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。

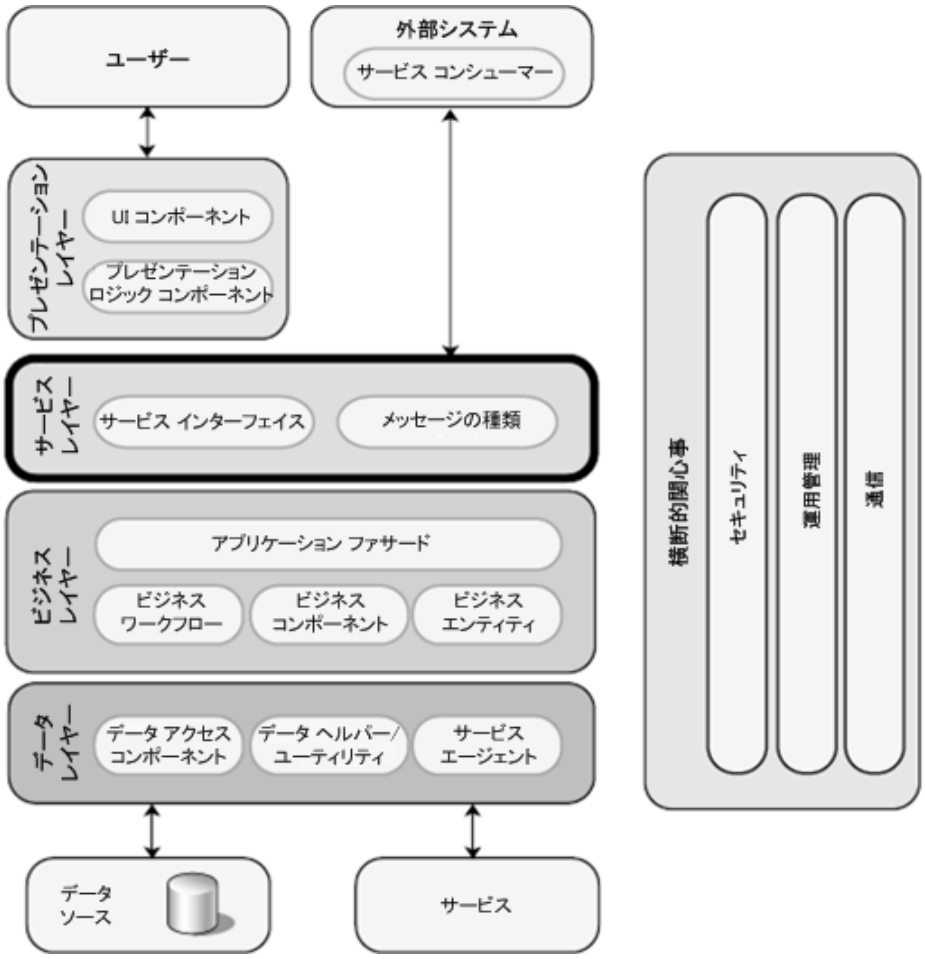

図 1

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)