサポートしています。カスタムの例外クラスを設計する際は、シリアル化もサポートするようにします。

手順 5 – 収集する適切な情報を決定する

例外のハンドルする際に最も重要になる要素の 1 つは、例外情報を収集するための確かな方針です。キャプチャされる情報は、例外の状態を正確に表している必要があります。また、例外情報を見るユーザーにとって適切で有用な情報である必要もあります。通常、ユーザーは、エンド ユーザー、アプリケーション開発者、およびオペレーターの 3 つのカテゴリのいずれかに分類されます。シナリオと個々のコンテキストを調べて、対象ユーザーを分析します。

エンド ユーザーには、意味があり、適切に表現された説明が必要です。エンド ユーザーに提示する例外情報を収集する場合は、エラーの性質を示すわかりやすいメッセージと、必要に応じてエラーから回復する方法についての情報を提供することを検討します。アプリケーション開発者には、問題の診断に役立つ、より詳細な情報を提供する必要があります。

アプリケーション開発者に表示する例外情報を収集する場合は、例外が発生した正確な位置と、例外の種類や例外が発生したときのシステムの状態など、例外についての詳細を必ず提供します。オペレーターには、適切に対応し、必要な回復手順を実行できるようにする関連情報を提供する必要があります。オペレーターに表示する例外情報を収集する場合は、通知が必要なユーザーをオペレーターが特定できるようにする例外の詳細とナレッジ、および問題の解決に必要な情報を提供することを検討します。

例外情報を受け取るユーザーのカテゴリに関係なく、詳細な例外情報を提供することは有益です。情報はログ ファイルに保存して、後で検証できるように、また例外の発生頻度と詳細を分析できるようにします。既定では、少なくとも、発生日時、コンピューター名、例外のソースと種類、例外メッセージ、スタックとコール トレース、アプリケーション ドメイン名、アセンブリの名前とバージョン、スレッド ID、およびユーザーの詳細をキャプチャします。

手順 6 – 例外のログ記録の方針を決定する

例外情報のログ記録には、さまざまなオプションを利用できます。次の重要な検討事項は、ログ記録のオプションを選択するうえで役立ちます。

- アプリケーションを 1 台のコンピューターに配置する場合、既存のツールを利用してログを参照する必要がある場合、または信頼性が最大の関心事である場合は、Windows イベント ログまたは Windows Eventing 6.0 を選択します。

- アプリケーションをファームまたはクラスターに配置する場合、一元的にログを記録する必要がある場合、または例外情報の構造化とログ記録に柔軟性が必要な場合は、SQL Server データベースを選択します。
- アプリケーションを 1 台のコンピューターに配置する場合、ログの形式を自由自在に選択できる必要がある場合、またはシンプルで簡単にログ ストアを実装する必要がある場合は、カスタム ログ ファイルを選択します。ログは定期的にトリミングまたは統合して、ログ ファイルのサイズを制限し、ファイルが大きくなり過ぎないようにします。
- 信頼性が最大の関心事である場合、アプリケーションをファームまたはクラスターに配置する場合、または一元的にログを記録する必要がある場合は、例外メッセージを最終的な宛先に渡す配信メカニズムとしてメッセージ キューを使用します。

どのようなアプリケーションでも、実際のシナリオと例外ポリシーに基づいて、これらのオプションを組み合わせて使用できます。たとえば、セキュリティ例外はセキュリティ イベント ログに記録し、ビジネス例外はデータベースに記録できます。

手順 7 – 例外の通知方針を決定する

例外管理の設計の一環として、通知方針を決定する必要もあります。エンタープライズ アプリケーションの場合、通常は例外管理とログ記録だけでは十分ではありません。管理者とオペレーターが例外を確実に把握できるように、補完機能として通知を利用することを検討します。通知には、WMI イベント、SMTP 電子メール、SMS テキスト メッセージ、その他のカスタムの通知システムなどのテクノロジを使用できます。
ログ監視システムや、ログ データのエラー状態を検出し、適切な通知を生成するサードパーティの環境など、外部の通知メカニズムを使用することを検討します。これは、アプリケーション コードから監視と通知システムを切り離して、アプリケーションにログ記録用のコードのみを実装する場合に有効な選択肢です。または、外部の監視システムを利用せずに直ちに通知を生成する場合は、アプリケーションにカスタム通知メカニズムを追加することを検討します。

手順 8 – ハンドルされていない例外の処理方法を決定する

例外が最後のポイントや境界までハンドルされず、制御がユーザーに返されるまで例外から回復する手段がない場合、このハンドルされていない例外はアプリケーションで処理する必要があります。ハンドルされていない例外については、必要な情報を収集して、ログ ファイルまたは監査ファイルに書き込み、例外に必要な通知を送信し、必要なクリーンアップを実行して、最終的にエラー情報をユーザーに伝達します。

例外の詳細情報をすべて公開することは避け、 ユーザーにはわかりやすい汎用的なエラー メッセージを提供します。Web サービスなど、ユーザー インターフェイスがないクライアントの場合は、詳細な例外の代わりに汎用例外をスローすることをお勧めします。汎用例外をスローすることで、エンド ユーザーにシステムの詳細情報が公開されることを回避できます。

patterns & practices の Exception Handling Application Block と patterns & practices の Logging Application Block を使用して、一貫性のある例外管理、ログ記録、および通知の方針をアプリケーションに実装することを検討します。Exception Handling Application Block では、さまざまな例外ハンドル オプションをサポートしています。また、Logging Application Block では、ログ メッセージを受信して形式を整え、さまざまなログやその他の宛先に、ログメッセージと通知を送信できます。詳細については、付録 F「patterns & practices の Enterprise Library」を参照してください。

入力とデータ検証の設計手順

次の手順は、アプリケーションに適切な検証方針を設計するうえで役立ちます。アプリケーションの入力とデータ検証を設計する際には、まず、データを検証する必要がある主要なシナリオと信頼境界を特定する必要があります。次に、検証するデータとデータを検証する場所を特定します。また、再利用可能な検証方針を実装する方法も決定する必要があります。最後に、アプリケーションに適した検証方針を決定します。

手順 1 – 信頼境界を特定する

信頼境界は、信頼できるデータと信頼できないデータの区分けを定義します。信頼境界の一方の側にあるデータは信頼できますが、もう一方の側にあるデータは信頼できません。まず、信頼境界を越えるデータを特定し、検証が必要なものを決定します。クロスサイト スクリプトやコード インジェクションなど、セキュリティの脅威の対策として、すべての信頼境界で入力検証とデータ検証を実施することを検討します。信頼境界の例には、境界ファイアウォール、Web サーバーとデータベース サーバーの境界、アプリケーションとサードパーティ製のサービス間の境界などがあります。

アプリケーションが通信するシステムとサブシステム、およびサーバー上にあるファイルへの書き込み、データベース サーバーへの呼び出し、または Web サービスの呼び出し時に境界を越える外部システムを特定します。クライアント入力からのデータと、共有データベースなどの信頼されないソースからのデータを書き込む信頼境界のエントリ ポイントと終了ポイントを特定します。

手順 2 – 主要なシナリオを特定する

アプリケーション内の信頼境界を特定したら、データの検証が必要になる主要なシナリオを定義する必要があります。ユーザーが入力したデータはすべて、検証されるまでは悪意があるものと見なす必要があります。たとえば、Web アプリケーションの場合、プレゼンテーション レイヤーのデータで検証が必要なものには、フォーム フィールド、クエリ文字列、および非表示フィールドの値、GET および POST 要求で送信されるパラメーター、アップロードされるデータ (悪意のあるユーザーが HTTP 要求を傍受して内容を変更する可能性があるため)、Cookie (クライアント コンピューター上にあり、変更される可能性があるため) があります。

ビジネス レイヤーでは、ビジネス ルールを使用してデータに制約を適用します。このようなルールに違反した場合は検証エラーと見なし、ビジネス レイヤーから、違反があったことを示すエラーを生成します。ルール エンジンやワークフローを使用する場合は、各ルールに必要な情報と先行ルールの検証結果に基づいて、各ルールの結果を検証します。

手順 3 – 検証する場所を決定する

この手順では、検証を実行する場所、つまりクライアントのみで検証を実行するか、サーバーとクライアントの両方で検証を実行するかを決定します。ただし、クライアント側の検証だけに頼らないようにする必要があります。クライアント側の検証を使用してより対話的な UI を提供しますが、信頼境界内で安全にデータを検証できるように、必ずサーバー側の検証も実装します。クライアント側でデータとビジネス ルールを検証することで、サーバーへのラウンド トリップ数が削減され、ユーザー エクスペリエンスが向上します。Web アプリケーションの場合は、クライアント ブラウザーで DHTML と JavaScript がサポートされるようにします。理想的には別個の .js ファイルを実装して、再利用できるようにしたり、ブラウザーがこのファイルをキャッシュできるようにします。Web アプリケーションの場合、最も簡単な手法は、ASP.NET の検証コントロールを使用する方法です。これは、クライアント側でデータを検証することが可能で、サーバー側でも自動的に検証を行う一連のサーバー コントロールです。

Web アプリケーションの場合、サーバー側のデータとビジネス ルールの検証は、ASP.NET の検証コントロールを使用して実装できます。また、Web アプリケーションでもその他の種類のアプリケーションでも、patterns & practices の Validation Application Block を使用して、複数のレイヤーをまたいで再利用できる検証ロジックを作成することを検討します。Validation Application Block は、Windows フォーム、ASP.NET、および WPF アプリケーションで使用できます。Validation Application Block の詳細については、付録 F「patterns & practices の Enterprise Library」を参照してください。

手順 4 – 検証方針を特定する

一般的なデータ検証方針は次のとおりです。

- **既知の適切な値の受け入れ** (許可リスト、肯定的検証): 一致条件を満たすデータのみを受け入れ、他のすべての値を拒否します。これは最も安全な手法なので、可能な限りこの方針を使用します。
- **既知の不適切な値の拒否** (禁止リスト、否定的検証): 特定の条件 (特定の文字が含まれていないなど) に合わないデータを許可します。既知の無効な入力を網羅する完全な条件の一覧を作成することは非常に難しいため、この方針は補佐的な対策として、注意して使用します。
- **一部削除**: 入力を安全なデータにするため、文字を削除または変換します。既知の不適切な値の拒否 (否定的検証) の手法と同様に、既知の無効な入力を網羅する完全な条件の一覧を作成することは非常に難しいため、この方針は補佐的な対策として、注意して使用します。

関連する設計パターン

横断的関心事に関連する主要なパターンは、次の表に示すカテゴリに分類できます。各カテゴリの設計を行う際には、これらのパターンの使用を検討します。

カテゴリ	関連するパターン
キャッシュ	**Cache Dependency**: 外部からの情報を使用して、キャッシュに格納されているデータの状態を判断します。 **Page Cache:** 頻繁にアクセスされ、変更の頻度が低く、構築に大量のシステム リソースを消費する動的な Web ページの応答時間が向上します。
通信	**Intercepting Filter**: Web ページの要求中に、一般的なプリプロセスとポストプロセス タスクを実装する、一連の組み合わせ可能なフィルター (個別のモジュール) です。 **Pipes and Filters**: メッセージがパイプを通過する際に、メッセージを変更または検証できるパイプとフィルター経由でメッセージをルーティングします。 **Service Interface**: 他のシステムがサービスとの通信に使用できる、プログラマティック インターフェイスです。

Page Cache、Intercepting Filter、および Service Interface パターンの詳細については、「Microsoft .NET を使用したエンタープライズ ソリューション パターン」(http://msdn.microsoft.com/ja-jp/library/ms998469.aspx) を参照してください。
Pipes and Filters パターンの詳細については、「Integration Patterns」(http://msdn.microsoft.com/en-us/library/ms978729.aspx、英語) を参照してください。

patterns & practices のソリューションの資産

マイクロソフトの patterns & practices グループが提供している関連サービスの詳細については、次のリソースを参照してください。

- **Enterprise Library**: キャッシュ、例外管理、検証、ログ記録、暗号化、資格情報管理など、一般的なタスクを簡略化し、Inversion of Control や Dependency Injection などの設計パターンの実装を支援する一連のアプリケーション ブロックを提供します。詳細については、「Microsoft Enterprise Library」(http://msdn2.microsoft.com/en-us/library/cc467894.aspx、英語) を参照してください。
- **Unity Application Block**: 疎結合アプリケーションの構築に役立つ、軽量で拡張可能な依存性注入（DI）コンテナーです。詳細については、「Unity Application Block」(http://msdn.microsoft.com/en-us/library/cc468366.aspx、英語) を参照してください。

関連情報

Web リソースに簡単にアクセスするには、http://www.microsoft.com/architectureguide (英語) でオンライン版の参考文献を参照してください。

認証と承認の詳細については、以下のリソースを参照してください。

- Authorization (http://msdn.microsoft.com/en-us/library/cc949059.aspx、英語)
- WCF ベースのサービスでの承認 (http://msdn.microsoft.com/ja-jp/magazine/cc948343.aspx)

- Designing Application-Managed Authorization (http://msdn.microsoft.com/en-us/library/ms954586.aspx、英語)
- Enterprise Authorization Strategy (http://msdn.microsoft.com/en-us/library/bb417064.aspx、英語)
- Choosing the Right Presentation Layer Architecture (http://msdn.microsoft.com/en-us/library/aa480039.aspx、英語)
- Patterns & Practices のガイダンス: セキュリティ (http://msdn.microsoft.com/ja-jp/library/ms954624.aspx)
- Trusted Subsystem Design (http://msdn.microsoft.com/en-us/library/aa905320.aspx、英語)

この章で説明した他のトピックの詳細については、以下のリソースを参照してください。

- Caching Architecture Guide for .NET Framework Applications (http://msdn.microsoft.com/en-us/library/ms978498.aspx、英語)
- 実践的なパターン: 凝集度と結合度 (http://msdn.microsoft.com/ja-jp/magazine/cc947917.aspx)
- Joe Duffy 著『Concurrent Programming on Windows』(Addison-Wesley、2009 年)
- Microsoft .NET を使用したエンタープライズ ソリューション パターン (http://msdn.microsoft.com/ja-jp/library/ms998469.aspx)
- Exception Management Architecture Guide (http://msdn.microsoft.com/en-us/library/ee817665.aspx、英語)
- Integration Patterns (http://msdn.microsoft.com/en-us/library/ms978729.aspx、英語)
- マイクロソフト プロジェクト コード名 Velocity (http://msdn.microsoft.com/en-us/data/cc655792.aspx、英語)

横断的関心事の管理に役立つ、よく使用されるサードパーティ製のライブラリとフレームワークのいくつかについては、以下のリソースを参照してください。

- Castle Project (http://www.castleproject.org/index.html、英語)
- Ninject (http://ninject.org/、英語)
- PostSharp (http://www.postsharp.org/、英語)

- StructureMap (http://japanese.osstrans.net/software/structuremap.html)
- memcached (http://www.danga.com/memcached/、英語)
- NLog (http://www.nlog-project.org/、英語)
- log4net (http://logging.apache.org/log4net/、英語)

18

通信とメッセージ

概要

アプリケーションの各部分の通信インフラストラクチャを設計する方法は、アプリケーション (特に分散アプリケーション) の設計に影響する主な要素の 1 つです。たとえば、コンポーネントは相互に通信し、ユーザーによる入力をビジネス レイヤーに送信してから、データ レイヤーを通じてデータ ストアを更新する必要があります。コンポーネントが同じ物理ティアに配置されると、多くの場合、コンポーネント間の直接的な通信をあてにできます。ただし、多くのシナリオで行われるように、コンポーネントとレイヤーを物理的に別のサーバーとクライアント コンピューターに配置する場合、このようなレイヤーのコンポーネント間で、効率的かつ正確に通信する方法を検討する必要があります。

一般的には、直接的な通信 (コンポーネント間でメソッドを呼び出すなど) かメッセージ ベースの通信のどちらかを使用します。メッセージ ベースの通信では、コンポーネントを分離できるなど、多くのメリットがあります。コンポーネントを分離すると、保守容易性が向上するだけでなく、柔軟性も提供されるので、後から展開の戦略を簡単に変更できるようになります。ただし、メッセージ ベースの通信では、パフォーマンス、信頼性、セキュリティなど、考慮しなければならない問題もあります。

この章では、適切な通信手法を選択し、選択した手法で最高の結果を得るための方法を理解して、発生する可能性があるセキュリティと信頼性に関する問題の予測に役立つ設計ガイドラインを示します。ただし、この章の大部分では、適切なメッセージ ベースの通信メカニズムの設計と、非同期通信および同期通信、データ形式、パフォーマンス、セキュリティ、相互運用性、および実装テクノロジの選択に関するガイドラインに重点を置いて説明します。

一般的な設計ガイドライン

アプリケーションの通信方針を設計する際には、レイヤー間およびティア間の通信によるパフォーマンスへの影響を考慮します。論理上または物理上の境界を越える通信によって処理のオーバーヘッドが増加するので、ラウンド トリップの回数を減らし、ネットワーク経由で送信されるデータの量を最小限に抑える効率的な通信を設計します。通信方針を決定する際には、次のガイドラインを考慮します。

- **境界を越える場合は通信方針を考慮する**: 各境界について理解し、これらの境界が通信のパフォーマンスにどのように影響するかを理解します。たとえば、コンピューターのプロセス、コンピューター、マネージ コードからアンマネージ コードの呼び出しはすべて、アプリケーションのコンポーネントや、外部サービスおよび外部アプリケーションと通信する際に越える可能性がある境界です。
- **プロセス境界を越える場合はメッセージ ベースの通信の使用を検討する**: Windows Communication Foundation (WCF) を TCP または名前付きパイプ プロトコルと使用して、パフォーマンスを最大限に高めます。
- **物理的な境界を越える場合はメッセージ ベースの通信の使用を検討する**: 物理的な境界を越えてリモート コンピューターと通信する場合は、WCF の使用を検討します。信頼できる 1 回限りのメッセージの配信には、Microsoft Message Queuing の使用を検討します。
- **リモート レイヤーにアクセスする場合はパフォーマンスと応答性を最大限に高める**: リモート レイヤーと通信する場合は、粒度の粗いメッセージ ベースの通信方法を使用することで通信要件を減らし、できる限り非同期通信を使用して UI のブロックやフリーズを回避します。
- **境界を越えるデータ形式のシリアル化を考慮する**: 他のシステムとの相互運用性を確保する必要がある場合は、XML シリアル化の使用を検討します。XML シリアル化では、オーバーヘッドが増加する点に留意する必要があります。パフォーマンスが重要な場合は、バイナリ シリアル化の使用を検討します (バイナリ シリアル化は XML シリアル化よりも高速で、シリアル化されたデータのサイズが XML よりも小さくなります)。
- **通信中にメッセージや機密データを保護する**: 暗号化、デジタル証明書、およびチャネル セキュリティ機能を使用することを検討します。
- **べき等性と交換性を強化するメカニズムを実装する**: アプリケーション コードで、複数回到着するメッセージ (べき等性) と、バラバラの順序で到着する複数のメッセージ (交換性) を検出して管理できるようにします。

メッセージ ベースの通信のガイドライン

メッセージ ベースの通信では、クライアントが XML ベースのメッセージをトランスポート チャネル経由で渡すことによって呼び出されるサービス インターフェイスを定義することにより、呼び出し元にサービスを公開できます。通常、メッセージ ベースの呼び出しは、リモート クライアントから実行されますが、メッセージ ベースのサービス インターフェイスでは、ローカルの呼び出し元も同じようにサポートできます。メッセージ ベースの通信は、次のシナリオに適しています。

- 中長期的な投資が必要なビジネス システムを実装する場合 (たとえば、長期にわたってパートナーに公開され、パートナーが使用するサービスを構築する場合)。
- 高可用性を提供する必要があるか、信頼されないネットワーク経由で操作する必要がある大規模なシステムを実装する場合。この場合は、メッセージのストア アンド フォワードのメカニズムによって信頼性を向上できます。
- 構築しているサービスで使用する他のサービスから当該サービスを分離したり、当該サービスを使用する他のサービスから当該サービスを分離する必要がある場合。インターフェイスの詳細をクライアントに通知するメッセージ ベースのサービス インターフェイスを使用すると、個々のクライアントで具体的な実装を必要とすることなく、すべてのクライアントがサービスを簡単に使用できます。
- 非同期モデルを使用する実際のビジネス プロセスを使用する場合。

メッセージ ベースの通信を使用する際には、次のガイドラインを考慮します。

- 接続は必ず使用できるとは限らず、メッセージを格納してから、接続が使用できるようになったときに送信しなければならない場合があるので注意が必要です。
- メッセージに応答がない場合の対処方法を考慮します。通信の状態を管理するには、応答メッセージがない場合に後で処理できるように、ビジネス ロジックで送信メッセージをログに記録できます。
- 受信確認を使用して、強制的にメッセージが正しい順序になるように考慮します。
- インターネット通信には HTTP、イントラネット通信には TCP など、標準的なプロトコルを使用します。ニーズを満たすエンドポイント、プロトコル、および形式の既定の組み合わせが存在しない限り、カスタムの通信チャネルは実装しないようにします。
- 通信でメッセージ応答のタイミングが重要な場合は、応答メッセージが到着までクライアントが待機する非同期のプログラミング モデルの使用を検討します。また、クライアントが応答を待機しながら処理を継続して実行できる場合は、非同期モデルの使用を検討します。

メッセージ ベースの通信方針を設計する際には、安定性、再利用性、パフォーマンス、および設計の全体的な完成度に影響するトピックについても考慮する必要があります。次のセクションでは、これらの問題について詳しく説明します。

- 非同期通信と同期通信
- 結合度と凝集性
- データ形式
- 相互運用性
- パフォーマンス
- 状態管理

非同期通信と同期通信

同期通信と非同期通信を使用した場合の主要なトレードオフについて考慮します。同期通信は、呼び出しを受け取る順序を保証する必要があるシナリオや、呼び出しが返されるまで待機しなければならないシナリオに最適です。一方、非同期通信は、応答性が重要なシナリオや、呼び出し先が使用できるかどうかを保証できないシナリオに最適です。同期通信と非同期通信のどちらを使用するかを決める際には、次のガイドラインを考慮します。

- パフォーマンスを最大限に高め、疎結合を実現し、システムのオーバーヘッドを最小限に抑えるには、非同期通信モデルの使用を検討します。一部のクライアントが同期呼び出ししか実行できない場合は、既存の非同期のサービス メソッドを、クライアントとの同期通信を実行するコンポーネントにラップすることを検討します。
- 処理の実行順序を保証する必要がある場合や、以前の処理の結果に依存する操作を使用する場合は、非同期モデルの使用を検討します。
- 非同期のインプロセス呼び出しでは、プラットフォーム機能 (Begin や End のようなメソッドとコールバックなど) を使用して非同期のメソッド呼び出しを実装します。非同期のアウト プロセスの呼び出し (物理的なティアと境界をまたぐ呼び出しなど) では、メッセージングや非同期のサービス要求を使用することを検討します。

非同期通信を選択したときに、ネットワーク接続や呼び出し先の可用性を保証できない場合は、ストア アンド フォワードのメッセージ配信メカニズムを使用して、メッセージが紛失しないようにすることを考慮します。ストア アンド フォワードの設計方針を選択する場合は、システムやネットワークへの接続が切断した場合にメッセージを後

で配信できるように、ローカル キャッシュを使用してメッセージを格納することを検討します。また、システムやネットワークへの接続が切断されたり、システムやネットワークで障害が発生した場合にメッセージを後で配信できるように、メッセージ キューを使用してメッセージをキューに登録することを検討します。メッセージ キューでは、トランザクション メッセージを配信することが可能で、信頼できる 1 回限りの配信をサポートします。エンタープライズ レベルで他のシステムやプラットフォームと相互運用する必要がある場合や、電子データ交換を実行する必要がある場合は、BizTalk Server を使用して堅牢な配信メカニズムを提供することを検討します。

結合度と凝集性

アプリケーションの分散パーツ間に相互依存性をもたらす通信方法では、アプリケーションが密結合された状態になります。疎結合されたアプリケーションでは、通信を開始するために必要な要件を最小限に抑える方法を使用します。結合度と凝集性を設計する際には、次のガイドラインを考慮します。

- 疎結合では、ASP.NET Web サービス (ASMX) や WCF などのメッセージ ベースのテクノロジ、または自己記述型のデータと HTTP、REST、SOAP などの広く普及しているプロトコルの使用を検討します。
- 結合を維持するには、インターフェイスに、目的と機能分野に密接に関連したメソッドのみが含まれるようにします。

データ形式

ティア間でデータを渡す際に使用する最も一般的なデータ形式は、スカラ値、XML、DataSet、およびカスタム オブジェクトです。データ型を選択する際の主な考慮事項を次の表に示します。

データ型	**考慮事項**
スカラ値	組み込みのシリアル化のサポートが必要です。 スキーマが変更される可能性に対処できます。スカラ値は、メソッド シグネチャの変更を必要とする密結合を作成するので、呼び出し元のコードに影響を及ぼします。
XML	疎結合を使用する必要があります。疎結合では、呼び出し元が把握している必要がある情報は、ビジネス エンティティを定義するデータと、ビジネス エンティティのメタデータを提供するスキーマについてのみです。 サード パーティ製のクライアントを含む、さまざまな種類の呼び出し元をサポートする必要があります。

	組み込みのシリアル化のサポートが必要です。
DataSet	複雑なデータ構造をサポートする必要があります。 一連のデータと複雑なリレーションシップに対処する必要があります。 DataSet に含まれるデータへの変更をトラックする必要があります。 組み込みのシリアル化のサポートが必要です。
カスタム オブジェクト	複雑なデータ構造をサポートする必要があります。 オブジェクト型を認識するコンポーネントと通信します。 バイナリ シリアル化をサポートして、パフォーマンスを向上する必要があります。

通信チャネルのデータ形式を選択する際には、次のガイドラインを考慮します。

- アプリケーションで主にインスタンス データを操作する場合は、単純な値を使用してパフォーマンスを向上することを考慮します。単純な値の型を使用すると、初期の開発コストを削減できます。ただし、密結合が作成されるので、今後この型を変更しなければならない場合に、保守にかかるコストが増加する可能性があります。
- XML では、事前のスキーマ定義が他のデータ形式よりも多く必要になる場合がありますが、疎結合が作成されるので、将来の保守にかかるコストを削減して、相互運用性を向上できます (たとえば、インターフェイスを、XML 準拠の呼び出し元に公開しなければならない場合など)。
- DataSet は、複雑なデータ型 (特に、データベースから直接読み込まれる場合) に適切です。ただし、DataSet にはスキーマと状態に関する情報も含まれるため、ネットワーク経由でやり取りするデータの全体量が増加することと、DataSet の特殊な形式によって他のシステムとの相互運用性が制限される可能性があることを理解する必要があります。アプリケーションで主に一連のデータを操作し、並べ替え、検索、データ バインドなどの機能が必要な場合は、DataSet の使用を検討します。
- カスタム オブジェクトは、他のデータ形式では要件が満たされない場合や、カスタム オブジェクトを必要とするコンポーネントと通信する場合に適しています。カスタム オブジェクトは、DataSet に比べてオーバーヘッドが少なく、バイナリ シリアル化と XML シリアル化の両方をサポートします。通常、通信チャネル間のデータ送信に使用するカスタム オブジェクトは、ビジネス エンティティから抽出したデータを含むデータ転送オブジェクト (DTO) になります。
- 通信プロセスで型情報が失われないようにします。バイナリ シリアル化では、型の忠実度が保たれます。これは、クライアントとサーバー間でオブジェクトをやり取りする際に便利です。ただし、この手法では、より厳密なバージョン管理システムをインターフェイスに実装する必要があります。既定の

XML シリアル化では、パブリック プロパティとパブリック フィールドのみがシリアル化され、型の忠実度は保たれません。

相互運用性

アプリケーションとコンポーネントの相互運用性に影響する主な要素は、適切な通信チャネルを使用できるかどうかということと、通信に関連するコンポーネントが認識できる形式とプロトコルです。相互運用性を最大限に高めるには、次のガイドラインを考慮します。

- さまざまなプラットフォームおよびデバイスと通信できるようにするには、HTTP などの標準的なプロトコルと、XML のようなデータ形式の使用を検討します。選択するプロトコルは、ターゲット クライアントと通信できるかどうかに影響を及ぼす可能性があることに留意します。たとえば、通信先のシステムは、いくつかのプロトコルをブロックするファイアウォールで保護されている場合があります。
- インターフェイスとコントラクトのバージョン管理に関する問題を考慮します。ビジネス要件の変更、情報技術の要件、または他の問題によって、サービスを変更しなければならない場合があります。このような変更によってインターフェイス、メッセージ コントラクト、またはデータ コントラクトの互換性が失われる場合は、クライアントが使用できる新しいバージョンを作成することを検討します。この新しいバージョンでは、既存のクライアントが新しいインターフェイスで公開される機能にアクセスしなくても、以前のバージョンを使用することができます。詳細については、「サービスのバージョン管理」(http://msdn.microsoft.com/ja-jp/library/ms731060.aspx) を参照してください。
- 選択するデータ形式は、相互運用性に影響を及ぼすことがあります。たとえば、ターゲット システムが特定の種類のプラットフォームを認識しない場合や、ターゲット システムの処理とシリアル化の方法が異なる場合があります。
- 選択するセキュリティの暗号化と復号化の技法は、相互運用性に影響を及ぼすことがあります。たとえば、メッセージの暗号化と復号化の技法は、すべてのシステムで使用できるとは限りません。

パフォーマンス

通信インターフェイスの設計と使用するデータ形式は、(特にプロセス境界やコンピューターの境界を越える場合に) アプリケーションのパフォーマンスに大きな影響を及ぼします。相互運用性など、他の考慮事項では具体的なインターフェイスとデータ形式について考慮する必要がありますが、アプリケーションのレイヤー間やティア間の通信に関連するパフォーマンスの向上に役立つ技法があります。パフォーマンスについては、次のガイドラインを考慮します。

- 不要なデータは、できる限りリモート メソッドに渡さないようにして、ネットワーク経由で送信するデータ量を最小限に抑えます。このようにすると、シリアル化によるオーバーヘッドとネットワークの待ち時間が減少します。ただし、プロセス間の通信とコンピューター間の通信で、粒度の細かい (chatty な) インターフェイスを使用することは避けます (粒度の細かいインターフェイスでは、クライアントは、1 つの論理単位の作業を実行するために複数のメソッドを呼び出す必要があります)。Façade パターンを使用して、既存の粒度の細かいインターフェイスに粒度の粗いラッパーを提供することを検討します。
- 個々のデータ型を 1 つずつ渡すのではなく、DTO を使用してデータを 1 つの単位として渡すことを検討します。
- アプリケーションでシリアル化のパフォーマンスが重要な場合は、カスタム クラスとバイナリ シリアル化の使用を検討します。
- 相互運用性を確保するために XML を使用する必要がある場合、大量のデータには、要素ベースの構造ではなく、属性ベースの構造を使用することを検討します。

状態管理

アプリケーションの通信に関連する要素では、複数の要求にまたがって状態を管理しなくてはならない場合があります。状態管理の実装方法を決定する際には、次のガイドラインを考慮します。

- 呼び出し間の状態管理は、やむを得ない場合にのみ行います。状態管理はリソースを消費するので、アプリケーションのパフォーマンスに影響を及ぼしたり、配置オプションが制限されたりする場合があります。
- コンポーネントやサービス内でステートフルなプログラミング モデルを使用する場合は、データベースなどの永続的なデータ ストアを使用して状態情報を格納し、トークンを使用して情報にアクセスすることを検討します。
- ASMX サービスを設計する場合は、ApplicationContext クラスを使用して状態を保持することを検討します (ApplicationContext クラスを使用すると、アプリケーション スコープとセッション スコープで、既定の状態ストアへのアクセスが提供されます)。
- WCF サービスを設計する場合は、プラットフォームで提供される拡張可能なオブジェクトを使用して状態を管理することを検討します。拡張可能なオブジェクトを使用すると、サービス ホスト、サービス インスタンス コンテキスト、操作コンテキストなど、さまざまなスコープで状態を維持できます。この状態は、すべてメモリに保持され、永続的なものではありません。状態を永続的に保持する必要が

ある場合は、(.NET Framework 3.5 で導入された) 永続的なストレージを使用したり、独自のカスタム ソリューションを実装したりできます。

コントラクト ファーストの設計

従来、開発者は、要件に基づいてサービスを設計してから、コードと要件に適したインターフェイスを公開する、"コード ファースト" の手法を使用してサービスを構築していました。ただし、コントラクト ファーストの手法では、異なる種類のシステム間やさまざまなクライアント間で発生する可能性がある互換性の欠如を軽減できることから、この手法がより一般的になっています。

コントラクト ファーストの設計は、公開するデータ、メッセージ、およびインターフェイスに関するサービス コントラクトを設計してから、そのコントラクトに基づいてサービス インターフェイス コードを生成するプロセスです。そこから、必要な処理を実行する、サービス インターフェイスのコードを実装できます。これにより、プロセスの最初の段階で使用するメッセージの形式とデータ型に集中できるので、相互運用性と互換性を最大限に高められます。patterns & practices の Web Service Software Factory: Modeling Edition など、インターフェイスの設計に役立つモデリング ツールを使用することもできます (このツールの詳細については、http://msdn.microsoft.com/servicefactory/ (英語) を参照してください)。また、XML、XSD、およびスキーマを使用してインターフェイスを設計してから、WSDL.exe (/server スイッチを指定) などのツールを使用してインターフェイス定義を生成することも可能です。Microsoft BizTalk Server などのメッセージ バス テクノロジを使用すると、コントラクト ファーストの設計の原理を使用しやすくなります。

コントラクト ファーストの設計を適用する場合に覚えておく必要がある原理は、次のとおりです。

- XML スキーマとデータ型を使用すると、プラットフォーム固有のデータ型を考慮しない (できない) ことになります。このため、インターフェイスを定義するのはさらに難しくなりますが、相互運用性と互換性を最大限に高められます。複雑なデータ構造が必要な場合は、すべてのクライアントが使用できる単純で標準的な XML 形式のデータから、複雑なデータ構造を構成します。
- サービスと通信する可能性があるプラットフォーム、クライアント、およびシステムについて考慮します。これらによって発生する可能性があるデータ型やデータ形式に関する制限に備えます。
- サービス コントラクトの設計に役立つツールの使用を検討します。ツールを使用することで、プロセスを大幅に簡略化および高速化できます。

- コントラクトの設計プロセスでは、可能な限り関係者全員と共同作業を行います。他の関係者が抱えている要件や要求によって、コントラクトが使用しやすくなり、さらに広く受け入れられるようになって、再利用性が最大限に高まることがあります。

コントラクト ファーストの設計の詳細については、「Contract-First Service Development」(http://msdn.microsoft.com/en-us/magazine/cc163800.aspx、英語) を参照してください。

セキュリティに関する考慮事項

安全な通信方針は、機密データがネットワーク経由で渡される際に読み取られるのを防ぎ、機密データが改ざんされるのを防止し、必要に応じて呼び出し元の ID を保証します。通信の保護には、トランスポート セキュリティとメッセージ セキュリティという 2 つの重要な領域があります。保護を最大限に強化するために、トランスポート セキュリティとメッセージ セキュリティの技法を組み合わせることを検討します。

トランスポート セキュリティ

トランスポート セキュリティは、2 つのエンドポイント間のポイント ツー ポイントのセキュリティを提供するために使用し、トランスポート レイヤーでは、ユーザー資格情報と要求を受信者に渡します。チャネルを保護すると、攻撃者がチャネル上のすべてのメッセージにアクセスするのを防ぎます。トランスポート セキュリティの一般的な手法は、Secure Sockets Layer (SSL) 暗号化とインターネット プロトコル セキュリティ (IPSec) です。トランスポート セキュリティを使用するかどうかを決める際には、次の事項を考慮します。

- トランスポート セキュリティでは、優れた相互運用性を実現する一般的な業界標準を使用します。トランスポート セキュリティは、下位レイヤーで実現できるので (ネットワーク ハードウェアでも実現できることがあります)、通常、暗号化と署名でより高速に機能します。ただし、サポートされる資格情報と要求は、メッセージ セキュリティより少ないというデメリットがあります。
- サービスとコンシューマーの間の通信が中間デバイス経由でルーティングされない場合は、トランスポート セキュリティを使用できます。メッセージが中間デバイスを経由する場合は、メッセージ セキュリティを使用します。この場合にトランスポート セキュリティを使用すると、メッセージは、経由する中間デバイスで復号化されてから再度暗号化されるので、セキュリティ リスクとなります。
- トランスポート セキュリティは、イントラネットなどの社内ネットワークに配置されたクライアントとサービス間の通信を保護するのに適しています。

メッセージ セキュリティ

メッセージ セキュリティは、あらゆるトランスポート プロトコルと併用できます。チャネル経由で渡される個々のメッセージのコンテンツは、安全なネットワークの外側で渡される際には必ず、機密性の高いコンテンツについては、安全なネットワーク内で渡される場合であっても、保護する必要があります。メッセージ セキュリティの一般的な手法は、暗号化とデジタル署名です。メッセージ セキュリティを使用するかどうかを決める際には、次のガイドラインを考慮します。

- インターネット経由で公開されるサービスなど、安全なネットワークの外側に渡される機密メッセージには、メッセージ セキュリティを実装することを検討します。ただし、一般的に、メッセージ セキュリティはトランスポート セキュリティよりも通信のパフォーマンスに大きな影響を及ぼすので、注意が必要です。メッセージの暗号化と署名は、部分的または特定のメッセージにのみ使用して、全体的なパフォーマンスを向上することもできます。
- クライアントとサービスの間に中間デバイスが存在する場合は、エンド ツー エンドのセキュリティが保証されるので、機密メッセージにはメッセージ セキュリティを使用します。中間サーバーは、メッセージを受信したときに SSL 接続や IPSec 接続を終了し、メッセージを次のサーバーに渡すために、新しい SSL 接続や IPSec 接続を作成します。そのため、メッセージ セキュリティを使用していないメッセージは、中間サーバーでアクセスされる危険性があります。

テクノロジの選択肢

マイクロソフト プラットフォームでは、Windows Communication Foundation (WCF) と ASP.NET Web サービス (ASMX) のどちらかのメッセージ テクノロジを選択できます。これ以降のセクションでは、各テクノロジの機能を理解し、シナリオに最適なテクノロジを選択するのに役立つ情報を提供します。

WCF テクノロジの選択肢

WCF では、さまざまな状況でサービスを実装するための包括的なメカニズムが提供され、サービスの構成と内容を細かく制御できます。次のガイドラインは、WCF の使用方法について理解するのに役立ちます。

- 次の状況では、WCF を使用することを検討します。
    - SOAP をサポートする他のプラットフォームとの相互運用性を確保する必要がある Web サービスと通信する場合 (J2EE ベースのアプリケーション サーバーなど)

    - SOAP ベースではないメッセージを使用して Web サービスと通信する場合 (RSS (Really Simple Syndication) などの形式を使用するアプリケーションなど)
    - サーバーとクライアントの両方が WCF を使用する場合に、SOAP メッセージとデータ構造のバイナリ エンコードを使用して通信する場合
    - REST Singleton & Collection Services (REST シングルトンと収集サービス)、ATOM Feed and Publishing Protocol Services (ATOM フィードと出版プロトコル サービス)、および HTTP Plain XML Services (HTTP プレーン XML サービス) を構築している場合

- SOAP 要求で WS-MetadataExchange を使用して、サービスに関する情報 (サービスの Web サービス記述言語 (WSDL) 定義とポリシーなど) を取得することを検討します。
- WS-Security を使用して、認証、データ整合性、データ プライバシー、およびその他のセキュリティ機能を実装することを検討します。
- WS-Reliable メッセージングを使用して、複数の Web サービスの中間デバイスを経由する必要がある場合でも、信頼できるエンド ツー エンドの通信を実装することを検討します。
- WS-Coordination を使用して、Web サービスのメッセージ交換コンテキストで、2 フェーズ コミット トランザクションを調整することを検討します。

WCF では、いくつかの通信プロトコルがサポートされます。

- インターネットからアクセスするサービスでは、HTTP プロトコルの使用を検討します。
- プライベート ネットワーク内でアクセスするサービスでは、TCP プロトコルの使用を検討します。
- 同じコンピューターからアクセスするサービスでは、名前付きパイプ プロトコルの使用を検討します。このプロトコルでは、共有バッファやデータを渡すためのストリームがサポートされます。

ASMX テクノロジの選択肢

ASMX では、インターネット インフォメーション サービス (IIS) Web サーバー経由で公開される ASP.NET ベースの Web サービスを構築するための、簡略化されたソリューションが提供されます。ASMX には次の特性があります。

- インターネット経由でアクセスできますが、使用できるプロトコルは HTTP のみです。既定でポート 80 が使用されますが、これは簡単に再構成できます。

- 分散トランザクション コーディネーター (DTC) のトランザクション フローがサポートされません。カスタム実装を使用して、長時間トランザクションをプログラムする必要があります。
- IIS 認証、承認に使用する Windows グループとして格納される役割、IIS と ASP.NET の偽装、および SSL トランスポート セキュリティがサポートされます。
- IIS に実装されているエンドポイント テクノロジをサポートします。
- プラットフォーム間の相互運用性が提供されます。

関連情報

Web リソースに簡単にアクセスするには、http://www.microsoft.com/architectureguide (英語) でオンライン版の参考文献を参照してください。

- データ転送とシリアル化 (http://msdn.microsoft.com/ja-jp/library/ms730035.aspx)
- エンドポイント: アドレス、バインディング、およびコントラクト (http://msdn.microsoft.com/ja-jp/library/ms733107.aspx)
- SOA のメッセージングパターン (パート 1) (http://msdn.microsoft.com/ja-jp/library/cc947720.aspx)
- Principles of Service Design: Service Versioning (http://msdn.microsoft.com/en-us/library/ms954726.aspx、英語)
- Web Service Messaging with Web Services Enhancements 2.0 (http://msdn.microsoft.com/en-us/library/ms996948.aspx、英語)
- Web サービス プロトコルの相互運用性ガイド (http://msdn.microsoft.com/ja-jp/library/ms734776.aspx)
- Windows Communication Foundation セキュリティ (http://msdn.microsoft.com/ja-jp/library/ms732362.aspx)
- ASP.NET を使用した XML Web サービス (http://msdn.microsoft.com/ja-jp/library/ba0z6a33.aspx)

第Ⅲ部　アプリケーションの原型

このセクションは、一般的なアプリケーションの種類ごとに、その機能、メリット、およびデメリットを理解するのに役立つトピックで構成されています。1 つ目のトピックでは、Web アプリケーション、モバイル アプリケーション、リッチ クライアント アプリケーション、サービス アプリケーション、RIA など基本的な種類のアプリケーションに関する概要を提供します。それ以降のトピック (章) では、各アプリケーションの種類の詳細について説明します。また、ホストされているクラウド ベースのサービス、SharePoint や Microsoft Office を活用するアプリケーションなど、特殊な種類のアプリケーションについて説明します。詳細については、次の章を参照してください。

- 第 20 章「アプリケーションの種類の選択」
- 第 21 章「Web アプリケーションの設計」
- 第 22 章「リッチ クライアント アプリケーションの設計」
- 第 23 章「リッチ インターネット アプリケーションの設計」
- 第 24 章「モバイル アプリケーションの設計」
- 第 25 章「サービス アプリケーションの設計」
- 第 26 章「ホストされているクラウド サービス」
- 第 27 章「Office Business Application の設計」
- 第 28 章「SharePoint LOB アプリケーションの設計」

19

物理ティアと配置

概要

アプリケーション アーキテクチャの設計は、モデル、ドキュメント、およびシナリオという形で存在しています。しかし、アプリケーションは物理環境に配置する必要があるので、アーキテクチャに関する決定の一部がインフラストラクチャの制限事項によって実現できなくなることがあります。そのため、アプリケーションの設計プロセスでは、提案された配置シナリオとインフラストラクチャを検討する必要があります。この章では、分散スタイルと非分散スタイルなど、さまざまな種類のアプリケーションの配置に使用できるオプション、アプリケーションの拡張方法、およびパフォーマンス、信頼性、セキュリティの各問題についてのガイダンスとパターンについて説明します。設計プロセスの一環としてアプリケーションで可能な配置シナリオを考慮することで、技術的なインフラストラクチャの制限事項によりアプリケーションを正常に配置できなかったり設計要件を達成できなかったりする事態を回避できます。配置に関する方針を選択する際には設計のトレードオフが発生します。たとえば、プロトコルやポートの制約が存在する場合や、配置先の環境で特定の配置トポロジがサポートされていない場合があります。設計の早い段階で配置に関する制約を特定して、後で予想外の事態が発生しないようにします。また、ネットワーク チームとインフラストラクチャ チームのメンバーにも、設計プロセスに協力してもらいます。配置に関する方針を選択する際は、次の項目について理解します。

- 配置先の物理環境
- 配置環境に基づいたアーキテクチャや設計に関する制約
- 配置環境がセキュリティとパフォーマンスに及ぼす影響

分散配置と非分散配置

配置に関する方針を作成する際には、まず分散配置と非分散配置のどちらのモデルを使用するかを決定します。限られたユーザーがアクセスする、組織内で使用する単純なイントラネット アプリケーションを構築する場合は、非分散配置の使用を検討します。スケーラビリティと保守容易性を最大限に高める必要がある、より複雑なアプリケーションを構築する場合は、分散配置の使用を検討します。

非分散配置

非分散配置では、データ ストレージ機能を除くすべての機能とレイヤーが 1 台のサーバーに配置されます (図 1 参照)。

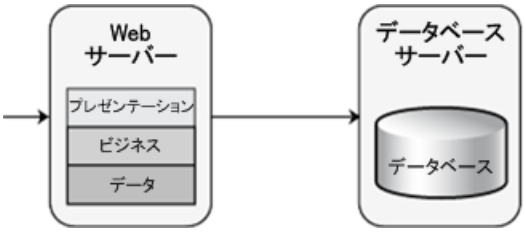


図 14

非分散配置

この手法のメリットは構造が単純なことで、必要な物理サーバー数を最小限に抑えられます。また、レイヤー間の通信がサーバーやサーバー クラスター間の物理的な境界を越える必要がある場合に発生するパフォーマンスへの影響を、最小限に抑えられます。1 台のサーバーを使用することで、通信に関するパフォーマンスのオーバーヘッドを最小限に抑えられますが、他の領域でパフォーマンスが低下することがある点に注意してください。すべてのレイヤーでリソースが共有されるので、あるレイヤーが集中的に使用されると、他のすべてのレイヤーがそのレイヤーの影響を受ける場合があります。また、非常に厳密な運用上の要件に従ってサーバーを汎用的に構築および設計する必要があり、システム リソースを最も多く使用するコンシューマーの最大使用量をサーバーでサポートする必要があります。単一のティアを使用すると、すべてのレイヤーで同じ物理ハードウェアが共有されるので、全体的なスケーラビリティと保守容易性が低下します。

分散配置

分散配置では、アプリケーションのレイヤーが別個の物理ティアに配置されます。ティア型の分散によって、システムのインフラストラクチャが一連の物理ティアに分類され、特定の運用上の要件とシステム リソース使用量に合わ

せて最適化されたサーバー環境が提供されます。これにより、アプリケーションのレイヤーを異なる物理ティアに分割できます (図 2 参照)。

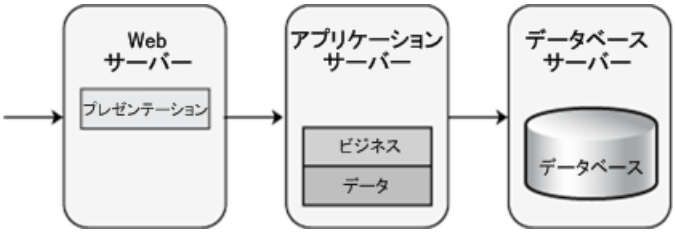


図 15

分散配置

分散アプローチを使用すると、さまざまなレイヤーをホストするアプリケーション サーバーを構成して、各レイヤーの要件を最適な状態で満たすことができます。ただし、コンポーネントの配置を最適化する最大の手段はコンポーネントのリソース使用量についての特性を適切なサーバーに合わせることなので、レイヤーとティアの直接的なマッピングは、多くの場合、配置に関する方針として最適ではありません。

複数のティアを使用すると複数の環境を実現できます。一連の運用上の要件とシステム リソース使用量に合わせて、各環境を最適化できます。このようにすると、リソースの要件に最も合うティアにコンポーネントを配置して、運用上のパフォーマンスと動作を最大限に高めることができます。使用するティアが増えるほど、各コンポーネントで選択できる配置オプションが増えます。分散配置によって柔軟な環境が提供され、この環境ではパフォーマンスの制約事項が発生した場合や処理の要求が増加したときに各物理ティアをさらに柔軟にスケールアウトまたはスケールアップできます。ただし、ティアを追加するほど、複雑さ、配置の手間、およびコストが増大する点に注意してください。

ティアを追加するもう 1 つの目的は、特定のセキュリティ ポリシーを適用することです。分散配置を採用すると、より厳密なセキュリティをアプリケーション サーバーに適用できます。このようなセキュリティを適用するには、たとえば、Web サーバーとアプリケーション サーバーの間にファイアウォールを追加したり、さまざまな認証と承認のオプションを使用したりします。

分散環境のパフォーマンスと設計に関する考慮事項

ティア間でコンポーネントを分散すると、物理的な境界を越えるリモート呼び出しのコストによってパフォーマンスが低下することがあります。ただし、コンポーネントを分散すると、スケーラビリティを拡大する機会が増加し、管理容易性を向上して、時間の経過に伴うコストを削減できます。物理的に分散したインフラストラクチャで実行されるアプリケーションを設計する際には、次のガイドラインを考慮します。

- パフォーマンスの低下を最小限に抑えながらコンポーネントが安全に通信できるように、ティア間の通信パスとプロトコルを選択します。非同期呼び出し、一方向の呼び出し、またはメッセージ キューを使用して、物理的な境界を越える呼び出しを行う際のブロックをできる限り防止します。
- 分散トランザクションのサポートや認証など、設計を簡略化して相互運用性を向上できるサービスやオペレーティング システムの機能の使用を検討します。
- コンポーネントのインターフェイスを簡略化します。タスクの実行に多数の呼び出しを必要とする、粒度の細かいインターフェイス (chatty なインターフェイス) は、同じ物理コンピューター上に配置されている場合に最適な状態で動作します。1 回の呼び出しで 1 つのタスクを実行するインターフェイス (chunky なインターフェイス) は、コンポーネントが別個の物理ティアに分散されている場合に最高のパフォーマンスを発揮します。ただし、インプロセス呼び出しだけでなく他の物理ティアからの呼び出しもサポートする必要がある場合は、インプロセス呼び出し用に粒度の細かいインターフェイスを実装し、他の物理ティアで使用するために、呼び出しをラップして chunky なインターフェイスを提供するファサードを実装することを検討できます。
- 別個の物理クラスターを使用して、エラーが発生する恐れのある他のプロセスから長時間に渡る重要なプロセスを分離することを検討し、フェールオーバーの方針を決定します。たとえば、通常、Web サーバーには、大量のメモリが搭載されて高い処理能力が備わっていますが、ハードウェアの障害時に迅速に置き換えられる堅牢なストレージ機能 (RAID によるミラーリング) が用意されていないことがあります。
- パフォーマンスと可用性を向上するサーバーやリソースを追加する計画を立てるための最適な方法を特定します。
- レイヤー間で物理的な境界を越えて通信する場合は、スケーラビリティとパフォーマンスに影響を及ぼすので、ティア間で状態を管理する方法を検討する必要があります。状態管理の一般的なオプションは次のとおりです。
    - **ステートレス**: ティアを呼び出すと、必要な状態がすべて提供されます。スケーラビリティが向上しやすい一方、多くの場合、クライアントが状態情報を提供する必要があります。
    - **ステートフル**: クライアント要求があるたびに状態が格納または回復されます。より多くのリソースが必要になるのでステートレスよりスケーラビリティの低いソリューションになりますが、クライアントで状態情報を追跡して提供する必要がないので、多くの場合に便利なオプションです。

分散配置にコンポーネントを配置する際の推奨事項

分散配置を設計する際には、各物理ティアに配置する論理レイヤーと論理コンポーネントを決定する必要があります。ほとんどの場合は、プレゼンテーション レイヤーをクライアントまたは Web サーバーに配置し、サービス レイヤー、ビジネス レイヤー、およびデータ レイヤーをアプリケーション サーバーに、データベースを専用のサーバーに配置します。場合によっては、この構成の変更が必要になることがあります。分散環境でコンポーネントの配置先を決定する際には、次のガイドラインを考慮します。

- 必要な場合にのみコンポーネントを分散します。分散配置を実装する一般的な理由は、セキュリティ ポリシー、物理的な制約、共有のビジネス ロジック、およびスケーラビリティです。
- 複数のプレゼンテーション コンポーネントで複数のビジネス コンポーネントが同時に使用される場合、ビジネス コンポーネントをプレゼンテーション コンポーネントと同じ物理ティアに配置します。この配置により、パフォーマンスが最大限に高まり、運用管理が容易になります。
- セキュリティ上の影響によりコンポーネント間の信頼境界が必要になる場合は、プレゼンテーション コンポーネントとビジネス コンポーネントを同じティアに配置しないようにします。たとえば、リッチ クライアント アプリケーションでは、プレゼンテーション コンポーネントをクライアントに配置し、ビジネス コンポーネントをサーバーに配置して、ビジネス コンポーネントとプレゼンテーション コンポーネントを分離する場合があります。
- コンポーネント間の信頼境界が必要になるセキュリティ上の影響がない限り、サービス エージェント コンポーネントは、コンポーネントを呼び出すコードと同じティアに配置します。
- ワークフロー コンポーネントと共に非同期に呼び出されるビジネス コンポーネントは、できる限り他のレイヤーとは別個の物理ティアに配置します。
- ビジネス エンティティは、エンティティを使用するコンポーネントと同じ物理ティアに配置します。

分散配置のパターン

多くのソリューションで採用されているアプリケーションの配置構造には、いくつかの一般的なパターンがあります。アプリケーションに最適な配置ソリューションを決定する際には、まず、一般的なパターンを確認します。さまざまなパターンを十分に理解したら、次にシナリオ、要件、およびセキュリティ上の制約を検討して最も適切なパターンを選択します。

クライアント/サーバーの配置

このパターンは、クライアントおよびサーバーという 2 つの主要コンポーネントで構成される基本的な構造を表します。通常、このシナリオでは、クライアントとサーバーは 2 つの別個のティアに配置されます。図 3 に、クライアントと Web サーバーが通信する一般的な Web アプリケーションのシナリオを示します。

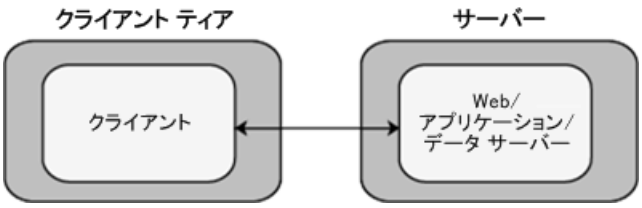


図 16

一般的な Web アプリケーションのシナリオ

アプリケーション サーバーにアクセスするクライアント、または別のデータベース サーバーにアクセスするスタンドアロン クライアントを開発する場合は、クライアント/サーバーのパターンの使用を検討します。

n ティアの配置

n ティアのパターンは、アプリケーションのコンポーネントが 1 台以上のサーバーに分割された汎用的な分散パターンを表します。一般的には、次のセクションで説明するように 2 ティア、3 ティア、または 4 ティアのいずれかのパターンを使用します。1 つのレイヤーのコンポーネントはすべて同じティアに配置する場合が多いですが、必ずしもこのように配置するとは限りません。レイヤーを単一のティアに限定する必要はありません。必要に応じて、複数のサーバーに負荷を分散できます。たとえば、ビジネス ロジックのさまざまな側面を含むサイド バイ サイドのティアを使用できます。

2 ティアの配置

実質的に、これはクライアント/サーバーのパターンと同じ物理レイアウトです。主な違いは、ティア上のコンポーネント間の通信方法です。図 4 のように、すべてのアプリケーション コードがクライアントに配置され、データベースが別個のサーバーに配置されることがあります。クライアントでは、データベース サーバーのストアド プロシージャや最小限のデータ アクセス機能が使用されます。

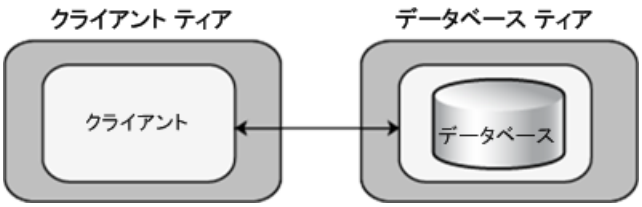


図 17

すべてのアプリケーション コードがクライアントに配置された 2 ティアの配置

アプリケーション サーバーにアクセスするクライアント、または別のデータベース サーバーにアクセスするスタンドアロン クライアントを開発している場合は、2 ティアのパターンの使用を検討します。

3 ティアの配置

3 ティアの設計では、クライアントは別個のサーバーに配置されたアプリケーション ソフトウェアと通信し、アプリケーション サーバーは別のサーバーに配置されたデータベースと通信します (図 5 参照)。これは、ほとんどの Web アプリケーションや Web サービスで非常に一般的なパターンであり、ほとんどの一般的なシナリオに十分対応できます。クライアントと Web ティアまたはアプリケーション ティアとの間にファイアウォールを実装し、Web ティアまたはアプリケーション ティアとデータベース ティアとの間に別のファイアウォールを実装することがあります。

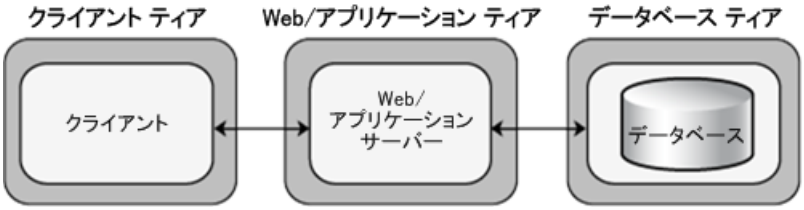


図 18

アプリケーション コードが別個のティアに配置された 3 ティアの配置

すべてのサーバーが社内ネットワーク内に配置されているイントラネット ベースのアプリケーションや、セキュリティ要件によって、一般に公開される Web サーバーやアプリケーション サーバーにビジネス ロジックを実装できないインターネット ベースのアプリケーションを開発する場合は、3 ティアのパターンの使用を検討します。

4 ティアの配置

このシナリオでは、図 6 のように、Web サーバーがアプリケーション サーバーと物理的に分離しています。このシナリオはセキュリティ上の理由で選択されることが多く、Web サーバーが境界ネットワークに配置され、別のサブネット上のアプリケーション サーバーにアクセスします。このシナリオでは、クライアント ティアと Web ティ

アとの間にファイアウォールを実装し、Web ティアとアプリケーション ティアまたはビジネス ロジック ティアとの間に別のファイアウォールを実装することがあります。

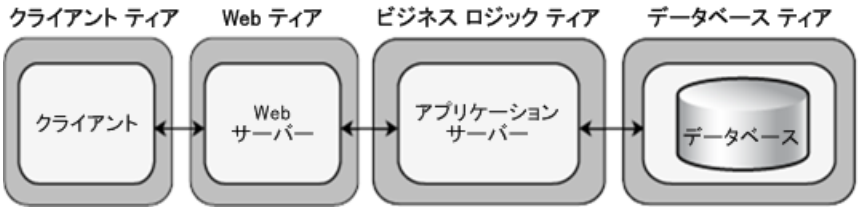


図 19

Web コードとビジネス ロジックが個別のティアに配置された 4 ティアの配置

セキュリティ要件によって、ビジネス ロジックを境界ネットワークに配置できない場合、またはサーバー リソースを大量に使用するアプリケーション コードがあり、その機能を別のサーバーにオフロードする必要がある場合は、4 ティアのパターンの使用を検討します。

Web アプリケーションの配置

セキュリティの懸念事項によりフロントエンド Web サーバーにビジネス ロジックを配置できない場合は、分散配置を Web アプリケーションに使用することを検討します。ビジネス レイヤーにメッセージ ベースのインターフェイスを使用し、パフォーマンスを最適化するために TCP プロトコルとバイナリ エンコードを使用したビジネス レイヤーとの通信を検討します。また、負荷分散によって要求を分散して要求がさまざまな Web サーバーで処理されるようにすることを検討し、スケーラビリティが高い Web アプリケーションを設計する際にはサーバー アフィニティを回避し、Web アプリケーションで使用するステートレスなコンポーネントを設計する必要もあります。詳細については、この章の後半の「パフォーマンスに関するパターン」を参照してください。

リッチ インターネット アプリケーションの配置

リッチ インターネット アプリケーション (RIA) では、プレゼンテーション ロジックがクライアントに移動されるので、分散アーキテクチャは、RIA で最も使用される可能性が高い配置シナリオです。ビジネス ロジックを他のアプリケーションと共有している場合は、分散配置の使用を検討します。また、ビジネス ロジックにメッセージ ベースのインターフェイスを使用することも検討します。

リッチ クライアント アプリケーションの配置

n ティアの配置では、プレゼンテーション ロジックとビジネス ロジックをクライアントに配置するか、プレゼンテーション ロジックだけをクライアントに配置できます。図 7 に、プレゼンテーション ロジックとビジネス ロジックをクライアントに配置した場合を示します。

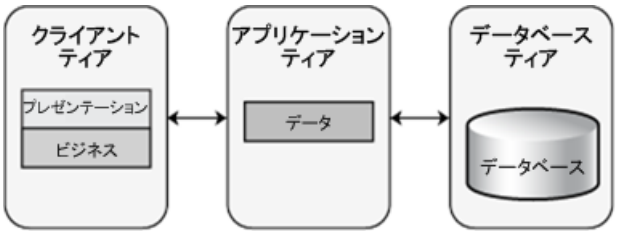


図 20

ビジネス レイヤーがクライアント ティアに配置されたリッチ クライアント

図 8 に、ビジネス ロジックとデータ アクセス ロジックをアプリケーション サーバーに配置した場合を示します。

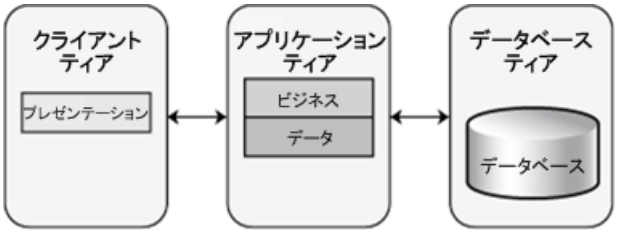


図 21

ビジネス レイヤーがアプリケーション ティアに配置されたリッチ クライアント

パフォーマンスに関するパターン

パフォーマンスに関する配置パターンは、一般的なパフォーマンスの問題に対する実証済みの設計ソリューションを表します。パフォーマンスの高い配置を検討している場合は、スケールアップまたはスケールアウトの手法を使用できます。スケールアップすると、現在アプリケーションを実行しているハードウェアが強化されます。スケールアウトすると、アプリケーションが複数の物理サーバーに分散され、負荷が分散されます。スケールアウトを採用する予定の場合は、主に負荷分散の方針を使用します。この配置は、通常、負荷分散クラスターと呼ばれ、Web サーバーの場合は Web ファームと呼ばれます。次のセクションでは、これらのパターンについて説明します。スケールアップまたはスケールアウトするタイミングと方法の選択に関する詳細については、この章の後半の「スケールアップとスケールアウト」を参照してください。

負荷分散クラスター

負荷を共有するように構成された複数のサーバーに、サービスまたはアプリケーションをインストールできます (図 9 参照)。この種類の構成は、負荷分散クラスターと呼ばれます。

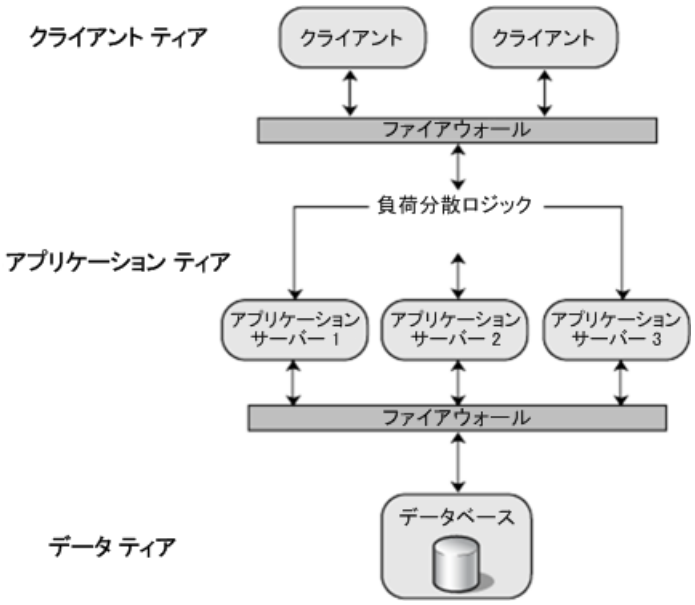


図 22

負荷分散クラスター

負荷分散では、クライアント要求を複数のサーバーに分散することで、サーバー ベースのプログラム (Web サーバーなど) のパフォーマンスが強化されます。この負荷分散テクノロジ (通称、ロードバランサー) では、受信要求を受け取ると、必要に応じて特定のホストにリダイレクトします。負荷分散するように構成された複数のホストでは、同じクライアントから複数の要求が行われた場合でも、同時に異なるクライアント要求に応答します。たとえば、Web ブラウザーでは、単一の Web ページに含まれる複数の画像をクラスター内の異なるホストから取得する場合があります。これによって負荷が分散され、処理速度が向上し、応答時間が短縮されます。

使用するルーティング テクノロジによっては、障害の発生したサーバーが検出されてルーティング先の一覧から削除されるので、障害の影響を最小限に抑えられます。単純なシナリオでは、ルーティングはラウンド ロビン方式で行われ、DNS サーバーによって各サーバーのアドレスが順に指定されます。図 10 に、各サーバーでデータ ストアを除くすべてのアプリケーションのレイヤーがホストされている単純な Web ファーム (Web サーバーの負荷分散クラスター) を示します。

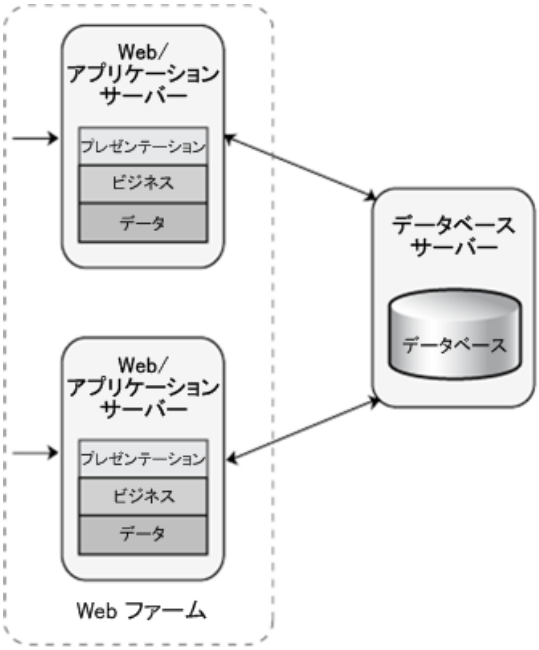


図 23

単純な Web ファーム

負荷分散クラスターは、各クライアントの要求間で情報を追跡して保存する必要がない場合 (つまりステートレスな場合) に、スケーラビリティと効率が最も高くなります。クラスターで状態を追跡する必要がある場合は、アフィニティとセッションの技法を使用しなければならないことがあります。

アフィニティとユーザー セッション

アプリケーションでは、同じクライアントからの要求間でセッション状態が保持されることが前提となっていることがあります。たとえば、Web サーバーでは要求間のユーザー情報を追跡しなければならない場合があります。同一ユーザーからのすべての要求を同一サーバーにルーティングするように Web ファームを構成して (この処理はアフィニティと呼ばれます)、この情報が Web サーバーのメモリに格納されている状態を保持できます。しかし、可用性と信頼性を向上するには、個別のセッション状態ストアを Web ファームと併用して、アフィニティの必要性をなくす必要があります。開発時にインターネット インフォメーション サービス (IIS) 6.0 以降を使用していれば、Web ガーデン モードで動作するように IIS を構成して、アプリケーションの開発時にセッション状態がアプリケーションで適切に処理されるようにできます。

ASP.NET の場合、アフィニティを実装しないときは一貫した暗号化キーと暗号化方法が ViewState の暗号化に使用されるように、すべての Web サーバーを構成する必要もあります。また、システムで Secure Sockets Layer (SSL) 暗号化がサポートされている場合は SSL 暗号化を使用するセッションでアフィニティを有効にし、SSL がサポートされていない場合は SSL 要求に別のクラスターを使用する必要があります。

アプリケーション ファーム

ビジネス レイヤーとデータ レイヤーがプレゼンテーション レイヤーと異なる物理ティアに配置されている場合は、Web サーバーや Web ファームと同様に、これらのレイヤーもアプリケーション ファームを使用してスケールアウトできます。プレゼンテーション ティアからの要求は、サーバー間の負荷がほぼ同じになるようにファーム内の各サーバーに分散されます。各レイヤーの要件、および予期される負荷とユーザー数によっては、ビジネス レイヤーのコンポーネントとデータ レイヤーのコンポーネントを異なるアプリケーション ファームに分離できます。

信頼性に関するパターン

信頼性に関する配置パターンは、一般的な信頼性の問題に対する実証済みの設計ソリューションを表します。配置の信頼性を向上する最も一般的な手法は、フェールオーバー クラスターを使用して、サーバーで障害が発生したときでもアプリケーションの可用性を確保することです。

フェールオーバー クラスター

フェールオーバー クラスターは、あるサーバーが使用できなくなると、障害が発生したサーバーの処理が別のサーバーに自動的に引き継がれて処理が続行されるように構成された一連のサーバーです。図 11 に、フェールオーバー クラスターを示します。

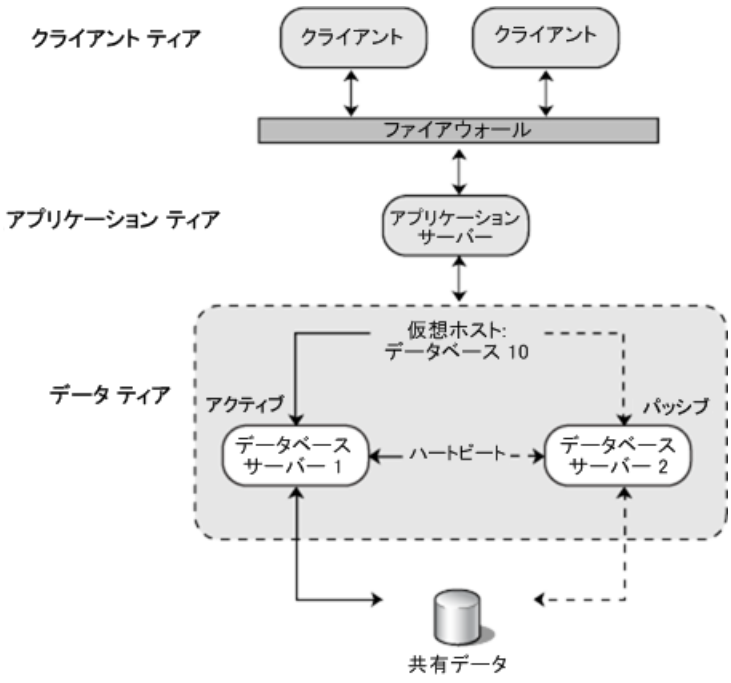


図 24

フェールオーバー クラスター

障害発生時に互いの処理が引き継がれるように構成された複数のサーバーに、アプリケーションまたはサービスをインストールします。障害が発生したサーバーの処理を別のサーバーで引き継ぐ処理は、一般にフェールオーバーと呼ばれています。クラスター内の各サーバーには、スタンバイ サーバーとして認識されているサーバーが少なくとも 1 台あります。

セキュリティに関するパターン

セキュリティに関するパターンは、一般的なセキュリティの問題に対する実証済みの設計ソリューションを提示します。偽装とデリゲートの手法は、最初の呼び出し元のコンテキストをアプリケーションのダウンストリーム レイヤーやコンポーネントに渡す必要がある場合に適したソリューションです。信頼されたサブシステムの手法は、アップストリーム コンポーネントで認証と承認を処理し、単一の信頼された ID でダウンストリーム リソースにアクセスする場合に適したソリューションです。

偽装とデリゲート

偽装とデリゲートの承認モデルでは、各ユーザーに許可されたリソースと操作の種類 (読み取り、書き込み、削除など) は、Windows のアクセス制御リスト (ACL)、またはアクセス先リソース (SQL Server のテーブルとプロシージャなど) の ACL に相当するセキュリティ機能を使用してセキュリティ保護されます。ユーザーは、偽装により各自の本来の ID を使用してリソースにアクセスします (図 12 参照)。この手法ではドメイン アカウントが必要になることがあるので、シナリオによっては適さない場合があることに注意してください。

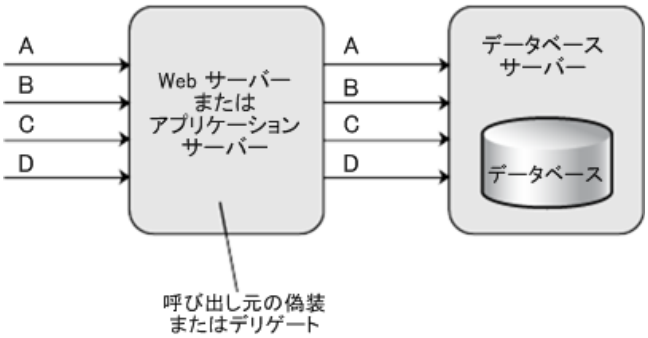


図 25

偽装とデリゲートの承認モデル

信頼されたサブシステム

信頼されたサブシステム (信頼されたサーバー) モデルでは、ユーザーはアプリケーションで定義された論理的な役割に分類されます。特定の役割のメンバーは、アプリケーション内で同じ特権を共有します。操作へのアクセス (通常、メソッドの呼び出し) は、呼び出し元に割り当てられている役割に基づいて承認されます。この役割ベース (操作ベース) のセキュリティ手法では、操作 (ネットワーク リソース以外) へのアクセスは呼び出し元に割り当てられている役割に基づいて承認されます。役割はアプリケーションの設計時に分析および定義され、アプリケーション内で同じセキュリティ特権や機能を共有しているユーザーをまとめる論理的なコンテナーとして使用されます。中間ティア サービスでは、固定 ID を使用してダウンストリーム サービスやダウンストリーム リソースにアクセスします (図 13 参照)。

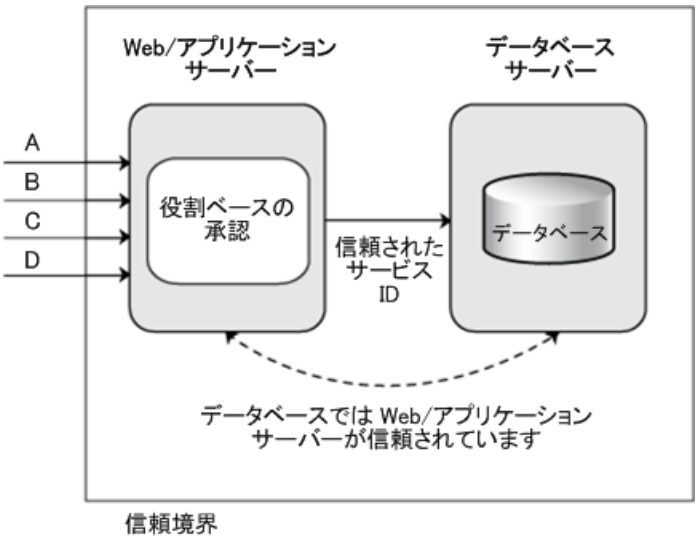


図 26

信頼されたサブシステム (信頼されたサーバー) モデル

複数の信頼されたサービス ID

状況によっては、複数の信頼された ID が必要なことがあります。たとえば、2 つのユーザー グループがあり、一方のグループは読み書き操作の実行が承認される必要があり、もう一方のグループは読み取り専用の操作の実行が承認される必要があるとします。複数の信頼されたサービス ID を使用すると、スケーラビリティに大きな影響を及ぼすことなく、リソースのアクセスと監査をより細かく制御できるようになります。図 14 に、複数の信頼されたサービス ID のモデルを示します。

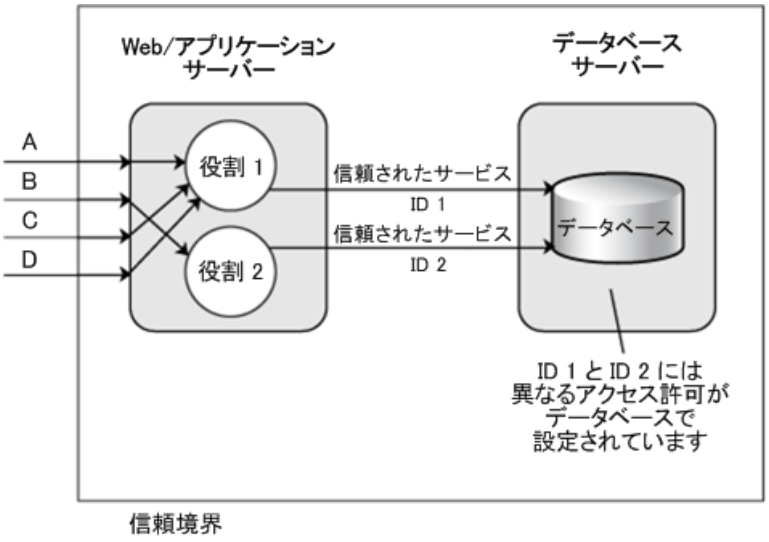


図 27

複数の信頼されたサービス ID のモデル

スケールアップとスケールアウト

スケーリングの手法は、設計に関する重要な考慮事項です。ソリューションのスケールアウトに負荷分散クラスターと分割データベースのどちらを使用する予定でも、選択するオプションは設計でサポートされるようにする必要があります。一般的に、アプリケーションを拡張する際には、スケールアップ (コンピューターの機能拡張) とスケールアウト (コンピューターの台数増加) の 2 つの基本的なオプションを選んで組み合わせることができます。

スケールアップの手法では、プロセッサ、RAM、ネットワーク インターフェイス カード (NIC) などのハードウェアを既存のサーバーに追加して、より高い性能をサポートします。これは単純なオプションで、保守とサポートのコストが増加しないので、ある程度まではコスト効率のよいオプションです。しかし、単一障害点が残り、これはリスクになります。また、一定のしきい値を超えると、既存のサーバーにハードウェアを追加しても目的の結果が得られなくなることがあり、アップグレードによって 1 台のサーバーで論理的に向上できるとされているパフォーマンスの最後の 10% を達成するためのコストは非常に高くなることがあります。

アプリケーションを効率的にスケールアップするには、基になるフレームワーク、ランタイム、およびコンピューターのアーキテクチャもスケールアップする必要があります。スケールアップする際には、アプリケーションのパフォーマンスを制限しているリソースを考慮します。たとえば、メモリやネットワークに関するリソースがパフォーマンスのボトルネックとなっている場合は、CPU リソースを追加しても効果はありません。

スケールアウトの手法では、サーバーを追加し、負荷分散とクラスタリングのソリューションを使用します。スケールアウトのシナリオでは、より多くの負荷を処理できるようになるだけでなく、ハードウェア障害も軽減されます。

というのも、1 台のサーバーで障害が発生しても、負荷を引き継ぐことのできる他のサーバーがクラスター内にあるからです。たとえば、Web ファームに複数の、負荷分散するように構成された Web サーバーを配置して、これらのサーバーでプレゼンテーション レイヤーとビジネス レイヤーをホストできます。また、アプリケーションのビジネス ロジックを物理的に分割してそのロジックに別個の、負荷分散するように構成された中間ティアを使用しながら、負荷分散するように構成されたフロント ティアでプレゼンテーション レイヤーをホストすることもできます。アプリケーションが I/O の制約を受けていて、非常に大きなデータベースをサポートする必要がある場合は、データベースを複数のデータベース サーバーに分配できます。一般に、アプリケーションをスケールアウトできるかどうかは、基になるインフラストラクチャよりもアプリケーションのアーキテクチャによって決まります。

スケールアップに関する考慮事項

プロセッサの処理能力の向上やメモリの増設によるスケールアップは、コスト効率のよいソリューションになる場合があります。この手法では、スケールアウトの手法や Web ファームとクラスタリング テクノロジの使用に伴う追加の管理コストが発生しません。まず、スケールアップのオプションを確認し、パフォーマンス テストを実施する必要があります。テストでは、ソリューションをスケールアップすることで、定義したスケーラビリティの条件が満たされるかどうか、および必要な数の同時接続ユーザーが許容できるパフォーマンスのレベルでサポートされるかどうかを確認します。実際の成長に応じたシステムの拡張計画を立てる必要があります。

スケールアウトのサポートに関する設計

CPU、I/O、またはメモリのしきい値に達したために、ソリューションをスケールアップしても適切なスケーラビリティが得られない場合は、スケールアウトして追加のサーバーを導入する必要があります。アプリケーションを正常にスケールアウトできるようにするには、次の設計の慣例を考慮します。

- **ボトルネックの特定とスケールアウト**: それ以上スケールアップできない共有リソースは、多くの場合ボトルネックになります。たとえば、複数のアプリケーション サーバーからアクセスする単一の SQL Server インスタンスがあるとします。この場合、複数の SQL Server インスタンスでデータが提供されるようにデータを分割すると、ソリューションをスケールアウトできます。データベース サーバーがボトルネックになることが予想される場合は、初期設計時にデータの分割も考慮すると、後で必要となる労力を大幅に節約できます。
- **疎結合されたレイヤー型の設計の定義**: 整っていてリモートで操作できるインターフェイスを備えた、疎結合されたレイヤー型の設計は、chatty な通信を行う密結合されたレイヤーを使用する設計よりも簡単にスケールアウトできます。レイヤー型の設計には元々クラッチ ポイントがあるので、レイヤー

の境界でのスケールアウトに最適です。ここで重要なのは、適切な境界を見つけることです。たとえば、ビジネス ロジックの配置は、負荷分散するように構成された中間ティアのアプリケーション サーバー ファームに簡単に変更できます。

設計の影響とトレードオフ

アプリケーション レイヤー、ティア、またはデータの種類によって異なる可能性がある、スケーラビリティのさまざまな側面を考慮する必要があります。必要なトレードオフを特定して、柔軟性のある側面とない側面を把握するようにします。場合によっては、Web サーバーやアプリケーション サーバーをスケールアップしてからスケールアウトするのは、最適な手法ではないことがあります。たとえば、8 基のプロセッサが搭載されたサーバーを使用できたとしても、おそらく経済的な事情から大規模な 1 台のサーバーではなく、小規模な複数台のサーバーを使用するでしょう。

一方、データの役割やデータの使用方法によっては、スケールアップしてからスケールアウトするのがデータベース サーバーにとって最適な手法になることがあります。負荷分散またはフェールオーバーを実行できるサーバーの台数には制限があり、データベースの分割方法など他の問題によって処理に影響が及びます。また、技術とパフォーマンスに関する考慮事項以外に、運用と管理に関する影響や関連する総保有コストについても考慮する必要があります。一般的に、その他すべての制約の境界内でコストとパフォーマンスを最適化します。たとえば、4 基のプロセッサを搭載した 2 台のサーバーを使用した場合と比較してコストとパフォーマンスを評価すると、2 基のプロセッサを搭載した Web サーバーまたはアプリケーション サーバーを 4 台使用する方が適していることがあります。ただし、特定の負荷分散インフラストラクチャに配置できるサーバーの最大数、データ センターの電力消費や床面積の制約など、他の制約も考慮する必要があります。

また、サーバー ファームを実装してサービスをホストするには、仮想サーバーの使用も検討します。この手法を使用すると、リソース使用率や投資収益率を最大限に高められるだけでなく、パフォーマンスとコストのバランスを取ることもできます。

ステートレスなコンポーネント

インプロセスの状態が保持されない Web フロントエンドに実装するコンポーネントなど、ステートレスなコンポーネントを使用すると、スケールアップとスケールアウトの両方をより適切にサポートする設計を作成できます。ステートレスな設計を実現するには、アプリケーションでは、多数の設計上のトレードオフが必要になる可能性が高くなりますが、一般にスケーラビリティに関しては、このデメリットを上回るメリットがあります。

データとデータベースの分割

アプリケーションで非常に大きなデータベースを使用していて、I/O がボトルネックとなることが予想される場合は、事前にデータベースの分割を設計するようにします。一般的に、後で分割データベースに移行すると、コストがかかる大量の手直しが必要となり、多くの場合はデータベースの完全な再設計が必要になります。分割には、いくつかのメリットがあります。たとえば、1 つのパーティションにクエリを制限したり (その結果、リソース使用量がわずかなデータに限定される)、複数のパーティションを使用したり (その結果、より多くのディスクでデータを取得できるので、並列処理が強化されてパフォーマンスが向上する) することができます。
ただし、場合によっては、複数のパーティションの使用が適さず、マイナスの結果を招くことがあります。たとえば、複数のディスクを使用する操作には、一元化されたデータを使用した方が効率よく実行できるものがあります。
配置シナリオでデータ ストレージの分割による影響を考慮する際の決定事項は、主にデータの種類によって変わります。関連する要因には、次のようなものがあります。

- **静的で参照用の読み取り専用データ**: この種類のデータの場合、パフォーマンスとスケーラビリティが向上するのであれば、簡単に多数のレプリカを適切な場所で保持できます。このような操作が設計に及ぼす影響はごくわずかで、通常は最適化に関する考慮事項により実行されます。1 台のデータベース サーバーに、論理的に分離して独立したいくつかのデータベースを統合することは、ディスク容量が十分でも、適切な場合と不適切な場合があります。また、このようなデータのコンシューマーの近くにレプリカを配置することも、同様に有効な手法になる場合があります。ただし、データを複製すると、適切な同期を維持するメカニズムが必要な、緩やかに同期されるシステムが必要になることに注意してください。
- **簡単に分割できる動的な (多くの場合は一時的な) データ**: これは、ショッピング カートなど特定のユーザーやセッションに関連するデータであり、ユーザー A のデータはユーザー B のデータとまったく関連がありません。このデータは、静的で読み取り専用のデータよりもやや扱いが難しくなりますが、この種類のデータも分割できるので、非常に簡単に最適化して分散できます。各ユーザーのレベルに至るまで、データのグループ間に依存関係はありません。このデータの重要な側面は、複数のパーティションにまたがってクエリしないことです。たとえば、クエリを実行してユーザー A のショッピング カートの内容を取得するのは問題ありませんが、特定の商品が含まれたすべてのカートについてのクエリは実行しないようにします。その後の要求が別の Web サーバーやアプリケーション サーバーに対して行われる可能性がある場合は、関連するパーティションに要求の送信先となるすべてのサーバーからアクセスできる必要があります。

- **コア データ**: 一般的にスケールアップしてからスケールアウトする手法が適用されるのは、主にこの場合です。データを同期するのが複雑になるので、通常はこの種類のデータを多数の場所で保持することはお勧めしません。これは、できる限りスケールアップする必要がある典型的な場合であり (理想的には、適切なクラスタリング機能を備えた単一の論理インスタンスのままにします)、スケールアウト以外に選択肢がないときにのみ分割と分散を検討します。分散パーティション ビューなどデータベース テクノロジの進歩により分割はずっと簡単になりましたが、分割は必要な場合にのみ使用するようにします。データベースのサイズが大きくなりすぎたことが原因で分割の使用を決定することはほとんどなく、データの所有者、データ使用の地理的分散、コンシューマーへの近さ、可用性など、他の考慮事項に基づいて決定される方が多いです。
- **後で同期されるデータ**: アプリケーションで使用されるデータには、すぐには同期する必要がないか、まったく同期する必要がないものもあります。このようなデータの好例は、"X を購入したユーザーは Y と Z も購入した" など、小売店のデータです。このデータはコア データから取得されますが、リアルタイムで更新する必要はありません。コア データを分割可能な (動的な) データにしてから静的なデータにする方針を設計することは、非常にスケーラビリティが高いアプリケーションの構築における重要な要素です。

データを移動して複製するパターンについては、「Data Movement Patterns」(http://msdn.microsoft.com/en-us/library/ms998449.aspx、英語) を参照してください。

ネットワーク インフラストラクチャのセキュリティに関する考慮事項

配置先の環境に用意されているネットワーク構造について理解し、フィルタリング規則、ポートの制限、サポートされるプロトコルなどに関するネットワークのベースライン セキュリティ要件を理解します。ネットワークのセキュリティを最大限に高めるための推奨事項は、次のとおりです。

- ファイアウォールとファイアウォールのポリシーがアプリケーションの設計と配置に影響を及ぼす可能性を特定します。ファイアウォールは、インターネットに公開されるアプリケーションを内部ネットワークから分離し、データベース サーバーを保護するために使用する必要があります。ファイアウォールを使用すると特別に構成されたポート経由の通信のみが許可されるので、一部のプロトコルをブロックしたり、通信オプションの一部を使用できないようにしたりすることができます。制限対象には、Windows 認証など、Web サーバーとファイアウォールの内側にあるアプリケーション サーバーまたはデータベース サーバーとの間の認証が含まれます。

- 境界ネットワークの Web サーバーから、またはリッチ クライアント アプリケーションから内部リソースにアクセスできるプロトコル、ポート、およびサービスを検討します。アプリケーションの設計に必要なプロトコルとポートを特定し、追加のポートを開いたり標準に準拠していないプロトコルを使用したりした場合に発生する潜在的な脅威を分析します。
- ネットワークとアプリケーション レイヤーのセキュリティ、および各コンポーネントで処理できるセキュリティ機能に関して立てられたすべての仮定を伝達して記録します。このようにすると、開発チームとネットワーク チームの両者が他方のチームが問題に対処しているであろうと見なすことが原因でセキュリティ コントロールやポリシーが見落とされることがなくなります。
- アプリケーションが使用するファイアウォール、パケット フィルター、ハードウェア システムなどのセキュリティ対策に注意し、これらの対策がネットワークで有効になっていることを確認します。
- ネットワーク構成の変更による影響と、それによってセキュリティが受ける影響について検討します。

管理容易性に関する考慮事項

アプリケーションの配置時に行う選択は、アプリケーションを管理および監視する機能に影響を及ぼします。次の推奨事項を考慮する必要があります。

- 複数のコンシューマーで使用されるアプリケーションのコンポーネントは、すべてのアプリケーションでアクセスできるサーバー ファームやアプリケーション ファームなど、単一の一元的な場所に配置して、重複しないようにします。
- バックアップと復元機能がアクセスできる場所にデータが格納されるようにします。
- 既存のソフトウェアやハードウェアに依存するコンポーネント (特定のコンピューターからしか接続できない専用ネットワークなど) は、物理的に同じコンピューターに配置する必要があります。
- 一部のライブラリやアダプターは、自由に配置するには追加のコストがかかったり、CPU 単位でコストがかかります。そのため、このような機能を一元化してコストを最低限に抑えることをお勧めします。
- 組織内のグループは、各グループがローカルで管理する必要があるサービス、コンポーネント、またはアプリケーションを所有していることがあります。
- System Center Operations Manager などの監視ツールでは、物理コンピューターにアクセスして管理情報を取得する必要があるので、配置オプションに影響を及ぼすことがあります。

関連する設計パターン

次の表に示すように、主要なパターンは、配置、管理容易性、パフォーマンスと信頼性、セキュリティなどのカテゴリに分類されます。各カテゴリの設計を行う際には、これらのパターンの使用を検討します。

カテゴリ	関連するパターン
配置	**Layered Application**: システムがレイヤーで構成されたアーキテクチャ パターンです。 **Three-Layered Services Application**: 他のアプリケーションで使用できるサービスを公開しながら、パフォーマンスを最大限に高めるようにレイヤーが設計されたアーキテクチャ パターンです。 **Tiered Distribution**: 物理的な境界を越えて設計のレイヤーを分散できるアーキテクチャ パターンです。 **Three-Tiered Distribution**: 3 つの物理ティアに設計のレイヤーが分散されるアーキテクチャ パターンです。 **Deployment Plan**: インフラストラクチャによって課される制約を考慮した、論理レイヤーを物理ティアに割り当てる手順を表します。
管理容易性	**Adapter**: 一般的なインターフェイスをサポートし、一般的なインターフェイスと、インターフェイスが異なる同様の機能を実装している他のオブジェクトとの間で操作を変換するオブジェクトです。 **Provider**: あらゆるカスタム実装がシームレスにプラグインされるようにするために、クライアント API とは異なる API が公開されるコンポーネントです。インストルメンテーションが用意されている多くのアプリケーションでは、アプリケーションの状態とパフォーマンスに関する情報、およびアプリケーションをホストしているシステムに関する情報の取得に使用できるプロバイダーが公開されます。
パフォーマンスと信頼性	**Server Clustering**: 複数のサーバーで負荷を共有し、クライアントでは単一のコンピューターまたはリソースとして認識されるように構成された分散パターンです。 **Load-Balanced Cluster**: 複数のサーバーが負荷を共有するように構成された分散パターンです。負荷分散を行うと、複数のサーバーに負荷を分散することでパフォーマンスが向上するだけでなく、1 台のサーバーで障害が発生しても残りのサーバーで引き続き負荷を処理できるので信頼性も向上します。 **Failover Cluster**: 非常に可用性の高いインフラストラクチャ ティアを提供することで、1 台のサーバーやそのサーバーでホストされているソフトウェアで発生し

	た障害によるサービスの停止を防ぐ分散パターンです。
セキュリティ	**Brokered Authentication**: ブローカーに対して認証します。ブローカーでは、サービスやシステムにアクセスする際の認証に使用するトークンが提供されます。 **Direct Authentication**: アクセスするサービスやシステムに対して直接認証します。 **Impersonation and Delegation**: 一時的に異なる ID を偽装して、異なるセキュリティ コンテキストや資格情報を使用してリソースにアクセスできるようにする処理です。また、別のユーザーの代わりにサービスのアカウントでリモート リソースにアクセスできるようにします。 **Trusted Subsystem**: アプリケーションが信頼されたサブシステムとして機能して、追加のリソースにアクセスします。リソースにアクセスする際には、ユーザーの資格情報ではなくアプリケーション自体の資格情報を使用します。

Layered Application、Three-Layered Services Application、Tiered Distribution、Three-Tiered Distribution、および Deployment Plan の各パターンの詳細については、「配置のパターン」(http://msdn.microsoft.com/ja-jp/library/ms998478.aspx) を参照してください。

Adapter パターンの詳細については、Erich Gamma、Richard Helm、Ralph Johnson、John Vlissides 共著『オブジェクト指向における再利用のためのデザイン パターン』(ソフトバンク クリエイティブ、1999 年) の第 4 章「構造に関するパターン」を参照してください。

Provider パターンの詳細については、「Provider Model Design Pattern and Specification, Part 1」(http://msdn.microsoft.com/en-us/library/ms972319.aspx、英語) を参照してください。

Server Clustering パターン、Load-Balanced Cluster パターン、および Failover Cluster パターンの詳細については、「パフォーマンスと信頼性のパターン」(http://msdn.microsoft.com/ja-jp/library/ms998503.aspx) を参照してください。

Brokered Authentication、Direct Authentication、Impersonation and Delegation、および Trusted Subsystem の各パターンの詳細については、「Web Service Security」(http://msdn.microsoft.com/en-us/library/aa480545.aspx、英語) を参照してください。

関連情報

配置に関する方針の設計に役立つ Web リソースに、より簡単にアクセスするには、
http://www.microsoft.com/architectureguide (英語) でオンライン版の参考文献を参照してください。

- 承認の技法の詳細については、「Designing Application-Managed Authorization」(http://msdn.microsoft.com/en-us/library/ee817656.aspx、英語) を参照してください。
- 配置シナリオと考慮事項の詳細については、「Deploying .NET Framework-based Applications」(http://msdn.microsoft.com/en-us/library/ee817655.aspx、英語) を参照してください。
- 設計パターンの詳細については、「Microsoft .NET を使用したエンタープライズ ソリューション パターン」(http://msdn.microsoft.com/ja-jp/library/ms998469.aspx) を参照してください。

20

アプリケーションの種類の選択

概要

この章では、このガイドで紹介するアプリケーションの種類、必要なトレードオフ、およびアプリケーションを選択することによる設計への影響について理解するのに役立つ情報を提供します。この章を読むと、シナリオと要件に適したアプリケーションの種類を決定できるようになります。この章では、5 つの基本的なアプリケーションの原型について簡単に説明し、詳細情報を提供している他の章へのリンクを提示します。

要件、テクノロジの制約、および提供する予定のユーザー エクスペリエンスの種類によって、選択するアプリケーションの種類が決まります。たとえば、サービスの提供先となるクライアントで常時ネットワーク接続を利用できるようにするかどうか、匿名ユーザーがリッチ メディア コンテンツを Web ブラウザーで表示できるようにするかどうか、または主に組織のイントラネット上で少数のユーザーにサービスを提供するかどうかについて判断する必要があります。

次のセクション「アプリケーションの原型の概要」では、アプリケーションの種類、その説明、および一般的なシナリオを確認できます。このセクションの表は、各アプリケーションの種類のメリットと考慮事項に基づいて、十分な情報を得たうえでアプリケーションの種類を選択するのに役立ちます。

アプリケーションの原型の概要

構築するアプリケーションの一般的で基本的な種類は、次のとおりです。

- **モバイル アプリケーション**: シン クライアント アプリケーションまたはリッチ クライアント アプリケーションとして開発できます。リッチ クライアント モバイル アプリケーションでは、非接続型のシナリオや不定期に接続するシナリオをサポートできます。Web またはシン クライアント アプリケーションでは、

接続型のシナリオだけをサポートします。モバイル アプリケーションを設計する際には、デバイスのリソースが制約事項になることがあります。

- **リッチ クライアント アプリケーション**: 通常、さまざまなコントロールを使用してデータを表示するグラフィカル ユーザー インターフェイスを備えたスタンドアロン アプリケーションとして開発されます。リモートのデータや機能にアクセスする必要があれば、リッチ クライアント アプリケーションを非接続型のシナリオや不定期に接続するシナリオ向けに設計できます。
- **リッチ インターネット アプリケーション**: 複数のプラットフォームと複数のブラウザーをサポートし、リッチ メディア コンテンツやグラフィカル コンテンツを表示するように開発できます。リッチ インターネット アプリケーションは、クライアントの一部の機能へのアクセスを制限するブラウザーのサンドボックス内で実行されます。
- **サービス アプリケーション**: 共有のビジネス機能が公開され、クライアントがローカル システムやリモート システムからその機能にアクセスできるようになります。サービスの操作は、XML スキーマに基づく、トランスポート チャネル経由で渡されるメッセージを使用して呼び出されます。この種類のアプリケーションの目的は、クライアントとサーバーの間の疎結合を実現することです。
- **Web アプリケーション**: 一般に接続型のシナリオをサポートし、さまざまなオペレーティング システムやプラットフォームで実行している多種多様なブラウザーをサポートできます。

他にも多数の特殊な種類のアプリケーションを設計および構築できます。概して、このような種類のアプリケーションは、上記一覧で説明している基本的な種類を特殊化したり組み合わせたりしたものです。

アプリケーションの種類に関する考慮事項

次の表に、一般的なアプリケーションの原型に関するメリットと考慮事項を示します。

アプリケーションの種類	メリット	考慮事項
モバイル アプリケーション	ハンドヘルド デバイスのサポート 外出中のユーザーにとっての可用性と使いやすさ オフライン シナリオや不定期に接続するシナリオのサポート	入力とナビゲーションが制限されている。 画面の表示領域が制限されている。
リッチ クライアント	クライアント リソースを活用する機	配置が複雑 (ただし、ClickOnce、

アプリケーション	能 応答性の向上、機能豊富な UI 機能、およびユーザー エクスペリエンスの強化 非常に動的で応答性の高い操作 オフライン シナリオや不定期に接続するシナリオのサポート	Windows インストーラー、XCOPY など、さまざまなインストール オプションを使用できる)。 時間の経過と共にバージョン管理が難しくなる。 プラットフォームに固有。
リッチ インターネット アプリケーション (RIA)	リッチ クライアント アプリケーションと同じ機能豊富なユーザー インターフェイス リッチ メディア、ストリーミング メディア、およびグラフィカルな表示のサポート Web クライアントと同じ配布機能 (対象範囲) による簡単な配置 簡単なアップグレードとバージョン更新 さまざまなプラットフォームやブラウザーのサポート	Web アプリケーションに比べてアプリケーションのリソース使用量が多い。 リッチ クライアント アプリケーションに比べてクライアント リソースを使用する際の制約が厳しい。 クライアントに適切なランタイム フレームワークを配置する必要がある。
サービス アプリケーション	クライアントとサーバーの間の疎結合された対話 さまざまな無関係のアプリケーションで使用できる 相互運用性のサポート	UI がサポートされない。 ネットワーク接続に依存している。
Web アプリケーション	幅広い対象範囲と複数のプラットフォーム間での標準に準拠した UI 配置と変更管理の容易性	常時ネットワーク接続に依存している。 機能豊富なユーザー インターフェイスの提供が難しい。

それぞれのアプリケーションの種類は、1 つ以上のテクノロジを使用して実装できます。シナリオとテクノロジの制約、および開発チームの能力と経験によって、選択するテクノロジが決定します。

次のセクションでは、各アプリケーションの種類について詳しく説明します。

- モバイル アプリケーションの原型
- リッチ クライアント アプリケーションの原型
- リッチ インターネット アプリケーションの原型
- サービス アプリケーションの原型
- Web アプリケーションの原型

このガイドでは、いくつかの特殊なアプリケーションの種類についても詳細を説明しています。詳細については、次の章を参照してください。

- 第 26 章「ホストされているクラウド サービス」
- 第 27 章「Office Business Application の設計」
- 第 28 章「SharePoint LOB アプリケーションの設計」

モバイル アプリケーションの原型

モバイル アプリケーションは、通常、ユーザー エクスペリエンス (プレゼンテーション) レイヤー、ビジネス レイヤー、およびデータ レイヤーで構成されたマルチレイヤー型のアプリケーションとして構築されています (図 1 参照)。

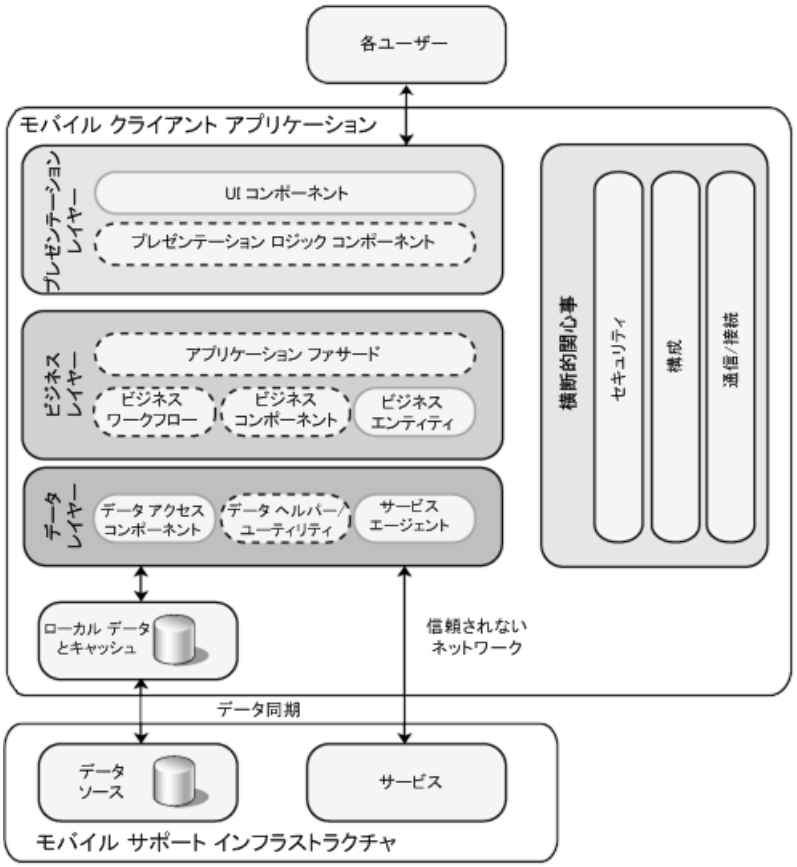


図 28

モバイル アプリケーションの一般的な構造

モバイル アプリケーションを開発する場合は、Web ベースのシン クライアント、またはリッチ クライアントを開発できます。リッチ クライアントを構築すると、ビジネス レイヤーとデータ レイヤーがデバイス自体に配置される可能性が高くなります。シン クライアントでは、ビジネス レイヤーとデータ レイヤーはサーバーに配置されます。モバイル アプリケーションでは、多くの場合、ローカルにキャッシュされたデータを使用して、オフラインまたは非接続型の操作をサポートし、接続時にこのデータを同期します。モバイルアプリケーションでは、S+S のホストされているサービス、Web サービスなど、他のアプリケーションで公開されたサービスも使用できます。データ ソースの同期などを行うサービスは、多くの場合、特定のサーバー ベースのインフラストラクチャを使用して制御された方法でモバイル クライアント アプリケーションに公開されます。

次のような場合は、モバイル アプリケーションの使用を検討します。

- ユーザーがハンドヘルド デバイスに依存している
- 小さい画面で使用するのに適した単純な UI がアプリケーションでサポートされている

- オフライン シナリオや不定期に接続するシナリオをアプリケーションでサポートする必要がある (この場合、通常はモバイル リッチ クライアント アプリケーションが最適です)
- アプリケーションがデバイスから独立している必要があり、ネットワーク接続を確保できる (この場合、通常はモバイル Web アプリケーションが最適です)

モバイル アプリケーションの設計方法については、第 24 章「モバイル アプリケーションの設計」を参照してください。

リッチ クライアント アプリケーションの原型

リッチ クライアントのユーザー インターフェイスでは、スタンドアロンのシナリオ、接続型のシナリオ、不定期に接続するシナリオ、および非接続型のシナリオで動作する必要があるアプリケーションで、非常に応答性が高く対話型の機能豊富なユーザー エクスペリエンスを提供できます。リッチ クライアント アプリケーションは、通常、ユーザー エクスペリエンス (プレゼンテーション) レイヤー、ビジネス レイヤー、およびデータ レイヤーで構成されたマルチレイヤー型のアプリケーションとして構築されています (図 2 参照)。

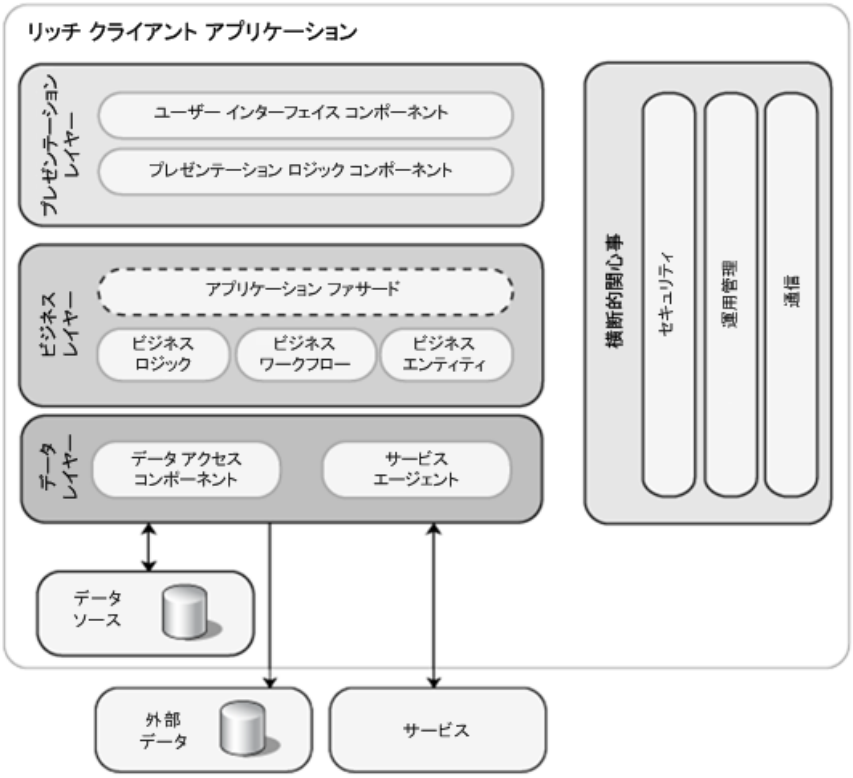


図 29

リッチ クライアント アプリケーションの一般的な構造

リッチ クライアント アプリケーションでは、リモート サーバーに格納されたデータ、ローカルに格納されたデータ、またはその両方の組み合わせを使用できます。また、S+S のホストされているサービス、Web サービスなど、他のアプリケーションで公開されたサービスも使用できます。
次のような場合は、リッチ クライアント アプリケーションの使用を検討します。

- 非接続型のシナリオや不定期に接続するシナリオをアプリケーションでサポートする必要がある
- アプリケーションがクライアント PC 上に配置される
- アプリケーションが高度な対話型で高い応答性を備えている必要がある
- アプリケーションの UI で豊富な機能とユーザー操作を提供する必要があるが、RIA の高度なグラフィック機能やメディア機能は必要ない
- アプリケーションでクライアント PC のリソースを使用する必要がある

リッチ クライアント アプリケーションの設計方法については、第 22 章「リッチ クライアント アプリケーションの設計」を参照してください。

リッチ インターネット アプリケーションの原型

リッチ インターネット アプリケーション (RIA) は、サンドボックス内のブラウザーで実行されます。従来の Web アプリケーションと比較した RIA のメリットは、より優れたユーザー エクスペリエンスが提供され、ユーザーの応答性とネットワーク効率が向上することです。RIA は、通常、ユーザー エクスペリエンス (プレゼンテーション) レイヤー、サービス レイヤー、ビジネス レイヤー、およびデータ レイヤーで構成されたマルチレイヤー型のアプリケーションとして構築されています (図 3 参照)。

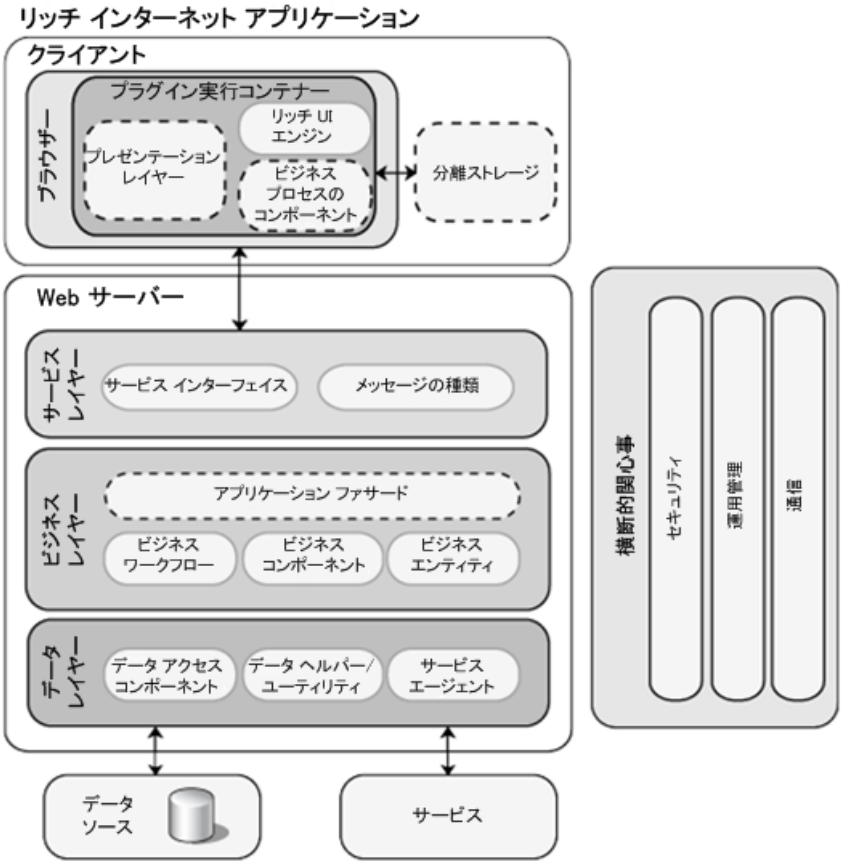


図 30

リッチ インターネット アプリケーションの一般的な構造

通常、RIA では、クライアント側のプラグインやホストされた実行環境 (XAML ランタイム、Silverlight など) が使用されます。このプラグインでは、クライアントのプラグインや実行環境で使用されるコードとデータを生成するリモート Web サーバー ホストと通信します。

次のような場合は、リッチ インターネット アプリケーションの使用を検討します。

- アプリケーションでリッチ メディアをサポートする必要があり、多数のグラフィックスを表示する必要がある
- Web アプリケーションよりも機能豊富で対話型の応答性の高い UI を提供する必要がある
- アプリケーションで、制限された方法でクライアント側の処理を利用する
- アプリケーションで、制限された方法でクライアント側のリソースを使用する
- Web ベースの配置モデルを簡略化する必要がある

リッチ インターネット アプリケーションの設計方法については、第 23 章「リッチ インターネット アプリケーションの設計」を参照してください。

サービス アプリケーションの原型

このガイドでは、機能の単位へのアクセスを提供するパブリック インターフェイスのことを "サービス" と言います。サービスでは、名前のとおりプログラムによるなんらかの "サービス" が、そのサービスを使用する呼び出し元に提供されます。このようなサービスを公開するサービス アプリケーションは、通常、サービス レイヤー、ビジネス レイヤー、およびデータ レイヤーで構成されたマルチレイヤー型のアプリケーションとして構築されています (図 4 参照)。

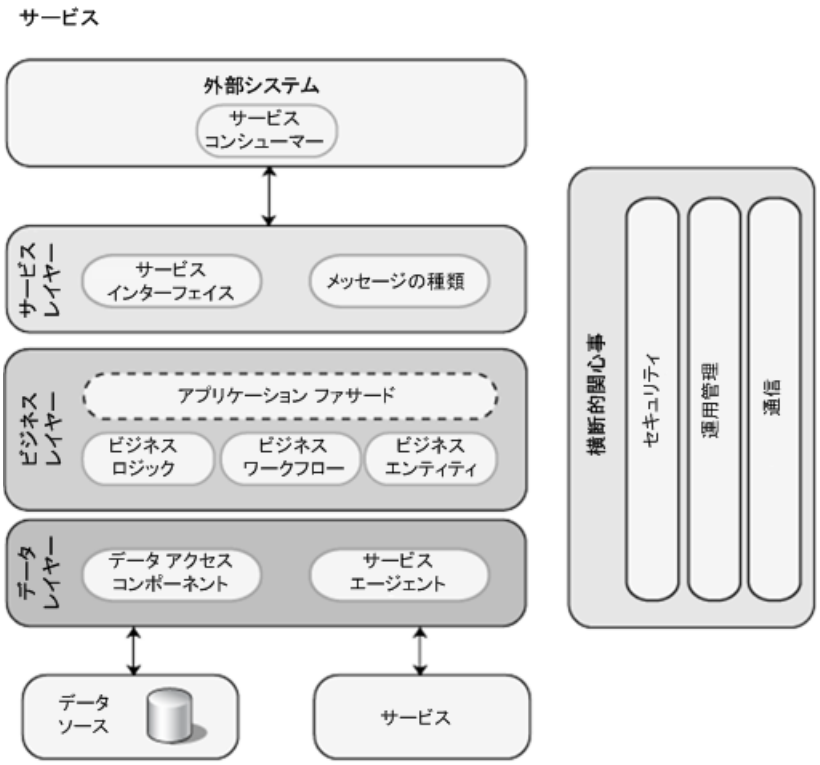


図 31

サービス アプリケーションの一般的な構造

サービスは疎結合されており、組み合わせることでより複雑な機能を提供できます。サービスは分散可能で、リモート コンピューターからも、サービスが実行されているコンピューターからもアクセスできます。また、サービスはメッセージ指向です。つまり、インターフェイスは Web サービス記述言語 (WSDL) ドキュメントで定義され、操作はトランスポート チャネル経由で渡される XML スキーマに基づくメッセージを使用して呼び出されます。また、

メッセージとインターフェイスの定義に重点を置いて相互運用性が確保されているので、サービスでは異種環境がサポートされます。コンポーネントでメッセージとインターフェイスの定義が認識されれば、コンポーネントが基づいているテクノロジに関係なくサービスを使用できます。

次のような場合は、サービス アプリケーションの使用を検討します。

- アプリケーションで UI を必要としない機能を公開する
- アプリケーションがクライアントと疎結合されている必要がある
- 他の外部のアプリケーションでアプリケーションを共有または使用する必要がある
- インターネット経由、イントラネット経由、またはローカル コンピューター上で他のアプリケーションによって使用される機能をアプリケーションで公開する必要がある

サービスとサービス アプリケーションの設計方法については、第 25 章「サービス アプリケーションの定義」を参照してください。

Web アプリケーションの原型

Web アプリケーションの中核は、そのサーバー側のロジックです。このロジックは、多数の別個のレイヤーで構成されています。一般的な例は、プレゼンテーション レイヤー、ビジネス レイヤー、およびデータ レイヤーで構成された 3 つのレイヤーによるアーキテクチャです (図 5 参照)。

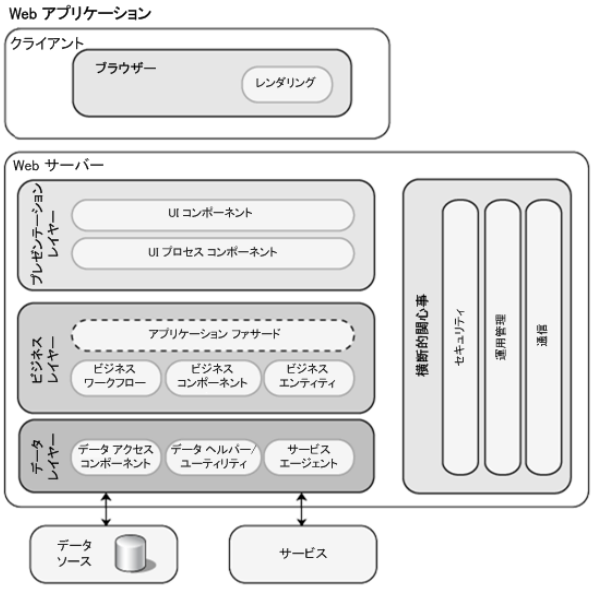


図 32

Web アプリケーションの一般的な構造

通常、Web アプリケーションでは、リモート データベース サーバーに格納されたデータにアクセスします。また、S+S のホストされているサービス、Web サービスなど、他のアプリケーションで公開されたサービスも使用できます。

次のような場合は、Web アプリケーションの使用を検討します。

- アプリケーションで、リッチ インターネット アプリケーションで提供される機能豊富な UI やメディアのサポートが必要ない
- Web ベースの配置モデルを簡略化する必要がある
- ユーザー インターフェイスがプラットフォームから独立している必要がある
- アプリケーションにインターネット経由でアクセスできる必要がある
- クライアント側の依存関係、およびリソースの使用量 (ディスクやプロセッサの使用量など) を最小限に抑える必要がある

Web アプリケーションの設計方法については、第 21 章「Web アプリケーションの設計」を参照してください。

21

Web アプリケーションの設計

概要

この章では、Web アプリケーションの設計に関する一般的な考慮事項と主要な特性について紹介します。具体的には、レイヤー型構造のガイドライン、パフォーマンス、セキュリティ、および展開のガイドライン、主要なパターンとテクノロジに関する考慮事項などについて説明します。

Web アプリケーションは、ユーザーが Web ブラウザーや専用のユーザー エージェントを使用してアクセスできるアプリケーションです。ブラウザーでは、Web サーバー上のリソースに対応付けられている特定の URL への HTTP 要求を作成します。サーバーでは、ブラウザーで表示できる HTML ページをレンダリングし、そのページをクライアントに返します。Web アプリケーションの中核は、そのサーバー側のロジックです。アプリケーションにはいくつかの別個のレイヤーが含まれます。一般的な例は、プレゼンテーション レイヤー、ビジネス レイヤー、およびデータ レイヤーで構成される 3 つのレイヤーによるアーキテクチャです。図 1 に、一般的なコンポーネントを関連領域でグループ化した、典型的な Web アプリケーションのアーキテクチャの例を示します。

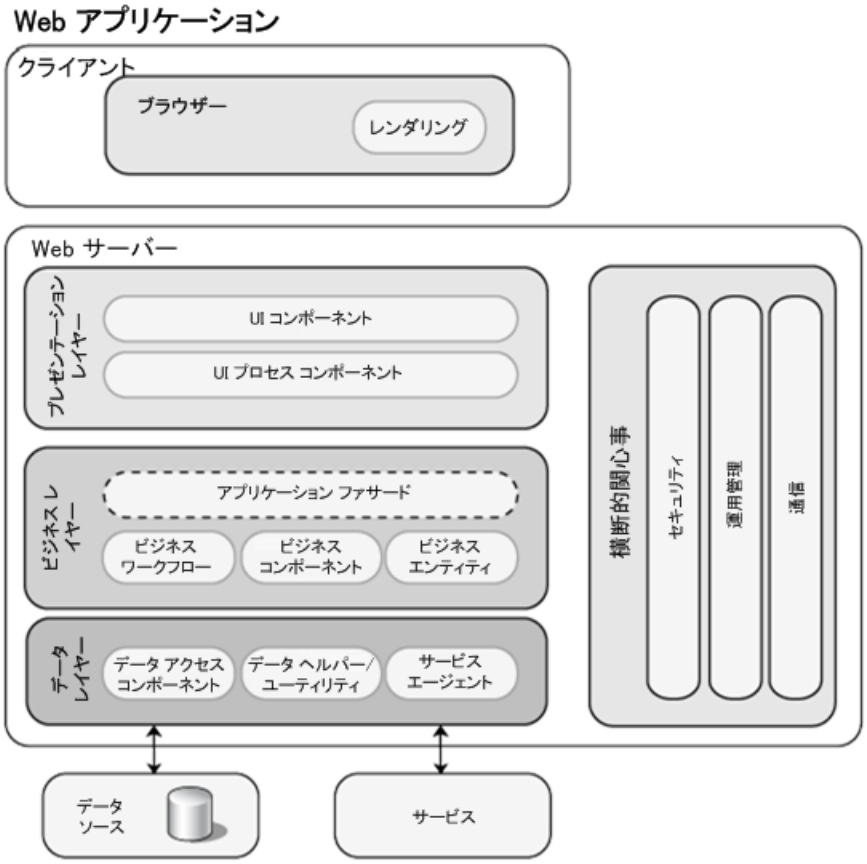


図 33

Web アプリケーションの一般的な構造

通常、プレゼンテーション レイヤーには UI コンポーネントとプレゼンテーション ロジック コンポーネントが、ビジネス レイヤーにはビジネス ロジック、ビジネス ワークフロー、およびビジネス エンティティ (オプションでファサード) が、データ レイヤーにはデータ アクセス コンポーネントとサービス エージェント コンポーネントが含まれます。レイヤー型の設計の詳細については、第 5 章「レイヤー型アプリケーションのガイドライン」を参照してください。各レイヤーで使用されるコンポーネントの詳細については、第 10 章「コンポーネントのガイドライン」を参照してください。

設計に関する一般的な考慮事項

Web アプリケーションを設計する際、ソフトウェア アーキテクトの目標は、安全で高いパフォーマンスを発揮するアプリケーションを設計しながら、タスクを関心領域に分離して、アプリケーションがなるべく複雑にならないようにすることです。アプリケーションが要件を満たし、Web アプリケーションで一般的なシナリオで効率的に機能するためには、次のガイドラインに従います。

- **アプリケーションを論理的に分割する**: レイヤー化を使用して、アプリケーションをプレゼンテーション レイヤー、ビジネス レイヤー、およびデータ アクセス レイヤーに論理的に分割します。このようにすると、保守容易性の高いコードを作成して、各レイヤーのパフォーマンスを別個に監視および最適化できます。また、明確かつ論理的に分割することで、アプリケーションを拡張する際の選択肢が増えます。
- **抽象化を使用してレイヤー間の疎結合を実装する**: これは、インターフェイス コンポーネント (レイヤーのコンポーネントで理解できる形式に要求を変換する既知の入出力の機能を持つファサードなど) を定義することで実現できます。また、インターフェイス型や抽象型の基本クラスを使用して、インターフェイス コンポーネントで実装する必要がある共通の抽象化を定義することもできます。
- **コンポーネント間の通信方法を理解する**: これには、アプリケーションでサポートしなければならない配置シナリオを理解する必要があります。物理境界間やプロセス境界間の通信がサポートされる必要があるか、またはすべてのコンポーネントが同じプロセスで実行されるかどうかを判断することが必要です。
- **キャッシュを使用してサーバーへのラウンド トリップを最小限に抑えることを検討する**: Web アプリケーションを設計する際には、キャッシュや出力バッファーなどの技法を使用して、ブラウザーと Web サーバー間、Web サーバーとダウンストリーム サーバー間のラウンド トリップを減らすことを検討します。適切に設計されたキャッシュ方針は、パフォーマンスに関連する設計における最も重要な考慮事項です。ASP.NET のキャッシュ機能には、出力キャッシュ、部分的なページ キャッシュ、およびキャッシュ API があります。これらの機能を活用するようにアプリケーションを設計します。
- **ログ記録とインストルメンテーションの使用を検討する**: アプリケーションのレイヤーやティアをまたいで操作を監査して、ログに記録する必要があります。このログ記録は、システムへの攻撃の初期兆候を示すことが多いので、疑わしい操作を検出するのに役立ちます。ただし、ブラウザーで実行しているスクリプト コードで発生する問題をログに記録することは難しい場合があるので注意してください。
- **信頼境界を越えるときにユーザーを認証することを検討する**: ユーザーが信頼境界を越えるときに (たとえば、プレゼンテーション レイヤーからリモート ビジネス レイヤーにアクセスするときに) ユーザーを認証するようにアプリケーションを設計する必要があります。
- **プレーンテキストの機密データをネットワーク経由で送信しないようにする**: パスワードや認証 Cookie などの機密データをネットワーク経由で送信しなければならない場合は、データの暗号化と署名を使用したり、Secure Sockets Layer (SSL) 暗号化を使用することを検討します。

- **最小限の特権を持つアカウントを使用して Web アプリケーションを実行するように設計する**: 攻撃者がプロセスを制御してしまった場合に備えて、プロセス ID によるファイル システムやその他のシステム リソースへのアクセスを制限して、損害が発生する可能性を制限する必要があります。

設計に関する一般的な考慮事項の詳細については、第 17 章「横断的関心事」を参照してください。

設計に関する具体的な問題

設計を策定する際に考慮する必要がある一般的な問題がいくつかあります。これらの問題は、設計の具体的な領域に分類できます。次のセクションでは、各領域の一般的な問題を回避するのに役立つガイドラインを示します。

- アプリケーションの要求処理
- 認証
- 承認
- キャッシュ
- 例外管理
- ログ記録とインストルメンテーション
- ナビゲーション
- ページ レイアウト
- ページ レンダリング
- セッション管理
- 検証

アプリケーションの要求処理

大まかに言うと、Web アプリケーションでは要求処理を 2 つの方法で実行できます。ポスト バックの手法では、Web フォームのポスト バックを使用して、ブラウザーが主にサーバーと通信します。一般的な代替手法は、ブラウザーとサーバー間で RESTful サービスの呼び出しを使用することです。これらの 2 つの手法にはそれぞれ長所と短所があるので、選択した手法は、この章の後半で説明する設計の問題の解決方法に影響を及ぼすことがあります。要求処理の方針を選択する際には、アプリケーションの UI をどの程度制御する必要があるのか、開発とテストの方法、およびパフォーマンスと拡張の要件について考慮する必要があります。

一般的に、ポスト バックの手法では、フォーム ベースの開発エクスペリエンスを実現することが可能で、対応する HTML、関連するビュー ステート、および相互作用ロジックをブラウザーにレンダリングする、サーバー側の豊富なコントロールを使用します。フォーム ベースの Web アプリケーションを開発しており、すばやいアプリケーション開発 (RAD) エクスペリエンスを実現する必要がある場合は、この手法の使用を検討します。

一般的に、RESTful の手法では、アプリケーションの UI をより細かく制御することが可能で、ナビゲーション、テスト容易性、および関心を分離する際の柔軟性が高まります。アプリケーションで柔軟なナビゲーションや UI の細かい制御が必要で、代替の UI レンダリング テクノロジを使用する可能性がある場合、またはテスト駆動開発の手法を使用する場合は、この手法の使用を検討します。

選択する要求処理の方針にかかわらず、要求処理ロジックとアプリケーション ロジックを UI から切り離して実装することで、関心領域を分離する必要があります。これを実現するのに役立つパターンがいくつかあります。一般的に、モデル ビュー プレゼンター(MVP:Model-View-Presenter) パターンやこれと同様のパターンを Web フォームのポスト バックの手法で使用すると、関心領域を明確に分離できます。モデル ビュー プレゼンター (MVP) パターンは、通常、RESTful の要求処理の手法で使用します。

また、要求処理の方針を設計する際には、次のガイドラインを検討します。

- Web ページ要求の一般的な処理手順と処理後の手順を一元化して、ページ間でのロジックの再利用を促進することを検討します。たとえば、HTTP モジュールや ASP.NET の Page クラスから派生する基本クラスを作成して、一般的な処理と処理後のロジックを含めることを検討します。
- UI 処理に適した手法またはパターンを選択します。MVC、MVP、または同様のパターンを使用して、UI 処理を Model、View、および Controller/Presenter の 3 つの異なる役割に分割することを検討します。コンポーネントで処理ロジックとレンダリング ロジックが混在しないようにします。
- 大量のデータを処理する View を設計する場合には、MVP パターンの 1 つの形式である監視プレゼンター (もしくは監視コントローラー) パターンを使用して、View がモデルにアクセスできるようにすることを検討します。アプリケーションがビュー ステートに依存しておらず、管理イベントの数が限られている場合は、MVC パターンの使用を検討します。
- 受信フィルタパターンを使用し、必要に応じて、処理手順をプラグ可能なフィルターとして実装することを検討します。
- ネットワーク経由 (特にインターネット経由) で送信されるすべての機密データを保護するようにします。SSL などのセキュリティで保護されたチャネル プロトコルを使用して、内部ネットワークと外部ネットワーク経由で送信される機密性の高いすべてのデータを暗号化してデジタル署名することを検討します。

認証

効果的な認証方針を設計することは、アプリケーションのセキュリティと信頼性の両方において重要です。認証が不適切または不十分だと、スプーフィング攻撃、辞書攻撃、セッション ハイジャックなどの攻撃に対して、アプリケーションが脆弱になる可能性があります。認証の方針を設計する際には、次のガイドラインを考慮します。

- Web アプリケーションのレイヤー内の信頼境界を特定します。信頼境界を特定すると、ユーザーの認証場所を決定するのに役立ちます。
- アカウント ロックアウト、パスワードの有効期限、およびパスワードの最低限の長さや複雑性を指定する強力なパスワード ポリシーなど、安全性の高いアカウント管理手法を使用します。
- 可能であれば、Windows 認証など、プラットフォームでサポートされている認証メカニズムを使用します。フォーム認証を使用する場合は、カスタム認証のメカニズムを設計するのではなく、ASP.NET の組み込みのサポートを使用します。ユーザーが同じ資格情報で複数のサイトにログオンできるようにする必要がある場合は、統合されたサービスやシングル サインオン (SSO) の使用を検討します。
- データベースにパスワードを格納する必要がある場合は、プレーンテキストで格納するのではなく、パスワードのハッシュ (または salt 処理されたハッシュ) を格納します。

承認

承認は、認証済み ID が実行できるタスクを特定したり、アクセスできるリソースを識別したりします。効果的な承認方針を設計することは、アプリケーションのセキュリティと信頼性の両方において重要です。承認が不適切または不十分だと、情報漏えい、データの改ざん、および特権の昇格が発生します。アプリケーションの承認方針における重要なセキュリティ原理は、徹底的に防衛することです。承認の方針を設計する際には、次のガイドラインを考慮します。

- 信頼境界を越える際には、必ずユーザーを承認します。ページとディレクトリへのアクセスを制御するには URL 認証を使用します。第 19 章「物理ティアと配置」で説明している信頼されたサブシステム モデルに基づく信頼された ID を使用して、ダウンストリーム リソースにアクセスします。
- 承認の設定の粒度を検討します。粒度が大きすぎると管理のオーバーヘッドが増加しますが、粒度が小さいと柔軟性が低下することがあります。
- 偽装とデリゲートを使用すると、ユーザーごとの監査やプラットフォームの細かいアクセス制御機能を使用できますが、パフォーマンスとスケーラビリティへの影響を考慮する必要があります。

キャッシュ

キャッシュを使用すると、アプリケーションのパフォーマンスと応答性が向上します。ただし、キャッシュの選択やキャッシュの設計が不適切だと、パフォーマンスと応答性が低下することがあります。キャッシュは、データ検索処理の最適化、ネットワーク ラウンド トリップの回避、および不要に重複する処理の回避に使用します。キャッシュを実装するには、まず、データをキャッシュに読み込むタイミングを決定する必要があります。クライアント側で遅延が発生するのを回避するには、キャッシュ データを非同期で読み込むか、バッチ処理を使用します。キャッシュを設計する際には、次のガイドラインを考慮します。

- 可能であれば、すぐに使用できる形式でデータをキャッシュして、定期的に変更される揮発性データはキャッシュしないようにします。機密情報をキャッシュする場合は、必ず暗号化します。
- 比較的静的なページのキャッシュには、出力キャッシュを使用します。出力キャッシュを使用すると、送信された値に基づいて複数の選択肢をサポートしながら、パフォーマンスが大幅に向上します。ページの一部のみが比較的静的である場合は、部分的なページ キャッシュとユーザー コントロールを使用することを検討します。
- ネットワーク接続などのコストのかかる共有リソースは、キャッシュではなくプールします。

キャッシュの詳細については、第 17 章「横断的関心事」を参照してください。

例外管理

効果的な例外管理方針を設計することは、アプリケーションのセキュリティと信頼性の両方において重要です。Web ページで適切な例外処理を行うと、機密性の高い例外の詳細がユーザーに開示されるのを防ぎ、アプリケーションの堅牢性を向上し、エラーが発生した場合にアプリケーションが一貫性のない状態になることを回避するのに役立ちます。例外管理の方針を設計する際には、次のガイドラインを考慮します。

- ユーザーには、わかりやすいエラー メッセージを提供して、アプリケーションで発生したエラーを通知します。ただし、エラー ページ、エラー メッセージ、ログ ファイル、および監査ファイルでは機密データを公開しないようにします。
- ハンドルされない例外をキャッチして、例外が発生したときにリソースと状態がクリーンアップされるようにします。グローバルなエラー ページやエラー メッセージを表示するグローバル例外ハンドラーを、すべてのハンドルされない例外用に設計します。必要でない場合は、カスタムの例外を使用しないようにします。

- 例外をハンドルしなければならない場合 (たとえば、機密情報を削除したり、例外に情報を追加したりする場合) を除き、例外はキャッチしないようにします。アプリケーション ロジックのフローを制御するために、例外を使用しないでください。

例外管理の詳細については、第 17 章「横断的関心事」を参照してください。

ログ記録とインストルメンテーション

効果的なログ記録とインストルメンテーションの方針を設計することは、アプリケーションのセキュリティと信頼性の両方において重要です。アプリケーションのティアをまたいで操作を監査して、ログに記録する必要があります。このログ記録は、システムへの攻撃の初期兆候を示すことが多いので、疑わしい操作を検出するのに使用でき、(悪意のある操作を実行したことをハッカーが否定できる) 否認の脅威に対処するのに役立ちます。個々の不正行為を証明するには、法的措置をとる際に、ログ記録ファイルと監査ファイルが必要になる場合があります。一般的に、監査がリソースへのアクセスと同じタイミングで生成され、リソースにアクセスしたルーチンで生成された場合に、監査は最も確実だと見なされます。ログ記録とインストルメンテーションの方針を設計する際には、次のガイドラインを考慮します。

- ユーザー管理イベント、重大なシステム イベント、ビジネスに重大な影響を及ぼす操作、および例外的な作業を、アプリケーションのすべてのレイヤーで監査することを検討します。
- ログ ファイルへのアクセスを制限して、ユーザーに書き込みアクセスのみを許可するなど、安全性の高いログ ファイル管理ポリシーを作成します。開発中および運用中に、ログ記録とインストルメンテーションのメカニズムを構成できるようにします。
- ログ ファイルや監査ファイルには、機密情報を格納しないようにします。

ナビゲーション

ナビゲーションの方針を設計する際には、処理ロジックからナビゲーションの方針を分離するようにします。ナビゲーションの方針では、ユーザーが画面やページ間を簡単に移動できるようにする必要があります。アプリケーションで一貫したナビゲーション構造を設計すると、ユーザーの混乱を最小限に抑えられるだけでなく、アプリケーションの見た目の複雑さを軽減するのに役立ちます。ナビゲーションの方針を設計する際には、次のガイドラインを考慮します。

- Web フォームのポスト バックの手法を使用する場合は、MVP などの設計パターンを使用して、出力のレンダリングから UI 処理を分離することを検討します。Presenter でナビゲーションを処理するこ

とで、ナビゲーション ロジックとユーザー インターフェイス コンポーネントが混在しないようにします。

- RESTful の手法を使用する場合は、MVC パターンを使用して、アプリケーション ロジック、データ、およびナビゲーションを別個のコンポーネントに分離することを検討します。一般的な MVC アプリケーションの実装では、アプリケーションの UI とデータを調整するコントローラー コンポーネントに要求をリダイレクトすることで、柔軟なナビゲーションがサポートされます。
- ナビゲーションをマスター ページにカプセル化して、アプリケーション全体でナビゲーションの一貫性を保つことを検討します。ただし、ハードコーディングされたナビゲーション パスをアプリケーションで使用することは避けます。また、ユーザーが承認されているビューのみに移動できるようにします。
- サイト マップを使用して、ユーザーがサイト上でページを簡単に見つけられるようにすると同時に、必要に応じて、検索エンジンがサイトをクロールできるようにすることを検討します。埋め込まれたリンク、ナビゲーション メニュー、階層リンク ナビゲーションなどの視覚的要素を UI で使用して、ユーザーの現在地、サイトで使用できる項目、およびサイト内を迅速に移動する方法をユーザーが理解できるようにすることを検討します。ウィザードを使用して、フォーム間のナビゲーションを予測可能な方法で実装することを検討します。

ページ レイアウト

特定の UI コンポーネントや UI 処理からページ レイアウトを分離できるように、アプリケーションを設計します。レイアウトの方針を選択する際には、デザイナーと開発者のどちらがレイアウトを作成するのかを考慮します。デザイナーがレイアウトを作成する場合は、コーディングや開発に特化したツールを使用する必要がないレイアウトの手法を選択します。レイアウトの方針を設計する際には、次のガイドラインを考慮します。

- レイアウトには、テーブル ベースのレイアウトではなく、できる限りカスケード スタイル シート (CSS) を使用します。ただし、グリッド レイアウトをサポートしなければならない場合や、データがテーブルとして表されている場合は、テーブル ベースのレイアウトを使用します。テーブル ベースのレイアウトでは、レンダリングが低速になることがあり、複雑なレイアウトによる問題が発生する可能性があることに注意してください。
- ページでは、可能な限り一般的なレイアウトを使用して、アクセシビリティと使いやすさを最大限に高めます。複数のタスクを実行する、サイズの大きいページを設計および開発しないようにします。特に、

各要求で通常実行されるタスクが少ない場合はなおさらです。ページ サイズを可能な限り小さくすることで、パフォーマンスを最大限に高め、帯域幅の要件を減らします。

- ASP.NET アプリケーションではマスター ページを使用して、すべてのページに共通の外観と動作を提供し、サイトを簡単に更新できるようにします。ページの共通するセクションを抽出して別個のユーザー コントロールに分離することで、全体的な複雑さを軽減し、このようなコントロールを再利用できるようにすることを検討します。
- ASP.NET AJAX サーバー コントロールと ASP.NET AJAX クライアント側ライブラリを使用して、クライアント スクリプトをさまざまなブラウザーでより簡単に使用できるようにすることを検討します。また、クライアント側のスクリプトと HTML コードが混在しないようにします。コードが混在すると、ページがより複雑になり、管理が困難になります。クライアント側スクリプトを別個のスクリプト ファイルに配置して、ブラウザーでキャッシュできるようにします。
- 既存の Web アプリケーションを移行する場合は、ASP.NET ページで Silverlight コントロールを使用して、豊富なユーザー エクスペリエンスを提供し、アプリケーションのリエンジニアリングを最小限に抑えることを検討します。

ページ レンダリング

ページ レンダリングを設計する際には、ページを効率的にレンダリングして、インターフェイスのユーザビリティを最大限に高める必要があります。ページ レンダリングの方針を設計する際には、次のガイドラインを考慮します。

- クライアント側スクリプトか ASP.NET AJAX を使用して、必要なポスト バック数を削減することで、ユーザー エクスペリエンスと応答性を向上することを検討します。カスタムのクライアント側スクリプトを使用すると、ブラウザーやバージョンによってスクリプトのサポートが異なるため、アプリケーションのテストが困難になる場合があります。代わりに、最も一般的なブラウザーをサポートする、ASP.NET AJAX の使用を検討します。クライアント側コード (組み込みの ASP.NET コントロールで生成されたスクリプトを含む) を使用すると、その種類に関係なくアクセシビリティに影響する可能性があることに注意してください。専用のユーザー エージェントや障害のあるユーザーのために、適切なアクセシビリティをサポートするようにします。
- データ バインドのオプションについて考慮します。たとえば、コレクション、DataReader オブジェクト、DataSet テーブル、およびカスタム オブジェクトを、多数の ASP.NET コントロールにバインドできます。データ ページングの技法を使用して、大量のデータに関連するスケーラビリティの問題を最小限に抑え、パフォーマンスと応答時間を向上します。

- UI コンポーネントのローカリゼーションをサポートする設計を検討します。
- データのレンダリング関数と取得関数から、ユーザー プロセス コンポーネントを抽象化します。

セッション管理

Web アプリケーションを設計する際には、効率的かつ安全なセッション管理方針を設計することが、パフォーマンスと信頼性の両方において重要です。格納する情報、情報を格納する場所、情報の保持期間などの、セッション管理の要素について考慮する必要があります。セッション管理の方針を設計する際には、次のガイドラインを考慮します。

- セッション状態を本当に格納する必要があるかどうかを検討します。セッション状態を使用すると、各ページ要求でオーバーヘッドが増します。
- 必要な場合にはセッション データを格納するようにしますが、必要に応じて、読み取り専用のセッションを使用したり、セッション状態を完全に無効にすることで、パフォーマンスを向上することを検討します。
- Web サーバーが 1 台の場合、最適なセッション状態のパフォーマンスを実現する必要がある場合、および同時セッションの数が比較的限られている場合は、インプロセス状態ストアを使用します。ただし、セッション データの再作成の負荷が高く、ASP.NET の再起動時に接続が必要な場合は、ローカル Web サーバーで実行しているセッション状態サービスを使用します。サーバー間でセッション データのストレージを一元化する必要がある、複数のサーバー (Web ファーム) のシナリオでは、SQL Server 状態ストアの使用を検討します。
- 状態を別個のサーバーに格納する場合は、SSL や IPSec などの技法を使用して、セッション状態の通信チャネルを保護します。
- シリアル化のコストを削減するには、セッション データに基本型を使用します。

検証

効果的な検証のソリューションを設計することは、アプリケーションのセキュリティと信頼性の両方において重要です。検証が不適切または不十分だと、クロス サイト スクリプト攻撃や、SQL インジェクション攻撃、バッファーオーバーフロー、その他の種類の入力による攻撃などに対してアプリケーションが脆弱になる場合があります。検証の方針を設計する際には、次のガイドラインを考慮します。

- アプリケーションの信頼境界を越えるすべてのデータを検証します。クライアントが制御するすべてのデータに悪意があり、検証する必要があることを前提とします。

- 悪意のある入力を制限、拒否、および一部削除し、すべての入力データを長さ、範囲、形式、および型に基づいて検証するように、検証方針を設計します。
- 最適なユーザー エクスペリエンスを実現してネットワークのラウンド トリップを減らすためにクライアント側の検証機能を使用しますが、セキュリティ上の理由から、サーバーで必ず検証します。
- 検証規則と検証コードを一元管理して再利用するのに役立つサード パーティ製のソリューション、設計パターン、およびライブラリを調査します。

検証の技法に関する詳細については、第 17 章「横断的関心事」を参照してください。

レイヤーの設計に関する考慮事項

アプリケーションにレイヤー型の設計を使用する場合は、次のセクションで説明する、各レイヤー固有の問題について考慮します。

プレゼンテーション レイヤー

Web アプリケーションのプレゼンテーション レイヤーでは、UI を表示して、ユーザー操作を容易にします。設計では、UI コンポーネントからユーザー操作のロジックを分離する、関心の分離に重点を置く必要があります。また、複雑なインターフェイスでは、別個の UI コンポーネントとプレゼンテーション ロジック コンポーネントを使用することを検討し、可能であれば、標準的な Web コントロールに基づいて UI コンポーネントを設計します。アプリケーション間でコントロールを再利用するために、または既存のサーバー コントロールに機能を追加する必要がある場合に、コントロールをアセンブリにコンパイルできます。
Web アプリケーションでは、プレゼンテーション レイヤーは (HTML をレンダリングする) サーバー側コンポーネントと (ブラウザーやユーザー エージェントがスクリプトを実行して HTML を表示する) クライアント側コンポーネントで構成されます。通常、すべてのプレゼンテーション ロジックはサーバー コンポーネントに存在し、クライアント コンポーネントは HTML を表示するだけです。AJAX などのクライアント側の技法を使用すると、クライアント側でロジックを実行することが可能で、一般的にユーザー エクスペリエンスが向上します。ただし、クライアント側の技法を使用するには、追加の開発作業とテストを行う必要があります。クライアント側でなんらかの検証を実行する場合は、クライアント側の検証は簡単に回避される可能性があるため、サーバー側でも検証を行うようにします。

ビジネス レイヤー

Web アプリケーションのビジネス レイヤーを設計する際には、ビジネス ロジックと実行時間の長いワークフローの実装方法について検討します。ビジネス ロジックとワークフローを実装する別個のビジネス レイヤーを使用すると、アプリケーションの保守容易性とテスト容易性を向上して、ビジネス ロジックの共通機能を一元化して再利用できます。実際のデータを表すビジネス エンティティを設計し、このエンティティを使用してコンポーネント間でデータを渡すことを検討します。

ビジネス レイヤーがステートレスになるように設計することで、リソースの競合を減らしてパフォーマンスを向上します。また、メッセージ ベースのインターフェイスを使用することを検討します。メッセージ ベースのインターフェイスは、ステートレスな Web アプリケーションのビジネス レイヤーで適切に機能します。ビジネス レイヤーでビジネス クリティカルな操作を実行する場合は、トランザクションを使用するように設計することで、整合性を維持し、データの損失を防ぎます。

データ レイヤー

データベースにアクセスするのに必要なロジックを抽象化する Web アプリケーションで、データ レイヤーを設計することを検討します。別個のデータ レイヤーを使用することで、アプリケーションの構成と管理が容易になり、データベースの詳細がアプリケーションの他のレイヤーに公開されなくなります。データ ソースを更新するためにデータ レイヤーで設定または使用できるエンティティ オブジェクトを設計し、他のレイヤーと通信したり、レイヤー間でデータを渡す際に、データ転送オブジェクト (DTO) を使用します。

接続プールを利用して、開かれた接続の数を最小限に抑えるようにデータ レイヤーを設計し、バッチ操作を使用してデータベースへのラウンド トリップを減らすことを検討します。データ レイヤーでは、サービス エージェントを使用して外部サービスにアクセスしなければならない場合もあります。また、データ アクセス エラーをハンドルし、例外をビジネス レイヤーに伝達するように、例外ハンドルの方針を設計するようにします。

サービス レイヤー

ビジネス レイヤーをリモート ティアに配置する場合や、Web サービスを使用してビジネス ロジックを公開する場合は、別個のサービス レイヤーを設計することを検討します。サービスを使用するクライアントの具体的な詳細を想定しないことで再利用性を最大限に高め、既存のクライアントのサービス インターフェイスに支障をきたす可能性がある、時間の経過と共に行われる変更を回避するように、サービスを設計します。また、複数のバージョンのインターフェイスを実装して、クライアントが適切なバージョンに接続できるようにします。

ビジネス レイヤーがリモート ティアに配置されている場合は、粒度の粗いサービス メソッドを設計することで、ラウンド トリップの回数を最小限に抑え、疎結合を実現します。また、(同じ要求メッセージが複数回到着する状況をサービスが管理できるようにするために) べき等性を保持し、(決められた一連のタスクの手順を実行するメッセージが間違った順序で到着する状況をサービスが管理できるようにするために) 交換性を保持するように、サービスを設計します。サービス インターフェイスにビジネス ルールを実装すると、インターフェイスの安定性を保つのが難しくなることがあり、コンポーネントとクライアント間で不必要な依存関係が生じる場合があるので、サービス インターフェイスにビジネス ルールを実装しないようにします。

最後に、適切なプロトコルとトランスポート メカニズムを選択することで、相互運用性の要件について検討します。たとえば、対象範囲が幅広い場合は ASMX を使用し、構成をより細かく制御する場合には WCF を使用します。インターフェイスで SOAP または REST、あるいは両方のメソッドを公開するかどうかを決定します。サービスの公開に関する詳細については、第 9 章「サービス レイヤーのガイドライン」と第 18 章「通信とメッセージ」を参照してください。

テストとテスト容易性に関する考慮事項

テスト容易性は、システムとそのコンポーネントでどの程度簡単に、テスト条件を作成して、条件が満たされているかどうかを判断するためにそのテストを実施できるかを示す指標です。テスト容易性は、問題をより迅速に診断することを容易にし、保守にかかるコストを削減するので、アーキテクチャを設計する際にはテスト容易性について考慮する必要があります。テスト容易性については、次のガイドラインを考慮します。

- 設計段階で、アプリケーションのレイヤーとコンポーネントの入出力を明確に定義します。
- プレゼンテーション レイヤーで MVC や MVP などの プレゼンテーションの分離 パターンを使用することを検討します。このパターンを使用すると、プレゼンテーション ロジックで単体テストを実行できるようになります。
- ビジネス ロジックとビジネス ワークフローを実装するための別個のビジネス レイヤーを設計します。こうすることで、アプリケーションのテスト容易性が向上します。
- 個別にテストできる疎結合コンポーネントを設計します。
- ログ記録とトレースの効果的な方針を設計します。これにより、他の方法では見つけるのが難しいエラーを検出したり、トラブルシューティングしたりすることが可能になります。監視ツールや管理ツールで使用できる、ログ記録とトレースの情報を提供します。これは、エラーが発生したときに、問題のあるコードを特定して修正するのに役立ちます。ログ ファイルには、問題を再現するのに使用する情報を含める必要があります。

テクノロジに関する考慮事項

ASP.NET の観点では、マイクロソフト プラットフォームでは、ASP.NET Web フォーム モデルを他のさまざまなテクノロジ (ASP.NET AJAX、ASP.NET MVC、Silverlight、ASP.NET Dynamic Data など) と組み合わせることができます。テクノロジについては、次のガイドラインを考慮します。

- Web ブラウザーや専用のユーザー エージェント経由でアクセスするアプリケーションを構築する必要がある場合は、ASP.NET の使用を検討します。
- ページの再読み込み回数を少なくして、対話性とバックグラウンド処理を強化するアプリケーションを構築する必要がある場合は、ASP.NET と AJAX の併用を検討します。
- リッチ メディア コンテンツと対話性を備えたアプリケーションを構築する必要がある場合は、ASP.NET と Silverlight コントロールの併用を検討します。
- ASP.NET を使用している場合は、マスター ページを使用して、すべてのページで使用する一貫した UI を実装することを検討します。
- 基になるデータベースのデータ モデルに基づくページを使用するデータ ドリブン Web アプリケーションを構築する場合は、ASP.NET Dynamic Data を使用することを検討します。
- テスト駆動開発の手法を使用する場合や、UI を細かく制御する必要がある場合は、MVC パターンと ASP.NET MVC を併用して、アプリケーションの UI からアプリケーション ロジックとナビゲーション ロジックを明確に分離することを検討します。

配置に関する考慮事項

Web アプリケーションを配置する際には、レイヤーとコンポーネントの配置が、アプリケーションのパフォーマンス、スケーラビリティ、およびセキュリティに及ぼす影響を考慮する必要があります。また、設計のトレードオフについても考慮する必要があります。ビジネス要件とインフラストラクチャの制約に応じて、分散配置か非分散配置のいずれかの手法を使用します。非分散配置では、通常、物理的な境界を越える必要がある呼び出しの数を減らすことで、パフォーマンスを最大限に高めます。ただし、分散配置では、スケーラビリティを向上し、各レイヤーのセキュリティを別個に確保することが可能です。

非分散配置

非分散配置のシナリオでは、Web アプリケーションの論理的に分離されているすべてのレイヤーが、同じ Web サーバー上に物理的に配置されます (ただし、データベースは除きます)。この場合は、アプリケーションで複数の同時接続ユーザーを処理する方法や、同じサーバーに配置されているレイヤーのセキュリティを保護する方法について考慮する必要があります。図 2 に、このシナリオを示します。

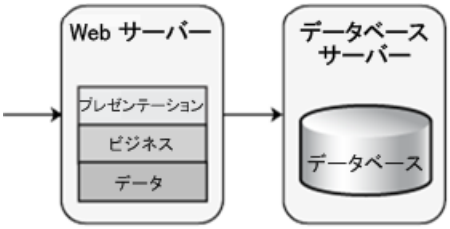


図 34

Web アプリケーションの非分散配置

非分散配置を選択する際には、次のガイドラインを考慮します。

- Web アプリケーションがパフォーマンスの影響を受けやすい場合は、非分散配置を使用します。これは、他のレイヤーへのローカル呼び出しが、ティア間のリモート呼び出しで発生するパフォーマンスへの影響を軽減するためです。
- ビジネス ロジックを他のアプリケーションと共有する必要がなく、プレゼンテーション レイヤーのみがビジネス ロジックにアクセスする場合は、ビジネス レイヤーにコンポーネント ベースのインターフェイスを設計します。
- ビジネス ロジックとプレゼンテーション ロジックを同じプロセスで実行する場合は、ビジネス レイヤーで認証を行わないようにします。
- 信頼されたサブシステム モデルを通じて、信頼された ID を使用し、データベースにアクセスします。これにより、アプリケーションのパフォーマンスとスケーラビリティが向上します。
- Web サーバーとデータベース サーバー間で渡される機密データを保護する方法を決定します。

分散配置

分散配置のシナリオでは、Web アプリケーションのプレゼンテーション レイヤーとビジネス レイヤーが別個の物理ティアに配置され、リモートで通信します。通常、ビジネス レイヤーとデータ アクセス レイヤーは同じサーバーに配置します。図 3 に、このシナリオを示します。

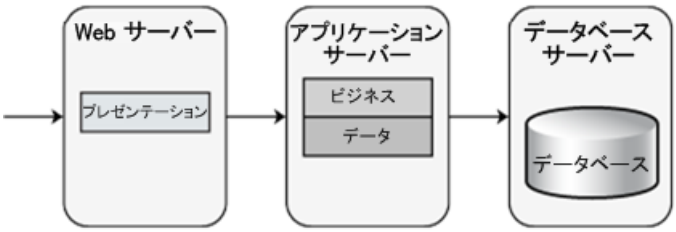


図 3

Web アプリケーションの分散配置

分散配置を選択する際には、次のガイドラインを考慮します。

- 必要でない限り、ビジネス レイヤーを別のティアに配置しないようにします。たとえば、スケーラビリティを最大限に高めるため、またはセキュリティ要件によりビジネス ロジックをフロントエンド Web サーバーに配置できない場合などは、ビジネス レイヤーを別のティアに配置します。
- ビジネス レイヤーにメッセージ ベースのインターフェイスを使用することを検討します。
- TCP プロトコルとバイナリ エンコードを使用してビジネス レイヤーと通信して、最適なパフォーマンスを実現することを検討します。
- 異なる物理ティア間で渡される機密データを保護することを検討します。

負荷分散

Web アプリケーションを複数のサーバーに配置する場合は、負荷分散を使用して要求を分散し、別の Web サーバーで要求を処理することが可能です。負荷を分散すると、応答時間、リソースの使用率、およびスループットを最大限に高められます。図 4 に、このシナリオを示します。

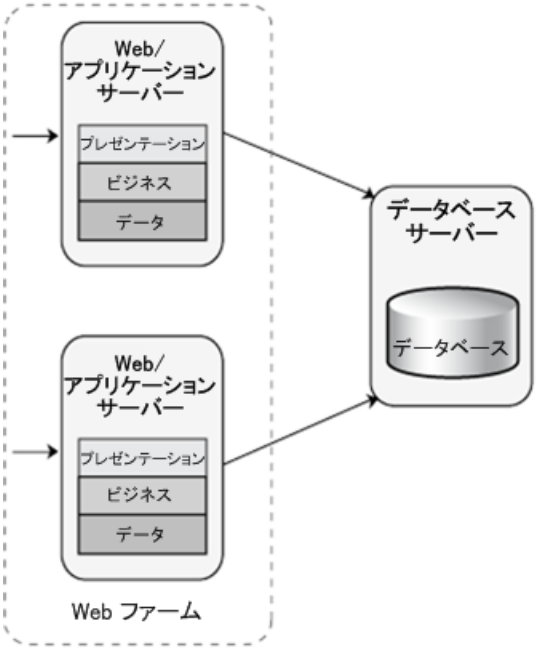


図 4

Web アプリケーションの負荷分散

負荷分散を使用するように Web アプリケーションを設計する場合は、次のガイドラインを考慮します。

- サーバー アフィニティは、アプリケーションのスケールアウトの機能に悪影響を及ぼす可能性があるため、Web アプリケーションを設計する際には、サーバー アフィニティはできる限り回避します。サーバー アフィニティは、特定のクライアントからのすべての要求を同じサーバーで処理しなければならない場合に発生します。これは、通常、ローカルで更新可能なキャッシュ、インプロセス セッション状態ストア、またはローカル セッション状態ストアを使用する際に発生します。サーバー アフィニティをサポートしなければならない場合は、同じユーザーからのすべての要求を、同一のサーバーにルーティングするように、クラスターを構成します。
- Web アプリケーションに、ステートレスなコンポーネント (たとえば、インプロセスの状態が保持されない Web フロントエンドや、ステートフルではないビジネス コンポーネントなど) を設計することを検討します。ユーザーの状態を格納する必要がある場合は、Web ファームでインプロセス セッション管理を使用しないようにします (ただし、アフィニティを構成して、同じユーザーからの要求を同一のサーバーにルーティングするようにできる場合は除きます)。代わりに、アウトプロセスの状態サーバー サービスやデータベース サーバーを使用します。
- Windows ネットワーク負荷分散 (NLB) をソフトウェア ソリューションとして使用して、要求をリダイレクトする機能をアプリケーション ファームのサーバーに実装します。

- クラスタリングを使用して、ハードウェア障害への影響を最小限に抑えることを検討します。
- アプリケーションに、高入出力に関する要件がある場合は、複数のデータベース サーバー間でデータベースを分割することを検討します。

配置パターンの詳細については、第 19 章「物理ティアと配置」を参照してください。

関連する設計パターン

次の表に示すように、主要なパターンは、キャッシュ、例外管理、ログ記録とインストルメンテーション、ページ レイアウト、プレゼンテーション、要求処理、サービス インターフェイス レイヤーなどのカテゴリに分類されます。各カテゴリの設計を行う際には、これらのパターンの使用を検討します。

カテゴリ	関連するパターン
キャッシュ	**Cache Dependency**: 外部からの情報を使用して、キャッシュに格納されているデータの状態を判断します。 **Page Cache**: 頻繁にアクセスされ、変更の頻度が低く、構築に大量のシステム リソースを消費する動的な Web ページの応答時間が向上します。
例外管理	**Exception Shielding**: 外部のシステムやユーザーに公開してはならない例外データをフィルター処理します。
ログ記録とインストルメンテーション	**Provider**: あらゆるカスタム実装がシームレスにプラグインされるようにするために、クライアント API とは異なる API を公開するコンポーネントを実装します。
ページ レイアウト (UI)	**Composite View**: 個々のビューを組み合わせて、複合的に表します。 **Template View**: 共通のテンプレート ビューを実装し、このテンプレート ビューを使用してビューを取得または構成します。 **Transform View**: プレゼンテーション ティアに渡されたデータを HTML に変換して、UI で表示します。 **Two-Step View**: モデル データを、具体的な形式を指定しない論理表現に変換してから、その論理表現を、必要な実際の形式に変換します。
プレゼンテーション	**Model-View-Controller**: ユーザーの入力に基づいて、ドメイン、プレゼンテーション、およびアクションのデータを、3 つの別個のクラスに分離します。Model では、アプリケーション ドメインの動作とデータを管理し、その

	状態に関する情報を求める (通常、View からの) 要求に応答して、状態を変更するための (通常、Controller からの) 命令に応答します。View では、情報の表示を管理します。Controller では、ユーザーからのマウスとキーボードによる入力を解釈し、必要に応じて、Model と View に変更を通知します。 **Model-View-Presenter**: 要求処理を 3 つの役割に分割します。View ではユーザーの入力を処理し、Model ではアプリケーション データとビジネス ロジックを管理し、Presenter ではプレゼンテーション ロジックを管理して View と Model 間の通信を調整します。 **Passive View**: MVC パターンの変化形です。コントローラーでユーザーの入力を処理できるようにして、View の更新という役割を維持しながら、View の役割を最小限に抑えます。 **Supervising Presenter** (または Supervising Controller): Model-View-Controller パターンの一種で、コントローラーが複雑なロジックを処理する (具体的には View 間の調整をする) 一方で、View 固有の簡単なロジックは View で処理されます。
要求処理	**Intercepting Filter**: 一連の構成可能なフィルター (独立したモジュール) を作成して、Web ページ要求の一般的な処理と処理後のタスクを実装します。 **Page Controller**: 要求からの入力を受け取り、その入力を特定のページや Web サイトの操作で処理します。 **Front Controller**: 単一のハンドラー オブジェクトを通じて、すべての要求を渡すことで要求処理を統合します。このハンドラー オブジェクトは、実行時にデコレータを使用して変更できます。
サービス インターフェイス レイヤー	**Façade**: 一連の操作に統一されたインターフェイスを実装します。これにより、システム間の結合が簡略化および緩和されます。 **Service Interface**: 他のシステムがサービスとの通信に使用できる、プログラマティック インターフェイスです。

Page Cache パターンの詳細については、「Microsoft .NET を使用したエンタープライズ ソリューション パターン」(http://msdn.microsoft.com/ja-jp/library/ms998469.aspx) を参照してください。

Model-View-Controller (MVC)、Page Controller、Front Controller、Template View、Transform View、および Two-Step View の各パターンの詳細については、Martin Fowler 著 『エンタープライズ アプリケーション アーキテクチャ パターン』 (翔泳社、2005 年) を参照するか、http://martinfowler.com/eaaCatalog (英語) を参照してください。

Composite View、Supervising Presenter、および Presentation Model の各パターンの詳細については、「Composite Application Library のパターン」(http://msdn.microsoft.com/ja-jp/library/cc707841.aspx) を参照してください。

Exception Shielding パターンの詳細については、「Useful Patterns for Services」(http://msdn.microsoft.com/en-us/library/cc304800.aspx、英語) を参照してください。

Service Interface パターンの詳細については、「サービス インターフェイス」(http://msdn.microsoft.com/ja-jp/library/ms998421.aspx) を参照してください。

Provider パターンの詳細については、「Provider Model Design Pattern and Specification, Part 1」(http://msdn.microsoft.com/en-us/library/ms972319.aspx、英語) を参照してください。

関連情報

Web リソースに簡単にアクセスするには、http://www.microsoft.com/architectureguide (英語) でオンライン版の参考文献を参照してください。

- Web クライアント アプリケーションの設計と実装に関する詳細については、「Web クライアントの設計と実装に関するガイドライン」(http://msdn.microsoft.com/ja-jp/library/ms978605.aspx) を参照してください。
- 分散 Web アプリケーションの設計に関する詳細については、「分散アプリケーションのデザイン」(http://msdn.microsoft.com/ja-jp/library/aa292470(VS.71).aspx) を参照してください。
- Web アプリケーションのパフォーマンスの問題に関する詳細については、「Improving .NET Application Performance and Scalability」(http://msdn.microsoft.com/en-us/library/ms998530.aspx、英語) を参照してください。
- Web アプリケーションのセキュリティに関する詳細については、「Improving Web Application Security: Threats and Countermeasures」(http://msdn.microsoft.com/en-us/library/ms994921.aspx、英語) を参照してください。

22

リッチ クライアント アプリケーションの設計

概要

この章では、リッチ クライアント アプリケーションの使用に関する主要なシナリオ、リッチ クライアント アプリケーションで使用されるコンポーネント、およびリッチ クライアント アプリケーションの設計に関する重要な考慮事項について説明します。また、リッチ クライアント アプリケーションの配置シナリオと、リッチ クライアント アプリケーションを設計する際の主要なパターンとテクノロジに関する考慮事項についても紹介します。

リッチ クライアントの UI では、スタンドアロンのシナリオ、接続型のシナリオ、不定期に接続するシナリオ、および非接続型のシナリオで動作する必要があるアプリケーションで、パフォーマンスが高く、機能が豊富な対話型のユーザー エクスペリエンスを提供できます。Windows フォーム、Windows Presentation Foundation (WPF)、および Microsoft Office Business Application (OBA) の開発環境とツールを使用できるので、開発者はすばやく簡単にリッチ クライアント アプリケーションを構築できます。

これらのテクノロジを使用すると、スタンドアロン アプリケーションだけでなく、クライアント コンピューターで実行しながら、クライアントに必要な操作を公開する他のレイヤー (論理レイヤーと物理レイヤーの両方を含む) や他のアプリケーションによって公開されるサービスと通信するアプリケーションを作成することもできます。クライアントに必要な操作には、データ アクセス、情報の取得、検索、他のシステムへの情報の送信、バックアップなどがあります。図 1 は、一般的なリッチ クライアント アプリケーションの概要と各レイヤーで一般的に使用されるコンポーネントを示しています。

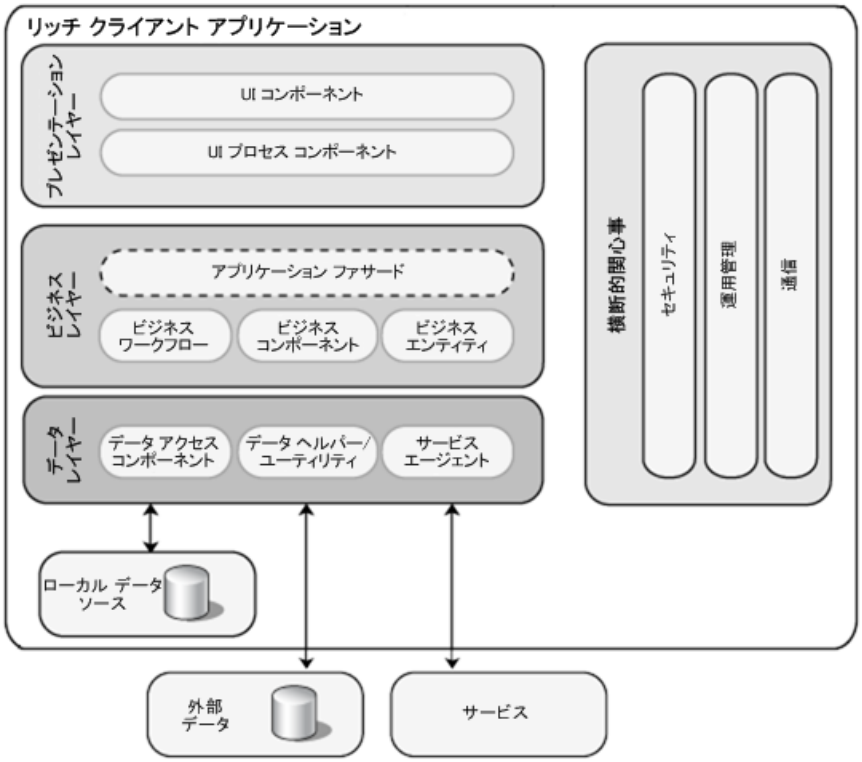


図 35

一般的なリッチ クライアント アプリケーションの概要

一般的なリッチ クライアント アプリケーションは、プレゼンテーション レイヤー、ビジネス レイヤー、およびデータ レイヤーの 3 つのレイヤーに分割できます。通常、プレゼンテーション レイヤーには UI コンポーネントとプレゼンテーション ロジック コンポーネントが、ビジネス レイヤーにはビジネス ロジック コンポーネント、ビジネス ワークフロー コンポーネント、およびビジネス エンティティ コンポーネントが、データ レイヤーにはデータ アクセス コンポーネントとサービス エージェント コンポーネントが含まれます。レイヤー型の設計の詳細については、第 5 章「レイヤー型アプリケーションのガイドライン」を参照してください。各レイヤーに適したコンポーネントの詳細については、第 10 章「コンポーネントのガイドライン」を参照してください。

リッチ クライアント アプリケーションは、プレゼンテーション レイヤーを中心に構成された非常に軽量なアプリケーションになることもあります。この場合、アプリケーションでは、サーバー コンピューターにホストされたリモート ビジネス レイヤーにサービス経由でアクセスします。データを処理して保存するためにすべてのデータをサーバーに送信するデータ入力アプリケーションが、その一例です。

逆に、ほとんどの処理を内部で実行し、情報を使用および送信する際にのみ、他のサービスやデータ ストアと通信する複雑なアプリケーションになることもあります。Microsoft Excel® など、複雑なローカル タスクを実行し、状態とデータをローカルに保存して、リンクしたデータの取得と更新を行うときにだけ、リモート サーバーと通信す

るアプリケーションが、この一例です。このような種類のリッチ クライアント アプリケーションには、専用のビジネス レイヤーとデータ レイヤーが含まれていることがあります。このようなアプリケーションのビジネス レイヤーとデータ レイヤーのガイドラインは、すべての種類のアプリケーションについて説明したガイドラインと同じです。

設計に関する一般的な考慮事項

リッチ クライアント アプリケーションを設計する際、ソフトウェア アーキテクトの目標は、適切なテクノロジを選択し、タスクを関連領域に分離してアプリケーションがなるべく複雑にならないような構造を設計することです。設計は、パフォーマンス、セキュリティ、再利用性、および保守容易性に関するアプリケーションの要件を満たす必要があります。

リッチ クライアント アプリケーションを設計する際には、次のガイドラインを考慮します。

- **アプリケーションの要件に基づいて適切なテクノロジを選択する**: 適切なテクノロジには、Windows フォーム、WPF、XAML ブラウザー アプリケーション (XBAP)、OBA などがあります。
- **プレゼンテーション ロジックとインターフェイスの実装を分離する**: プレゼンテーション モデル、監視プレゼンター(または 監視コントローラー) など、UI レンダリングと UI ロジックを分離する設計パターンの使用を検討します。これらのパターンを使用すると、保守が容易になり、再利用性とテスト容易性が向上します。アプリケーションで分離したコンポーネントを使用すると、依存関係を軽減でき、保守とテストを容易にでき、再利用性を高められます。
- **プレゼンテーション タスクとプレゼンテーション フローを特定する**: これらを特定すると、複数の画面やウィザードのプロセスで、個別の画面や手順を設計するのに役立ちます。
- **適切で使いやすいインターフェイスが提供されるように設計する**: レイアウト、ナビゲーション、コントロールの選択、ローカリゼーションなどの機能を考慮して、アクセシビリティとユーザビリティを最大限に高めます。
- **すべてのレイヤーで関心事を分離する**: たとえば、ビジネス ルールやプレゼンテーションと無関係な処理を抽出して、別個のビジネス レイヤーに配置します。また、データ アクセス コードを、データ レイヤーに配置された別個のコンポーネントに分離します。
- **共通のプレゼンテーション ロジックを再利用する**: テンプレート、汎用的なクライアント側の検証機能、およびヘルパー クラスを含むライブラリは、複数のアプリケーションで再利用できます。
- **クライアントをそのクライアントで使用するすべてのリモート サービスと疎結合する**: メッセージ ベースのインターフェイスを使用して、別個の物理ティアに配置されたサービスと通信します。

- **他のレイヤーのオブジェクトとの密結合を避ける**: アプリケーションの他のレイヤーと通信する際には、一般的なインターフェイスの定義、抽象型基本クラス、またはメッセージによって提供される抽象化を使用します。たとえば、依存関係の注入 (Dependency Injection) パターンや 制御の反転 (Inversion of Control) パターンを実装すると、レイヤー間で共通の抽象化を提供できます。
- **リモート レイヤーにアクセスする際のラウンド トリップの回数を減らす**: 粒度の粗いメソッドを使用して、できる限り非同期に実行し、UI のブロックやフリーズを回避します。

ビジネス レイヤーの設計に関する詳細については、第 7 章「ビジネス レイヤーのガイドライン」を参照してください。データ レイヤーの設計に関する詳細については、第 8 章「データ レイヤーのガイドライン」を参照してください。

設計に関する具体的な問題

設計を策定する際に考慮する必要がある一般的な問題がいくつかあります。これらの問題は、設計の具体的な領域に分類できます。次のセクションでは、各領域で一般的に発生する問題を解決するのに役立つガイドラインを提供します。

- ビジネス レイヤー
- 通信
- 構成
- 構成管理
- データ アクセス
- 例外管理
- 保守容易性
- プレゼンテーション レイヤー
- 状態管理
- ワークフロー

ビジネス レイヤー

一般的なシン リッチ クライアントは、ビジネス システムのインターフェイスとして機能し、ビジネス レイヤーはビジネス システムに含まれ、通常はサービスとして公開されます。一方、一般的なシック リッチ クライアントでは、ビジネス レイヤーはクライアント自体に配置されます。リッチ クライアントのビジネス レイヤーを設計する際には、次のガイドラインを考慮します。

- アプリケーションで使用するビジネス レイヤーとサービス インターフェイスを特定します。アプリケーションがリモート サービスにアクセスする場合は、インターフェイスの定義をインポートし、そのインターフェイスを使用してリモート サービスの機能にアクセスするコードを作成します。このようにすると、クライアントと、クライアントで使用するビジネス レイヤーやサービスの結合を最小限に抑えることができます。
- ビジネス ロジックに機密情報が含まれていない場合は、一部のビジネス ルールをクライアントに配置して、UI とクライアント アプリケーションのパフォーマンスを向上することを検討します。ビジネス ロジックに機密情報が含まれている場合は、ビジネス レイヤーを別個のティアに配置する必要があります。
- ビジネス ルールなど、クライアント側の処理を実行するのに必要な情報をクライアントで取得する方法、この情報が変更されたときにクライアントで自動的に情報を更新する方法、および要件の変化に応じてユーザーや管理者がビジネス ルールを更新する方法を検討します。クライアントの起動時にリモート サーバーからビジネス ルールの情報を取得することもできます。

ビジネス レイヤーの設計に関する詳細については、第 7 章「ビジネス レイヤーのガイドライン」を参照してください。

通信

リッチ クライアント アプリケーションのビジネス レイヤーとデータ レイヤーがリモート ティアに配置されてサービスとして公開されている場合、またはリッチ クライアント アプリケーションで他のリモート サービスを使用している場合、リッチ クライアント アプリケーションでは、さまざまなプロトコルや手法を使用してこのようなサービスと通信できます。これらのプロトコルや手法には、HTTP 要求、単純な SMTP 電子メール メッセージ、SOAP Web サービス メッセージ、リモート コンポーネントの DCOM、リモート データベースのアクセス プロトコル、またはその他の TCP/IP に準拠した標準の通信プロトコルやカスタム通信プロトコルなどがあります。ビジネス レイヤーとデータ レイヤーがクライアントに配置されている場合は、プレゼンテーション レイヤーで、オブジェクト

ベースの手法を使用してこれらのレイヤーと通信できます。通信の方針を設計する際には、次のガイドラインを考慮します。

- リモート物理ティアのサービスと通信する場合は、できる限りメッセージ ベースのプロトコルを使用します。このようにすると、より自然な方法で非同期呼び出しを行ってプレゼンテーション レイヤーのブロックを回避し、負荷分散された構成やフェールオーバー サーバーの構成をサポートできます。粒度の粗いインターフェイスを使用して、ネットワーク トラフィックを最小限に抑え、パフォーマンスを最大限に高めます。
- 必要に応じてアプリケーションのオフライン処理を有効にします。また、接続状態を検出して監視します。接続が切断されているときはローカルに情報をキャッシュし、再度通信できるようになったら再度同期します。ネットワークに接続されていない状態で情報を失わずに起動とシャットダウンまたは再起動を繰り返すことができるように、アプリケーションの状態とデータをローカルの永続的なキャッシュに保持することを検討します。
- 機密情報と通信チャネルを保護するために、IPSec や SSL を使用してチャネルをセキュリティで保護し、暗号化を使用してデータを保護し、デジタル署名を使用してデータの改ざんを検出することを検討します。
- アプリケーションで大量のデータやデータ セットを使用したり送信したりする必要がある場合は、パフォーマンスやネットワークが影響を受ける可能性を考慮します。メッセージ ベースのプロトコル (SMTP、SOAP など) でのデータ ペイロードのサイズを最小化する圧縮メカニズムを使用して、TCP などのより効率的な通信プロトコルを選択します。通信に関するオープン スタンダードをアプリケーションでサポートする必要がなければ、カスタム バイナリ形式を選択します。

アプリケーションのクライアントとレイヤーの間の通信に関する詳細については、第 18 章「通信とメッセージ」を参照してください。

構成

アプリケーションの拡張性と保守容易性を最大限に高めるには (特に、多くのビジネス シナリオでよく行われるように複雑な UI を公開する場合は)、"構成" カテゴリの設計パターンを使用したインターフェイスの実装を検討します。このパターンでは、実行時に動的に読み込まれる独立したモジュールやフォームで UI が構成されています。この方法は、ユーザーが特定のタスクを実行するために複数のフォームを開いたり、データをさまざまな方法で操作したりする可能性がある場合に便利です。ユーザーは必要に応じてフォームを開いたり閉じたりすることが可能で、アプリケーションでは、このようなフォームを必要なときにのみ読み込むことで、パフォーマンスを最大限に高めて起

動時の遅延を短縮できます。また、ユーザーが各自の要件に合わせてレイアウトやコンテンツを変更できるように、ユーザーによるパーソナル化をサポートする方法についても検討します。構成の方針を設計する際には、次のガイドラインを考慮します。

- 機能の仕様と要件に基づいて、必要なインターフェイス コンポーネントの適切な種類を特定します。たとえば、使用できるコンポーネントとしては、Windows フォーム、WPF フォーム、Office 形式のドキュメント、ユーザー コントロール、カスタム モジュールなどがあります。
- 構成が適している場合は適切な構成メカニズムを特定し、再利用可能なモジュールからビューを構成することを検討します。たとえば、複合ビュー (Composite View) パターンを使用して、モジュール形式のアトミックなコンポーネントからビューを構築します。また、構成フレームワーク (patterns & practices の Composite Client Application Guidance など) や開発環境の組み込み機能 (ユーザー コントロール、ドキュメント パネルなど) を使用することもできます。ただし、コンポーネント間の依存関係に注意し、できる限り抽象パターンを使用して、保守容易性に関する問題を回避します。また、構成可能なコンポーネントの自動更新を管理する機能とバージョン管理機能についてもできる限り実装します。
- 複合インターフェイスを構成するさまざまなフォームやプレゼンテーション コンポーネントの間の通信をサポートする必要がある場合は、Publish/Subscribe パターン、Command パターンなど、分離された通信技法の実装を検討します。このような技法を実装すると、これらのコンポーネント間の結合が最小限に抑えられ、テスト容易性が向上します。
- 構成可能なインターフェイスで使用するすべてのフォームに必要なコードを簡略化して最小限に抑えるために、選択した実装テクノロジで使用できる適切なテンプレートとデータ バインド技法を利用します。
- ユーザーがインターフェイスで構成可能なコンポーネントのレイアウトをカスタマイズできるように、パーソナル化の実装を検討します。

構成管理

通常、リッチ クライアント アプリケーションには、起動時 (場合によっては実行中) に読み込まれる構成情報が必要です。構成情報には、ネットワークの情報、接続情報、ユーザー設定、UI のビジネス ルール、表示とレイアウトに関する一般的な設定などがあります。このような構成情報の全部または一部は、ローカルに格納するか、アプリケーションの起動時にリモート サーバーからダウンロードできます。また、ユーザーのローカル プロファイルにユー

ザー設定、レイアウト設定、およびその他の UI データを格納するなど、アプリケーションの実行中や終了時に構成情報の変更内容を保存する必要もあります。構成管理の方針を設計する際には、次のガイドラインを考慮します。

- アプリケーションの実行中に変更される可能性のある構成データを特定します。たとえば、ファイルの場所、運用時と異なる開発者の設定、ログ記録、アセンブリ参照、通知の連絡先情報などです。必要に応じて、構成の変更を検出して動的に適用するようにアプリケーションを設計します。
- ローカルのストレージの場所と一元的なストレージの場所のいずれかを選択します。通常、ユーザーが管理するデータ (プロファイル情報、パーソナル化設定など) はローカルに格納しますが、ローミングを有効にするためにデータを一元的に格納することを検討することもできます。グローバルなアプリケーション設定は一元的な場所に格納する必要がありますが、パフォーマンス上の理由から、ローカルにダウンロードする場合があります。
- 機密の構成情報を特定して、ネットワーク経由の伝送中、ローカルに配置されているとき、およびメモリに格納されているときでも、この構成情報を保護するための適切なメカニズムを実装します。
- ローカルの構成が影響を受けたりオーバーライドされたりする可能性のある、すべてのグローバルなセキュリティ ポリシーとグループ ポリシーのオーバーライドを考慮します。

データ アクセス

通常、リッチ クライアント アプリケーションは、リモート サーバーに格納されたデータとローカル コンピューターに格納されたデータの両方にアクセスします。データ アクセスは、多くの場合パフォーマンスに大きな影響を及ぼすので、ユーザーがアプリケーションとそのユーザビリティや応答性を判断する最も明白な要素になります。開発者は、データ アクセス ルーチンやティア間のデータ転送に関するパフォーマンスを最大限に高めることを目標にする必要があります。また、使用するデータ型を考慮してアプリケーションを設計する必要もあります。公開されている形式のデータをクライアント アプリケーションで処理できない場合は、データの形式を変換する変換メカニズムを実装する必要があります。ただし、このメカニズムを実装するとパフォーマンスに影響が及びます。データ アクセスの方針を設計する際には、次のガイドラインを考慮します。

- データの読み込み中も UI の応答性が確保されるように、できる限りデータを非同期に読み込みます。ただし、読み込みが完了する前にユーザーがデータを操作しようとした場合に発生する競合を把握し、このような場合に発生するエラーを防止するようにインターフェイスを設計する必要もあります。
- クライアントで膨大な量のデータを使用する場合は、パフォーマンスを向上するために、データをチャンクに分割してローカル キャッシュに非同期に読み込むことを検討します。この場合、タイム スタン

プやイベントなどの手法を使用して、ローカル コピーと元のデータの間の不整合を処理する方法を計画する必要があります。

- 不定期に接続するシナリオでは、接続を監視し、サービス ディスパッチャー メカニズムを実装してバッチ処理をサポートして、ユーザーが同時に複数のデータの更新処理を実行できるようにします。
- 複数のユーザーが一元管理されているデータ ストアを更新しようとした場合に発生する、同時実行制御の競合を検出して対応する方法を決定します。また、オプティミスティック同時実行制御モデルとペシミスティック同時実行制御モデルについて調査します。

リッチ クライアント アプリケーションのデータ アクセスとデータ処理の詳細については、この章の「データ処理に関する考慮事項」を参照してください。

例外管理

すべてのアプリケーションとサービスでは、エラーや例外が発生する可能性があるので、このようなエラーや例外の検出と管理に適した方針を実装する必要があります。例外管理の方針が堅牢で適切に設計されていると、アプリケーションの設計を簡略化して、セキュリティと管理性を向上できます。また、アプリケーションが開発しやすくなり、開発にかかる時間とコストを削減できます。通常、リッチ クライアント アプリケーションでは、ユーザーに通知する必要があります。また、検証メッセージなど、簡単な UI エラーを除くすべてのエラーと例外については、運用スタッフや監視システムが使用できるようにログ記録を検討する必要があります。通常、ログ記録に関する主な課題は、ログ情報の照合、またはすべてのクライアントからアクセスできる一元的なサーバー ベースのログ シンクの設計です。例外管理の方針を設計する際には、次のガイドラインを考慮します。

- アプリケーション内で発生しやすいエラーや例外を特定し、ユーザーに通知するだけでかまわないエラーや例外を特定します。通常、検証エラーなどのエラーは、ローカルでユーザーに通知されるだけで、ログには記録されません。しかし、無効なログオン試行が繰り返されたり、悪意のあるデータが検出されたりした場合のエラーは、ログに記録して管理者に通知する必要があります。すべての実行例外とアプリケーション エラーはログに記録し、必要に応じて管理者に通知する必要があります。
- 例外ハンドルの全体的な方針を決定します。たとえば、エラーの解決に役立つ追加データを含む、他のアプリケーション固有の例外やカスタム例外で例外をラップしたり、機密情報が公開されないように例外を置換したりする必要があります。また、ハンドルされない例外を検出してログに記録するメカニズムも実装します。このような作業には、patterns & practices の Enterprise Library など、例外管理のフレームワークが役立つ場合があります。

- 例外情報を格納する方法、必要に応じてアプリケーションの他のレイヤーに例外情報を渡す方法、および管理者に通知する方法を決定します。ローカル コンピューターからイベントを読み取れる監視ツールや監視環境の使用を検討し、アプリケーションの状態を確認できるビューを管理者に表示します。
- 機密情報が表示されたりログ ファイルや監査ファイルに格納されたりしないよう、ユーザーに公開される例外情報の一部を削除します。アプリケーションの他の物理ティアに例外やエラーを伝達する際には、必要に応じて、情報を暗号化してセキュリティで保護されたチャネルを使用します。
- ハンドルできる例外のみをキャッチします。たとえば、null 値を変換する際に発生するデータ変換の例外をキャッチします。例外を使用してビジネス ロジックを制御しないようにします。

例外ハンドルの詳細については、第 17 章「横断的関心事」を参照してください。

保守容易性

すべてのアプリケーションとコンポーネントの保守にかかるコストと労力を最低限に抑えることは、きわめて重要です。保守に関するデメリットを軽減するメカニズムを実装する必要があります。そのためには、たとえば懸念事項を明確に分離してコンポーネント間の疎結合を提供する設計パターンを使用します。リッチ クライアント アプリケーションは、通常はリモート クライアント コンピューターに配置されるので、配置後の操作はサーバーにインストールするアプリケーションよりも困難です。したがって、配置、更新プログラム、修正プログラム、バージョン管理などについても考慮する必要があります。保守容易性の方針を設計する際には、次のガイドラインを考慮します。

- アプリケーションとそのコンポーネントを手動および自動で更新するための、適切なメカニズムを実装します。バージョン管理に関する問題を考慮して、アプリケーションで使用するすべてのコンポーネントに、一貫性があって相互運用可能なバージョンを用意する必要があります。
- アプリケーションを使用する環境に適切な配置手法を選択します。たとえば、一般公開されるアプリケーションのインストール プログラムが必要な場合や、Microsoft System Center などのシステム ツールを使用して閉鎖環境でアプリケーションを配置できる場合があります。
- コンポーネントが疎結合され、必要に応じて置き換えられるようにアプリケーションを設計します。このように設計すると、要件、実行時のシナリオ、およびユーザー個人の要件や設定に応じて各コンポーネントを変更できるようになります。また、コンポーネント間やレイヤー間の依存関係が最小限になるように設計することで、アプリケーション、個別のレイヤー、または個別のコンポーネントを必要に応じてさまざまなシナリオで使用できるようになります。
- アプリケーションのデバッグや実行時の問題解決においてアプリケーションが管理者や開発者の役に立つように、必要に応じてログ記録と監査を実装します。

保守容易性の詳細については、第 16 章「品質特性」を参照してください。

プレゼンテーション レイヤー

プレゼンテーション レイヤーは、アプリケーションの中でもユーザーが目にして操作する部分なので、多くの要件を満たす必要があります。これらの要件には、ユーザビリティ、パフォーマンス、設計、対話性などの一般的な要素が含まれます。ユーザー エクスペリエンスが不適切だと、他のすべての点で適切に機能していても、アプリケーションに深刻な悪影響が及びます。ユーザー エクスペリエンスはアプリケーション アーキテクチャのさまざまな要素の影響を受けるので、魅力的で直感的なユーザー エクスペリエンスを最初からサポートするようにアプリケーションを設計することが重要です。アプリケーションのプレゼンテーション機能を設計する際には、次のガイドラインを考慮します。

- ユーザー操作を管理するロジックを、UI やユーザーが操作するデータから (場合によっては、Separated Presentation のスタイルを適用して) 分離する方法を確認します。このようにすると、アプリケーションの部分的な更新が容易になり、開発者とデザイナーがコンポーネントの作業を個別に行えるようになるので、テスト容易性が向上します。
- 柔軟で簡単に更新できるコマンドとナビゲーションに関する方針とメカニズムを実装します。Command、Publish/Subscribe、Observer などの既知の設計パターンを実装して、コマンドとナビゲーションをアプリケーションのコンポーネントから分離し、テスト容易性を向上することを検討します。
- データの表示、特に表形式データや複数行データの表示には、できる限りデータ バインド機能を使用します。データ バインド機能を使用すると、必要なコードが減少するので、開発が簡略化され、コード エラーが減少します。また、異なるビューやフォームのデータを自動的に同期することもできます。ユーザーがデータを更新できるようにする必要がある場合は、両方向のバインドを使用します。
- Office ドキュメント スタイルのインターフェイスでドキュメントを表示する方法や、ドキュメントのコンテンツや HTML を他の UI 要素で表示するタイミングを検討します。ドキュメントに含まれていることがある無効なコンテンツや悪意のあるコンテンツからユーザーが保護されるようにします。
- アプリケーションの UI を国際化できるようにし、アプリケーションを使用する可能性のあるすべての地域と文化のシナリオに合わせてローカライズできるようにします。この作業では、言語、テキストの方向、およびコンテンツのレイアウトをユーザーの文化の構成や自動検出に基づいて変更します。また、アクセシビリティとナビゲーションを適切にサポートします。

プレゼンテーション レイヤーの設計に関する考慮事項の詳細については、第 6 章「プレゼンテーション レイヤーのガイドライン」を参照してください。

状態管理

状態管理は、プロセスのコンポーネント、操作、またはステップの状態を表すデータの維持に関する領域です。状態データには、ユーザー設定、構成情報、ワークフローの情報、ビジネス ルールの値、UI に表示されるデータなどがあります。アプリケーションでは、状態データを保存し、必要に応じてアクセスし、競合、再起動、および接続状態の変更を処理できるようにする必要があります。状態管理の方針を設計する際には、次のガイドラインを考慮します。

- アプリケーションで格納する必要がある状態情報を決定します。このような状態情報には、サイズの推定値、変更の頻度、データの再作成や再取得にかかる処理コストやオーバーヘッドのコストなどがあります。また、選択した状態管理メカニズムでは適切なサポートを提供できる必要があります。
- 状態データが大量にある場合は、ローカル ディスク ベースのメカニズムを使用してデータを格納することを検討します。起動時にアプリケーションでデータにアクセスできる必要がある場合は、分離ストレージやディスク ファイルなどの永続的なメカニズムを使用します。
- 機密情報を格納する場合は、暗号化やデジタル署名を使用して適切なレベルの保護を実装します。
- 状態情報を管理する必要がある粒度を検討します。たとえば、アプリケーションの全ユーザーに適用する状態情報と、特定のユーザーや役割だけに適用する状態情報を決定します。

ワークフロー

一部のリッチ クライアント アプリケーションでは、多段階の操作やウィザード スタイルの UI 要素を可能にするビュー フローやワークフローをサポートする必要があります。別のコンポーネントやカスタム ソリューションを使用してこれらの機能を実装することも、Windows Workflow Foundation (WF) などのフレームワークを使用することもできます。ワークフローの方針を設計する際には、次のガイドラインを考慮します。

- 多段階のプロセスや実行時間が長いプロセスを伴う操作のビジネス コンポーネントでは、ワークフローを使用します。個別のコンポーネントを作成して、ワークフローやビュー フローのタスクを実装することを検討します。このようにすると、依存関係が軽減され、要件の変更に応じて簡単にコンポーネントを交換できるようになります。
- ワークフローや ビュー フローの要件が単純な場合、通常はユースケース コントローラー (Use Case Controller) やビュー フロー (View Flow) などの既知のパターンに基づくカスタム コードを使用すれ

ば十分に対応できます。ワークフローやビュー フローの要件が複雑な場合は、WF などのワークフロー エンジンの使用を検討します。

- ワークフローでエラーのキャプチャ、管理、および表示を行う方法を検討します。また、部分的に完了したタスクの処理方法や、障害から回復してタスクを続行できるのか、またはプロセスを再起動する必要があるのかということについても検討します。

ワークフロー コンポーネントの詳細については、第 14 章「ワークフロー コンポーネントの設計」を参照してください。

セキュリティに関する考慮事項

セキュリティにはさまざまな要因が含まれ、あらゆる種類のアプリケーションに不可欠です。リッチ クライアント アプリケーションの設計と実装を行う際には、セキュリティを考慮する必要があります。また、リッチ クライアント アプリケーションがビジネス アプリケーションのプレゼンテーション レイヤーとして機能する場合は、アプリケーションの他のレイヤーを保護してセキュリティを確保する役割を果たす必要があります。セキュリティの問題には、機密データの保護、ユーザーの認証と承認、悪意のあるコードやユーザーによる攻撃からの保護、イベントとユーザー操作のログ記録と監査など、さまざまな懸念事項が関連しています。セキュリティの方針を設計する際には、次のガイドラインを考慮します。

- リッチ クライアント アプリケーションの同一インスタンスで複数のユーザーをサポートするなど、ユーザーを認証するための適切なテクノロジと手法を決定します。ユーザーのログオン方法とそのタイミング、異なるアクセス許可 (管理者、標準ユーザーなど) を使用するさまざまな種類のユーザー (さまざまな役割) をサポートする必要があるかどうか、およびログオンの成功と失敗の記録方法を検討する必要があります。オフラインまたは非接続型の認証が必要な場合は、このような認証に関する要件も考慮します。
- ユーザーが同じ資格情報や ID を使用して複数のアプリケーションにアクセスできるようにする必要がある場合は、Windows 統合認証または統合された認証ソリューションの使用を検討します。Windows 統合認証を使用できない場合でも、統合された認証のサポートを提供する外部機関を使用できることがあります。外部機関を使用できない場合は、証明書ベースのシステムを使用するか、独自のカスタム ソリューションを作成することを検討します。
- ユーザーによる入力と、サービスや他のアプリケーション インターフェイスなどのソースからの入力の両方を検証するための要件を検討します。カスタム検証メカニズムを作成しなければならない場合も

あれば、使用しているテクノロジの検証機能を利用できる場合もあります。Microsoft Visual Studio® の Windows フォーム開発環境には、検証コントロールが用意されています。また、Enterprise Library の Validation Application Block など、UI とビジネス レイヤーに包括的な検証機能を提供するサード パーティ製の検証フレームワークの使用も検討します。選択する検証方法にかかわらず、データが信頼境界を越えるときには必ず検証する必要があることを覚えておいてください。

- アプリケーションやリソース (アプリケーションで使用されるファイル、キャッシュ、ドキュメントなど) に格納されたデータの保護方法を検討します。機密データが公開される可能性がある場合は暗号化し、デジタル署名を使用した改ざんの防止策を検討します。アプリケーションのセキュリティを最大限に高めるために、メモリに格納された揮発性情報を暗号化することを検討します。また、ネットワークや通信チャネル経由でアプリケーションから送信される機密情報の保護に留意します。
- アプリケーションの監査とログ記録の実装方法、およびログに記録する情報を検討します。暗号化を使用してログに記録された機密情報を保護し、改ざんに対して脆弱な最も機密性の高い情報については、必要に応じてデジタル署名を使用して保護します。

データ処理に関する考慮事項

アプリケーション データは、サーバー側のアプリケーションから Web サービス経由で提供されることがあります。このデータをクライアント側でキャッシュすると、パフォーマンスが向上し、オフラインで使用できるようになります。リッチ クライアント アプリケーションでは、ローカル データも使用できます。リッチ クライアント アプリケーションで使用するデータは、次の 2 つのカテゴリに分類されます。

- **読み取り専用の参照データ**: これは、頻繁には変更されず、クライアントで参照目的に使用されるデータ (製品カタログなど) です。アプリケーションのパフォーマンスを向上し、オフライン機能を有効にし、早期にデータ検証を行い、通常はアプリケーションのユーザビリティを向上するために、参照データをクライアントに格納して、クライアントとサーバーの間でやり取りされるデータ量を削減します。
- **一時データ**: これは、クライアントでもサーバーでも変更される可能性があるデータです。リッチ クライアント アプリケーションで一時データを操作する際の最も困難な側面は、同じデータを同時に複数のクライアントで変更した場合に発生する、同時実行の問題に対処することです。クライアントでは、クライアント側にある一時データに加えられたすべての変更を追跡し、競合する変更が含まれている可能性のある更新はサーバーで管理します。

データのキャッシュ

多くの場合、リッチ クライアントでは、読み取り専用の参照データと一時データのどちらであってもデータをローカルにキャッシュする必要があります。データをキャッシュすると、アプリケーションのパフォーマンスを向上して、オフラインで処理するのに必要なデータを提供できます。データのキャッシュを有効にするには、リッチ クライアント アプリケーションに、データのキャッシュ処理を透過的に行えるキャッシュ インフラストラクチャを実装する必要があります。一般的なキャッシュの種類は次のとおりです。

- **短期間のデータ キャッシュ**: データは永続的ではないので、アプリケーションをオフラインで実行することはできません。
- **長期間のデータ キャッシュ**: 分離ストレージ、ローカル ファイル システムなど、永続的なメディアにデータをキャッシュするので、サーバーに接続していないときでもアプリケーションを実行できます。リッチ クライアント アプリケーションでは、サーバーと同期されているデータと一時的なデータを区別する必要があります。

データの同時実行制御

同時に複数のクライアントにサービスを提供している場合、特定のクライアントで行われた変更をサーバーと同期する前に、サーバーで保持しているデータが変更されることがあります。これは、データの破損や不整合につながることがあります。したがって、データを同期する際には、すべてのデータの競合が適切に処理され、同期されたデータの一貫性と正確さが確保されるメカニズムを実装する必要があります。データの同時実行を制御する一般的な手法は次のとおりです。

- **ペシミスティック同時実行制御**: ペシミスティック同時実行制御では、データが競合するリスクが高いと想定します。データの競合を防ぐために、1 つのクライアントでデータのロックを維持できるようにするので、そのクライアントによる変更が完了してコミットされるまで、他のクライアントはそのデータにアクセスしたり変更したりできません。このパターンは、Pessimistic Offline Lock パターンとも呼ばれます。
- **オプティミスティック同時実行制御**: オプティミスティック同時実行制御では、データが競合するリスクが低いと想定します。オプティミスティック同時実行制御を使用すると、更新中のデータはクライアントによってロックされません。データの競合を検出するために、元のデータと変更後のデータの両方がサーバーに送信されます。元のデータを現在のデータと比較して、クライアントがデータを取得してからデータが更新されたかどうかを確認します。データが更新されていない場合は変更を適用し、更新

されている場合はデータ競合の例外を生成します。このパターンは、Optimistic Offline Lock パターンとも呼ばれます。

ADO.NET の DataSet を使用すると、オフライン時にクライアントでデータを操作できます。DataSet では、ローカルでデータに加えられた変更を追跡できるので、サーバーとデータを同期してデータの競合を解決するのが容易になります。また、DataSet を使用してさまざまなソースのデータをマージすることもできます。

データ バインド

Windows フォーム、WPF、および Silverlight のデータ バインドでは、双方向のバインドがサポートされています。双方向のバインドを使用すると、データ構造を UI コンポーネントにバインドし、現在のデータ値をユーザーに表示し、ユーザーがデータを編集できるようにし、ユーザーが入力した値を使用して基になるデータを自動更新できます。データ バインドを使用すると、読み取り専用のデータをユーザーに表示し、ユーザーが UI からデータを更新できるようにし、データのマスター/詳細ビューを提供し、ユーザーが複雑な関連データ項目を調べられるようにし、データ列のキー値の代わりにユーザーにわかりやすいデータ項目の名前を UI に表示するルックアップ テーブル機能を提供できます。
データ レイヤーの設計に関する詳細については、第 8 章「データ レイヤーのガイドライン」を参照してください。リッチ クライアント アプリケーションのデータ コンポーネントの設計に関する詳細については、第 15 章「データ レイヤーのコンポーネントの設計」を参照してください。

オフラインまたは不定期に接続する場合の考慮事項

不特定の期間、アプリケーションがネットワーク経由でサービスやデータをすぐに操作できない場合、アプリケーションは不定期に接続している状態だと言えます。不定期に接続するリッチ クライアント アプリケーションでは、ネットワーク リソースに接続していない場合でも処理を実行でき、接続を使用できるようになったらバックグラウンドでネットワーク リソースを更新できます。
不定期に接続するアプリケーションを設計する際には、ネットワーク経由でサービスやデータを操作するときに非同期通信を優先するようにし、ネットワークに配置されたデータやサービスの複雑な操作を最小限に抑えるか、またはなくします。このようにすると、接続が使用できるようになったときに使用する同期メカニズムを実装しやすくなります。
接続が切断されているときに作業できるようにするには、ユーザーがオフライン時に作業を続行するうえで必要なすべてのデータをクライアントが提供するデータ キャッシュ機能をアプリケーションに実装する必要があります。ま

た、アプリケーションで古いデータが使用されないようにする方法も決定する必要があります。一般に、ストア アンド フォワードのメカニズムの設計を検討する必要があります。このメカニズムでは、メッセージが作成され、接続が切断されている間はローカルに格納されて、接続が使用できるようになったときにそれぞれの転送先に転送されます。ストア アンド フォワードの最も一般的な実装が、メッセージ キューです。
不定期に接続するシナリオ向けに設計する際には、次の 2 つの手法を考慮します。

- **データ中心**: データ中心の方針を使用するアプリケーションでは、クライアントのローカルにリレーショナル データベース管理システム (RDBMS) がインストールされています。また、データベース システムの組み込み機能を使用してローカル データの変更をサーバーに反映し、同期処理を制御し、あらゆるデータの競合を検出して解決します。
- **サービス指向**: サービス指向の手法を使用するアプリケーションでは、情報をメッセージに格納し、クライアントがオフラインの間はメッセージをキューに配置します。接続が再確立されたら、キューに配置したメッセージをサーバーに送信します。

テクノロジに関する考慮事項

リッチ クライアント アプリケーションの実装に使用できるテクノロジはいくつかあります。次のガイドラインは、適切な実装テクノロジを選択するのに役立ちます。また、構成と監視についての適切なパターンとシステムの機能の使用に関するガイダンスを提供します。

- **適切な開発テクノロジを選択する**
    - リッチ メディアとグラフィックスを完全にサポートするアプリケーションの場合は、WPF の使用を検討します。
    - 既存の Windows フォームがある場合、または豊富な視覚表現が不要で最小限のハードウェア要件でも動作する必要がある LOB アプリケーションを構築している場合は、Windows フォームの使用を検討します。
    - Web サーバーからダウンロードしてブラウザーで実行するアプリケーションの場合は、XBAP の使用を検討します。
    - 主にドキュメント ベースのアプリケーションやレポート作成に使用するアプリケーションの場合は、OBA の使用を検討します。

- **アプリケーションの設計と実装に役立つ patterns & practices ソリューションの資産について調査する**
    - Windows フォームを使用して複合インターフェイスを設計する場合は、patterns & practices の Smart Client Software Factory の使用を検討します。
    - WPF または Silverlight、あるいはその両方を使用する場合や、複数の画面、機能豊富で柔軟なユーザー操作とデータの視覚表現、および役割に応じた動作を通常備えたモジュール形式のアプリケーションを開発する必要がある場合は、Composite Client Application Guidance の使用を検討します。
    - 例外ハンドル、キャッシュ、検証などの横断的関心事のソリューションを実装するのに役立つ Enterprise Library の使用を検討します。
- **WPF を使用する場合は次のガイドラインを考慮する**
    - UI ロジックの単体テストを実行できるようにし、アプリケーションのスキンを変更しやすくするために、プレゼンテーション モデル (Presentation Model) パターンまたはビュー モデル (View Model) パターンを実装することを検討します。
    - WPF を使用すると、既存のコントロールの実装に追加の動作をアタッチできます。コントロールをサブクラス化する代わりに、この手法を使用します。
- **リモートでの管理と監視をサポートする場合は次のガイドラインを考慮する**
    - アプリケーションの構成にグループ ポリシーのオーバーライドを実装することを検討します。これは、Certified for Windows ロゴの要件を満たすうえで必要です。
    - SNMP や WMI などのテクノロジを使用して、例外や正常性状態を公開することを検討します。

配置に関する考慮事項

リッチ クライアント アプリケーションの配置に関するオプションはいくつかあります。たとえば、データを含めすべてのアプリケーション ロジックがクライアント コンピューターに配置されるスタンドアロン アプリケーションを構築する場合があります。また、アプリケーション ロジックがクライアントに配置され、データがデータベース ティアに配置されるクライアント/サーバー型のアプリケーションを構築する場合もあります。さらには、1 台以上のアプリケーション サーバーでアプリケーション ロジックの一部がホストされる n ティア アプリケーションを構築する場合もあります (この場合は複数のオプションがあります)。

スタンドアロンの配置

図 2 に、すべてのアプリケーション ロジックとデータがクライアントに配置されるスタンドアロンの配置を示します。

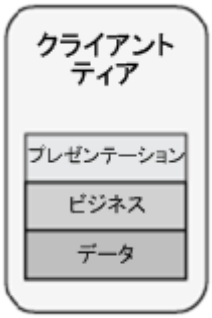


図 36

リッチ クライアント アプリケーションのスタンドアロンの配置

クライアント/サーバーの配置

クライアント/サーバーの配置では、すべてのアプリケーション ロジックはクライアントに配置され、データはデータベース サーバーに配置されます (図 3 参照)。

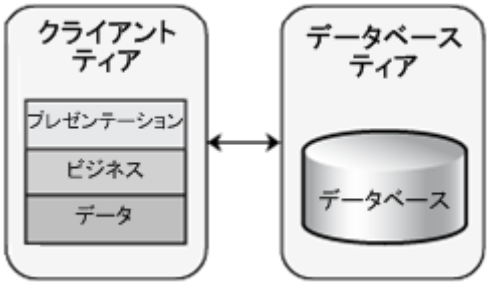


図 37

リッチ クライアント アプリケーションのクライアント/サーバーの配置

n ティアの配置

n ティアの配置では、プレゼンテーション ロジックとビジネス ロジックをクライアントに配置するか、プレゼンテーション ロジックだけをクライアントに配置できます。図 4 に、プレゼンテーション ロジックとビジネス ロジックをクライアントに配置した場合を示します。

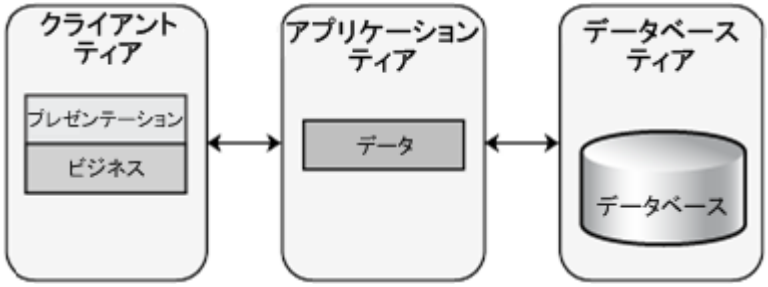


図 38

ビジネス レイヤーをクライアント ティアに配置した n ティアの配置

図 5 に、ビジネス ロジックとデータ アクセス ロジックをアプリケーション サーバーに配置した場合を示します。

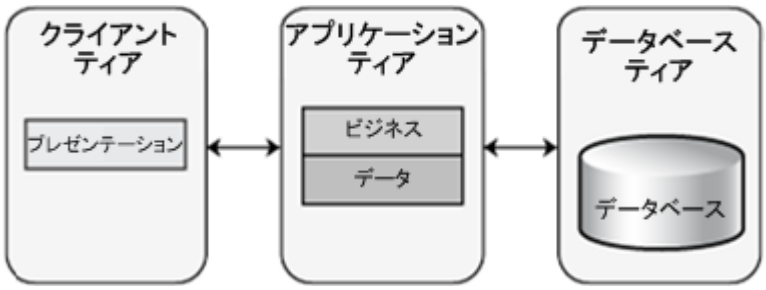


図 39

ビジネス レイヤーをアプリケーション ティアに配置した n ティアの配置

配置に関する方針の詳細については、第 19 章「物理ティアと配置」を参照してください。

配置に関するテクノロジ

物理コンピューターにリッチ クライアント アプリケーションを配置する際には、いくつかのオプションがあります。各オプションには特定のメリットとデメリットがあり、これらのオプションを調査して、アプリケーションを実行する配置先の環境に適したオプションを選択する必要があります。オプションは次のとおりです。

- **ClickOnce の配置**: この手法では、ユーザーによる操作はほとんど必要がなく、自動更新を利用することが可能で、開発者の労力もほとんど必要ありません。しかし、ClickOnce の配置は、大きなソリューションの構成要素ではない、単一ソリューションの配置にしか使用できません。追加のファイルやレジストリ キーは配置できません。また、ユーザーと対話してインストールを構成することも、ブランド化されたインストールを提供することもできません。
- **XCOPY 配置**: レジストリ設定とコンポーネントの登録のどちらも必要ない場合は、実行可能ファイルをクライアント コンピューターのハード ディスクに直接コピーできます。

- **Windows インストーラー (.MSI) パッケージ**: これは、コンポーネント、リソース、レジストリ設定など、アプリケーションに必要な成果物をインストールできる包括的なセットアップ プログラムです。ユーザーが MSI パッケージを各自でインストールするには、管理者特権が必要です。Microsoft System Center Configuration Manager など、企業環境でアプリケーションを配布するためのソリューションも提供されています。
- **XBAP パッケージ**: アプリケーションをブラウザー経由でダウンロードし、コンピューター上の制約付きセキュリティ環境で実行します。更新プログラムは、自動的にクライアントに適用されます。

関連する設計パターン

次の表に示すように、主要なパターンは、通信、構成、構成管理、例外管理、プレゼンテーション、状態管理、ワークフローなどのカテゴリに分類されます。各カテゴリの設計を行う際には、これらのパターンの使用を検討します。

カテゴリ	関連するパターン
通信	**Asynchronous Callback**: 実行時間が長いタスクを、バックグラウンドで実行される独立したスレッドで実行し、タスクが完了したときにスレッドがコールバックする機能を提供します。 **Gateway**: コンシューマーが外部システムのインターフェイスを認識する必要がなくなるように、一般的な抽象インターフェイスを通じた外部システムへのアクセスを提供します。 **Service Locator**: 分散サービス オブジェクトの検索処理を一元化し、一元的に制御する場所を提供し、重複する検索処理を削減するキャッシュとして機能します。 **Service Agent and Proxy**: 実際のアクセス先コンポーネントやサービスにアクセスしていないことを認識することなく、コンシューマーとなるアプリケーションで使用できるコンポーネントを実装します。このコンポーネントでは、呼び出しをリモート コンポーネントやリモート サービスに渡し、結果をコンシューマーとなるアプリケーションに返します。プロキシでは、(通常、ASMX サービスや WCF サービスを使用しているときに) 他のリモート コンポーネントとの通信を抽象化します。 **Service Interface**: 他のシステムがサービスとの通信に使用できる、プログラマティック インターフェイスです。

構成	**Composite View**: 個々のビューを複合ビューに統合します。 **Template View**: 共通のテンプレート ビューを実装し、このテンプレート ビューを使用してビューを取得または構成します。 **Two-Step View**: モデル データを、具体的な形式を指定しない論理表現に変換してから、その論理表現を、必要な実際の形式に変換します。 **View Helper**: ビジネス データの処理機能をヘルパー クラスに委任します。
構成管理	**Provider**: あらゆるカスタム実装がシームレスにプラグインされるようにするために、クライアント API とは異なる API を公開するコンポーネントを実装します。
例外管理	**Exception Shielding**: 例外が発生したときに、サービスの内部実装に関する情報が公開されないようにします。
プレゼンテーション	**Application Controller**: すべてのフロー ロジックが含まれ、Model と連携して適切な View を表示する他の Controller によって使用されるオブジェクトです。 **Model-View-Presenter**: 要求処理を 3 つの役割に分割します。View ではユーザーの入力を処理し、Model ではアプリケーション データとビジネス ロジックを管理し、Presenter ではプレゼンテーション ロジックを管理して View と Model 間の通信を調整します。 **Model-View-ViewModel**: 現在の UI 開発プラットフォーム (View が従来のように開発者によって作成されるのではなく、デザイナーによって作成されます) に合わせてカスタマイズされた Model-View-Controller (MVC) の変化形です。 **Presentation Model**: ユーザー インターフェイスの視覚的な表示に関する役割と、プレゼンテーションの状態や動作に関する役割を、ビュー、プレゼンテーション モデルという別々のクラスに分離します。ビュー クラスは、ユーザー インターフェイスのコントロールを管理し、UI に固有の視覚的な状態や動作をカプセル化します。プレゼンテーション モデル クラスは、プレゼンテーションの状態や動作をカプセル化し、基になるモデルのファサードとして機能します。
状態管理	**Context Object**: 現在の実行コンテキストの管理に使用するオブジェクトです。
ワークフロー	**View Flow**: あるビューから別のビューへのナビゲーションをアプリケーションや環境の状態に基づいて管理し、アプリケーションの適切な動作に必要な条件と

	制限を管理します。 **Work Flow**: 複雑なプロセス指向アプリケーションに含まれるコントロールのフローを事前に定義された方法で管理すると同時に、要求のルーティングを変更できる判断と分岐の構造を使用して、動的にルーティングを変更できるようにします。

Template View、Transform View、および Two-Step View の各パターンの詳細については、Martin Fowler 著『エンタープライズ アプリケーション アーキテクチャ パターン』(翔泳社、2005 年) を参照するか、http://martinfowler.com/eaaCatalog (英語) を参照してください。

Provider パターンの詳細については、「Provider Model Design Pattern and Specification, Part 1」(http://msdn.microsoft.com/en-us/library/ms972319.aspx、英語) を参照してください。

Asynchronous Callback パターンの詳細については、「Creating a Simplified Asynchronous Call Pattern for Windows Forms Applications」(http://msdn.microsoft.com/en-us/library/ms996483.aspx、英語) を参照してください。

Service Interface パターンの詳細については、「サービス インターフェイス」(http://msdn.microsoft.com/ja-jp/library/ms998421.aspx) を参照してください。

Exception Shielding パターンの詳細については、「Useful Patterns for Services」(http://msdn.microsoft.com/en-us/library/cc304800.aspx、英語) を参照してください。

Composite View パターンの詳細については、「Patterns in the Composite Application Library」(http://msdn.microsoft.com/en-us/library/dd458924.aspx、英語) を参照してください。

Presentation Model パターンの詳細については、「Presentation Model」(http://msdn.microsoft.com/en-us/library/dd458863.aspx、英語) を参照してください。

関連情報

Web リソースに簡単にアクセスするには、http://www.microsoft.com/architectureguide (英語) でオンライン版の参考文献を参照してください。

- 複合アプリケーションの構築の詳細については、「Composite Client Application Guidance」(http://msdn.microsoft.com/en-us/library/cc707819.aspx、英語) を参照してください。

- リッチ クライアント アプリケーションとスマート クライアント アプリケーションの設計に関する詳細については、「Smart Client Architecture and Design Guide」(http://msdn.microsoft.com/en-us/library/ms998506.aspx、英語) を参照してください。
- キャッシュ アーキテクチャの詳細については、「Caching Architecture Guide for .NET Framework Applications」(http://msdn.microsoft.com/en-us/library/ee817645.aspx、英語) を参照してください。
- 配置シナリオと考慮事項の詳細については、「Deploying .NET Framework-based Applications」(http://msdn.microsoft.com/en-us/library/ee817655.aspx、英語) を参照してください。

23

リッチ インターネット アプリケーションの設計

概要

この章では、リッチ インターネット アプリケーション (RIA) の主要なシナリオについて説明し、RIA で使用されるコンポーネントについて説明します。また、RIA の設計に関する重要な考慮事項についても紹介します。具体的には、パフォーマンス、セキュリティ、および配置のガイドラインに加え、RIA を設計する際の主要なパターンとテクノロジに関する考慮事項についても説明します。

RIA では、リッチ グラフィックスとストリーミング メディアのシナリオをサポートするだけでなく、Web アプリケーションの配置と保守容易性に関して多くのメリットを提供します。RIA は、Asynchronous JavaScript and XML (AJAX) などのブラウザー コードを活用する拡張機能内ではなく、Microsoft® Silverlight® などのブラウザー プラグイン内で実行できます。一般的な RIA では、プレゼンテーションを処理するクライアント側アプリケーションと組み合わせて Web インフラストラクチャを使用します。プラグインでは、リッチ グラフィックスをサポートするライブラリのルーチン、およびセキュリティを確保するためにローカル リソースへのアクセスを制限するコンテナーを提供します。RIA では、通常の Web アプリケーションで実行できるコードよりも大規模で複雑なクライアント側コードを実行できるので、Web サーバーの負荷を削減できます。図 1 に RIA の一般的な構造を示します。

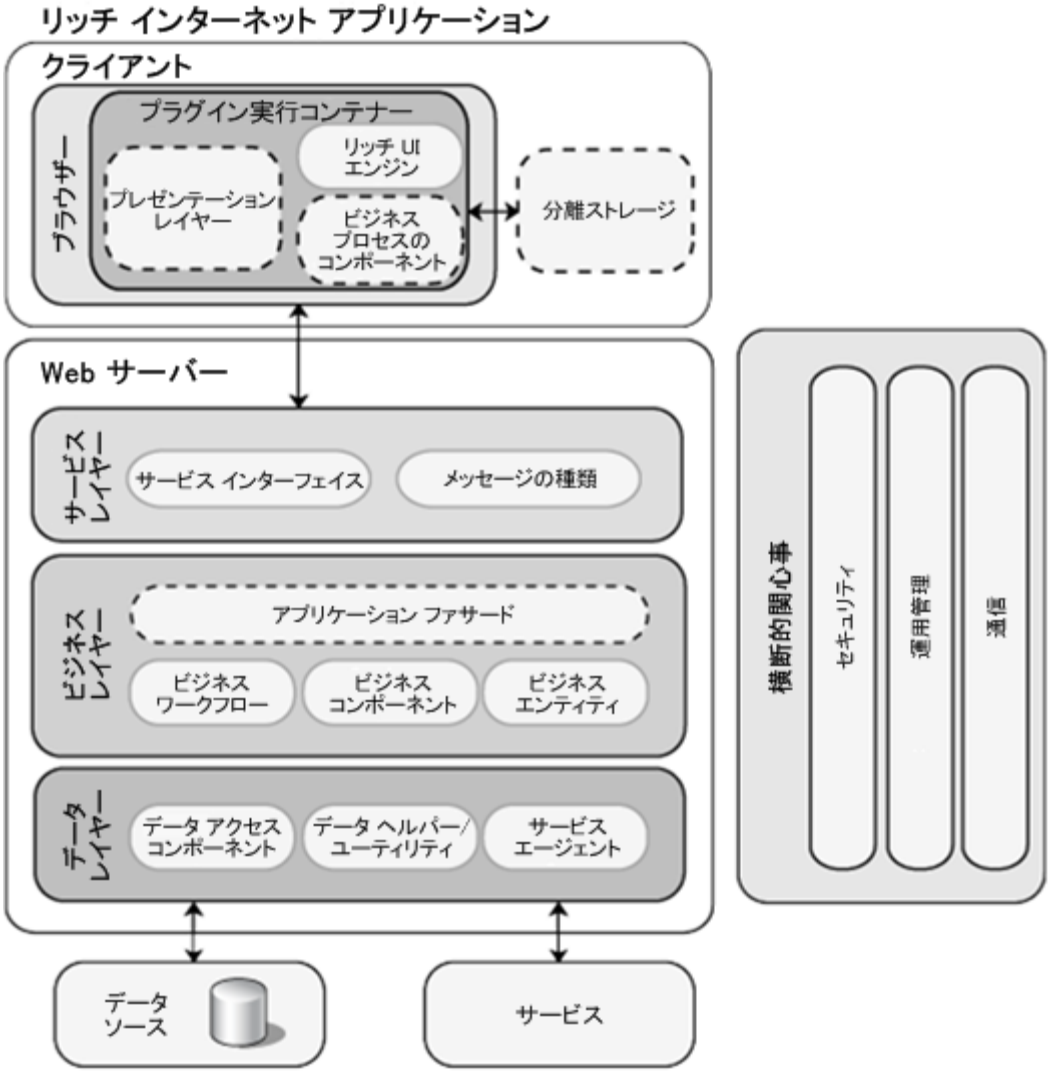


図 40

一般的な RIA のアーキテクチャ (破線はオプション コンポーネント)

一般的なリッチ インターネット アプリケーションは、プレゼンテーション レイヤー、ビジネス レイヤー、およびデータ レイヤーの 3 つのレイヤーに分割できます。通常、プレゼンテーション レイヤーには UI コンポーネントとプレゼンテーション ロジック コンポーネントが、ビジネス レイヤーにはビジネス ロジック コンポーネント、ビジネス ワークフロー コンポーネント、およびビジネス エンティティ コンポーネントが、データ レイヤーにはデータ アクセス コンポーネントとサービス エージェント コンポーネントが含まれます。

RIA では、ビジネス プロセスの一部がクライアントに移動されることがよくあり、場合によってはデータ アクセス コードもクライアントに移動されます。そのため、アプリケーションのシナリオによっては、ビジネス レイヤーと

データ レイヤーの機能の一部または全部がクライアントに含まれていることがあります。図 1 は、ビジネス プロセスの一部をクライアントに実装する通常の方法を示しています。

RIA は、バックエンドのビジネス サービスを表示する軽量なインターフェイスになることもあれば、ほとんどの処理を内部で実行し、情報を使用および送信する際にのみ、バックエンド サービスと通信する複雑なアプリケーションになることもあります。そのため、RIA にはさまざまな設計と実装があります。ただし、プレゼンテーション レイヤーとプレゼンテーション レイヤーがバックエンド サービスと通信する方法に関しては、適切にアーキテクチャを設計するための一般的な手法がいくつかあります。このような手法のほとんどは、分離したコンポーネントをアプリケーションで使用しやすくすることで、依存関係を軽減し、保守とテストを容易にし、再利用性を高める既存の設計パターンに基づいています。
レイヤー型の設計の詳細については、第 5 章「レイヤー型アプリケーションのガイドライン」を参照してください。各レイヤーに適したコンポーネントの詳細については、第 10 章「コンポーネントのガイドライン」を参照してください。

設計に関する一般的な考慮事項

次のガイドラインでは、RIA を設計する際に考慮する必要があるいくつかの側面についての情報を提供します。アプリケーションが要件を満たし、RIA に共通するシナリオで効率的に機能するためには、次のガイドラインに従います。

- **ユーザー、機能豊富なインターフェイス、および配置しやすさに基づいて RIA を選択する**: 重要なユーザーが、RIA をサポートするブラウザーを使用している場合は、RIA の設計を検討します。重要なユーザーの一部が RIA をサポートしていないブラウザーを使用している場合は、ブラウザーの選択肢を RIA をサポートしているバージョンに限定できるかどうか検討します。ブラウザーの選択肢を変更できない場合は、そのことによって失うユーザー数が、別の種類のアプリケーション (AJAX を使用した Web アプリケーションなど) を選択する必要があるほど多いかどうか検討します。RIA の場合、クライアントで信頼できるネットワーク接続が使用できることを想定すると、配置と保守は Web アプリケーションと同じくらい簡単に行えます。RIA は、基本的な HTML よりも高度な視覚表現が必要な Web ベースのシナリオに適しています。高度な機能を備え、コードのカスタマイズが可能な Web アプリケーションと比べて、RIA は動作の一貫性を維持しやすく、サポートするさまざまなブラウザー間での必

要なテストが少なく済みます。RIA は、ストリーミング メディアのアプリケーションにも最適です。ただし、非常に複雑な複数ページにわたる UI にはあまり適していません。

- **サービスを利用する Web インフラストラクチャを使用するように設計する**: RIA には、Web アプリケーションと同様のインフラストラクチャが必要です。通常、RIA ではクライアントで処理を実行しますが、たとえばデータをデータベースに保存するために、ネットワークに接続している他のサービスとも通信します。
- **クライアントの処理能力を利用するように設計する**: RIA はクライアント コンピューターで実行されるので、クライアント コンピューターで提供されるすべての処理能力を利用できます。ユーザー エクスペリエンスを向上するために、できる限り多くの機能をクライアントに移動することを検討します。ただし、クライアントのロジックは回避される可能性があるので、機密なビジネス ルールはサーバーで実行する必要があります。
- **ブラウザーのサンドボックス内で実行するように設計する**: RIA では、既定で高レベルのセキュリティが設定されているので、カメラ、ハードウェアのビデオ アクセラレータなど、コンピューター上のすべてのリソースにアクセスできるわけではありません。ローカルファイル システムへのアクセスは制限されています。ローカル記憶域は使用できますが、使用できる領域には制限があります。
- **UI の要件の複雑さを特定する**: UI の複雑さを検討します。RIA は、1 つの画面ですべての操作を実行できる場合に最適な状態で動作します。実装を複数の画面に拡張することもできますが、そのためには追加のコードが必要になったり、画面のフローについて考慮する必要があります。すべての手順を最初からやり直さなくても、ユーザーが簡単に画面の間を移動および一時停止して、画面のフローの適切な位置に戻れるようにする必要があります。複数ページの UI の場合は、ディープ リンクの設定の手法を使用します。また、Uniform Resource Locator (URL)、履歴リスト、およびブラウザーの [戻る] ボタンと [進む] ボタンを操作して、ユーザーが画面から画面に移動する際に混乱が生じないようにします。
- **アプリケーションのパフォーマンスや応答性が向上するシナリオを使用する**: 一般的なアプリケーションのシナリオを一覧にして確認し、アプリケーションのコンポーネントを分割して読み込む方法、データのキャッシュ方法、およびビジネス ロジックをクライアントに移動する方法を決定します。アプリケーションのダウンロードと起動にかかる時間を短縮するために、機能を個別のダウンロード可能なコンポーネントに分割します。
- **プラグインがインストールされていないシナリオを考慮して設計する**: RIA にはブラウザー プラグインが必要なので、処理を中断することなくプラグインをインストールできるように設計する必要があります。クライアントがプラグインにアクセスできるかどうか、必要なアクセス許可を持っているかどうか、

およびプラグインをインストールする必要があるかどうかを考慮します。インストール プロセスをどのように制御できるか検討します。有用なエラー メッセージを表示したり、代わりの Web UI を表示したりすることで、ユーザーがプラグインをインストールできない場合に備えます。

RIA 固有の上記ガイドラインに加えて、リッチ クライアント アプリケーション全般 (モバイル リッチ クライアント アプリケーションを含む) に関するより一般的なガイドラインも考慮します。具体的には、プレゼンテーション ロジックとインターフェイスの実装の分離、プレゼンテーション タスクとプレゼンテーション フローの特定、ビジネス ルールやインターフェイスと無関係な処理の分離、共通のプレゼンテーション ロジックの再利用、クライアントとそのクライアントで使用するすべてのリモート サービスの疎結合、他のレイヤーのオブジェクトとの密結合の回避、リモート レイヤーにアクセスする際のラウンド トリップの回数削減などのガイドラインを考慮します。詳細については、第 22 章「リッチ クライアント アプリケーションの設計」を参照してください。

設計に関する具体的な問題

設計を策定する際に考慮する必要がある一般的な問題がいくつかあります。これらの問題は、設計の具体的な領域に分類できます。次のセクションでは、各領域で一般的に発生する問題を解決するのに役立つガイドラインを提供します。

- ビジネス レイヤー
- キャッシュ
- 通信
- 構成
- データ アクセス
- 例外管理
- ログ記録
- メディアとグラフィックス
- モバイル
- 移植性
- プレゼンテーション
- 状態管理
- 検証

ビジネス レイヤー

ほとんどのシナリオでは、RIA は、アプリケーションの外部に配置されたデータや情報にアクセスします。情報の性質はさまざまですが、多くの場合はビジネス システムから抽出されます。パフォーマンスとユーザビリティを最大限に高めるために、一部のビジネス プロセスのタスクをクライアントに配置することを検討します。ビジネス レイヤーとサービス レイヤーの間の通信を設計する際には、次のガイドラインを考慮します。

- アプリケーションで使用するビジネス レイヤーとサービス インターフェイスを特定します。クライアントのビジネス レイヤーでは、サービス エージェントを使用してサービス インターフェイスにアクセスする必要があります。通常、サービスの定義を使用してサービス プロキシを生成すると、サービス エージェントを実装できます。
- ビジネス ロジックに機密情報が含まれていない場合は、一部のビジネス ルールをクライアントに配置して、アプリケーションのパフォーマンスと応答性を向上することを検討します。ビジネス ロジックに機密情報が含まれている場合は、ビジネス ロジックをアプリケーション サーバーに配置する必要があります。
- ビジネス ルールなど、クライアント側の処理を実行するのに必要な情報をクライアントで取得する方法と要件の変化に応じてクライアントで自動的にビジネス ルールを更新する方法を検討します。クライアントの起動時にビジネス レイヤーからビジネス ルールの情報を取得することもできます。
- RIA で UI が表示されないインスタンスを作成できる場合は、そのインスタンスを使用して、ブラウザーでサポートされる柔軟性の低い言語ではなく、より構造化され、強力な、または使い慣れたプログラミング言語 (C# など) を使用するビジネス プロセスを実装することを検討します。
- ビジネス ロジックがクライアントとサーバーで重複している場合、RIA で使用できれば、クライアントとサーバーで同じコード言語を使用します。このようにすると、言語の実装の違いが減少し、ルールの処理方法の一貫性を保ちやすくなります。サーバー側にもクライアント側にも配置できるドメイン モデルは、できる限り同じモデルにする必要があります。
- セキュリティ上の理由から、暗号化されていない機密ビジネス ロジックをクライアントに配置しないようにします。ダウンロードされる XAP ファイルのコードは、簡単に逆コンパイルされる可能性があります。機密ビジネス ロジックは、サーバーに配置し、Web サービスを使用してアクセスするようにします。

ビジネス レイヤーの実装に関する詳細については、第 7 章「ビジネス レイヤーのガイドライン」を参照してください。

キャッシュ

一般に、RIA ではブラウザーの標準的なキャッシュ メカニズムを使用します。リソースを効率的にキャッシュすると、アプリケーションのパフォーマンスが向上します。キャッシュの方針を設計する際には、次のガイドラインを考慮します。

- 大規模なクライアント アプリケーションを小規模で個別にダウンロード可能なコンポーネントに分割すると、これらのコンポーネントをキャッシュしてパフォーマンスを向上し、ネットワークのラウンドトリップの回数を最小限に抑えることができます。できる限り、起動時にアプリケーション全体をダウンロードしてインスタンス化することは避けます。インストール、更新、およびユーザーに関する各シナリオを使用して、アプリケーションのモジュールを分割して読み込む方法を開発します。たとえば、起動時にスタブを読み込んでから、バックグラウンドで追加機能を動的に読み込みます。イベントを使用して、必要になる直前にモジュールを効率的に読み込むことを検討します。
- セッション中に変更される可能性が低いオブジェクトをブラウザーでキャッシュできるようにします。セッション中に変更される情報、または複数のセッションにわたって保持される情報には、専用の RIA のローカル記憶域を使用します。
- 意図しない例外を回避するために、書き込むデータを格納するのに十分な領域が分離ストレージにあることを確認します。ストレージの容量は自動的に増加しないので、ユーザーに容量を増やすよう依頼する必要があります。

キャッシュ方針の設計の詳細については、第 17 章「横断的関心事」を参照してください。

通信

RIA では、ブラウザーの処理がブロックされないように、サービスの非同期呼び出しモデルを使用する必要があります。設計時には、ドメイン間、プロトコル、およびサービス効率に関する問題を考慮する必要があります。RIA のバックグラウンド操作には、できる限り独立したスレッドを使用することを検討します。通信の方針を設計する際には、次のガイドラインを考慮します。

- 実行時間の長い操作がある場合は、バックグラウンド スレッドや非同期実行を使用して UI スレッドのブロックを回避することを検討します。
- RIA と RIA で呼び出されるサービスで、セキュリティ情報を含む互換性のあるバインドが使用されるようにします。サービスを通じて認証する場合は、RIA でサポートされているバインドを使用するようにサービスを設計します。

- 機密情報と通信チャネルを保護するために、インターネット プロトコル セキュリティ (IPSec) や Secure Sockets Layer (SSL) を使用してチャネルをセキュリティで保護し、暗号化を使用してデータを保護し、デジタル署名を使用してデータの改ざんを検出することを検討します。
- RIA クライアントからダウンロード元以外のサーバーにアクセスする必要がある場合は、ドメイン間の構成メカニズムを使用して、他のサーバーやドメインにアクセスできるようにします。
- Windows Communication Foundation (WCF) で二重化メカニズムを使用して、クライアントでポーリングを行うとサーバーに高い負荷がかかる場合はデータをクライアントに配置し、サービスを使用するよりもデータをサーバーに配置した方がはるかに効率的な場合 (中央サーバーを使用してリアルタイムで複数のプレーヤーが参加するゲームなど) はデータをサーバーに配置することを検討します。ただし、ファイアウォールとルーターで一部のポートやプロトコルがブロックされる場合があることに注意してください。詳細については、この章の最後の「関連情報」を参照してください。

サービスの設計に関する詳細については、第 25 章「サービス アプリケーションの設計」を参照してください。通信プロトコルと通信技法の詳細については、第 18 章「通信とメッセージ」を参照してください。

構成

構成を使用すると、アプリケーション全体を再実装または再配置しなくてもより簡単に管理または拡張できるアプリケーションを構築できます。また、多数のモジュールに基づいてアプリケーションを実装して、これらのモジュールに含まれるコンポーネントを疎結合できます。このようにすると、新しいモジュールや新しいバージョンのモジュールを配置してアプリケーションを拡張したり、ユーザーによるカスタマイズやパーソナル化、またはユーザーの役割やタスクに合わせたカスタマイズを簡略化できます。構成の方針を設計する際には、次のガイドラインを考慮します。

- 構成がシナリオに適しているかどうかを評価します。適している場合は、最適な構成モデル パターンを特定します。構成を使用すると、ほとんどまたはまったく変更しなくても、さまざまなシナリオで再利用できるアプリケーションを設計できます。ただし、依存関係が発生して頻繁にアプリケーションを再配置しなければならない設計は避けます。
- 構成が適しているのは、異なるソースの情報や機能を統合するマッシュアップ アプリケーション、またはユーザーがアプリケーションを拡張したりカスタマイズしたりできる場合です。

データ アクセス

RIA では、AJAX クライアントと同じ方法で、Web サーバーからサービス経由でデータを要求します。クライアントでデータを取得したら、データをキャッシュしてパフォーマンスを最大限に高めることができます。データ アクセスの方針を設計する際には、次のガイドラインを考慮します。

- クライアント側でキャッシュを使用して、サーバーへのラウンド トリップの回数を最小限に抑え、より応答性の高い UI を提供します。
- クライアント側ではなくサーバー側でデータをフィルター処理して、ネットワーク経由で送信する必要があるデータ量を削減します。

データ レイヤーの設計に関する詳細については、第 8 章「データ レイヤーのガイドライン」を参照してください。

例外管理

例外管理の方針が堅牢で適切に設計されていると、アプリケーションの設計を簡略化して、セキュリティと管理容易性を向上できます。また、アプリケーションが開発しやすくなり、開発にかかる時間とコストを削減できます。通常、RIA では、エラーの発生時にユーザーに通知する必要があります。また、検証メッセージなど、簡単な UI エラーを除くすべてのエラーと例外については、運用スタッフや監視システムが使用できるようにサーバーでのログ記録を検討する必要があります。さらに、非同期例外の管理や、クライアントとサーバーのコード間で例外を調整することも検討する必要があります。例外管理のメカニズムを設計する際には、次のガイドラインを考慮します。

- 同期例外と非同期例外の両方を考慮して設計します。同期コードの例外をトラップするには、try ブロックと catch ブロックを使用します。非同期のサービス呼び出しの例外ハンドルは、そのような例外専用に設計されたハンドラーに配置します。たとえば、Silverlight では OnError ハンドラーに配置します。
- ハンドルされない例外をキャッチしてハンドルする方法を設計します。RIA のハンドルされない例外は、ブラウザーに渡されます。ハンドルされない例外を使用すると、ユーザーがブラウザーのエラー メッセージを破棄した後でも RIA を実行し続けることができます。できる限り、ユーザーにはわかりやすいエラー メッセージを提供します。実行し続けるとアプリケーション データの整合性に悪影響が及ぶ場合や、アプリケーションの状態がまだ安定しているとユーザーが誤解する可能性がある場合は、プログラムの実行を停止します。
- ハンドルできる内部例外のみをキャッチします。たとえば、null 値を変換する際に発生するデータ変換の例外をキャッチします。例外を使用してビジネス ロジックを制御しないようにします。

- 適切な例外の伝達に関する方針を設計します。たとえば、例外の境界レイヤーへの伝播を許可し、次のレイヤーに渡される前に、必要に応じて境界レイヤーでログ記録や変換が実行されるようにします。
- 重大なエラーと例外のログ記録および通知に関する適切な方針を設計して、機密情報を開示しないようにします。

例外管理方針の設計の詳細については、第 17 章「横断的関心事」を参照してください。

ログ記録

デバッグ目的や監査目的のログ記録は、RIA では困難な場合があります。たとえば、Silverlight アプリケーションではクライアントのファイル システムを使用できないので、クライアントとサーバーの処理は非同期に行われます。クライアント ユーザーのログ ファイルをサーバーのログ ファイルにまとめて、プログラムの実行に関する全体像を把握する必要があります。ログ記録の方針を設計する際には、次のガイドラインを考慮します。

- RIA のログ記録コンポーネントに関する制限事項を考慮します。一部の RIA では、各ユーザーの情報を別個のログ ファイルに記録します。場合によっては、これらのファイルはディスクのさまざまな場所に配置されています。
- クライアントのログを処理する目的でサーバーに転送する方針を決定します。クライアント コンピューターに固有の問題のトラブルシューティングを行う場合は、同じコンピューターを使用している別のユーザーのログを再統合しなければならないこともあります。クライアント コンピューターは複数のユーザーが使用している場合があるので、ログは、コンピューターごとではなく、ユーザーごとに分割します。
- ログ記録に分離ストレージを使用している場合は、容量の上限について、および必要に応じてユーザーにストレージの容量を増やすよう依頼する必要があるかどうかについて検討します。
- 重大なエラーはログに記録します。また、例外の発生もログに記録してログをサーバーに転送できるようにすることを検討します。

メディアとグラフィックス

RIA では、通常の Web アプリケーションよりもはるかに優れたエクスペリエンスと高度なパフォーマンスが提供されます。このため、RIA プラットフォームに組み込まれたメディア機能を調べて活用します。スタンドアロンのメディア プレーヤーでは使用できるが、RIA プラットフォームでは使用できない可能性がある機能に注意します。マルチメディアとグラフィックスを設計する際には、次のガイドラインを考慮します。

- 個別のプレーヤー ユーティリティを呼び出すのではなく、ブラウザーでストリーミング メディアやストリーミング ビデオを使用するように設計します。一般に、変化する帯域幅の問題にシームレスに対処するには、必ず RIA クライアントと組み合わせてアダプティブ ストリーミングを使用する必要があります。
- パフォーマンスを向上するために、メディア オブジェクトをピクセル単位で配置してネイティブ サイズで表示し、最高の描画パフォーマンスを実現するために、ネイティブなベクター グラフィックス エンジンを使用します。
- グラフィックスを非常に多く使用するアプリケーションのプログラムを作成している場合は、RIA でハードウェア アクセラレータが提供されているかどうか確認します。アクセラレータが提供されていない場合は、許容できる描画パフォーマンスのベースラインを作成します。描画パフォーマンスが許容できる範囲を下回る場合は、グラフィックス エンジンの負荷を軽減する計画を検討します。
- 描画領域のサイズに注意します。実際に変化している部分の領域だけを再描画します。重なった領域が不要な場合は削減して、ブレンドを削減します。プロファイルとデバッグの手法 (Silverlight の EnableRedrawRegions = true という設定など) を使用して、再描画される領域を特定します。ぼかしなどの一部の効果を使用すると、領域内のすべてのピクセルが再描画される可能性があることに注意してください。ウィンドウのないコントロールや透明なコントロールを使用する場合も、意図しない再描画やブレンドが発生する可能性があります。

モバイル

RIA では、通常のモバイル アプリケーションよりもはるかに優れたエクスペリエンスが提供されます。このため、使用する RIA プラットフォームに組み込まれているメディア機能を活用します。モバイル デバイスのマルチメディアとグラフィックスを設計する際には、次のガイドラインを考慮します。

- RIA をモバイル クライアントに配置する必要がある場合は、サポート対象のデバイスで RIA のプラグイン実装を使用できるかどうか調べます。非モバイル プラットフォームと比べて、RIA のプラグインの機能が少ないかどうかを確認します。
- 可能な場合は、単一のコードベースや類似したコードベースを使用します。そのうえで、必要に応じて特定のデバイス用にコードを分岐します。
- UI のレイアウトと実装がモバイル デバイスの小さな画面サイズに適していることを確認します。RIA はモバイル デバイス上で動作しますが、Windows Mobile 向けに設計する場合は、デバイスの種類ごとに異なるレイアウト コードを使用してさまざまな画面サイズによる影響を軽減します。

モバイル アプリケーションの実装に関する詳細については、第 24 章「モバイル アプリケーションの設計」を参照してください。

移植性

RIA の主なメリットの 1 つは、異なるブラウザー間、オペレーティング システム間、およびプラットフォーム間におけるコンパイル済みコードの移植性です。同様に、単一ソースのコードベースや類似したコードベースを使用すると、プラットフォームの柔軟性を維持しながら開発と保守にかかる時間とコストを削減できます。移植性に関する設計を行う際には、次のガイドラインを考慮します。

- ネイティブな RIA のコード ライブラリを最大限に活用し、"1 回記述すれば任意の箇所で実行できる" ことを目標に設計します。ただし、全体的なプロジェクトの複雑さや機能のトレードオフのために必要な場合は、コードを分岐します。
- RIA アプリケーションと Web アプリケーションのどちらを作成するかを決める際には、Web アプリケーションではブラウザー間の違いが原因で ASP.NET コードや JavaScript コードの大規模なテストが必要になることを考慮します。RIA アプリケーションの場合、開発者ではなくプラグインの作成者が、さまざまなプラットフォーム間の一貫性を確保する必要があります。このため、プラットフォームとブラウザーの組み合わせごとにテストするコストが大幅に減少します。
- ユーザーが RIA を複数のプラットフォームで実行する場合は、1 つのプラットフォームでしか使用できない機能 (Windows 統合認証など) は使用しないようにします。さまざまなクライアントで使用できる移植可能な RIA のルーチンと機能に基づいて、ソリューションを設計します。
- リッチ クライアント アプリケーションと RIA を開発する場合は、patterns & practices の Composite Client Application Guidance など、両方のプラットフォームに対応できる言語や開発環境の使用を検討します。詳細については、「Composite Client Application Guidance」(http://msdn.microsoft.com/en-us/library/cc707819.aspx、英語) を参照してください。

プレゼンテーション

通常、RIA の実行はブラウザーに限定されているので、多くの場合は単一の一元的なインターフェイスとして設計すると最適な状態で動作します。複数のページを備えたアプリケーションでは、ページ間のリンク方法を検討する必要があります。プレゼンテーションに関する設計を行う際には、次のガイドラインを考慮します。

- Separated Presentation パターンを使用して、アプリケーションの視覚表現とアプリケーションのプレゼンテーション ロジックを分離します。
- データの表示、特に表形式データや複数行データの表示には、できる限りデータ バインド機能を使用します。データ バインド機能を使用すると、必要なコードが減少するので、開発が簡略化され、コード エラーが減少します。また、異なるビューやフォームのデータを自動的に同期することもできます。ユーザーがデータを更新できるようにする必要がある場合は、両方向のバインドを使用します。
- 複数ページの UI の場合は、ディープ リンクの設定の手法を使用することで、アプリケーションの個々のページを一意に識別し、そのページに移動できるようにします。
- ブラウザーの [戻る] ボタンと [進む] ボタンのイベントをトラップして、ページ外への意図しないナビゲーションを回避します。また、通常の Web ページと同様のナビゲーションを実装するために、ブラウザーのアドレス バーのコンテンツを操作する機能や履歴リストの使用を検討します。

プレゼンテーション レイヤーの実装に関する詳細については、第 6 章「プレゼンテーション レイヤーのガイドライン」を参照してください。

状態管理

分離ストレージを使用すると、アプリケーションの状態をクライアントに格納できます。分離ストレージは、状態を複数のユーザー セッション間でローカルで保持したり、キャッシュしたりするのに役立ちます。分離ストレージの管理方法は、ブラウザー キャッシュとは異なります。データを分離ストレージに書き込むアプリケーションでは、書き込んだデータを直接削除するか、ユーザーにデータを削除するよう明示的に通知する必要があります。状態管理を設計する際には、次のガイドラインを考慮します。

- アプリケーションで格納する必要がある状態情報を決定します。このような状態情報には、サイズの推定値、変更の頻度、データの再作成や再取得にかかる処理コストやオーバーヘッドのコストなどがあります。
- 状態をクライアントの分離ストレージに格納して、セッション中や複数のセッション間で状態を保持します。アプリケーションが機能するために必要な状態は、必ずサーバーに保存する必要があります。また、状態をサーバーに保存すると、ユーザーが別のコンピューターからログオンする際に、保存された状態にアクセスすることもできます。
- 複数の RIA インスタンスが起動することは回避できないので、複数の同時セッションを考慮して設計します。状態管理の同時実行を考慮して設計するか、アプリケーションの状態が不適切にならないように複数のセッションを検出するように設計します。

検証

検証を実行するには、クライアントのコードを使用するか、サーバーに配置されたサービスを使用する必要があります。クライアントで複雑な検証が必要な場合は、検証ロジックを別個のダウンロード可能なアセンブリに分離します。このようにすると、検証規則を管理しやすくなります。検証に関する設計を行う際には、次のガイドラインを考慮します。

- ユーザー エクスペリエンスを最大限に高めるためにクライアント側の検証を使用しますが、セキュリティを確保するために必ずサーバー側の検証も使用します。一般に、クライアントが制御するすべてのデータに悪意があることを前提とします。サーバーでは、サーバーに送信されたすべてのデータを再検証する必要があります。クエリ文字列、Cookie、HTML コントロールなど、すべてのソースからの入力を検証するように設計します。
- データを制限、拒否、および一部削除する検証メカニズムを設計します。また、入力の長さ、範囲、形式、および型を検証します。サーバーの信頼境界を特定し、信頼境界を越えるデータを検証します。
- 分離ストレージを使用してクライアント固有の検証規則を保持することを検討します。サーバーのリソースにアクセスする必要がある規則の場合は、サーバー上で検証を実行するサービスを呼び出すことで効率が高まるかどうかを評価します。
- 変更される可能性のあるクライアント側の検証コードが大量にある場合は、このようなコードを別個のダウンロード可能なモジュールに配置して、RIA アプリケーション全体を再度ダウンロードしなくても簡単に置き換えられるようにします。

検証の技法に関する詳細については、第 17 章「横断的関心事」を参照してください。

セキュリティに関する考慮事項

セキュリティにはさまざまな要素が含まれ、あらゆる種類のアプリケーションに不可欠です。リッチ インターネット アプリケーションの設計と実装を行う際には、セキュリティを考慮する必要があります。また、リッチ インターネット アプリケーションがビジネス アプリケーションのプレゼンテーション レイヤーとして機能する場合は、アプリケーションの他のレイヤーを保護してセキュリティを確保する役割を果たす必要があります。セキュリティの問題には、機密データの保護、ユーザーの認証と承認、悪意のあるコードやユーザーによる攻撃からの保護、イベントとユーザー操作のログ記録と監査など、さまざまな懸念事項が関連しています。

セキュリティの方針を設計する際には、次のガイドラインを考慮します。

- ユーザーを認証するための適切なテクノロジと手法を決定します。ユーザーのログオン方法とそのタイミング、異なるアクセス許可 (管理者、標準ユーザーなど) を使用するさまざまな種類のユーザー (さまざまな役割) をサポートする必要があるかどうか、およびログオンの成功と失敗の記録方法を検討する必要があります。
- ユーザーが同じ資格情報や ID を使用して複数のアプリケーションにアクセスできるようにする必要がある場合は、Windows 統合認証、シングル サインオン (SSO) のメカニズム、または統合された認証ソリューションの使用を検討します。Windows 統合認証を使用できない場合でも、統合された認証のサポートを提供する外部機関を使用できることがあります。外部機関を使用できない場合は、証明書ベースのシステムを使用するか、独自のカスタム ソリューションを作成することを検討します。
- ユーザーによる入力と、サービスや他のアプリケーション インターフェイスなどのソースからの入力の両方を検証するための要件を検討します。カスタム検証メカニズムを作成しなければならない場合もあれば、使用している UI テクノロジの検証機能を利用できる場合もあります。
- アプリケーションの監査とログ記録の実装方法、およびログに記録する情報を検討します。暗号化を使用してログに記録された機密情報を保護し、改ざんに対して脆弱な最も機密性の高い情報については、必要に応じてデジタル署名を使用して保護します。

データ処理に関する考慮事項

通常、アプリケーション データにはネットワークに接続しているサービスを経由してアクセスします。このデータをクライアント側でキャッシュすると、パフォーマンスが向上し、オフラインで使用できるようになります。通常、アプリケーション データは次の 2 つのカテゴリに分類されます。

- **読み取り専用の参照データ**: これは、頻繁には変更されず、クライアントで参照目的で使用されるデータ (製品カタログなど) です。アプリケーションのパフォーマンスを向上し、オフライン機能を有効にし、早期にデータ検証を行い、通常はアプリケーションのユーザビリティを向上するために、参照データをクライアントに格納して、クライアントとサーバーの間でやり取りされるデータ量を削減します。
- **一時データ**: これは、クライアントでもサーバーでも変更される可能性があるデータです。リッチ インターネット アプリケーションで一時データを操作する際の最も困難な側面は、同じデータを同時に複数のクライアントで変更した場合に発生する、同時実行の問題に対処することです。クライアントでは、

クライアント側にある一時データに加えられたすべての変更を追跡し、競合する変更が含まれている可能性のある更新はサーバーで管理します。

テクノロジに関する考慮事項

次のガイドラインでは、Silverlight と Microsoft Windows Communication Foundation (WCF) について説明し、これらのテクノロジに関する具体的なガイダンスを提供します。現時点では、最新バージョンは WCF 3.5 と Silverlight 3.0 です。このガイドラインは、適切なテクノロジを選択して実装するのに役立ちます。

バージョンとターゲット プラットフォーム

- このガイドの公開時には、Silverlight for Mobile はリリースが予定された製品であり、開発中の製品でした。
- 現在、Silverlight ではプラグインを使用して、Safari、Firefox、および Microsoft Internet Explorer の各ブラウザーをサポートしています。これらのブラウザーを通じて、現在、Silverlight では Mac と Windows をサポートしています。また、2008 年には Windows Mobile のサポートも発表されました。Moonlight と呼ばれるオープン ソースの Silverlight の実装では、Linux システムと Unix X11 システムをサポートしています。
- Silverlight では、C#、Iron Python、Iron Ruby、および Visual Basic® .NET の各開発言語をサポートしています。また、ほとんどの XAML コードは WPF ホストと Silverlight ホストの両方で実行できます。
- Silverlight 2.0 では、入力とデータ検証を実行するにはカスタム コードを実装する必要がありました。Silverlight 3.0 では、データ バインドを使用した例外に基づくデータ検証をサポートしています。この機能が Silverlight 3.0 以降のバージョンでサポートされているかどうかを確認するには、該当バージョンのドキュメントを参照してください。

セキュリティ

- Silverlight では .NET の暗号化 API を使用できます。まだ別のメカニズムを使用して暗号化していない場合に機密情報を格納したりサーバーに送信したりするときには、このような暗号化 API を使用する必要があります。
- Silverlight では、ログインしている特定のユーザーに関するログをユーザー ストア内の個別のファイルに記録します。コンピューター全体に関するログを 1 つのファイルに記録することはできません。

- Silverlight では、ダウンロードされるモジュールが隠ぺいされないので、モジュールが逆コンパイルされてプログラミング ロジックが抽出される可能性があります。

通信

- Silverlight では、Web サービスへの非同期呼び出しのみをサポートしています。
- Silverlight では、BasicHttpBinding のみをサポートしています。.NET Framework 3.5 の WCF では BasicHttpBinding をサポートしていますが、既定ではセキュリティが有効になっていません。少なくともトランスポート セキュリティを有効にして、サービスの通信をセキュリティで保護するようにします。
- Silverlight では、別のドメインのサービスを現在のページのソースで呼び出すために、2 つのファイル形式をサポートしています。Silverlight 専用の clientaccesspolicy.xml ファイルを使用することも、Adobe Flash と互換性がある crossdomain.xml ファイルを使用することもできます。Silverlight クライアントがアクセスする必要があるサーバーのルートに、このファイルを配置します。
- 大量のデータをサーバーから転送する必要がある場合は、Silverlight アプリケーションで ADO.NET Data Services を使用することを検討します。
- ブラウザーのセキュリティ モデルのため、現在、Silverlight ではサービスで公開される SOAP エラーをサポートしていません。サービスでは、別のメカニズムを使用してクライアントに例外を返す必要があります。

コントロール

- Silverlight には、専用に設計されたコントロールが用意されています。また、サード パーティから追加のコントロール パッケージが公開される可能性があります。
- 表示可能な HTML コンテンツやコントロールを Silverlight アプリケーションで表示する場合は、Silverlight のウィンドウのないコントロールを使用します。
- Silverlight を使用すると、既存のコントロールの実装に追加の動作をアタッチできます。コントロールをサブクラス化する代わりに、この手法を使用します。
- Silverlight では、すべての UI コンポーネントでアンチエイリアシングが実行されるので、ピクセル単位での UI 要素のスナップに関する推奨事項を検討します。

ストレージ

- Silverlight のローカル記憶域メカニズムは、クライアント コンピューターにある分離ストレージのキャッシュです。最大容量の初期値は 1 MB です。ストレージの最大容量に制限はありませんが、Silverlight では、アプリケーションからユーザーにストレージの容量を増やすように要求する必要があります。

「Contrasting Silverlight and WPF」(http://msdn.microsoft.com/en-us/library/dd458872.aspx、英語) も参照してください。

配置に関する考慮事項

RIA では、配置と保守容易性に関して Web アプリケーションと同じメリットが多数提供されます。個別にダウンロードしてキャッシュできる独立した複数のモジュールとして RIA を設計し、アプリケーション全体ではなく 1 つのモジュールを置き換えられるようにします。また、アプリケーションとコンポーネントのバージョンを管理して、クライアントで実行しているバージョンを検出できるようにします。配置と保守容易性に関する設計を行う際には、次のガイドラインを考慮します。

- RIA のブラウザー プラグインがインストールされていないシナリオの管理方法を検討します。
- クライアントでアプリケーションのインスタンスを実行しているときにモジュールを再配置する方法を検討します。
- 個別にキャッシュして、ユーザーにアプリケーション全体を再度ダウンロードするように要求しなくても簡単に置き換えられる、論理的なモジュールにアプリケーションを分割します。
- コンポーネントのバージョンを管理します。

RIA のプラグインのインストール

RIA のブラウザー プラグインがインストールされていない場合、次のようにプラグインのインストールを管理する方法を検討します。

- **イントラネット**: アプリケーション配布ソフトウェアまたは Microsoft Active Directory® ディレクトリ サービスのグループ ポリシー機能を使用して組織内の各コンピューターにプラグインを事前にインストールします (ただし、これらのソフトウェアや機能を使用できる場合に限ります)。または、Silverlight がオプション コンポーネントとして提供される Windows Update の使用を検討します。

最後に、ブラウザーを通じた手動インストールを検討します。この作業では、ユーザーがクライアントコンピューターで管理者特権を持っている必要があります。

- **インターネット**: ユーザーが手動でプラグインをインストールする必要があるので、最新のプラグインをダウンロードする適切な場所へのリンクをユーザーに提供する必要があります。Windows ユーザーの場合、Windows Update でプラグインがオプション コンポーネントとして提供されます。
- **プラグインの更新**: 一般に、プラグインの更新では下位互換性を考慮します。特定のプラグインのバージョンを対象とする場合もありますが、新しいバージョンのブラウザー プラグインの公開時には、そのバージョンでアプリケーションの動作を確認する計画を実施することを検討します。イントラネットのシナリオでは、新しいプラグインをアプリケーションでテストしてから配布します。インターネットのシナリオでは、プラグインが自動更新されると仮定します。プラグインのベータ版を使用してアプリケーションをテストして、プラグインのリリース時にユーザーが円滑に移行できるようにします。

分散配置

RIA では、プレゼンテーション ロジックがクライアントにコピーまたは移動されるので、分散アーキテクチャは、RIA で最も使用される可能性が高い配置シナリオです。RIA の分散配置では、プレゼンテーション ロジックをクライアントに配置し、ビジネス レイヤーをクライアントまたはサーバーに配置するかクライアントとサーバーで共有することが可能で、データ レイヤーを Web サーバーまたはアプリケーション サーバーに配置します。通常は、パフォーマンスを最大限に高めるために、ビジネス ロジックの一部を (場合によってはデータ アクセス ロジックの一部も) クライアントに移動します。この場合、ビジネス レイヤーとデータ アクセス レイヤーは、クライアントとアプリケーション サーバーにまたがって拡張されます (図 2 参照)。

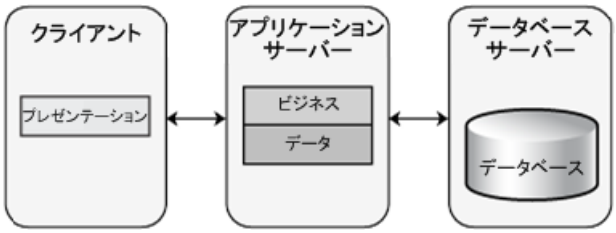


図 41

RIA の分散配置

RIA の配置については、次のガイドラインを考慮します。

- アプリケーションが大規模な場合は、RIA のコンポーネントをクライアントにダウンロードする際の処理要件を考慮します。
- ビジネス ロジックを他のアプリケーションと共有している場合は、すべてのアプリケーションがアクセスできるように、このビジネス ロジックをサービスとしてサーバー上で公開することを検討します。
- アプリケーションでソケットや WCF を使用し、ポート 80 を使用していない場合は、他のポートを通常ブロックしているファイアウォールがどのように影響するかを考慮します。
- RIA クライアントが必要に応じて他のドメインにアクセスできるように、crossdomain.xml ファイルを使用します。

負荷分散

アプリケーションを複数のサーバーに配置する場合は、負荷分散を使用して RIA のクライアント要求を別のサーバーに分散することが可能です。負荷を分散すると、応答時間が短縮し、リソースの使用率が高まり、スループットが最大限に高まります。図 3 に、負荷分散のシナリオを示します。

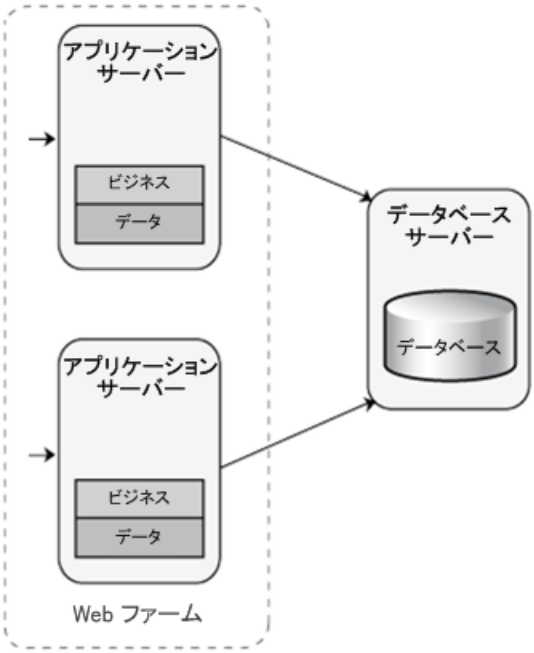


図 42

RIA の配置の負荷分散

負荷分散を使用するようにアプリケーションを設計する際には、次のガイドラインを考慮します。

- サーバー アフィニティを回避します。サーバー アフィニティは、特定のクライアントからのすべての要求を同じサーバーで処理しなければならない場合に発生します。サーバー アフィニティが最も発生しやすいのは、ローカルで更新可能なキャッシュ、インプロセス セッション状態ストア、またはローカル セッション状態ストアを使用する場合です。
- すべての状態をクライアントに格納して、ステートレスなビジネス コンポーネントを設計することを検討します。
- ネットワーク負荷分散ソフトウェアを使用して、アプリケーション ファームのサーバーに要求をリダイレクトする機能を実装することを検討します。

Web ファームに関する考慮事項

RIA アプリケーションのアプリケーション サーバーに、ビジネス ロジック、データ アクセス、またはデータ処理に関する重要な要件がある場合は、RIA クライアントから複数のサーバーに要求を分散する Web ファームの使用を検討します。Web ファームを使用すると、アプリケーションをスケールアウトして、ハードウェア障害の影響を軽減できます。アプリケーション用のサーバーを追加するには、負荷分散とクラスタリングのいずれかのソリューションを使用できます。次のガイドラインを考慮します。

- クラスタリングを使用してハードウェア障害の影響を軽減することを検討します。また、アプリケーションに、高い I/O に関する要件がある場合は、複数のデータベース サーバー間でデータベースを分割することを検討します。
- サーバー アフィニティまたはユーザー固有のキャッシュされたデータや状態をサポートする必要がある場合は、同じユーザーからのすべての要求を同一サーバーにルーティングするように Web ファームを構成します。
- サーバー アフィニティを実装しない場合、同じユーザーからの要求を同一のサーバーにルーティングできないため、サーバー アフィニティを実装しない限り、Web ファームでインプロセス セッション管理は使用しないようにします。このシナリオには、アウトプロセスのセッション サービスまたはデータベース サーバーを使用します。

配置パターンとシナリオの詳細については、第 19 章「物理ティアと配置」を参照してください。

関連する設計パターン

次の表に示すように、主要なパターンは、レイヤー、通信、構成、プレゼンテーションなどのカテゴリに分類されます。各カテゴリの設計を行う際には、これらのパターンの使用を検討します。

カテゴリ	関連するパターン
レイヤー	**Service Layer**: サービスのインターフェイスと実装が単一のレイヤーにグループ化されたアーキテクチャの設計パターンです。
通信	**Asynchronous Callback**: 実行時間が長いタスクを、バックグラウンドで実行される独立したスレッドで実行し、タスクが完了したときにスレッドがコールバックする機能を提供します。 **Command**: 要求処理を、共通の実行インターフェイスを公開する別個のコマンド オブジェクトにカプセル化します。
構成	**Composite View**: 個々のビューを複合ビューに統合します。 **Inversion of Control**: アプリケーションでオブジェクトを使用するために満たす必要がある、他のオブジェクトやコンポーネントへのオブジェクトの依存関係を設定します。
プレゼンテーション	**Application Controller**: すべてのフロー ロジックが含まれ、Model と連携して適切な View を表示する他の Controller によって使用されるオブジェクトです。 **Supervising Presenter**: プレゼンテーションの設計を 3 つの別個の役割に分離します。View ではユーザーの入力を処理して Model コンポーネントに対してデータ バインドし、Model ではビジネス データをカプセル化します。Presenter オブジェクトではプレゼンテーション ロジックを実装して、View と Model 間の通信を調整します。 **Presentation Model**: 現在の UI 開発プラットフォーム (View が開発者によって作成されるのではなく、デザイナーによって作成されます) に合わせてカスタマイズされた Model-View-Presenter (MVP) パターンの変化形です。

Composite View パターンの詳細については、「Patterns in the Composite Application Library」(http://msdn.microsoft.com/en-us/library/dd458924.aspx、英語) を参照してください。

Model-View-Controller (MVC) パターンと Application Controller パターンの詳細については、Martin Fowler 著『エンタープライズ アプリケーション アーキテクチャ パターン』(翔泳社、2005 年) を参照するか、http://martinfowler.com/eaaCatalog (英語) を参照してください。

Command パターンの詳細については、Erich Gamma、Richard Helm、Ralph Johnson、John Vlissides 共著『オブジェクト指向における再利用のためのデザイン パターン』(ソフトバンク クリエイティブ、1999 年) の第 5 章「振る舞いに関するパターン」を参照してください。

Asynchronous Callback パターンの詳細については、「Creating a Simplified Asynchronous Call Pattern for Windows Forms Applications」(http://msdn.microsoft.com/en-us/library/ms996483.aspx、英語) を参照してください。

Service Layer パターンの詳細については、「P of EAA: Service Layer」(http://www.martinfowler.com/eaaCatalog/serviceLayer.html、英語) を参照してください。

関連情報

Web リソースに簡単にアクセスするには、http://www.microsoft.com/architectureguide (英語) でオンライン版の参考文献を参照してください。

- Silverlight の詳細については、Silverlight の公式 Web サイト (http://silverlight.net/default.aspx、英語) を参照してください。
- WCF と Silverlight の併用する方法の詳細については、「方法: 双方向サービスを構築する」(http://msdn.microsoft.com/ja-jp/library/cc645027(VS.95).aspx) と「方法: チャネル モデルで双方向サービスにアクセスする」(http://msdn.microsoft.com/ja-jp/library/cc645028(VS.95).aspx) を参照してください。
- Silverlight に関するブログについては、Brad Abrams のブログ (http://blogs.msdn.com/brada/、英語) と Scott Guthrie のブログ (http://weblogs.asp.net/Scottgu/、英語) を参照してください。

24

モバイル アプリケーションの設計

概要

この章では、モバイル アプリケーションが適切なソリューションとなる場合や状況、およびモバイル アプリケーションの設計に関する重要な考慮事項を理解するのに役立つ情報を提供します。モバイル アプリケーションに含まれるコンポーネント、モバイル アプリケーション固有の問題 (配置、消費電力、同期など)、および主要なパターンとテクノロジに関する考慮事項についても説明します。

モバイル アプリケーションは、通常、プレゼンテーション レイヤー、ビジネス レイヤー、およびデータ レイヤーで構成されたマルチレイヤー型のアプリケーションとして構築されています。モバイル アプリケーションを開発する場合は、Web ベースのシン クライアント、またはリッチ クライアントを開発できます。リッチ クライアントを構築すると、ビジネス レイヤーとデータ サービス レイヤーがデバイス自体に配置される可能性が高くなります。シン クライアントでは、すべてのレイヤーがサーバーに配置されます。図 1 に、コンポーネントを関連領域でグループ化した、一般的なリッチ クライアント モバイル アプリケーションのアーキテクチャの例を示します。

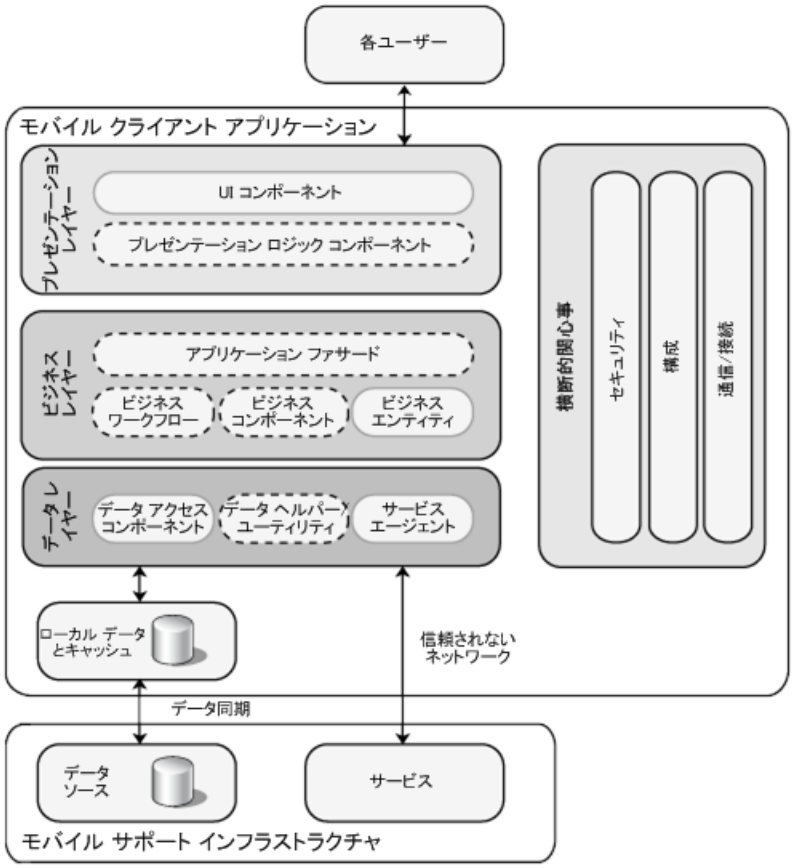


図 43

モバイル アプリケーションの一般的な構造

通常、モバイル アプリケーションでは、プレゼンテーション レイヤーにユーザー インターフェイス コンポーネントが含まれており、場合によってはプレゼンテーション ロジック コンポーネントが含まれることもあります。ビジネス レイヤーが存在する場合は、通常、このレイヤーに、ビジネス ロジック コンポーネント、アプリケーションで必要なすべてのビジネス ワークフローとビジネス エンティティ コンポーネント、および (オプションで) ファサードが含まれます。データ レイヤーには、通常、データ アクセス コンポーネントとサービス エージェント コンポーネントが含まれます。デバイスのリソースの使用量を最小限に抑えるため、一般的に、モバイル アプリケーションでは、堅固なレイヤーの手法や独立したコンポーネントはあまり使用されません。レイヤー型の設計の詳細については、第 5 章「レイヤー型アプリケーションのガイドライン」を参照してください。各レイヤーに適したコンポーネントの詳細については、第 10 章「コンポーネントのガイドライン」を参照してください。

設計に関する一般的な考慮事項

次の設計ガイドラインでは、モバイル アプリケーションを設計する際に考慮する必要がある、さまざまな側面についての情報を提供します。アプリケーションが要件を満たし、モバイル アプリケーションに共通するシナリオで効率的に機能するためには、次のガイドラインに従います。

- **リッチ クライアント、Web ベースのシン クライアント、またはリッチ インターネット アプリケーション (RIA) を構築するかどうか判断する**: アプリケーションがローカルな処理を必要とし、不定期に接続するシナリオで機能しなければならない場合は、リッチ クライアントを設計することを検討します。リッチ クライアント アプリケーションでは、インストールと管理が複雑になります。アプリケーションでサーバー処理を使用することが可能で、ネットワーク接続が常時利用できる場合は、シン クライアントを設計することを検討します。アプリケーションでリッチ UI が必要で、ローカル リソースへのアクセスが制限されていて、その他のプラットフォームに移植できる必要がある場合は、RIA クライアントを設計します。
- **サポートするデバイスの種類を決定する**: サポートするデバイスの種類を選択する際には、画面のサイズと解像度、CPU のパフォーマンス特性、メモリと記憶域、および開発ツール環境の可用性を考慮します。また、ユーザーの要件と組織の制約についても考慮します。GPS (Global Positioning System) やカメラなど、特定のハードウェアが必要な場合もあります。このような要素は、アプリケーションの種類だけでなく、デバイスの選択にも影響することがあります。
- **適切な場合は、不定期に接続するシナリオと限られた帯域幅を使用するシナリオを検討する**: モバイル デバイスがスタンドアロン デバイスの場合は、接続の問題について考慮する必要はありません。ネットワーク接続が必要な場合、ネットワーク接続が断続的または使用できないときに、モバイル アプリケーションで、このような状況に対処する必要があります。この場合は、ネットワーク接続が断続的な状態であることを考慮して、キャッシュ、状態管理、およびデータ アクセス メカニズムを設計することが不可欠です。たとえば、ネットワーク接続を使用できるときに、通信を一括で行ってデータを配信するようにします。プログラミングのしやすさだけでなく、処理速度、電力消費、および粒度に基づいてハードウェアとソフトウェアのプロトコルを選択します。
- **プラットフォームの制約を考慮して、モバイル デバイスに適した UI を設計する**: デバイス ハードウェアによる制約内でモバイル デバイスが機能するには、より単純なアーキテクチャ、よりシンプルな UI、およびその他の特定の設計に関する判断が必要になります。これらの制約を念頭に置いて、デスクトップ アプリケーションや Web アプリケーションのアーキテクチャまたは UI を再利用するのではな

く、デバイス専用に設計します。主要な制約は、メモリ、バッテリー残量、異なる画面のサイズや向きへの対応機能、セキュリティ、およびネットワーク帯域幅です。

- **モバイル デバイスに適したレイヤー型アーキテクチャを設計して、再利用性と保守容易性を向上する**: アプリケーションの種類によっては、複数のレイヤーがデバイス自体に配置されることがあります。レイヤーの概念を使用して、懸念事項をできる限り分離し、モバイル アプリケーションの再利用性と保守容易性を向上します。ただし、デスクトップ アプリケーションや Web アプリケーションよりも設計を簡略化して、デバイスのリソースの使用量を最小限に抑えるようにします。
- **デバイスでのリソースの制約 (バッテリー残量、メモリ サイズ、プロセッサの処理速度など) を考慮する**: 設計上の判断を行う際には、モバイル デバイスの限られた CPU、メモリ、記憶容量、およびバッテリー残量を考慮する必要があります。バッテリー残量は、一般的に、モバイル デバイスで制限が最も厳しい要素です。バックライト、メモリの読み取りと書き込み、ワイヤレス接続、特殊なハードウェア、およびプロセッサの処理速度はすべて、全体的な消費電力に影響を及ぼします。使用できるメモリの量が少ない場合、Windows Mobile オペレーティング システムは、アプリケーションに対して、アプリケーションを終了するか、キャッシュされたデータを消去してプログラムの実行速度を下げるかを問い合わせます。このプロセスでは、パフォーマンスを考慮しながら、電力とメモリの消費量を最小限に抑えるようにアプリケーションを最適化します。

設計に関する具体的な問題

設計を策定する際に考慮する必要がある一般的な問題がいくつかあります。これらの問題は、設計の具体的な領域に分類できます。次のセクションでは、各領域で一般的に発生する問題を解決するのに役立つガイドラインを提供します。

- 認証と承認
- キャッシュ
- 通信
- 構成管理
- データ アクセス
- デバイスの特性
- 例外管理
- ログ記録

- アプリケーションの移植
- 電源管理
- 同期
- テスト
- ユーザー インターフェイス
- 検証

認証と承認

効果的な認証と承認の方針を設計することは、アプリケーションのセキュリティと信頼性の両方において重要です。認証が不十分だと、不正なアクセスに対して、アプリケーションが脆弱になるおそれがあります。通常、モバイル デバイスは、1 人のユーザーが使用することを想定して設計されているので、多くの場合、基本的なユーザー プロファイルや単純なパスワード以上のセキュリティの追跡機能は備わっていません。また、その他の一般的なデスクトップ コンピューターに備わっているメカニズムも存在しない可能性があります。Bluetooth などのプロトコル経由でモバイル デバイスを検出できることにより、ユーザーに予期しないリスクがもたらされることがあります。また、接続の中断により、モバイル アプリケーションの設計が、さらに困難になる場合があります。無線または有線にかかわらず、考えられるすべての接続シナリオについて考慮します。認証と承認を設計する際には、次のガイドラインを考慮します。

- 常時接続シナリオと不定期に接続するシナリオの両方に対応するように認証と承認を設計します。このシナリオには、無線での同期、クレイドルにドッキングする (PC による) 同期、Bluetooth 経由の検出、仮想プライベート ネットワーク (VPN) 経由の同期、ローカルの SD メモリ カードの同期などが含まれます。
- デバイスによって、プログラミングのセキュリティ モデルが異なる可能性があり、リソースにアクセスするための認証に影響を及ぼす場合があることを考慮します。
- 同じツールを使用していても、モバイル プラットフォームでは、大きなプラットフォームで使用できるセキュリティ メカニズムが使用できるとは限らないことに注意してください。たとえば、Windows Mobile ではアクセス制御リスト (ACL) を使用できません。そのため、オペレーティング システム レベルのファイルのセキュリティは存在しません。

- モバイル アプリケーションのレイヤー内の信頼境界 (たとえば、クライアントとサーバー間、またはサーバーとデータベース間など) を特定します。信頼境界を特定すると、認証を行う場所と方法を決定するのに役立ちます。

キャッシュ

キャッシュを使用して、アプリケーションのパフォーマンスと応答性を向上し、ネットワーク接続を使用できない場合の操作をサポートします。キャッシュでは、参照データの検索処理を最適化し、ネットワークのラウンド トリップを回避して、同じ処理が不要に何度も実行されることを回避できます。モバイル デバイスでは、使用できる記憶域がデスクトップ コンピューターよりも少なくなるので、キャッシュするデータを決める際には、デバイスの限られたリソースを考慮します。キャッシュを設計する際には、次のガイドラインを考慮します。

- パフォーマンス目標を特定します。たとえば、応答時間とバッテリー残量の最低ラインを判断します。使用するデバイスのパフォーマンスをテストします。ほとんどのモバイル デバイスでは、フラッシュ メモリのみが使用されています。そのため、デスクトップ コンピューターで使用されるメモリよりも低速になる可能性があります。
- メモリの消費量を最小限に抑えるように設計します。アプリケーションが機能するために不可欠なデータのみ、またはすぐに使用できる形式への変換にコストがかかるデータのみをキャッシュします。メモリを集中的に使用するアプリケーションを設計する場合は、メモリが不足するシナリオを特定して、使用できるメモリが減少したときに破棄するデータの優先順位を決定するメカニズムを設計します。ただし、不定期に接続するシナリオやオフライン シナリオでアプリケーションが必要とするデータ (揮発性データを含む) をキャッシュすることを検討します。また、オフライン シナリオや不定期に接続するシナリオでキャッシュされたデータを使用できない場合に、アプリケーションがこの状況を切り抜けられるようにします。
- 適切なキャッシュの場所 (デバイス、モバイル ゲートウェイ、またはデータベース サーバー) を選択します。アプリケーションで使用するメモリは、メモリが不足するとクリアされることがあるので、キャッシュには、デバイスのメモリではなく SQL Server Compact Edition を使用することを検討します。
- 機密情報をキャッシュする際には、データを暗号化します。リムーバブル メモリ メディアのデータをキャッシュする場合だけでなく、デバイスのメモリ内データをキャッシュする場合にも、データを暗号化することを検討します。

キャッシュ方針の設計の詳細については、第 17 章「横断的関心事」を参照してください。

通信

デバイスの通信には、ワイヤレス通信 (無線) およびホスト コンピューターとの有線の通信があり、その他に Bluetooth や Infrared Data Association (IrDA) といった特殊な通信があります。無線で通信する場合は、データのセキュリティを考慮して、データの窃盗や改ざんから機密データを保護します。Web サービス インターフェイスを通じて通信する場合は、WS-Security 標準などのメカニズムを使用してデータのセキュリティを確保します。ワイヤレス デバイスの通信がコンピューターの通信よりも中断しやすいことと、ネットワークに接続されていない状態でアプリケーションを長期間操作しなければならない可能性があることを覚えておく必要があります。通信の方針を設計する際には、次のガイドラインを考慮します。

- 非同期のスレッド通信を設計して、不定期に接続するシナリオのパフォーマンスとユーザビリティを向上します。モバイル デバイスで一般的な限られた帯域幅を使用する接続では、(特にユーザー インターフェイスがブロックされる場合に) パフォーマンスが低下し、ユーザビリティに影響を及ぼす可能性があります。適切な通信プロトコルを使用すると共に、複数の接続の種類を使用できる場合のアプリケーションの動作を考慮します。ユーザーが使用する接続を選択し、必要に応じて通信を切断してバッテリー残量を維持できるようにすることを検討します。
- 携帯電話で実行するアプリケーションを設計する場合は、通信中やプログラムの実行中に電話がかかってきたときの影響について考慮します。アプリケーションを中断して再開したり、終了したりできるように、アプリケーションを設計します。
- 信頼されない接続 (Web サービスやその他の無線による方法) 経由の通信を保護します。機密データに暗号化とデジタル署名を使用して、VPN 経由で渡されるデータが保護されるように考慮します。ただし、通信のセキュリティがパフォーマンスやバッテリー残量に与える影響についても考慮する必要があります。
- 複数ソースのデータにアクセスしたり、他のアプリケーションと連動したり、接続が切断されている間に作業したりしなければならない場合は、Web サービスを使用して通信することを検討します。特に、限られた帯域幅を使用する通信のシナリオでは、接続を効率的に管理するようにします。
- WCF を使用して通信し、メッセージ キューを実装する必要がある場合は、WCF のストア アンド フォワードを使用することを検討します。

通信プロトコルと通信技法の詳細については、第 18 章「通信とメッセージ」を参照してください。

構成管理

デバイスの構成管理を設計する際には、デバイスがリセットされたときの対応と、アプリケーションの構成を無線通信とホスト コンピューターからの通信のどちらを使用して行えるようにするのかについて考慮します。構成管理の方針を設計する際には、次のガイドラインを考慮します。

- 構成情報に適切な形式を選択します。XML でバイナリ形式を使用して、メモリ使用率を最小限に抑えることを検討します。圧縮ライブラリのルーチンを使用して、構成情報と状態情報を格納するためのメモリの要件を減らすことを検討します。構成ファイルに格納される機密データを暗号化します。
- 設計でデバイスをリセットした後の構成の復元をサポートします。無線経由で構成情報を同期する方法、およびクレイドルにドッキングしたときにホスト コンピューターを使用して構成情報を同期する方法について考慮し、構成設定を読み込むためにさまざまな製造元で使用されている技法を理解します。
- Microsoft SQL Server 2005 または Microsoft SQL Server 2008 データベースに企業データが格納されており、製品化までの時間を短縮する必要がある場合は、サード パーティ製の構成可能なアプリケーションとマージ レプリケーションを使用することを検討します。マージ レプリケーションでは、ネットワーク帯域幅やデータのサイズに関係なく、1 回の操作でデータを同期できます。
- Active Directory インフラストラクチャを使用している場合は、System Center Mobile Device Manager のインターフェイスを使用して、グループ構成、デバイスの認証と承認を管理することを検討します。Mobile Device Manager の要件の詳細については、「テクノロジに関する考慮事項」を参照してください。

データ アクセス

モバイル デバイスのデータ アクセスは、信頼されないネットワーク接続の制約と、デバイス自体のハードウェアの制約を受けます。データ アクセスを設計する際には、低帯域幅、長い待ち時間、および断続的な接続が設計に及ぼす影響を考慮します。データ アクセスを設計する際には、次のガイドラインを考慮します。

- 同期サービスを提供するローカル デバイス データベース (SQL Server Compact Edition など) の使用を検討します。標準的なデータ同期の機能で要件を満たすことができない場合にのみ、カスタム メカニズムを設計してデータを同期します。
- データの整合性が保たれるようにプログラミングします。デバイスが中断されているときに開いたままになっているファイルや電源障害により、データの整合性に問題が生じることがあります。この問題は、

特にデータがリムーバブル記憶装置に格納されている場合に発生しやすくなります。例外ハンドルと再試行ロジックを含めることで、ファイル操作が成功するようにします。デバイスで電源が供給されなくなったり、接続が切断された場合にデータの整合性を確保するには、SQL Server Mobile でトランザクションを使用することを検討します。

- 取り外し可能なメモリ カードは、ユーザーがいつでも取り外せるので、常に使用できると見なさないようにする必要があります。リムーバブル記憶装置に書き込む前または FlushFileBuffers を使用する前に、リムーバブル記憶装置の有無を確認します。
- XML を使用してデータを格納または転送する場合は、データ全体のサイズと、パフォーマンスへの影響を考慮します。XML を使用すると、帯域幅とローカル記憶域の両方の要件が増加します。圧縮アルゴリズムまたは XML 以外の転送方法を使用します。
- 効率的なデータベース アクセスとデータ処理を設計することで、パフォーマンスへの影響を最小限に抑えます。DataSet ではなく型指定されたオブジェクトを使用して、メモリのオーバーヘッドを減らし、パフォーマンスを向上することを検討します。データの読み取りのみ行い、書き込みを行わない場合は、DataReader を使用します。大量のプロセスを使用する操作 (大規模なデータ セット内の移動など) を回避します。

データ レイヤーの設計に関する詳細については、第 8 章「データ レイヤーのガイドライン」を参照してください。

デバイスの特性

モバイル デバイスのハードウェアには制約があり、他のデバイスのハードウェアとは性質が異なっているので、モバイル デバイスの設計と開発は独特です。まったく異なるハードウェアの特徴を持つ複数のデバイスを対象としたアプリケーションを開発する場合もあります。モバイル アプリケーションを設計する際には、異なるデバイス環境を念頭に置く必要があります。このようなものには、画面のサイズと向き、メモリと記憶域の制限、およびネットワーク帯域幅と接続が含まれます。一般的に、選択するモバイル オペレーティング システムは、対象のデバイスの種類によって異なります。デバイスの方針を決定する際には、次のガイドラインを考慮します。

- 画面のサイズと向き、ネットワーク帯域幅、メモリと記憶域、プロセッサのパフォーマンス、その他のハードウェアの機能などを考慮することで、デバイスに合わせてアプリケーションを最適化します。
- 加速度計、グラフィックス プロセッシング ユニット、GPS、触覚に基づく (タッチ、フォース、および振動) フィードバック、コンパス、カメラ、指紋リーダーといった、アプリケーションの機能を強化できるデバイス固有の機能について考慮します。

- 1 つ以上のデバイスを対象にしたアプリケーションを開発する場合は、最初にすべてのデバイスに存在している機能のサブセット用に設計してから、コードをカスタマイズして、デバイス固有の機能を使用できるときに、その機能を検出して使用します。
- 限られたメモリ リソースを考慮し、アプリケーションを最適化して、最低限の量のメモリを使用するようにします。メモリが不足している場合には、システムにより、キャッシュされた中間言語 (IL) コードが解放され、メモリの使用量が減少し、解釈モードに戻る場合があるので、全体的な実行速度が低下します。
- モジュール コードを作成して、実行可能ファイルからモジュールを簡単に削除できるようにします。この動作により、デバイスのメモリ サイズが原因で、別個のより小さな実行可能ファイルが必要な場合にも対応できます。
- コード サイズとメモリの使用量が増加する可能性がある単純なプログラミングの手法に従うのではなく、プログラミング ショートカットの使用を検討します。たとえば、単純なオブジェクト指向の手法 (抽象型基本クラスや繰り返されるオブジェクトのカプセル化) を使用する際のコストを調べます。限定的な初期化を使用して、オブジェクトが必要な場合にのみ初期化されるようにすることを検討します。

例外管理

効果的な例外管理方針を設計することは、アプリケーションのセキュリティと信頼性の両方において重要です。モバイル アプリケーションで適切に例外をハンドルすると、機密性の高い例外の詳細がユーザーに開示されるのを防ぎ、アプリケーションの堅牢性を向上し、エラーが発生した場合にアプリケーションが一貫性のない状態にならないようにするのに役立ちます。例外管理を設計する際には、次のガイドラインを考慮します。

- エラーが発生した後に、機密情報をエンド ユーザーに開示することなく既知の適切な状態に回復するようにアプリケーションを設計します。
- 例外をハンドルできる場合のみ例外をキャッチします。また、例外を使用してロジックのフローを制御しないようにします。グローバル エラー ハンドラーを設計して、ハンドルされない例外をキャッチするようにします。
- ログ記録と通知に関する適切な方針を設計して、例外に関する十分な詳細情報を記録します。ただし、モバイル デバイスのメモリと記憶域の制限には留意します。わかりやすい例外メッセージを表示して、重大なエラーと例外に関する機密情報が開示されないようにします。

例外管理方針の設計の詳細については、第 17 章「横断的関心事」を参照してください。

ログ記録

モバイル デバイスで使用できるメモリの量は限られているので、ログ記録とインストルメンテーションは必須の場合 (たとえば、デバイスへの侵入が試みられた場合など) のみに制限する必要があります。デバイスがより大きなインフラストラクチャの一部として設計されている場合は、インフラストラクチャ レベルでほとんどのデバイス操作を追跡するようにします。一般的に、監査は、監査情報がリソースへのアクセスと同じタイミングで生成され、リソースにアクセスした同じルーチンで生成された場合に、最も確実だと見なされます。デバイス側でログ記録を生成して、ネットワークに接続されている間にサーバーと同期しなければならないことを考慮します。ログ記録を設計する際には、次のガイドラインを考慮します。

- Windows Mobile には、イベント ログのメカニズムがありません。.NET Compact Framework をサポートするサードパーティ製のログ記録メカニズム (OpenNetCF、NLog、log4Net など) の使用を検討します (詳細については、この章の最後にある「関連情報」を参照してください)。また、デバイスに格納されたログ記録へのアクセス方法についても考慮します。
- デバイスで大規模なログ記録を実行する場合は、簡略化または圧縮された形式を使用して、メモリと記憶域への影響を最小限に抑えることを考慮します。また、ログ記録はデバイス側で行うのではなく、リモートで行うことを検討します。
- プラットフォーム機能 (サーバーでは正常性の監視、デバイスではモバイル デバイス サービスなど) を使用して、イベントを監査してログに記録することを検討します。Open Mobile Alliance Device Management (OMA DM) 標準を使用して、リモートの正常性の監視機能を追加します。
- モバイル データベースのログ記録とサーバー データベースのログ記録を同期して、サーバーの監査機能を管理します。Active Directory インフラストラクチャを使用している場合は、System Center Mobile Device Manager を使用して、モバイル デバイスからログ記録を抽出することを検討します。Mobile Device Manager の要件の詳細については、「テクノロジに関する考慮事項」を参照してください。
- やむを得ない場合を除き、ログ ファイルと監査ファイルには機密情報を保存しないようにし、機密情報は暗号化して保護します。
- デバイスの例外的な操作や疑わしい操作の要素を決定し、これらのシナリオに基づいて情報をログに記録します。

アプリケーションの移植

多くの場合、開発者は、既存のアプリケーションの一部または全部をモバイル デバイスに移植する必要があります。アプリケーションの種類によっては簡単に移植できるものもありますが、コードを変更することなく直接移植できることは、ほとんどありません。既存のアプリケーションをモバイル デバイスに移植する設計を行う際には、次のガイドラインを考慮します。

- デスクトップからリッチ クライアント アプリケーションを移植する場合は、アプリケーション全体を作成し直します。リッチ クライアントが小さな画面サイズや限られたメモリおよびディスクのリソースを考慮して設計されていることはほとんどありません。
- モバイル デバイスに Web アプリケーションを移植する場合は、小さな画面サイズに合うように UI を作成し直すことを検討します。また、消費電力やユーザーの接続コストの増加につながる可能性があるので、通信上の制限やインターフェイスで小さな要求が過剰に行われないかどうかを考慮します。
- RIA クライアントを移植する場合は、コードを調査して、変更なしに移植できるコードを特定します。具体的なアドバイスの詳細については、この章の「テクノロジに関する考慮事項」を参照してください。
- 移植に役立つツールを調査して使用します。たとえば、Java から C++ へのコンバーターを使用できます。Smartphone のコードから Pocket PC のコードに変換する場合は、Visual Studio を使用してターゲット プラットフォームを変更することが可能で、Smartphone 固有の機能を使用している場合には警告が表示されます。また、Visual Studio のデスクトップ プロジェクトとモバイル プロジェクトをリンクして、2 つのプロジェクト間で移植できる機能を検出することもできます。
- カスタム コントロールを変更なしにモバイル アプリケーションに移植できるとは見なさないでください。サポートされる API、メモリの使用量、および UI の動作は、モバイル デバイスによって異なります。コントロールを作成し直す計画を立てたり、必要に応じて代替手段を見つけられるように、コントロールをできるだけ早い段階でテストします。

電源管理

電源は、モバイル デバイスで制限される主な設計要素です。すべての設計上の判断を下す際には、デバイスで消費される電力と、その判断による全体的なバッテリー残量への影響について考慮する必要があります。可能な場合は、ユニバーサル シリアル バス (USB) やその他の種類のデータ接続から電力を消費するデバイスについても考慮します。通信プロトコルについて調べて、その関連する電力消費を調査します。電力消費を設計する際には、次のガイドラインを考慮します。

- 電源プロファイルを実装することで、デバイスが外部電源に接続されてバッテリーの電力を消費していないときのパフォーマンスを向上します。デバイスの機能を使用していない場合や不要な場合に、ユーザーがその機能を無効にできるようにします。一般的な例としては、画面のバックライト、ハード ドライブ、GPS 機能、スピーカー、およびワイヤレス通信があります。
- バッテリー残量を節約するには、アプリケーションをバックグラウンドで実行している間に UI を更新しないようにします。
- 無線で転送するデータが可能な限り少なくなるように、プロトコルを選択して、サービス インターフェイスを設計し、通信を一括で行います。通信方法を選択する際には、消費電力とネットワーク速度の両方を考慮して、デバイスで外部電源が供給されるまで、不必要なワイヤレス通信を延期することを検討します。
- 3G ハードウェアの通信プロトコル (Edge プロトコルなど) の使用を検討している場合、これは非常に高速ですが、以前の通信プロトコルよりもはるかに多くの電力が消費されることを考慮してください。3G を使用している場合は、一括のバーストで通信を行い、不要な場合に通信を終了するようにします。

同期

無線による同期またはクレイドルにドッキングする同期、あるいはこれらの両方をサポートするかどうかを検討します。同期には機密データが含まれることが多いので、特に無線で同期する場合には、同期データの保護方法について考慮します。操作をキャンセルするか、接続が使用できるようになったときに操作を再開できるようにすることで、接続の中断に対処するように同期を設計します。マージ レプリケーションでは、アップロードのみの同期と双方向の同期の両方を使用できるので、新しいバージョンの SQL Server を使用しているインフラストラクチャに適しています。さまざまな状況で堅牢な同期サービスを提供することができる、Microsoft Sync Framework の使用を検討します。同期を設計する際には、次のガイドラインを考慮します。

- ユーザーがホスト コンピューターと同期する場合は、クレイドルにドッキングする同期を設計に含めることを検討します。ユーザーが社外からデータを同期する必要がある場合は、無線による同期を設計に含めることを検討します。
- 同期がリセットまたは中断された場合にアプリケーションを回復できるようにして、同期の競合の処理方法を決定します。
- 暗号化、デジタル証明書、およびセキュリティで保護されたチャネルを使用するなどして、同期通信が保護されるようにします。特に、Bluetooth を使用して同期する場合は、適切な認証と承認を使用するようにします。

- SQL Server への双方向の同期をサポートする必要がある場合は、マージ レプリケーションによる同期の使用を検討します。マージによる同期では、マージのセットに含まれるすべてのデータが同期されますが、追加のネットワーク帯域幅が必要になることがあるので、パフォーマンスに悪影響を及ぼす可能性があることに留意します。
- WCF ではデータが確実に配信され、不定期に接続するシナリオで適切に機能するので、電子メールや SMS (テキスト メッセージ) ではなく、WCF を使用してストア アンド フォワードの同期を使用することを検討します。

テスト

モバイル アプリケーションのデバッグは、コンピューターで同様のアプリケーションをデバッグする場合よりもはるかにコストがかかる場合があります。アプリケーションでサポートするデバイスの種類や数を決定する際には、このデバッグにかかるコストを考慮します。また、デバイスからデバッグ情報を取得するのがさらに難しくなる可能性があることと、デバイス エミュレーターでデバイス ハードウェアの環境が完全にシミュレーションされるわけではないことを覚えておいてください。デバッグの方針を設計する際には、次のガイドラインを考慮します。

- サポートするデバイスを選択する際には、デバッグにかかるコストを把握します。ツールのサポート、初期テスト用デバイスのコスト (おそらく差し替え用デバイスのコストもかかります)、ソフトウェア ベースのデバイス エミュレーターのコストを考慮します。
- アプリケーションを実行する対象の物理的なデバイスを使用できる場合は、エミュレーターを使用するのではなく、実際のデバイスでコードをデバッグします。デバイスを使用できない場合は、初期テストとデバッグにエミュレーターを使用します。エミュレーターでは、実際のデバイスよりもコードの実行が遅くなる場合があることを考慮します。
- 物理的なデバイスを入手したらすぐ、標準的なコンピューターに接続したデバイスでコードを実行するように切り替えます。デバイスがネットワークにまったく接続されていないシナリオ (これには、コンピューターのデバッグ セッションに接続されていない場合も含まれます) をテストして、コンピューターに接続されていない状態のデバイスで最終テストを実行します。このシナリオの問題をデバッグする際には、一時的または永続的なメカニズムを追加します。また、デバイスをサポートするユーザーの要件を考慮します。
- デバイスがまだ作成されていない OEM では、専用の x86 ベースの Windows CE コンピューターでモバイル プログラムをデバッグできます。デバイスが使用できるようになるまでは、このオプションの使用を検討します。

ユーザー インターフェイス

モバイル アプリケーションの UI を設計する際には、デスクトップ アプリケーションの UI を適応したり再利用したりしないようにします。デバイスの UI は可能な限りシンプルになるように設計し、ペン ベースの入力と限られたデータ入力機能専用に設計します。モバイル アプリケーションが全画面表示モードで実行される、一度に 1 つのウィンドウしか表示できなくなるので、ブロックしている操作によってユーザーがアプリケーションを操作できなくなることを考慮します。モバイル アプリケーションの UI を設計する際には、次のガイドラインを考慮します。

- 1 つのウィンドウおよび全画面表示に合わせた UI を設計します。主要なアプリケーションのみを実行する 1 人のユーザーが使用することを想定したデバイスをサポートする場合は、キオスク モードの使用を検討します。Windows Mobile ではキオスク モードがサポートされていないので、Windows CE を使用する必要があることを覚えておいてください。
- アプリケーションの UI を設計する際には、サポートするデバイスでさまざまな画面のサイズと向きが使用される可能性があることを考慮します。また、デスクトップ環境に比べて小さな画面サイズ、限られた API、および少ない範囲の UI コントロールによる制限についても考慮します。
- タッチスクリーンやスタイラス駆動の UI をサポートすることで、ユーザビリティを考慮した設計にします。メニュー バーやその他のコントロールを画面下部に配置し (必要に応じて上方向に拡張して)、ユーザーの手で画面が隠れないようにします。十分な大きさのボタンを配置することでタッチスクリーン入力をサポートし、入力に指やスタイラスを使用することで UI が使用しやすくなるように、コントロールのレイアウトを決めます。
- ブロックしている操作を視覚的に表示します (たとえば、砂時計カーソルなどを表示します)。

検証

検証機能を使用して、デバイスとアプリケーションを保護し、ユーザビリティを向上します。特に、不定期に接続するシナリオや非接続型のシナリオでは、入力値をリモート サーバーに送信する前に検証することで、通信のラウンド トリップを減らし、アプリケーションのパフォーマンスとユーザビリティを向上できます。検証を設計する際には、次のガイドラインを考慮します。

- ユーザーによるデータ入力を可能な限り検証することで、不要な通信やサーバーのラウンド トリップを防ぎます。これにより、ユーザーが無効な値を入力したときのアプリケーションの反応が向上します。
- ホスト コンピューターとの通信と無線通信で受け取ったすべてのデータを検証します。

- カメラや通話の開始などの機能を自動的に開始するコードと操作を検証することで、このようなハードウェア リソースを保護します。
- メモリの使用量を最小限に抑えた効率的な検証メカニズムを設計することで、デバイスの限られたリソースとパフォーマンスを考慮します。

検証の技法に関する詳細については、第 17 章「横断的関心事」を参照してください。

テクノロジに関する考慮事項

このセクションでは、モバイル アプリケーションとテクノロジの一般的なシナリオに関する提案やアドバイスを提供します。

Microsoft Silverlight for Mobile

このガイドの公開時には、Silverlight for Mobile は開発中のリリースされていない製品として発表されていました。Silverlight for Mobile を使用する場合は、次のガイドラインを考慮します。

- リッチ メディアと対話性をサポートし、モバイル デバイスとデスクトップの両方で実行できるアプリケーションを作成する必要がある場合は、Silverlight for Mobile の使用を検討します。デスクトップの Silverlight 2.0 プラグインで実行することを想定して作成した Silverlight 2.0 のコードは、モバイル ブラウザー用の Microsoft Internet Explorer® の最新バージョンを使用して、Windows Mobile Silverlight プラグインで実行できます。モバイル デバイスとデスクトップで同じ Silverlight のコードを使用することはできますが、モバイル デバイス画面サイズやリソースの制約を考慮する必要があります。Windows Mobile 用にコードを最適化することを検討します。
- デスクトップとモバイルの両方のプラットフォームに対応した Web ページを作成する必要がある場合は、ASP.NET for Mobile ではなく、Silverlight for Mobile または通常の ASP.NET と HMTL の使用を検討します (ただし、デバイスでこれらのいずれの代替手段もサポートできない場合は除きます)。デバイスのブラウザーは進化しているので、モバイル デバイス用のブラウザーでもデスクトップを対象としたネイティブな HTML と ASP.NET を処理できます。これにより、ASP.NET モバイル固有の開発の重要性が低くなっています。現在、ASP.NET for Mobile では、特定のマークアップのアダプターとデバイス プロファイルを通じて、さまざまなモバイル デバイスがサポートされています。ASP.NET for Mobile では、実行時にデバイスの機能に合わせてコンテンツが自動的にレンダリンクされますが、デバイス プロファイルのテストと管理に関連するオーバーヘッドが発生します。これらのコントロー

ルの開発のサポートは Microsoft Visual Studio 2003 と Visual Studio 2005 に含まれていますが、Visual Studio 2008 には含まれていません。現在、実行時のサポートは提供されていますが、今後廃止される可能性があります。詳細については、「関連情報」を参照してください。

.NET Compact Framework

Microsoft .NET Compact Framework を使用する場合は、次のガイドラインを考慮します。

- Microsoft .NET Framework になじみがあり、デスクトップとモバイルの両方のプラットフォーム用のアプリケーションを同時に開発する場合は、.NET Framework クラス ライブラリのサブセットである .NET Compact Framework の使用を検討します。.NET Compact Framework には、Windows Mobile 用に特別に設計されたいくつかのクラスが用意されています。.NET Compact Framework では、Microsoft Visual Basic® および Microsoft Visual C#® の開発システムのみがサポートされています。
- Visual Studio デバッガーを使用して Windows Mobile のコードのサブセットを追跡するのに問題がある場合は、複数のデバッグ セッションが必要になる可能性があることを考慮します。たとえば、デバッグ セッションにネイティブ コードとマネージ コードの両方が含まれている場合、Visual Studio ではコード間の境界を越えてセッションを続けることはできません。この場合は、実行中の 2 つの Visual Studio インスタンスが必要になり、これらのインスタンス間のコンテキストを手動で追跡する必要があります。

Windows Mobile

Windows Mobile アプリケーションに関しては、次の一般的なガイドラインを考慮します。

- Windows Mobile Professional と Windows Mobile Standard の両方のエディションに対応したアプリケーションを作成する必要がある場合は、Windows Mobile のバージョンによって Windows Mobile セキュリティ モデルが異なることを考慮します。プラットフォームによって API のセキュリティ モデルが異なるため、一方のプラットフォームで機能するコードが、もう一方のプラットフォームで機能しないことがあります。使用するデバイスとバージョンに関する Windows Mobile のドキュメントを確認します。この章の最後にある「関連情報」も参照してください。

- 今後アプリケーションを管理しなければならない場合や既存のアプリケーションをアップグレードする場合は、Windows Mobile オペレーティング システムの派生、製品の名前、およびバージョン ツリーについて理解するようにします。各バージョン間には、アプリケーションに影響を及ぼす可能性がある、わずかな違いがあります。
    - Windows Mobile は、Windows CE オペレーティング システムのリリースから派生しています。
    - Windows Mobile 5.x と 6.x はいずれも、Windows CE 5.0 に基づいています。
    - Windows Mobile Pocket PC は、Windows Mobile 6.0 のリリース時に Windows Mobile Professional と改名されました。
    - Windows Mobile Smartphone は、Windows Mobile 6.0 のリリース時に Windows Mobile Standard と改名されました。
    - Windows Mobile Professional と Windows Mobile Standard では、API がわずかに異なっています。たとえば、Windows Mobile Standard (Smartphone) では、データ入力にソフトキーが使用されるので、Compact Framework の実装に Button クラスがありません。
- メモリとファイルの構造にアクセスする際には、必ず Windows Mobile API を使用します。どちらか一方の構造へのハンドルを取得した後は、これらの構造に直接アクセスしないようにします。
  Windows CE のバージョン 6.x とそれ以降 (および Windows Mobile の次期リリース) では、仮想化されたメモリ モデルと、以前のバージョンとは異なるプロセスの実行モデルが使用されます。つまり、ファイルのハンドルやポインターなどの構造が、メモリへの実際の物理的なポインターではなくなる場合があります。Windows Mobile 6.x とそれ以前の、この実装に依存している Windows Mobile プログラムは、次のバージョンの Windows Mobile に移行したときに動作しなくなります。
- Active Directory インフラストラクチャを使用していれば、Mobile Device Manager (MDM) は、モバイル デバイスのログを承認、追跡、および収集するソリューションとして使用できます。管理されているデバイスの Windows Mobile 6.1 と同じように、MDM が完全に機能するには、次に示す他のいくつかの製品をサーバーにインストールする必要があります。
    - Windows Mobile 6.1 (デバイスにインストール)
    - Windows Server Update Service (WSUS) 3.0
    - Windows Mobile デバイス管理サーバー
    - 登録サーバー
    - ゲートウェイ サーバー

- Active Directory (Windows Server の一部として)
- SQL Server 2005 以上
- マイクロソフト証明機関
- インターネット インフォメーション サービス (IIS) 6.0
- .NET Framework 2.0 以上

Windows Embedded

Windows Embedded テクノロジを使用する際には、次のガイドラインを考慮します。

- セットトップ ボックスやその他のリソースの使用量が多いデバイス用に設計する場合は、Windows Embedded Standard の使用を検討します。
- 現金自動預払機 (ATM)、顧客が使用するキオスク、またはセルフ チェックアウト システムなどの POS デバイス用に設計する場合は、Windows Embedded for Point of Service の使用を検討します。
- GPS 対応デバイスやナビゲーション機能付きのデバイス用に設計する場合は、Microsoft Windows Embedded NavReady™ ソフトウェアの使用を検討します。Windows Embedded NavReady 2009 は Windows Mobile 5.0 ベースで構築されていますが、最新バージョンの Windows Mobile Standard と Windows Mobile Professional では Windows Mobile 6.1 が使用されています。NavReady デバイスとその他の Windows Mobile デバイスの両方に共通のコードベースを作成する場合は、両方のプラットフォームで利用できる API を使用します。

配置に関する考慮事項

モバイル アプリケーションは、さまざまな方法を使用して配置できます。配置を設計する際には、ユーザーの要件とアプリケーションの管理方法を考慮します。アプリケーションの配置に適した操作、管理、およびセキュリティを使用できるように設計します。Windows Mobile デバイス アプリケーションの配置シナリオを一般的なものから順に示します。

- Windows インストーラー ファイル (MSI) を使用して Microsoft Exchange ActiveSync® テクノロジでインストールします。
- 無線通信で、HTTP、SMS、または CAB ファイルを使用して、インストールと実行機能を提供します。

- Mobile Device Manager ベースで、Active Directory を使用して CAB ファイルまたは MSI ファイルから読み込みます。
- ポスト ロードと自動実行を使用して、企業固有のパッケージを、オペレーティング システムの一部として読み込みます。
- SD カードを手動で使用して、サイトを読み込みます。

配置に関する方針を設計する際には、次のガイドラインを考慮します。

- ユーザーが社外からアプリケーションをインストールおよび更新する必要がある場合は、無線通信による配置を設計することを検討します。
- CAB ファイルを使用して複数のデバイスにアプリケーションを配布する必要がある場合は、CAB ファイルに複数のデバイスの実行可能ファイルを含めるようにします。デバイスが、インストールする実行可能ファイルを検出して、残りのファイルを破棄するようにします。
- アプリケーションがホスト コンピューターに大きく依存している場合は、ActiveSync を使用してアプリケーションを配置することを検討します。
- Windows Mobile の基本的なエクスペリエンスを提供するアプリケーションを配置する場合は、ポスト ロードのメカニズムを使用して、Windows Mobile オペレーティング システムが起動したら、すぐにアプリケーションを自動的に読み込むことを検討します。
- 特定のサイトのみでアプリケーションが実行され、配布を手動で制御する必要がある場合は、SD メモリ カードを使用して配置することを検討します。

配置パターンとシナリオの詳細については、第 19 章「物理ティアと配置」を参照してください。

関連する設計パターン

次の表に示すように、主要なパターンは、キャッシュ、通信、データ アクセス、同期、UI などのカテゴリに分類されます。各カテゴリの設計を行う際には、これらのパターンの使用を検討します。

カテゴリ	関連するパターン
キャッシュ	**Lazy Acquisition**: デバイスにおけるリソースの使用を最適化するために、リソースの取得をできるだけ遅らせます。
通信	**Active Object**: サービス要求とサービスが完了したことを示す応答をカプセル化すること

	で、非同期処理をサポートします。 **Communicator**: 通信に関する内部の詳細を、さまざまなチャネルを通じて通信できる別個のコンポーネントにカプセル化します。 **Entity Translator**: 要求ではメッセージ データ型をビジネス型に変換し、応答では逆の変換を行うオブジェクトです。 **Reliable Sessions**: 送信元と送信先との間のエンド ツー エンドの信頼できるメッセージ送信です。エンドポイント間にある中間デバイスの数や種類は関係ありません。
データ アクセス	**Active Record**: ドメイン エンティティのデータ アクセス オブジェクトを含みます。 **Data Transfer Object (DTO)**: プロセス間で転送されるデータを格納して、必要なメソッド呼び出しの回数を削減するオブジェクトです。 **Domain Model**: ドメイン内のエンティティとエンティティ間の関係を表す一連のビジネス オブジェクトです。 **Transaction Script**: 各トランザクションのビジネス ロジックを 1 つのプロシージャにまとめて、データベースを直接または軽量なデータベース ラッパー経由で呼び出します。
同期	**Synchronization**: デバイスにインストールされるコンポーネントです。データへの変更を追跡し、ネットワーク接続を使用できるときに、サーバー上のコンポーネントと情報をやり取りします。
UI	**Application Controller**: すべてのフロー ロジックが含まれ、Model と連携して適切な View を表示する他の Controller によって使用されるオブジェクトです。 **Model-View-Controller**: ユーザーの入力に基づいて、ドメイン、プレゼンテーション、およびアクションのデータを、3 つの別個のクラスに分離します。Model では、アプリケーション ドメインの動作とデータを管理し、その状態に関する情報を求める (通常、View からの) 要求に応答して、状態を変更するための (通常、Controller からの) 命令に応答します。View では、情報の表示を管理します。Controller では、ユーザーからのマウスとキーボードによる入力を解釈し、必要に応じて、Model と View に変更を通知します。 **Model-View-Presenter**: 要求処理を 3 つの役割に分割します。View ではユーザーの入力を処理し、Model ではアプリケーション データとビジネス ロジックを管理し、Presenter ではプレゼンテーション ロジックを管理して View と Model 間の通信を調整します。 **Pagination**: 大量のコンテンツを別個のページに分割することで、システム リソースを最適化し、使用する画面領域を最小限に抑えます。

関連情報

Web リソースに簡単にアクセスするには、http://www.microsoft.com/architectureguide (英語) でオンライン版の参考文献を参照してください。

- Windows Embedded テクノロジのオプションの詳細については、「Windows Embedded デベロッパー センター」を参照してください。
- モバイル デバイス専用のソフトウェア ファクトリの詳細については、「patterns & practices Mobile Client Software Factory」(英語) を参照してください。
- Microsoft Sync Framework の詳細については、「Microsoft Sync Framework Developer Center」(英語) を参照してください。
- Smart Device Framework の OpenNETCF.Diagnostics.EventLog の詳細については、「Instrumentation for .NET Compact Framework Applications」(英語) を参照してください。
- ASP.NET for Mobile の詳細については、「Roadmap for ASP.NET Mobile Development」(英語) を参照してください。
- ASP.NET for Mobile のソース コードのサポートを Visual Studio 2008 に追加する方法の詳細については、「Tip/Trick: ASP.NET Mobile Development with Visual Studio 2008」(英語) を参照してください。
- Windows Mobile 6.x のセキュリティ モデルのアクセス許可に関する詳細については、「Security Model for Windows Mobile 5.0 and Windows Mobile 6」(英語) を参照してください。
- Apache Logging Services の log4net の詳細については、http://logging.apache.org/log4net/index.html (英語) を参照してください。
- Jarosław Kowalski が提供している NLog の詳細については、http://www.nlog-project.org/introduction.html (英語) を参照してください。
- OpenNetCF Community の詳細については、http://community.opennetcf.com/ (英語) を参照してください。

25

サービス アプリケーションの設計

概要

この章では、サービスの性質と使用方法、さまざまなサービスのシナリオに関する一般的なガイドライン、およびサービスの主要な特性について紹介します。また、サービス アプリケーションのレイヤーに関するガイドラインや、パフォーマンス、セキュリティ、配置、パターン、およびテクノロジに関する考慮事項の観点から検討する必要がある主な要素についても説明します。

サービスとは、機能の単位へのアクセスを提供するパブリック インターフェイスのことです。サービスでは、名前のとおりプログラムによるなんらかの "サービス" が、サービスを使用する呼び出し元に提供されます。サービスは疎結合されており、クライアント内または他のサービス内で組み合わせることで、より複雑な機能を提供できます。また、サービスは分散可能で、リモート コンピューターからも、サービスが実行されているコンピューターからもアクセスできます。サービスはメッセージ指向です。つまり、サービス インターフェイスは Web サービス記述言語 (WSDL) ファイルで定義され、操作はトランスポート チャネル経由で渡される拡張マークアップ言語 (XML) ベースのメッセージ スキーマを使用して呼び出されます。メッセージとインターフェイスの定義に重点を置いて相互運用性が確保されているので、サービスでは異種環境がサポートされます。コンポーネントでメッセージとインターフェイスの定義が認識されれば、コンポーネントが基づいているテクノロジに関係なくサービスを使用できます。図 1 に、一般的なサービス アプリケーション アーキテクチャの概要を示します。

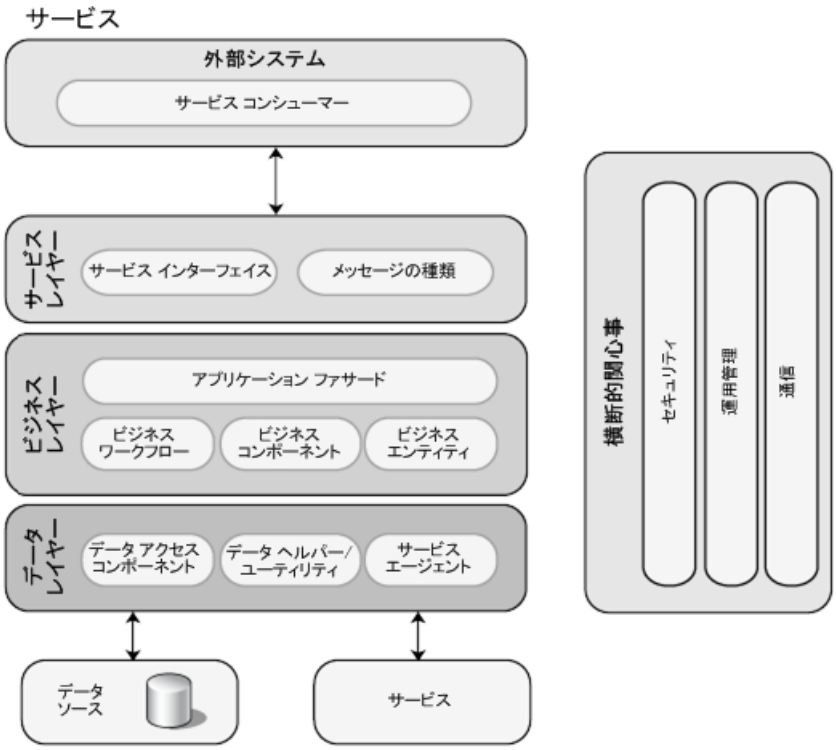


図 44

一般的なサービス アプリケーション アーキテクチャ

一般的なサービス アプリケーションは、サービス レイヤー、ビジネス レイヤー、およびデータ レイヤーの 3 つのレイヤーで構成されています。サービス レイヤーには、サービス インターフェイス、メッセージの種類、およびデータ型の各コンポーネントが、ビジネス レイヤーにはビジネス ロジック、ビジネス ワークフロー、およびビジネス エンティティの各コンポーネントが、データ レイヤーにはデータ アクセス コンポーネントとサービス エージェント コンポーネントが含まれる場合があります。レイヤー型の設計の詳細については、第 5 章「レイヤー型アプリケーションのガイドライン」を参照してください。各レイヤーに適したコンポーネントの詳細については、第 10 章「コンポーネントのガイドライン」を参照してください。

サービスは本質的に柔軟なので、さまざまなシナリオや組み合わせで使用できます。一般的なシナリオは、次のとおりです。

- **インターネット経由で公開されるサービス**: インターネット経由でさまざまなクライアントで使用されるサービスを表すシナリオです。このシナリオには、企業間サービスと、コンシューマー向けサービスが含まれます。株式取引の Web サービスを使用して株価情報を提供する株式仲買人用 Web サイトが、この一例です。認証と承認は、インターネットの信頼境界と資格情報のオプションに基づいて決定する

必要があります。たとえば、ユーザー名とパスワードによる認証や証明書の使用は、イントラネット シナリオよりも、インターネットのシナリオでよく使用されます。

- **イントラネット経由で公開されるサービス**: (通常、制限された) 社内クライアントや企業クライアントによって、イントラネット経由で使用されるサービスを表すシナリオです。エンタープライズ レベルのドキュメント管理アプリケーションが、この一例です。認証と承認は、イントラネットの信頼境界と資格情報のオプションに基づいて決定する必要があります。たとえば、Active Directory を使用する Windows 認証は、インターネット シナリオよりもイントラネットのシナリオでユーザー ストアとしてよく使用されます。
- **ローカル コンピューターで公開されるサービス**: ローカル コンピューターのアプリケーションで使用されるサービスを表すシナリオです。トランスポートとメッセージの保護は、ローカル コンピューターの信頼境界とユーザーに基づいて決定する必要があります。
- **混在するシナリオ**: インターネット、イントラネット、およびローカル コンピューター経由で、複数のアプリケーションによって使用されるサービスを表すシナリオです。リッチ クライアント アプリケーションによって社内で使用されるか、Web アプリケーションによってインターネット経由で使用される、基幹業務 (LOB) サービス アプリケーションが、この一例です。

設計に関する一般的な考慮事項

サービス ベースのアプリケーションを設計する際には、粒度の粗い操作の設計、サービス コントラクトの優先、無効な要求や不適切な順序で到着する要求の想定など、あらゆるサービスに適用する一般的なガイドラインに従う必要があります。一般的なガイドラインに加えて、さまざまなサービスで従う必要があるガイドラインも存在します。たとえば、サービス指向アーキテクチャ (SOA) では、操作がアプリケーションによってスコープされ、サービスが自律するようにする必要があります。また、ワークフロー サービスを提供するアプリケーションを使用したり、サービス ベースのインターフェイスを提供する運用データ ストアを設計する場合もあります。サービス アプリケーションを設計する際には、次のガイドラインを考慮します。

- **レイヤー型の手法を使用してサービス アプリケーションを設計し、レイヤー間の密結合を避ける**: 必要に応じて、ビジネス ルールとデータ アクセス機能を別個のコンポーネントに分離します。抽象化を使用して、ビジネス レイヤーにインターフェイスを提供します。この抽象化は、パブリック オブジェクト インターフェイス、一般的なインターフェイスの定義、抽象型基本クラス、またはメッセージを使

用して実装できます。レイヤー型アーキテクチャでの抽象化については、第 5 章「レイヤー型アプリケーションのガイドライン」を参照してください。

- **粒度の粗い操作を設計する**: パフォーマンスが大幅に低下する可能性がある、サービス インターフェイスへの粒度の細かい (chatty な) 呼び出しを回避します。サービスの操作は、粒度が粗く、アプリケーションの操作に焦点を合わせたものにする必要があります。Façade パターンを使用して、より小さく粒度の細かい複数の操作を、1 つの粒度の粗い操作にパッケージ化することを検討します。たとえば、人口統計データでは、データのサブセットを返す複数の呼び出しを行うのではなく、1 回の呼び出しですべてのデータを返す操作を提供する必要があります。
- **データ コントラクトを設計して、拡張性と再利用性を向上する**: データ コントラクトは、サービスのコンシューマーに影響を及ぼすことなく拡張できるように設計する必要があります。可能であれば、標準的な要素から、サービスで使用する複雑な型を構成します。サービス レイヤーは、ビジネス レイヤーで使用されるビジネス エンティティを把握する必要があります。一般的に、これは、サービス レイヤーとビジネス レイヤー間で共有されるビジネス エンティティを含む別個のアセンブリを作成することで実現できます。
- **サービス コントラクトのためだけの設計を行う**: サービス レイヤーでは、サービス コントラクトで指定されている機能のみを実装および提供する必要があり、内部の実装やサービスの詳細は、外部のコンシューマーに公開するべきではありません。また、サービスで実装する新機能を含むようにサービス コントラクトを変更する必要があり、新しい操作と型が既存のコントラクトとの下位互換性を持たない場合は、コントラクトをバージョン管理することを検討します。サービスで公開される新しい操作を新しいバージョンのサービス コントラクトで定義し、新しいスキーマ型を新しいバージョンのデータ コントラクトで定義します。メッセージ コントラクトの設計に関する詳細ついては、第 18 章「通信とメッセージ」を参照してください。
- **サービスが自律するように設計する**: サービスでは、サービスのコンシューマーに何かを要求するべきではなく、コンシューマーや提供するサービスをコンシューマーがどのように使用する予定であるかに関して、仮定を立てるべきではありません。
- **無効な要求を受信する可能性を想定して設計する**: サービスで受信するメッセージがすべて有効であるとは限りません。検証ロジックを実装して、すべてのメッセージを適切なスキーマに基づいてチェックし、すべての無効なメッセージを拒否したり、一部を削除したりします。検証の詳細については、第 17 章「横断的関心事」を参照してください。既知のパターンを実装するか、インフラストラクチャ サービスを使用し、重複するメッセージ (べき等性) をサービスで検出して管理できるようにすることで、重複するメッセージが処理されないようにします。また、メッセージを格納してから適切な順序で処理

する設計を実装するなどして、バラバラの順序で到着するメッセージ (交換性) をサービスで管理できるようにします。

- **明確な境界がある状態で、ポリシーに基づいてサービスを設計する**: サービス アプリケーションは独立し、厳密な境界がなければなりません。サービスへのアクセスは、サービス インターフェイス レイヤーを通じてのみ行えるようにする必要があります。サービスでは、コンシューマーがサービスをどのように操作できるかについて示したポリシーを公開する必要があります。これは、コンシューマーがポリシーを確認して操作の要件を特定できるので、一般に公開されるサービスでは特に重要です。
- **サービスに関する関心事をインフラストラクチャの運用に関する関心事から分離する**: 横断的なロジックは、アプリケーション ロジックと混在すべきではありません。ロジックが混在すると、拡張や保守が困難な実装になる場合があります。
- **サービス レイヤーの主要なコンポーネントに、別個のアセンブリを使用する**: たとえば、インターフェイス、実装、データ コントラクト、サービス コントラクト、エラー コントラクト、およびトランスレーターは、それぞれのアセンブリに分離する必要があります。
- **データベースに含まれる個々のテーブルの公開にデータ サービスを使用しないようにする**: データ サービスを使用してデータベース内の個々のテーブルを公開すると、粒度の細かい (chatty な) サービス呼び出しが発生し、サービス操作の間で相互依存性が生じるので、サービス コンシューマーの依存関係の問題が発生するおそれがあります。また、データのさまざまなコンシューマーでは独自のビューポイントや規則を持っており、サービス内にビジネス ルールを実装すると、データの使用に関する制約が課されるため、サービス内には、ビジネス ルールを実装しないようにします。
- **ワークフロー エンジンでサポートされているインターフェイスを使用するようにワークフロー サービスを設計する**: カスタム インターフェイスを作成しようとすると、サポートされる操作の種類が制限される場合があるので、結果として、サービスの拡張と管理に必要な作業が増えます。既存のサービス アプリケーションにワークフロー サービスを追加するのではなく、ワークフローの要件のみをサポートする自律したサービスを設計することを検討します。

設計に関する具体的な問題

次のセクションでは、サービスのアーキテクチャを開発する際に発生する一般的な問題を解決するのに役立つガイドラインを提供します。

- 認証

- 承認
- ビジネス レイヤー
- 通信
- データ レイヤー
- 例外管理
- メッセージの構築
- メッセージ エンドポイント
- メッセージの保護
- メッセージの変換
- メッセージ交換パターン
- Representational State Transfer
- サービス レイヤー
- SOAP
- 検証

認証

サービスの効果的な認証方針の設計は、使用しているサービス ホストの種類によって異なります。たとえば、サービスをインターネット インフォメーション サービス (IIS) でホストしている場合は、IIS で提供されている認証のサポートを使用できます。Windows サービスを使用してサービスをホストしている場合は、メッセージ ベースの認証かトランスポート ベースの認証を使用する必要があります。認証の方針を設計する際には、次のガイドラインを考慮します。

- ユーザーを安全に認証するための適切なメカニズムを特定して、すべての信頼境界にわたって認証を適用します。必要に応じて、統合されたサービスやシングル サインオン (SSO) メカニズムの使用を検討します。
- 異なる信頼設定を使用するとサービス コードの実行にどのような影響があるかを考慮します。
- 基本認証を使用する場合、または資格情報がプレーンテキスト形式で渡される場合は、Secure Sockets Layer (SSL) などのセキュリティで保護されたプロトコルを使用するようにします。また、Web Services Security (WS-Security) などのセキュリティ メカニズムを SOAP メッセージで使用します。

承認

効果的な承認方針を設計することは、サービス アプリケーションのセキュリティと信頼性の両方において重要です。適切な承認方針を設計しないと、情報漏えい、データの改ざん、および特権の昇格に対して、アプリケーションが脆弱になる可能性があります。承認の方針を設計する際には、次のガイドラインを考慮します。

- ユーザー、グループ、および役割ごとに、リソースに適切なアクセス許可を設定し、すべての信頼境界で粒度の細かいレベルの承認を行うようにします。最も権限の低い適切なアカウントでサービスを実行します。
- URL ベースおよびファイル ベースのリソースを保護する場合は、Uniform Resource Locator (URL) 承認またはファイルの承認、あるいはその両方を使用することを検討します。
- 必要に応じて、宣言型の主要なアクセス許可の要求を使用して、一般に公開されているサービス メソッドへのアクセスを制限します。

ビジネス レイヤー

サービス アプリケーションのビジネス レイヤーでは、ファサードを使用して、サービス操作をビジネス操作に変換します。サービス インターフェイスを設計する主な目的は、内部で複数のビジネス操作に変換できる粒度の粗い操作を使用することです。ビジネス レイヤーのファサードには、適切なビジネス プロセス コンポーネントと通信する役割があります。ビジネス レイヤーを設計する際には、次のガイドラインを考慮します。

- ビジネス レイヤーのコンポーネントは、サービス レイヤーについて把握しないようにする必要があります。ビジネス レイヤーとすべてのビジネス ロジック コードは、サービス レイヤーのコードに依存すべきではなく、サービス レイヤーのコードを実行すべきではありません。
- サービスをサポートする際には、ビジネス レイヤーのファサードを使用します。ファサードはビジネス レイヤーの主要なエントリ ポイントを表し、粒度の粗い操作を受け取って、その操作を複数のビジネス操作に分割します。ただし、ローカル コンピューターまたは物理的な境界を越えてサービスにアクセスしないクライアントがサービスにアクセスする場合、クライアントにとって有益ならば、粒度の細かい操作の公開も検討することをお勧めします。
- ビジネス レイヤーがステートレスになるように設計します。サービス操作には、ビジネス レイヤーが要求の処理に使用する状態情報など、すべての情報を含める必要があります。サービスでは、多数のコンシューマーとの通信を処理できるので、ビジネス レイヤーで状態を管理しようとすると、各コンシ

ユーマーの状態を管理するために、かなりの量のリソースが使用されます。そのため、サービスで常時処理できる要求の数が制限されます。

- すべてのビジネス エンティティを、別個のアセンブリに実装します。このアセンブリは、ビジネス レイヤーとデータ アクセス レイヤーの両方で使用できる共有コンポーネントを表します。ただし、ビジネス エンティティは、サービス境界を越えて公開するべきではなく、データ転送オブジェクト (DTO) を使用して、サービス間でデータを転送する必要があります。

ビジネス レイヤーの実装に関する詳細については、第 7 章「ビジネス レイヤーのガイドライン」を参照してください。ビジネス エンティティの詳細については、第 13 章「ビジネス エンティティの設計」を参照してください。

通信

サービス アプリケーションの通信方針を設計する場合は、サービスの配置シナリオに基づいてプロトコルを選択する必要があります。通信の方針を設計する際には、次のガイドラインを考慮します。

- サービスを閉じたネットワーク内に配置する場合は、通信効率の高い伝送制御プロトコル (TCP) を使用できます。サービスを一般に公開されるネットワークに配置する場合は、ハイパーテキスト転送プロトコル (HTTP) を使用することをお勧めします。
- 不安定な通信や断続的な通信の処理方法を決定します。考えられる処理方法としては、メッセージをキャッシュして、接続が使用できるようになったときにメッセージを送信するという方法があります。また、非同期呼び出しの処理方法を決定します。メッセージ通信を一方向と両方向のどちらにする必要があるかを判断し、Duplex、Request Response、および Request-Reply の各パターンを使用する必要があるかどうかを判断します。
- 柔軟性を最大限に高めるために、URL の動的な動作を構成したエンドポイントを使用して、メッセージのエンドポイント アドレスの検証方法を決定します。
- 適切な通信プロトコルを選択し、暗号化とデジタル署名を使用して、通信チャネル間で送信されるデータを保護するようにします。

通信プロトコルと通信技法の詳細については、第 18 章「通信とメッセージ」を参照してください。

データ レイヤー

サービス アプリケーションのデータ レイヤーには、外部データ ソースと通信するデータ アクセス機能が含まれます。この外部データ ソースは、データベース、その他のサービス、ファイル システム、SharePoint リスト、また

はデータを管理するその他のアプリケーションの場合があります。データの一貫性は、サービスの実装の安定性と整合性において重要です。サービスで受信したデータの一貫性を検証できないと、データ ストアへの無効なデータの挿入、予期しない例外、およびセキュリティ侵害が発生するおそれがあります。そのため、サービスを実装する場合は、データの整合性チェックを必ず含める必要があります。データ レイヤーを設計する際には、次のガイドラインを考慮します。

- 可能な場合は、データ レイヤーをビジネス レイヤーと同じティアに配置する必要があります。データ レイヤーを別個の物理ティアに配置すると、物理的な境界を越えるときにオブジェクトをシリアル化する必要があります。
- データ レイヤーにインターフェイスを実装する場合は、必ず抽象化を使用します。これは、一般的に、入出力に既知の型を使用する、Data Access パターンや Table Data Gateway パターンを使用して実現します。
- 単純な作成、読み取り、更新、および削除 (CRUD) 操作では、データベース内の各テーブルまたはビュー用にクラスを作成することを検討します。これは、(各テーブルが、対応するクラスおよびテーブルと通信する操作を保持する) Table Module パターンを表します。トランザクションの処理方法を計画します。
- 偽装やデリゲートを使用してデータベースにアクセスしないようにします。代わりに、一般的なエンティティを使用してデータベースにアクセスする一方で、ユーザーの ID 情報を提供して、ログ記録と監査のプロセスで、ユーザーとそのユーザーが実行した操作を関連付けられるようにします。

データ レイヤーの実装に関する詳細については、第 8 章「データ レイヤーのガイドライン」を参照してください。

例外管理

効果的な例外管理方針を設計することは、サービス アプリケーションのセキュリティと信頼性の両方において重要です。正しく設計しないと、アプリケーションがサービス拒否 (DoS) 攻撃に対して脆弱になったり、機密情報や重要な情報が開示されたりする場合があります。例外の発生とハンドルは負荷の高い操作なので、例外管理を設計する際には、パフォーマンスへの影響を考慮する必要があります。良い方法は、例外管理とログ記録の一元化されたメカニズムを設計し、システム管理者を支援するために、インストルメンテーションと一元的な監視をサポートするアクセス ポイントの提供を検討することです。例外管理の方針を設計する際には、次のガイドラインを考慮します。

- ハンドルされない例外をキャッチして、例外が発生した後にリソースがクリーンアップされるようにします。サービスの例外、ログ ファイル、および監査ファイルに機密データを含めないようにします。

- 例外をハンドルできる場合 (たとえば、機密情報を削除したり、例外に情報を追加したりする場合) を除き、例外はキャッチしないようにします。アプリケーション ロジックのフローを制御するために、例外を使用しないでください。必要でない場合は、カスタムの例外を使用しないようにします。
- SOAP エラー要素またはカスタムの拡張を使用して、例外の詳細を呼び出し元に返します。

例外管理方針の設計の詳細については、第 17 章「横断的関心事」を参照してください。

メッセージの構築

サービスとコンシューマーの間でデータをやり取りする際、データはメッセージ内にラップされている必要があります。このメッセージの形式は、サポートする必要がある操作の種類によって異なります。操作の例には、ドキュメントのやり取り、コマンドの実行、イベントの生成などがあります。低速なメッセージ配信チャネルを使用する場合は、メッセージに有効期限の情報を含めることも検討する必要があります。メッセージ構築の方針を設計する際には、次のガイドラインを考慮します。

- メッセージ構築に関連するパターン (Command Message、Document Message、Event Message、Request-Reply など) を特定します。
- 膨大な量のデータを比較的小さなチャンクに分割して、順に送信します。
- 時間に制約のあるメッセージには有効期限情報を含めます。サービスでは、期限切れのメッセージを無視するようにします。

メッセージ エンドポイント

メッセージ エンドポイントは、アプリケーションがサービスとの通信に使用する接続を表します。サービス インターフェイスの実装によって、メッセージ エンドポイントが提供されます。サービスの実装を設計する際には、使用しているメッセージの種類を考慮する必要があります。また、メッセージの処理に関連するさまざまなシナリオを考慮して設計する必要があります。メッセージ エンドポイントを設計する際には、次のガイドラインを考慮します。

- メッセージ エンドポイントに関連するパターン (Messaging Gateway、Messaging Mapper、Competing Consumer、Message Dispatcher など) を特定します。
- 非接続型のシナリオを考慮して設計を行います。たとえば、後で配信できるようにメッセージをキャッシュまたは格納して、確実な配信をサポートすることが必要な場合があります。接続が切断されている間は、エンドポイントをサブスクライブしないようにする必要があります。

- すべてのメッセージを受け入れるのか、特定のメッセージを処理するフィルターを実装する必要があるのかを判断します。
- サービス インターフェイスのべき等性を考慮して設計を行います。べき等性とは、同じコンシューマーから重複するメッセージを受信する可能性があるが、その中の 1 つだけを処理する必要がある状況を指します。つまり、べき等性を考慮したエンドポイントでは、1 つのメッセージだけが処理され、重複するメッセージはすべて無視されることを保証します。
- 交換性を考慮して設計を行います。交換性とは、メッセージがバラバラの順序で到着する可能性がある状況を指します。つまり、交換性を考慮したエンドポイントでは、バラバラの順序で到着したメッセージが格納され、適切な順序で処理されることを保証します。

メッセージの保護

サービスとコンシューマーの間で機密データを転送する場合は、メッセージの保護を考慮した設計を行う必要があります。トランスポート レイヤーの保護 (IPSec、SSL など) またはメッセージ ベースの保護 (暗号化、デジタル署名など) を使用できます。メッセージの保護を設計する際には、次のガイドラインを考慮します。

- エンド ツー エンドのセキュリティを確保する必要があって、サービスと呼び出し元の間にサーバーやルーターなどの中間デバイスを配置できる場合は、メッセージ ベースのセキュリティを使用します。メッセージ ベースのセキュリティは、メッセージに含まれる機密データを暗号化して保護するのに役立ち、デジタル署名は、メッセージの否認と改ざんを防ぐのに役立ちます。ただし、セキュリティ対策がパフォーマンスに影響することに注意してください。
- サービスとコンシューマーの間の通信が中間デバイス (他のサーバーやルーターなど) 経由でルーティングされない場合は、トランスポート レイヤーのセキュリティ (IPSec や SSL など) を使用できます。しかし、メッセージが中間デバイスを経由する場合は、必ず、メッセージ ベースのセキュリティを使用します。トランスポート レイヤーのセキュリティを使用すると、メッセージは、経由する中間デバイスで復号化されてから暗号化されるので、これはセキュリティ リスクとなります。
- セキュリティを最大限に高めるために、設計では、トランスポート レイヤーのセキュリティとメッセージ ベースのセキュリティの両方を使用することを検討します。トランスポート レイヤーのセキュリティは、メッセージ ベースのセキュリティを使用して暗号化できないヘッダー情報を、保護するのに役立ちます。
- エクストラネット サービスや企業間サービスを設計する場合は、X.509 証明書で仲介されたメッセージ ベースの認証を使用することを検討します。企業間のシナリオでは、証明書を販売する証明機関で、

証明書が発行されます。エクストラネット サービスでは、組織の証明書サービスで発行された証明書を使用できます。

メッセージの変換

サービスとコンシューマーの間でメッセージをやり取りする際には、メッセージをコンシューマーが認識できる形式に変換しなければならない場合がよくあります。この状況は、通常、メッセージングを使用できないコンシューマーが、メッセージ ベースのシステムから取得したデータを処理する必要がある場合に発生します。アダプターを使用して、このようなコンシューマーにメッセージ チャネルへのアクセスを提供することができます。また、トランスレーターを使用して、メッセージのデータを、各コンシューマーが認識できる形式に変換することもできます。メッセージの変換を設計する際には、次のガイドラインを考慮します。

- 変換を実行する必要があるかどうかを判断し、実行する必要がある場合は、メッセージの変換に関連するパターンを特定します。たとえば、Normalizer パターンを使用すると、意味が同じメッセージを一般的な形式に変換できます。必要でない場合は、正規モデルを使用しないようにします。
- 適宜変換を実行して、繰り返しの処理や不要なオーバーヘッドを回避します。
- メタデータを使用してメッセージ形式を定義し、このメタデータを外部リポジトリに格納することを検討します。
- さまざまな形式とクライアントの種類の間でメッセージを変換できる、BizTalk Server などのメカニズムの使用を検討します。

メッセージ交換パターン

メッセージ交換パターンでは、サービスとサービス コンシューマー間の通信を定義します。この通信は、サービスとコンシューマーの間のコントラクトを表します。W3C 標準化団体によって、Request-Response および SOAP Response という SOAP メッセージ用の 2 つのパターンが定義されています。また、OASIS という名前の別の標準化団体によって、サービス向けに Business Process Execution Language (BPEL) という言語が定義されています。BPEL では、ビジネス プロセスでメッセージを交換するためのプロセスが定義されています。また、その他の団体でも、ビジネス プロセスでメッセージを交換するためのメッセージ交換パターンが定義されています。メッセージ交換パターンを設計する際には、次のガイドラインを考慮します。

- 不要な複雑さを増すことなく、要件を満たすパターンを選択します。たとえば、Request-Response パターンで十分に対応できる場合は、複雑なビジネス プロセスのメッセージ交換パターンを使用しないようにします。一方向のメッセージでは、Fire and Forget パターンの使用を検討します。
- ビジネス プロセスのモデル化の技法を使用する場合は、プロセスの手順に基づいてメッセージ交換パターンを設計しないように注意します。メッセージ交換パターンでは、プロセスの手順を組み合わせた操作をサポートする必要があります。
- 独自のメッセージ交換パターンを作成するのではなく、既存のメッセージ交換パターンの標準を使用します。このようにすると、多くのコンシューマーで認識される、標準ベースのインターフェイスが実現します。つまり、コンシューマーは、標準に準拠していないコントラクトを検出して適応しなければならない状況よりも、標準ベースのコントラクトと通信する方を好みます。

Representational State Transfer

Representational State Transfer (REST) は、HTTP ベースのアーキテクチャ スタイルで、Web アプリケーションとほとんど同じように機能します。ただし、ユーザーが Web ページを操作して移動するのではなく、Web アプリケーションと同じセマンティクスを使用して、アプリケーションが REST リソースを操作して移動します。REST では、Uniform Resource Identifier (URI) でリソースを特定し、リソースに対して実行できる操作が HTTP 動詞 (GET、POST、PUT、および DELETE) を使用して定義されています。REST サービスとの通信は、URI に対して HTTP 操作を実行することで行います。この HTTP 操作は、HTTP ベースの URL の形式で行われることが一般的です。この操作により、リソースに関する現在の状態の表現が提供されます。また、この表現には、現在のリソースから移動できる他のリソースのリンクが含まれることがあります。

REST について最も一般的な誤解は、リソースに対する CRUD 操作でしか REST が役に立たないというものです。REST は、ステート マシンとして表すことができるすべてのサービスで使用できます。つまり、サービスを区別できる状態 (取得済みや更新済みなど) に分類できるなら、その状態を操作に変換して、各状態が別の状態に遷移するしくみを示すことができます。Web アプリケーションの UI でステート マシンがどのように表されるかを考慮します。ページにアクセスすると、表示される情報は、その情報の現在の状態を表します。フォーム フィールドを POST するか、現在のページに含まれているリンクを使用して別のページに移動することで、その状態を変更できる場合があります。アプリケーションがリソースに対して GET 操作を実行して、そのリソースの現在の状態を取得したり、PUT 操作を実行してリソースの状態を変更したり、現在のリソースで提供されているリンクを使用して別のリソースに移動したりできる点で、RESTful サービスも同じように機能します。

RESTful サービスには、アプリケーションの状態とリソースの状態の両方が存在します。クライアントでは、すべてのアプリケーションの状態を格納しますが、サーバーはリソースの状態しか格納しません。クライアントからサーバーに送信される各要求には、サーバーが要求を完全に理解するために必要なすべての情報が含まれていなければなりません。つまり、クライアントは、要求を送信する際に、すべての関連するアプリケーションの状態を転送する必要があります。その後、クライアントは、サーバーの応答に基づいて、リソースの状態をどのように変更するかを決定できます。クライアントがある特定のサーバーとのアフィニティや共有のアプリケーションの状態を必要としない方法で、同じ構成の Web サーバーを複数追加して負荷分散できるので、アプリケーションの状態を毎回渡すことで、アプリケーションの設計を拡張できます。

REST スタイルのサービスには、安全性とべき等性という性質があります。安全性とは、要求を複数回繰り返し行った場合に、副作用が発生することなく同じ応答を取得できることを指します。べき等性とは、1 回の呼び出しを行った場合の結果が、同じ呼び出しを何度も行った場合の結果と同じになる動作のことです。サーバーが応答しない場合やエラーを返す場合に、HTTP 接続が不安定でも、要求を安全に再発行できるので、安全性とべき等性の性質により、堅牢性と信頼性が強化されます。

REST リソースを設計する際には、次のガイドラインを考慮します。

- ステート図を使用して、REST サービスでサポートされるリソースをモデル化および定義することを検討します。REST サービスではセッション状態を使用しないようにします。
- リソースの特定手法を選択します。REST の URI の開始位置に意味のある名前を使用し、全体的なパスの一部として、特定のリソース インスタンスに一意識別子を使用することをお勧めします。QueryString 値を使用して操作を URI に含めることは避けます。
- 異なるリソース用に複数の表現をサポートする必要があるかどうかを判断します。たとえば、リソースで XML 形式、Atom 形式、または JavaScript Object Notation (JSON) 形式をサポートする必要があるかどうかを判断し、それをリソース要求に含めます。リソースは両方の形式で公開できます (たとえば、http://www.contoso.com/example.atom と http://www.contoso.com/example.json で公開できます。これは、プレースホルダーとして使用しているサイトへのリンクです)。QueryString 値を使用して、URI の操作を定義しないようにします。すべての操作は、URI に対して実行される HTTP 操作に基づいて定義します。
- POST 操作を乱用しないようにします。必要に応じて、特定の HTTP 操作 (PUT、DELETE など) を使用して、リソース ベースの設計と一貫したインターフェイスの使用を強化することをお勧めします。
- HTTP のアプリケーション プロトコルを使用して、一般的な Web インフラストラクチャ (キャッシュ、認証、一般的なデータ表現の種類など) を使用します。

- GET 要求が安全に行われるようにします (つまり、呼び出したときに、必ず同じ結果が返されるようにします)。PUT 要求と DELETE 要求をべき等にする (つまり、まったく同じ要求を繰り返し行ったときの結果が、1 回の要求を行ったときの結果と同じになるようにする) ことを検討します。

サービス レイヤー

サービス レイヤーには、アプリケーションのサービスを定義して実装するコンポーネントが含まれます。具体的には、(コントラクトで構成される) サービス インターフェイス、サービスの実装、および内部のビジネス エンティティと外部のデータ コントラクト間で変換を行うトランスレーターが含まれています。サービス レイヤーを設計する際には、次のガイドラインを考慮します。

- サービス レイヤーにビジネス ルールを実装することは避けます。サービス レイヤーでは、サービス要求を管理することと、ビジネス レイヤーで使用するためにデータ コントラクトをエンティティに変換することのみを行う必要があります。
- サービス レイヤーへのアクセスは、ポリシーで定義する必要があります。ポリシーでは、サービス コンシューマーが、接続とセキュリティに関する要件と、サービスとの通信に関連する他の詳細を判断するための方法が提供されます。
- サービス レイヤーの主要なコンポーネントに、別個のアセンブリを使用します。たとえば、インターフェイス、実装、データ コントラクト、サービス コントラクト、エラー コントラクト、およびトランスレーターは、それぞれのアセンブリに分離する必要があります。
- ビジネス レイヤーとの通信は、既知のパブリック インターフェイス以外を通じて行わないようにする必要があります。サービス レイヤーでは、基盤となるビジネス ロジック コンポーネントを呼び出すべきではありません。
- サービス レイヤーは、ビジネス レイヤーで使用されるビジネス エンティティを把握する必要があります。一般的に、これは、サービス レイヤーとビジネス レイヤー間で共有されるビジネス エンティティを含む別個のアセンブリを作成することで実現できます。

SOAP

SOAP は、メッセージがヘッダーと本文を含む XML エンベロープで構成されたメッセージ ベースのプロトコルです。ヘッダーでは、サービスで実行される操作の範囲外にある情報を提供できます。たとえば、ヘッダーには、セキュリティ、トランザクション、またはルーティングに関する情報が含まれる場合があります。本文には、サービスと

そのサービスで実行できる操作を定義するコントラクトが XML スキーマの形で含まれます。REST に比べて、SOAP ではプロトコルの柔軟性が高いので、TCP など、より高度なパフォーマンスのプロトコルを使用できます。SOAP では、セキュリティ、トランザクション、および信頼性を備えた WS-* 標準がサポートされています。メッセージのセキュリティと信頼性により、メッセージが送信先に配信されるだけでなく、メッセージが送信中に読み取られたり変更されたりしないことを保証できます。トランザクションでは、操作のグループ化機能と、操作が失敗した場合のロールバック機能が提供されます。

SOAP は、サービス間やアプリケーションの分離されたレイヤー間で RPC 型の通信を行う際に役立ち、内部ネットワークにある新しいシステムと従来のシステムの間にインターフェイスを提供することに優れています。サービス レイヤーは、従来のシステムをベースに配置できるので、システムを再設計して REST のリソース モデルを公開する必要なく、システムとの API 型の通信を行えます。SOAP は、システムが稼動している間に通信プロトコルが頻繁に変わる可能性のある 1 つ以上のシステムに、情報が動的にルーティングされる場合にも役に立ちます。また、情報やオブジェクトをあいまいな方法でカプセル化し、その情報を別のシステムに格納または中継する必要がある場合にも役立ちます。

Web ファームや負荷分散を使用してアプリケーションにスケーラビリティを持たせる必要がある場合は、セッション状態をサーバーに格納しないようにします。セッション状態をサーバーに格納すると、セッションを通じて特定のサーバーが特定のクライアントにサービスを提供する必要があり、負荷分散を行う場合には、セッション情報を別のサーバーに渡す必要が生じます。サーバー間でセッション状態を渡すと、フェールオーバーやスケールアウトのシナリオの実装がはるかに難しくなります。

SOAP メッセージを設計する際には、次のガイドラインを考慮します。

- SOAP エンベロープでは、SOAP ヘッダーは省略可能ですが、SOAP 本文は必ず含める必要があります。すべてのメッセージ スキーマで、複雑な型を使用しないようにします。
- エラーには、既定のエラー ハンドル動作ではなく、SOAP フォールトの使用を検討します。エラー情報を返す場合は、SOAP フォールトは SOAP 本文内の唯一の子要素である必要があります。
- SOAP ヘッダー ブロックを強制的に処理するには、ブロックの mustUnderstand 属性に true または 1 を設定します。SOAP ヘッダーの処理中に発生するエラーは、SOAP ヘッダー要素で SOAP フォールトとして返されます。
- WS-* 標準を調査して使用します。これらの標準を使用すると、メッセージング アーキテクチャでよく発生する問題に対処する一貫性のある規則と方法が提供されます。

検証

入力とデータの検証に関する効果的な方針を設計することは、アプリケーションのセキュリティを確保するうえで重要です。サービスのコンシューマーから受信したデータの検証規則を決定する必要があります。検証の方針を設計する際には、次のガイドラインを考慮します。

- サービス インターフェイスで受信したすべてのデータ (メッセージに関連付けられているデータ フィールドを含む) を検証し、検証で問題が発生した場合は、必要な情報を提供するエラー メッセージを返します。受信メッセージの検証に XML スキーマを使用することを検討します。
- 危険性や悪意があるコンテンツのすべての入力を確認して、そのデータが使用される方法を考慮します。たとえば、データベース クエリの開始にデータが使用される場合は、SQL インジェクション攻撃からデータベースを保護する必要があります。この攻撃では、ストアド プロシージャやパラメーター化されたクエリを使用して、データベースにアクセスする場合があります。
- 署名、暗号化、およびエンコードの方針を決定します。
- レイヤーと他の場所の間の信頼境界を理解して、このような信頼境界を越えるデータを検証できるようにします。ただし、その他のレイヤーで行われる検証で問題なく対応できるかどうかを判断します。データが既に信頼されている場合は、再度検証する必要はありません。

検証の技法に関する詳細については、第 17 章「横断的関心事」を参照してください。

テクノロジに関する考慮事項

サービスを設計する際には、次のテクノロジに関する事項を考慮する必要があります。

- 簡略化のために ASMX を使用することを検討します (ただし、IIS を実行している Web サーバーを使用できる場合に限ります)。
- 信頼できるセッションとトランザクション、アクティビティのトレース、メッセージのログ記録、パフォーマンス カウンター、複数のトランスポート プロトコルのサポートなどの高度な機能が必要な場合は、WCF サービスの使用を検討します。
- ASP.NET Web サービスを使用していて、メッセージ ベースのセキュリティとバイナリ データの転送が必要な場合は、Web サービス拡張 (WSE) の使用を検討します。
- WCF を使用する場合は、次の事項を考慮します。

- WCF 以外のクライアントや Windows 以外のクライアントとの相互運用性が必要な場合は、SOAP 仕様に基づく HTTP トランスポートを使用することを検討します。
- イントラネット クライアントをサポートする必要がある場合は、TCP プロトコルとバイナリ メッセージ エンコードをトランスポート セキュリティと Windows 認証と共に使用することを検討します。
- 同じコンピューター上の WCF クライアントをサポートする必要がある場合は、名前付きパイプ プロトコルとバイナリ メッセージ エンコードを使用することを検討します。
- 暗黙のメッセージ ラッパーではなく明示的なメッセージ ラッパーを使用するサービス コントラクトを定義することを検討します。これにより、メッセージ コントラクトを操作の入出力として定義することが可能になります。つまり、サービス コントラクトに影響を及ぼすことなく、メッセージ コントラクトに含まれるデータ コントラクトを拡張できます。

配置に関する考慮事項

通常、サービス アプリケーションは、サービス インターフェイス レイヤー、ビジネス レイヤー、およびデータ レイヤーを分離したレイヤー型の手法を使用して設計します。その他のアプリケーションの種類とまったく同じ方法で、サービス アプリケーションの分散配置を使用できます。サービスは、社内の 1 台のクライアント、1 台のサーバー、または複数のサーバーに配置できます。ただし、サービス アプリケーションを配置する場合は、分散シナリオに固有のパフォーマンスとセキュリティの問題について考慮し、運用環境で課される制限に配慮する必要があります。サービス アプリケーションを配置する際には、次のガイドラインを考慮します。

- アプリケーションのパフォーマンスを向上するため、可能な場合は、サービス レイヤーをビジネス レイヤーと同じティアに配置します。
- サービスがサービス コンシューマーと同じ物理ティアに配置されている場合は、名前付きパイプや共有メモリを使用して通信することを検討します。
- ローカル ネットワーク内の他のアプリケーションのみがサービスにアクセスする場合は、通信に TCP を使用することを検討します。
- コンシューマーがサービスに直接アクセスし、要求が中間デバイスを経由しない場合に限り、サービス ホストを構成して、トランスポート レイヤーのセキュリティを使用します。
- 開発とテスト以外では、すべてのサービスに関して、トレースとデバッグ モードのコンパイルを無効にします。

配置パターンとシナリオの詳細については、第 19 章「物理ティアと配置」を参照してください。

関連する設計パターン

次の表に示すように、主要なパターンは、通信、データの一貫性、メッセージの構築、メッセージ エンドポイント、メッセージの保護、メッセージの交換、REST、サービス インターフェイス、SOAP などのカテゴリに分類されます。各カテゴリの設計を行う際には、これらのパターンの使用を検討します。

カテゴリ	関連するパターン
通信	**Duplex**: サービスとクライアントの両方が相互に独立してメッセージを送信する双方向のメッセージ通信です。One-Way パターンと Request-Reply パターンのどちらが使用されるかは関係ありません。 **Fire and Forget**: 応答が期待されない場合に使用される、一方向のメッセージ通信メカニズムです。 **Reliable Sessions**: 送信元と送信先との間のエンド ツー エンドの信頼できるメッセージ送信です。エンドポイント間にある中間デバイスの数や種類は関係ありません。 **Request Response**: クライアントが送信したメッセージすべてに対して応答を受信することを期待する、双方向のメッセージ通信メカニズムです。
データの一貫性	**Atomic Transactions**: 1 つのサービス操作を対象とするトランザクションです。 **Cross-service Transactions**: 複数のサービスをまたぐことができるトランザクションです。 **Long-running transactions**: ワークフロー プロセスの一部になっているトランザクションです。
メッセージの構築	**Command Message**: コマンドをサポートするために使用するメッセージ構造です。 **Document Message**: アプリケーション間でドキュメントやデータ構造を信頼できる方法で転送するために使用する構造です。 **Event Message**: アプリケーション間の信頼できる非同期のイベント通知を提供する構造です。 **Request-Reply**: 要求と応答の送信に別個のチャネルを使用します。

メッセージ エンドポイント	**Competing Consumer**: 1 つのメッセージ キューに複数のコンシューマーを設定し、これらのコンシューマーがメッセージを処理する権利を奪い合うようにします。これにより、メッセージング クライアントは複数のメッセージを同時に処理できるようになります。 **Durable Subscriber**: 非接続型のシナリオでは、メッセージが確実に配信されるようにするため、メッセージは、保存されて、メッセージ チャネルへの接続時にクライアントからアクセスできるようになります。 **Idempotent Receiver**: サービスがメッセージを 1 回しか処理しないようにします。 **Message Dispatcher**: 複数のコンシューマーにメッセージを送信するコンポーネントです。 **Messaging Gateway**: メッセージ ベースの呼び出しを、残りのアプリケーション コードから分離するために 1 つのインターフェイスにカプセル化します。 **Messaging Mapper**: 受信メッセージの要求をビジネス オブジェクトに変換します。また、逆に、ビジネス オブジェクトを応答メッセージに変換する処理も行います。 **Polling Consumer**: メッセージの有無を確認するために定期的にチャネルをチェックするサービス コンシューマーです。 **Service Activator**: 非同期要求を受信してビジネス コンポーネントの操作を呼び出すサービスです。 **Selective Consumer**: サービス コンシューマーでは、フィルターを使用して、特定の条件を満たすメッセージを受信します。 **Transactional Client**: サービスと通信する際にトランザクションを実装できるクライアントです。
メッセージの保護	**Data Confidentiality**: メッセージ ベースの暗号化を使用してメッセージに含まれる機密データを保護します。 **Data Integrity**: メッセージが伝送中に改ざんされないようにします。 **Data Origin Authentication**: 高度な形態のデータ整合性を維持するために、メッセージの送信元を検証します。 **Exception Shielding**: 例外が発生したときに、サービスの内部実装に関する情報が公開されないようにします。 **Federation**: 複数のサービスやコンシューマーに分散した情報の統合ビューです。

	Replay Protection: 攻撃者がメッセージを傍受して何度も実行するのを防ぐことにより、メッセージのべき等性を維持します。 **Validation**: メッセージのコンテンツとメッセージに含まれる値をチェックして、不適切な形式のコンテンツや悪意のあるコンテンツからサービスを保護します。
メッセージの変換	**Canonical Data Mapper**: 共通のデータ形式を使用して、種類の異なる 2 つのデータ形式間の変換を実行します。 **Claim Check**: 必要に応じて永続的なストアからデータを取得します。 **Content Enricher**: メッセージに不足している情報を、外部データ ソースから取得して加えるコンポーネントです。 **Content Filter**: メッセージから機密データを取り除きます。また、メッセージから不要なデータを取り除くことでネットワーク トラフィックを削減します。 **Envelope Wrapper**: メッセージの保護、ルーティング、認証などに使用するヘッダー情報を含む、メッセージのラッパーです。 **Normalizer**: 組織でさまざまな形式のデータを使用する場合に、データを共通の交換形式に変換します。
REST	**Behavior**: 操作を実行するリソースに適用します。通常、このようなリソースでは、独自の状態を保持しておらず、POST 操作しかサポートしていません。 **Container**: エンティティ パターンを基盤として、入れ子になったリソースを動的に追加/更新する手段を提供します。 **Entity**: 読み取りは GET 操作で行えますが、変更は PUT 操作と DELETE 操作でしか行えないリソースです。 **Store**: PUT 操作でエントリを作成および更新できるようにします。 **Transaction**: トランザクション操作をサポートするリソースです。
サービス インターフェイス	**Façade**: 一連の操作に統一されたインターフェイスを実装します。これにより、簡略化されたインターフェイスが提供され、システム間の結合が緩和されます。 **Remote Façade**: 粒度の細かい操作に粒度の粗いインターフェイスを提供して、リモート サブシステムの一連の操作または処理への大まかな統一されたインターフェイスを作成します。これにより、サブシステムが使いやすくなり、ネットワーク経由の呼び出しを最小限に抑えられます。 **Service Interface**: 他のシステムがサービスとの通信に使用できる、プログラム用の インターフェイスです。
SOAP	**Data Contract**: サービス要求で渡されるデータ構造を定義するスキーマです。

	Fault Contracts: サービス要求から返される可能性のあるエラーやフォールトを定義するスキーマです。 **Service Contract**: サービスが実行できる操作を定義するスキーマです。

Duplex パターンと Request Response パターンの詳細については、「サービス コントラクトの設計」(http://msdn.microsoft.com/ja-jp/library/ms733070.aspx) を参照してください。

Request-Reply パターンの詳細については、「Request-Reply」(http://www.eaipatterns.com/RequestReply.html、英語) を参照してください。

Atomic Transactions パターンと Cross-service Transactions パターンの詳細については、「Cross-Service Transactions with WS-AtomicTransaction」(http://www.soaspecs.com/ws-atomictransaction.php、英語) を参照してください。

Command Message パターン、Document Message パターン、Event Message パターン、Durable Subscriber パターン、Idempotent Receiver パターン、Polling Consumer パターン、および Transactional Client パターンの詳細については、「SOA のメッセージングパターン (パート 1)」(http://msdn.microsoft.com/ja-jp/library/cc947720.aspx) を参照してください。

Data Confidentiality パターンと Data Origin Authentication パターンの詳細については、「Chapter 2: Message Protection Patterns」(http://msdn.microsoft.com/en-us/library/aa480573.aspx、英語) を参照してください。

Replay Detection パターン、Exception Shielding パターン、および Validation パターンの詳細については、「Chapter 5: Service Boundary Protection Patterns」(http://msdn.microsoft.com/en-us/library/aa480597.aspx、英語) を参照してください。

Claim Check パターン、Content Enricher パターン、Content Filter パターン、および Envelope Wrapper パターンの詳細については、「SOA のメッセージングパターン (パート 2)」(http://msdn.microsoft.com/ja-jp/library/cc947734.aspx) を参照してください。

Remote Façade パターンの詳細については、「P of EAA: Remote Facade」(http://martinfowler.com/eaaCatalog/remoteFacade.html、英語) を参照してください。

REST パターン (Behavior パターン、Container パターン、Entity パターンなど) の詳細については、「REST Patterns」(http://wiki.developer.mindtouch.com/REST/REST_Patterns、英語) を参照してください。

関連情報

Web リソースに簡単にアクセスするには、http://www.microsoft.com/architectureguide (英語) でオンライン版の参考文献を参照してください。

- 分散システムの詳細については、「分散システムのパターン」(http://msdn.microsoft.com/ja-jp/library/ms998483.aspx) を参照してください。
- Enterprise Service Bus を使用するシナリオの詳細については、「Microsoft BizTalk ESB Toolkit」(http://msdn.microsoft.com/en-us/library/dd897973.aspx、英語) を参照してください。
- 統合パターンの詳細については、「Integration Patterns」(http://msdn.microsoft.com/en-us/library/ms978729.aspx、英語) を参照してください。
- サービスのパターンの詳細については、「サービスのパターン」(http://msdn.microsoft.com/ja-jp/library/ms998508.aspx) を参照してください。
- Web Services Security パターンの詳細については、「Web Service Security」(http://msdn.microsoft.com/en-us/library/aa480545.aspx、英語) を参照してください。

26

ホストされているクラウド サービスの設計

概要

この章では、リモートでホストされているサービスやアプリケーションを構築して使用するための、新しい成長中のテクノロジや手法を紹介します。このようなサービスやアプリケーションは、インターネット経由でアクセスして、クラウドと呼ばれる場所で実行することから、一般に "クラウド コンピューティング" と呼ばれます。通常、クラウド ソリューションのホスティング プロバイダーホストやベンダーは、さまざまなレベルで構成してカスタマイズできる構築済みのサービス アプリケーションを提供します。また、独自のアプリケーションを社内で作成して、社内のシステムや社外のホスティング プロバイダーのクラウドでホストすることもできます。

独立系ソフトウェア ベンダー (ISV) や企業にとって、社外でホストされているサービスやアプリケーションを構築して使用するという考え方は、コストを削減し、効率を最大限に高め、機能を拡張する手段としてより魅力的なものになっています。この章では、クラウドでホストされているサービスやアプリケーションの性質と使用方法を理解するのに役立つ情報を提供します。また、メリット、設計に関する一般的な問題、およびこのようなアプリケーションを構築して使用する際に発生する制約やテクノロジに関する考慮事項についても説明します。

クラウド コンピューティング

多くの点において、クラウド コンピューティングは、スケーラビリティの高い分散ソリューションを構築して使用するためのコンピューティング インフラストラクチャとアプリケーション モデルを融合して発展したものです。このようなアプリケーションの構築手法が進化してきたのと同じように、アプリケーションを実行するインフラストラクチャの機能も進化してきました。この相乗的な発展により、インフラストラクチャでホストされているアプリケー

ションとはほぼ無関係に、インフラストラクチャの準備と保守を行えます。そのため、アプリケーションでは、アプリケーション固有のビジネス機能に重点を置きながら、サポート インフラストラクチャのサービスや機能を利用できます。

これまで、多くの企業では、スケーラビリティの高いアプリケーション モデルとサポート インフラストラクチャの相乗効果を、社内のデータ センターで享受できました。しかし、クラウド コンピューティングに関する魅力のほとんどは、社外のアウトソーシングしたアプリケーション ホスティング インフラストラクチャを活用できることから生まれています。インフラストラクチャ プロバイダーが、ハードウェア、ネットワーク、電力、冷却、および (アプリケーションの管理容易性、信頼性、およびスケーラビリティをサポートする) 運用環境に集中するので、企業はアプリケーションのビジネス機能に集中できます。このため、資本支出と運用コストを削減し、容量、スケーラビリティ、および可用性が向上するという形で、さまざまなメリットを得ることができます。

このようなメリットを活用するには、通常、クラウドでホストされるアプリケーション アーキテクチャは、特定のアプリケーション モデルに従って設計する必要があります。このように設計すると、クラウドをホストするプロバイダーでは、アプリケーションの管理容易性、信頼性、またはスケーラビリティに関する運用環境のサポートを汎用的なものにして最適化できます。

アプリケーション モデルの要件は、クラウドをホストするプロバイダーによって異なります。一部のプロバイダーでは、仮想マシンの手法を採用しています。この手法では、オペレーティング システムのイメージや依存関係のあるランタイム フレームワークと共に、アプリケーションを開発してパッケージ化します。データ アクセスとストレージ (この章の後半参照) および処理と通信について、高度な抽象化を実現するアプリケーション モデルを利用しているプロバイダーもあります。また、エンタープライズ リソース プランニング (ERP)、顧客関係管理 (CRM) など、特定の機能に重点を置いた、柔軟に構成できるアプリケーションに基づいて高度なアプリケーション モデルを提供しているプロバイダーもあります。これらの手法には、それぞれ明確な長所と短所があります。

さらに、社外でホストされているアプリケーションは、独立したアプリケーションであったり、専用の UI を使用してアプリケーションを操作するユーザー向けに設計されている場合があります。このようなアプリケーションには、サービスに対応し、UI を提供すると同時に (多くの場合は REST、SOAP などの標準で公開される) API 経由で機能を公開するものもあり、他の (社内でも社外でもホスト可能な) アプリケーションと統合できます。また、社外でホストされているサービスには、他のアプリケーションと統合する機能を提供するように設計され、まったく UI が表示されないものもあります。

一般に、クラウド ベースのサービスは、ストレージ サービスとコンピューティング サービス、ビジネス サービス、小売サービスと卸売サービスなどのカテゴリに分類されます。このようなリモート サービスの一般的な例としては、次のようなサービスがあります。

- **ビジネス サービス**: 株式情報、請求支払いシステム、データ交換機能、商用サービス、ビジネス情報のポータルなど。
- **小売サービスと卸売サービス**: カタログ、在庫照会/注文システム、気象情報や交通情報、地図サービス、通信販売ポータルなど。
- **ストレージ サービスとコンピューティング サービス**: データの保存と処理、データのバックアップ、ソース管理システム、技術的な処理や科学的な処理のサービスなど。

これらのリモート サービスは、企業のデータ センターやユーザーのコンピューターなど、社内で実行されるソフトウェアから使用できます (ユーザーのコンピューターはデスクトップ コンピューターの場合もあれば、その他のインターネット対応デバイスの場合もあります)。この場合、通常は、Software plus Services (S+S: ソフトウェア プラス サービス) と呼ばれる、テクノロジと技法を組み合わせたものを使用します。S+S は、ホストされているサービスとローカルで実行されるソフトウェアを組み合わせたアプリケーション開発手法です。リモート サービスとローカルで実行されるソフトウェアを組み合わせて、機能豊富でシームレスに統合されたインターフェイスとユーザー エクスペリエンスを提供すると、従来の社内の閉鎖的なアプリケーションよりも包括的で効率的なソリューションを実現できます。S+S は、サービス指向アーキテクチャ (SOA)、Software as a Service (SaaS: サービスとしてのソフトウェア)、Platform as a Service (PaaS: サービスとしてのプラットフォーム)、Web 2.0 におけるコミュニティ指向のアーキテクチャ手法など、いくつかの他のテクノロジが発展したものです。

クラウド コンピューティングは新しい分野です。この章では、クラウド コンピューティングのメリットをいくつか簡単に紹介し、クラウドでホストされているアプリケーションやサービスを構築したり使用したりする際に考慮が必要な、アーキテクチャに関する大まかな考慮事項について概説します。近い将来、フレームワーク、ツール、およびホスティング環境がさらに向上して、このような課題の対処に役立つことが予想されます。

ホストされているクラウド サービスに関する一般的な用語

この章やホストされているクラウド サービスのシナリオで一般的に使用される用語には、次のようなものがあります。

- **ビルディング ブロック サービス**: 他のアプリケーションやサービスで使用したり、他のアプリケーションやサービスと統合したりするように設計されたサービス。たとえば、ストレージ サービス、または Windows Azure Platform の AppFabric アクセス コントロールなどのホストされているセキュリティ トークン サービス (STS) は、ビルディング ブロック サービスです。

- **クラウドをホストする環境**: ホスティング アプリケーションの主要なランタイムを提供する環境。オプションで、ビルディング ブロック サービス、ビジネス サービス、ソーシャル ネットワーキング サービス、および (計測、請求、管理などを行うための) ホスティング サービスが提供されます。
- **社内で構築したアプリケーション**: 社内で作成され、通常は、必要な作業、シナリオ、または処理を対象とするアプリケーション。多くの場合、サード パーティ製のアプリケーションでは対応できないニーズに対処します。
- **ホストされているアプリケーション**: サービスとしてホストされている (パッケージまたは社内で構築した) アプリケーション。社内独自のシステムでホストすることも、社外でパートナーやホスティング プロバイダーがホストすることもできます。
- **パッケージ アプリケーション**: サード パーティやベンダーが作成したアプリケーション。構成やプラグインに基づく制限されたカスタマイズ機能しか提供されないことがあります。
- **Platform as a Service (PaaS: サービスとしてのプラットフォーム)**: 主要なホスティング オペレーティング システムと、オプションでプラグイン形式のビルディング ブロック サービスが付属することがあります。PaaS を利用すると、リモートのクラウドをホストする環境で、独自のアプリケーションやベンダーが提供するアプリケーションを実行できます。
- **Software as a Service (SaaS: サービスとしてのソフトウェア)**: 包括的なビジネス タスクを実行したりビジネス サービスを提供するアプリケーション。SaaS を利用すると、ビジネス タスクやビジネス サービスを、構成と UI 以外に内部アプリケーションの要件がないサービスとして使用できます。

クラウド アプリケーションのメリット

ISV、または (サービスの構築、ホスト、および提供を行う) サービスを提供する企業やホスティング プロバイダーでは、クラウドでホストされているアプリケーションやサービスが非常に役立つことがあります。また、ホストされているクラウド ベースのソリューションを使用することが多い大規模な企業でも、このようなアプリケーションやサービスが役立ちます。

ISV とホスティング プロバイダーのメリット

ISV やクラウド ベースのソリューションを構築して提供するサービス ホスティング プロバイダーには、主に次のようなメリットがあります。

- **アーキテクチャの柔軟性**: ベンダーは、顧客が必要とするサービスのホスティングなど、さまざまな配置オプションを顧客に提供できます。また、さまざまな組み込みの機能を用意して、ユーザーが使用する機能を選択できるようにしたり、アプリケーションに実装する機能をユーザー自身が選択できるようにすることも可能です。このため、サービスを開発するエンド ユーザーが感じるアーキテクチャのデメリットを軽減できます。
- **機能豊富なユーザー エクスペリエンス**: ISV やホスティング プロバイダーは、既存の専用サービス (Virtual Earth など) を活用して、より機能豊富なエクスペリエンスを顧客に提供できます。ホスティング プロバイダーは、自社のサービスと他社が所有する他のクラウド サービスを組み合わせて付加価値を提供し、エンド ユーザーが簡単にサービスを統合できるようにします。
- **ユビキタスな アクセス**: クラウド上のサービスでは、ユーザーのデータや状態を保持し、ユーザーが任意の場所からサービスに再接続するとデータと状態を再度同期します。オフライン シナリオと不定期に接続するシナリオの両方がサポートされるので、この機能は、安定した接続や帯域幅を確保できないモバイル デバイスで特に役立ちます。

ISV やサービス ホスティング プロバイダーは、この市場に参入して、利益創出機会の活用を検討することをお勧めします。たとえば、次のような場合があります。

- ベンダーは、現在他社から入手できないか入手するのが困難な製品を提供することで未開発の市場機会を活用する場合や、クラウドを使用して機能が制限されたバージョンの製品を提供し、高機能な主力製品を保護する場合があります。
- 新興企業は、クラウドでホストされている手法を使用して、初期の資本支出を最小限に抑える場合や、融通性 (初期コストを抑え、必要に応じて規模を拡大できる性能) など、クラウドの特性を利用できます。
- ベンダーやユーザーは、既存の補助サービスを利用して、より収益性の高いアプリケーションを作成できます。たとえば、クラウドでホストされている支払い経理システムを利用できます。また、ユーザーは、IT 機器やネットワーク機能に大幅な投資を行わなくても仮想ストアを構築できます。

企業のサービス コンシューマーのメリット

クラウド ベースのソリューションを使用する企業には、主に次のようなメリットがあります。

- **アーキテクチャの柔軟性**: 社内開発者は、クラウドのサービスをローカル アプリケーションのコードや企業独自のサービスと組み合わせて、完全なソリューションを作成できます。IT 部門では、社内で実装するアプリケーションの機能を選択して、他の必要なサービスを購入できます。
- **コストと時間の節約**: IT 部門では、各作業に最適なクラウド ベースのサービスを選択して組み合わせることで、短い開発期間と少ないコストで十分な機能を備えたアプリケーションを公開できます。また、社内の IT インフラストラクチャに関する要件が少なくなるので、管理、セキュリティ、および保守にかかるコストを削減できます。
- **スケールメリット**: 企業は、業界標準の機能のスケールメリットを活用して、自社の中核となる機能に重点を置くことができます。ホストされているアプリケーションで達成できるスケールメリットが向上する要因には、社内のインフラストラクチャにかかるコストの削減、ハードウェア使用率の向上による運用コストの削減など、さまざまなものがあります。ただし、スケールメリットの向上は、社内アプリケーションから完全にホストされているアプリケーションに移行する際に生じる制御機能の低下との間でバランスを取る必要があります。
- **オフライン機能**: クラウドは、移動ユーザーのハブとして使用できます。ユーザーのデータや状態をクラウドに格納して、ユーザーが再接続したときに再度同期できます。ユーザーは、ネットワークをそれほど詳細に構成しなくても、デスクトップ クライアントとモバイル クライアントをシームレスに使用できます。

設計に関する問題

一般的な問題のいくつかは、ISV と企業顧客の両方にかかわる懸念事項です。このような問題は、ホストされているクラウド ベースのシナリオのさまざまな側面に関連していますが、具体的な領域に分類できます。ホストされているクラウド ベースのサービスに関する方針を決定する際には、次の事項を考慮します。

- データの分離と共有
- データのセキュリティ
- データ ストレージと拡張性
- ID 管理
- マルチテナント
- 社内と社外、構築と購入
- パフォーマンス

- サービスの構成
- サービスの統合
- サービスの管理

データの分離と共有

ホスティング プロバイダーは、データベースとデータベース スキーマの分離と共有を実装できます。分離と共有には、次の 3 つの基本モデルがあります。

- **個別のデータベース**: 各テナントには、独自のデータ スキーマを含む個別のデータベースがあります。このモデルには、実装しやすいというメリットがある一方で、データベース サーバーあたりのテナント数が比較的少ないために効率が大きく低下する場合や、サービスを提供するインフラストラクチャのコストが急激に増加する場合があります。このモデルが最も有効なのは、テナントにデータの分離やセキュリティに関する要件があり、その要件に関する料金をテナントに請求できる場合です。
- **共有データベース、個別のスキーマ**: すべてのテナントで同じデータベースを使用しますが、各テナントで使用できるのは、定義済みフィールドの個別のセットだけです。この手法も実装しやすく、データベース サーバーあたりのテナント数を最大まで増やし、データベースの効率が向上します。ただし、通常は、データベースのテーブルが分散して配置されます。この手法が最も有効なのは、さまざまなテナントのデータを同じテーブルに格納すること (混在) をセキュリティと分離の観点から許容でき、将来必要になる定義済みのカスタム フィールドを予想できる場合です。
- **共有データベース、共有スキーマ**: すべてのテナントでは、同じデータベースを使用し、特殊な技法を使用してデータ拡張を格納します。この手法には、カスタム フィールドをほぼ無制限に提供できるというメリットがあります。ただし、インデックス作成、検索、クエリ、および更新の処理は、より複雑になります。この手法が最も有効なのは、さまざまなテナントのデータを同じテーブルに格納すること (混在) を、セキュリティと分離の観点から許容できても、将来必要になる定義済みのカスタム フィールドの範囲を予想するのが難しい場合です。

次の表に、上記の分離と共有に関する 3 つのモデルのメリットとデメリットを示します。表の上部にあるほど、コストが高くなり、開発と運用の労力が少なくなります。表の下部にあるほど、コストが低くなり、開発と運用の労力が多くなります。

	メリット	デメリット

個別のデータベース	実装しやすい。 社内環境からホストされている環境にアプリケーションを簡単に移行できる。 ほとんどの既存のツールがデータベース レベルで動作するので、バックアップ、復元、および監視が簡単。 データの分離レベルが高い。	テナントのデータベース間でドメイン モデルの共通テーブルが重複する。 ハードウェアのコストが高い。
共有データベース、 個別のスキーマ	メモリの使用量が少ない。 サーバーあたりのテナント数が多い。 テナント間で共通テーブルが共有される。 テーブル名を傍受するデータ アクセス コンポーネントが必要。 データにアクセスするにはテナント レベルの承認が必要。	分離レベルが低い。 カスタム ソリューションが必要なのでバックアップと復元が困難。 テナントの活動を監視することが困難。
共有データベース、 共有スキーマ	メモリ消費量が最も少ない (データベース オブジェクトが少ない)。 サーバーあたりのテナント数が最も多い。	分離レベルが最も低く、高い分離レベルを確保するには追加の開発作業が必要。 テナントのデータが他のテナントと共有される。 カスタム ソリューションが必要なのでバックアップと復元が困難。 テナントの活動を監視することが困難。

共有データベースの手法が適しているアプリケーションは、より複雑で、開発に必要なコストと作業が増加する場合があります。しかし、通常はサーバーあたりのサポートされるテナント数が増加するので、運用コストが減少することがあります。共有スキーマの手法が適しているアプリケーションは単純になります。一般に、長期的な運用コストは減少しますが、共有データベースの手法ほど大きな減少はありません。

アプリケーションをすばやく作成する必要がある場合は、専用のデータベースを使用するように各テナントを構成して、"個別のデータベース" 手法を検討します。この手法を使用するうえで特別な設計は必要ありません。また、個々のテナントのデータについて非常に高いセキュリティ要件がある場合、膨大な量のデータを格納する場合、また

は同時接続エンド ユーザー数が非常に多い場合にも、この手法の使用を検討します。"個別のデータベース" 手法は、アプリケーションを社内環境からホストされている環境に、またはホストされている環境から社内環境に、簡単に移行できるようにする必要がある場合にも適しており、この手法を使用すると、必要に応じてより簡単にスケールアウトできます。

分離レベルが高いと、付加価値のあるサービスをテナント単位で提供する場合にも役立ちます。この場合は、"個別のデータベース" 手法または "共有データベース、個別のスキーマ" 手法の使用を検討する必要があります。ただし、テナント数が膨大で、テナントあたりのデータ量が比較的少ない場合は、"共有データベース、個別のスキーマ"、"共有データベース、共有スキーマ" など、分離レベルの低い手法の使用を検討します。

データのセキュリティ

クラウドでホストされているアプリケーションでは、強力なセキュリティを実装する必要があります。実装の際には、相互に補完する複数の防御レベルを使用して、さまざまな方法とさまざまな状況下で、内外の脅威からデータを保護する必要があります。セキュリティの方針を計画する際には、次のガイドラインを考慮します。

- **フィルター**: テナントとデータ ソースの間で、ふるいの役目を果たす中間レイヤーを使用して、テナントからは、データベース内にそのテナントのデータだけが存在しているように見えるようにします。これは、すべてのテナントに共有データベース インスタンスを使用している場合は特に重要です。
- **アクセス許可**: アクセス制御リスト (ACL) を使用して、アプリケーションのデータにアクセスできるユーザーと、そのユーザーがデータに対して実行できる操作を決定します。
- **暗号化**: すべてのテナントの重要なデータを隠ぺいし、第三者が不当にデータにアクセスしてもデータを読み取れないようにします。

データのセキュリティに関するパターン

採用したマルチテナント モデルに応じて、セキュリティに関する次のパターンの使用を検討します。

- **Trusted Database Connections** (3 つのマルチテナント モデルすべてに適用): ユーザーの ID とは関係なく、アプリケーションでは必ず独自のアプリケーション プロセス ID を使用してデータベースに接続し、サーバーでは、このアプリケーションで読み取りまたは操作可能なデータベース オブジェクトへのアクセスを許可します。追加のセキュリティをアプリケーション自体に実装して、個々のユーザーが、そのユーザーに公開できないデータベース オブジェクトにアクセスしないようにする必要があります。アプリケーションを使用する各テナント (組織) には、テナント アカウントに関連付けられた資格情報が複数あり、これらの資格情報を使用したアプリケーションへのアクセスをエンド ユーザ

ーに許可する必要があります。エンド ユーザーは、テナント アカウントに関連付けられた各自の資格情報を使用してアプリケーションにアクセスしますが、アプリケーションからデータベースへは、そのアプリケーションに関連付けられた特定の資格情報を使用してアクセスします。つまり、アプリケーションごとに必要なデータベース アクセス アカウントは 1 つ (テナントごとに 1 つ) です。また、STS を使用すると、ユーザーに関係ない、テナントの暗号化されたログイン資格情報を取得し、アプリケーションでセキュリティ コードを使用して、各ユーザーがアクセスできるデータを制御できます。

- **Secure Database Tables** ("個別のデータベース" モデルと "共有データベース、個別のスキーマ" モデルに適用): テーブルや他のデータベース オブジェクトへのアクセスをテナントのユーザー アカウントに許可します。"個別のデータベース" モデルでは、データベース単位で、そのデータベースに関連付けられたテナントにアクセスを制限します。"共有データベース、個別のスキーマ" モデルでは、テーブル単位で、その特定のテーブルに関連付けられたテナントにアクセスを制限します。
- **Tenant Data Encryption** (3 つのマルチテナント モデルすべてに適用): 対称暗号化を使用してデータのセキュリティを確保し、非対称暗号化 (公開キーと秘密キーの組み合わせ) を使用してテナントの秘密キーのセキュリティを確保します。偽装を使用して、テナントのセキュリティ コンテキストでデータベースにアクセスし、テナントの秘密キーを使用して、データベースのデータを復号化して使用できるようにします。デメリットは、暗号化した列にインデックスを作成できないことです。つまり、データのセキュリティとパフォーマンスの間にはトレードオフが存在します。機密データを含むインデックス フィールドは使用しないようにします。
- **Tenant Data Filter** ("共有データベース、共有スキーマ" モデルに適用): SQL ビューを使用して、テナント ID、ユーザー ID、またはテナント アカウントのセキュリティ ID に基づいてテーブル データのサブセットを選択します。テナントには、そのテナントのビューへのアクセスだけを許可し、ビューが基づいているテーブルへのアクセスは許可しません。このようにすると、共有テーブルに含まれる他のテナントや他のユーザーが所有する行をユーザーが参照またはアクセスできないようにすることが可能です。

データ ストレージと拡張性

ホストされているデータはさまざまな方法で格納できます。ホストされているアプリケーションには、ホストされているリレーショナル データベース管理システム (RDBMS) およびリレーショナルではないクラウド ベースのストレージという 2 種類の新しいデータ ストレージの実装方法があります。リレーショナル データベース管理システムでは、構造化データのストレージが提供されるので、多数の I/O 処理を実行するトランザクション システムやアプリ

ケーションに適しています。通常、RDBMS は待ち時間が短く、高度なクエリ機能も提供されます。一方、クラウドベースのストレージは、クラウドに配置されたあらゆる種類のデータ ストレージです。たとえば、データベースと同様の機能を提供するサービス、構造化されていないデータのサービス (デジタル メディアのファイル ストレージなど)、データ同期サービス、ネットワーク接続ストレージ (NAS) サービスなどがあります。多くの場合、データ サービスでは、従量制課金モデル、つまりこの場合は (保存するデータと転送するデータの両方を対象とする) GB 単位の課金モデルを採用しています。

クラウド ストレージには、場所と時間に関係なく大量のデータの格納や取得が可能なことなど、多数のメリットがあります。データ ストレージ サービスは、高速で、コストがかからず、ほとんど無制限に拡張できますが、最高のサービスでも障害が発生する可能性はあるので、信頼性が問題になることがあります。ストレージ サービスと通信するたびにネットワークを経由する必要があるので、待ち時間の長さの影響を受けやすいアプリケーションに悪影響が及ぶこともあります。最後に、クラウド ベースのストレージ システムでは、トランザクションのサポートが問題になることがあります。通常、このようなシステムでは分割と可用性に重点を置くので、必ずしも一貫性を確保できるとは限りません。

Windows Azure Platform のストレージ (現時点では初期のプレビュー リリース) は、次のように、ストレージに関するさまざまなニーズに対応した多数のサービスで構成され、これらのサービスには REST API を使用してアクセスできます。

- **テーブル ストレージ サービス**: テーブル形式の構造化されたストレージを提供します。ただし、これらのテーブルには定義されたスキーマがなく、代わりに、多数のプロパティを保持するエンティティが含まれています。どのようなプロパティの組み合わせでも、LINQ などの一般的に使用される API を使用できます。テーブル ストレージ サービスは、非常にスケーラビリティの高いテーブルを低コストで提供することに重点を置いています。しかし、リレーショナル データベースではないので、結合、複数のテーブルにわたる外部キーなど、RDBMS で使用できる機能の多くは提供されません。
- **BLOB ストレージ サービス**: ユーザー定義のコンテナーに格納されたバイナリ データのストレージを提供します。このコンテナーは、ストレージ アカウント内の 複数 BLOB のセットを構成します。
- **キュー サービス**: ストレージ アカウントにアクセスできるすべてのクライアントで、キューのセマンティクスを使用して読み取ることが可能なメッセージを格納します。

RDBMS を使用する際に解決する必要がある主な課題は、スキーマの拡張性です。具体的には、テーブルを再コンパイルまたは再構築しなくても、アプリケーションをカスタム フィールドで拡張できる性能です。実行時にスキーマを拡張するパターンは、次の 4 種類です。

- **Fixed Columns**: このパターンでは、拡張可能なエンティティごと (テーブルごと) に、拡張フィールドを一連の名前付き固定列としてモデル化します。固定列の数は、エンティティの性質とその使用パターンによって異なります。Fixed Columns パターンには、名前付き固定列とメタデータ テーブルをカプセル化して抽象化するデータ アクセス レイヤーが必要です。Fixed Columns パターンについては、次の事項を考慮します。
    - データ型が事前に定義されているので、拡張可能な列に基づいたフィルター処理が困難です。たとえば、varchar などの長さが一定でないデータ型をすべての拡張可能な列に使用すると、演算子 (<、>、および =) を使用して数値でフィルター処理する機能が制限されます。解決策としては、共通のデータ型ごとに一定数のフィールドを割り当てたり、ユーザーが列を検索可能に設定できるようにし、個別のテーブルを使用してデータ型固有のフィールドを格納することが挙げられます。
    - Fixed Columns パターンは、拡張性という点において最も高速で比較的スケーラビリティが高い手法の 1 つですが、通常は、文字列ベースではないインデックス付きの列に関するソリューションが必要になります。
    - データベースで null 値を処理する方法によっては、データがあちこちに分散し、領域を無駄に使用することがあります。あるテナントがフィールドを 1 つだけ拡張し、別のテナントが 20 個のフィールドを拡張する場合、メモリ内のデータベースとページが増加します。Microsoft SQL Server 2008 には、この問題の軽減に役立つ SPARSE という名前の列の修飾子が用意されています。
- **Custom Schemas**: このパターンは、マルチテナントの "共有データベース、個別のスキーマ" 手法と組み合わせて使用します。各スキーマはテナントに属し、拡張可能な厳密に型指定されたさまざまな列が含まれています。Custom Schemas パターンについては、次の事項を考慮します。
    - このパターンには、データ アクセス レイヤーのカプセル化と抽象化、およびクエリ プロセッサが必要です。ただし、Microsoft Entity Framework (EF) やオープン ソースの NHibernate フレームワークなど、O/RM フレームワークが実装に役立ちます。詳細については、「関連情報」を参照してください。
    - 各テナントには専用のテーブルがあり、テナントがフィールドを追加または削除するたびにテーブル スキーマが変更されます。クエリは、実際のデータ型に対して機能します (すべての列が文字列型になるとは限りません)。ただし、テナントの共有フィールド間で、かなりのフィールドが重複しているので、データベース スキーマの更新内容をまとめるのが難しくなりま

す (Fixed Columns パターンでは、すべてのテナントを対象とするテーブルが 1 つ以上存在する点が異なります)。
    - プリミティブなデータ型を使用するので、拡張した列に基づいてフィルター処理する場合、この手法は Fixed Columns パターンよりも高速になります。

- **Name Value Extension Table**: このパターンを使用すると、顧客は、カスタム データを個別のテーブルに格納し、メタデータを使用して各テナントのカスタム フィールドのラベルやデータ型を定義して、自由に (列数を制限されずに) データ モデルを拡張できます。Name Value Extension Table パターンについては、次の事項を検討します。
    - インデックス作成、クエリ、レコードの更新などを行うデータベース関数がさらに複雑になります。たとえば、データを取得するために複数回結合する必要があり、フィルター処理やグループ化が困難になります。
    - 管理が必要なデータベースは 1 つだけです。ただし、ユーザー数が増加するとデータベースのサイズも増加する場合は、テナントを分割して個別のデータベースを使用するようにすることで、水平方向に拡張できます。
    - データを取得するために複数回結合する必要があるので、このパターンは他のパターンより低速になります。

- **XML Columns**: このパターンを使用すると、顧客は、拡張データを XML 列に格納して、列数を制限されることなく自由にデータ モデルを拡張できます。XML Columns パターンについては、次の事項を考慮します。
    - このパターンは、拡張性に関する自然な手法のように思われますが、他のパターンに比べて、スケーラビリティ (レコードを追加する性能) とパフォーマンス (クエリの応答時間) が低くなります。
    - データベースで XML 列を使用すると実装は単純になりますが、ISV や開発者には、データベースで XML を操作するスキルが必要になります。
    - XML 列のインデックスを定義することは可能ですが、インデックスを定義すると、より複雑になり、ストレージの容量を追加する必要があります。

ID 管理

すべてのアプリケーションとサービスでは、ユーザー ID を管理する必要があります。これは、多数の顧客にサービスを提供して、その各顧客が独自の ID フレームワークを使用している可能性のある、ホストされているクラウドベースのシナリオでは特に重要です。一般的な管理方法は、ユーザー アカウントを管理する役割をホスティング プロバイダーから各顧客に委任することです。また、最適な方法は、顧客の既存の社内または統合されたディレクトリ サービスを利用して、顧客のローカル サービスと外部でホストされている全サービスの間でシングル サインオン (SSO) を有効にするソリューションです。このようなソリューションを使用すると、独立した個別の ID 管理システムを構築する際の開発作業を軽減できます。顧客は、使い慣れたツールを使用してアプリケーションへのアクセスを構成できます。また、SSO を使用すると、ユーザーは既存の資格情報を使用してアプリケーションやサービスにアクセスできます。

このようなシナリオを実現するには、プラットフォーム間や組織の境界を越えて相互運用する、業界標準に基づいたソリューションを採用する必要があります。通常は、統合された ID サービスに基づくクレーム ベースの ID モデルを検討する必要があります (図 1 参照)。このモデルを使用すると、アプリケーションやサービスを認証メカニズムから分離できます。

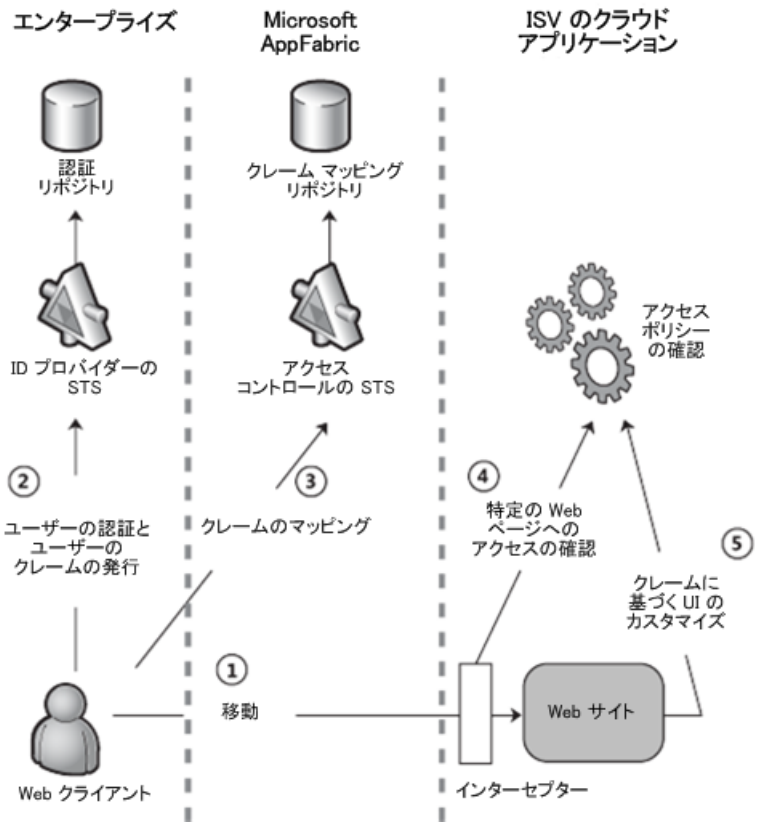


図 45

統合された ID サービスに基づくクレーム ベースの ID モデル

顧客の既存の ID システムでは、ユーザーがアプリケーションに対して要求を行うたびに、各ユーザーに関する一連のクレームを含む暗号化および署名されたセキュリティ トークンを送信します。
顧客の ID システムを信頼するホスティング プロバイダーは、関連するクレームの承認だけに重点を置いたアプリケーションやサービスを設計できます。顧客の ID システムでは STS を実装する必要があり、この STS では、ユーザーを認証し、標準的な形式を使用してセキュリティ トークンを作成および署名し、サービスを公開して WS-Trust、WS-Federation などの業界標準に基づいてこれらのトークンを発行します。Windows Identity Foundation と Active Directory フェデレーション サービス 2.0 では、このような要件を実装するために必要なインフラストラクチャのほとんどが提供されます。クレーム ベースの ID モデルを実装する際には、次の問題を考慮します。

- 適切な ID ストアを使用できる場合は、この ID ストアを使用して、ローカル アプリケーション、ホストされている Web アプリケーション、およびその他のホストされているサービスの間でシングル サインオンを利用できるようにすることを検討します。
- 既存の ID ストアを使用できない小規模なアプリケーションやコンシューマー向けアプリケーションの場合は、Windows Live と統合するサービス (AppFabric アクセス コントロールなど)、またはサードパーティ製のオンライン ソリューションの使用を検討します。
- ホスティング プロバイダーの要件を満たすために、ローカルの STS から生成されたクレームを変換しなければならない場合があります。また、さまざま顧客の STS メカニズムに対応した変換をホスティング プロバイダー側で実装しなければならない場合もあります。AppFabric アクセス コントロールを使用した変換レイヤーの提供を検討するか、Windows Identity Foundation を使用して独自の変換レイヤーを実装します。Windows Identity Foundation の詳細については、http://msdn.microsoft.com/en-us/security/aa570351.aspx (英語) を参照してください。
- さまざまなベンダーが提供している標準に準拠した製品の実装方法は、わずかに異なっていることがあります。これらの製品が複雑な標準に厳密に準拠していない場合や、提供される実装がわずかに異なる場合は、互換性と相互運用性に関して問題が発生することがあります。
- 独自の STS を設計する場合は、攻撃から STS を保護します。STS にはすべての認証情報が含まれているので、脆弱性があるとすべてのアプリケーションが悪用される危険にさらされます。また、STS の実装は、堅牢で信頼できて、予測可能なすべての要求に対応できるようにします。STS で障害が発生すると、ユーザーはその STS に依存するすべてのアプリケーションにアクセスできなくなります。

マルチテナント

個々のテナントは、ホスティング プロバイダーのハードウェアとインフラストラクチャを共用し、データベースとデータベース システムを共有します (各テナントは、少なくとも 1 人のユーザーがいる組織です)。サービス サプライヤーは、ホストされているサービスに適した容量とパフォーマンスを備えたプラットフォームを提供する必要があります。また、コスト構造を制御する方法や、構成を通じたカスタマイズを提供する方法についても検討する必要があります。ユーザーによる構成が可能な効率的なマルチテナント アーキテクチャへの移行には、一般的に 4 つの段階があります。次のセクションでは、これらの段階について説明します。

- **カスタム**: その顧客だけに割り当てられたソフトウェアの個別のコピーを各顧客が実行します。複数の顧客をサポートするには、顧客にソフトウェアのコピーを個別に提供する必要があります (これが唯一の方法です)。また、構成を通じたカスタマイズを可能にする機能がほとんど用意されていないので、各コピーには、カスタム拡張コード、カスタム プロセス、カスタム データ拡張という形で特定の顧客のカスタム設定が含まれています。ソフトウェアは厳密にはサービスとして提供されますが (顧客の社内では実行されません)、各顧客はソフトウェアの異なるインスタンスを実行するので、スケールメリットを達成できません。また、ビジネス モデルを検証する出発点として役立つ場合もありますが、顧客の規模が拡大した場合はこのモデルの使用を避ける必要があります。このモデルを使用して数千もの顧客を管理することは現実的ではありません。
- **構成可能**: 構成を使用し、カスタム コードを使用しないようにして、それぞれの顧客に合わせてソフトウェアをカスタマイズできます。すべてのテナントで同じコードを実行しますが、カスタム モデルと同様にアーキテクチャはマルチテナントではなく、顧客は、コピーの内容が同じであっても、その顧客専用のコードのコピーを実行します。分離は、仮想的に行う (同一サーバーで複数の仮想マシンを実行する) 場合も、物理的に行う (個別のコンピューターで実行する) 場合もあります。このモデルは最初に説明したカスタム モデルよりかなり優れており、このアーキテクチャでは構成を通じたカスタマイズが可能ですが、処理能力はインスタンス間で共有されません。このため、プロバイダーではスケールメリットを達成できません。
- **マルチテナント**: UI は、ビジネス ルールやデータ モデルと同様に、テナントごとにカスタマイズできます。テナントごとのカスタマイズは、セルフ サービス ツールを使用して、構成のみを通じて行います。このため、サービス プロバイダーがサービスを構成する必要はありません。このレベルは、スケールアウトする機能を除き、ほぼ完全な SaaS と言えます。また、データを分割すると、スケールアップが拡張できる唯一の方法になります。

- **拡張可能**: このアーキテクチャでは、マルチテナントと構成に加えて、アプリケーションをスケールアウトする機能をサポートします。ソフトウェアの新しいインスタンスをインスタンス プールに透過的に追加して、増加する負荷を動的にサポートできます。設計では、適切なデータ分割、ステートレスなコンポーネント設計、および共有メタデータのアクセスが考慮されています。このレベルでは、(ラウンド ロビンまたはルール ベースのメカニズムを使用して実装される) テナント負荷分散装置が導入され、CPU やストレージなどのホスティング プロバイダーのリソースの使用率を最大限に高めることができます。つまり、使用できるインフラストラクチャ全体にすべての負荷が分散されます。また、インスタンスあたりのデータ負荷を均等に分散するために、データが定期的に再構成されます。アーキテクチャは、スケーラビリティが高く、マルチテナントで、構成を通じてカスタマイズできます。

社内と社外、構築と購入

クラウドでホストされているアプリケーションを使用すると、ISV やホスティング プロバイダーはスケールメリットを達成できます。また、競争市場では、このような削減したコストを企業顧客に還元できます。ただし、企業は、社外のホストされているシナリオに移行すると、アプリケーション、データ、およびサービス レベルの制御機能がある程度低下することを受け入れる必要があります。企業は、このようなサービスに移行する際のトレードオフを考慮する必要があり、独自のアプリケーションを構築するか、サード パーティ製のアプリケーションを購入するかどうかを決定する必要もあります。次の表に、ホストされているアプリケーションを構築する場合と購入する場合の違いを示します。

	社内	**ホスティング プロバイダー**	**クラウド サービス**
構築	社内で開発し、社内のデータ センターで実行するアプリケーション	社内で開発し、ホスティング プロバイダーで実行するアプリケーション	ベンダーでホストされる開発環境とランタイム環境
購入	既製品として購入し、社内のデータ センターで実行するパッケージ アプリケーション	既製品として購入し、ホスティング プロバイダーで実行するパッケージ アプリケーション	アプリケーションのホストも行うベンダーから購入し、ベンダーホストされるアプリケーション

スケールメリットの向上は、社内アプリケーションから社外でホストされているアプリケーションに移行する際に発生する制御機能の低下との間でバランスを取る必要があります。クラウド ベースのソリューションに移行するかど

うか、採用するアプリケーション開発手法の種類、およびアプリケーションをホストする場所を決める際には、次のガイドラインを考慮します。

- アプリケーションとデータを完全に制御できる必要がある場合、ホストされているサービスを使用できないセキュリティ要件がある場合、またはホストされているサービスの使用が法律や国内の政策で禁じられている場合は、社内でホストすることを検討します。社内のインフラストラクチャでアプリケーションをホストする場合は、次のオプションから選択します。
    - 適切な構築済みのアプリケーションを入手できない場合や、今後もアプリケーションの機能を完全に制御する必要がある場合は、社内アプリケーションを開発します。
    - コスト効率が高くすべての要件を満たすアプリケーションを入手できる場合は、構築済みのパッケージ アプリケーションを選択します。
- 運用の最適化を検討しながらソフトウェアを引き続き制御できるようにする必要がある場合は、ホスティング プロバイダーでホストすることを検討します。たとえば、大幅にカスタマイズしたエンタープライズ リソース プランニング (ERP) パッケージをホストに配置して、電力、ハードウェア、ネットワーク、およびオペレーティング システムの管理をホストにオフロードできます。通常、ホスティング プロバイダーでは、仮想プライベート ネットワーク (VPN) の設定や特殊なハードウェアの追加など、企業固有の要件に対応します。外部のホスティング プロバイダーでアプリケーションをホストする場合は、次のオプションから選択します。
    - 要件を満たす場合は、ホスティング プロバイダーが提供している構築済みのパッケージ アプリケーションを選択します。使用するホスティング プロバイダーは、そのプロバイダーが提供している構築済みのパッケージ アプリケーションによって異なる可能性があります。
    - 適切な構築済みのアプリケーションを提供するホストが見つからない場合は、社内で構築したアプリケーションを使用します。この場合、独自のアプリケーションの開発にかかるコストと時間を考慮する必要があります。
- ホストや ISV から SaaS アプリケーションを購入する場合、必要なアプリケーションの仕様を指定できる場合、アプリケーションを構築するためのリソース、時間、または社内のスキルがない場合、または緊急に既存の標準的なアプリケーションやカスタマイズされたアプリケーションが必要な場合は、クラウド サービス (ベンダーでホストする) 手法を検討します。また、社内でアプリケーションを構築する際に、この手法を使用すると、クラウド ベースのビルディング ブロック サービスに固有の特性 (融

通性など) を利用できるというメリットもあります。ベンダーからクラウド アプリケーション サービスを購入する場合は、次のオプションから選択します。

- 長期的または短期的な要件を満たす場合は、ベンダーが作成した構築済みのパッケージ アプリケーションを選択します。これは、SaaS の手法です。
- 適切な構築済みのアプリケーションを入手できない場合は、ベンダーが提供するホスティング プラットフォームを選択して、社内で構築したアプリケーションを実行します。独自のアプリケーションの開発にかかるコストと時間を考慮する必要があります。これは、PaaS の手法です。

パフォーマンス

増加するサービスの数と、各サービスや各テナントで増加する負荷をサポートするように、クラウドでホストされているアプリケーションは拡張できる必要があります。サービスを設計する際には、アプリケーションの拡張に関する次のガイドラインを考慮します。

- サービスやコンポーネントができる限りステートレスになるように設計します。このように設計すると、サービスのメモリ使用量が最小限に抑えられ、サーバーをスケールアウトして負荷分散する機会が多くなります。
- 入出力に関する非同期呼び出しを使用して、アプリケーションが I/O 処理の完了を待機している間に有益な処理を実行できるようにします。
- パフォーマンスを向上できるホスティング プラットフォームの機能を確認します。たとえば、Windows Azure Platform では、キューを使用して要求を管理し、ワーカー プロセスを使用してバックグラウンド処理を実行します。
- スレッド、ネットワーク、およびデータベース接続にリソースのプールを使用します。
- やむを得ない場合にのみロックを使用して、同時実行制御を最大限に高めます。

データ ストレージとアプリケーションを拡張する際には、次のガイドラインを考慮します。

- データ パーティションを拡張する場合は、サブスクライバーのデータを小さなパーティションに分割して、目標のパフォーマンスを達成します。ハッシュ (コンテンツの細分化)、範囲 (データが有効な時間や日付の範囲に基づく分割) などの方式を使用します。

- データベースのサイズが一定の最大サイズに達したときに自動的にデータが再分割される動的な再分割の実装を検討します。
- データ ストレージとアプリケーションを拡張する場合は、標準的なパターンを確認します。また、ホスティング プラットフォームに用意されている固有の技法と実装 (データの分割、負荷分散、フェールオーバー、地理的分散など) を確認します。

サービスの構成

エンタープライズ レベルの組織に所属するユーザーは、さまざまなドキュメント レポジトリ、データ型、情報源、および特定の機能を実行するアプリケーションにアクセスする必要があります。これまでユーザーは、それぞれのストアやアプリケーションを直接操作し、多くの場合は専用の独立したアプリケーションを使用していました。しかし、企業は徐々にシステムの統合を進めており、多くの場合、統合には、適切なダウンストリーム アプリケーションに接続するイントラネット Web ポータルやファサード形式のアプリケーションを使用してきました。
サービスと SOA アプリケーションの登場により、IT 部門は、社内でホストしているか SaaS として購入したアプリケーションやデータをサービスとして公開できるようになりました。現在でも、サービス ポートフォリオでは、従来のローカル アプリケーション、社内でホストされているサービス、およびリモート サービスを組み合わせてポータル経由で公開できるので、ユーザーは実装を意識せず、IT 部門はさまざまなサービスにすばやく簡単に対応できます。一方、S+S と SaaS の設計とテクノロジを使用すると、IT 部門や企業顧客はサービスを完全に統合できます。サービスを統合すると、多対一モデルの目標を達成できます。このモデルでは、ユーザーは、すべてのアプリケーションとサービスを実質的に 1 つのアプリケーションとして公開する構成アーキテクチャを通じて、アプリケーションとサービスを使用できます (図 2 参照)。サービス統合メカニズムでは、ポートフォリオに含まれるアプリケーションのグループをまとめ、任意のサービスやアプリケーションと通信できるリッチ クライアントを通じて公開します。

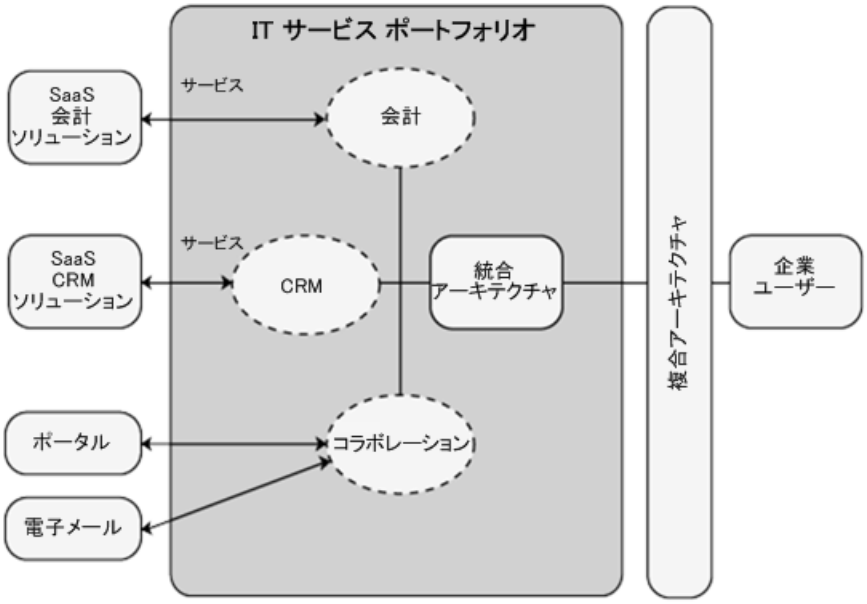


図 2

複数のサービスを組み合わせて 1 つのインターフェイスを構成できるサービス統合メカニズム

通常、クラウドでホストされているサービスを使用する企業では、ホスティング プロバイダーで公開されているサービスをそのまま使用する構成システムを作成し、社内のポータルやポートフォリオを通じてサービスを公開する必要があります。効果的なコンシューマーの構成アーキテクチャを使用すると、さまざまなソースの情報を統合してエンド ユーザーに提供できます。このため、重複したデータ エントリが減少し、ユーザーのコラボレーションが促進され、完了していないタスクやその状態に気付きやすくなります。また、関連性のあるビジネス情報を確認しやすくなり、ユーザーは十分な情報に基づいてビジネスにおける決定を下せるようになります。通常、クラウドでホストされているサービスを使用する統合されたソリューションの構成には、次の 3 つのレイヤーが含まれます。

- **入力ソース レイヤー**: 入力ソースには、クラウドでホストされているサービス、内部アプリケーション、内部データベース、Web サービス、フラット ファイルなどがあります。内部リソースは、IT サービス ポートフォリオを通じて公開できます。
- **構成レイヤー**: 生データを収集し、新しい統一された形式でユーザーに提供するレイヤーです。構成レイヤーの機能は、データをビジネス情報やプロセス インテリジェンスに変換することです。通常、構成レイヤーには、次のコンポーネントが含まれます。
    - アクセス、データ、ワークフロー、および規則を管理するコンポーネント
    - アプリケーション、データベース、Web サービスなどのリソースとネゴシエートしてメッセージを交換するサービス エージェント

    - ユーザーを認証して承認し、Web サービスと通信するための資格情報を管理する ID 管理コンポーネント
    - アプリケーション エンティティ モデルに合わせてデータを変換するデータ収集コンポーネント
- **ユーザー中心のレイヤー**: このレイヤーでは、タスクに重点を置く一元的で統合されたユーザー インターフェイスでユーザーに複合データを表示します。

通常、クラウドでホストされているサービスを複合ユーザー インターフェイスの一部として使用するには、外部サービスと内部サービスを統合するワークフローや段階的なプロセスが必要です。一般的なソリューションには、統合ブローカーがあります。これは、モジュール化されたプラグ可能なパイプラインと、メッセージの送信とルーティングを制御する関連メタデータ サービスで構成されています。統合ブローカーのパイプラインの一般的な操作には、次のような操作があります。

- **セキュリティ**: セキュリティ モジュールでは、データ ソースやデジタル署名を認証し、データを復号化して、ウイルスなどのセキュリティ リスクの有無を検査します。セキュリティ操作は、既存のセキュリティ ポリシーと連携してアクセスを制御できます。
- **検証**: 検証モジュールでは、関連するスキーマとデータを比較し、適合しないデータを拒否します。
- **変換**: 変換モジュールでは、データを適切な形式に変換します。
- **同期ワークフロー**: 同期モジュールでは、ワークフローと規則を使用して、メッセージを適切な転送先に伝達するための論理的な転送先と順序を決定します。また、ワークフロー プロセスのトランザクションを管理して、データの一貫性を保証することもできます。
- **ルーティング**: ルーティング モジュールでは、物理的な転送先を定義するルーティング規則を使用し、データのメッセージを特定の宛先に転送します。また、メッセージに含まれる情報を使用して、内容に基づいて転送先を決定することもあります。

サービスの統合

クラウドでホストされているソリューションを使用すると、従来のソフトウェアで発生する課題の一部を軽減できますが、このようなサービスのコンシューマーでは、新しい別の課題も発生します。ホストされているクラウド サービスやクラウド アプリケーションに移行する際には、次の課題を考慮します。

- **ID 管理**: ユーザーを追加、更新、および削除する社内の処理を、リモート サービスに拡張する必要があります。SaaS や S+S では非常によくあることですが、外部サービスでユーザー ID を使用する場合は、準備と準備解除のプロセスを拡張する必要があります。また、リモート サービス ホストにおける個々のユーザー ID の移行や重複を最小限に抑えるために、(たとえば、統合されたサービスを使用して) 社内のユーザー ID を特定の役割に変換しなければならないことがあります。さらに、企業のユーザー アカウント ポリシー (パスワードの複雑さ、アカウント ロックアウトなど) とリモート サービス サプライヤーのポリシーの間には互換性が必要です。SSO 機能を使用できないと、デメリットや保守にかかるコストが増大したり、運用の効率が低下したりするおそれがあります。
- **データ**: 抽出、変換、読み込み (ETL)、データ統合など、データ操作の要件を分析して、サービスの機能との互換性を確保する必要があります。ホストされているサービスでは、複雑なデータ ストレージのパターンをサポートしていないことがあるので、データ エンティティとアプリケーション アーキテクチャの設計に影響が及ぶことがあります。また、社内でホストしているときの物理的なセキュリティを利用できない代わりとして、データのセキュリティ保護を強化しなければならないことがあります。ただし、アプリケーションでは、ローカルに機密データや個人情報を格納して、機密ではないデータについてのみクラウド サービスを使用することが可能です。また、データをサービス プロバイダーに移行する方法や、必要に応じて別のプロバイダーとの間で移行する方法についても計画する必要があります。
- **運用**: 業界標準のプロトコルを使用している場合でも、社内の統合サービスやクライアント アプリケーションには、サービス サプライヤーが公開するサービスと互換性がないことがあります。また、サービス プロバイダーで適切なレポート情報を生成できるようにし、生成された情報を社内の管理システムやレポート システムと統合する方法を決定する必要もあります。サービス レベルに関しては、サービス プロバイダーにサポートを依頼するときもサービス レベル アグリーメント (SLA) が満たされるように、SLA を改訂しなければならなくなる場合があります。また、企業では、ユーザーの最初の問い合わせ先としてヘルプ デスク機能を実装する準備をし、サービス プロバイダーに問題を報告する手順を規定する必要もあります。
- **セキュリティ**: 企業のプライバシー ポリシーとサービス プロバイダーのポリシーの互換性を確保する必要があり、リモート サービスの機能に規則が含まれていない場合でも、ユーザーが実行できる操作の規則 (トランザクション サイズの制限などのビジネス ルール) を管理する必要があります。このため、サービスの統合インフラストラクチャが複雑になることがあります。
- サービスや相互接続の障害時にデータのセキュリティと整合性を維持するための手続きやポリシーも必要です。

- 認証し、暗号化し、デジタル署名を使用するには、認定されたプロバイダーから証明書を購入する必要があり、場合によっては公開キーのインフラストラクチャ (PKI) を実装する必要もあります。また、統合する際には、ファイアウォール規則を変更する必要がある場合や、アプリケーション データをフィルター処理したり XML スキーマを検証したりする際には、ファイアウォールのハードウェアやソフトウェアを更新する必要がある場合があります。
- **接続**: 一部の種類のクラウド ベースのアプリケーションでは、高品質なブロードバンド インターネット接続を利用して高度な機能を実現します。このようなアプリケーションには、ボイス オーバー IP (VoIP)、Microsoft Office Communications Server など、オンラインのトランザクション処理やリアルタイムのサービスがあります。地域や国によっては、このようなサービスを使用できないことがあります。また、大量のデータを転送する必要があるサービス (バックアップ サービスやファイル配信サービスなど) は、一般的に、インターネット接続を経由するとローカルや社内で実装した場合よりも実行速度が遅くなるので、問題になる可能性があります。ただし、メッセージなどのサービスは接続の帯域幅の影響を受けず、不定期に接続が切断されてもそれほど大きな影響を受けることはありません。
- **サービス レベル アグリーメント**: サプライヤーをより総合的に評価し、サービスの入手と契約について判断するには、スキルと専門知識が必要です。リモート プロバイダーでホストされているサービスを利用するときも SLA が満たされるように、SLA の改訂が必要になる場合もあります。
- **コンプライアンスと法的義務**: サービス サプライヤーのパフォーマンスによって法的または社内の指示が異なったり、サービス プロバイダーが他の国や地域にある場合には、このようなコンプライアンスの指示や法的義務の間で矛盾が発生したりすることがあります。また、サービス サプライヤーからコンプライアンスに関する報告書を取得する際には、コストがかかることがあります。国内の法律や政策によっては、金融アプリケーションなど、一部の種類のアプリケーションをホストされているシナリオで実行できないこともあります。

サービスの管理

クラウド サービスのプロバイダーは、クラウドで実行するサービスをホストして提供する場合、課題 (特にサービスの提供とサポートに関する課題) に直面することがあります。別の ISV でもホストされるアプリケーションを構築する ISV もあれば、社内でアプリケーションを構築してホストする ISV もありますが、ホストされているサービスの開発を検討する際にはいくつかの課題を考慮する必要があります。独自のサービスをホストする ISV に該当する課題もあれば、ホスティング プロバイダーだけに該当する課題もあります。次のセクションでは、このような課題について説明します。

- **サービス レベルの管理**: それぞれの企業ユーザーがホスティング プロバイダーの標準的な SLA を変更するように要求することがあるので、すべての顧客の要求に応じるのが困難になる場合があります。顧客は、クラウドでホストされているソリューションを、可用性とパフォーマンスを向上する手段として選択する可能性があるので、一般的に、顧客の期待値は社内アプリケーションよりも高くなります。通常は、運用上の依存関係 (ネットワークや電力のプロバイダーなど) にかかわる要求や異なる国や地域の顧客からのさまざまな要求が発生するので、このような期待を管理して、期待に応えるのが難しいことがあります。さまざまな企業のサービスをホストする場合、特にこれらの企業の所在地のタイム ゾーンがさまざまで、最大使用量に達する時刻や曜日が企業によって異なるときは、保守とサービスのダウンタイムについても慎重に計画する必要があります。
- **容量と継続性の管理**: サービス プロバイダーと社内チームでは、顧客の容量に関する要件が将来どのように変化するかについての予測が異なるので、使用量が予期しないタイミングで増加して、追加の容量を利用できるようにする必要に迫られることがあります。各顧客の拡張や使用に関するパターンが変化する前兆は、ほとんどまたはまったく見られないので、事前に計画することは困難です。多数の顧客の要件を満たす必要がある場合、顧客の要件に適したサービスの実装と変更はさらに困難になります。容量について短期的な判断を下すと、長い目で見るれば段階的に容量を拡張する計画よりもコストがかかりますが、顧客による拡張の見積もり情報なしに計画することは、さらに困難です。
- **顧客サポート**: ヘルプ デスクのスタッフは、最適なサポートを提供するために、顧客の要件と使用シナリオを認識して考慮しなければならないことがあります。サービスあたりの顧客数が多い場合、そのサービスで障害や問題が発生すると、ヘルプ デスクに問い合わせが殺到して過剰な負荷がかかります。特に、サポートが有料オプションとなっているモデルでは、ヘルプ デスクのスタッフは、ユーザー単位でかかるコストを算出できなければならないこともあります。理想的には、ホストされているクラウド ソリューションでプロアクティブ サポートを提供する必要があります。プロアクティブ サポートでは、プロバイダーがソリューションの正常性を監視して問題を把握し、顧客が報告するより前にその問題の解決に着手します。セルフ サービスのサポート メカニズムを使用して、顧客に効率的な専用の問題追跡システムを提供することもできます。

関連する設計パターン

次の表に示すように、主要なパターンは、データの可用性、データ転送、データ変換、統合と構成、パフォーマンスと信頼性、ユーザー エクスペリエンスなどのカテゴリに分類されます。各カテゴリの設計を行う際には、これらのパターンの使用を検討します。

カテゴリ	関連するパターン
データの可用性	**Polling**: 通常は、定期的に一方のソースからもう一方のソースに変更をクエリします。 **Push**: データ ソースのデータが変更されるたびに、または定期的に、データが変更されたソースからデータ シンクに変更を伝達します。 **Publish/Subscribe**: Polling パターンと Pushing パターンの性質を組み合わせたハイブリッドの手法です。変更が加えられるとデータ ソースで変更通知イベントを発行します (データ シンクでは、このイベントをサブスクライブできます)。
データ転送	**Asynchronous Data Transfer**: 送信側と受信側が相手側の応答を待機せずにデータを交換するメッセージ ベースの手法です。 **Synchronous Data Transfer**: 送信側と受信側がリアルタイムでデータを交換するインターフェイス ベースの手法です。
データ変換	**Shared Database**: 統合するすべてのアプリケーションで、同一のデータベースからデータを直接読み取ります。 **Maintain Data Copies**: 他のアプリケーションでデータを読み取れるように (可能であればデータを更新できるように)、アプリケーションのデータベースのコピーを保持します。 **File Transfer**: 他のアプリケーションでデータをファイルから読み込めるように、アプリケーションのデータベースから抽出したファイルを転送してデータを提供します。
統合と構成	**Broker**: ビジネス コンポーネント自体とは異なるレイヤーに、リモート サービス呼び出しの実装の詳細をカプセル化して、その詳細を隠すことができます。 **Composition**: 複数のサービス、アプリケーション、またはドキュメントを組み合わせて統合されたインターフェイスを作成すると同時に、各データ ソースのセキュリティ、検証、変換、および関連タスクを実行します。 **Portal Integration**: 複数のアプリケーションから取得した情報を統一された UI

	で表示する、ポータル アプリケーションを作成します。ユーザーは、このポータルの情報に基づいて、必要なタスクを実行できます。
パフォーマンスと信頼性	**Server Clustering**: ユーザーやアプリケーションで統合された仮想コンピューティング リソースとして複数のサーバーが認識されるようにアプリケーション インフラストラクチャを設計して、可用性またはスケーラビリティ、あるいはその両方を向上します。 **Load-Balanced Cluster**: 負荷を共有するように構成された複数のサーバーに、サービスまたはアフリケーションをインストールします。負荷分散するように構成された複数のホストでは、同じクライアントから複数の要求が行われた場合でも、同時に異なるクライアント要求に応答します。 **Failover Cluster**: 障害発生時に互いの処理が引き継がれるように構成された複数のサーバーに、アプリケーションまたはサービスをインストールします。クラスター内の各サーバーには、スタンバイ サーバーとして認識されているサーバーが少なくとも 1 台あります。
ユーザー エクスペリエンス	**Universal Web**: 対象範囲を最大限に広げると同時に配置を簡略化し、幅広くインストールされる拡張を使用して Web ブラウザーで動作します。 **Experience First**: 最適なコンピューターやデバイスの機能を使用することで、ユーザー エクスペリエンスの品質を最大限に高めます。

関連情報

Web リソースに簡単にアクセスするには、http://www.microsoft.com/architectureguide (英語) でオンライン版の参考文献を参照してください。

Windows Azure Platform の詳細については、「Windows Azure Platform」(http://www.microsoft.com/japan/windowsazure/) を参照してください。

Windows Identity Foundation の詳細については、「Identity Management」(http://msdn.microsoft.com/en-us/security/aa570351.aspx、英語) を参照してください。

Software plus Services (ソフトウェア プラス サービス) の詳細については、MSDN デベロッパー センターで公開されている以下のリソースを参照してください。

- マルチテナント データ アーキテクチャ (http://msdn.microsoft.com/ja-jp/architecture/aa479086.aspx)

- Software + Services (S+S)
  (http://msdn.microsoft.com/en-us/architecture/aa699384.aspx、英語)
- Gianpaolo Carraro による Webcast「Software + Services for Architects」
  (http://www.microsoft.com/feeds/msdn/en-us/architecture/media/SaaS/ssForArchitects.asx 、英語)

Software plus Services (ソフトウェア プラス サービス) のアーキテクチャの詳細については、MSDN のアーキテクチャ ジャーナル (http://msdn.microsoft.com/ja-jp/architecture/cc904280.aspx) で公開されている次のリソースを参照してください。

- ソフトウェア アーキテクチャに統治される惑星
- 積極的で現実的なクラウドの利用法
- エンタープライズ マッシュアップ
- エンタープライズ IT 向けソフトウェア + サービスの消費に関する考慮事項
- ソフトウェア + サービスのプラットフォームとしての Microsoft Office
- インターネット サービス バス

オープン ソースである NHibernate フレームワークの詳細については、「NHibernate Forge」
(http://nhforge.org/Default.aspx、英語) を参照してください。

27

Office Business Application の設計

概要

この章では、Office Business Application (OBA) について説明し、一般的な OBA アーキテクチャと関連コンポーネントを示します。OBA の使用が適している一般的なシナリオについて説明し、OBA の設計に関する考慮事項と重要なパターンについてのガイダンスを提供します。また、Microsoft Office SharePoint Server (MOSS) および基幹業務 (LOB) アプリケーションと OBA の統合に関する情報も提供します。

OBA は、企業の複合アプリケーションに分類されます。OBA では、ネットワークに接続しているビジネス システムの中核機能と、Microsoft Office System の構成要素として、幅広く配置および使用されている企業の生産性サービスと生産性アプリケーションを統合するソリューションが提供されます。OBA は、エンド ユーザーのフォームを通じて管理されるビジネス ロジックを実装することにより、豊富なユーザー エクスペリエンスを提供してビジネスにおける洞察力を向上し、既存の内部システムと外部システムを統合するのに役立ちます。

通常、OBA は、新しい LOB アプリケーションや既存の LOB アプリケーションと統合されます。また、Office クライアントのリッチな UI と自動化機能を活用して、ユーザー操作を必要とする複雑なプロセスを簡略化し、エラーを最小限に抑えてプロセスを向上するのに役立ちます。実質的に、OBA では、Office クライアント アプリケーションを使用して、既存の LOB システムとユーザーの間を補完します。図 1 は、OBA の主要なコンポーネントとレイヤーを示しています。この図には、プレゼンテーション レイヤーとアプリケーション サービス レイヤーの間に、生産性レイヤーと呼ばれるレイヤーが含まれています。この生産性レイヤーには、ドキュメント ベースで共同作業の流れを保存して管理するために使用するコンポーネントが含まれます。

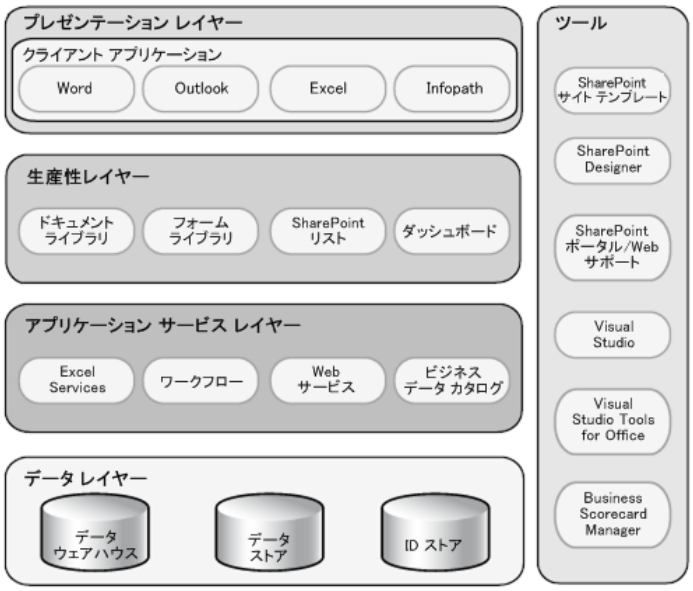


図 46

OBA の主要なコンポーネント

Office Business Application のコンポーネント

OBA は、さまざまなアプリケーションとサービスで構成されます。このようなアプリケーションとサービスが対話することで、ビジネス上の問題に対するエンド ツー エンドのソリューションを提供します。OBA には、次のいくつかまたはすべての項目が含まれていることがあり、次のいくつかまたはすべての項目を使用して作成できます。

- **Microsoft Office クライアント アプリケーション**: クライアント アプリケーションには、Outlook® (メッセージング クライアントと共同作業クライアント)、Word、Excel、InfoPath® (情報収集プログラム)、PowerPoint® (プレゼンテーション グラフィックス プログラム) などがあります。Outlook のカスタム フォームを使用して、ビジネス ロジックとさまざまなソースのデータを統合する機能を備えた UI コントロールをホストできます。Word と Excel については、作業ウィンドウ、スマート タグ、およびリボンという形でプログラミングを行えます。これにより、自然なドキュメント操作と、構造化されたビジネス データおよびビジネス プロセスの両方を実現できます。スマート タグでは、正規表現パターンを使用して、ドキュメントのテキストに含まれる ID (電話番号、政府発行の ID 番号、独自のアカウント番号など) を識別します。データと併せて関連する操作もドキュメントに含めることができます。
- **Windows SharePoint Services (WSS)**: Windows Server をベースに構築されている WSS では、ビジネス プロセスとチームの生産性の向上に役立つ、コンテンツ管理機能とコラボレーション機能が

提供されます。OBA では、WSS を使用して、ドキュメント、フォーム、およびリストを保存して共有することが可能で、オフライン同期とタスク管理がサポートされます。

- **Microsoft Office SharePoint Server (MOSS)**: MOSS では、WSS で提供される機能を拡張して、コンテンツ管理、ワークフロー、検索、ポータル、および個人用にカスタマイズされたサイトのための、企業全体で使用できる機能が提供されます。OBA では、MOSS を使用して、これらの機能を活用できるだけでなく、Excel Services を使用してレポートを作成したり、ビジネス データ カタログ (BDC) を使用して LOB にアクセスしたり、セキュリティ フレームワークを使用してシングル サインオン (SSO) 機能を活用したりすることができます。
- **テクノロジとサービス**: Excel Services を使用すると、クライアントは、Excel をいつもどおりの方法で使用してドキュメントを作成し、SharePoint サーバーに保存できます。エンド ユーザーは Web ブラウザーでドキュメントを閲覧および操作することが可能で、ソフトウェア開発者はプログラムを使用してドキュメントに格納されているビジネス ロジックを呼び出すことができます。MOSS に組み込まれている Windows Workflow Foundation (WF) の機能を使用して、注文の承認などのプロセスをキャプチャして、ユーザーによるエラーや関連する遅延を軽減することもできます。また、ASP.NET の Web ページと Web パーツによるレンダリングを使用して、企業の要件を反映した独自の Web サイトを作成できます。
- **コラボレーション機能**: コラボレーションは、Microsoft Office Communications Server (OCS)、Microsoft Office Groove® ソフトウェア、および Microsoft Exchange Server を使用して管理できます。
- **開発ツール**: 開発ツールには、SharePoint サーバーの全体管理、SharePoint Designer、Visual Studio、Visual Studio Tools for Office などを使用できます。

Office Business Application の主要なシナリオ

OBA は、オープン スタンダード、標準的なファイル形式、および Web サービスを使用して相互運用するように設計します。OBA ソリューション オブジェクトのメタデータ定義は、XML スキーマに基づいています。すべての Microsoft Office 製品は、すべてのレベルでサービスに対応しており、Office 製品で作成するビジネス ドキュメントの既定のスキーマとして、相互運用性のある Open XML ファイル形式が使用されます。OBA は、一般的に、主要なシナリオを実装する 3 つのカテゴリのいずれかに分類されます。これ以降のセクションでは、次の 3 つの分類カテゴリについて説明します。

- **企業コンテンツ管理**: ユーザーは、それぞれの役割に基づいて情報を検索して使用できます。
- **ビジネス インテリジェンス**: サーバー ベースの Excel ソリューションなどの機能を通じて、ビジネスにおける洞察力を高めることができます。
- **ユニファイド メッセージング**: コミュニケーションとコラボレーションを可能にして、チーム管理を簡略化します。

企業コンテンツ管理

企業コンテンツ管理のシナリオでは、ユーザーが Office クライアント アプリケーションを使用して、それぞれのビジネス ロールやタスクの要件に基づいて情報を検索して使用できます。Office アプリケーションは、データを提供する LOB システムと直接通信することができます。ただし、ビジネス環境では、MOSS や WSS を Office クライアントのドキュメントのコンテンツ管理ツールとして使用するのが一般的です。

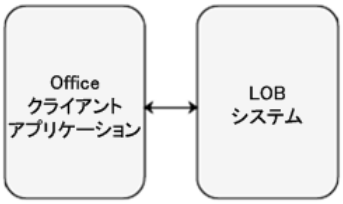


図 47a

LOB システムと直接通信する Office クライアント

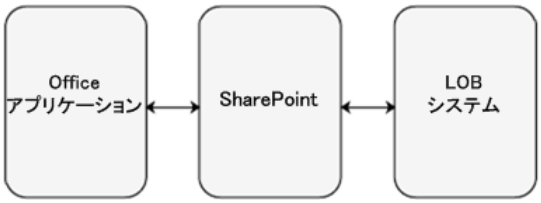


図 2b

SharePoint がインストールされている中間デバイス経由で
LOB システムと通信する Office クライアント

SharePoint ソリューションでは、Office クライアント アプリケーションに関連付けられたファイルにバージョン管理とワークフローを実装できます。また、ファイルの多くは SharePoint 環境内で変更することが可能で、MOSS で提供される機能では、Excel を使用してレポートを作成および表示できます。そのため、主要なシナリオの多くでは、SharePoint と Office クライアント アプリケーションを併用します。次の OBA パターンは、企業コンテンツ管理のシナリオを実装する際に役立ちます。これらのパターンの詳細については、この章の後半で詳しく説明します。

- Extended Reach Channel パターンでは、Office アプリケーションをチャネルとして使用して、より多くのユーザーが LOB アプリケーションの機能を使用できるようにします。
- Document Workflow パターンでは、ドキュメント ベースのプロセスを制御および監視して、ベスト プラクティスを導入し、基盤となるビジネス プロセスを強化できます。
- Collaboration パターンでは、構造化されていないヒューマン コラボレーションを使用して、構造化されたビジネス プロセスを強化します。

ビジネス インテリジェンス

ビジネス インテリジェンスのシナリオでは、サーバー ベースの Excel ソリューションなどの機能を通じて、ビジネスにおける洞察力を高めることができます。次の OBA パターンは、ビジネス インテリジェンスのシナリオを実装する際に役立ちます。これらのパターンの詳細については、この章の後半で詳しく説明します。

- Document Integration パターンでは、LOB アプリケーションから Office ドキュメントを生成したり、インフォメーション ワーカーが、ドキュメントの作成時に LOB データを操作することで、LOB データを Office ドキュメントに埋め込めるようにしたり、LOB データが含まれているドキュメントをサーバー側で処理したりすることができます。
- Composite UI パターンでは、Office ドキュメントや SharePoint Web ページで、複数のアプリケーション UI の複合がサポートされます。
- Data Consolidation パターンでは、ユーザーが複数の LOB アプリケーションを検索してデータを検出し、その結果に基づいて作業できるようにすることで、LOB データをより自然な方法で操作できます。Data Consolidation パターンでは、Discovery Navigation パターンが使用されます。

ユニファイド メッセージング

ユニファイド メッセージングのシナリオでは、コミュニケーションとコラボレーションがサポートされ、チーム管理が簡略化されます。Notification and Tasks パターンは、ユニファイド メッセージングのシナリオを実装する際に役立ちます。このパターンについては、この章の後半で詳しく説明します。Notification and Tasks パターンでは、Outlook を主要な UI として使用して、LOB アプリケーションで生成されたタスクと警告を受信し、その内容に基づいて処理を行います。

一般的な OBA パターン

OBA には、とても簡単なソリューションから非常に複雑なカスタム ソリューションまで、さまざまなソリューションがあります。一般的に、OBA には、次のセクションで説明する 1 つ以上の一般的なパターンが組み込まれています。

- Extended Reach Channel
- Document Integration
- Document Workflow
- Composite UI
- Data Consolidation (Discovery Navigation)
- Collaboration
- Notifications and Tasks

Extended Reach Channel

Extended Reach Channel パターンを使用するアプリケーションでは、Office アプリケーションをチャネルとして使用して、より多くのユーザーが LOB アプリケーションの機能を使用できるようにします。このパターンは、次のシナリオを実装する際に役立ちます。

- 現在、社内で行われている作業の重複を削減する (コンサルタントが Outlook 機能を使用して、有償プロジェクトの会議の時間を指定するなど)。
- より多くのエンド ユーザーが LOB 機能を使用できるようにする (従業員が個人情報を更新できるセルフ サービス アプリケーションを使用するなど)。
- 作業が重複していたりトレーニングが不足していることが原因で、現在ユーザーが使用していない既存システムの使用率を向上する。
- 電子メール経由でユーザーから情報を収集し、システムを自動的に更新する。

Extended Reach Channel の手法では、Direct Integration パターンおよび Mediated Integration パターンという 2 種類の統合パターンがサポートされます。次のセクションでは、これらのパターンについて説明します。

Direct Integration パターン

Direct Integration パターンは、Office クライアント アプリケーションを使用して LOB 機能をより多くのユーザーに直接公開できる方法を表します。このパターンでは、LOB インターフェイスへのアクセスを Office クライアントに直接投影するか、予定表管理などの既存の動作に拡張します。クライアント アプリケーションは、Web サービス経由で LOB データにアクセスするか、単純に LOB システムで生成された出力 (HTML など) を表示する場合があります (図 3 参照)。

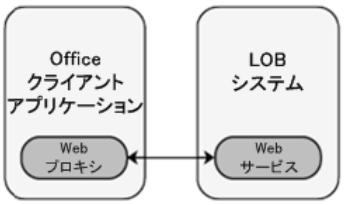


図 48a

Web サービスを使用する Direct Integration パターン

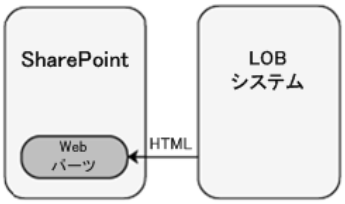


図 3b

HTML を使用する Direct Integration パターン

Mediated Integration パターン

Mediated Integration パターンは、BDC などのメタデータ ストアを使用して、より高度なレベルの抽象化を提供することにより、LOB のドキュメントを管理するための一般的な手法 (資格情報のマッピングに基づく SSO メカニズムを使用したセキュリティを含む) を提供できる方法を表します。このパターンでは、サービスとデータを構成して、複合 UI を作成する機会が増えます。メディエイター (仲介役。これは、BDC になる場合があります) は、異なる複数のソースからデータを収集し、そのデータを、クライアント アプリケーションが使用できる Office と互換性のある形式やサービスで公開します。図 5 に、Mediated Integration パターンを示します。

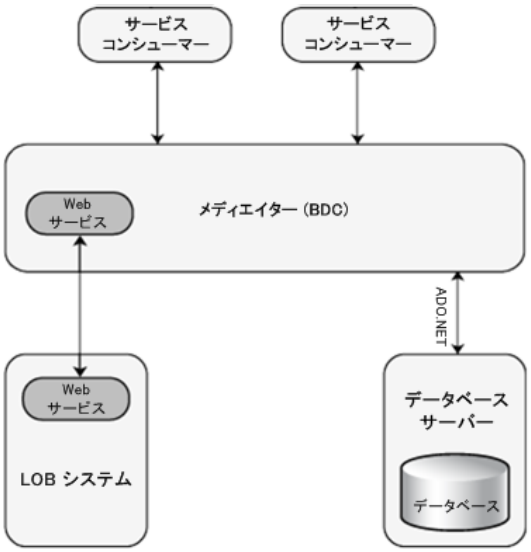


図 49

Mediated Integration パターン

Document Integration

Document Integration パターンを使用するアプリケーションでは、LOB アプリケーションから Office ドキュメントを生成したり、インフォメーション ワーカーが、ドキュメントの作成時に LOB データを操作することで、LOB データを Office ドキュメントに埋め込めるようになったり、LOB データが含まれているドキュメントをサーバー側で処理したりすることができます。Document Integration パターンは、次のシナリオを実装する際に役立ちます。

- ユーザーのデスクトップ システムにある個々の Office ドキュメントに格納される LOB データの重複を減らす。
- LOB データの特定のサブセットを Office アプリケーションに公開して、差し込み印刷やレポートの作成などのタスクを実行する。
- LOB データの項目が含まれている Office ドキュメントを適切な形式で生成して、データが変更されたときに自動的に更新されるようにする: 一般的なレイアウトを手動で作成しないようにします。Office アプリケーションでは、テンプレートを適用できる場合は、テンプレートを使用して一般的なレイアウトを作成する必要があります。
- LOB データをサーバー側でカスタム処理する必要があるドキュメントを生成する: このように処理した LOB データを埋め込む際には、オープン スタンダードを使用します。
- 受信ドキュメントを受け取り、埋め込みデータを処理して、そのデータを LOB システムに適用する。

Document Integration の手法では、XML を使用して LOB システムと情報をやりとりする 4 種類の統合パターンがサポートされます。Application Generated Documents が最もシンプルなパターンですが、3 つのインテリジェント ドキュメントの統合パターン (Intelligent Documents/Embedded LOB Information パターン、Intelligent Documents/Embedded LOB Template パターン、および Intelligent Documents/LOB Information Recognizer パターン) もあります。次のセクションでは、これらのパターンについて説明します。

Application Generated Documents パターン

Application Generated Documents パターンは、バッチ指向のサーバー側の処理を使用して、LOB システムがビジネス データと Office ドキュメントをマージできる方法を示します (ただし、クライアント側で生成することもできます)。一般的な例として、Excel へのデータのエクスポートや、Word でのレポートやレターの作成などが挙げられます。ドキュメントとデータの統合で最も一般的に使用されるパターンを次に示します。

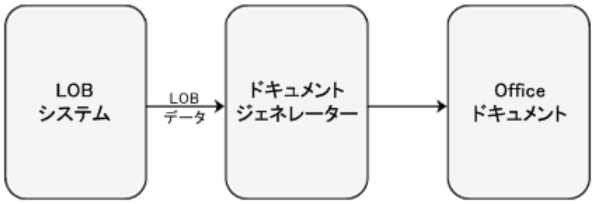


図 50

Application Generated Documents パターン

Intelligent Documents/Embedded LOB Information パターン

Intelligent Documents/Embedded LOB Information パターンは、LOB データを Offfice ドキュメントの本文に直接埋め込んだり、XML ドキュメントに埋め込んで、コンテンツ コントロールを使用して公開したりできる方法を表します。また、Office アプリケーションでは、Office のカスタム作業ウィンドウ (CTP) を使用して、インフォメーション ワーカーが参照または検索できる LOB データを表示し、ドキュメントに埋め込むことができます。図 6 に、Intelligent Documents/Embedded LOB Information パターンを示します。

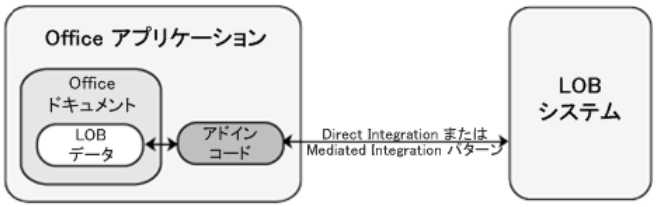


図 51

Intelligent Documents/Embedded LOB Information パターン

Intelligent Documents/Embedded LOB Template パターン

Intelligent Documents/Embedded LOB Template パターンは、テンプレートを使用して、LOB システムのメタデータとドキュメント マークアップ (コンテンツ コントロール、XML スキーマ、ブックマーク、名前付き範囲、スマート タグなど) を組み合わせることができる方法を表します。実行時に、テンプレートが LOB データの適切なインスタンスとマージされ、ドキュメントが作成されます。マージは、Office クライアント アプリケーションのアドインを通じて行うか、サーバーで行えます。

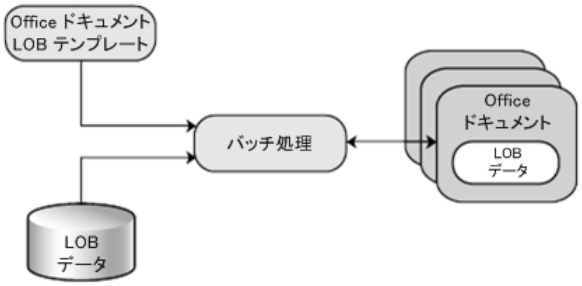


図 52

Intelligent Documents/Embedded LOB Template パターン

Intelligent Documents/LOB Information Recognizer パターン

Intelligent Documents/LOB Information Recognizer パターンは、メタデータとドキュメント マークアップ (コンテンツ コントロール、XML スキーマ、ブックマーク、名前付き範囲、スマート タグなど) に、LOB システムで認識されるデータを含められる方法を表します。アプリケーションでは、このデータを使用して LOB システムを更新したり、ユーザーに追加機能を提供したりできます。サーバー側では、アプリケーションがこの情報を使用して、ワークフローを開始する場合があります。また、クライアント側では、コンテキストの影響を受ける情報 (Word ドキュメントで名前が認識される顧客の詳細など) がアプリケーションで提供される場合があります。

Document Workflow

Document Workflow パターンを使用するアプリケーションでは、ドキュメント ベースのプロセスを制御および監視して、ベスト プラクティスを導入し、基盤となるビジネス プロセスを強化できます。Document Workflow パターンは、次のシナリオを実装する際に役立ちます。

- (多くの場合は電子メール経由で) 情報をやり取りして、多段階のタスク (予測、予算作成、インシデント管理など) を実行する。
- 特定の法的コンプライアンスや企業コンプライアンスの手続きに従い、監査情報を管理する必要がある。

- 複雑なドキュメント処理や条件付きのルーティング タスクを実行したり、ベスト プラクティスに基づいた規則を実装する必要がある。

このパターンを実装する場合は、ワークフローの要件を考慮する必要があります。ただし、可能な場合は、カスタムのワークフロー コンポーネントを構築することは避けて、SharePoint のワークフロー機能を使用するようにします。Document Workflow の手法では、ワークフローを開始する次の 2 種類の統合パターンがサポートされます。

- **LOB Initiated Document Workflow パターン**: ドキュメントを SharePoint ドキュメント ライブラリに保存したり、InfoPath フォームを送信したりする操作によって、ドキュメントが SharePoint のドキュメント ワークフローに自動的に渡されます。ワークフローでは、アプリケーションの要件に応じて、ドキュメントをリスト内の次の受信者に送信したり、コピーを保存したり、ドキュメントで処理を実行したりする場合があります。
- **Cooperating Document Workflow パターン**: ドキュメントと LOB システムの間で、特定の規則に従ったり特定の操作を行えないようにする必要がある一連のやり取りが発生します (たとえば、処理の特定の段階で送信済みドキュメントを編集できないようにしたり、特定の情報を抽出したり、この情報を LOB システムに返したりします)。通常、このパターンでは、フロー ロジックを提供する SharePoint の補完的なワークフローが使用されますが、インテリジェント ドキュメントでは LOB 操作のメカニズムが提供されます。複雑なシナリオでは、LOB システムは、ワークフローでドキュメントを更新することもあります。

Composite UI

Composite UI パターンを使用するアプリケーションでは、Office ドキュメントや SharePoint Web ページで、複数のアプリケーション UI の複合がサポートされます。Composite UI パターンは、次のシナリオを実装する際に役立ちます。

- さまざまな種類の情報を収集し、収集した情報を 1 つの UI ページや UI 画面に表示する。
- ネットワークに接続している複数のシステムで公開されるデータを使用して、そのデータを 1 つの UI ページや UI 画面に表示する。
- カスタマイズ可能な複合インターフェイスを提供し、ユーザーがインターフェイスを要件に最適な状態に変更できるようにする必要がある。

このパターンを実装する場合は、Office 標準に従うようにします。また、必要な機能が提供される Web パーツを使用できる場合にはカスタム コンポーネントを作成しないようにします。Composite UI の手法では、情報を複合 UI に統合する、さまざまな統合パターンがサポートされます。

- **Context Driven Composite User Interface パターン**: コンテキスト情報によって UI の構成が決まります。コンテキスト情報には、静的な情報 (アプリケーションの構成、Outlook ビューに追加されるタブなど) と動的な情報 (ソース ドキュメントでタブ ベースのデータの表示/非表示を切り替えるなど) があります。複合 UI の各領域では、Office のクライアント コンポーネントを通じて情報が提供されます。ただし、ユーザーは、ドキュメント コンポーネントと LOB システムにあるソース データ間で、実行時にリンクを動的に変更することはできません。
- **Mesh Composite View パターン**: UI に、ASP.NET Web パーツや MOSS コンポーネントなどのコンポーネントが含まれます。これらのコンポーネントが補完的にやり取りすることで、同じ LOB システムや異なる LOB システムのデータが公開されます。たとえば、カスタマー リレーションシップ マネジメント (CRM) システムの顧客のビューを表す Web パーツは、このビューが構築されているときには、エンタープライズ リソース プランニング (ERP) システムの未出荷注文の一覧を表す Web パーツに接続される場合があります。CRM の Web パーツで顧客が選択されると、イベントが発生し、選択された顧客の ID に関する情報が未発注注文の Web パーツに提供されることで、その注文の状態を表示できます。
- **RSS and Web Services Composition パターン**: RSS フィードとして発行されたデータや Web サービス経由で発行されたデータを組み合わせる、Mesh Composite View パターンの特殊なバージョンです。複数の SharePoint データ ビュー Web パーツ (またはカスタム パーツ) では、発行されたデータを書式設定して UI に表示します。たとえば、いくつかのサプライヤーのカタログを表示する複合ビューでは、詳細情報を提供するサプライヤーの Web サイトのページへのリンクが、発行された各項目に表示されます。
- **Analytics パターン**: エンド ユーザーにデータ分析ダッシュボードを提供する、Mesh Composite View パターンの特殊なバージョンです。このパターンでは、複合 UI で、Excel Services と MOSS 2007 の Excel Services Web パーツを使用してデータやグラフを表示したり、その他の Web パーツを使用して LOB システムや他のソースのカスタム データと情報を表示したりできます。MOSS に用意されている便利な Web パーツに、主要業績評価指標 (KPI) という Web パーツがあります。この Web パーツを使用すると、ユーザーは、BDC のリストを含む任意の SharePoint リストのデータに基づいて KPI を定義できるようになります。

Data Consolidation (Discovery Navigation)

Data Consolidation パターンを使用するアプリケーションでは、ユーザーが複数の LOB アプリケーションを検索してデータを検出し、その結果に基づいて作業できるようにすることで、LOB データをより自然な方法で操作できます。このアプリケーションでは、Office アプリケーションで使用できる、LOB の十分なエンティティ データを使用して、このデータに基づいて作業を行います。Data Consolidation パターンでは Discovery Navigation パターンが使用され、次のシナリオを実装する際に役立ちます。

- 1 つの LOB システムを検索する機能を提供する。
- 複数の LOB システムを検索する機能を提供する。
- さまざまな LOB システムと他のデータ ソースを検索する機能を提供する。

Data Consolidation パターン

Data Consolidation パターンでは、1 つまたは複数のソースの検索結果を 1 つの結果セットに統合し、その結果にリンクする Uniform Resource Identifier (URI) だけでなく、検出された項目に関連する操作を提供することで、インフォメーション ワーカーに一貫した検索エクスペリエンスを提供します。図 8 に、コンテキスト インデックスを作成する Data Consolidation パターンを示します。

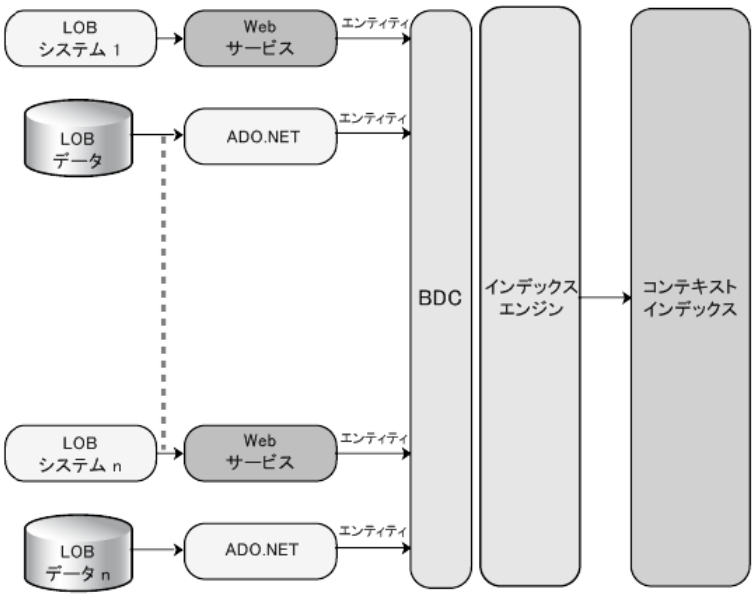


図 53

さまざまなソースで照合された情報が含まれるコンテキスト インデックス

LOB プロセスの開始

Data Consolidation パターンのサブパターンでは、アクション リンクを使用して、ワークフローの開始やドキュメントでの処理の実行などの LOB 操作を開始できます (図 9 参照)。

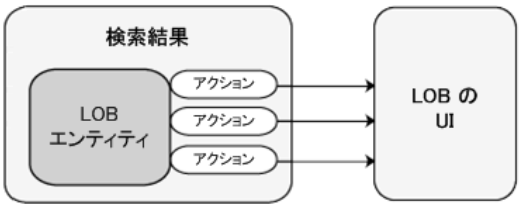


図 54

検索結果の項目のアクションに基づいて開始される LOB プロセス

Collaboration

Collaboration パターンを使用するアプリケーションでは、構造化されないヒューマン コラボレーションを使用して、構造化されたビジネス プロセスを強化します。Collaboration パターンは、次のシナリオを実装する際に役立ちます。

- LOB システムの操作につながるユーザーどうしのやり取りが発生する (たとえば、顧客が注文する前の販売機会についてのディスカッションなど)。
- LOB アプリケーションでコンテンツとユーザー操作を構造化されない形式で照合し、後から、その情報を構造化された形式で公開する必要がある。
- ユーザーが編集できる構造化されない形式 (Wiki、ディスカッション サイトなど) で情報を提供する。

Collaboration パターンでは MOSS のチーム サイト テンプレートを使用しています。このテンプレートにより、ユーザーは、ドキュメント ライブラリ、ディスカッションとタスク リスト、チーム予定表、および単純なプロジェクト管理機能を使用して、特定の業務上の問題について共同作業を行うことができます。このサイトは、LOB のデータを使用して準備および設定することが可能で、適切なライブラリやリストで LOB プロセスへのリンクが公開されます。このサイトには、Office ドキュメントか Web ブラウザーを使用してアクセスできます。

Notifications and Tasks

通知とタスクをサポートする必要があるアプリケーションでは、Outlook を主要な UI として使用して、LOB アプリケーションで生成されたタスクと警告を受信し、その内容に基づいて処理を行うことができます。Outlook に加えて、SharePoint には、簡易メール転送プロトコル (SMTP) を使用してほとんどの電子メール システムとやり取

りできる、通知とタスクのサービスが用意されています。Notifications and Tasks パターンは、次のシナリオを実装する際に役立ちます。

- タスクを割り当てて、エンド ユーザー向けに通知を生成する。
- 複数の LOB 操作を統合し、状態やプロセスに関する要件をユーザーに通知する必要がある。

電子メール ベースの Notifications and Tasks の手法では、タスクと状態についてユーザーに通知することができる、さまざまな統合パターンがサポートされます。

- **Simple Task and Notification Delivery パターン**: LOB システムは、Outlook タスクと電子メール メッセージの形でユーザーにタスクと通知を配信します。この情報の流れは、一方向です。タスクや通知の詳細は、タスクと電子メール メッセージの本文に埋め込まれますが、ユーザーによる変更は LOB システムに反映されません。タスクと通知の配信に関するオプションには、Microsoft Exchange Server にタスクや通知を配信したり (押し出し型)、Outlook のアドインを使用してタスクや通知をフェッチしたり (引き出し型)、ユーザーが購読できる RSS フィードを発行したりする方法があります。
- **Direct Task Synchronization パターン**: LOB システムでは、Exchange Server または Outlook を通じてタスクをユーザーに送信し、情報が双方向に同期されます。ユーザーと LOB は、タスクをいつでも更新することが可能で、変更は LOB システムに反映されます。タスクは、LOB のワークフローに組み込まれる場合があります。
- **Mediated Task Synchronization パターン**: Direct Task Synchronization パターンの変化形です。MOSS が LOB システムと Outlook の間のメディエイターとして動作して、タスクを同期します。タスクは、LOB システムによって SharePoint タスク リストに発行されます。このタスク リストは、Outlook のネイティブな同期メカニズムを使用して、Outlook の仕事と同期されます。ユーザーが Outlook で仕事を更新すると、その変更は自動的に SharePoint に反映されます。SharePoint では、変更が加えられたことを示すイベントが発生するので、カスタム コードによって LOB システムが更新されます。
- **Intelligent Tasks and Notifications パターン**: ユーザーは、Outlook の CTP に表示されるアクション リンクを使用して、LOB システムから送信されたタスクや通知に基づいて、特定のアクションを開始できます。一般的なタスクには、LOB システムへの自動ログイン、適切な情報の検索、および情報の更新があります。たとえば、上司が部下の休暇申請を承認するために、人事部から送信された電子メール メッセージを閲覧しているときに、CTP に表示されたアクション リンクを使用して LOB システムを更新することで、休暇申請を承認または却下できます。

- **Form-based Tasks and Notifications パターン**: Intelligent Tasks and Notification パターンの変化形です。電子メール メッセージに、LOB システムであらかじめデータが設定された InfoPath フォームが添付されます。ユーザーは、電子メール メッセージを開封して、フォームに情報を入力し、LOB システムに送信できます。InfoPath には、データの検証、カスタムの演算、およびロジックが用意されているので、ユーザーがフォームに情報を入力する際に役立ちます。InfoPath の CTP には、ユーザーの入力を支援するために、LOB システムから抽出した追加情報を表示できます。このパターンの変化形では、MOSS に組み込まれている InfoPath Forms Services を使用して、ユーザーが InfoPath をインストールしなくても、Web ブラウザーでフォームに情報を入力することができます。

設計に関する一般的な考慮事項

サポートする必要があるシナリオと、そのシナリオに適した Office クライアント アプリケーションの種類に基づいて、適切な OBA を設計します。OBA を設計する際には、前のセクションで説明した基本的なパターンだけでなく、次のガイドラインについても考慮します。

- **直接統合するのではなく Mediated Integration パターンを使用することを検討する**: OBA を拡張リーチ チャネルとして設計する場合は、ドキュメントにインターフェイスを直接実装できます。たとえば、Excel ブックに独自の入力フォームを含めることができます。ただし、この手法ではカスタム コードが必要になり、機能の再利用が制限されます。Mediated Integration パターンでは、SharePoint やビジネス データ カタログなどのアプリケーションを利用して、インターフェイスを物理的なドキュメントから分離できます。
- **Open XML ベースのスキーマを使用して LOB データをドキュメントに埋め込む**: Open XML は、European Computer Manufacturers Association (ECMA) によって策定された国際標準です。これは、Office 2007 アプリケーションのほかに、多くの独立系ベンダーやプラットフォームでサポートされています。Open XML を使用すると、Office アプリケーションと他のプラットフォーム用に開発されたアプリケーションとの間でデータを共有できます。
- **再利用される一般的なレイアウト用に LOB のドキュメント テンプレートを作成する**: LOB テンプレートには、LOB に関連するマークアップとメタデータが含まれています。これらは、後から、特定の LOB データのインスタンスにバインドできます。つまり、LOB データをドキュメント テンプレートにマージすることで、新しいドキュメントを生成できます。エンド ユーザーは、開発者に頼らずにカス

タム ドキュメントを作成することが可能で、サーバー側のバッチ処理を使用して複雑なドキュメントを生成できます。

- **MOSS を使用して、ドキュメントの確認と承認のプロセスを制御する**: MOSS には、ドキュメントを確認して承認するための基本的なワークフロー プロセスをサポートする、すぐに使える機能が用意されています。処理要件が複雑な場合は、WF を使用して、SharePoint のワークフロー機能を拡張できます。
- **ヒューマン コラボレーションに Collaboration パターンを使用する**: ほとんどの LOB アプリケーションは、構造化されたビジネス プロセスの処理に優れていますが、ビジネス プロセスに伴う構造化されないユーザーどうしのやり取りの処理には適していません。Collaboration パターンを実装するサイトでは、他のユーザーとのコラボレーションを行うためのインターフェイスを提供することで、この問題が解決されます。SharePoint チーム サイト テンプレートには、このパターンが実装されています。
- **リモート データ同期の要件を考慮する**: 作成、更新、または配布されるドキュメントは、LOB システムと同期してから、後で使用できるように格納する必要があります。LOB システムはトランザクション指向の作業を処理する際に非常に役立ちますが、作業間で発生する大規模な処理のキャプチャには適していません。

セキュリティに関する考慮事項

さまざまなクライアント アプリケーションを使用してデータや機能を公開し、企業の LOB データにアクセスする Office Business Application では、セキュリティが重要になります。リソースへのすべてのアクセスをセキュリティで保護し、ネットワーク上で渡されるデータを保護することが重要です。OBA を作成する際には、セキュリティを確保するために、次のガイドラインを考慮します。

- SSO を実装して、ユーザーが、現在のログオン資格情報や、統合されたサービス (Active Directory、SharePoint など) によって検証された資格情報を使用して、クライアント アプリケーションとネットワーク機能にアクセスできるようにすることを検討します。
- 可能な場合は、セキュリティで保護されたネットワーク外に渡されるメッセージの暗号化を検討します。チャネルの暗号化メカニズム (インターネット プロトコル セキュリティ (IPSec) など) を使用して、サーバーとクライアント間のネットワーク接続をセキュリティで保護できます。

- 役割の資格情報を使用してデータにアクセスする際には、信頼されたサブシステム モデルを使用して、必要な接続の数を最小限に抑えます。信頼されたサブシステム モデルの詳細については、第 19 章「物理ティアと配置」を参照してください。
- 機密データを公開する必要がない場合は、サーバーでデータをフィルター処理して、クライアント アプリケーションに機密データが公開されないようにすることを検討します。

配置に関する考慮事項

OBA ソリューションは、Windows インストーラー パッケージか ClickOnce テクノロジを使用して配置できます。

- ClickOnce インストールでは、ユーザーによる操作はほとんど必要がなく、自動更新を利用することが可能で、開発者による作業もほとんど必要ありません。しかし、ClickOnce の配置は、大きなソリューションの構成要素ではない、単一ソリューションの配置にしか使用できません。追加のファイルやレジストリ キーは配置できません。また、ユーザーと対話してインストールを構成することも、ブランド化されたインストールを提供することもできません。
- Windows インストーラーによるインストールでは、追加のコンポーネントやレジストリ設定を配置できます。また、ユーザーと対話してインストールを構成することが可能で、ブランド化された独自のインストールがサポートされます。ただし、高度な構成が必要になり、開発者の作業が増加し、自動更新を提供することはできません。

関連する設計パターン

次の表に示すように、主要なパターンは、コラボレーション、複合 UI、データの統合、ドキュメントの統合、ドキュメント ワークフロー、拡張リーチ チャネル、タスクと通知などのカテゴリに分類されます。各カテゴリの設計を行う際には、これらのパターンの使用を検討します。

カテゴリ	関連するパターン
コラボレーション	**Collaboration**: 構造化されないヒューマン コラボレーションを使用して、構造化されたビジネス プロセスを強化します。
複合 UI	**Analytics**: エンド ユーザーにデータ分析ダッシュボードを提供する、Mesh Composite View パターンの特殊なバージョンです。

	Context Driven Composite User Interface: コンテキスト情報を使用して、UI の構成を決定します。 **Mesh Composite View**: UI に含まれるコンポーネント (ASP.NET Web パーツ、MOSS コンポーネントなど) を使用し、これらのコンポーネントが補完的にやり取りすることで、同じ LOB システムや異なる LOB システムのデータが公開されます。 **RSS and Web Services Composition**: RSS フィードとして発行されたデータや Web サービス経由で発行されたデータを組み合わせる、Mesh Composite View パターンの特殊なバージョンです。
データの統合	**Discovery Navigation**: ユーザーは、複数の LOB アプリケーションを検索してデータを検出し、その結果に基づいて操作を行えます。
ドキュメントの統合	**Application Generated Documents**: バッチ指向のサーバー側の処理を使用して、LOB システムがビジネス データと Office ドキュメントをマージします。 **Embedded LOB Information**: LOB データを Offfice ドキュメントの本文に直接埋め込んだり、LOB データを XML ドキュメントの一部として埋め込み、コンテンツ コントロールを使用して公開したりします。 **Embedded LOB Template**: テンプレートを使用して、LOB システムのメタデータとドキュメント マークアップ (コンテンツ コントロール、XML スキーマ、ブックマーク、名前付き範囲、スマート タグなど) を組み合わせます。 **LOB Information Recognizer**: メタデータとドキュメント マークアップ (コンテンツ コントロール、XML スキーマ、ブックマーク、名前付き範囲、スマート タグなど) に、LOB システムで認識されるデータを含めます。
ドキュメント ワークフロー	**Cooperating Document Workflow**: ドキュメントと LOB システムの間で、特定の規則に従ったり特定の操作を行えないようにする必要がある、一連のやり取りが発生します。 **LOB Initiated Document Workflow**: ドキュメントを SharePoint ドキュメント ライブラリに保存したり、InfoPath フォームを送信したりする操作によって、ドキュメントが SharePoint のドキュメント ワークフローに自動的に渡されます。
拡張リーチ チャネル	**Direct Integration**: LOB インターフェイスへのアクセスを Office クライアントに直接投影するか、予定表管理などの既存の動作に拡張します。 **Mediated Integration**: メディエイター (仲介役。これは、BDC になる場合

	があります) は、異なる複数のソースからデータを収集し、そのデータを、クライアント アプリケーションが使用できる Office と互換性のある形式やサービスで公開します。
タスクと通知	**Direct Task Synchronization**: LOB システムでは、Exchange Server または Outlook を通じてタスクをユーザーに送信して、情報が双方向に同期されます。 **Form-based Tasks and Notifications**: Intelligent Tasks and Notification パターンの変化形です。電子メール メッセージに、LOB システムであらかじめ設定された InfoPath フォームが添付されます。 **Intelligent Tasks and Notifications**: ユーザーは、Outlook の CTP に表示されるアクション リンクを使用して、LOB システムから送信されたタスクや通知に基づいて、特定のアクションを開始できます。 **Mediated Task Synchronization**: Direct Task Synchronization パターンの変化形です。MOSS が LOB システムと Outlook の間のメディエイターとして動作して、タスクを同期します。 **Simple Task and Notification Delivery**: LOB システムは、Outlook タスクと電子メール メッセージの形でユーザーにタスクと通知を配信します。この情報の流れは、一方向です。

OBA パターンの詳細については、Rob Barker、Joanna Bichsel、Adam Buenz、Steve Fox、John Holliday、Bhushan Nene、Karthik Ravindran 共著『6 Microsoft® Office Business Applications for Office SharePoint® Server 2007』(Microsoft Press、2008 年) を参照してください。また、この書籍の引用については、「Office Business Applications の概要」(http://msdn.microsoft.com/ja-jp/library/bb614538.aspx) を参照してください。

関連情報

Web リソースに簡単にアクセスするには、http://www.microsoft.com/architectureguide (英語) でオンライン版の参考文献を参照してください。

- Automating Public Sector Forms Processing and Workflow with Office Business Application (http://blogs.msdn.com/singaporedpe/archive/tags/OBA/default.aspx、英語)

- Office Business Applications の概要 (Part 1 of 2)
  (http://msdn.microsoft.com/ja-jp/library/bb614538.aspx)
- MSDN Industry Center - Public Sector - OBA RAP for E-Forms Processing
  (http://msdn2.microsoft.com/en-us/architecture/bb643796.aspx、英語)
- OBA の PowerPoint スライドとソース コード
  (http://msdn2.microsoft.com/en-us/architecture/bb643796.aspx、英語)
- OBA Central (http://www.obacentral.com/、英語)
- Integrating LOB Systems with the Microsoft Office System (http://msdn.microsoft.com/en-us/architecture/bb896607.aspx、英語)
- **Office を使用した開発について**
  (http://msdn.microsoft.com/ja-jp/office/aa905371.aspx**)**

28

SharePoint LOB アプリケーションの設計

概要

この章では、典型的な SharePoint 基幹業務 (LOB) アプリケーションのアーキテクチャとこのアプリケーションに含まれているコンポーネントを紹介します。また、SharePoint LOB アプリケーションの主要なシナリオと設計に関する重要な考慮事項についても説明します。さらに、SharePoint LOB アプリケーションを設計する際の、配置、主要なパターン、およびテクノロジに関する考慮事項についても紹介します。

Microsoft Windows Server® は、SharePoint LOB アプリケーションを実行する主要なオペレーティング システムです。SharePoint では、フロントエンド Web サーバーとしてインターネット インフォメーション サービス (IIS) を使用して Web サイトをホストし、サイト定義、コンテンツ タイプの定義、発行されたコンテンツ、および構成データに関するネットワークに接続されたストアとして SQL Server を使用して、幅広いマイクロソフト プラットフォームと密接に統合します。SharePoint LOB アプリケーションを構成すると、Web ファームに配置することでスケールアウトして多数のユーザーにサービスを提供できる Web サイトを通じてインターネットに公開されるコンテンツを発行したり、ASP.NET と統合して、これらのサイトで LOB データを表示したりすることができます。SharePoint LOB アプリケーションでは、ASP.NET の Web パーツ、スタイル、テーマ、テンプレート、サーバー コントロール、およびユーザー コントロールを UI に使用できます。図 1 は、SharePoint LOB アプリケーションの主要な機能とレイヤーを示しています。

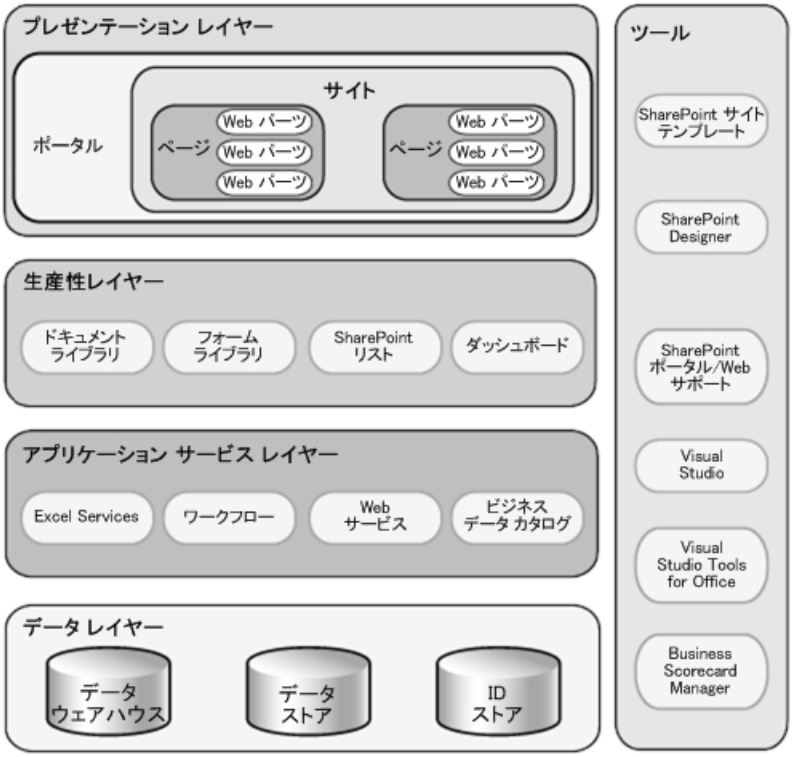


図 55

SharePoint LOB アプリケーションの主要な機能

第 27 章で説明した Office Business Application (OBA) でも、LOB プロセス同士を統合し、Windows SharePoint Services (WSS) と Microsoft Office SharePoint Server (MOSS) をベースに構築され役割がカスタマイズされたビジネス ポータルを使用して、データ アクセス、データ分析、およびデータ操作の機能豊富なユーザー エクスペリエンスを提供できます。

SharePoint LOB アプリケーションの論理レイヤー

SharePoint LOB アプリケーションの各レイヤーは次のとおりです。

- **プレゼンテーション レイヤー**: これはアプリケーションの UI です。ユーザーは Web ブラウザーを使用して、Web ページで構成された SharePoint Server のポータル サイトに接続します。Web ページは Web パーツを使用して組み立てることが可能で、Web パーツではプレゼンテーション レベルの機能豊富な構成が提供されます。Office クライアント アプリケーション向けの Web パーツも使用できるので、アプリケーション固有の機能を実装するカスタム Web パーツを構築できます。
- **生産性レイヤー**: Excel などの Office ドキュメントは、ドキュメント ライブラリに格納されます。Office アプリケーションのタスクを自動化するフォームは、フォーム ライブラリに格納されます。生

産性レイヤーには、SharePoint リストや Excel スプレッドシートの形式でレポートを作成して発行する機能も実装されています。また、複数のサービスから取得したデータで構成された、ダッシュボードの形式で出力を生成することもできます。さらに、第 27 章で説明したように、Office クライアント アプリケーションは情報処理とコラボレーションにも使用できます。

- **アプリケーション サービス レイヤー**: これは、生産性レイヤーとプレゼンテーション レイヤーで使用されるサービスを公開する、アプリケーション内の再利用可能なレイヤーです。このレイヤーには、レポート作成用の Excel Services、Windows Workflow Foundation (WF) を使用してビジネス プロセスやドキュメントのライフサイクル管理を実装するワークフロー、およびその他のビジネス Web サービスが含まれています。また、クライアントはビジネス データ カタログ (BDC) を使用してデータにアクセスできます。
- **データ レイヤー**: このレイヤーでは、アプリケーションで必要なあらゆる種類のデータを格納してアクセスするためのメカニズムをカプセル化します。このレイヤーには、役割と ID、操作データ、および LOB データが格納されたデータ ウェアハウスが含まれています。

物理ティアの配置

前のセクションでは、SharePoint LOB アプリケーションのコンポーネントや機能を独立したレイヤーに論理的にグループ化して説明しました。同様に、インフラストラクチャの個別のサーバーにコンポーネントが物理的に分散していることについても理解する必要があります。次に、一般的なシナリオとガイドラインを示します。

- 信頼性とパフォーマンスを最大限に高めるために、独立したデータベース サーバーまたはデータベース クラスターに SharePoint のデータベースを配置します。
- 非分散型シナリオでは、プレゼンテーション レイヤー、生産性レイヤー、およびアプリケーション サービス レイヤーを同じ Web サーバーまたは Web ファームに配置します。
- 分散シナリオでは、プレゼンテーション レイヤーのコンポーネント (ポータル、サイト、ページ、および Web パーツ) を Web サーバーまたは Web ファームに配置して、他のレイヤーとコンポーネントを独立したアプリケーション サーバーまたはアプリケーション ファームに配置できます。
- 負荷が高い場合のパフォーマンスを最大限に高めるために、アプリケーション サービス レイヤーのコンポーネントを独立したアプリケーション サーバーまたはアプリケーション ファームに配置することがあります。

主要なシナリオと機能

SharePoint LOB アプリケーションは、オープン スタンダード、標準的なファイル形式、および Web サービスを使用して相互運用するように設計します。SharePoint LOB ソリューション オブジェクトのメタデータ定義は、XML スキーマに基づいています。すべての Office system 製品は、すべてのレベルでサービスに対応しており、製品で作成するビジネス ドキュメントの既定のスキーマとして、相互運用性のある Open XML ファイル形式が使用されます。

MOSS を使用すると、SharePoint LOB アプリケーションにおけるコンテンツ管理機能の提供とビジネス プロセスの実装に役立ちます。SharePoint サイトでは、特定のコンテンツの発行、コンテンツ管理、レコード管理、およびビジネス インテリジェンスに関する要件がサポートされます。また、人、ドキュメント、およびデータを効果的に検索し、フォーム ベースのビジネス プロセスに参加し、大量のビジネス データにアクセスして分析できます。

SharePoint LOB アプリケーションの機能は次のとおりです。

- **ワークフロー**: MOSS は WF と統合され、開発者が MOSS を使用して単純なワークフローを作成し、そのワークフローを SharePoint のドキュメント ライブラリに関連付けることができます。また、ユーザーは SharePoint Designer を使用してカスタム ワークフローを作成できます。
- **ビジネス インテリジェンス**: MOSS では、大規模なデータ操作と分析をサポートする対話型のビジネス インテリジェンス ポータルがユーザーに提供されます。ユーザーは、コードを記述しなくても複数のデータ ソースからダッシュボードを作成できます。主要業績評価指標 (KPI) は、Excel Services、SharePoint リスト、SQL Server Analysis Services キューブ、およびその他さまざまなソースに基づいて定義できます。データは SharePoint 内部でホストされているので、検索、ワークフローなど、他の SharePoint サービスで積極的に使用できます。
- **コンテンツ管理**: MOSS には Microsoft Content Management Server (MCMS) の機能が組み込まれているので、包括的な Web コンテンツ管理機能を利用することが可能で、この機能には SharePoint プラットフォームから直接アクセスできます。
- **検索**: MOSS のエンタープライズ検索は、幅広く拡張可能なコンテンツの収集機能、インデックス作成機能、およびクエリ機能が備わり、フルテキスト検索とキーワード検索がサポートされた共有サービスです。
- **ビジネス データ カタログ**: BDC を使用すると、Web パーツ、InfoPath Forms Server、および検索機能に企業データを公開できます。開発者は BDC を使用して、なじみのあるインターフェイスでユーザーが LOB データを操作できるアプリケーションを構築できます。

- **Open XML ファイル形式**: Office system のすべてのアプリケーションで Open XML ファイル形式が採用されているので、サーバー側では、さまざまなドキュメントの操作を容易に実行できます。

設計に関する一般的な考慮事項

SharePoint では、LOB アプリケーションとやり取りする際に使用する基本機能の多くが提供されますが、設計に関して考慮しなければならない一般的な問題がいくつかあります。このような問題には、ユーザー エクスペリエンス、クライアント インターフェイスの選択、運用と保守に関する問題などがあります。SharePoint LOB アプリケーションを設計する際には、次のガイドラインを考慮します。

- **ユーザーの役割に合わせてカスタマイズされたユーザー エクスペリエンスを実現する**: ユーザーの役割に基づいて提供する UI オプションを変更します。SharePoint には、ユーザーの役割やグループに基づいて表示を自動的にカスタマイズできる機能が用意されています。セキュリティ グループや対象ユーザーを使用して、ユーザーに関連したオプションのみを提供します。
- **LOB システムを Office クライアント アプリケーションと統合する**: Direct Access パターン、Mediated パターンなどのパターンを選択して、LOB システムをソリューションや機能要件に固有の Office クライアント アプリケーションと統合します。Direct Access パターンの場合は、ADO.NET サービスまたは Web サービスの使用を検討します。Mediated パターンの場合は、中間ティアのアプリケーション サーバーとして MOSS を使用することを検討します。これらのパターンの詳細については、第 27 章「Office Business Application の設計」を参照してください。
- **レイヤー間の密結合を避ける**: Web サービスを使用して依存関係を解決し、レイヤー間の密結合を避けます。
- **リモート データ同期の要件を考慮する**: 作成、更新、または配布されるすべてのドキュメントは、LOB システムと同期してから、後で使用できるように格納する必要があります。LOB システムはトランザクション指向の作業の処理をする際に非常に役立ちますが、作業間で発生する大規模な処理のキャプチャには適していません。
- **SharePoint と OBA で使用できるようにサービス経由でバックエンド LOB データを公開する**: ネットワークに接続しているデータ システムをサービス経由で公開すると、SharePoint と OBA の拡張機能ではユーザーがデータの要求、操作、および形式変更を実行できるようになります。このように、大量のコードを開発しなくても、SharePoint を使用して、ネットワークに接続しているシステムの動作を拡張できます。

設計に関する具体的な問題

設計を策定する際に考慮する必要がある一般的な問題がいくつかあります。これらの問題は、設計の具体的な領域に分類できます。次のセクションでは、各領域で一般的に発生する問題を解決するのに役立つガイドラインを提供します。

- ビジネス データ カタログ
- ドキュメントとコンテンツのストレージ
- Excel Services
- InfoPath Forms Services
- SharePoint オブジェクト モデル
- Web パーツ
- ワークフロー

ビジネス データ カタログ

BDC を使用すると、Web パーツ、InfoPath Forms Server、および検索機能に企業データを公開できます。開発者は BDC を使用して、なじみのあるインターフェイスでユーザーが LOB データを操作できるアプリケーションを構築できます。BDC ベースのアプリケーションを開発する際には、次のガイドラインを考慮します。

- データ ソースの構造を確認して、BDC で直接使用するのに適した構造にし、データの用途 (検索、ユーザー プロファイル、単純な表示など) を決定します。適切な検索範囲を定義して、不要なデータが公開されないようにします。また、データ ソースで最新のデータ アクセス ドライバーを使用していることをチェックして、パフォーマンスを最大限に高めます。
- ユーザーを認証し、データ ソースに接続する場合はプロセスを認証します。SharePoint のエンタープライズ シングル サインオン (SSO) 機能を使用して、ネットワークに接続しているデータ ソースに対して認証を行うことを検討します。
- アプリケーション定義ファイル (ADF) を作成する際のエラーを最小限に抑えるために、Office Server SDK の BDC Definition Editor を使用することを検討します。ADF を手動で編集する場合は、エラーを最小限に抑えるために、BDCMedata.xsd スキーマを Visual Studio に読み込むことを検討します。

- 必要に応じて、BDC のセキュリティ トリマーを使用して、エンティティのインスタンスをカスタムのセキュリティでトリミングすることを検討します。
- ステージング領域に過剰な負荷をかけないようにします。

ドキュメントとコンテンツのストレージ

Excel などの Office ドキュメントは、ドキュメント ライブラリに格納されます。Office デスクトップ アプリケーションを使用すると、複数のデータ ソースから取得したさまざまなデータを統合できます。SharePoint にコンテンツを格納する際には、次のガイドラインを考慮します。

- ドキュメントをドキュメント ライブラリに格納する場合は、コンテンツ タイプとその継承機能を使用して、他の一元化されたメタデータをドキュメントの種類ごとに定義します。ルート サイトに作成したコンテンツ タイプは、自動的に子サイトで使用できます。また、既存のコンテンツ タイプを基にして、新しいコンテンツ タイプを派生したり拡張したりすることができます。各コンテンツ タイプを単独で管理するのではなく、この動作を使用してコンテンツ タイプの管理を簡略化します。
- 将来必要なコンテンツ タイプを特定して計画し、そのコンテンツ タイプを使用して、一意なメタデータ フィールドの名前と関連付け、ドキュメント テンプレート、およびカスタム フォームを定義します。コンテンツ タイプをすべての子サイトで使用できるようにする必要がある場合は、サイト レベルで作成します。コンテンツ タイプを特定のリストだけで使用できるようにする必要がある場合のみ、リスト レベルで作成します。
- ドキュメントのメタデータを追跡して編集するために、ドキュメント情報パネルを使用してコンテンツ タイプのメタデータを収集することを検討します。ドキュメント情報パネルには、ビジネス ロジックやデータ検証を追加できます。
- ユーザーが構成できる参照データや永続的なデータをリストに格納することを検討します。ただし、SharePoint リストをデータベース テーブルとして扱わないようにします。一時的なデータやトランザクションのデータを格納するには、データベースを使用します。アプリケーションでリストを複数回クエリする場合は、DataTable または DataSet にリストのコンテンツをキャッシュすることを検討します。
- ファイル システムを SharePoint ドキュメント ライブラリに置き換えないようにします。また、SharePoint ドキュメント ライブラリを、ソース コード管理メカニズムや開発チーム メンバーがソース コードで共同作業するためのプラットフォームとして使用しないようにします。ドキュメント ライブラリは、共同作業と管理が必要なドキュメントを格納する目的にのみ使用します。

- ドキュメント ライブラリとリストでは、リスト コンテナーごとのアイテム数が 2,000 個に制限されていることを考慮します。リスト コンテナーのアイテム数が 2,000 個を超える場合は、独自の UI を作成してリスト内のアイテムを取得することを検討します。
- フィルター表示を使用するのではなく、ドキュメントをフォルダーに分類することを検討します。この方法を使用すると、すばやくドキュメントを取得できます。

Excel Services

Excel Services は、主に次の 3 つのコンポーネントで構成されています。まず、Excel Calculation Services では、ブックの読み込み、計算の実行、外部データの更新、およびセッションの維持を行います。次に、Excel Web Access は、ブラウザーで Excel ブックを表示してブックを操作できるようにする Web パーツです。最後に、Excel Web Services は SharePoint でホストされている Web サービスで、開発者が Excel ブックに基づくカスタム アプリケーションの構築に使用できるメソッドを提供します。Excel Services の使用法を設計する際には、次のガイドラインを考慮します。

- すべてのユーザーを認証し、Office データ接続ファイルをセキュリティで保護します。Excel Services で Kerberos 認証または SSO を構成して、他のサーバーにある SQL Server データベースに対して認証することを検討します。
- ブックを発行する前に、信頼できるファイルの場所や信頼できるデータ接続ライブラリを構成し、必要な情報だけを発行します。
- 発行前に Excel ブックを信頼できるファイルの場所に格納し、ブックの発行前に Office データ接続ファイルを信頼できるデータ接続ライブラリにアップロードします。

InfoPath Forms Services

InfoPath Forms Services では、SharePoint に格納されたテンプレートに基づいて構築され InfoPath を通じてユーザーに公開される、ブラウザー ベースのフォームを使用する機能がユーザーに提供されます。InfoPath Forms Services が実行されているサーバーにブラウザー互換テンプレート (.xsn) に基づくフォームを配置すると、Office InfoPath 2007 がインストールされていないコンピューターの Web ブラウザーでフォームを開くことができます。ただし、Office InfoPath 2007 がインストールされている場合は、Office InfoPath 2007 でフォームが開かれます。InfoPath Forms Services での InfoPath フォームの使用を設計する際には、次のガイドラインを考慮します。

- 対称フォームの作成を検討します。対称フォームとは、SharePoint Server の Web インターフェイスでも、Office system のクライアント アプリケーション (Word、Excel、PowerPoint など) でも、まったく同じように表示して操作できるフォームです。
- InfoPath の [デザイン チェック] 作業ウィンドウを使用して、ブラウザー フォームでの互換性の問題をチェックします。また、ブラウザー向けフォームを設計する場合は、サポートされていないコントロールが表示されないように [ブラウザー互換の機能のみを有効にする] チェック ボックスをオンにすることを検討します。
- フォームのパフォーマンスと応答性が向上するように、非表示コンテンツを含む単一ビューではなく、複数のビューを使用することを検討します。ただし、ビューを使用して情報を非表示にすると発生する、うわべだけのセキュリティを利用しないようにします。
- 保護を有効にして、フォーム テンプレートの整合性を保ち、ユーザーがフォーム テンプレートを変更しないようにすることを検討します。フォームをパブリック サイトに公開する場合は、サービス拒否 (DoS) 攻撃を防ぐために、スクリプトや自動プロセスからフォーム テンプレートにアクセスできないようにします。また、認証情報、サーバー名、データベース名などの機密情報がパブリック フォームに含まれないようにします。
- 大量のデータが必要なレポート作成ソリューションを設計する場合、InfoPath Forms Services の使用を避けます。
- レポートを作成する必要がある場合にフォームのデータをデータベースに送信し、フォームで収集される機密情報をデータベースに格納することを検討します。
- データ接続を柔軟に管理して再利用性を確保するために、ユニバーサル データ接続 (UDC) ファイルの使用を検討します。
- InfoPath Forms Services のセッション状態を構成する場合は、フォーム ビューの使用を検討します。

SharePoint オブジェクト モデル

SharePoint では、プロセスを自動化するコードを作成できるオブジェクト モデルが公開されます。たとえば、ドキュメントのカスタム バージョン管理を実装したり、カスタム チェックイン ポリシーを適用したりすることができます。SharePoint オブジェクト モデルを使用してカスタム コードを作成する際には、次のガイドラインを考慮します。

- 作成した SharePoint オブジェクトを使用後に破棄して、アンマネージ リソースを解放します。また、SharePoint オブジェクトを例外ハンドラーで適切に破棄します。

- 適切なキャッシュの手法を選択し、機密データや揮発性データをキャッシュしないようにします。キャッシュする必要がある場合は、データを SharePoint オブジェクトから DataSet または DataTable に読み込むことを検討します。ただし、SharePoint オブジェクトをキャッシュする場合は、スレッド同期とスレッド セーフを考慮する必要があります。
- 特権を昇格する際には、昇格後に新規作成した SharePoint オブジェクトだけで、昇格した特権が使用されることに注意してください。

Web パーツ

Web パーツでは、プレゼンテーション レベルの機能豊富な構成を提供できます。アプリケーション固有の機能を実装するカスタム Web パーツを構築することは可能ですが、SharePoint やその他の環境 (ASP.NET など) で提供される Web パーツも使用できます。Web パーツを使用すると、ネットワークに接続している LOB アプリケーションや Web サービスと通信したり、SharePoint LOB アプリケーションのパーソナル化をサポートするカスタマイズ可能な複合インターフェイスを作成できます。ASP.NET で使用できるアクセス許可以外の追加のアクセス許可を Web パーツに提供する必要がある場合は、カスタムのコード アクセス セキュリティ ポリシーの作成を検討します。Web パーツを開発する際には、次のガイドラインを考慮します。

- Web パーツに実装する適切な機能を特定し、Web パーツで操作するすべてのデータ ソースを特定します。ユーザーが個別の操作を実行できるように Web パーツ動詞を使用し、カスタム プロパティを分類して Web パーツのプロパティと区別します。
- レイヤーのガイドラインを使用して Web パーツを設計し、プレゼンテーション ロジック、ビジネス ロジック、およびデータ ロジックを分割して、保守容易性を向上します。再利用性を向上するために、各 Web パーツで 1 つの機能だけが実行されるように設計し、必要に応じてユーザーが Web パーツを構成したりカスタマイズしたりできるように設計します。
- 適切なセキュリティ対策を Web パーツに実装します。ユーザー単位のセキュリティが必要ない場合にのみ、Web パーツをグローバル アセンブリ キャッシュに配置します。
- Web パーツ領域を使用して、実行時にユーザー自身が管理できる Web パーツをホストします。ASP.NET マスター ページを使用している場合は、Web パーツ ページで使用するマスター ページに Web パーツ マネージャーを含めます。また、Web パーツに含まれているコントロールにスタイル属性を直接指定しないようにします。
- Web パーツで作成したすべての SharePoint オブジェクトとアンマネージ リソースを適切に破棄します。

ワークフロー

SharePoint では、開発者が MOSS を使用して単純なワークフローを作成し、そのワークフローを SharePoint のドキュメント ライブラリに関連付けることができます。また、ユーザーが SharePoint Designer を使用してカスタム ワークフローを作成することも、開発者が Visual Studio を使用してカスタム ワークフローを作成することもできます。ワークフローを設計する際には、次のガイドラインを考慮します。

- 自動化されるビジネス プロセスまたはビジネス プロセスの一部を特定します。ワークフローを物理的に実装する前に、既存のビジネス プロセスが正確で関連ドキュメントが作成されていることを確認します。また、その技術の専門家またはビジネス アナリストに既存のビジネス プロセスのレビューを依頼することを検討します。
- ビジネス要件を満たす適切なワークフロー テクノロジを選択します。たとえば、ドキュメント承認など、ビジネス要件が単純な場合は、組み込みの SharePoint ワークフローを使用します。組み込みのワークフローでビジネス要件を満たせない場合は、SharePoint Designer を使用してワークフローを作成することを検討します。
- ビジネス要件で複雑なワークフローが必要とされている場合や、LOB システムと統合する必要がある場合は、Visual Studio を使用してカスタム ワークフローを開発することを検討します。また、Visual Studio を使用して、SharePoint Designer に登録できるワークフロー アクティビティを作成し、インフォメーション ワーカーを支援することを検討します。
- カスタム ワークフローを作成する場合は、シナリオに適した種類のワークフローを選択します。ステート マシン ベースのモデルとシーケンシャル モデルを考慮します。また、デバッグに役立つように、コードに包括的なインストルメンテーションを実装することを検討します。
- カスタム ワークフローをデバッグする場合は、ログ レベルを詳細レベルに設定することを検討します。
- 以前のワークフローをアップグレードする場合は、ワークフロー アセンブリをバージョン管理することと、ソリューションのグローバル一意識別子 (GUID) を変更することを検討します。また、新しいバージョンを配置する際には、実行中の既存のワークフロー インスタンスへの影響を考慮します。
- エンド ユーザーが作成するワークフローについては、個別のワークフロー履歴リストとタスク リストを作成することを検討します。
- 管理容易性を向上するために、ワークフローをコンテンツ タイプに割り当てることを検討します。ワークフローをコンテンツ タイプに割り当てると、ワークフローをさまざまなコンテンツ ライブラリで使用できますが、ワークフローを 1 か所で管理する必要があります (この機能は、組み込みのワークフ

ローまたは Visual Studio のワークフローで使用できますが、SharePoint Designer のワークフローでは使用できません)。実行中のワークフロー インスタンスはリスト アイテムごとに 1 つしか存在できず、ワークフロー インスタンスを開始できるのはリスト アイテムだけで、リスト自体からは開始できないことに注意してください。

テクノロジに関する考慮事項

次のガイドラインは、SharePoint ワークフローに適した実装テクノロジを選択するのに役立ちます。また、独自の SharePoint インターフェイスを実現する Web パーツの作成に関するガイダンスを提供します。

- 安全で信頼できるトランザクションによるデータ交換がサポートされ、さまざまなトランスポートとエンコードのオプションが用意され、継続的な実行とアクティビティの追跡が組み込みで提供されるワークフローが必要な場合は、WF の使用を検討します。
- 複雑なオーケストレーションが実装され、信頼できるストア アンド フォワード メッセージング機能がサポートされるワークフローが必要な場合は、BizTalk Server の使用を検討します。
- マイクロソフト以外のシステムとのやり取り、電子データ交換 (EDI) 操作の実行、または Enterprise Service Bus (ESB) パターンの実装が必要な場合は、Microsoft BizTalk ESB Toolkit の使用を検討します。
- ビジネス レイヤーが単一の SharePoint サイトに限定され、他のサイトにある情報にアクセスする必要がない場合は、MOSS の使用を検討します。MOSS は、複数のサイトが必要なシナリオには適していません。
- アプリケーションで ASP.NET Web パーツを作成する場合は、SharePoint 2003 との下位互換性が必要ない限り、System.Web.UI.WebControls.WebParts.WebPart クラスの継承を検討します。SharePoint 2003 をサポートする必要がある場合は、Microsoft.SharePoint.WebPartPages.WebPart クラスの継承を検討します。

配置に関する考慮事項

SharePoint LOB アプリケーションは、機能の大半を提供するうえで SharePoint 自体に依存しています。ただし、コンポーネントなどの追加の成果物は、SharePoint からアクセスして使用できる方法で配置する必要があります。SharePoint LOB アプリケーションの配置に関する方針を設計する際には、次のガイドラインを考慮します。

- ファーム、Web アプリケーション、サイト コレクション、サイトなど、機能の範囲を決定します。
- 機能をソリューションとしてパッケージ化することを検討します。
- 低レベルのコード アクセス セキュリティ メカニズムを利用するために、アセンブリをグローバル アセンブリ キャッシュではなく BIN フォルダーに配置することを検討します。
- ソリューションを配置したら、管理者以外のアカウントを使用してテストします。

関連する設計パターン

次の表に、主要なパターンを示します。SharePoint LOB アプリケーションの設計を行う際には、これらのパターンの使用を検討します。

カテゴリ	関連するパターン
ワークフロー	**Data-Driven Workflow**: ワークフローまたはシステムのデータの値に基づいてシーケンスが決定されるタスクを含むワークフローです。 **Human Workflow**: ユーザーが手動で実行するタスクを含むワークフローです。 **Sequential Workflow**: 先行するタスクが完了してから次のタスクが開始されるというシーケンスに従うタスクを含むワークフローです。 **State-Driven Workflow**: システムの状態によってシーケンスが決定されるタスクを含むワークフローです。

Data-Driven Workflow、Human Workflow、Sequential Workflow、および State-Driven Workflow の各パターンの詳細については、次のリソースを参照してください。

- Windows Workflow Foundation の概要 (http://msdn.microsoft.com/ja-jp/library/ms734631.aspx)
- Workflow Patterns (http://www.workflowpatterns.com/、英語)

関連情報

MOSS と WSS を使用した SharePoint LOB アプリケーションの構築に関する Web リソースに、より簡単にアクセスするには、http://www.microsoft.com/architectureguide (英語) でオンライン版の参考文献を参照してください。

- SharePoint Server 2007 と Windows Workflow Foundation を使用してワークフロー ソリューションを開発する (http://msdn.microsoft.com/ja-jp/library/cc514224.aspx)
- ベスト プラクティス: SharePoint オブジェクト モデル使用時のコーディング上の一般的な問題 (http://msdn.microsoft.com/ja-jp/library/bb687949.aspx)
- ベスト プラクティス: Windows SharePoint Services の破棄可能なオブジェクトを使用する (http://msdn.microsoft.com/ja-jp/library/aa973248.aspx)
- InfoPath Forms Services のベスト プラクティス (http://technet.microsoft.com/ja-jp/library/cc261832.aspx)
- ホワイト ペーパー: Office SharePoint® Server で大きなリストを操作する (http://technet.microsoft.com/ja-jp/library/cc262813.aspx)

付録

このセクションは、マイクロソフト アプリケーション プラットフォームの概要、プレゼンテーション、データ アクセス、統合に関するテクノロジの一覧表など、開発するアプリケーションのシナリオに適したテクノロジを評価して選ぶのに役立つ付録トピックで構成されています。一般的な横断的関心事に対応することで、アプリケーションの設計と作成にかかる時間を短縮するのに役立つ patterns & practices の Enterprise Library の特徴についても取り上げます。また、patterns & practices のパターン カタログも紹介しています。このカタログではアーキテクチャの設計時に役立つ主要なパターンの情報を一覧形式で提供しています。

詳細については、次の付録を参照してください。

- 付録 A「マイクロソフト アプリケーション プラットフォーム」
- 付録 B「プレゼンテーションテクノロジ」
- 付録 C「データ アクセス テクノロジ」
- 付録 D「統合テクノロジ」
- 付録 E「ワークフロー テクノロジ」
- 付録 F「patterns & practices の Enterprise Library」
- 付録 G「patterns & practices パターン カタログ」

付録 A

マイクロソフト アプリケーション プラットフォーム

概要

この付録では、まず、マイクロソフト プラットフォームの機能の概要について説明します。この情報は、.NET Framework とマイクロソフト サーバー テクノロジを対象とするアプリケーションの作成に慣れていない場合に、適切なテクノロジを特定するのに役立ちます。次の「情報とリソースの検索」セクションは、広大なマイクロソフト Web サイトを構成している多数のサイトにアクセスする際に、情報をより簡単に見つけるのに役立つ情報を提供しています。

次に、.NET Framework と共通言語ランタイム (CLR) の概要を説明してから、さまざまなアプリケーションの種類、コラボレーション、統合、データ アクセス、およびワークフローで使用できる、さまざまなマイクロソフト アプリケーション プラットフォーム テクノロジについて説明します。その後、SQL Server、IIS (Web サーバー) などの製品のテクノロジ、Visual Studio などの開発ツール、およびサードパーティ製のライブラリの概要を紹介します。

マイクロソフト アプリケーション プラットフォームは、次表の製品、インフラストラクチャ コンポーネント、ランタイム サービス、および .NET Framework で構成されています。

カテゴリ	テクノロジ
アプリケーション インフラストラクチャ	共通言語ランタイム (CLR) .NET Framework
コラボレーション、統合、ワークフロー	Windows Workflow Foundation (WF) Microsoft Office SharePoint Server (MOSS)

	Microsoft BizTalk Server
データ アクセス	ADO.NET Core ADO.NET Data Services フレームワーク ADO.NET Entity Framework Sync Services for ADO.NET 統合言語クエリ (LINQ)
データベース サーバー	Microsoft SQL Server
開発ツール	Microsoft Visual Studio Microsoft Expression® Studio デザイン ソフトウェア
モバイル	.NET Compact Framework ASP.NET for Mobile Silverlight for Mobile
リッチ クライアント	Windows フォーム Windows Presentation Foundation (WPF)
リッチ インターネット アプリケーション (RIA)	Microsoft Silverlight
サービス	ASP.NET Web サービス (ASMX) Windows Communication Foundation (WCF)
Web	ASP.NET
Web サーバー	インターネット インフォメーション サービス (IIS)

情報とリソースの検索

このガイドでは、アーキテクチャ設計のための指針を紹介し、.NET Framework とマイクロソフト アプリケーション プラットフォームで実行するアプリケーションを設計する際に考慮する必要がある事項について、ベスト プラクティスを説明します。このガイドでは、多くの一般的なシナリオについて順を追って説明したトピックなど、豊富な情報を提供していますが、すべてのトピックを取り上げることはできません。マイクロソフトでは、次のセクションに示すように、すべてのテクノロジ、製品、およびサービスに関する広範かつ詳細なライブラリを管理、提供しています。

マイクロソフトが Web 上で技術情報を分類している方法

広範囲にわたるマイクロソフトの技術ドキュメントを閲覧し始めたばかりの場合は、いくつか知っておくべきことがあります。これは、探している情報をよりすばやく見つけるのに役立ちます。

まず、マイクロソフトにはいくつかの大きな Web サイトがあり、各サイトにはさまざまな役割があります。まず、マイクロソフトの企業 Web サイト (http://www.microsoft.com/japan) では、膨大な量の技術的なマーケティング情報を含む、すべてのマイクロソフト製品のマーケティング情報を提供しています。たとえば、サービス指向アーキテクチャ (SOA) などのトピックに関するマイクロソフトの公式見解を知りたい場合は、microsoft.com/japan にアクセスして、SOA を検索することで、その情報を見つけることができます。一般的に、microsoft.com/japan では、企業イメージの提示と製品のメリットに関する説明に重点を置いた情報を提供し、技術情報は他の専用 Web サイトで提供するようにしています。

技術コンテンツは他の場所で提供されていますが、そのページへのリンクは microsoft.com/japan ホーム ページで簡単に見つけることができます。ホーム ページには、"おすすめ情報"、"個人で利用される方へ"、"法人、企業で利用される方へ" などのコンテンツ カテゴリと、"IT 技術者/SE （IT Pro） の方へ" および "開発者の方へ" という 2 つの技術コンテンツ カテゴリがあります。マイクロソフトでは、ソフトウェア開発者と、その他のすべての IT プロフェッショナルを明確に区別しています。これは主に、開発者向けのコンテンツが非常に多いことと、開発者が必要とする情報の種類が、ネットワーク管理者や運用担当者が必要とするものとは大幅に異なっているからです。これらのカテゴリに含まれるリンクの多くは頻繁に変更されますが、技術情報の主要な Web サイトが変更されることはありません。開発者向けの技術情報は、Microsoft Developer Network (MSDN) サイト (http://msdn.microsoft.com/ja-jp/) で提供されています。また、その他のすべての IT プロフェッショナル向け技術情報の主要なサイトは、Microsoft TechNet (http://technet.microsoft.com/ja-jp/) です。

Microsoft Developer Network

MSDN は、デベロッパー センター (一般的に、主要な開発ツール、言語、テクノロジ、および技術的な分野ごとに用意されています)、ライブラリ (検索可能なコンテンツの大きなリポジトリ)、ダウンロード、サポート、フォーラム、およびコミュニティ (他のユーザーの意見や知識を閲覧したり、コミュニティ プログラムに参加したりできる場所) という、いくつかの領域に分かれています。

また、これ以外にも重要かつ特殊なサブサイトがいくつかあります。Channel 9 (http://channel9.msdn.com、英語) では、一般的に、マイクロソフト製品グループのソフトウェア エンジニアやアーキテクトが現在取り組んでいるテクノロジについて説明したり、開発ツールやテクノロジの今後の計画について議論したりしている非公式なビデオが公開されています。CodePlex (http://codeplex.com、英語) は、マイクロソフトのオープン ソース プロジェ

クトのホスティング サイトです。このサイトでは、パブリック コミュニティで他のユーザーが取り組んでいるプロジェクトを参照したり、自分のプロジェクトを開始したりすることもできます。マイクロソフトの patterns & practices チームは、CodePlex のパブリック コミュニティですべてのサービスを開発し、コミュニティ全体から、開発サイクル全体を通じてプロジェクトへのフィードバックを受け付けています。
MSDN では、Webcast などの技術に関するチャットやイベントも開催しています。これは、新しい技術分野や新しいテクノロジについて学習するうえで非常に有益です。

Microsoft TechNet

同様に、Microsoft TechNet でも同じような情報と機会が提供されています。当然のことながら、TechNet には、デベロッパー センターではなく TechCenter があります。それ以外では、TechNet では、ネットワーク インフラストラクチャの設計、開発、運用、およびマイクロソフト製品のインストールと管理に関するガイダンスなどのトピックに関する技術的なコンテンツが提供されている点が大きく異なります。

.NET Framework

大まかに言うと、.NET Framework は、仮想ランタイム エンジン、クラス ライブラリ、および .NET アプリケーションの開発と実行に使用するランタイム サービスで構成されています。当初、.NET Framework は、ランタイム エンジンとアプリケーションの開発に使用する主要なクラス セットとしてリリースされました。
基本クラス ライブラリ (BCL) で提供される主要なクラス セットは、さまざまな領域 (UI、データ アクセス、データベース接続、暗号化、数値アルゴリズム、ネットワーク通信など) の多種多様なプログラミング要件に対応しています。
BCL と重複するのは、.NET アプリケーション開発のコア テクノロジです。このテクノロジには、アプリケーションの機能 (リッチ クライアントやデータ アクセスなど) でグループ化されるクラス ライブラリとランタイム サービスが含まれます。マイクロソフトの .NET プラットフォームが発展するにつれて、コア テクノロジには、WCF、WPF、WF などの新しいテクノロジが追加されました。

共通言語ランタイム

.NET Framework には、プログラムのランタイム要件を管理する仮想環境が用意されています。CLR と呼ばれるこの環境では、仮想マシンの環境が提供されるので、プログラマは、プログラムを実行する特定の CPU や他のハードウェアの性能について考慮する必要がありません。CLR 内で実行されるアプリケーションは、"マネージ アプリケ

ーション" と呼ばれます。.NET Framework アプリケーションは、マネージ コード (CLR 内で実行されるコード) を使用して開発されますが、いくつかの機能 (カーネル API を使用する必要があるデバイス ドライバーなど) はアンマネージ コードを使用して開発される場合がよくあります。CLR では、セキュリティ、メモリ管理、例外処理などのサービスも提供されます。

データ アクセス

マイクロソフト プラットフォームで使用できるデータ アクセス テクノロジは次のとおりです。

- **ADO.NET Core**: ADO.NET Core では、一般的なデータの取得、更新、および管理を行うための機能を提供します。SQL Server、OLE DB、Open Database Connectivity (ODBC)、SQL Server Compact Edition、および Oracle の各データベースのプロバイダーが含まれます。
- **ADO.NET Data Services フレームワーク**: このフレームワークでは、HTTP 経由でアクセスされる RESTful Web サービスを通じて、LINQ に対応したすべてのデータ ソース (一般的には、エンティティ データ モデル) のデータが公開されます。このデータは、Uniform Resource Identifier (URI) を使用して直接アドレス指定できます。データをシンプルな Atom 形式および JavaScript Object Notation (JSON) 形式で返すように、Web サービスを構成できます。
- **ADO.NET Entity Framework**: このフレームワークでは、リレーショナル データベースに関して、厳密に型指定されたデータ アクセス エクスペリエンスが提供されます。このフレームワークでは、データ モデルを、リレーショナル テーブルの物理的な構造から、一般的なビジネス オブジェクトを正確に反映した概念モデルに移行します。Entity Framework では、ADO.NET 環境内に一般的なエンティティ データ モデルが導入されているので、開発者はリレーショナル データへの柔軟なマッピングを定義できます。このマッピングは、基盤となるストレージ スキーマの変更からアプリケーションを分離するのに役立ちます。また、Entity Framework では、LINQ to Entities もサポートされているので、Entity Framework を通じて公開されるビジネス オブジェクトに LINQ のサポートが提供されます。O/R(オブジェクト/リレーショナル マッピング)ツール (O/RM ツール)として使用すると、開発者は LINQ to Entities をビジネス オブジェクトに対して使用します。その結果、ビジネス オブジェクトが、Entity Framework で管理されるエンティティ データ モデルにマップされる Entity SQL に変換されます。開発者は、エンティティ データ モデルを直接操作したり、アプリケーションで Entity SQL を使用したりすることもできます。
- **Sync Services for ADO.NET**: Sync Services for ADO.NET は、Microsoft Sync Framework に含まれているプロバイダーです。ADO.NET 対応のデータベースに同期機能を実装するために使用されま

す。このプロバイダーを使用すると、不定期に接続するアプリケーションにデータを同期する機能を組み込むことができます。このプロバイダーでは、クライアント データベースから情報を定期的に収集して、この情報をサーバー データベースと同期します。

- **統合言語クエリ (LINQ)**: LINQ では、クエリのネイティブな言語構文で C# や Visual Basic を拡張するクラス ライブラリが提供されます。これは基本的には、.NET Framework 全体でさまざまなアセンブリによってサポートされるクエリ テクノロジです。たとえば、LINQ to Entities は ADO.NET Entity Framework アセンブリに、LINQ to XML は System.Xml アセンブリに、LINQ to Objects は .NET Framework の中核となるシステム アセンブリに含まれます。クエリは、さまざまな形式のデータに対して実行できます。たとえば、DataSet (LINQ to DataSet)、XML (LINQ to XML)、メモリ内のオブジェクト (LINQ to Objects)、ADO.NET Data Services (LINQ to Data Services)、リレーショナル データ (LINQ to Entities) などです。
- **LINQ to SQL**: LINQ to SQL では、SQL Server に対する軽量で厳密に型指定されたクエリ ソリューションを提供します。LINQ to SQL は、オブジェクトを簡単かつ迅速に永続化するシナリオを実現するために設計されています (このシナリオでは、中間ティアのクラスが、データベースのテーブル構造に緊密にマップされます)。.NET Framework 4 以降では、LINQ to SQL のシナリオが ADO.NET Entity Framework によって統合およびサポートされます。ただし、LINQ to SQL のテクノロジは引き続きサポートされます。詳細については、ADO.NET チームのブログ (http://blogs.msdn.com/adonet/archive/2008/10/31/clarifying-the-message-on-l2s-futures.aspx、英語) を参照してください。

モバイル アプリケーション

.NET プラットフォームでは、モバイル アプリケーション向けに次のテクノロジを提供しています。

- **Microsoft .NET Compact Framework**: モバイル デバイス専用に設計された Microsoft .NET Framework のサブセットです。デバイスでスタンドアロン アプリケーションや不定期に接続するアプリケーションとして実行する必要があるモバイル アプリケーションには、このテクノロジを使用します。
- **ASP.NET for Mobile**: モバイル デバイス専用に設計された、ASP.NET のサブセットです。ASP.NET for Mobile アプリケーションは、標準的な IIS Web サーバーでホストできます。多数のモバイル デバ

イスとブラウザーをサポートする必要があり、常時ネットワーク接続を利用できるモバイル Web アプリケーションには、このテクノロジを使用します。

- **Silverlight for Mobile**: この Silverlight クライアントのサブセットでは、Silverlight プラグインがモバイル デバイスにインストールされている必要があります。既存の Silverlight アプリケーションをモバイル デバイスに移植するために、または他のテクノロジを使用して作成できる UI よりも機能が豊富な UI を作成する必要がある場合は、このテクノロジを使用します。

リッチ クライアント

Windows ベースのアプリケーションは、.NET Framework で実行されます。.NET Framework では、リッチ クライアント アプリケーション向けに次のテクノロジを提供しています。

- **Windows フォーム**: .NET Framework の標準的な UI 設計テクノロジです。WPF を使用できる場合でも、Windows フォームの技術的専門知識があるときや、アプリケーションに多数のグラフィックスを使用する UI やストリーミング メディアの UI に関する要件がない場合は、Windows フォームは UI 設計に最適です。
- **Windows Presentation Foundation (WPF) アプリケーション**: WPF アプリケーションでは、より高度なグラフィック機能がサポートされます。たとえば、2D と 3D のグラフィックス、画面解像度への依存からの解放、高度なドキュメントと文字体裁のサポート、タイムラインを含むアニメーション、ストリーミング配信されるオーディオとビデオ、ベクター ベースのグラフィックなどです。WPF では、Extensible Application Markup Language (XAML) を使用して、UI、データ バインド、およびイベント定義を実装します。また、高度なデータ バインド機能とテンプレート機能も用意されています。WPF アプリケーションは、デスクトップに配置したり、XAML ブラウザー アプリケーション (XBAP) を使用してブラウザー内に配置したりできます。WPF アプリケーションでは、開発者とデザイナーの連携がサポートされます。開発者はビジネス ロジックに集中し、デザイナーは外観と動作を制御できます。
- **WPF ユーザー コントロールを使用する Windows フォーム**: WPF コントロールで提供される、より強力な UI 機能を使用できるようになります。また、WPF を既存の Windows フォーム アプリケーションに追加できます。ただし、WPF コントロールが最適な状態で動作するには、高性能なクライアント コンピューターが必要になる場合が多いことに注意してください。

- **Windows フォーム ユーザー コントロールを使用する WPF**: このテクノロジを使用すると、WPF で提供されないコントロールによって、WPF を補完できます。WindowsFormsIntegration アセンブリで提供される WindowsFormsHost コントロールを使用して、Windows フォーム コントロールを追加できます。ただし、重複するコントロール、インターフェイスのフォーカス、およびさまざまなテクノロジで使用されるレンダリング技法に関する制約と不整合があります。
- **WPF を使用する XAML ブラウザー アプリケーション (XBAP)**: このテクノロジでは、Windows で実行されている Microsoft Internet Explorer または Mozilla Firefox で、サンドボックス化された WPF アプリケーションをホストします。Silverlight とは異なり、WPF フレームワークの大部分を使用できますが、部分信頼されたサンドボックスからシステム リソースへのアクセスには制限があります。XBAP を利用するには、クライアント デスクトップに Windows Vista がインストールされているか、.NET Framework 3.5 と XBAP ブラウザー プラグインの両方がインストールされている必要があります。XBAP が適しているのは、必要な機能が Silverlight で提供されていない場合と、クライアント プラットフォームおよび信頼の要件を指定できる場合です。

リッチ インターネット アプリケーション

マイクロソフト アプリケーション プラットフォームには、リッチ インターネット アプリケーション (RIA) を作成するための Silverlight テクノロジが含まれています。RIA は、Windows Server インターネット インフォメーション サービス (IIS) などの Web サーバーでホストする必要があります。RIA の開発に使用できるテクノロジは次のとおりです。

- **Silverlight**: さまざまなプラットフォームやブラウザーで動作する、ブラウザー用に最適化された WPF のサブセットです。XBAP に比べて、Silverlight は軽量で短時間にインストールできますが、3D グラフィックスとテキスト フローを使用するドキュメントはサポートされません。Silverlight は、リソースの使用量が少なく、さまざまなプラットフォームでサポートされるので、高品質な WPF グラフィックのサポートが不要な WPF アプリケーションに適しています。
- **AJAX を使用する Silverlight**: Silverlight は、Asynchronous JavaScript and XML (AJAX) をネイティブにサポートしており、オブジェクト モデルを Web ページに含まれる JavaScript に提供します。この機能を使用すると、ページ コンポーネントとサーバー間のバックグラウンド操作が可能になり、より応答性の高い UI を提供できます。

サービス

.NET プラットフォームでは、サービス ベースのアプリケーションを作成するために、次のテクノロジを提供しています。

- **Windows Communication Foundation (WCF)**: WCF は、分散コンピューティングの管理しやすい手法と幅広い相互運用性を提供するように設計されており、サービス指向の直接的なサポートが含まれています。HTTP、TCP、Microsoft Message Queuing、名前付きパイプなどのさまざまなプロトコルがサポートされています。
- **ASP.NET Web サービス (ASMX)**: ASMX では、分散コンピューティングのより簡単な手法と相互運用性が提供されますが、HTTP プロトコルしかサポートされていません。

ワークフロー

.NET プラットフォームでは、ワークフローを実装するために、次のテクノロジを提供しています。

- **Windows Workflow Foundation (WF)**: ワークフローの実装を可能にする基本的なテクノロジです。シーケンシャル ワークフローやステート マシン ベースのワークフローを作成する必要がある開発者や独立系ソフトウェア ベンダー (ISV) 向けのツールキットとして、WF では、シーケンシャル、ステート マシン、データ ドリブン、およびカスタムの各ワークフローの種類をサポートしています。Visual Studio の Windows ワークフロー デザイナーを使用してワークフローを作成できます。
- **ワークフロー サービス**: WCF と WF を統合することで、ワークフロー向けの WCF ベースのサービスを提供します。Microsoft .NET Framework 3.5 以降、WCF は、サービスとして公開されるワークフローをサポートし、ワークフロー内からサービスを呼び出す機能を提供するように拡張されました。また、Visual Studio 2008 には、ワークフロー サービスをサポートする新しいテンプレートとツールが用意されています。
- **Microsoft Office SharePoint Services (MOSS)**: WF に基づくワークフローのサポートを提供するコンテンツ管理とコラボレーションのプラットフォームで、SharePoint サーバーに関連するヒューマン ワークフローとコラボレーション向けのソリューションを提供します。MOSS のインターフェイスで、ドキュメント承認のためのワークフローを直接作成できます。また、SharePoint Designer または Visual Studio の Windows ワークフロー デザイナーを使用してワークフローを作成することもできます。ワークフローをカスタマイズするには、Visual Studio で WF オブジェクト モデルを使用できます。

- **Microsoft BizTalk Server**: 現在、BizTalk Server には、オーケストレーション (システム レベルのワークフローをエンタープライズ規模で統合するなど) を対象とした独自のワークフロー エンジンが用意されています。今後リリースされる BizTalk Server のバージョンでは、WF と XLANG (サービスのオーケストレーションとコラボレーションをモデル化するために使用される Web サービス定義言語の拡張) が使用される可能性があります。XLANG は、BizTalk Server で現在使用されている既存のオーケストレーション テクノロジです。BizTalk Orchestration Services を使用して、アプリケーション内およびアプリケーション間で疎結合された、実行時間の長いビジネス プロセスの全体的な設計とフローを定義できます。

MOSS と BizTalk Server は、.NET Framework や Visual Studio に含まれているテクノロジではありません。これらは、独立した製品ですが、マイクロソフト プラットフォームの構成要素です。

Web アプリケーション

.NET プラットフォームには、Web アプリケーションと単純な Web サービスを作成するために、ASP.NET が用意されています。ASP.NET アプリケーションは、IIS などの Web サーバーでホストする必要があります。ASP.NET を使用して Web アプリケーションを作成する際に、使用できるテクノロジは次のとおりです。

- **ASP.NET の Web フォーム**: これは、.NET Web アプリケーション向けの UI を設計および実装する標準的なテクノロジです。ASP.NET の Web フォーム アプリケーションは Web サーバーにのみインストールする必要があり、クライアント デスクトップにコンポーネントをインストールする必要はありません。
- **AJAX を使用する ASP.NET の Web フォーム**: サーバーとクライアント間の要求を非同期に処理して応答性を向上し、クライアントに対してより優れたエクスペリエンスを提供し、サーバーへのポストバック数を削減するために、AJAX と ASP.NET の Web フォームを併用します。AJAX は、.NET Framework 3.5 以降で ASP.NET に不可欠な要素です。
- **Silverlight を使用する ASP.NET の Web フォーム**: 既存の ASP.NET アプリケーションがあれば、Silverlight コントロールを使用してユーザー エクスペリエンスを向上し、Silverlight アプリケーションをゼロから作成することを回避できます。これは、既存のアプリケーションに Silverlight コンテンツを追加するのに適した手法です。
- **ASP.NET の MVC**: このテクノロジを使用すると、ASP.NET を使用して Model-View-Controller (MVC) パターンに基づくアプリケーションを構築できます。ASP.NET の MVC では、テスト駆動開発

がサポートされ、UI 処理と UI レンダリングの間における懸念事項が明確に分離されます。この手法は、プレゼンテーション情報とロジック コードが混在しないようにするのに役立ちます。

- **ASP.NET の Dynamic Data**: このテクノロジを使用すると、LINQ to Entities 機能を利用するデータ ドリブン ASP.NET アプリケーションを作成できます。単純なスキャフォールディングと完全なカスタマイズ機能の両方がサポートされている、基幹業務 (LOB) スタイルのデータ ドリブン アプリケーションの迅速な開発モデルが提供されます。

Web サーバー (インターネット インフォメーション サービス)

マイクロソフト プラットフォームには IIS が含まれています。IIS では、トランスポート サービス、クライアント アプリケーション、管理ツール、データベースとアプリケーションの接続、暗号化された通信など、インターネットへの発行の完全なサポートが提供されます。IIS では、次のサービスがサポートされます。

- **World Wide Web サービス**: ハイパーテキスト ドキュメントの発行と、HTTP を使用するその他の種類のコンテンツの配信に必要なすべての機能が提供されます。高度なパフォーマンス、圧縮、幅広い構成が提供され、さまざまなセキュリティと認証のオプションがサポートされます。
- **ファイル転送プロトコル (FTP) サービス**: このサービスでは、FTP を使用してファイルを受信および配信できます。ただし、使用できるのは基本認証のみです。
- **Gopher サービス**: このサービスでは、分散しているドキュメントの検索とネットワーク プロトコルの取得がサポートされます。現在ではほとんど使用されていません。
- **インターネット データベース コネクタ**: World Wide Web サービスで Open Database Connectivity (ODBC) データベースにアクセスするための、統合されたゲートウェイとテンプレートのスクリプト メカニズムです。一般的に、ASP.NET や ADO.NET Data Services などの新しいデータ アクセス テクノロジとスクリプト テクノロジに取って代わられました。
- **Secure Sockets Layer (SSL) クライアントとサーバー**: HTTP 経由の暗号化された通信をサポートするメカニズムが提供されるので、クライアントとサーバーは、コンテンツをプレーン テキスト形式で送信する場合よりも安全に通信できます。
- **インターネット サービス マネージャー サーバー**: IIS のローカルおよびリモート管理機能を提供する、管理コンソールと関連するツールです。

- **ASP.NET との統合**: IIS 7.0 以降は、ASP.NET と緊密に統合するように設計されています。この設計により、ASP.NET を使用してコンテンツを作成して配信するときに、パフォーマンスを最大限に高め、サーバーの負荷を最小限に抑えることができます。

データベース サーバー (SQL Server)

リレーショナル データベースは、エンタープライズ アプリケーションでデータを格納してアクセスするための一般的な手法です。マイクロソフト アプリケーション プラットフォームでは、アプリケーションのデータベース エンジンとして SQL Server を提供しています。SQL Server は、単一インスタンスのローカル データベース (SQL Server Express) 規模から、SQL Server Enterprise Edition を使用するエンタープライズ レベルのアプリケーションまで、さまざまな形で使用できます。
.NET Framework に含まれているデータ アクセス テクノロジでは、すべてのバージョンの SQL Server にアクセスできるので、より強力なバージョンに拡張する場合でもアプリケーションを変更する必要はありません。

Visual Studio 開発環境

.NET プラットフォームには、Visual Studio と呼ばれる包括的な開発環境が用意されています。Microsoft Visual Studio は、.NET アプリケーションの主要な開発環境であり、アプリケーション開発のライフサイクル全体に携わる特定のグループを対象とする、さまざまなバージョンを提供しています。
Visual Studio では、お好みの言語を使用して、.NET Framework を対象とするアプリケーションを作成できます。これは統合開発環境 (IDE)であり、リッチ クライアント、RIA、Web、モバイル、サービス、および Office ベースのソリューションを設計、開発、デバッグ、および配置するのに必要なすべてのツールを提供します。複数のバージョンを同時にインストールして、必要な機能を組み合わせて使用することができます。

その他のツールとライブラリ

Visual Studio に加えて、その他のツールやフレームワークを使用することで、開発にかかる時間を短縮したり、運用管理に役立てたりできます。たとえば、次のようなものがあります。

- **System Center**: エンタープライズ レベルでアプリケーションの監視、配置、構成、および管理を行うための一連のツールと環境を提供します。詳細については、「Microsoft System Center」(http://www.microsoft.com/japan/systemcenter/default.mspx) を参照してください。
- **Expression Studio**: グラフィック デザイナーが豊富なインターフェイスやアニメーションを作成することを目的とするツールを提供します。詳細については、「Microsoft Expression」(http://www.microsoft.com/japan/products/expression/products/studio_overview.aspx) を参照してください。

patterns & practices のソリューションの資産

マイクロソフトの patterns & practices グループが提供しているソリューションの資産の詳細については、次のリソースを参照してください。

- **Composite Client Application Guidance for WPF** (デスクトップと Silverlight の両方を対象): モジュール形式のアプリケーションの作成を簡略化します。詳細については、「Composite Client Application Guidance」(http://msdn.microsoft.com/en-us/library/cc707819.aspx、英語) を参照してください。
- **Enterprise Library**: 横断的関心事に対処する一連のアプリケーション ブロックを提供しています。詳細については、「Enterprise Library」(http://msdn.microsoft.com/en-us/library/cc467894.aspx、英語) を参照してください。
- **ソフトウェア ファクトリ**: スマート クライアント、WPF アプリケーション、Web サービスなど、特定の種類のアプリケーションを迅速に開発できます。詳細については、「patterns & practices: Catalog by Application Type」(http://msdn.microsoft.com/en-us/practices/bb969054.aspx、英語) を参照してください。
- **Unity Application Block** (エンタープライズ シナリオと Silverlight シナリオの両方を対象): 依存関係の挿入、サービスの場所、および制御の反転を実装する機能を提供します。詳細については、「Unity Application Block」(http://msdn.microsoft.com/en-us/library/dd203101.aspx、英語) を参照してください。
- **詳細なガイダンス**: エンタープライズ アーキテクチャ、設計、開発、および配置のさまざまなシナリオを提供しています。ソリューション開発の基礎、クライアント開発、サーバー開発、サービス開発など

のシナリオがあります。詳細については、patterns & practices のホーム ページ (http://msdn.microsoft.com/en-us/library/ms998572.aspx、英語) を参照してください。

関連情報

Web リソースに簡単にアクセスするには、http://www.microsoft.com/architectureguide (英語) でオンライン版の参考文献を参照してください。

.NET Framework の詳細については、以下のリソースを参照してください。

- .NET Framework の概要 (http://msdn.microsoft.com/ja-jp/library/a4t23ktk.aspx)
- Overview of the .NET Framework (http://msdn.microsoft.com/en-us/library/zw4w595w(VS.71).aspx、英語)
- .NET Compact Framework の概要 (http://msdn.microsoft.com/ja-jp/library/w6ah6cw1(VS.80).aspx)

Web サービスの詳細については、以下のリソースを参照してください。

- Windows Communication Foundation (http://msdn.microsoft.com/ja-jp/library/ms735119.aspx)
- ASP.NET を使用した XML Web サービス (http://msdn.microsoft.com/ja-jp/library/ba0z6a33.aspx)

ワークフロー サービスの詳細については、以下のリソースを参照してください。

- Microsoft BizTalk ESB Toolkit (http://msdn.microsoft.com/en-us/library/dd897973.aspx、英語)
- Office SharePoint Server 2007 でのワークフロー (http://msdn.microsoft.com/ja-jp/library/ms549489(office.12).aspx)
- Windows Workflow Foundation (http://msdn.microsoft.com/ja-jp/netframework/aa663328.aspx)

マイクロソフト プラットフォームのその他の機能の詳細については、以下のリソースを参照してください。

- データ アクセスの詳細については、「データ プラットフォーム (データ アクセス) 開発」 (http://msdn.microsoft.com/ja-jp/data/default.aspx) を参照してください。

- IIS Web サーバーの詳細については、「A High-Level Look at Microsoft Internet Information Server」(http://msdn.microsoft.com/en-us/library/ms993571.aspx、英語) を参照してください。
- SQL Server の詳細については、「SQL Server」(http://msdn.microsoft.com/ja-jp/sqlserver/default.aspx) を参照してください。
- Visual Studio の詳細については、「Visual Studio ホームページ」(http://www.microsoft.com/japan/visualstudio/default.mspx) を参照してください。

付録 B

プレゼンテーション テクノロジ

概要

この付録は、プレゼンテーション テクノロジを選択する際に必要となるトレードオフについて理解するのに役立ちます。また、特定のテクノロジによる設計への影響を把握したり、シナリオやアプリケーションの種類に基づいて使用するプレゼンテーション テクノロジを選択したりする際にも役立ちます。

使用するプレゼンテーション テクノロジは、開発しているアプリケーションの種類と、提供する予定のユーザー エクスペリエンスの種類によって異なります。「プレゼンテーション テクノロジの概要」を参照して、各アプリケーションの種類で使用できるテクノロジを理解できます。また、「メリットと考慮事項」を参照すると、各プレゼンテーション テクノロジの利点や考慮事項に基づいて、プレゼンテーション テクノロジを適切に選択できます。「一般的なシナリオとソリューション」では、アプリケーションのシナリオを、一般的なプレゼンテーション テクノロジのソリューションにマップするのに必要な情報を提供しています。

プレゼンテーション テクノロジの概要

これ以降のセクションでは、4 つの基本的なアプリケーションの原型 (モバイル アプリケーション、リッチ クライアント アプリケーション、リッチ インターネット アプリケーション (RIA)、および Web アプリケーション) で使用できるマイクロソフト テクノロジについて説明します。

モバイル アプリケーション

次のプレゼンテーション テクノロジは、モバイル アプリケーションで使用するのに適しています。

- **Microsoft .NET Compact Framework**: モバイル デバイス専用に設計された Microsoft .NET Framework のサブセットです。デバイスでスタンドアロン アプリケーションや不定期に接続するアプリケーションとして実行する必要があるモバイル アプリケーションには、このテクノロジを使用します。
- **ASP.NET for Mobile**: モバイル デバイス専用に設計された、ASP.NET のサブセットです。ASP.NET for Mobile アプリケーションは、標準的な Web サーバーでホストできます。多数のモバイル デバイスとブラウザーをサポートする必要があり、ネットワーク接続を常時利用できるモバイル Web アプリケーションには、このテクノロジを使用します。
- **Microsoft Silverlight for Mobile**: この Silverlight クライアントのサブセットでは、Silverlight プラグインがモバイル デバイスにインストールされている必要があります。既存の Silverlight アプリケーションをモバイル デバイスに移植するために、または他のテクノロジを使用して作成できるユーザー インターフェイス (UI) よりも機能が豊富な UI を作成する必要がある場合は、このテクノロジを使用します。

リッチ クライアント アプリケーション

次のプレゼンテーション テクノロジは、リッチ クライアント アプリケーションで使用するのに適しています。

- **Windows フォーム**: .NET Framework の標準的な UI 設計テクノロジです。WPF を使用できる場合でも、Windows フォームの技術的専門知識があるときや、アプリケーションに多数のグラフィックスを使用する UI やストリーミング メディアの UI に関する要件がないときは、Windows フォームは UI 設計に最適です。
- **Windows Presentation Foundation (WPF) アプリケーション**: WPF アプリケーションでは、より高度なグラフィック機能がサポートされます。たとえば、2D と 3D のグラフィックス、画面解像度への依存からの解放、高度なドキュメントと文字体裁のサポート、タイムラインを含むアニメーション、ストリーミング配信されるオーディオとビデオ、ベクター ベースのグラフィックなどです。WPF では、Extensible Application Markup Language (XAML) を使用して、UI、データ バインド、およびイベント定義を実装します。また、高度なデータ バインド機能とテンプレート機能も用意されています。WPF アプリケーションは、デスクトップに配置したり、XAML ブラウザー アプリケーション (XBAP) を使用してブラウザー内に配置したりできます。WPF アプリケーションでは、開発者とデザイナーの連携がサポートされます。開発者はビジネス ロジックに集中し、デザイナーは外観と動作を制御できます。

- **WPF ユーザー コントロールを使用する Windows フォーム**: WPF コントロールで提供される、より強力な UI 機能を使用できるようになります。また、WPF を既存の Windows フォーム アプリケーションに追加できます。ただし、WPF コントロールが最適な状態で動作するには、高性能なクライアント コンピューターが必要になる場合が多いことに注意してください。
- **Windows フォーム ユーザー コントロールを使用する WPF**: このテクノロジを使用すると、WPF で提供されないコントロールによって、WPF を補完できます。WindowsFormsIntegration アセンブリで提供される WindowsFormsHost コントロールを使用して、Windows フォーム コントロールを追加できます。ただし、重複するコントロール、インターフェイスのフォーカス、およびさまざまなテクノロジで使用されるレンダリング技法に関する制約と不整合があります。
- **WPF を使用する XAML ブラウザー アプリケーション (XBAP)**: このテクノロジでは、Windows で実行されている Microsoft Internet Explorer または Mozilla Firefox で、サンドボックス化された WPF アプリケーションをホストします。Silverlight とは異なり、WPF フレームワークの大部分を使用できますが、部分信頼されたサンドボックスからシステム リソースへのアクセスには制限があります。XBAP では、クライアント デスクトップに Windows Vista がインストールされているか、.NET Framework 3.5 と XBAP ブラウザー プラグインの両方がインストールされている必要があります。XBAP が適しているのは、Silverlight で必要な機能が提供されていない場合と、クライアント プラットフォームおよび信頼の要件を指定できる場合です。

リッチ インターネット アプリケーション

次のプレゼンテーション テクノロジは、RIA で使用するのに適しています。

- **Silverlight**: さまざまなプラットフォームやブラウザーで動作する、ブラウザー用に最適化された WPF のサブセットです。XBAP に比べて、Silverlight は軽量で短時間にインストールできますが、3D グラフィックスとテキスト フローを使用するドキュメントはサポートされません。Silverlight は、リソースの使用量が少なく、さまざまなプラットフォームでサポートされるので、高品質な WPF グラフィックのサポートが不要な WPF アプリケーションに適しています。
- **AJAX を使用する Silverlight**: Silverlight は、Asynchronous JavaScript and XML (AJAX) をネイティブにサポートしており、オブジェクト モデルを Web ページに含まれる JavaScript に提供します。この機能を使用すると、ページ コンポーネントとサーバー間の操作が可能になり、より応答性が高い対話的な UI を提供できます。

Web アプリケーション

次のプレゼンテーション テクノロジは、Web アプリケーションで使用するのに適しています。

- **ASP.NET の Web フォーム**: これは、.NET Web アプリケーション向けの UI を設計および実装する標準的なテクノロジです。ASP.NET の Web フォーム アプリケーションは Web サーバーにのみインストールする必要があり、クライアント デスクトップにコンポーネントをインストールする必要はありません。
- **AJAX を使用する ASP.NET の Web フォーム**: サーバーとクライアント間の要求を非同期に処理して応答性を向上し、クライアントに対してより優れたエクスペリエンスを提供し、サーバーへのポストバック数を削減するために、AJAX と ASP.NET の Web フォームを併用します。AJAX は、.NET Framework 3.5 以降で ASP.NET に不可欠な要素です。
- **Silverlight を使用する ASP.NET の Web フォーム**: 既存の ASP.NET アプリケーションがあれば、Silverlight コントロールを使用してユーザー エクスペリエンスを向上し、Silverlight アプリケーションをゼロから作成することを回避できます。これは、既存のアプリケーションに Silverlight コンテンツを追加するのに適した手法です。
- **ASP.NET の MVC**: このテクノロジを使用すると、ASP.NET を使用して Model-View-Controller (MVC) パターンに基づくアプリケーションを構築できます。ASP.NET の MVC では、テスト駆動開発がサポートされ、UI 処理と UI レンダリングの間における懸念事項が明確に分離されます。この手法は、プレゼンテーション情報とロジック コードが混在しないようにするのに役立ちます。
- **ASP.NET の Dynamic Data**: このテクノロジを使用すると、LINQ to Entities 機能を利用するデータ ドリブン ASP.NET アプリケーションを作成できます。単純なスキャフォールディングと完全なカスタマイズ機能の両方がサポートされている、基幹業務 (LOB) スタイルのデータ ドリブン アプリケーションの迅速な開発モデルが提供されます。

メリットと考慮事項

次の表に、前のセクションで説明した各プレゼンテーション テクノロジのメリットと考慮事項を示します。

モバイル アプリケーション

テクノロジ	メリット	考慮事項

.NET Compact Framework	クライアント コンピューターで実行して、パフォーマンスと応答性を向上できます。 ネットワーク接続が常時確立されている必要はありません。 Windows フォームを使い慣れている場合は、なじみのあるプログラミング モデルを使用できます。 Visual Studio でデザイナーがサポートされます。 通常は、デバイスの ROM にインストールされます。	デスクトップの Windows フォーム アプリケーションと比べて、使用できる API が限られています。 ASP.NET for Mobile アプリケーションよりもクライアント側で多くのリソースを必要とします。 ASP.NET for Mobile アプリケーションよりも Web に配置するのが困難です。
ASP.NET for Mobile	さまざまなデバイス (Web ブラウザーを搭載している任意のデバイス) がサポートされます。 アプリケーションをインストールする必要がないので、デバイスのリソースが消費されません。 ASP.NET Web フォームを使い慣れている場合は、なじみのあるプログラミング モデルを使用できます。 デザイナーをサポートする Visual Studio のテンプレートを Web からダウンロードできます。	Visual Studio 2008 ではデザイナーがサポートされなくなりましたが、コントロールは、デバイスでレンダリングされます。 実行するには、ネットワーク接続が常時確立されている必要があります。 パフォーマンスと応答性は、ネットワーク帯域幅とネットワークの待ち時間の影響を受けます。 現在、多くのデバイスで HTML が完全にサポートされているので、標準的な ASP.NET アプリケーションの方が適しています。
Silverlight for Mobile	2D グラフィックス、ベクター グラフィックス、アニメーションなどの、機能豊富な UI と豊富な視覚表現を使用できます。 デスクトップで実行している Silverlight コードは、Silverlight for Mobile でも実行できます。	Web アプリケーションよりもデバイスのリソースを多く使用します。 デスクトップの Silverlight アプリケーションをモバイルで実行する場合は、メモリが少なく、ハードウェアの動作が遅いため、最適化が必要になることがあります。

	ブラウザー キャッシュ外でオブジェクトを保持するのに分離ストレージを使用できます。	Silverlight プラグインをインストールする必要があります。 プラグインのインストール要件により、Web アプリケーションほど多くの種類のデバイスで実行できません。

リッチ クライアント アプリケーション

テクノロジ	メリット	考慮事項
Windows フォーム	なじみのあるプログラミング モデルを使用できます。 Microsoft Visual Studio でデザイナーがサポートされます。 さまざまなクライアント ハードウェアで、良いパフォーマンスを得られます。	3D グラフィックス、ストリーミング メディア、フロー可能なテキスト、または UI のスタイルや UI テンプレートなどの WPF で使用できる他の高度な UI 機能はサポートされません。 クライアントにインストールする必要があります。
WPF ユーザー コントロールを使用する Windows フォーム	機能豊富な UI を、既存の Windows フォーム アプリケーションに追加できます。 完全な WPF アプリケーションへの移行戦略が提供されます。	UI の複雑さによっては、高性能なグラフィックス ハードウェアが必要になる場合があります。 Windows フォーム コントロールと WPF コントロールを重複して使用することはできません。
WPF アプリケーション	2D と 3D のグラフィックスなどの機能豊富な UI と豊富な視覚表現、画面解像度への依存からの解放、ベクター グラフィックス、フロー可能なテキスト、およびアニメーションがサポートされます。 可変帯域幅のストリーミング メディア (アダプティブ メディア ストリー	UI の複雑さによっては、高性能なグラフィックス ハードウェアが必要になる場合があります。 デザイン チームにとって、Expression Blend は Visual Studio よりもなじみが薄い可能性があります。 WPF に付属している組み込みのコン

	ミング) がサポートされます。 XAML を使用すると、UI、データ バインド、およびイベントが定義しやすくなります。 開発者とデザイナーの連携がサポートされ、それぞれが別の作業を進めながら連携できます。	トロールは、Windows フォームほど多くありません。
Windows フォーム コントロールを使用する WPF	グリッド コントロールなど、WFP で提供されていないコントロールを WPF に追加できます。	WindowsFormsHost が必要です。 境界を越えた遷移を行うために、フォーカスと入力を取得することが困難です。 WPF コントロールと Windows フォーム コントロールを重複して使用することはできません。 WPF コントロールと Windows フォーム コントロールでは、それぞれ別のレンダリング技法が使用されるので、プラットフォームによって表示が変わる場合があります。
WPF を使用する XBAP	Web に WPF アプリケーションを配置できます。 WPF のメリットである、豊富な視覚表現と機能豊富な UI をすべて使用できます。 WPF アプリケーションや Windows フォーム アプリケーションよりも配置しやすく、簡単に更新できます。	Windows Vista、または .NET Framework 3.5 と XBAP ブラウザー プラグインがインストールされているクライアントでしか機能しません。 Internet Explorer ブラウザーと Mozilla Firefox ブラウザーでしか機能しません。また、クライアントでのリソース アクセスがいくらか制限される場合があります。

リッチ インターネット アプリケーション

テクノロジ	メリット	考慮事項

Silverlight	クライアント コンピューターへの簡易インストールが可能です。 メディア ストリーミング、2D グラフィックス、ベクター グラフィックス、アニメーション、解像度への依存からの解放など、WPF の UI と視覚表現の機能のほとんどを使用できます。 分離ストレージにより、ブラウザー キャッシュから独立したアプリケーション キャッシュが提供されます。 高解像度ビデオがサポートされます。 クライアント側の処理によって、Web アプリケーションよりもユーザー エクスペリエンスと応答性が向上します。 C#、Visual Basic .NET、Ruby、Python など、さまざまな言語がサポートされます。 JavaScript の代わりに、ウィンドウなしのバックグラウンド処理をサポートします。 Mac や Linux など、さまざまなプラットフォームがサポートされます。 Firefox や Safari など、さまざまなブラウザーがサポートされます。	Silverlight プラグインをクライアントにインストールする必要があります。 デザイン チームにとって、Expression Blend は Visual Studio よりもなじみが薄い可能性があります。 WPF の高度な 3D グラフィックスやフロー可能なテキストはサポートされません。 XAML とコントロールの違いにより、WPF または XBAP から移行することが困難です。
AJAX を使用する Silverlight	既存の AJAX ライブラリと Silverlight アプリケーションのルーチンを使用できます。 応答性の高い対話型インターフェイスを提供するため、ユーザーのアプリケーション操作に合わせて、サーバーとの通信により Silverlight オブジェクトを動的に作成または破棄できます。	チームが純粋な ASP.NET か Silverlight を使い慣れている場合、プログラミング モデルになじみがない可能性があります。

Web アプリケーション

テクノロジ	メリット	考慮事項
ASP.NET の Web フォーム	Windows フォームに似た開発エクスペリエンスを Web 上で実現できます。 クライアントとの依存関係がありません。 クライアントには何もインストールする必要がありません。 さまざまなプラットフォームやブラウザーがサポートされます。 Visual Studio でデザイナーがサポートされます。 さまざまなコントロールが使用できます。	UI では、HTML とダイナミック HTML (DHTML) しかサポートされません。 クライアント側のストレージは、Cookie とビュー ステートに制限されます。 ページのコンテンツを更新するには、ページ全体のポストバックとページの更新を行う必要があります。 すべての処理がサーバーで行われるので、UI の応答性が制限されます。
AJAX を使用する ASP.NET の Web フォーム	UI の応答性とエクスペリエンスが向上します。 遅延読み込みがサポートされます。 ページを部分的に更新できます。 ASP.NET 3.5 に不可欠な要素です。	チームが純粋な ASP.NET を使い慣れている場合、プログラミング モデルになじみがない可能性があります。 クライアントで JavaScript が無効になっている場合は機能しません。
Silverlight を使用する ASP.NET の Web フォーム	既存の ASP.NET アプリケーションに、Silverlight の豊富な視覚表現と機能豊富な UI を追加できます。 完全な Silverlight アプリケーションへの移行戦略を提供します。	Silverlight プラグインをクライアントにインストールする必要があります。 デザイン チームにとって、Expression Blend は Visual Studio よりもなじみが薄い可能性があります。
ASP.NET の MVC	テスト駆動開発がサポートされます。 UI 処理と UI レンダリングの分離が	ビュー ステートはサポートされません。 管理イベントはサポートされませ

	強化されます。 ユーザーにとってわかりやすく、検索エンジンで検出されやすい URL を作成できます。 マークアップを完全に制御できます。 コンテンツがレンダリングされる方法を完全に制御できます。 ナビゲーションは構成によって制御されるので、必要なコードの量が大幅に減ります。	ん。
ASP.NET の Dynamic Data	自動的にレンダリングする完全なデータ ドリブン サイトを作成できます。 LINQ クエリ言語のサポートが組み込まれています。 ADO.NET Entity Framework のサポートが組み込まれています。 LINQ を使用すると、データベースをモデル化して、オブジェクトをデータにマッピングできます。	現在、このテクノロジをサポートするコントロールは少ししかありません。

一般的なシナリオとソリューション

これ以降のセクションでは、4 つの基本的なアプリケーションの原型 (モバイル アプリケーション、リッチ クライアント アプリケーション、RIA、および Web アプリケーション) に、適切な種類のプレゼンテーション テクノロジを選択するためのガイダンスを提供します。

モバイル アプリケーション

モバイル アプリケーションで使用するプレゼンテーション テクノロジを選択する際には、次のガイドラインを考慮します。

次のような場合は、.NET Compact Framework の使用を検討します。

- 不定期に接続するシナリオやオフライン シナリオをサポートする必要があるモバイル アプリケーションを構築している。
- パフォーマンスと応答性を最大限に高めるためにクライアント側で実行するモバイル アプリケーションを構築している。

次のような場合は、ASP.NET for Mobile の使用を検討します。

- ASP.NET に関する技術的な専門知識があり、できる限り多くの種類のデバイスを対象にする必要がある。
- クライアントへのインストールまたはプラグインの依存関係を回避する必要があるアプリケーションを構築している。
- ネットワーク接続が常時確立されていることを前提としたアプリケーションを構築している。
- 使用するデバイスのリソースをできる限り少なくするか、デバイスにおけるリソースの使用量を最小限に抑える必要がある。

次のような場合は、Silverlight for Mobile の使用を検討します。

- モバイル Web アプリケーションを構築していて、Silverlight の豊富な視覚表現と UI の機能を活用する必要がある。
- 対象のデバイスでは、Silverlight プラグインに簡単にアクセスできるか、Silverlight プラグインが既にインストールされている。

リッチ クライアント アプリケーション

リッチ クライアント アプリケーションで使用するプレゼンテーション テクノロジを選択する際には、次のガイドラインを考慮します。

次のような場合は、Windows フォームの使用を検討します。

- Windows フォーム アプリケーションを構築した経験があり、別のテクノロジに移行する余裕がない。
- 既存の Windows フォーム アプリケーションを拡張または変更している。
- リッチ メディアやアニメーションをサポートする必要がない。

次のような場合は、WPF の使用を検討します。

- リッチ クライアント アプリケーションを構築していて、WPF の豊富な視覚表現と UI の機能を活用する必要がある。

- XBAP を使用して Web に配置する必要があるリッチ クライアント アプリケーションを構築している。

次のような場合は、WPF ユーザー コントロールを使用する Windows フォームの使用を検討します。

- Windows フォーム アプリケーションが既にあり、高度なグラフィック、フロー可能なテキスト、ストリーミング メディア、アニメーションなどの WPF の機能を最大限に活用する必要がある。

次のような場合は、Windows フォーム コントロールを使用する WPF の使用を検討します。

- WPF を使用してリッチ クライアント アプリケーションを構築していて、WPF にないコントロールを使用する必要がある。

次のような場合は、XBAP の使用を検討します。

- Web に配置する WPF アプリケーションが既にある。
- Silverlight にない WPF の豊富な視覚表現と UI の機能を活用する必要がある。

リッチ インターネット アプリケーション

RIA の実装で使用するプレゼンテーション テクノロジを選択する際には、次のガイドラインを考慮します。

次のような場合は、Silverlight の使用を検討します。

- Silverlight の豊富な視覚表現、ストリーミング メディア、および UI の機能を活用する必要がある。
- シームレスな配置と、アプリケーションで使用する個々のモジュールの遅延読み込みをする機能が必要なアプリケーションを構築している。
- さまざまなプラットフォームで実行しているさまざまなブラウザーをサポートする必要がある。

次のような場合は、AJAX を使用する Silverlight の使用を検討します。

- Web ページで、Silverlight オブジェクト モデルのオブジェクト インスタンスを動的に管理できる必要がある。
- Web ページで行われたユーザー操作に基づいて Silverlight コントロールを操作する必要がある。

Web アプリケーション

Web アプリケーションで使用するプレゼンテーション テクノロジを選択する際には、次のガイドラインを考慮します。

次のような場合は、ASP.NET の Web フォームの使用を検討します。

- ASP.NET の Web フォームを構築した経験がある。
- 拡張または変更が必要な既存の ASP.NET の Web フォームがある。
- できる限り多い種類のクライアント コンピューターで実行できるようにする必要がある。
- クライアントには何もインストールしないようにする必要がある。
- 機能豊富な UI やアニメーションを使用しないで、作成、読み取り、更新、および削除 (CRUD) 操作のような単純な機能を設計する必要がある。

次のような場合は、AJAX を使用する ASP.NET の Web フォームの使用を検討します。

- 応答性が高く、機能豊富な UI を備えた ASP.NET の Web フォームを作成する必要がある。
- 遅延読み込みと部分的なページの更新をサポートする必要がある。

次のような場合は、Silverlight を使用する ASP.NET の Web フォームの使用を検討します。

- ASP.NET Web フォーム アプリケーションが既にあり、Silverlight の豊富な視覚表現と UI の機能を活用する必要がある。
- 既存の Web アプリケーションを Silverlight に移行することを計画している。

次のような場合は、ASP.NET の MVC の使用を検討します。

- Model-View-Controller (MVC) パターンで実装する必要がある。
- マークアップを完全に制御する必要がある。
- UI 処理と UI レンダリングの間における懸念事項を明確に分離する必要がある。
- テスト駆動開発を実践する必要がある。

次のような場合は、ASP.NET の Dynamic Data の使用を検討します。

- データ ドリブン アプリケーションをすばやく構築する必要がある。
- LINQ クエリ言語か、Entity Framework データ モデルを使用する必要がある。
- LINQ に組み込まれているモデル機能を使用して、オブジェクトをデータに簡単にマップする必要がある。

関連情報

Web リソースに簡単にアクセスするには、http://www.microsoft.com/architectureguide (英語) でオンライン版の参考文献を参照してください。

- Silverlight の詳細については、Silverlight の公式 Web サイト (http://silverlight.net/default.aspx、英語) を参照してください。
- Islands of Richness (ページの一部分に追加する機能豊富な UI 要素) の詳細については、http://blogs.msdn.com/brada/archive/2008/02/18/islands-of-richness-with-silverlight-on-an-asp-net-page.aspx (英語) を参照してください。

付録 C

データ アクセス テクノロジ

概要

この付録は、データ アクセス テクノロジを選択する際に必要となるトレードオフについて理解するのに役立ちます。また、特定のテクノロジによる設計への影響を把握したり、シナリオやアプリケーションの種類に基づいて使用するデータ アクセス テクノロジを選択したりする際にも役立ちます。
選択するデータ アクセス テクノロジは、開発しているアプリケーションの種類と使用するビジネス エンティティの種類によって異なります。「データ アクセス テクノロジの概要」を参照して、各テクノロジとその説明を確認できます。また、「メリットと考慮事項」を参照して、データ アクセスで使用できるテクノロジの範囲を理解できます。「一般的なシナリオとソリューション」では、アプリケーションのシナリオを、一般的なデータ アクセス テクノロジのソリューションにマップするのに必要な情報を提供しています。

データ アクセス テクノロジの概要

マイクロソフト プラットフォームで使用できるデータ アクセス テクノロジは次のとおりです。

- **ADO.NET Core**: ADO.NET Core では、一般的なデータの取得、更新、および管理を行うための機能を提供します。SQL Server、OLE DB、Open Database Connectivity (ODBC)、SQL Server Compact Edition、および Oracle の各データベースのプロバイダーが含まれます。
- **ADO.NET Data Services フレームワーク**: このフレームワークでは、HTTP 経由でアクセスする RESTful Web サービスを通じて、エンティティ データ モデルを使用してデータを公開します。このデータは、Uniform Resource Identifier (URI) を使用して直接アドレス指定できます。データをシンプ

ルな Atom 形式および JavaScript Object Notation (JSON) 形式で返すように、Web サービスを構成できます。

- **ADO.NET Entity Framework**: このフレームワークでは、リレーショナル データベースに関して、厳密に型指定されたデータ アクセス エクスペリエンスが提供されます。このフレームワークでは、データ モデルを、リレーショナル テーブルの物理的な構造から、一般的なビジネス オブジェクトを正確に反映した概念モデルに移行します。Entity Framework では、ADO.NET 環境内に一般的なエンティティ データ モデルが導入されているので、開発者はリレーショナル データへの柔軟なマッピングを定義できます。このマッピングは、基盤となるストレージ スキーマの変更からアプリケーションを分離するのに役立ちます。また、Entity Framework では、LINQ to Entities もサポートされているので、Entity Framework を通じて公開されるビジネス オブジェクトに LINQ のサポートが提供されます。O/R(オブジェクト/リレーショナル)ツール マッピング (O/RM ツール)として使用すると、開発者は LINQ to Entities をビジネス オブジェクトに対して使用します。その結果、ビジネス オブジェクトが、Entity Framework で管理されるエンティティ データ モデルにマップされる Entity SQL に変換されます。開発者は、エンティティ データ モデルを直接操作したり、アプリケーションで Entity SQL を使用したりすることもできます。
- **Sync Services for ADO.NET**: Sync Services for ADO.NET は、Microsoft Sync Framework に含まれているプロバイダーです。ADO.NET 対応のデータベースに同期機能を実装するために使用されます。このプロバイダーを使用すると、不定期に接続するアプリケーションにデータを同期する機能を組み込むことができます。このプロバイダーでは、クライアント データベースから情報を定期的に収集して、この情報をサーバー データベースと同期します。
- **統合言語クエリ (LINQ)**: LINQ では、クエリのネイティブな言語構文で C# や Visual Basic を拡張するクラス ライブラリが提供されます。これは基本的には、.NET Framework 全体でさまざまなアセンブリによってサポートされるクエリ テクノロジです。たとえば、LINQ to Entities は ADO.NET Entity Framework アセンブリに、LINQ to XML は System.Xml アセンブリに、LINQ to Objects は .NET Framework の中核となるシステム アセンブリに含まれます。クエリは、さまざまな形式のデータに対して実行できます。たとえば、DataSet (LINQ to DataSet)、XML (LINQ to XML)、メモリ内のオブジェクト (LINQ to Objects)、ADO.NET Data Services (LINQ to Data Services)、リレーショナル データ (LINQ to Entities) などです。
- **LINQ to SQL**: LINQ to SQL では、SQL Server に対する軽量で厳密に型指定されたクエリ ソリューションを提供します。LINQ to SQL は、オブジェクトを簡単かつ迅速に永続化するシナリオを実現するために設計されています (このシナリオでは、中間ティアのクラスが、データベースのテーブル構造

に緊密にマップされます)。.NET Framework 4 以降では、LINQ to SQL のシナリオが ADO.NET Entity Framework によって統合およびサポートされます。ただし、LINQ to SQL のテクノロジは引き続きサポートされます。詳細については、ADO.NET チームのブログ (http://blogs.msdn.com/adonet/archive/2008/10/31/clarifying-the-message-on-l2s-futures.aspx、英語) を参照してください。

メリットと考慮事項

次の表に、前のセクションで説明した各データ アクセス テクノロジのメリットと考慮事項を示します。各表では、オブジェクト リレーショナル データ アクセス、非接続型とオフラインのデータ アクセス、SOA とサービスのシナリオ、n ティアと一般的なシナリオなど、さまざまな使用シナリオについて説明します。この一連の表の後に続くセクションでは、この付録で説明したデータ アクセス テクノロジに関する一般的な推奨事項を提示します。

オブジェクト リレーショナル データ アクセス

テクノロジ	メリット	考慮事項
ADO.NET Entity Framework (EF)	基になるデータベース構造を、論理データ モデルから分離します。 Entity SQL は、すべてのデータ ソースとデータベースの種類で整合性のあるクエリ言語です。 メタデータを明確に定義されたアーキテクチャ レイヤーに分離します。 ビジネス ロジックの開発者は、データベースの詳細を把握していなくてもデータにアクセスできます。 Visual Studio で機能豊富なデザイナーがサポートされていることにより、データ エンティティの構造を視覚化できます。 プロバイダー モデルを使用すると、多数のデータベースにマップできます。	従来のデータ アクセス方法を使用している場合は、エンティティとクエリの設計を変更する必要があります。 個別のオブジェクト モデルを使用します。 LINQ to DataSet より多くの抽象化レイヤーが用意されています。 LINQ to Entities と併用することも、LINQ to Entities なしで使用することもできます。 データベースの構造を変更した場合は、エンティティ データ モデルを再生成し、EF ライブラリを再配置する必要があります。

LINQ to Entities	ADO.NET Entity Framework のリレーショナル データに対応した LINQ ベースのソリューションです。 リレーショナル データに対して、厳密に型指定された LINQ アクセスが提供されます。 EF のエンティティ データ モデルをベースに作成されたオブジェクトに対する LINQ ベースのクエリをサポートします。 すべての処理はサーバー側で行われます。	ADO.NET Entity Framework が必要です。
LINQ to SQL	データ オブジェクト モデルが物理データベース モデルと一致するときには、オブジェクトを簡単に読み取ったり書き込んだりできます。 SQL データに対して、厳密に型指定された LINQ クエリ アクセスを提供します。 すべての処理はサーバー側で行われます。	.NET Framework 4 では、この機能が Entity Framework に統合されています。 LINQ クエリは、プロバイダー経由ではなく、直接データベースにマップされるため、Microsoft SQL Server でしか機能しません。

非接続型とオフラインのデータ アクセス

テクノロジ	**メリット**	**考慮事項**
LINQ to DataSet	DataSet に対して完全な機能を備えたクエリを実行します。	すべての処理はクライアント側で行われます。
Sync Services for ADO.NET	データベース間、コラボレーション間、およびオフライン シナリオ間の同期を可能にします。 同期をバックグラウンドで実行できます。 データベース間のコラボレーションで、ハブ・スポーク型アーキテクチャを提供します。	独自の変更追跡機能を実装する必要があります。 同期中に大量のデータをやり取りすると、パフォーマンスが低下するおそれがあります。

SOA とサービスのシナリオ

テクノロジ	メリット	考慮事項
ADO.NET Data Services フレームワーク	データは、REST 形式を使用して、URI で直接アドレス指定できます。 データを Atom 形式または JSON 形式で返せます。 新しいサービス インターフェイスのリリースを簡略化する簡易版の形式が用意されています。 開発者は、.NET Framework、Silverlight、および AJAX クライアント ライブラリを利用して、オブジェクトを直接操作したり、ADO.NET Data Services に対して、厳密に型指定された LINQ アクセスを提供したりできます。 .NET Framework、Silverlight、および AJAX クライアント ライブラリでは、Windows Azure テーブル、SQL Server Data Services などのマイクロソフト サービスに対して、使い慣れた API サーフェイスを提供します。	サービス指向シナリオにしか適用されません。
LINQ to Data Services	ADO.NET Data Services から返されるクライアント側のデータに対して、LINQ ベースのクエリを作成できます。 REST 形式のデータに対して LINQ ベースのクエリをサポートします。	ADO.NET Data Services のクライアント側のフレームワークとしか併用できません。

n ティアと一般的なシナリオ

テクノロジ	メリット	考慮事項
ADO.NET Core	広範なデータ ストアにアクセスできる、.NET マネージ コードのプロバイダーが用意されています。 非接続型のデータ ストレージとデータ操作を実現する機能を提供します。	コードは特定のプロバイダーを対象に記述されているため、再利用性が低下します。 リレーショナル データベース構造がオブジェクト モデルと合わない可能性があります。この場合は、データ マッピング レイヤーの作成が必要となります。
ADO.NET Data Services フレームワーク	データは、REST 形式を使用して、URI で直接アドレス指定できます。 データを Atom 形式または JSON 形式で返せます。 新しいサービス インターフェイスのリリースを簡略化する簡易版の形式が用意されています。 プロバイダー モデルでは、IQueryable データ ソースを使用できます。 .NET Framework、Silverlight、および AJAX クライアント ライブラリでは、Windows Azure テーブル、SQL Server Data Services などのマイクロソフト サービスに対して、使い慣れた API サーフェイスを提供します。	サービス指向シナリオにしか適用されません。 大量のデータを使用するサービスに適切にマップできる、リソースを中心としたサービスを提供しますが、サービスのほとんどが操作を中心としている場合は、追加の作業が必要になることがあります。
ADO.NET Entity Framework	メタデータを明確に定義されたアーキテクチャ レイヤーに分離します。 複雑なオブジェクト モデルをクエリする LINQ to Entities をサポートしています。 プロバイダー モデルを使用すると、多数の種類のデータベースにマップできます。 境界が明確に定義されたサービスや、サービス境界間で明確に定義されたエンティティを送受信するためのデータ コントラクトとサービス コントラクトを構築できます。	従来のデータ アクセス方法を使用している場合は、エンティティとクエリの設計を変更する必要があります。 エンティティ オブジェクトをネットワーク経由で送信できます。また、Data Mapper パターンを使用して、より汎用的な DataContract 型のオブジェクトに、エンティティを変換できます。計画された POCO のサポートにより、ネットワーク経

	エンティティ データ モデルのエンティティのインスタンスは、Web サービスで直接シリアル化して使用できます。 ペイロードを構築する際に完全な柔軟性を提供します。つまり、個々のエンティティ、エンティティのコレクション、またはエンティティのグラフをサーバーに送信します。 最終的には、永続性を完全に無視したオブジェクトをサービス境界間で利用できます。	由で送信する際にオブジェクトを変換する必要がなくなります。 汎用的なエンティティのグラフを受信するサービスのエンドポイントよりも、受信するペイロードに対して厳しいコントラクトを施行するエンドポイントを構成する方がサービス指向です。
LINQ to Objects	メモリ内のオブジェクトに対して、LINQ ベースのクエリを作成できます。 コレクションからデータを取得する新しい方法を提示します。 IEnumerable または IEnumerable<T> をサポートするコレクションと直接組み合わせて使用できます。 文字列、リフレクション ベースのメタデータ、およびファイル ディレクトリのクエリに使用できます。	IEnumerable インターフェイスを実装したオブジェクトでしか機能しません。
LINQ to XML	XML データに対して、LINQ ベースのクエリを作成できます。 XML ドキュメントをメモリ内に格納するドキュメント オブジェクト モデル (DOM) と同じように機能しますが、ドキュメント オブジェクト モデルよりも使いやすくなっています。 クエリ結果は、XElement および XAttribute オブジェクト コンストラクターのパラメーターとして使用できます。	ジェネリック クラスに大きく依存しています。 信頼されていない XML ドキュメントを使用できるように最適化されていません。このようなドキュメントを使用するには、別のセキュリティ対策の技法が必要です。
LINQ to SQL	オブジェクト モデルとデータベース モデルが同じ場合、データをオブジェクトとして簡単に取得して更新できます。	.NET Framework 4 では、LINQ とリレーショナル データベースに関連するシナリオのデータ アクセス ソリューションとして、Entity Framework が推奨されています。 LINQ to SQL は今後も引き続きサポートされ、コミュニティから寄せ

		られるフィードバックに基づいて発展します。

一般的な推奨事項

データ アクセス テクノロジを選択する際には、次の一般的な推奨事項を検討します。

- **柔軟性とパフォーマンス**: パフォーマンスと柔軟性を最大限に高める必要がある場合は、ADO.NET Core の使用を検討します。ADO.NET Core は、多数の機能を提供し、最もサーバーに特化したソリューションです。このテクノロジを使用するときには、柔軟性の向上とカスタム コードを記述する必要性のトレードオフを考慮します。カスタム オブジェクトにマップするとパフォーマンスが低下することに注意してください。ADO.NET プロバイダーを使用して、構成を通じたデータベースの変更をサポートする軽量なフレームワークが必要な場合は、Enterprise Library の Data Access Application Block を使用することを検討します。
- **オブジェクト リレーショナル マッピング (O/RM)**: O/RM ベースのソリューションを必要としていたり、複数のデータベースをサポートしたりする必要がある場合、またはその両方が必要な場合は、ADO.NET Entity Framework の使用を検討します。このような状況下では、ドメインモデル パターンのシナリオで実装するのが最適です。
- **オフライン シナリオ**: 非接続型のシナリオをサポートする必要がある場合、DataSet または Sync Framework の使用を検討します。
- **n ティア シナリオ**: レイヤー間またはティア間でデータを渡す場合には、エンティティ オブジェクト、エンティティにマップされたデータ転送オブジェクト (DTO) 、DataSet、およびカスタム オブジェクトを渡すことができます。リソースを中心としたサービス (REST) を構築する場合は、ADO.NET Data Services の使用を検討します。操作を中心としたサービス (SOAP) を構築する場合は、Windows Communication Foundation (WCF) サービスと、明示的に定義されたサービスおよびデータ コントラクトを併用することを検討します。
- **SOA とサービスのシナリオ**: データベースをサービスとして公開する場合は、ADO.NET Data Services の使用を検討します。データをクラウドで保存する場合は、SQL Server Data Services の使用を検討します。
- **Microsoft Windows Mobile**: ほとんどの Windows Mobile デバイスに搭載されているメモリの性能には制限があるため、多くのデータ テクノロジは重すぎます。モバイル デバイス上でデータを保持し

たり、サーバー ベースのデータベース システムとデータを同期したりするには、SQL Server Compact Edition データベースと Sync Services for ADO.NET の使用を検討します。マージ レプリケーションなどの機能も、Windows Mobile シナリオで役に立ちます。

一般的なシナリオとソリューション

次のセクションでは、アプリケーションに適切な種類のデータ アクセス テクノロジを選択するためのガイダンスを提供します。

次のような場合は、ADO.NET Core の使用を検討します。

- 低レベルの API を使用して、アプリケーションで行うデータ アクセスを完全に制御する必要がある。
- ADO.NET プロバイダーへの既存の投資を活用する必要がある。
- データベースへのアクセスに従来のデータ アクセス ロジックを使用している。
- 他のデータ アクセス テクノロジで提供される追加の機能が不要である。
- ネットワークに接続されていない状態でのデータ アクセス エクスペリエンスをサポートする必要があるアプリケーションを構築している。

次のような場合は、ADO.NET Data Services フレームワークの使用を検討します。

- REST 形式の URI を使用して、サービスとして公開されるデータにアクセスする必要がある。

次のような場合は、ADO.NET Entity Framework の使用を検討します。

- アプリケーションとサービスの間で概念モデルを共有する必要がある。
- 継承を使用して 1 つのクラスを複数のテーブルにマップする必要がある。
- Microsoft SQL Server 製品ファミリ以外のリレーショナル ストアをクエリする必要がある。
- 柔軟性があるスキーマを使用してリレーショナル モデルにマップしなければならないオブジェクト モデルが存在する。
- オブジェクト モデルからマッピング スキーマを分離する柔軟性が必要である。

次のような場合は、Sync Services for ADO.NET の使用を検討します。

- 不定期に接続するシナリオをサポートするアプリケーションを構築する必要がある。
- Windows Mobile を使用して、中央データベース サーバーと同期する必要がある。

次のような場合は、LINQ to Data Services の使用を検討します。

- ADO.NET Data Services から返されるデータをクライアントで使用する。
- LINQ 構文を使用してクライアント側のデータをクエリする必要がある。
- LINQ 構文を使用して REST データをクエリする必要がある。

次のような場合は、LINQ to DataSet の使用を検討します。

- DataSet に対して、テーブルを結合するクエリなどを実行する。
- 反復的なコードを記述するのではなく、一般的なクエリ言語を使用する必要がある。

次のような場合は、LINQ to Entities の使用を検討します。

- ADO.NET Entity Framework を使用している。
- 厳密に型指定されたエンティティをクエリする必要がある。
- LINQ 構文を使用してリレーショナル データをクエリする必要がある。

次のような場合は、LINQ to Objects の使用を検討します。

- コレクションをクエリする必要がある。
- ファイル ディレクトリをクエリする必要がある。
- LINQ 構文を使用してメモリ内のオブジェクトをクエリする必要がある。

次のような場合は、LINQ to XML の使用を検討します。

- アプリケーションで XML データを使用している。
- LINQ 構文を使用して XML データをクエリする必要がある。

LINQ to SQL に関する考慮事項

LINQ to Entities は、LINQ とリレーショナル データベースのシナリオに推奨されるソリューションです。LINQ to SQL は今後もサポートされますが、革新や強化については、優先順位がそれほど高くありません。既に LINQ to SQL を利用している場合は、引き続き使用できますが、 新しいソリューションでは、LINQ to Entities の使用を検討することをお勧めします。詳細については、ADO.NET チームのブログ (http://blogs.msdn.com/adonet/、英語) を参照してください。

Windows Mobile に関する考慮事項

この付録で紹介した多数のテクノロジは、Windows Mobile オペレーティング システムでは使用できません。このドキュメントの発行時点において、次のテクノロジは Windows Mobile で使用できません。

- ADO.NET Entity Framework
- ADO.NET Data Services フレームワーク
- LINQ to Entities
- LINQ to SQL
- LINQ to Data Services
- ADO.NET Core (Windows Mobile では、SQL Server と SQL Server Compact Edition のみサポートされています)

最新のバージョンで使用できるテクノロジについては、製品ドキュメントを確認してください。

関連情報

Web リソースに簡単にアクセスするには、http://www.microsoft.com/architectureguide (英語) でオンライン版の参考文献を参照してください。

- ADO.NET (http://msdn.microsoft.com/ja-jp/library/e80y5yhx(VS.80).aspx)
- ADO.NET Data Services (http://msdn.microsoft.com/ja-jp/data/bb931106.aspx)
- ADO.NET Entity Framework (http://msdn.microsoft.com/ja-jp/data/aa937723.aspx)
- 統合言語クエリ (LINQ: Language-Integrated Query) (http://msdn.microsoft.com/ja-jp/library/bb397926.aspx)
- SQL Server Data Services (SSDS) 入門 (http://msdn.microsoft.com/en-us/library/cc512417.aspx、英語)
- Introduction to Microsoft Sync Framework (http://msdn.microsoft.com/en-us/sync/bb821992.aspx、英語)

付録 D

統合テクノロジ

概要

この付録は、統合テクノロジを選択する際に必要となるトレードオフについて理解するのに役立ちます。また、特定のテクノロジによる設計への影響を把握したり、シナリオやアプリケーションの種類に基づいて使用する統合テクノロジを選択したりする際にも役立ちます。

選択する統合テクノロジは、開発するアプリケーションの種類によって異なります。「統合テクノロジの概要」を参照して、各テクノロジとその説明を確認できます。また、「メリットと考慮事項」を参照して、統合に使用できるテクノロジの範囲を理解できます。「一般的なシナリオとソリューション」では、アプリケーションのシナリオを、一般的な統合テクノロジのソリューションにマップするのに必要な情報を提供しています。

統合テクノロジの概要

アプリケーションの統合に使用できるマイクロソフト テクノロジは次のとおりです。

- **Microsoft BizTalk® Server**: エンタープライズ アプリケーション統合 (EAI) に対応したシステムを構築するのに必要なアダプター、オーケストレーション、メッセージング、およびプロトコルを提供します。
- **Microsoft Host Integration Server**: アプリケーションを、IBM zSeries アプリケーションや iSeries アプリケーションと接続するためのプラットフォームを提供します。また、Microsoft Message Queuing と IBM WebSphere MQ の間のデータ接続もサポートします。
- **Microsoft Message Queuing**: キューに配信されたメッセージを使用して、アプリケーションを接続できます。メッセージ キューでは、確実なメッセージの配信、優先順位ベースのメッセージング、

およびセキュリティを提供します。不定期に接続したり、一時的にオフラインになる可能性があるシステムとの統合をサポートしたりできます。また、同期および非同期のメッセージング シナリオをサポートします。

- **Microsoft BizTalk ESB Toolkit**: BizTalk Server で提供されるサービスをベースに構築される、疎結合されたメッセージング アーキテクチャを提供する一連のエンティティです。また、基盤となる BizTalk Server の機能を活用して、柔軟性と拡張性を備えたアーキテクチャを提供します。このアーキテクチャには、変換、配信保証、メッセージ セキュリティ、サービスの登録、インテリジェントなルーティング、統合された例外ハンドルなどの機能が用意されています。

メリットと考慮事項

次の表に、各統合テクノロジのメリットと考慮事項を示します。

テクノロジ	メリット	考慮事項
BizTalk Server	電子データ交換 (EDI) 形式または拡張マークアップ言語 (XML) 形式、あるいはその両方を使用して企業間の電子文書交換関係を実現します。 サードパーティ製のシステムと統合できます。 ESB 機能を提供するように、簡単に拡張できます。 Windows Communication Foundation (WCF) 基幹業務 (LOB) アダプターを使用すると、BizTalk Server の内外で利用できるカスタム アダプターを開発できます。 SAP、Oracle、SQL Server データベースなどのシステムと統合するためのアダプターが用意されています。 SOAP アダプターを使用して、Web サービスを利用できます。	密結合されたインフラストラクチャになる可能性があります。 ESB 機能を使用するにはカスタマイズが必要です。
Host Integration Server	Windows Server と IBM メインフレームまたは AS/400 コンピューターの間のネットワーク統合をサポートします。	Windows Server 環境にインストールする必要があります。 Microsoft Visual Studio 2005 以降のバ

	セキュリティで保護されたホスト アクセスと ID 管理機能を提供し、Secure Sockets Layer (SSL)、トランスポート レイヤーのセキュリティ (TLS)、シングル サインオン (SSO)、およびパスワードの同期をサポートします。 データ統合を利用して、メッセージ キューと XML ベースの Web サービスをサポートします。 IBM DB2 データベースとの接続を作成して管理するデータ アクセス ツールが用意されています。 同時のホスト セッション、負荷分散、およびホット フェールオーバーにより、エンタープライズ レベルのスケーラビリティとパフォーマンスをサポートします。 BizTalk Adapters for Host Systems を使用して、DB2、IBM WebSphere MQ、Host Applications、および Host Files と BizTalk Server との統合をサポートします。	ージョンが必要です。 ルーティングをサポートするメッセージ キューが必要です。
Microsoft Message Queuing	メッセージ ベースの手法を使用して、異種ネットワーク間でアプリケーションの通信を可能にします。 社内外のアプリケーション間で信頼性の高いメッセージングをサポートします。 メッセージを 1 回だけ配信するようにする、メッセージを順番に配信するようにする、送信先キューからメッセージが取得されたことを確認するなどのトランザクション機能をサポートします。 ネットワーク トポロジ、トランスポートの接続性、およびセッション集中化の必要性に基づいてメッセージをルーティングします。 SOAP リライアブル メッセージ プロトコル (SRMP) のサポートにより、HTTP トランスポート経由でメッセージを配信できます。 複数の送信先に同一のメッセージを送信できま	メッセージ キューをインストールして構成する前に、配置モードを検討する必要があります。 ワークグループ配置モードを使用する場合は、メッセージを暗号化したり、内部証明書を使用したりすることはできません。また、プラットフォームをまたぐメッセージングはサポートされません。 依存型ではなく、独立型のクライアントを使用する必要があります。 メッセージ キューは、リモートで送信し、ローカルで受信することを前提に最適化されています。そのため、リモートではキューを読み取らないようにする必要があります。 Active Directory をクエリする機能は使用

	す。 Windows Server 2003 以降のバージョンに同梱されています。 Active Directory にアクセスできるドメイン モードおよびワークグループ モードという 2 つの配置モードをサポートします。 WCF が提供するメッセージ キューのエンドポイントが用意されています。	しないようにします。 イベントを使用した非同期の通知は失われる可能性があります。 WCF のエンドポイントには、Microsoft .NET Framework 3.0 以降のバージョンが必要です。
Microsoft BizTalk ESB Toolkit	実行時にサービス エンドポイントの動的な解決が提供されるので、エンドポイント定義が抽象化されます。 アプリケーションからメッセージの変換機能を分離します。 WCF と緊密に統合して、安全性と信頼性の高いメッセージングを提供します。 システム例外とビジネス例外の統合された例外ハンドルにより、エラーが検出および報告されます。 Universal Description, Discovery, and Integration (UDDI) などのサービス レジストリと通信するための、リゾルバーが用意されています。 ルーティングと変換でアイテナリ ベースの手法をサポートします。 クライアント側アイテナリとサーバー側アイテナリをサポートします。 カスタム リゾルバーを作成するために、リゾルバーの拡張性をサポートします。 アイテナリを作成するために BizTalk デザイナーをサポートします。 例外管理ポータルを使用できます。 例外ハンドル、ルーティング、解決などの主要な機能をすべて Web サービスとして公開します。	BizTalk Server 2006 R2 以降のバージョンが必要です。 特定のビジネス シナリオでは、カスタマイズが必要になる可能性があります。 既定では、ESB のアイテナリ トラッキングによるデータは表示されません。

	ビジネス アクティビティの監視 (BAM) を使用した、アイテナリ トラッキング機能を提供します。	

一般的なシナリオとソリューション

次のセクションでは、アプリケーションに適切な種類の統合テクノロジを選択するためのガイダンスを提供します。

次のような場合は、BizTalk Server の使用を検討します。

- サービス指向アーキテクチャ (SOA) の一環として、オーケストレーターを通じて複数の Web サービスと通信する必要がある。
- EDIFACT、ANSCI X12、HL7、HIPAA、SWIFT などの業界標準を含む、ビジネス間のプロセスをサポートする必要がある。
- サービスを並列実行する必要がある。
- 信頼性が高く、コードの変更が不要で、拡張可能な専用のサーバー インフラストラクチャを必要とするソリューションが必要である。
- アプリケーションのプロセス データをほぼリアルタイムで確認するために、BAM ソリューションを構成して、ビジネスの主要業績評価指標 (KPI) を測定できる必要がある。
- 動的に変化するビジネス要件を満たすために、簡単に変更できる宣言型のルールのポリシーにアプリケーションのビジネス ロジックを抽象化する必要がある。

次のような場合は、Host Integration Server の使用を検討します。

- IBM zSeries アプリケーションまたは iSeries アプリケーションとの通信をサポートする必要がある。
- BizTalk を DB2、WebSphere MQ、Host Applications、または Host Files と統合する必要がある。
- メッセージ キューを WebSphere MQ と統合する必要がある。

次のような場合は、Microsoft Message Queuing の使用を検討します。

- アプリケーション間でメッセージ ベースの通信をサポートする必要がある。
- サードパーティ製のプラットフォームと統合する必要がある。
- SRMP をサポートする必要がある。

次のような場合は、Microsoft BizTalk ESB Toolkit の使用を検討します。

- アイテナリ ベースの手法をサポートする必要がある。
- 動的な解決とルーティングをサポートする必要がある。

- 動的な変換を行う必要がある。
- EAI システムのために堅牢に統合された例外管理をサポートする必要がある。

関連情報

Web リソースに簡単にアクセスするには、http://www.microsoft.com/architectureguide (英語) でオンライン版の参考文献を参照してください。

- BizTalk Server の詳細については、「BizTalk Server Developer Center」(http://msdn.microsoft.com/en-us/biztalk/default.aspx、英語) を参照してください。
- Host Integration Server の詳細については、「Microsoft BizTalk Host Integration Server」(http://www.microsoft.com/hiserver/default.mspx、英語) を参照してください。
- MSMQ の詳細については、「Microsoft Message Queuing」(http://www.microsoft.com/windowsserver2003/technologies/msmq/default.mspx、英語) を参照してください。
- MSMQ のベスト プラクティスに関する詳細については、「MSMQ (Microsoft Message Queuing Services) を使ったプログラミング原則」(http://msdn.microsoft.com/ja-jp/library/ms811053.aspx) を参照してください。
- Microsoft BizTalk ESB Toolkit の詳細については、「Microsoft BizTalk ESB Toolkit」(http://msdn.microsoft.com/en-us/library/dd897973.aspx、英語) を参照してください。

付録 E

ワークフロー テクノロジ

概要

この付録は、ワークフロー テクノロジを選択する際に必要となるトレードオフについて理解するのに役立ちます。また、各テクノロジによる設計への影響を把握したり、シナリオやアプリケーションの種類に基づいて使用するワークフロー テクノロジを選択したりする際にも役立ちます。

選択するワークフロー テクノロジは、開発するワークフローの種類によって異なります。「ワークフロー テクノロジの概要」を参照して、各テクノロジとその説明を確認できます。また、「メリットと考慮事項」を参照して、ワークフローで使用できるテクノロジの範囲を理解できます。「一般的なシナリオとソリューション」では、アプリケーションのシナリオを、一般的なワークフロー テクノロジのソリューションにマップするのに必要な情報を提供しています。

ワークフロー テクノロジの概要

マイクロソフト プラットフォームで使用できるワークフロー テクノロジは次のとおりです。

- **Windows Workflow Foundation (WF)**: ワークフローの実装を可能にする基本的なテクノロジです。シーケンシャル ワークフローやステート マシン ベースのワークフローを作成する必要がある開発者や独立系ソフトウェア ベンダー (ISV) 向けのツールキットとして、WF では、シーケンシャル、ステート マシン、データ ドリブン、およびカスタムの各ワークフローの種類をサポートしています。Visual Studio の Windows ワークフロー デザイナーを使用してワークフローを作成できます。
- **ワークフロー サービス**: Windows Communication Foundation (WCF) と WF が統合されることで、ワークフロー向けの WCF ベースのサービスが提供されます。Microsoft .NET Framework 3.5 以降、

WCF は、サービスとして公開されるワークフローをサポートし、ワークフロー内からサービスを呼び出す機能を提供するように拡張されました。また、Visual Studio 2008 には、ワークフロー サービスをサポートする新しいテンプレートとツールが用意されています。

- **Microsoft Office SharePoint Services (MOSS)**: WF に基づくワークフローのサポートを提供するコンテンツ管理とコラボレーションのプラットフォームで、 SharePoint サーバーに関連するヒューマン ワークフローとコラボレーション向けのソリューションを提供します。MOSS のインターフェイスで、ドキュメント承認のためのワークフローを直接作成できます。また、SharePoint Designer または Visual Studio の Windows ワークフロー デザイナーを使用してワークフローを作成することもできます。ワークフローをカスタマイズするには、Visual Studio で WF オブジェクト モデルを使用できます。
- **Microsoft BizTalk Server**: 現在、BizTalk Server には、オーケストレーション (システム レベルのワークフローをエンタープライズ規模で統合するなど) を対象とした独自のワークフロー エンジンが用意されています。今後リリースされる BizTalk Server のバージョンでは、WF と XLANG (サービスのオーケストレーションとコラボレーションをモデル化するために使用される Web サービス定義言語の拡張) が使用される可能性があります。XLANG は、BizTalk Server で現在使用されている既存のオーケストレーション テクノロジです。BizTalk Orchestration Services を使用して、アプリケーション内およびアプリケーション間で疎結合された、実行時間の長いビジネス プロセスの全体的な設計とフローを定義できます。

ヒューマン ワークフローとシステム ワークフロー

ワークフローという言葉は、次の基本的な 2 種類の処理に用いられます。

- **ヒューマン ワークフロー**: これは、ユーザーの操作を含む処理が、一連の手順またはイベントに分けられる種類のワークフローです。イベントでは、条件付き評価に基づいて、手順を順次処理します。このワークフローのほとんどは、ユーザーによる操作で構成されます。
- **システム ワークフロー**: この特殊な種類のワークフローは、オーケストレーションと呼ばれることもあり、主にビジネス サービスとビジネス プロセス間を仲介するのに使用されます。オーケストレーションには、ユーザーの操作は含まれません。

メリットと考慮事項

次の表に、各ワークフロー テクノロジのメリットと考慮事項を示します。

テクノロジ	メリット	考慮事項
Windows Workflow Foundation (WF)	ワークフローを作成するための、開発者中心のソリューションです。 シーケンシャル ワークフロー、ステート マシン ワークフロー、およびデータ ドリブン ワークフローがサポートされます。 Visual Studio でデザイナーがサポートされます。 安全で信頼できるトランザクションによるデータのやり取りを実現するプロトコル機能が用意されています。 システムの再起動後も継続して実行される実行時間の長いワークフローがサポートされます。	アプリケーションでデザイナーをホストする場合は、カスタム コードが必要です。 実質的な並列実行はサポートされません。
ワークフロー サービス	WCF と WF が統合されます。 クライアント アプリケーションに、ワークフローをサービスとして公開できます。 ビジネス プロセスを完了するために、複数のサービスが調整されます。 ワークフロー サービスを呼び出すとき、新しいワークフロー インスタンスまたは既存のワークフロー インスタンスでは WF ランタイムが自動的に使用されます。 開発者へのサポートとして、ワークフロー サービスに対応した新しいテ	.NET Framework 3.5 以降が必要です。 既定のセキュリティ資格情報を使用しない場合は、追加のコードを記述する必要があります。

	ンプレートとツールが Visual Studio 2008 に用意されています。	
MOSS 2007 ワークフロー	ワークフロー エンジンは、WF に基づいています。 Web インターフェイスを使用して、承認ベースのワークフローを定義できます。 SharePoint Designer を使用して、条件付きワークフローやデータ ドリブン ワークフローを定義できます。 Visual Studio で、WF コンポーネントと WF サービスを使用して、カスタム ワークフローを作成できます。 Microsoft Office スイートとアプリケーションが統合されます。	ワークフローは単一サイトにバインドされ、他のサイトに存在する情報にアクセスできません。 基幹業務 (LOB) が統合された複雑なワークフロー ソリューションには、あまり適しません。
BizTalk Server	ビジネス プロセス管理向けに、単一ソリューションを提供します。 電子データ交換 (EDI) 形式または拡張マークアップ言語 (XML) 形式、あるいはその両方を使用して企業間の電子文書交換関係を実現します。 実行時間が長く疎結合されたビジネス トランザクションを設計して実行するための、オーケストレーション機能が用意されています。 サードパーティ製のシステムと統合します。 Enterprise Service Bus (ESB) 機能を使用するように、簡単に拡張できます。WCF LOB アダプターを使用すると、BizTalk Server の内外で利用可能なカスタム アダプターを開発	オーケストレーションの状態は SQL Server に保存されるので、オーケストレーションの実行中に待ち時間が発生します。 現在のバージョンでは WF がサポートされていません。ただし、新しいバージョンではサポートされる可能性があります。

	できます。	

一般的なシナリオとソリューション

次のセクションでは、アプリケーションに適切な種類のワークフロー テクノロジを選択するためのガイダンスを提供します。

次のような場合は、WF の使用を検討します。

- カスタム ワークフロー ソリューションを構築する必要がある。
- Visual Studio でワークフロー デザイナーを使用する必要がある。
- アプリケーションで WF デザイナーをホストする必要がある。

次のような場合は、ワークフロー サービスの使用を検討します。

- ワークフローをサービスとして公開する必要がある。
- ワークフローからサービスを呼び出す必要がある。
- ビジネス プロセスを完了するために、複数のサービスの呼び出しを調整する必要がある。

次のような場合は、MOSS 2007 の使用を検討します (ただし、SharePoint を既に使用している場合に限ります)。

- ユーザーによる共同作業を考慮したワークフローを有効にする必要がある。
- 承認プロセスをサポートするなどの理由で、SharePoint リストや SharePoint ライブラリでワークフローを有効にする必要がある。
- SharePoint ワークフローを拡張して、ユーザー設定のタスクを追加する必要がある。
- Visual Studio でワークフロー デザイナーを使用する必要がある。

次のような場合は、BizTalk Server の使用を検討します。

- 複数のアプリケーションとシステムで機能するワークフロー ソリューションが必要である。
- エンタープライズ規模での統合を可能にする、サーバーでホストするシステム ワークフロー製品が必要である。
- サービス指向アーキテクチャ (SOA) の一部として、複数の Web サービスからデータを収集する必要があるアプリケーションを開発している。
- 完了までに数日かかる実行時間の長いビジネス プロセスを含むアプリケーションを開発している。
- 業界標準に基づいた、ビジネス間のプロセスをサポートする必要がある。
- サービスを並列実行する必要がある。

- 絶えず変化するビジネス要件を満たすために、簡単に変更できる宣言型のルールにアプリケーションのビジネス ロジックを抽象化する必要がある。

関連情報

Web リソースに簡単にアクセスするには、http://www.microsoft.com/architectureguide (英語) でオンライン版の参考文献を参照してください。

- MOSS 2007 のワークフローの詳細については、「Office SharePoint Server 2007 でのワークフロー」(http://msdn.microsoft.com/ja-jp/library/ms549489(office.12).aspx) を参照してください。
- WF の詳細については、「Windows Workflow Foundation」(http://msdn.microsoft.com/ja-jp/netframework/aa663328.aspx) を参照してください。
- ワークフロー サービスの詳細については、「WF Scenarios Guidance: Workflow Services」(http://msdn.microsoft.com/en-us/library/cc825354.aspx、英語) を参照してください。
- BizTalk Server の詳細については、「BizTalk Server Developer Center」(http://msdn.microsoft.com/en-us/biztalk/default.aspx、英語) を参照してください。
- エンタープライズ規模のワークフローの詳細については、「Architecting Enterprise Loan Workflows and Orchestrations」(http://msdn.microsoft.com/en-us/library/bb330937.aspx、英語) を参照してください。

付録 F

patterns & practices の Enterprise Library

概要

この付録では、patterns & practices の Enterprise Library について説明します。また、お使いのアプリケーションで Enterprise Library を使用して、ログ記録、例外ハンドル、データ アクセスなどの横断的関心事をすばやく簡単に実装する方法を紹介します。

Enterprise Library の目標

Enterprise Library は次の特性を実現することを目的としています。

- **一貫性**: Enterprise Library のすべてのアプリケーション ブロックには、一貫性のある設計パターンと実装の手法が用意されています。
- **拡張性**: Enterprise Library のすべてのアプリケーション ブロックには、明確な拡張ポイントが用意されており、開発者は独自のコードを追加して、アプリケーション ブロックの動作をカスタマイズできます。
- **利便性**: Enterprise Library では、グラフィカルな構成ツール、簡単なインストール手順、わかりやすく包括的なドキュメントやサンプルなど、便利な機能を数多く提供しています。
- **統合**: Enterprise Library のアプリケーション ブロックは、連携して機能するように設計されており、連携の確認テストも行われています。また、アプリケーション ブロックは個別に使用することもできます。

Enterprise Library の機能

Enterprise Library には次の機能が用意されています。

- ログ記録、例外ハンドル、検証、データ アクセスなどの横断的関心事に関するソリューションの実装に使用できる再利用可能なコードで構成されたアプリケーション ブロック。
- 簡単に Enterprise Library のアプリケーション ブロックをアプリケーションに追加したり、構成情報を指定できる構成ツール。構成ツールには、スタンドアロンの構成エディターや、Visual Studio と統合される構成ツールがあります。
- ライブラリやアプリケーション ブロックのあらゆる場所で使用されているシリアル化などの一般的なユーティリティの機能。これは開発者が独自に記述したコードで使用できます。
- 開発者や管理者が、実行時にアプリケーション ブロックの動作やパフォーマンスを監視できるインストルメンテーション機能。
- Enterprise Library のソース コードを作成したり、アセンブリを適切な場所にコピーするバッチ ファイル。
- Enterprise Library で公開されるイベントやパフォーマンス カウンターのインストルメンテーションをインストールするユーティリティ。
- Enterprise Library の例やクイック スタートで使用されるサンプル データベースを作成するユーティリティ。
- 一連のクイック スタート アプリケーションは、各アプリケーション ブロックに対応し、アプリケーション ブロックの使用方法を示します。また、各アプリケーション ブロックの一般的なシナリオを実装しており、ガイダンス ドキュメントの関連セクションへのリンクが提供されています。
- Enterprise Library の完全なソース コード。Visual Studio のプロジェクトや単体テストも用意されており、開発者はこれを使用して、ライブラリやアプリケーション ブロックを拡張したり変更することができます。開発者は、単体テストを実行したり、新しいテストを作成して、アプリケーションが設計要件を満たしていることを確認できます。

アプリケーション ブロック

次の表に、開発者が一般的なエンタープライズ開発の課題を解決するのに役立つように設計された、アプリケーション ブロックの説明を示します。

アプリケーション ブロック	説明
Caching Application Block	開発者がローカル キャッシュをアプリケーションに組み込めるようにします。メモリ内キャッシュや、必要に応じて、データベースまたは分離ストレージとして機能するバッキング ストアをサポートします。キャッシュ データを取得、追加、および削除するために必要なすべての機能を提供し、構成可能な有効期限と清掃のポリシーをサポートします。
Cryptography Application Block	開発者が暗号化機能をアプリケーションに組み込む方法を簡略化します。アプリケーションでは、このアプリケーション ブロックを使用して、情報を暗号化する、データからハッシュを作成する、ハッシュ値を比較してデータが変更されていないことを確認するなど、さまざまなタスクを実行できます。
Data Access Application Block	データを表示するために読み取る、アプリケーション レイヤーを通じてデータを渡す、変更されたデータをデータベース システムに送信するなど、一般的なデータ アクセス機能を実装する開発タスクを簡略化します。ストアド プロシージャとインライン SQL がサポートされています。また、使いやすいクラスの形式で頻繁に使用する ADO.NET の機能にアクセスできます。
Exception Handling Application Block	このアプリケーション ブロックを使用して、開発者やポリシー上の意思決定者は、エンタープライズ アプリケーションのすべてのアーキテクチャ レイヤーで発生する例外をハンドルする一貫した方針を作成できます。例外情報をログに記録したり、元の例外を別の例外に置き換えて機密情報が公開されないようにしたり、元の例外を別の例外内にラップすることで例外のコンテキスト情報を管理したりできます。
Logging Application Block	一般的なログ記録の機能を簡単に実装できます。Windows イベント ログ、電子メール メッセージ、データベース、Windows メッセージ キュー、テキスト ファイル、WMI イベント、または独自の場所に情報を書き込むことができます。
Policy Injection Application Block	オブジェクト インスタンスにポリシーを自動的に適用することで、開発者が、横断的関心事をより適切に管理したり、横断的関心事をできる限り分離したり、動作をカプセル化したりするのに役立ちます。開発者は、構成を通じて、または対象のクラスの個々のメンバーに属

	性を適用して、クラスとそのメンバーを対象とした一連のポリシーを定義します。
Security Application Block	開発者が、アプリケーションに一般的な承認関連の機能を実装したり、ユーザーの承認と認証に関するデータをキャッシュするのに役立ちます。Microsoft .NET Framework 2.0 の機能を併用すると、開発者は、一般的なセキュリティ関連の機能を簡単に実装できます。
Unity Application Block	コンストラクター、プロパティ、およびメソッド呼び出しの挿入をサポートする、軽量で拡張可能な依存関係の挿入 (DI) のコンテナーを提供します。開発者は、このコンテナーを Enterprise Library と併用して、Enterprise Library のオブジェクトや独自のカスタム ビジネス オブジェクトを生成したり、コンテナーをスタンドアロンの DI メカニズムとして使用したりできます。
Validation Application Block	開発者が、構造化され保守が容易な検証シナリオをアプリケーションに実装できる、便利な機能を提供します。null 文字列、数字の範囲の検証コントロールなど、.NET Framework のデータ型を検証するための検証コントロールのライブラリが用意されています。また、複合検証コントロールも用意されており、ルール セットをサポートしています。

Caching Application Block

Caching Application Block を使用して、メモリ内キャッシュや、必要に応じて、データベースまたは分離ストレージとして機能するバッキング ストアを使用するローカル キャッシュをアプリケーションに組み込むことができます。キャッシュ データを取得、追加、および削除するために必要なすべての機能を提供し、構成可能な有効期限と清掃のポリシーをサポートします。また、分散キャッシュなどの機能をサポートするために、プラグ可能なプロバイダーを独自に作成したり、サード パーティ製のプロバイダーを使用して、アプリケーション ブロックを拡張することもできます。キャッシュを使用すると、多くのアプリケーションのシナリオでパフォーマンスと効率が大幅に向上します。

主要なシナリオ

Caching Application Block は、次のような状況に直面した場合に適しています。

- 静的なデータまたはほとんど変更されないデータに繰り返しアクセスしている。

- 作成、アクセス、または移動で負荷の高いデータ アクセスを行っている。
- サーバーなどのソースが使用できない場合でも、データは常に利用できる必要がある。

使用すべき状況

Caching Application Block は、高度なパフォーマンスとスケーラビリティを実現するように最適化されています。また、スレッド セーフであり、例外セーフでもあります。このアプリケーション ブロックは、拡張して、独自の有効期限ポリシーやバッキング ストアを含めることができます。アプリケーションとキャッシュが同じシステムに存在するという、最も一般的なデータ キャッシュで機能するように設計されています。つまり、キャッシュはローカルに配置して、そのアプリケーション専用のキャッシュとして使用される必要があります。アプリケーションが以上のガイドラインで動作する場合、このアプリケーション ブロックは次の要件に対処するのに最適です。

- 異なるアプリケーション環境のキャッシュ機能について、一貫性のある (使用しているキャッシュ ストアによって変わることがない) シンプルなインターフェイスと実装が必要である。たとえば、開発者は、インターネット インフォメーション サービス (IIS)、エンタープライズ サービス、およびスマート クライアント環境でホストされているアプリケーション コンポーネントにキャッシュを実装するために同様のコードを作成できます。また、すべての環境に同様のキャッシュ構成オプションが存在します。
- 構成可能で永続的なバッキング ストアが必要である。このアプリケーション ブロックでは、分離ストレージやデータベースとして機能するバッキング ストアをサポートしています。開発者は、追加のバッキング ストアのプロバイダーを作成し、構成設定を使用してアプリケーション ブロックに追加できます。また、アプリケーション ブロックでは、キャッシュ項目のデータを対称的に暗号化してから、バッキング ストアに配置できます。
- キャッシュ構成の設定を変更する際に、アプリケーションのソース コードを変更する必要がないようにする。開発者は、まず、名前付きキャッシュを 1 つ以上使用するコードを作成します。システム オペレーターと開発者は、Enterprise Library の構成ツールを使用して、異なる名前が付けられたキャッシュをそれぞれ構成することができます。
- キャッシュ項目に、絶対時間、相対時間、拡張された時刻の形式 (たとえば、毎晩真夜中になど)、ファイルの依存関係、有効期限なしなど、有効期限を設定する必要がある。
- アプリケーション ブロックのソース コードを変更して、拡張性を高めたり、カスタマイズすることができます。
- 1 つのアプリケーションで、複数の種類のキャッシュ ストアを (異なるキャッシュ マネージャーを通じて) 使用する必要がある。

Caching Application Block は次の種類のアプリケーションで使用できます。

- Windows フォーム
- コンソール アプリケーション
- Windows サービス
- COM+ サーバー
- ASP.NET Web アプリケーションまたは Web サービス (ASP.NET キャッシュに含まれていない機能が必要な場合)

考慮事項

Caching Application Block を使用する際には、次の事項を考慮します。

- このアプリケーション ブロックは、単一のアプリケーション ドメインに配置する必要があります。各アプリケーション ドメインには、バッキング ストアを使用するかどうかにかかわらず、1 つまたは複数のキャッシュ ストアを含められます。
- キャッシュ ストアは、複数のアプリケーション ドメインで共有できません。
- このアプリケーション ブロックでは、バッキング ストアにキャッシュされたデータは暗号化できますが、メモリ内にキャッシュされたデータの暗号化はサポートしていません。
- このアプリケーション ブロックでは、改ざん防止対策 (署名やキャッシュ内の項目の確認) をサポートしていません。

Cryptography Application Block

Cryptography Application Block を使用すると、情報を暗号化する、データからハッシュを作成する、ハッシュ値を比較してデータが変更されていないことを確認するなどの暗号化機能を簡単に組み込むことができます。

主要なシナリオ

Cryptography Application Block は、次のような状況に直面した場合に適しています。

- すばやく簡単に情報の暗号化と復号化を行っている。
- すばやく簡単にデータからハッシュを作成している。

- ハッシュ値を比較して、データが変更されていないことを確認している。

使用すべき状況

Cryptography Application Block は次の要件に対処するのに最適です。

- 標準的なデータの暗号化、復号化、およびハッシュのタスクを実行するために作成する、ひな型のコードの要件を減らす必要がある。
- アプリケーションと社内の両方で、一貫した暗号化のノウハウを保持する必要がある。
- さまざまな領域の機能で一貫したアーキテクチャのセキュリティ モデルを使用することで、開発者が習得しなければならないことを最小限に抑える必要がある。
- 暗号化プロバイダーの実装を追加したり拡張する必要がある。
- カスタマイズ可能な主要な保護モデルが必要である。

考慮事項

Cryptography Application Block を使用する際には、次の事項を考慮します。

- このアプリケーション ブロックでは、暗号化と復号化の両方に同じキーを使用する対称的なアルゴリズムのみサポートします。
- このアプリケーション ブロックでは、暗号化キーとキーの格納は自動的に管理しません。

Data Access Application Block

Data Access Application Block では、データを表示するために読み取る、アプリケーション レイヤーを通じてデータを渡す、変更されたデータをデータベース システムに送信するなど、多くの一般的なデータ アクセスのタスクを簡略化します。このアプリケーション ブロックでは、ストアド プロシージャとインライン SQL がサポートされています。また、使いやすいクラスの形式で頻繁に使用する ADO.NET の機能にアクセスできます。

主要なシナリオ

Data Access Application Block は、次のような状況に直面した場合に適しています。

- 複数のデータ行を取得するのに、DataReader または DataSet を使用している。

- コマンドを実行して、出力パラメーターまたは単一の値の項目を取得している。
- 1 つのトランザクションで複数の操作を実行している。
- SQL Server から XML データを取得している。
- DataSet オブジェクトに含まれるデータを使用してデータベースを更新している。
- データベース プロバイダーの実装を追加したり拡張したりしている。

使用すべき状況

Data Access Application Block は次の要件に対処するのに最適です。

- 開発者が ADO.NET で提供される機能をベスト プラクティスに従って使用する際に、使いやすさと利便性が求められる。
- 標準的なデータ アクセスのタスクを実行するために作成する、ひな型のコードの要件を減らす必要がある。
- アプリケーションと社内の両方で、一貫したデータ アクセスのノウハウを保持する必要がある。
- 使用するデータベースの種類を構成を通じて変更したり、開発者がアプリケーションを異なる種類のデータベースに移植するときに、記述する必要があるコードの量を簡単に減らせる必要がある。
- 開発者が、データベースの種類ごとに異なるプログラミング モデルを学習しないで済むようにする必要がある。

考慮事項

Data Access Application Block を使用する際には、次の事項を考慮します。

- Data Access Application Block は ADO.NET の機能を補完するものであり、置き換わるものではありません。アプリケーションで、特殊な方法でデータを取得したり、特定のデータベース固有の機能を使用する必要がある場合は、ADO.NET を直接使用することを検討します。

Exception Handling Application Block

Exception Handling Application Block を使用すると、アプリケーションのすべてのアーキテクチャ レイヤーで発生する例外を処理する一貫した方針をすばやく簡単に設計したり実装したりできます。例外情報をログに記録したり、

元の例外を別の例外に置き換えて機密情報が公開されないようにしたり、元の例外を別の例外内にラップすることで例外のコンテキスト情報を管理したりできます。

主要なシナリオ

開発者は、Exception Handling Application Block を使用して、アプリケーション コンポーネントの catch ステートメントに含まれているロジックを再利用可能な例外ハンドラーとしてカプセル化できます。このアプリケーション ブロックは、次のような要件に直面した場合に適しています。

- 例外をラップする。Wrap ハンドラーを使用して、新しい例外で例外をラップします。
- 例外を置き換える。Replace ハンドラーを使用して、元の例外を別の例外に置き換えます。
- 例外をログに記録する。Logging ハンドラーを使用して、例外情報をメッセージ、スタック トレースなどの形式にして、Enterprise Library の Logging Application Block に渡して公開できるようにします。
- WCF サービス境界で例外を隠ぺいする。Windows Communication Foundation (WCF) サービス境界で使用するために設計された Fault Contract Exception ハンドラーを使用して、例外から新しいエラー コントラクトを生成します。
- 例外を伝播して、わかりやすいメッセージを表示し、ユーザーに通知して、サポート スタッフを支援する。このアプリケーション ブロックのハンドラーを組み合わせて使用して、特定の例外の種類を処理したり、必要に応じて例外を再度スローします。
- 例外メッセージをローカライズする。ハンドラーと構成を使用して、ローカライズした例外メッセージ テキストを指定します。

使用すべき状況

Exception Handling Application Block は次の要件に対処するのに最適です。

- サービス インターフェイス境界だけでなく、アプリケーションのすべてのアーキテクチャ レイヤーで例外ハンドルをサポートする必要がある。
- 例外ハンドル ポリシーを、構成を通じて管理者レベルで定義したり管理する必要がある。また、アプリケーション ブロックのコードを変更することなく、例外ハンドルを管理するルールを保守および変更できる必要がある。

- 例外情報をログ記録する機能、元の例外を別の例外に置き換えて機密情報が公開されないようにする機能、元の例外を別の例外でラップして例外のコンテキスト情報を管理する機能など、一般的に使用される例外ハンドルの機能を利用する必要がある。
- 例外情報をログに記録し、元の例外を別の例外に置き換えるなど、例外に対して必要な対応をするために、例外ハンドラーを組み合わせる必要がある。
- アプリケーション内の複数の場所とアプリケーション間でハンドラーを使用できるように、一貫した方法で例外ハンドラーを呼び出す必要がある。
- 例外ハンドラーの実装を追加したり拡張する必要がある。
- 例外をログに記録するだけでなく、ポリシーを通じてハンドルする必要がある。

Logging Application Block

Logging Application Block を使用すると、Windows イベント ログ、電子メール メッセージ、データベース、Windows メッセージ キュー、テキスト ファイル、WMI イベント、または独自の場所に情報を書き込むなど、一般的なログ記録の機能を簡単に実装できます。

主要なシナリオ

Logging Application Block は、次のような状況に直面した場合に適しています。

- イベント情報を Windows イベント ログ、電子メール メッセージ、データベース、メッセージ キュー、テキスト ファイル、Windows Management Instrumentation (WMI) イベント、または独自の場所に設定してログに記録している。
- テンプレートを使用して、イベントのコンテキスト情報を変更したり書式設定をしている。
- アプリケーションの動作を追跡したり、イベント情報を組み合わせるために使用できる ID を提供している。
- フラット ファイルへのアクセスを制限するためにアクセス制御リスト (ACL) を使用したり、ログ情報を暗号化するカスタム フォーマッターを作成して、機密情報に不当にアクセスされないようにしている。

使用すべき状況

Logging Application Block は次の要件に対処するのに最適です。

- アプリケーションと社内の両方で、一貫したログ記録のノウハウを保持する必要がある。
- 一貫したアーキテクチャ モデルを使用することで、開発者が習得しなければならないことを最小限に抑える必要がある。
- カスタム コードまたはひな型のコードを繰り返し記述することなく、一般的なアプリケーションのログ記録に関するタスクを解決できる実装を提供する必要がある。
- ログ記録の実装や対象を追加したり拡張する必要がある。

考慮事項

Logging Application Block を使用する際には、次の事項を考慮します。

- このアプリケーション ブロックのログ記録のフォーマッターでは、ログ記録情報が暗号化されません。
- トレース リスナーの出力先は、クリア テキストでログ記録情報を受け取ります。
- Logging Application Block のトレース リスナーには、部分的に信頼された状態で実行するとエラーが発生するものがあります。

Policy Injection Application Block

Policy Injection Application Block で使用できる、オブジェクト インスタンスにポリシーを自動的に適用するメカニズムを利用することで、開発者は横断的関心事をより適切に管理したり、横断的関心事をできる限り分離したり、動作をカプセル化したりできます。開発者は、Policy Injection Application Block の構成を通じて、または対象のクラスの個々のメンバーに属性を適用して、クラスとそのメンバーを対象とした一連のポリシーを定義します。

主要なシナリオ

Policy Injection Application Block は、次のような状況に直面した場合に適しています。

- 操作に依存関係がないことから多くのメリットを得て、できる限り再利用できるようにするため、カプセル化や分離が必要なオブジェクトでアプリケーションを構築している。

- 開発者、オペレーター、および管理者が、通常、アプリケーションのコードや再コンパイルを変更することなく、構成を通じてインターセプト ポリシーを作成、変更、削除、および微調整できるようにしている。このようにすることで、コードでエラーが発生する可能性が減少し、バージョン管理が簡略化して、ダウンタイムを短縮できます。
- 既存のオブジェクト インスタンスを再利用している。このようにすると、新しいオブジェクト インスタンスを作成して、プロパティを設定したりメソッドを呼び出して準備をするためのコードを記述する必要性を低減しながら、クラスの特定のメンバーまたはすべてのメンバーに対して、ハンドラーのパイプラインを使用することができます。
- アプリケーションで、ログ記録、検証、承認、インストルメンテーションなど、一般的なタスクを実行するために必要な作業や開発者が記述しなければならないコードを最小限に抑えている。

使用すべき状況

Policy Injection Application Block は次の要件に対処するのに最適です。

- 新しいアプリケーションや既存のアプリケーション (特に Enterprise Library の機能を既に使用しているアプリケーション) に簡単に実装できる既製のソリューションが必要である。
- 一般的な機能 (ログ記録、検証など) にアクセスする必要があるオブジェクトの独立性に影響するおそれのある横断的関心事を管理する必要がある。
- 開発者や管理者が、一般的なタスクを実行するハンドラーや、カスタムの機能を追加するハンドラーを追加または削除して、構成を通じてアプリケーションのオブジェクトの動作を構成できるようにする必要がある。
- 開発者が、Enterprise Library Core の機能やエンタープライズ アプリケーションで一般的に必要なタスクを実装している個々のアプリケーション ブロックの機能を、簡単に利用できる必要がある。
- 開発にかかる時間とコストを削減したり、一般的な共有のタスクやサービスを使用する複雑なアプリケーションで、バグを最小限に抑える必要がある。

考慮事項

Policy Injection Application Block を使用する際には、次の事項を考慮します。

- このアプリケーション ブロックでは、コードを直接メソッドに挿入するのではなく、インターセプトを使用して前処理と後処理のハンドラーのみを有効にします。

- クラス コンストラクターではインターセプトを使用できません。
- アプリケーション ブロックが処理要件を可能な限り最小限に抑えるように設計されていますが、すべてのインターセプト技術と同様、アプリケーションには追加の要件が課されます。
- Call ハンドラーは呼び出しメッセージの情報にしかアクセスできず、内部状態を管理することはできません。
- ポリシーの挿入は、対象のクラスのパブリック メンバーに対してのみ実行できます。

Security Application Block

Security Application Block を使用すると、ユーザーの承認と認証に関するデータをキャッシュしたり、Microsoft .NET Framework のセキュリティ機能と統合するなど、一般的な承認関連の機能を簡単に実装できます。

主要なシナリオ

Security Application Block は、次のような状況に直面した場合に適しています。

- 承認の実行に使用する、セキュリティ関連の資格情報をキャッシュしている。
- 認証されたユーザーに関する一時的なトークンを取得し、トークンを使用してユーザーを認証している。
- ユーザー セッションを終了している (トークンを有効期限切れにしている)。
- タスクを実行する際に、ユーザーに権限があるかどうかを判断している。

使用すべき状況

Security Application Block は次の要件に対処するのに最適です。

- 資格情報のキャッシュ、認証の確認など、標準的なセキュリティ関連のタスクを実行するのに使用する、ひな型のコードの要件を減らす必要がある。
- アプリケーションと社内の両方で、一貫したセキュリティのノウハウを保持する必要がある。
- 利用するさまざまな領域の機能で一貫したアーキテクチャ モデルを使用することで、開発者が習得しなければならないことを最小限に抑える必要がある。
- セキュリティ プロバイダーのカスタム実装を使用する必要がある。

考慮事項

Security Application Block を使用する際には、次の事項を考慮します。

- キャッシュされるセキュリティ関連情報の既定のストアは、Caching Application Block です。Caching Application Block では、バッキング ストアのキャッシュ データを暗号化するように構成できますが、メモリ内のキャッシュ データの暗号化はサポートしていません。このことがアプリケーションにとって大きな脅威となる場合は、メモリ内の暗号化をサポートしている別のカスタムのキャッシュ ストアのプロバイダーを使用できます。
- 承認マネージャーは部分的に信頼された状態ではサポートされません。

Unity Application Block

Unity Application Block は、オブジェクト インターセプト、コンストラクターの挿入、プロパティの挿入、およびメソッド呼び出しの挿入をサポートする、軽量で拡張可能な依存関係の挿入のコンテナーです。また、Enterprise Library と併用して、Enterprise Library オブジェクトと独自のカスタム ビジネス オブジェクトを生成できます。

主要なシナリオ

Unity Application Block は、次のような状況に直面した場合に適しています。

- コンストラクター、プロパティ、およびメソッド呼び出しの挿入をサポートし、オブジェクト インスタンスの有効期限を管理できるコンテナーを通じて、依存関係の挿入を実行している。
- 他のオブジェクトやクラスと依存関係があり、その依存関係が複雑であったり抽象化を必要としているクラスに対して、依存関係の挿入を実行している。
- 実行時に依存関係を構成したり変更している。
- Web アプリケーションのポスト バック間でコンテナーをキャッシュしたり保持している。

使用すべき状況

Unity Application Block は次の要件に対処するのに最適です。

- オブジェクトを簡単に作成できる必要がある (特に階層構造になっているオブジェクトの構造や依存関係を簡略化する必要がある)。このようにすると、アプリケーション コードも簡略化されます。

- 横断的関心事の管理を簡略化するため、実行時または構成で依存関係を指定して、要件を抽象化する必要がある。
- コンポーネントの構成をコンテナーで行うことで、柔軟性を向上する必要がある。
- クライアントがコンテナーを格納したりキャッシュしたりできる、サービスの場所に関する機能が必要である。これは特に ASP.NET Web アプリケーションで役に立ち、開発者は ASP.NET セッションまたはアプリケーションでコンテナーを保持できます。

Unity Application Block は、次のような状況では使用しないようにします。

- オブジェクトとクラスが他のオブジェクトやクラスと依存関係がないか、依存関係が非常に単純で抽象化の必要がない。

考慮事項

Unity Application Block を使用する際には、次の事項を考慮します。

- 依存関係の挿入がパフォーマンスにわずかに影響する可能性があります。
- 単純な依存関係しか存在しない場合に依存関係の挿入を使用すると、複雑さが増す可能性があります。

Validation Application Block

Validation Application Block では、属性とルール セットを使用する構造化されて管理しやすい検証メカニズムを実装したり、多数のアプリケーション インターフェイスのテクノロジと統合するための、さまざまな機能を提供しています。

主要なシナリオ

Validation Application Block は、次のような状況に直面した場合に適しています。

- フィールド、プロパティ、および入れ子になったオブジェクトを検証したり、悪意のあるデータがアプリケーションに挿入されないようにする、構造化されて管理しやすい検証コードを実装しています。
- ビジネス ルールを強化して、ユーザー入力に応答を提供している。
- 同じアプリケーション内で同じ規則を使用してデータを複数回検証している。

- さまざまな構築済みの検証コントロールを組み合わせて、複雑なシナリオやさまざまな機能をサポートしている。

使用すべき状況

Validation Application Block は次の要件に対処するのに最適です。

- ASP.NET、Windows フォーム、および WCF アプリケーションの標準的な .NET のデータ型の大半について、一貫した検証のノウハウを保持する必要がある。
- 構成、属性、およびコードを使用して、検証規則を作成する必要がある。
- 複数のルール セットを、同じクラスやクラスのメンバーと関連付ける必要がある。
- オブジェクトを検証するときに、1 つ以上のルール セットを適用したり、ビジネス検証ロジックを再利用する必要がある。

考慮事項

Validation Application Block を使用する際には、次の事項を考慮します。

- ASP.NET、Windows フォームなどの一部のテクノロジでは、組み込みの検証機能を使用できます。そのため、これらのテクノロジにのみ検証ロジックを適用する必要がある場合、その検証ロジックを再利用する必要がない限り、このアプリケーション ブロックを使用する必要はありません。
- WCF アプリケーションと XML データを使用する他のアプリケーションでは、XML スキーマを使用して XML レベルでメッセージを検証できます。これらのテクノロジにのみ検証ロジックを適用する必要がある場合、その検証ロジックを再利用する必要がない限り、このアプリケーション ブロックを使用する必要はありません。
- 少しのオブジェクトしか検証する必要がないという非常に単純なシナリオでは、アプリケーション ブロックを追加して、オーバーヘッドを負うことはお勧めしません。

関連情報

Web リソースに簡単にアクセスするには、http://www.microsoft.com/architectureguide (英語) でオンライン版の参考文献を参照してください。

- Enterprise Library
  (http://msdn.microsoft.com/en-us/library/cc467894.aspx、英語)
- The Caching Application Block
  (http://msdn.microsoft.com/en-us/library/cc511588.aspx、英語)
- The Cryptography Application Block
  (http://msdn.microsoft.com/en-us/library/cc511721.aspx、英語)
- The Data Access Application Block
  (http://msdn.microsoft.com/en-us/library/cc511547.aspx、英語)
- Exception Handling Application Block
  (http://msdn2.microsoft.com/en-us/library/aa480461.aspx、英語)
- The Logging Application Block
  (http://msdn.microsoft.com/en-us/library/cc511708.aspx、英語)
- The Policy Injection Application Block
  (http://msdn.microsoft.com/en-us/library/cc511729.aspx、英語)
- The Security Application Block
  (http://msdn.microsoft.com/en-us/library/cc511928.aspx、英語)
- The Unity Application Block
  (http://msdn.microsoft.com/en-us/library/cc511654.aspx、英語)
- The Validation Application Block
  (http://msdn.microsoft.com/en-us/library/cc511802.aspx、英語)
- Microsoft Enterprise Library Frequently Asked Questions
  (http://www.codeplex.com/entlib/Wiki/View.aspx?title=EntLib%20FAQ、英語)

付録 G

patterns & practices パターン カタログ

Composite Application Guidance for WPF and Silverlight

カテゴリ	パターン
モジュール方式	**Service Locator** - サービスへの参照を含み、サービスを検索するロジックをカプセル化するサービス ロケーターを作成します。クラスでは、サービスのインスタンスを取得するためにサービス ロケーターを使用します。詳細については、「Service Locator」(http://msdn.microsoft.com/en-us/library/dd458903.aspx、英語) を参照してください。
テスト容易性	**Dependency Injection** - クラスでは、依存関係を明示的にインスタンス化するのではなく、クラス定義で依存関係を宣言によって表します。また、Builder オブジェクトを使用して、オブジェクトの依存関係の有効なインスタンスを取得し、オブジェクトの作成時か初期化時、またはその両方で、インスタンスをオブジェクトに渡します。詳細については、「Dependency Injection」(http://msdn.microsoft.com/en-us/library/dd458879.aspx、英語) を参照してください。 **Inversion of Control** - クラスの依存関係の具体的な実装型を選択する機能を、外部コンポーネントか外部ソースに委任します。詳細については、「Inversion of Control」(http://msdn.microsoft.com/en-us/library/dd458907.aspx、英語) を参照してください。 **Separated Presentation** - プレゼンテーション ロジックとビジネス ロジックを別々の成果物に分離します。Separated Presentation パターンは、

	Supervising Presenter パターンや Presentation Model パターンなど、複数の方法で実装できます。詳細については、「Separated Presentation」(http://msdn.microsoft.com/en-us/library/dd458859.aspx、英語) を参照してください。 **Presentation Model** - 視覚的な表示に関する役割とユーザー インターフェイス (UI) の状態や動作に関する役割を、ビューおよびプレゼンテーション モデルという別々のクラスに分離します。ビュー クラスは、UI のコントロールを管理します。一方、プレゼンテーション モデル クラスは、モデルへのアクセスをカプセル化して、ビューから簡単に使用できるパブリック インターフェイスを提供する (たとえば、データ バインドを使用する) ことで、UI 固有の状態や動作を持つモデルのファサードとして機能します。詳細については、「Presentation Model」(http://msdn.microsoft.com/en-us/library/dd458863.aspx、英語) を参照してください。 **Supervising Presenter** (または Supervising Controller) - 視覚的な表示に関する役割とイベント処理の動作に関する役割を、ビューおよびプレゼンターという別々のクラスに分離します。ビュー クラスでは、UI のコントロールを管理して、ユーザー イベントをプレゼンター クラスに転送します。一方、プレゼンター クラスは、イベントに応答するロジックを含み、モデル (アプリケーションのビジネス ロジックとデータ) を更新することでビューの状態を操作します。詳細については、「Supervising Presenter」(http://msdn.microsoft.com/en-us/library/dd490821.aspx、英語) を参照してください。

Data Movement パターン

カテゴリ	パターン
Data Movement パターン	**Data Replication** - 2 つの場所の間でデータを移動する、レプリケーション セットやレプリケーション リンクを作成します。Data Replication パターンは、より詳細な (この表で説明している) Data Movement パターンの一般的な処理を表す大まかなパターンです。詳細については、「Data Replication」(http://msdn.microsoft.com/en-us/library/ms978671.aspx、英語) を参照してください。 **Master-Master Replication** - データをソースからターゲットにコピーします。また、前回のレプリケーション以降に (ソースとターゲットに共通しているデータに加えられた変更によって) 発生した更新の競合を検出して解決します。

	ソリューションは、ターゲットとソース間の 2 つの逆方向のレプリケーション リンクで構成されています。どちらのレプリケーション リンクでも、両方向に同じレプリケーション セットを送信します。このような 1 対のレプリケーション リンクを "関連リンク" といいます。詳細については、「Master-Master Replication」(http://msdn.microsoft.com/en-us/library/ms978735.aspx、英語) を参照してください。 **Master-Subordinate Replication** - 前回のレプリケーション以降、ターゲットのレプリケーション セットで更新が発生したかどうかにかかわらず、データをソースからターゲットにコピーします。詳細については、「Master-Subordinate Replication」(http://msdn.microsoft.com/en-us/library/ms978740.aspx、英語) を参照してください。 **Master-Master Row-Level Synchronization** - ソースとターゲット間の 1 対の関連レプリケーション リンクを使用し、同期コントローラーを使用して、同期を両方向で管理します。複数のレプリケーション セットのコピーを同期するには、2 つ目以降のコピーに、1 対の適切なレプリケーション リンクをそれぞれ作成します。詳細については、「Master-Master Row-Level Synchronization」(http://msdn.microsoft.com/en-us/library/ms998434.aspx、英語) を参照してください。 **Master-Subordinate Snapshot Replication** - 特定の時点の (スナップショットといいます) ソース レプリケーション セットのコピーを作成して、それをターゲットに複製し、ターゲットのデータを上書きします。このようにすることで、ターゲット レプリケーション セットで発生した変更は、ソース レプリケーション セットのデータで新しく置き換えられます。詳細については、「Master-Subordinate Snapshot Replication」(http://msdn.microsoft.com/en-us/library/ms998430.aspx、英語) を参照してください。 **Capture Transaction Details** - トリガーや (シャドウ) テーブルなどのデータベース オブジェクトを追加で作成し、当該のレプリケーション セットに属するすべてのテーブルに加えられた変更を記録します。詳細については、「Capture Transaction Details」(http://msdn.microsoft.com/en-us/library/ms978709.aspx、英語) を参照してください。 **Master-Subordinate Transactional Incremental Replication** - コミットされたトランザクションに関する情報をソースから取得し、その情報をターゲットに書き込んだら、トランザクションを適切な順序で再生します。詳細については、「Master-Subordinate Transactional Incremental Replication」(http://msdn.microsoft.com/en-us/library/ms998441.aspx、英語) を参照

	してください。 **Master-Subordinate Cascading Replication** - ソース データベースとターゲット データベース間に、1 つ以上の中間ターゲットを追加して、ソースとターゲット間のレプリケーション リンク数を増やします。この中間に追加したターゲットは、レプリケーション セットをソースから取得するデータ ストアなので、最初のレプリケーション リンクのターゲットとして機能します。その後、中間ターゲットは、リンクされた最後のターゲットに到達するまで、データを次のレプリケーション リンクに移動するソースとして機能します。詳細については、「Master-Subordinate Cascading Replication」(http://msdn.microsoft.com/en-us/library/ms978712.aspx、英語) を参照してください。
パトレット	**Maintain Data Copies -** アプリケーションからデータのコピーに同時に書き込むか、非同期のサービスによって後で移動されるように、ローカル キャッシュにデータを同時に投稿します。詳細については、「Patterns and Pattlets」(http://msdn.microsoft.com/en-us/library/ms998465.aspx、英語) を参照してください。 **Application-Managed Data Copies** - 特定のアプリケーションによってデータのコピーが変更されたときには、他のコピーも変更する必要があります。アプリケーションでは、データのコピーか派生したデータ、またはその両方を、元のデータを変更したのと同じトランザクションで更新する必要があります。詳細については、「Patterns and Pattlets」(http://msdn.microsoft.com/en-us/library/ms998465.aspx、英語) を参照してください。 **Extract-Transform-Load** - 異なるソースからデータを取得するために、複雑なクエリを実行するデータ移動の種類です。集計と浄化を含む複雑な操作を適用しますが、ターゲットで発生した変更を置き換える単純な書き込みを実行します。詳細については、「Patterns and Pattlets」(http://msdn.microsoft.com/en-us/library/ms998465.aspx、英語) を参照してください。 **Topologies for Data Copies** - 複数のプラットフォームにデータのコピーを配置するアーキテクチャのアプローチです。詳細については、「Patterns and Pattlets」(http://msdn.microsoft.com/en-us/library/ms998465.aspx、英語) を参照してください。

エンタープライズ ソリューション パターン

カテゴリ	パターン
配置のパターン	**Deployment Plan** - 各アプリケーションのコンポーネントを配置するティアを示す配置計画を作成します。コンポーネントをティアに割り当てるとき、ティアがコンポーネントに適合しないことが判明した場合は、インフラストラクチャを正常に操作できるようにコンポーネントを変更することか、コンポーネントに適合するようにインフラストラクチャを変更することのコストとメリットを判断します。詳細については、「配置計画」(http://msdn.microsoft.com/ja-jp/library/ms978676.aspx) を参照してください。 **Layered Application** - ソリューションのコンポーネントをレイヤーに分離します。各レイヤーのコンポーネントは、まとまりがあり、抽象化のレベルがほぼ同じでなければなりません。また、各レイヤーは、下位層のレイヤーと疎結合されている必要があります。詳細については、「階層化アプリケーション」(http://msdn.microsoft.com/ja-jp/library/ms978678.aspx) を参照してください。 **Three-Layered Services Application** - 分離を提供してまとまりを強化するために、レイヤー型アーキテクチャを、プレゼンテーション レイヤー、ビジネス レイヤー、およびデータ レイヤーの 3 つのレイヤーで構築します。詳細については、「3 層サービス アプリケーション」(http://msdn.microsoft.com/ja-jp/library/ms978689.aspx) を参照してください。 **Tiered Distribution** - サーバーとクライアント コンピューターを一連の物理ティアに構築します。また、アプリケーション コンポーネントを特定のティアに適切に分散します。詳細については、「多階層分散」(http://msdn.microsoft.com/ja-jp/library/ms978701.aspx) を参照してください。 **Three-Tiered Distribution** - クライアント ティア、アプリケーション ティア、およびデータベース ティアの 3 つの物理ティアを中心にアプリケーションを構築します。詳細については、「3 層分散」(http://msdn.microsoft.com/ja-jp/library/ms978694.aspx) を参照してください。
分散システム	**Broker** - Broker パターンを使用すると、ビジネス コンポーネント自体とは異なるレイヤーに、リモート サービスを呼び出す実装の詳細をカプセル化して、その詳細を隠すことができます。詳細については、「ブローカー」

	(http://msdn.microsoft.com/ja-jp/library/ms978706.aspx) を参照してください。 **Data Transfer Object** - リモート呼び出しに必要なすべてのデータを保持する、データ転送オブジェクト (DTO) を作成します。DTO を 1 つのパラメーターとして受け取り、クライアントに 1 つの DTO パラメーターを返すには、リモート メソッドの署名を変更します。アプリケーションの呼び出しが DTO を受け取って、ローカル オブジェクトとして保存したら、アプリケーションでは DTO への一連のプロシージャ呼び出しを実行できます。ただし、この呼び出しにより、リモート呼び出しのオーバーヘッドが発生することはありません。詳細については、「データ転送オブジェクト」(http://msdn.microsoft.com/ja-jp/library/ms978717.aspx) を参照してください。 **Singleton** - シングルトンは、クラスでそのクラス自体の単一インスタンスが作成されるようにして、グローバルな単一インスタンスを提供します。また、インスタンスへの参照を返す、グローバルにアクセスできるクラス メソッドを使用して他のオブジェクトがこのインスタンスにアクセスできるようにします。また、他のオブジェクトが新しいインスタンスを作成できないように、クラス コンストラクターを private として宣言します。詳細については、「シングルトン」(http://msdn.microsoft.com/ja-jp/library/ms998426.aspx) を参照してください。
パフォーマンスと信頼性	**Server Clustering** - サーバー クラスターは、単一に見えるように相互に接続された複数台のサーバーを組み合わせたもので、可用性かスケーラビリティ、またはその両方を向上する仮想リソースを作成します。詳細については、「サーバー クラスタリング」(http://msdn.microsoft.com/ja-jp/library/ms998414.aspx) を参照してください。 **Load-Balanced Cluster** - 負荷を共有するように構成された複数のサーバーに、サービスまたはアプリケーションをインストールします。このような構成を、負荷分散クラスターと言います。負荷分散では、クライアント要求を複数のサーバーに分散することで、サーバー ベースのプログラム (Web サーバーなど) のパフォーマンスが強化されます。この負荷分散テクノロジ (通称、負荷分散装置) では、受信要求を受け取ると、必要に応じて特定のホストにリダイレクトします。負荷分散するように構成された複数のホストでは、同じクライアントから複数の要求が行われた場合でも、同時に異なるクライアント要求に応答します。詳細については、「負荷分散クラスタ」(http://msdn.microsoft.com/ja-jp/library/ms978730.aspx) を参照してください。 **Failover Cluster** - フェールオーバー クラスターは、あるサーバーが使用でき

	なくなると、障害が発生したサーバーの処理が別のサーバーに自動的に引き継がれて処理が続行されるように構成された一連のサーバーです。クラスター内の各サーバーには、スタンバイ サーバーとして認識されているサーバーが少なくとも 1 台あります。詳細については、「フェールオーバー クラスタ」(http://msdn.microsoft.com/ja-jp/library/ms978720.aspx) を参照してください。
サービスのパターン	**Service Interface** - アプリケーションのコンシューマーがサービスと通信するのに使用できるエントリ ポイントを提供し、ビジネス ロジックから実装を分離しながら、粒度の荒いインターフェイスを公開するコンポーネントを作成します。詳細については、「サービス インターフェイス」(http://msdn.microsoft.com/ja-jp/library/ms998421.aspx) を参照してください。 **Service Gateway** - コントラクトのコンシューマーの部分を実装するコードを、他のサービスへのプロキシとして機能するコンシューマー独自の Service Gateway コンポーネントにカプセル化して、ソースへの接続の詳細をカプセル化して、必要な変換を実行します。詳細については、「サービス ゲートウェイ」(http://msdn.microsoft.com/ja-jp/library/ms998420.aspx) を参照してください。
Web プレゼンテーションのパターン	**Model-View-Controller -** Model-View-Controller (MVC) パターンは、ユーザーの入力に基づいて、ドメイン、プレゼンテーション、およびアクションのデータを 3 つの別個のクラスに分離します。Model では、アプリケーション ドメインの動作とデータを管理し、その状態に関する情報を求める (通常、View からの) 要求に応答して、状態を変更するための (通常、Controller からの) 命令に応答します。View では、情報の表示を管理します。Controller では、ユーザーからのマウスとキーボードによる入力を解釈し、必要に応じて、Model と View に変更を通知します。詳細については、「モデル ビュー コントローラ」(http://msdn.microsoft.com/ja-jp/library/ms978748.aspx) を参照してください。 **Page Controller** - Page Controller パターンは、ページ要求からの入力を受け付け、モデルで要求された操作を呼び出し、結果のページに使用する適切なビューを決定するのに使用します。また、ディスパッチ ロジックを、ビューに関連するコードから分離します。また、必要に応じて、すべてのページ コントローラーに共通の基本クラスを作成し、コードが重複しないようにして、一貫性とテスト容易性を向上します。詳細については、「ページ コントローラ」(http://msdn.microsoft.com/ja-jp/library/ms978764.aspx) を参照してくだ

	さい。 **Front Controller** - Front Controller パターンは、単一のコントローラーを通じてすべての要求を渡すことで、Page Controller パターンに存在している分散の問題を解決します。通常、コントローラー自体は、ハンドラーおよびコマンドの階層の 2 つの部分に実装されます。ハンドラーでは、Web サーバーから HTTP Post 要求か HTTP Get 要求を受け取り、その要求から関連するパラメーターを取得します。ハンドラーでは、まず要求から取得したパラメーターを使用して適切なコマンドを選択し、それから制御をコマンドに転送して処理します。コマンド自体もコントローラーの一部で、 コマンドは Command パターンで表される特定の操作を表します。詳細については、「フロント コントローラ」(http://msdn.microsoft.com/ja-jp/library/ms978723.aspx) を参照してください。 **Intercepting Filter** - Intercepting Filter パターンは、一連の構成可能なフィルターを作成して、Web ページ要求の一般的な処理と処理後のタスクを実装するのに使用します。詳細については、「受信フィルタ」(http://msdn.microsoft.com/ja-jp/library/ms978727.aspx) を参照してください。 **Page Cache** - サーバーの処理負荷を減らすために、頻繁にアクセスされるが、あまり変更されないページについては、サーバーで生成された出力をキャッシュします。詳細については、「ページ キャッシュ」(http://msdn.microsoft.com/ja-jp/library/ms978759.aspx) を参照してください。 **Observer** - Observer パターンは、別のオブジェクト (サブジェクト) で関連のある依存関係 (オブザーバー) の一覧を管理するために使用します。個々のオブザーバーに共通の Observer インターフェイスを実装することで、サブジェクトおよび依存関係があるオブジェクト間の直接的な依存関係がなくなります。詳細については、「オブザーバ」(http://msdn.microsoft.com/ja-jp/library/ms978753.aspx) を参照してください。

統合パターン

カテゴリ	パターン
統合レイヤー	**Entity Aggregation** - エンタープライズ レベルでエンティティを論理的に表現する、Entity Aggregation レイヤーを導入します。また、物理的な接続を提供することで、アク

	セスをサポートし、バックエンド レポジトリにある、エンティティの論理表現の各インスタンスを更新します。詳細については、「Entity Aggregation」(http://msdn.microsoft.com/en-us/library/ms978573.aspx、英語) を参照してください。 **Process Integration** - 複雑なビジネス機能を構成する個々の手順を表す、ビジネス プロセス モデルを定義します。また、このビジネス プロセス モデルの複数の同時インスタンスを解釈して、既存のアプリケーションと連携してプロセスの個々の手順を実行できる、別個のプロセス マネージャー コンポーネントを作成します。詳細については、「Process Integration」(http://msdn.microsoft.com/en-us/library/ms978592.aspx、英語) を参照してください。 **Portal Integration** - 複数のアプリケーションから取得した情報を統一された UI で表示する、ポータル アプリケーションを作成します。ユーザーは、このポータルの情報に基づいて、必要なタスクを実行できます。詳細については、「Portal Integration」(http://msdn.microsoft.com/en-us/library/ms978585.aspx、英語) を参照してください。
統合トポロジ	**Message Broker** - Message Broker パターンを使用すると、統合ソリューションを拡張できます。メッセージ ブローカーは、アプリケーション間の通信を処理する物理コンポーネントです。アプリケーションは、相互に通信するのではなく、メッセージ ブローカーとのみ通信します。アプリケーションがメッセージ ブローカーにメッセージを送信し、受信者の論理名を提供したら、メッセージ ブローカーは論理名で登録されたアプリケーションを参照して、メッセージをそのアプリケーションに渡します。詳細については、「Message Broker」(http://msdn.microsoft.com/en-us/library/ms978579.aspx、英語) を参照してください。 **Message Bus** - メッセージ バスという論理コンポーネントを使用してすべてのアプリケーションを接続します。メッセージ バスは、アプリケーション間でのメッセージの転送に特化しています。メッセージ バスには、3 つの重要な要素が含まれています。その要素は、一連の合意に基づいたメッセージ スキーマ、一連の一般的なコマンド メッセージ、および受信者にバス メッセージを送信する共有インフラストラクチャです。詳細については、「Message Bus」(http://msdn.microsoft.com/en-us/library/ms978583.aspx、英語) を参照してください。 **Publish/Subscribe** - 特定のメッセージを受信するために他のアプリケーションがサブスクライブできるイベントをクラスが発行できるようにします。Publish/Subscribe メカニズムは、すべての関連するサブスクライバーにイベントまたはメッセージを送信します。詳細については、「Publish/Subscribe」(http://msdn.microsoft.com/en-us/library/ms978603.aspx、英語) を参照してください。

システム接続	**Data Integration** - あるアプリケーション (ソース) のデータが別のアプリケーション (ターゲット) でアクセスできるようにすることで、アプリケーションを論理データ レイヤーで統合します。詳細については、「Data Integration」(http://msdn.microsoft.com/en-us/library/ms978572.aspx、英語) を参照してください。 **Functional Integration** - あるアプリケーション (ソース) のビジネス機能が、別のアプリケーション (ターゲット) でアクセスできるようにすることで、アプリケーションをビジネス論理レイヤーで統合します。詳細については、「Functional Integration」(http://msdn.microsoft.com/en-us/library/ms978578.aspx、英語) を参照してください。 **Service-Oriented Integration** - アプリケーションをビジネス論理レイヤーで統合するために、システムで拡張マークアップ言語 (XML) ベースの Web サービスを使用および提供できるようにします。また、Web サービス記述言語 (WSDL) コントラクトを使用して、そのシステムへのインターフェイスを表します。そして、実装を Web サービス (WS-*) の仕様群に準拠するようにして、相互運用性を確保します。詳細については、「Service-Oriented Integration」(http://msdn.microsoft.com/en-us/library/ms978594.aspx、英語) を参照してください。 **Presentation Integration** - ユーザーの入力をシミュレートして、画面表示からデータを読み取ることで、UI からアプリケーションの機能にアクセスします。詳細については、「Presentation Integration」(http://msdn.microsoft.com/en-us/library/ms978588.aspx、英語) を参照してください。
その他の 統合パターン	**Pipes and Filters** - 各フィルター コンポーネントが、入力メッセージを受け取り、簡単な変換を適用して、次のコンポーネントに変換したメッセージを送信する一連のフィルター コンポーネントを使用して、変換を実装します。フィルターの入出力を接続し、フィルター間の通信をバッファーするパイプを通してメッセージを伝達します。詳細については、「Pipes and Filters」(http://msdn.microsoft.com/en-us/library/ms978599.aspx、英語) を参照してください。 **Gateway** - 外部システムへのアクセスを 1 つのインターフェイスに抽象化します。Gateway パターンは、外部システムに接続する方法を把握するために複数のシステムを必要としないので、 外部システムへのアクセスと関連している、開発プロセスと管理プロセスを簡略化します。詳細については、「Additional Integration Patterns」(http://msdn.microsoft.com/en-us/library/ms978722.aspx、英語) を参照してください。

Web Services Security パターン

カテゴリ	パターン
認証	**Brokered Authentication** - Web サービスが、クライアントとの直接的な関係がなくても、クライアントで提示される資格情報を検証します。Web サービスとクライアントの両方で信頼される認証ブローカーは、セキュリティ トークンをクライアントに発行します。その結果、クライアントが、セキュリティ トークンなどの資格情報を Web サービスに提示できるようになります。詳細については、「Brokered Authentication」(http://msdn2.microsoft.com/en-us/library/aa480560.aspx、英語) を参照してください。次の 3 つのパターンは、Brokered Authentication パターンに固有の実装を表します。 **Brokered Authentication: Kerberos** - Kerberos プロトコルは、クライアントと Web サービス間の認証を仲介するのに使用します。詳細については、「Brokered Authentication: Kerberos」(http://msdn2.microsoft.com/en-us/library/aa480562.aspx、英語) を参照してください。 **Brokered Authentication: X509 PKI** - 公開キーのインフラストラクチャ (PKI) の証明機関 (CA) によって発行された、X.509 証明書で仲介された認証を使用して、要求元のアプリケーションから提示された資格情報を検証します。詳細については、「Brokered Authentication: X.509 PKI」(http://msdn2.microsoft.com/en-us/library/aa480565.aspx、英語) を参照してください。 **Brokered Authentication: STS** - 仲介された認証を、セキュリティ トークン サービス (STS) で発行されたセキュリティ トークンと共に使用します。STS は、相互運用性のあるセキュリティ トークンを提供するものとして、クライアントと Web サービスの両方で信頼されています。詳細については、「Brokered Authentication: Security Token Service (STS)」(http://msdn2.microsoft.com/en-us/library/aa480563.aspx、英語) を参照してください。 **Direct Authentication** - Web サービスが認証サービスとして機能し、クライアントの資格情報を検証します。共有シークレットに基づいた所有の証明を含む資格情報は、ID ストアに対して検証されます。詳細については、「Direct Authentication」(http://msdn.microsoft.com/en-us/library/aa480566.aspx、英語) を参照してください。
承認	**Trusted Subsystem** - Web サービスが信頼されたサブシステムとして機能して、追加のリソースにアクセスします。リソースにアクセスする際には、ユーザーの資格情報ではなくアプリケーション自体の資格情報を使用します。詳細については、「Trusted

	Subsystem」(http://msdn2.microsoft.com/en-us/library/aa480587.aspx、英語) を参照してください。
例外管理	**Exception Shielding** - 設計が安全な例外と置き換えることで、安全でない例外の一部を削除します。そして、一部を削除した例外または設計が安全な例外のみをクライアントに返します。設計が安全な例外では、例外メッセージに機密情報が含まれません。また、そのいずれかが Web サービスの内部の動作についての機密情報を開示する場合がある詳細なスタック トレースも含まれません。詳細については、「Exception Shielding」(http://msdn2.microsoft.com/en-us/library/aa480591.aspx、英語) を参照してください。
メッセージの 暗号化	**Data Confidentiality** - 暗号化を使用して、メッセージに含まれる機密データを保護します。暗号化されていないデータ (通称、プレーンテキスト) は、暗号化されたデータ (通称、暗号化テキスト) に変換されます。データは、アルゴリズムと暗号化キーで暗号化されます。暗号化テキストは、最終的には再びプレーンテキストに変換されます。詳細については、「Data Confidentiality」(http://msdn.microsoft.com/en-us/library/aa480570.aspx、英語) を参照してください。
メッセージ リプレイ検出	**Message Replay Detection** - 受信メッセージの識別子をキャッシュします。また、メッセージ リプレイ検出を使用して、リプレイ検出キャッシュのエントリと一致するメッセージを特定して拒否します。詳細については、「Message Replay Detection」(http://msdn2.microsoft.com/en-us/library/aa480598.aspx、英語) を参照してください。
メッセージの署名	**Data Origin Authentication** - メッセージが伝送中に改ざんされていないこと (データ整合性) と、予期している送信者から送信されていること (信頼性) を受信者が確認できる 、データ送信元の認証を使用します。詳細については、「Data Origin Authentication」(http://msdn2.microsoft.com/en-us/library/aa480571.aspx、英語) を参照してください。
メッセージの検証	**Message Validator** - メッセージの検証ロジックは、サービスが要求メッセージを正常に処理するために必要な要求メッセージの部分を特定する、明確に定義されたポリシーを適用します。XML スキーマ定義 (XSD) に対して XML メッセージ ペイロードを検証して、メッセージが整形式になっていて、サービスで処理されることが想定されているものと一致していることを確認します。また、検証ロジックでは、メッセージのサイズ、メッセージの内容、使用されている文字を調べ、特定の条件に従ってメッセージを評価します。条件を満たさないメッセージは拒否されます。詳細については、「Message Validator」(http://msdn2.microsoft.com/en-us/library/aa480600.aspx、英語) を参照してください。

配置	**Perimeter Service Router** - Web サービスを、境界サービス ルーターとして機能するように、中間デバイスで設計します。境界サービス ルーターは、内部の Web サービスに、境界ネットワーク上にある外部のインターフェイスを提供します。また、外部のアプリケーションからメッセージを受け取り、そのメッセージをプライベート ネットワーク上の適切な Web サービスにルーティングします。詳細については、「Perimeter Service Router」(http://msdn2.microsoft.com/en-us/library/aa480606.aspx、英語) を参照してください。

関連情報

Web リソースに簡単にアクセスするには、http://www.microsoft.com/architectureguide (英語) でオンライン版の参考文献を参照してください。

- Composite Application Library のパターンの詳細については、「Composite Application Guidance for WPF and Silverlight」(英語) を参照してください。
- データ パターンの詳細については、「Data Patterns」(英語) を参照してください。
- エンタープライズ ソリューション パターンの詳細については、「Microsoft .NET を使用したエンタープライズ ソリューション パターン」を参照してください。
- 統合パターンの詳細については、「Integration Patterns」(英語) を参照してください。
- Web Service Security の詳細については、「Web Service Security Patterns - Community Technical Preview」(英語) を参照してください。


Microsoft®