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。

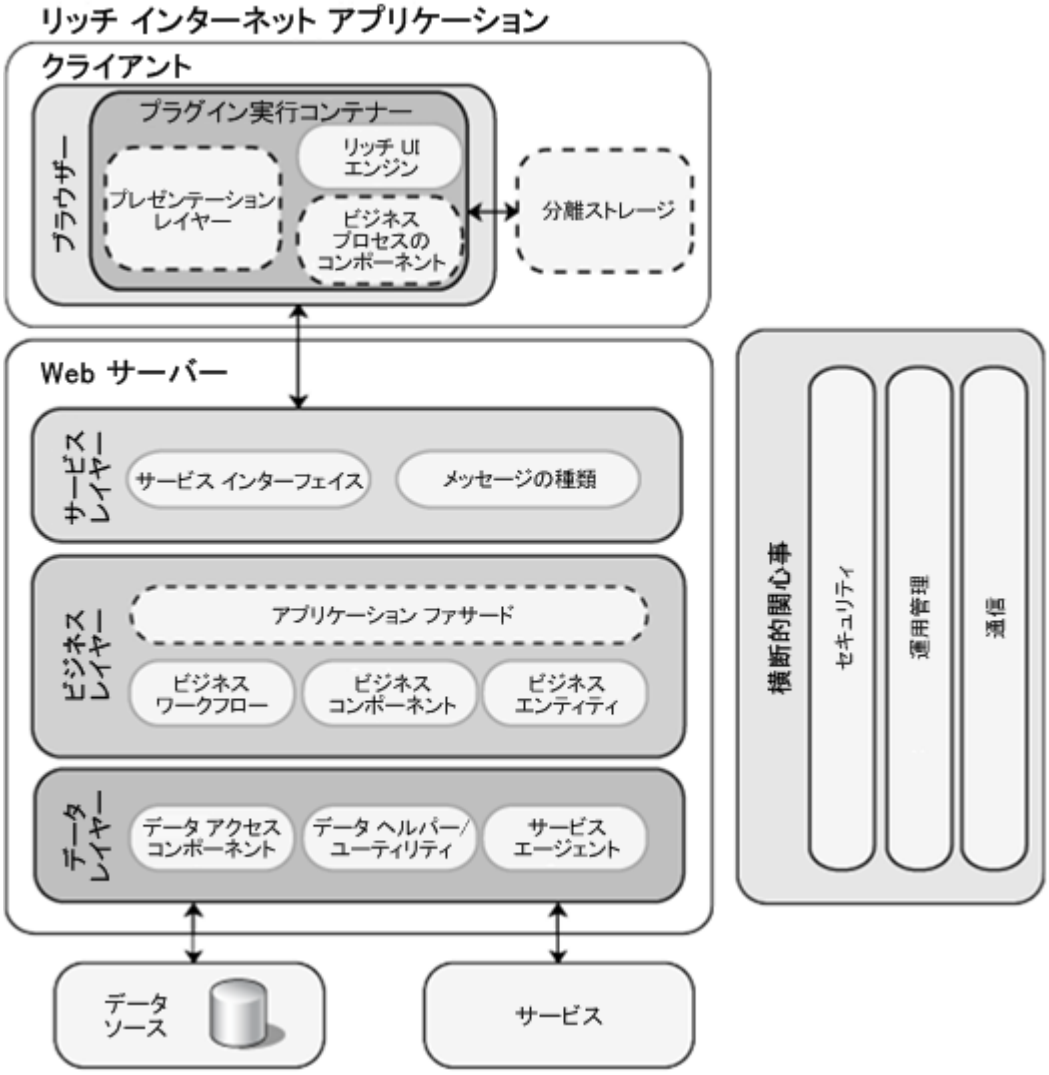

図 1

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)。

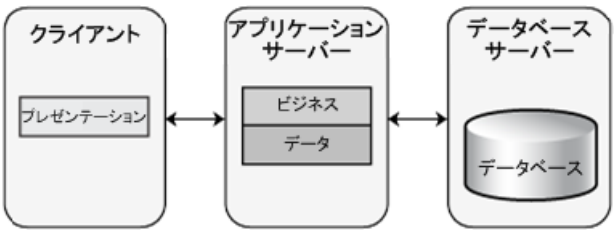

図 2

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。

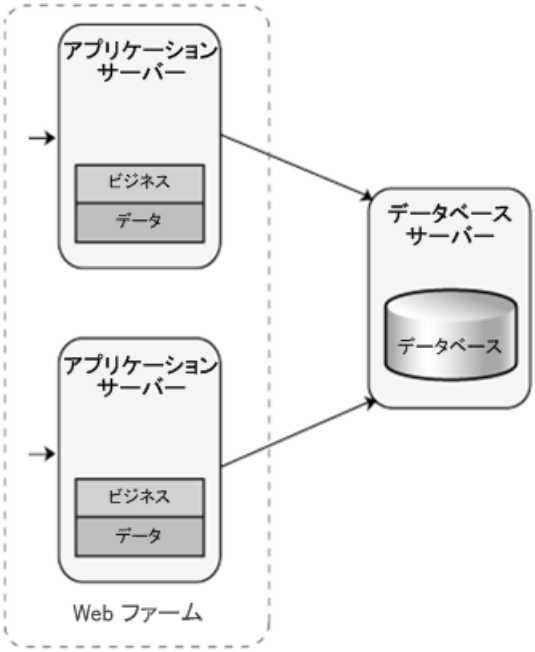

図 3

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。