ビジネス アプリケーション向け .NET テクノロジ ガイド

2013 年 6 月　Microsoft Corporation

2013 年 9 月　日本語版 Rev.1

Microsoft .NET

目次

日本語版 (Rev.1) では、ドキュメントに含まれる図は英語表記となっています。ご了承ください。

1. 重要ポイント

アプリケーションに固有の優先事項と要件に基づいて、アーキテクチャのアプローチや開発テクノロジを選択する

あらゆる種類のアプリケーションに対応する、万能のアーキテクチャやアプローチは存在しません。マイクロソフトの開発スタックおよび .NET はきわめて柔軟性に優れ、多くの選択肢を提供しますが、構築するアプリケーション、さらにはサブシステムの種類に応じて、特化したアプローチとテクノロジを選択することが非常に重要です。優先事項や妥協点はアプリケーションによってまったく異なるため、それぞれに合った対応をしなければなりません。

ビジネス アプリケーションの刷新は、単にモバイル アプリケーションを構築するだけで達成されるものではなく、モバイル アプリケーションは、基盤ビジネス アプリケーションに依存しつつ、さらに拡張されたものでなくてはならない

アプリケーションの効果を高めるには、現在の基盤ビジネス アプリケーションと緊密に連携するモバイル アプリケーションを構築する必要があります。モバイル対応のビジネス アプリケーションは、より規模の大きい企業エコシステムの一部として機能するべきものです。基盤となるシステムがレガシ アプリケーションであるか、または拡張性と弾力性に優れた革新的なサービスを使用して構築された、規模が大きく新しい、ミッション クリティカルなアプリケーションであるかに関係なく、基盤ビジネス アプリケーションを大幅に拡張するものでなくてはなりません。

アプリケーションやサブシステムの位置付けを包括的なパターン分類の中で判断することが、適切なアプローチおよびテクノロジの選択に役立つ

目的のアプリケーションやサブシステムに適切な領域を正しく見極めて分類することが何より重要です。次のアプリケーション タイプのどれに分類するかによって、採用すべきアプローチとテクノロジはまったく異なる場合があります。

- 次世代型のアプリケーション パターン
  - デバイスとサービス
- 従来型のアプリケーション パターン
  - 小規模/中規模ビジネス アプリケーション
  - ミッション クリティカルな大規模ビジネス アプリケーション

2. このガイドの目的

このガイドでは、.NET カスタム アプリケーションを開発するにあたり、それぞれのアプリケーションとビジネス ドメインの優先事項に応じて、マイクロソフトの適切な開発テクノロジおよびアプローチを的確に選択するための情報を提供します。

このガイドでは、アプリケーション ライフサイクル管理 (ALM) については扱いません。このトピックの詳細については、Visual Studio の ALM に関する Web サイト (http://www.microsoft.com/visualstudio/jpn/alm) を参照してください。

対象読者

このガイドは、マイクロソフトの開発プラットフォームをベースとしたアプリケーションおよびプロジェクトに使用するテクノロジの選定に関わる意思決定者、ソフトウェア アーキテクト、開発リーダー、および開発者に役立つ情報を提供します。

カスタム エンタープライズ アプリケーション開発を中心に説明しますが、ISV に役立つ情報および推奨事項も含まれます。

Dynamics CRM や Dynamics ERP を基盤とする業種別ソリューションなど、マイクロソフトの企業向け製品をベースとした開発ソリューションには触れません。

このガイドの使用方法

このガイドでは、ビジネス アプリケーションに焦点を当て、各種のソフトウェア開発オプションについて幅広く解説します。リファレンス マニュアルとして構成されており、「次世代型のアプリケーション パターン」や、「従来型のアプリケーション パターン」セクション内の「ミッション クリティカルな大規模ビジネス アプリケーション」など、関心のある項目を直接参照できるようになっています。

「概要」で全体像を把握してから、各セクションの具体的な内容に入ることをお勧めします。

3. 概要

現在、テクノロジの使用環境は、クラウドで提供される各種サービスを活用したマルチデバイス エクスペリエンスへの移行途上にあります。また、タッチ操作、センサー、モビリティといったローカル ハードウェアの機能に加え、Web との接続性や、データ ストレージ、メディア ストリーミング、ソーシャル機能のようなバックエンド サービスの性能が、ユーザーの利用パターンに影響を及ぼすようになっています。

こうしたデバイスとサービスの結び付きは、ビジネス シナリオとコンシューマー シナリオの両方で広がりつつあります。コンシューマー環境では、モバイル コンピューティングが普及し、情報の閲覧を目的としたデバイスが次々と開発されたことが契機となり、ハードウェア機能およびテクノロジの発展へとつながりました。一方、企業環境では、IT のコンシューマライゼーションと、個人所有デバイスの業務利用 (BYOD) という 2 つの社会現象によって、コンシューマー エクスペリエンスが、ビジネス コンピューティングおよび基幹業務 (LOB) アプリケーションの進化を後押しするという構図が出来上がっています。

デバイスおよびサービスに依存した次世代のアプリケーションは、単体では成立しません。既存アプリケーションときわめて密接に統合され、新たなユーザーに向けて、または新しい意思疎通の方法として、既存アプリケーションの価値を広げる存在となる必要があります。そのため、すべてのアプリケーション開発者は、2 つの異なるパターンに対処しなくてはなりません。

- **従来型のアプリケーション パターン:** クライアント/サーバー方式などのテクノロジ パターンを使用して開発されたアプリケーションや、デスクトップ ブラウザーに最適化された Web アプリケーション。基盤アプリケーションとして機能するもので、既存の**ビジネス プロセス**に大きな比重を置いて設計されています。
- **次世代型のアプリケーション パターン:** マルチデバイスやクラウドといった新しいテクノロジで実現されるアプリケーション。アプリケーションの対象を**エンド ユーザー**にまで拡大することで、従来型のパターンを補完します。

アプリケーション パターンの進化

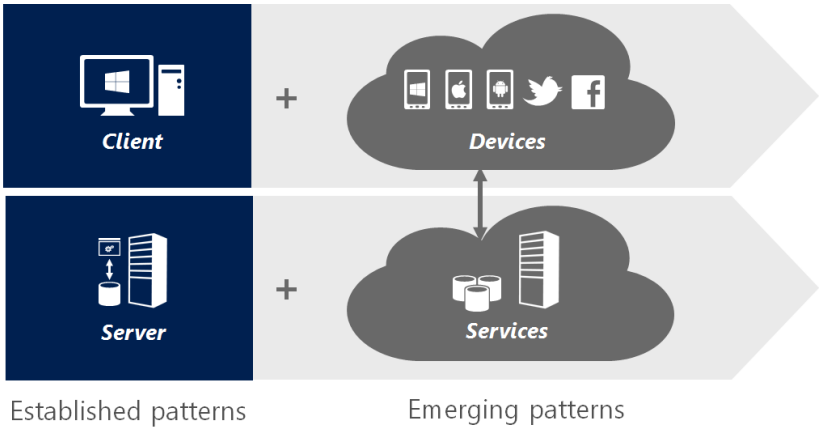

図 3-1

今日では、エンド ユーザー向けに従来型のパターンが拡張される傾向にあり、これによって開発者が革新的な技術を取り入れ、競合との差別化を図るための絶好の機会が生み出されています。小売、通信、金融、物流、顧客サービスなど、現在のビジネス環境ではあらゆる企業がソフトウェア企業であると言っても過言ではありません。企業が顧客ニーズを満たし、競争を優位に進められるかどうかは、ソフトウェアの革新にかかっています。

ただし、既存のアプリケーションを拡張して前述の新しいニーズに対応するには、困難な変革のプロセスが伴います。現在使用されている開発テクノロジは従来型のパターンに深く根差しており、最新のソフトウェアで必要とされる新たなパターンとの統合は簡単ではありません。既存のクライアント/サーバー方式から新たなデバイス/クラウド方式への明確な変換方法は、既存のツールから提供されません。

開発者のこうした課題を解消するのが、マイクロソフト プラットフォームです。マイクロソフト プラットフォームは**既存アプリケーション**に基づいて構築されており、次世代型のアプリケーション パターンへの拡張を可能にします。また、**複数の開発テクノロジ**がサポートされるため、開発者のスキルや、既存アプリケーションで使用されているテクノロジに応じ、最適な方法を選ぶことができます。サービスの開発に関しては、Windows Azure は Java、Node.js、PHP、Python、Ruby といった開発者の多くが使用できるテクノロジを豊富にサポートしており、特に .NET について抜群のサポート性を備えています。マイクロソフトのプラットフォームでは、クライアント開発に関しても、.NET、HTML5/JavaScript、C++ といった広範なテクノロジが標準でサポートされます。

モダン ビジネス アプリケーション

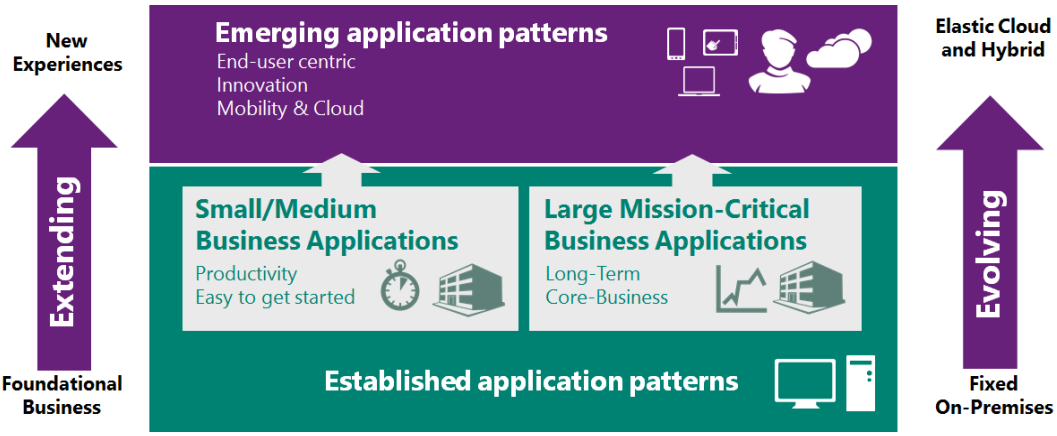

図 3-2

このガイドでは、.NET 開発に焦点を当て、特にビジネス アプリケーションを中心に説明を進めます。具体的には、既存アプリケーションを形成している従来型パターンでの .NET 開発の使用方法に加え、モダン **ビジネス アプリケーション**を実現する新たな次世代型パターンに対応する方法を解説します。

.NET Framework と開発の将来

Microsoft .NET Framework は、マイクロソフト プラットフォーム上で魅力的なアプリケーションを開発できるようにするために構築され、総じて市場では優れた成功を収めています。現在、規模や業種を問わずあらゆる企業において、何百万という開発者がアプリケーションの作成に .NET を利用しています。.NET は、コンシューマー アプリケーション、小規模ビジネス アプリケーション、およびミッション クリティカルな大規模アプリケーションの構築に必要となる、いずれも卓越した品質、パフォーマンス、生産性を誇る各種のコア サービスを提供します。

さらに、.NET は先ほど述べた、台頭しつつある新たなパターンも想定して構築されています。かつて Forum 2000 において、Bill Gates は「Web サイトが個々に独立している状態から、交換可能なコンポーネントから成るインターネットへと移行し、さまざまなデバイスとサービスを組み合わせることで、一貫性のあるユーザー主導のエクスペリエンスを実現する」ことが .NET の目標だと説明しました。こうした .NET の初期のビジョンは、多様なデバイスとサービスを通じたエクスペリエンスが、ソフトウェア開発に対する業界全体の意識を変化させつつある点など、今日開発者を取り巻いている環境と驚くほど一致しています。

各種サービスを活用したマルチデバイス エクスペリエンスの実現は、.NET の特性として、構想当初から重視されていました。その後、.NET は進化を続け、今ではアプリケーションの新しいニーズに対し、業界屈指の開発エクスペリエンスを提供しています。

- **サーバー側では、**.NET はオンプレミス環境向けサービスとクラウド環境向けサービスの両方に対し、共通の開発プラットフォームを提供します。Windows Server および Windows Azure との緊密な統合機能により、各プラットフォームの最も優れた点を生かしながら、アプリケーションを徐々にクラウドへと拡張すると共に、2 つの環境にまたがるハイブリッド アプリケーションを実現できます。また、.NET Framework ライブラリは短い間隔で提供されるため、最新技術を継続的に取り入れ、サービスの軽量化、リアルタイムの通信、モバイル Web アプリケーション、認証といった領域における、クラウド ベースのアプリケーションの新しいニーズにも応えられます。

- **クライアント側では、**.NET はマイクロソフト デバイスはもちろん、すべてのデスクトップ エクスペリエンス、Windows Phone アプリケーション、Windows ストア アプリにわたって、一貫性ある卓越した開発エクスペリエンスを提供します。.NET を使用すると、デスクトップ上で継続的に基盤アプリケーションを開発し、新しい画期的なエクスペリエンスを追加すると同時に、常に開発者の既存のスキルを生かしたり、デバイス間でコードを再利用したりすることが可能です。マイクロソフト デバイス以外も対象となるシナリオでは通常、HTML5 ブラウザー ベースのソリューションを使用します。.NET を Visual Studio と併用すると、多種多様なデバイス上で動作する標準ベースの Web アプリケーションを作成するためのモダン ソリューションとして使用できます。特定のデバイスにさらに特化した、ネイティブ エクスペリエンスの実現を求める開発者向けには、Visual Studio Industry Partner (VSIP) プログラムを通じて、C# のスキルやコードを Windows 以外のデバイスで再利用するためのソリューションが提供されています。

クライアント側とは切り離されたサーバー側の .NET 環境

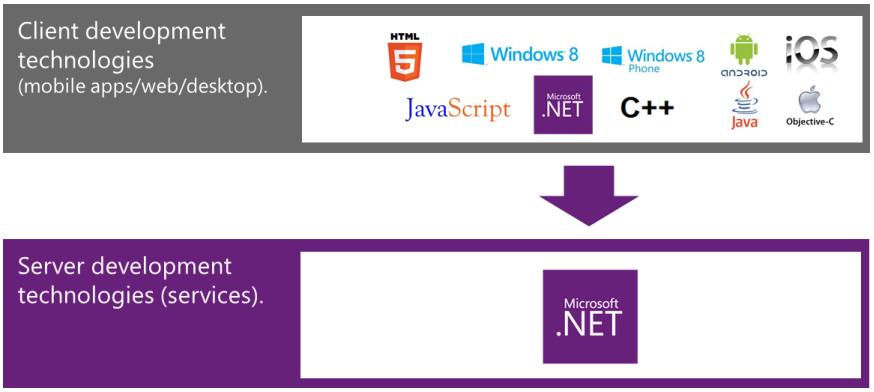

図 3-3

このガイドでは、今後アプリケーションを発展させるにあたり、現在のスキルとアプリケーション要件に応じた適切な決定を下せるよう、これまで紹介したすべての .NET 開発の選択肢を取り上げます。以降の内容は、次の 2 つのアプリケーション パターンに基づいて構成されています。

- 「次世代型のアプリケーション パターン」では、さまざまなデバイスとサービス間で実行される新たなアプリケーションを形作りつつある、次世代型のパターンを使用したアプリケーションの構築方法について説明します。
- 「従来型のアプリケーション パターン」では、基盤ビジネス アプリケーションを作成する際に利用可能なテクノロジを紹介し、こうしたテクノロジの刷新方法を提案します。

図 3-4 に、マイクロソフトの開発プラットフォーム テクノロジの全体図を示します。以降のセクションでは、アプリケーションのパターンと優先事項に応じて、どのテクノロジをいつ使用するべきかについて分析および提案していきます。

マイクロソフトの開発プラットフォーム テクノロジ

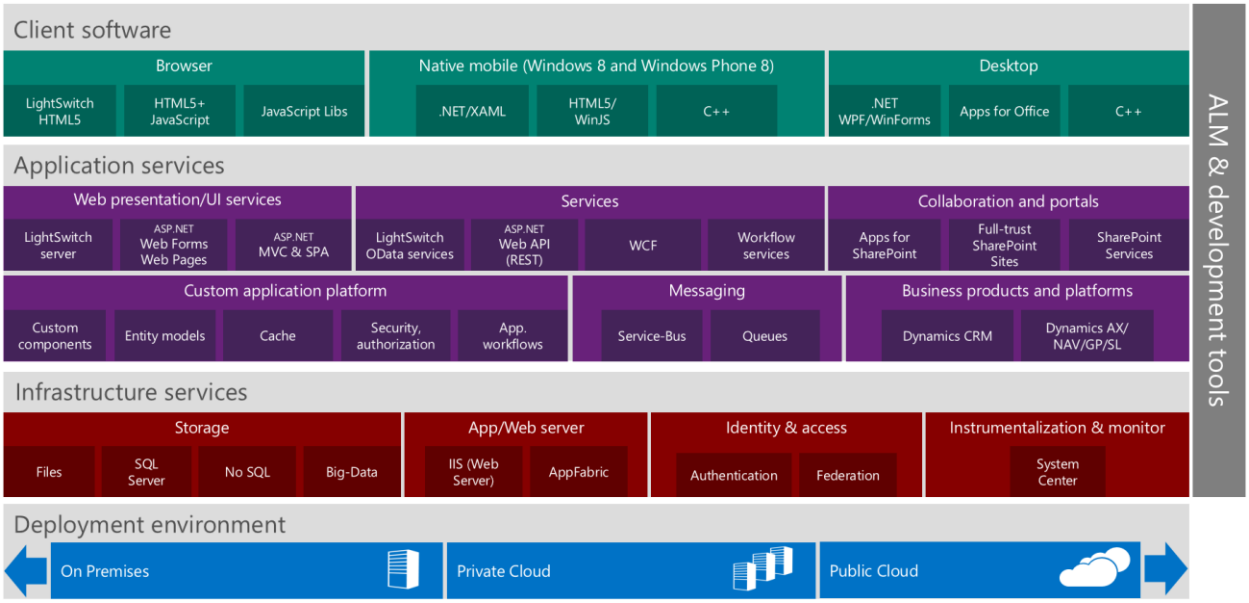

図 3-4

4. 次世代型のアプリケーション パターン

既に述べたように、現在、次世代型のアプリケーション パターンによって、将来のアプリケーション像が形作られつつあります。今日、企業のお客様や社員の皆様は、より個人のニーズに合ったエクスペリエンスを実現するアプリケーションを求めており、また同時に、必要なサービスに常に接続していたいとも考えています。

このセクションでは、こうした新種のアプリケーションの開発にあたって対処が必要となる、次の 2 つの重要なテーマについて考察していきます。

- 種類の異なる複数のデバイスにわたる、共通のエクスペリエンスを開発する
- クラウドを通じて拡張可能な、標準的かつ軽量なサービスを開発する

次世代型のアプリケーション パターンは、いわゆる「systems of engagement (エンゲージメントのシステム)」(Geoffrey Moore 氏が提唱し、Forrester の文書 (英語) によく使用される単語) と多くの点で似通っていますが、クラウドによるサポートを必要とする点と、既存の「記録のシステム」(基盤となるビジネス アプリケーション) に依存しながらそれに基づいて拡張する存在でもある点が異なっています。

図 4-1

"エンゲージメントのシステム" は、"コンシューマーのみをターゲットとするアプリケーション" ではありません。事実、最終顧客を対象としたシナリオ (オンライン バンキングや e コマースなど) 以外に、企業の中にも、このコンセプトが完全に合致する新しいシナリオ (ダッシュボードなどの、モビリティ要件を備えた社内アプリケーション) が数多く存在します。

そのため、次世代型のアプリケーション パターンに対するマイクロソフトのビジョンでは、モバイル ニーズおよび顧客との直接接続に加え、継続的かつ弾力性あるサービスが必要であるとされています (図 4-2 を参照)。

次世代型のアプリケーション パターン

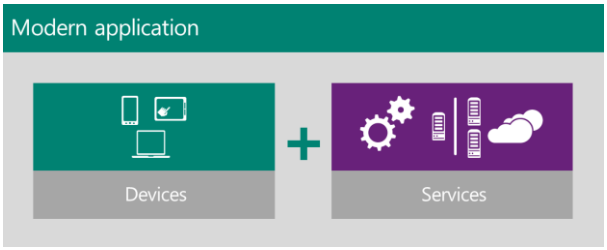

図 4-2

最先端のビジネス アプリケーションでは、クライアント アプリケーションがデータを利用し、オーケストレーションによって処理を集約するために、リモート サービスが必要となります。このリモート サービスを実現するうえで要求されるのが、インターネット標準に基づいたサービス指向の考え方です。最終的にこうしたサービスは、パブリック クラウドのインフラストラクチャおよびサービス内に展開され、次世代型パターンの展開環境に欠かせないものとなります。

次世代型のアプリケーション パターンを深く掘り下げていくにあたっては、ビジネス アプリケーションを新しい用途に対応するよう拡張し、あらゆる要求に応えて柔軟に拡大、縮小できる強固で継続的なサービスを構築する手段として、考えられるすべてのモバイル デバイスおよびソーシャル ネットワークを考慮に入れる必要があります。

デバイス、ソーシャル、サービス、データ、クラウドの関係

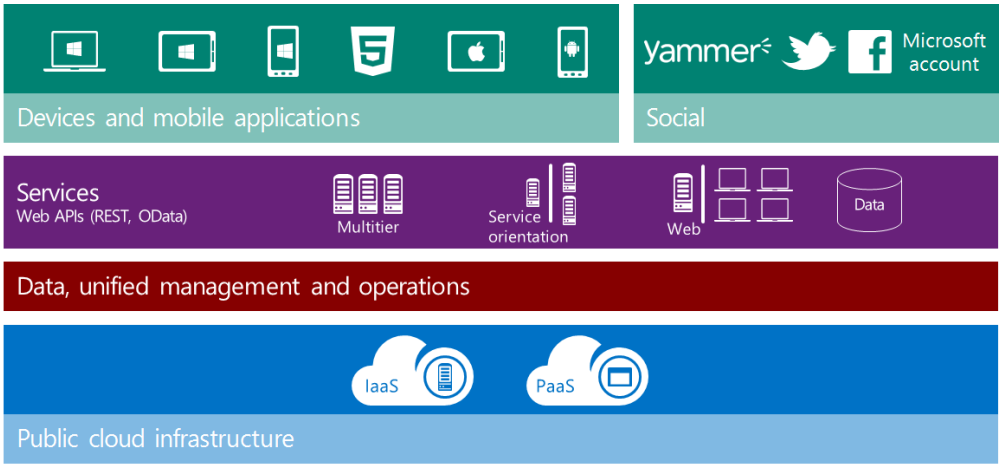

図 4-3

デバイス

次世代型のアプリケーション パターンの重要な特色として、各デバイスに特化した新しいエクスペリエンスを提供できる点が挙げられます。最適なテクノロジを選択して各デバイス専用のアプリケーションを作成する作業は難しく、次のような多くの要素が関係してきます。

- 担当者の既存スキルおよび得意とするテクノロジ
- ローカル ハードウェアの機能と連携可能な、デバイス専用のエクスペリエンスを作成する能力
- アプリケーションがターゲットとするデバイスの多様性
- 既存アプリケーションによって使用され、デバイスへの拡張または移行が必要となるテクノロジ

業界では 2 種類の対応方法が広く確立されていますが、互いにまったく異なるアプローチに基づいています。

- **ネイティブ アプリケーション:**それぞれのデバイスの特長を最大限に引き出せるが、プラットフォームごとに固有のスキルとコードが必要
- **Web アプリケーション:**共通のスキル セットとコードで作成できるが、各デバイスに特化したエクスペリエンスは実現不可

各種デバイスと マイクロソフトテクノロジのシナリオ

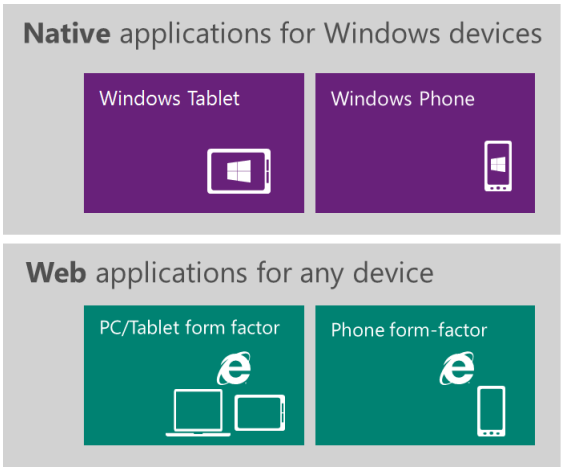

図 4-4

マイクロソフト プラットフォームは、両方のアプローチを完全にサポートするだけでなく、それぞれのデメリットを大幅に軽減します。まず、Windows デバイスでは固有のネイティブ開発モデルが要求されず、開発者のスキルと既存アプリケーションに応じた、最も合理的なテクノロジを使用できます。.NET、HTML/JavaScript、および C++ とデバイスとの高度な連携が実現されるため、エクスペリエンスについて妥協することなく、ニーズに最適な意思決定が可能です。さらに、.NET と Visual Studio を併用すると、任意のデバイス上で動作する Web アプリケーションをたいへん容易に作成できるようになります。ASP.NET は、近年使用されている標準をすべてサポートしているため、Visual Studio が備える最新かつ独自の革新的技術を併用することで、最新のブラウザーをフル活用し、さまざまなデバイスで動作する新しい種類の Web アプリケーションを構築できます。

Windows デバイス向けネイティブ アプリケーション

ネイティブ アプリケーションとは、クライアント デバイス上で実行され、そのデバイス固有の機能をフル活用することで、たいへん優れたカスタマー エクスペリエンスを実現するアプリケーションです。既に述べたように、Windows プラットフォームでは、このコンセプトを C++ 以外のテクノロジにまで拡張でき、新しいフォーム ファクターでも既存のコードやスキルを再利用できる可能性が大幅に向上します。

次の表で、使用可能なテクノロジの相違点を説明し、各アプリケーションの優先事項および具体的なコンテキストに応じた、適切な使用状況を示します。

図 4-5

Windows 8 向け Windows ストア アプリの UI 開発テクノロジ
(Windows ランタイム/WinRT)

テクノロジ	選択すべき状況と理由
.NET/XAML	▪ 既に .NET と XAML の使用経験が豊富な場合、または既存の .NET/XAML アプリケーションを拡張する場合に使用 ▪ 移植可能なクラス ライブラリを使用して、Windows ストア アプリ、Windows Phone アプリケーション、Windows デスクトップ アプリケーション、およびその他のマイクロソフト プラットフォームの間で、.NET Framework のコードおよびライブラリを共有することも可能 ▪ ローカルの軽量な SQL データベース エンジンを構築するための sqlite-net や、クライアントとサーバー間の双方向通信を実現するための ASP.NET SignalR .NET クライアントといった、オープン ソースの .NET ライブラリを活用
JavaScript/ HTML5	▪ 既に HTML テクノロジと JavaScript テクノロジの使用経験が豊富な場合、または既存の Web アプリケーションのために、Windows ストア仕様のエクスペリエンスを作成する場合に使用 ▪ 既存のブラウザー ベース Web アプリケーションに使用されている、JavaScript または HTML/CSS のカスタム アセットを再利用するほか、新しい WinJS ライブラリを使用して、Windows ストア アプリ/API のネイティブ機能にアクセス可能 ▪ ローカルの軽量な SQL データベース エンジンを構築するための SQLite3-WinRT や、クライアントとサーバー間の双方向通信を実現するための ASP.NET SignalR JavaScript クライアントといった、オープン ソースの JavaScript ライブラリを活用
C++	▪ 既に C++ の使用経験が豊富な場合に使用。グラフィックスを多用したアプリケーションやゲームを最適化し、最善のパフォーマンスを実現 ▪ Windows、Windows Phone、およびその他のプラットフォームの間で C++ コードを共有するほか、Direct3D を使用して低レベルのグラフィックスにアクセスすることも可能 ▪ SQLite などのオープン ソースの C/C++ コードを活用

表 4-1

参考資料	
Windows Phone 8 と Windows 8 の間で最大限にコードを再利用する	http://msdn.microsoft.com/ja-jp/library/windowsphone/develop/jj681693
Microsoft patterns & practices シリーズ: Prism for Windows Runtime	http://prismwindowsruntime.codeplex.com/ (英語)
ASP.NET SignalR: .NET 用のきわめてシンプルなリアルタイム Web 機能	http://signalr.net/ (英語)
SQLite: System.Data.SQLite ダウンロード ページ	http://system.data.sqlite.org/index.html/doc/trunk/www/downloads.wiki (英語)
Visual C++ と WinRT/Metro – 基本原則	http://www.codeproject.com/Articles/262151/Visual-Cplusplus-and-WinRT-Metro-Some-fundamentals (英語)

Windows Phone 8 アプリケーションの UI 開発テクノロジ

テクノロジ	選択すべき状況と理由
.NET/XAML	▪ 既に .NET と XAML の使用経験が豊富な場合、または既存の .NET/XAML アプリケーションを拡張する場合に使用 ▪ 移植可能なクラス ライブラリを使用して、Windows Phone、Windows ストア、Windows デスクトップ、およびその他のマイクロソフト プラットフォームの間で、.NET Framework のコードおよびライブラリを共有可能 ▪ sqlite-net-wp8 など、オープン ソースの .NET ライブラリを活用
C++	▪ 既に C++ の使用経験が豊富な場合に使用。グラフィックスを多用したアプリケーションやゲームを最適化し、最善のパフォーマンスを実現 ▪ Windows Phone、Windows ストア、およびその他のプラットフォームの間で C++ コードを共有するほか、こうしたプラットフォーム内で、Direct3D を使用して低レベルのグラフィックスにアクセスすることも可能 ▪ SQLite などのオープン ソースの C/C++ コードを活用

表 4-2

参考資料	
Windows Phone 8 開発者プラットフォームの特徴	http://blogs.windows.com/windows_phone/b/wpdev/archive/2012/11/05/windows-phone-8-developer-platform-highlights.aspx (英語)
Windows Phone 8 および Windows 8 のアプリ開発	http://msdn.microsoft.com/ja-jp/library/windowsphone/develop/jj714089(v=vs.105).aspx
Windows Phone 8 と Windows 8 の間で最大限にコードを再利用する	http://msdn.microsoft.com/ja-jp/library/windowsphone/develop/jj681693
Windows Phone 8 用の Direct3D アプリ開発	http://msdn.microsoft.com/ja-jp/library/windowsphone/develop/jj207052(v=vs.105).aspx

任意のデバイスで使用できる Web アプリケーション

マイクロソフトは、任意のデバイス上で動作するブラウザー ベースの HTML5 アプリケーションを作成できる、業界屈指のツールおよびテクノロジを提供しています。HTML5 は Windows でもネイティブ テクノロジとしてサポートされるため、複数のデバイスをターゲットとする HTML5 Web アプリケーションを開発すれば、そのコードをネイティブ Windows ストア アプリ用に最適化して再利用できます。

HTML の複数のレンダリング方法やサイジング方法を自動的に切り替えるには、ASP.NET、JavaScript、および LightSwitch によって提供されるさまざまなメカニズムや、各種のライブラリを活用します。たとえば、Modernizr を使用すると、使用されているブラウザーを検出し、HTML/JavaScript コードを適応させることができます (図 4-6 を参照)。

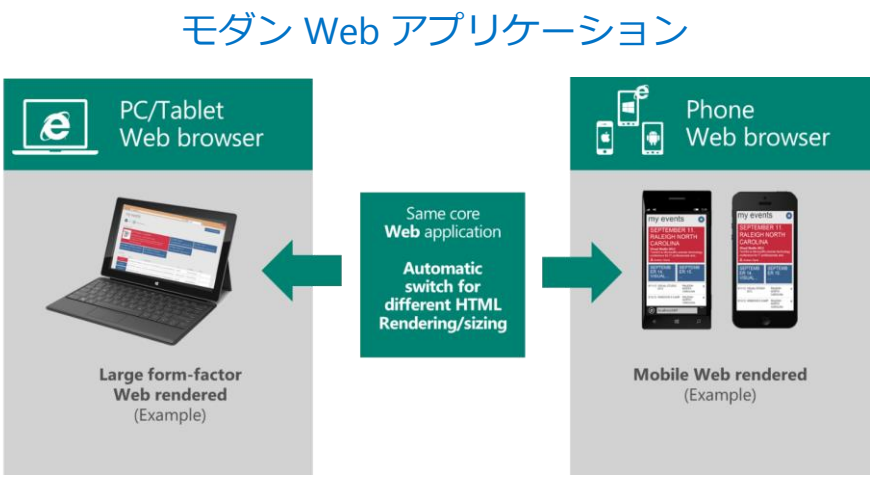

図 4-6

このアプローチは、複数のオペレーティング システム (Windows、iOS、Android など) をターゲットとする場合に、特に効果を発揮します。最も一般的な Web ブラウザーとアプリケーションの互換性を確保することで、主要モバイル オペレーティング システムとのプラットフォームをまたいだ互換性が実現されます。

次の表に、使用可能な各種の Web 開発テクノロジの一覧と、アプリケーションの優先事項およびコンテキストに応じた、各テクノロジの適切な使用状況を示します。

Web アプリケーションの UI 開発テクノロジ

テクノロジ	選択すべき状況と理由
HTML5 をサポートする ASP.NET **MVC**	**HTML5 をサポートする ASP.NET Model-View-Controller (MVC)** ▪ 柔軟性に優れたモダン モバイル Web アプリケーション、ブラウザー間での互換性、あらゆる最新デバイス上で動作するモバイル Web クライアントを実現し、ASP.NET MVC の各種モバイル機能 (複数ページの使用、検出された現在のブラウザーのユーザー エージェントに基づくレンダリングなど) の活用が可能 ▪ HTML のレンダリングを完全に制御でき、単体テストが容易で、フェイク機能を提供 ▪ Modernizr などのライブラリを使用し、HTML5 の機能サポートの有無と回避策を検出 ▪ CSS3 を利用してスクリプトを短くし、保守しやすいコードを作成 ▪ クライアント側のプログラミングには JavaScript と jQuery を使用 ▪ MVC は検索エンジン最適化 (SEO) のカスタマイズに対応 ▪ MVC は RESTful URL を標準サポート

ASP.NET **SPA**	**ASP.NET MVC、HTML5/JavaScript、Knockout、**および **Breeze** に基づく **Single Page Application (SPA)** ▪ SPA はフレームワークではなく、ASP.NET MVC を使用して実装できる設計パターン。Knockout (JavaScript 内で MVVM パターンをサポート) や Breeze (高度なデータ管理が可能) などの JavaScript ライブラリや、Durandal をはじめとする JavaScript フレームワークを多用 ▪ 対話式操作の多い Web アプリケーションやスムーズな UI の更新 (サーバー ページの再読み込みおよび可視性の切り替えに伴うラウンド トリップを最小限にする場合) のほか、クライアント側で HTML5、CSS3、および JavaScript による重要な対話式操作を行う場合にも使用 ▪ クライアントとサーバー間の双方向通信を実現するには、オープンソースの ASP.NET SignalR JavaScript クライアント ライブラリを活用 ▪ JavaScript から ASP.NET Web API サービスにアクセス
LightSwitch プロジェクトの HTML5 クライアント	**LightSwitch HTML5 クライアント** ▪ 主にデータ駆動型 (CRUD) Web アプリケーションまたはモジュールに使用 ▪ すべての最新デバイス上で動作し、自動 HTML レンダリングの活用や、さまざまなフォーム ファクターへの適応が可能な、データを中心としたクロスブラウザーのモバイル Web アプリケーションを作成する最も容易な方法 ▪ jQuery Mobile などの OSS JavaScript ライブラリ、CSS3、JavaScript、およびテーマを活用 ▪ ASP.NET ベースで構築された、エンドツーエンドの LightSwitch ランタイム サーバー エンジンと併用
ASP.NET **Web ページ**	**ASP.NET Web ページ** ▪ ASP.NET Web ページおよび Razor 構文は、サーバー コードと HTML を組み合わせて動的 Web コンテンツを作成するための、アプローチが容易で高速かつ軽量な手段を提供

表 4-3

参考資料

ASP.NET MVC 4 のモバイル機能	http://www.asp.net/mvc/tutorials/mvc-4/aspnet-mvc-4-mobile-features (英語)
ASP.NET SPA の概要	http://www.asp.net/single-page-application (英語)
Breeze/Knockout テンプレートの概要	http://www.asp.net/single-page-application/overview/templates/breezeknockout-template (英語)
Durandal SPA フレームワーク	http://durandaljs.com/ (英語)
RequireJS ライブラリ	http://requirejs.org/ (英語)
BootStrap 3	http://twitter.github.io/bootstrap (英語)
Modernizr (HTML5/CSS3 の機能検出ライブラリ)	http://www.modernizr.com (英語)
LightSwitch アーキテクチャについて学ぶ	http://msdn.microsoft.com/ja-jp/vstudio/gg491708 (英語)
OData による LightSwitch アプリケーションの拡張	http://blogs.msdn.com/b/bethmassi/archive/2012/03/06/enhance-your-lightswitch-applications-with-odata.aspx (英語)
LightSwitch アプリの HTML クライアント画面	http://msdn.microsoft.com/ja-jp/library/vstudio/jj674623.aspx

サービス

複数のデバイスをターゲットとする開発プロセスは、バックエンドの構築から始まります。アプリケーションでは、あらゆるデバイス上で使用でき、インターネットへと拡張可能なサービスを公開する必要があります。

インターネット経由のサービスには、モダン アプリケーションの動作に必要なバックエンド サービスを提供すると共に、**優れた可用性および継続性**を実現することが求められます。同時に、ビジネスの変化のペースに遅れることなく、継続的かつスムーズに発展させることのできる機動性も必要となります。

こうした背景がある中、**サービスの構築には ".NET エコシステム" および各種フレームワークが欠かせません。**

図 4-7 に示すように .NET Framework では、アプリケーションを構築するための広範なアプローチがサポートされます。たとえば、ASP.NET アプローチ (SPA、MVC、および Web フォーム) による Web 開発、ASP.NET Web API による HTTP/REST サービスの構築のほか、Entity Framework を使用したリレーショナル データベース内のデータへのアクセスに対応しています。**ほとんどのサーバー側開発は、.NET テクノロジ ベースで行うのが一般的です。**

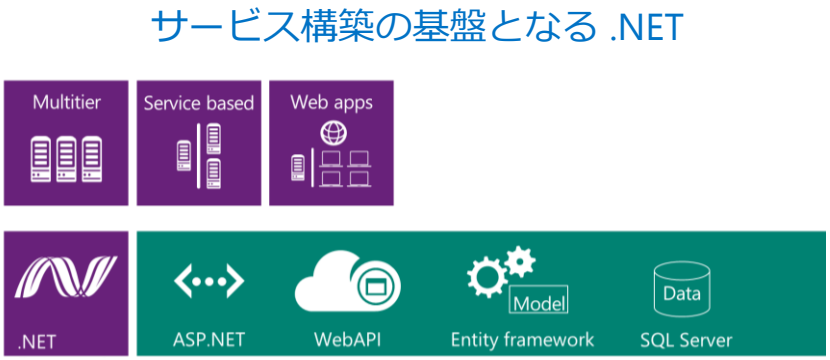

図 4-7

次の表に、サービス構築時に選択可能な、さまざまなアプローチに対応するテクノロジを紹介します。

バックエンド サービス テクノロジ

(ネイティブ アプリケーションまたは Web アプリケーションで使用)

テクノロジ	選択すべき状況と理由
ASP.NET **Web API**	**HTTP ベース、REST アプローチ、リソース指向、OData (Open Data Protocol) と JSON の重視** ▪ **REST** アプローチ、**OData**、**JSON** などの要件を満たし、サービス開発を柔軟に行えるため、最も推奨されるテクノロジ。使用するテクノロジを評価する際は、まず Web API を試し、Web API がニーズに合わない場合は、他のいずれかのテクノロジを使用 ▪ REST サービスに特化した設計。アプリケーション ドライバーとして HTTP のメソッド (PUT、GET、POST、DELETE など) を採用しており、リソース指向性が高い ▪ HTTP のメソッドに基づくインターネット キャッシュ (Akamai、Windows Azure CDN、Level3 など) を活用し、優れた拡張性を提供
Windows Communication Foundation (WCF)	**依存性のない柔軟なアプローチ** ▪ SOAP との相互運用性が必要な場合、または HTTP 以外で転送を行う場合に使用 ▪ 任意のプロトコル (HTTP、TCP、名前付きパイプなど)、データ形式 (SOAP、バイナリ、JSON など)、およびホスティング プロセスを使用可能 ▪ RPC スタイル (タスク/コマンド指向) のサービスと、企業のアプリケーション間通信に最適なテクノロジ。REST には対応するが、非推奨 ▪ マイクロソフトのサービス バス (Windows Azure または Windows Server に含まれる)、および AppFabric のホスティング/監視機能と併用すると効果的
WCF Data Services	**データ駆動型サービス** ▪ データ/リソース指向性が高く、主に CRUD のデータ駆動型サービスに使用 ▪ **OData** のみをサポート。使用方法は容易だが、柔軟性と制御性は ASP.NET Web API に劣る ▪ ASP.NET Web API と同じ OData コア ライブラリを使用

ワークフロー サービス	**ワークフロー ベースのサービス構築アプローチ** ▪ 内部的なサービス ロジックが Windows Workflow Foundation (WF) ワークフローの場合に使用。外観は WCF サービスそのもの ▪ WCF および WF のすべての利点と特徴を備えているが、WCF に依存する
SignalR	**ASP.NET SignalR ライブラリ** ▪ クライアント側でリアルタイム機能を提供 ▪ サーバー側コードから、接続されたクライアントへとリアルタイムでコンテンツをプッシュし、数百万にのぼる大量のユーザーに配信可能 ▪ HTTP および WebSocket ベースのアプローチ。ブラウザー クライアントの JavaScript、.NET ネイティブ Windows クライアントとサーバー側イベント、およびロング ポーリングから利用可能
LightSwitch OData サービス	**OData および REST アプローチ、リソース指向、データ駆動型サービスの使いやすい構築手段** ▪ クライアント アプリケーションとして LightSwitch アプリケーションを使用する場合や、シンプルなスタンドアロンのデータ駆動型 OData サービスに使用。視覚的なデータ モデルを作成すると、LightSwitch が自動的にサービスを構築 ▪ OData ベースのアプローチ。サービスを使用するためのクライアント ライブラリは共通 ▪ LightSwitch サーバー エンジン、Restful な OData、XML、JSON、および HTTP と併用 ▪ 柔軟性は Web API や WCF に劣るが、コーディングが不要

表 4-4

参考資料

ASP.NET Web API の概要	http://www.asp.net/web-api (英語)
Windows Communication Foundation とは	http://msdn.microsoft.com/ja-jp/library/ms731082.aspx
Windows ストア アプリ用 WCF Data Services Tools ダウンロード ページ	http://www.microsoft.com/en-us/download/details.aspx?id=30714 (英語)
LightSwitch OData サービスの作成と利用	http://blogs.msdn.com/b/bethmassi/archive/2012/03/09/creating-and-consuming-lightswitch-odata-services.aspx (英語)
ASP.NET SignalR	http://signalr.net/ (英語)

クラウドとハイブリッド クラウド

モダン ビジネス アプリケーションは、数多くのインターネット ユーザー (エンド カスタマーやパートナーなど) に対応することが当たり前のこととなっており、各種のバックエンド サービスを企業の社内データセンター内で維持することは、必ずしも合理的とは言えません。その点で、クラウド内でのサービスの展開は理にかなっています。また、弾力性や、実稼働環境を迅速にコスト効率よく準備できる点など、クラウドの核となる特長をサービスに活用できるというメリットもあります。

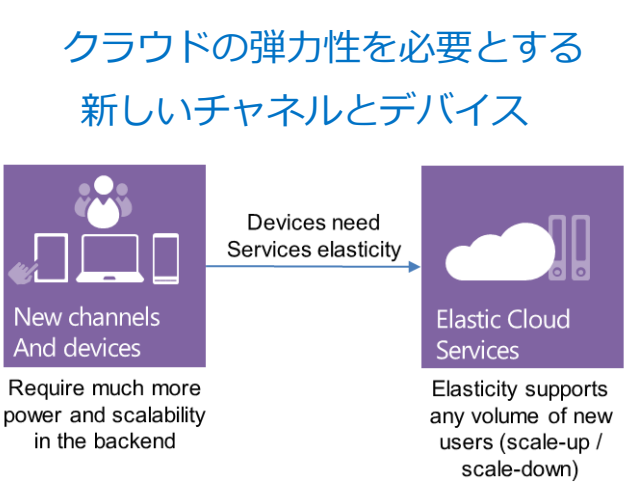

図 4-8

さらに、モダン ビジネス アプリケーションの多くは、新しいアイデア、新しいチャネル、新しいビジネス チャンスによって生み出されています。予想以上に多くの新しいユーザーを獲得し、社内のデータセンターでは必要なリソースをまかなえなくなる可能性もあります。機動性と拡張性を備えるクラウドなら、ハードウェアの購入やセットアップを必要とせずに、新たに考え出したコンセプトをすばやく実稼働に移

すことができます。こうした理由から、Windows Azure のような弾力性のあるプラットフォーム クラウドは、モダン ビジネス アプリケーションにとって最適な選択肢だと言えます。

ビジネス アプリケーションがさまざまな環境 (クラウドとオンプレミスの両方) に展開される多様なエコシステムでは、多くの新しいニーズを満たすことが求められます。さらに、現在オンプレミス環境で運用している基盤アプリケーションを拡張するため、自社専用のデータセンターとクラウドをリンクさせる必要がありますが、その際には図 4-9 で示すように、ハイブリッド クラウド アプローチを通じて双方の環境をセキュアに連携させなくてはなりません。

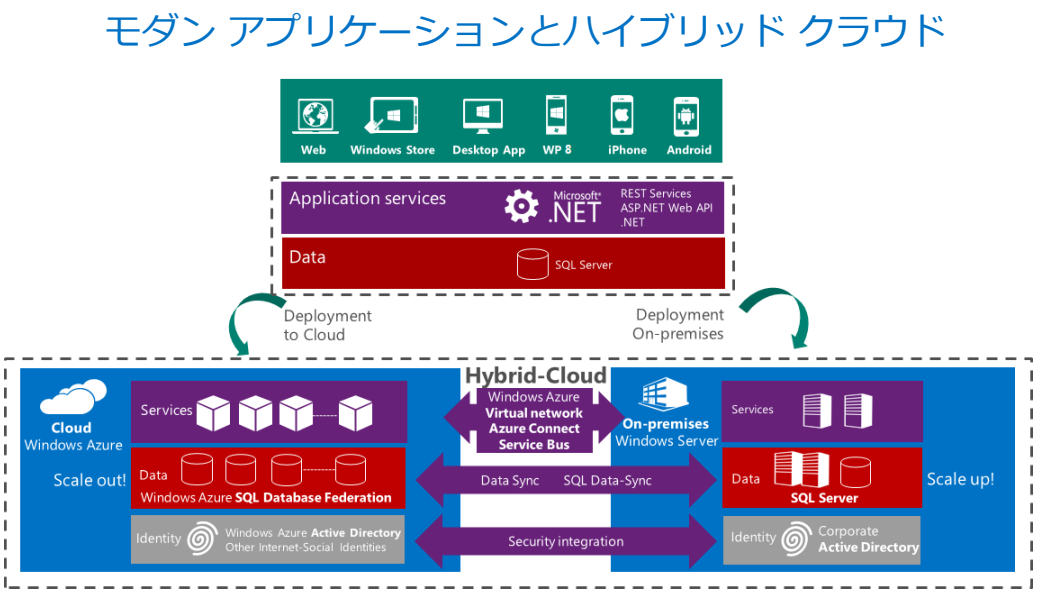

図 4-9

マイクロソフトのクラウド プラットフォームを使用すると、クラウドとオンプレミスの両方のインフラストラクチャをサポートする対称型のアーキテクチャ、テクノロジ、および製品だけでなく、ハイブリッド環境内で 2 種類のインフラストラクチャにまたがるアプリケーションを管理する共通のサービスが提供されます。さらに、アプリケーションの移行や構築を、簡単かつ段階的に実施することも可能です。

図 4-10 に示すように、マイクロソフトは、アプリケーションのターゲットとするインフラストラクチャがクラウド向けであるか社内運用向けであるかを問わず、一貫性ある単一のプラットフォームを提供しています。クラウド環境とオンプレミス環境のどちらにも、同じ開発プラットフォーム (たとえば Visual Studio や .NET) を使用することが、開発におけるベスト プラクティスです。同様に、オンプレミスまたはクラウドで運用するすべてのシステムについて、インフラストラクチャの制御と管理を 1 つのシステム (System Center など) に集約することが、効率的な IT ガバナンスの基本となります。

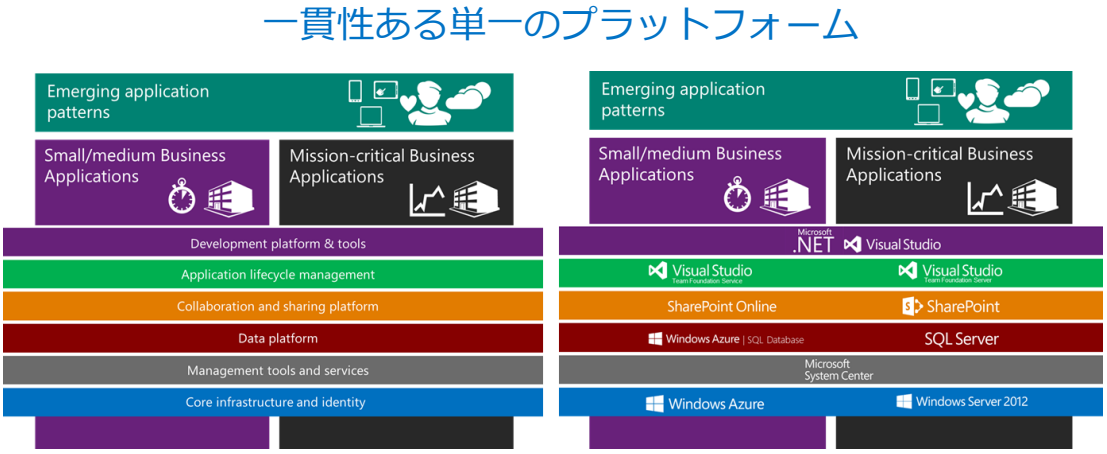

図 4-10

参考資料	
マイクロソフト パブリック クラウド	http://www.microsoft.com/ja-jp/server-cloud/public-cloud/default.aspx
マイクロソフト、新しい管理ソリューションでクラウド OS を推進	http://www.microsoft.com/ja-jp/presspass/detail.aspx?newsid=4238
ハイブリッド クラウド	http://www.microsoft.com/enterprise/en-nz/solutions/hybrid-cloud.aspx (英語)
マイクロソフトのクラウド OS ビジョン	http://www.microsoft.com/en-us/server-cloud/cloud-os/

次の表に、アプリケーション開発に関係した Windows Azure の主なテクノロジを示します。

Windows Azure のクラウド テクノロジ

テクノロジ	選択すべき状況と理由
実行モデル	**Web アプリケーションおよびサービスの実行モデル** Windows Azure は、ニーズに応じて選択可能な複数の実行モデルをサポートしています。 ▪ **インフラストラクチャ サービス (サービスとしてのインフラストラクチャ/IaaS):** 従来型の柔軟性の高いアプローチが必要な場合に使用。内部仮想マシン インフラストラクチャの運用、ソフトウェアの保守、および (オペレーティング システム、サービスなどの) 管理が必要で、SQL Server のフル インストールなど、ソフトウェアの従来型のインストールをサポート ▪ **Web サイト** (Web ホスティング): 管理された IIS Web サイト環境で Web アプリケーションを簡単に展開し、インフラストラクチャの管理業務を必要としないため、短期間での利用開始が可能。低コストで初期段階の拡張性に優れた、用途の広いプラットフォーム ▪ **クラウド サービス**または**サービスとしてのプラットフォーム (PaaS):** Web サイトと同様の考え方に基づくが、より高い拡張性と柔軟性を備えたプラットフォーム。高度なサービス品質要件が求められる場合や、制御性を高めるために使用。PaaS としての信頼性および管理に必要となるほとんどの作業にも対応 ▪ **Windows Azure モバイル サービス:** Windows ストア、Windows Phone、Apple iOS、Android、HTML/JavaScript 対応の各アプリケーションに、構造化ストレージ、ユーザー認証機能、プッシュ通知機能、およびバックエンドのジョブとサービスを提供する拡張性の高いクラウド バックエンドを実現
データ管理	**クラウド内データ ソースとツール** ▪ Windows Azure SQL データベース: オンプレミスの SQL Server に匹敵する、機能豊富なリレーショナル データベースを提供し、オンプレミスの SQL Server データベースからクラウドへの移行、または両環境の同期を簡単に実行可能。設定不要の高可用性に加え、保守/管理を Windows Azure インフラストラクチャに移管することにより、業務を大幅に簡素化できることが最大のメリット ▪ Windows Azure 仮想マシンでの SQL Server の実行: Windows Azure インフラストラクチャ サービス固有のシナリオの 1 つ。SQL Server のフル機能を必要とするアプリケーションにとって、Windows Azure 仮想マシンは最適なソリューション ▪ Windows Azure テーブル: NoSQL アプローチによるシンプルな非構造化データをベースとし、データ ソースにきわめて高い拡張性が求められる場合に最適 ▪ Windows Azure BLOB ストレージ: ビデオやファイル、バックアップ データまたはその他のバイナリ情報のような、非構造化バイナリ データを格納するためのストレージ
ビジネス分析	**ビッグ データとレポート生成** ▪ SQL レポート: レポート生成のプラットフォームを必要とする場合に、SQL Reporting Services と同様の機能を提供 ▪ Hadoop on Windows Azure: Windows Azure 内で、ビッグ データを PaaS としてホスト可能

ネットワーク	**ハイブリッド クラウドとネットワークの機能** ▪ Windows Azure 仮想ネットワーク: オンプレミスのローカル ネットワークを、定義済みの Windows Azure 仮想マシン (VM) に接続する、VPN に類似したサーバーおよびサブネットワーク指向のアプローチ ▪ Windows Azure トラフィック マネージャー: Windows Azure アプリケーションを複数のデータセンターで実行している場合に、ユーザーからの要求を、複数のアプリケーション インスタンスにインテリジェントにルーティング
メッセージング	**メッセージングと非同期通信** ▪ キュー: 同一の永続的キューにアクセスすることで、異なるアプリケーションまたはプロセス間の非同期通信を実現 ▪ サービス バス: キューと同様に永続的なメッセージング機能を提供するが、イベント駆動型アプリケーションと統合、およびパブリッシュ / サブスクライブ パターンのほか、サービス バスのリレー機能を通じてクラウドからオンプレミスのデータセンターに簡単にアクセスする場合 (ファイアウォールを間に挟んだピア ツーピア アプリケーションでも利用可能) など、より高度なシナリオに適する。Windows Server サービス バスにより、オンプレミス環境でも同等の機能を利用可能
キャッシュ	**アプリケーション内のデータ キャッシュ機能と、インターネットを使用した HTTP キャッシュ機能** ▪ Windows Azure キャッシュは、インメモリの分散キャッシュ。外部サービスとして使用するほか、自社の仮想マシン間で展開したキャッシュを共有することが可能 ▪ Windows Azure CDN は、HTTP 経由でアクセスするデータ (ビデオ、ファイルなど) をキャッシュできる "インターネット キャッシュ"
ID	**ID 管理サービス** ▪ Windows Azure Active Directory (AD): オンプレミスの ID 管理サービスとの統合により、クラウド アプリケーション間のシングル サインオンを実現。最新の REST ベースのサービスとして、クラウド アプリケーションに ID 管理機能およびアクセス制御機能を提供し、Windows Azure、Microsoft Office 365、Dynamics CRM Online、Windows Intune、およびその他のサード パーティ製クラウド サービスにわたって ID サービスを 1 つに集約 ▪ Windows Azure AD Access Control Service: Web アプリケーションおよびサービスにアクセスする必要のあるユーザーを、コードに複雑な認証ロジックを組み込むことなく、簡単に認証できる手段を提供。Windows Live ID、Google、Yahoo、Facebook などの Web ID プロバイダー (IP) や Windows Identity Foundation (WIF) のほか、Active Directory フェデレーション サービス (AD FS) 2.0、および OData をサポート ▪ AD FS (Windows Server): クレーム ベースのアクセス (CBA) 認証メカニズムを使用し、アプリケーションのセキュリティを維持しながら、オンプレミスのデータセンター、システム、およびアプリケーションへのアクセスを簡素化。Web シングル サインオン (SSO) テクノロジをサポートするため、組織の境界を超えた情報テクノロジ (IT) のコラボレーションが可能。オンプレミス環境の AD FS は、Windows Azure AD および ACS を通じて統合/拡張可能
HPC	**ハイパフォーマンス コンピューティング (HPC)** ▪ 多数の仮想マシンを同時に実行し、複数のタスクを並列処理する機能。Windows Azure では、これらのインスタンスに処理を分散するための HPC スケジューラを提供。これは、業界標準の Message Passing Interface (MPI) を使用するために構築された各種 HPC アプリケーションと連携させることができ、クラウド内で実行する HPC アプリケーションを簡単に構築することを目的とするコンポーネント
メディア	**Windows Azure メディア サービス** ▪ メディアの取り込み、エンコード、コンテンツ保護、広告の挿入、ストリーミングなどに対応した一連のクラウド コンポーネントを提供することで、ビデオなどのメディアを使用するアプリケーションの作成および実行に必要なプロセスを大幅に簡略化

表 4-5

Windows Azure およびクラウドに関する参考資料	
「Windows Azure によるクラウドでのハイブリッド アプリケーションの構築」 (Microsoft patterns & practices ガイド)	http://msdn.microsoft.com/en-us/library/hh871440.aspx (英語)
「Microsoft Windows Azure プラットフォームでのクラウドへのアプリケーションの移行」(Microsoft patterns & practices ガイド)	http://msdn.microsoft.com/ja-jp/library/ff728592.aspx
「クレーム ベース ID とアクセス制御のガイド」 (Microsoft patterns & practices ガイド)	http://msdn.microsoft.com/en-us/library/ff423674.aspx (英語)
「Microsoft Windows Azure™ プラットフォームでのクラウド用アプリケーションの開発」(Microsoft patterns & practices ガイド)	http://msdn.microsoft.com/ja-jp/library/ff966499.aspx
「アプリケーションを Windows Azure に移行する」 (Microsoft patterns & practices ガイド)	http://msdn.microsoft.com/ja-jp/magazine/jj991979.aspx
「Windows Azure 向け Enterprise Library 5.0 統合パック」 (Microsoft patterns & practices ガイド)	http://msdn.microsoft.com/en-us/library/hh680918.aspx (英語)
Windows Azure クラウド	http://www.windowsazure.com/

次世代型のアプリケーション パターンのエンドツーエンド シナリオ

シナリオ: Windows ストア アプリの連携

次の図 4-11 では、セクション 3 で紹介したテクノロジ マップ (図 3-4) を使用して、モダン Windows アプリケーション (タブレットおよびスマートフォンの両フォーム ファクターを含む) の典型的なシナリオおよびテクノロジの連携パターンを示しています。

シナリオ: Windows ストア アプリの連携

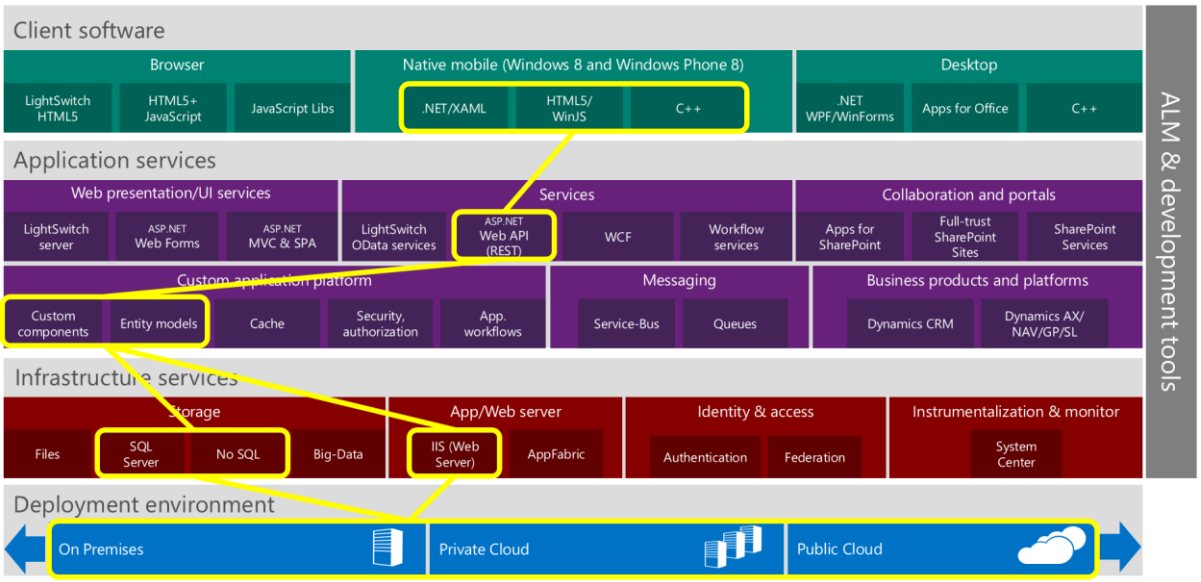

図 4-11

このシナリオは、UI クライアント アプリケーションが、Windows ストア アプリ (Windows タブレットなど) であっても Windows Phone のアプリケーションであっても、大きな違いはありません。開発には、担当者のスキルとアプリケーショ

ンの要件および優先事項に応じて、**.NET/XAML**、**HTML5/WinJS**、または **C++** を選択できますが、Windows ストアと Windows Phone の両方のネイティブ アプリケーション開発に対応するのは .NET および C++ のみです。

このシナリオの場合、モダン アプリケーションのバックエンドで提供されるカスタム サービスの開発には、**ASP.NET Web API** の使用をお勧めします。これにより、高度な柔軟性が得られると共に、通常 REST および OData または JSON のようなアプローチで使用される、インターネット サービス用の軽量かつパフォーマンスの高いフレームワークが提供されます。

次に、ASP.NET Web API サービスによって、中間層ロジックをラッピングした .NET カスタム クラス ライブラリおよびエンティティ モデルを作成します (リレーショナル データベースにアクセスする場合は Entity Framework を、NoSQL データ ソースにアクセスする場合は他のいずれかの API を使用)。

最後に、作成したサービスとサーバー コンポーネントを任意の展開環境に展開します。ただし、次世代型のアプリケーション パターンの場合、通常は Windows Azure などのパブリック クラウド内に展開することが多いです。

先ほども述べたように、モダン アプリケーションの典型的な特徴は、**さまざまなコンテキストやユーザーのシナリオに対してそれぞれ異なるアプリケーションを用意**するというものです (図 4-12 を参照)。ほとんどの場合、こうしたシナリオには固有の優先事項や要件があるため、それに応じたテクノロジを選択します。たとえば、Windows ストア アプリの開発で、インターネット フィードおよびソーシャル ネットワーク API にきわめて近い場合や、既存のスキルと JavaScript のコードを再利用したい場合は、HTML5/WinJS による開発が最適な選択だと言えます。そのほかに、C# または XAML に関するスキルを再利用しながら優れたパフォーマンスを実現したい場合は、.NET/XAML アプローチをお勧めします。また、グラフィックスのパフォーマンスを最大限に高めることを優先する一部のケースでは、C++ を選択すると良いでしょう。

Windows Phone のアプリケーション開発では、同様の点に注意して、.NET と C++ のいずれかを選択してください。

もう 1 つ重要なのが、あらゆるクライアント/シナリオにおいて、バックエンド サービスを共通化、再利用できる点です。少なくとも、同じ階層およびサーバー テクノロジを共有可能で、場合によっては各クライアント アプリケーションに最適化されたデータ モデル、サービス、およびサブシステムをそれぞれ用意することもできます。

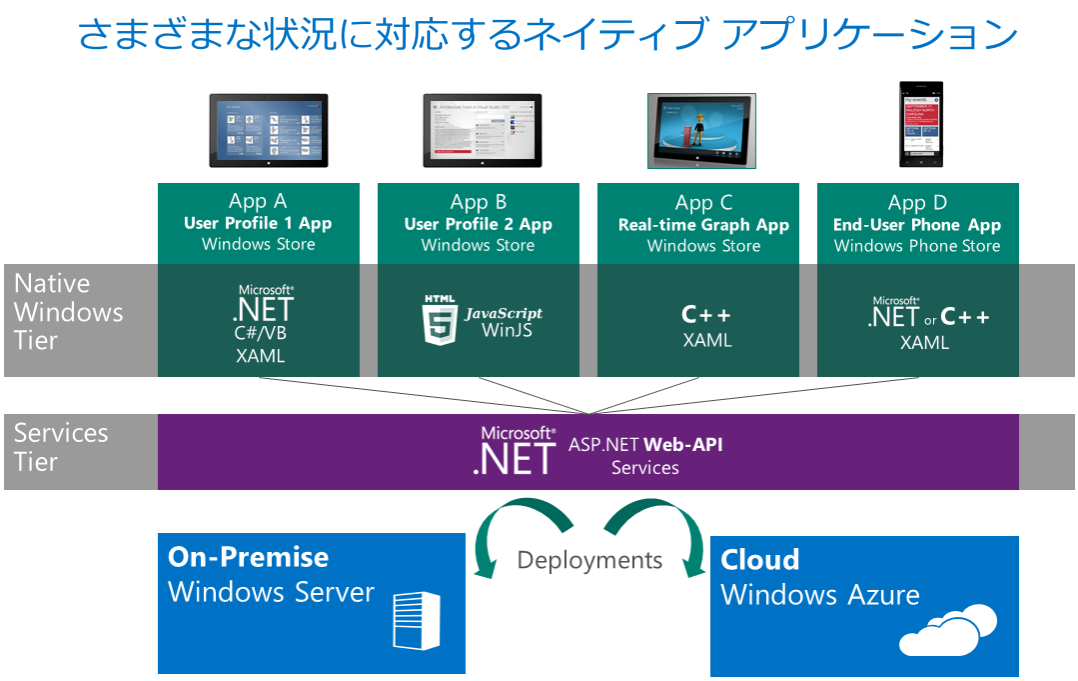

図 4-12

シナリオ: モバイル デバイス (タブレットとスマートフォン) 向けのモダン Web アプリケーション

図 4-13 は、任意のフォーム ファクターおよび任意のクライアント プラットフォーム (Windows またはサード パーティの OS) をターゲットとする場合の、典型的なモダン Web アプリケーション シナリオおよびテクノロジ連携パターンを示しています。

モバイル デバイスをサポートするモダン Web アプリケーションの開発にあたって Web 開発の柔軟性と完全な制御性を求める場合は、HTML5 対応の **ASP.NET MVC** の利用をお勧めします。また、比較的シンプルなデータ駆動型のシナリオについては、**LightSwitch プロジェクトの HTML5 クライアント**が適しています。

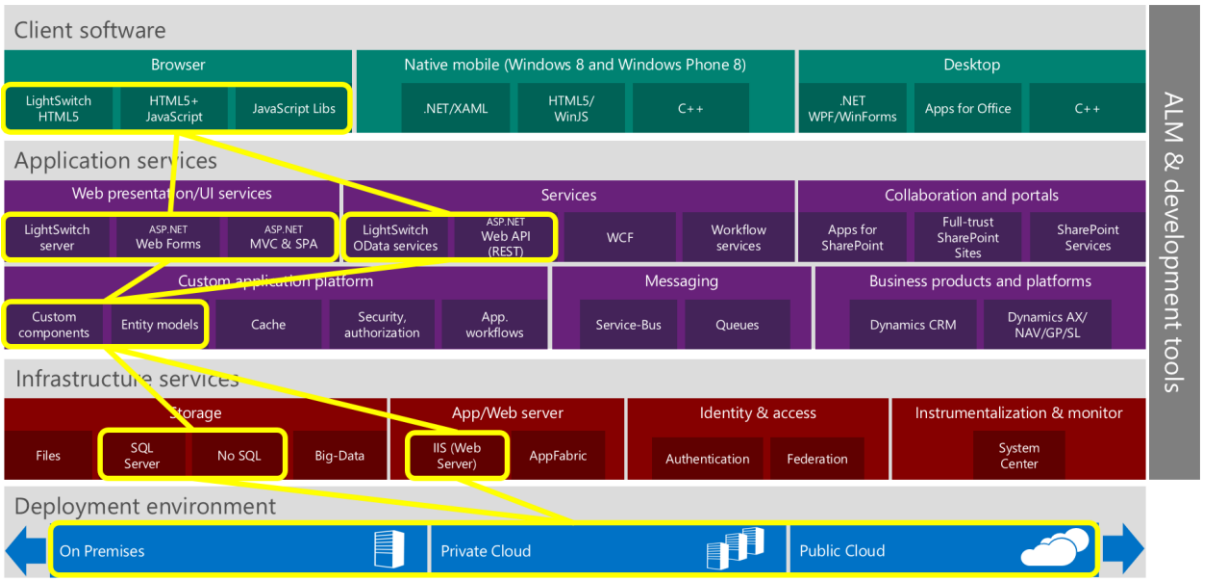

図 4-13

任意の Web ブラウザーおよびオペレーティング システムを対象としたモダン Web アプリケーションのアーキテクチャの簡易サンプル

Web アプリケーションの開発には、要件と優先事項に応じて、さまざまなアプローチを選択できます。データ駆動性の高いアプリケーションには、LightSwitch の使用をお勧めします。より複雑なシナリオには、ASP.NET MVC や一般的な HTML5/JS といった、きめ細かい構築が可能なテクノロジが適しています。

また、同じアプリケーション内に、優先事項や性質の異なる複数のモジュール (データ駆動型の UI Web とより複雑な UI Web) が含まれることもあるでしょう。こうした場合には、図 4-14 に示すように混合アプローチを採用するとよいでしょう。

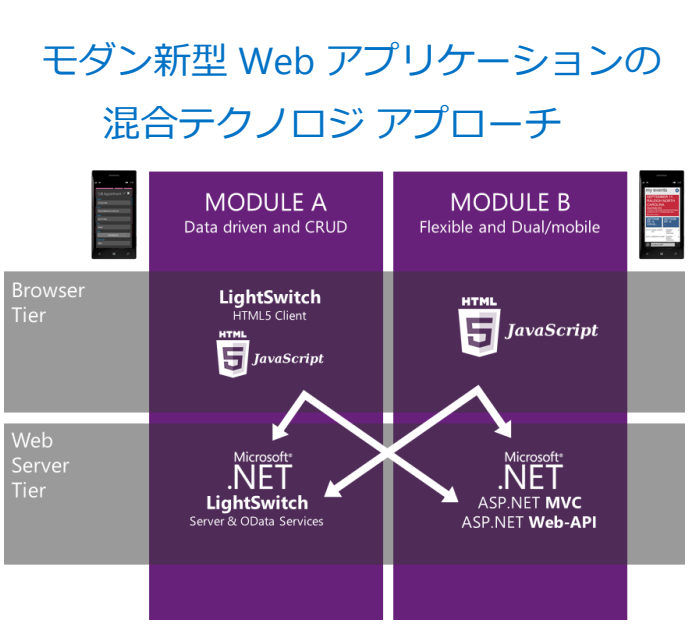

図 4-14

このシナリオでは、標準の Web テクノロジを使用することで、一部のモバイル デバイスだけでなく、あらゆるデバイスとオペレーティング システムをターゲットにすることが可能となります。

異なるフォーム ファクター (スマートフォンとタブレット) をターゲットとするアプリケーションでは、フォーム ファクターごとにレイアウトやビューを変えることでメリットがあるかどうかを検討してください。

図 4-15 に示すように、サーバーのレンダリング サービスは (ASP.NET MVC と LightSwitch の両方で)、多様なデバイスに対応した、HTML コードのレンダリング機能を提供しています。HTML のレンダリング方法は、Web ユーザー エージェントに基づいて、ブラウザーごとに変えることができます。この処理は低いレベルで行われるため、ASP.NET MVC の場合には、ユーザー エージェントごとに個別のファイルを作成する必要があります。LightSwitch では自動的に異なるレンダリング結果が生成されるため、マルチチャネルのアプリケーション開発が格段に容易になります。

フォーム ファクターに適応した HTML レンダリング

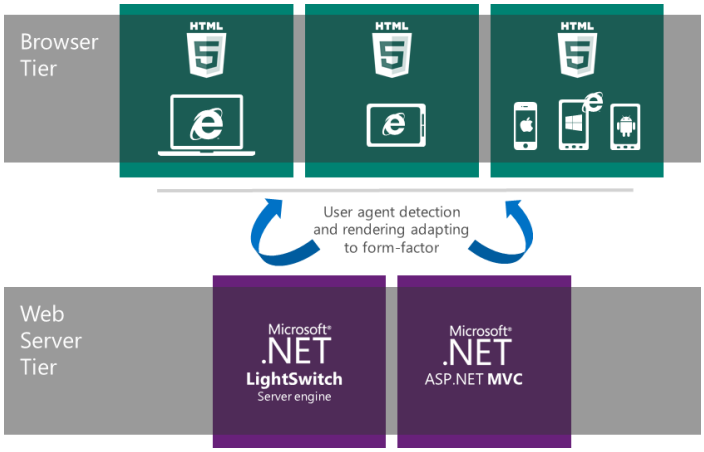

図 4-15

5. 従来型のアプリケーション パターン

モダン ビジネス アプリケーションを構築する際、重要となるのは新しいモバイル アプリケーションの作成だけではありません。各企業のコア アプリケーションが既に備えている価値を最大限に引き出すために、ユーザーから要求される新しいエクスペリエンスをビジネス プロセスと絶妙に統合させる必要があります。

このセクションでは、従来型のパターンで一般的に使用されているテクノロジについて説明すると共に、こうしたアプリケーションを拡張し、モダン ビジネス アプリケーションのコンセプトに対応するための推奨アプローチを紹介します。ここでは、企業内で見られるアプリケーションを大きく 2 つに分け、小規模/中規模アプリケーションと、ミッション クリティカルな大規模アプリケーションとして扱います。

優先事項によるビジネス アプリケーションの分類

このセクションでは、従来型のアプリケーション パターンをアプリケーションの優先事項別に整理して見ていきます。たとえば、小規模な部門内アプリケーションと、長期的に使用するコア ビジネスのミッション クリティカルなアプリケーションとでは、優先事項はまったく異なります。

こうしたビジネス上の優先事項の観点から、このセクションではビジネス アプリケーションを次の 2 つのカテゴリに分けて扱います。

- 小規模/中規模アプリケーション
- ミッション クリティカルな大規模アプリケーション

図 5-1 で示すように、ビジネス アプリケーションの優先事項を基準として、目的のアプリケーションがどのカテゴリに当てはまるかを判断できます。ここからさらに、具体的なニーズを精査したり、アプリケーション内のサブシステムがそれぞれどのカテゴリに属するかを見極めたりすることが可能です。

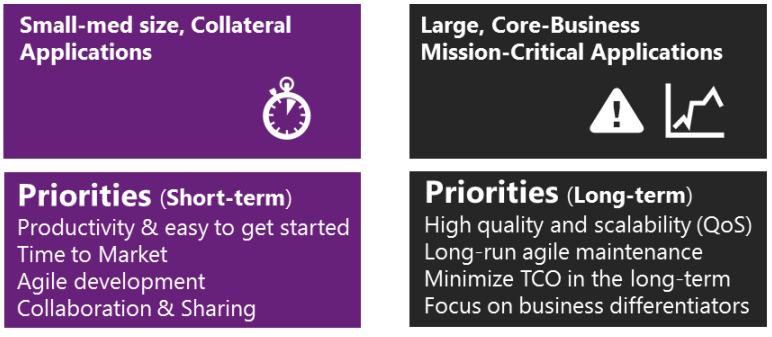

図 5-1

アプリケーションの規模は、一言で説明できるほど単純なものではありません。Twitter を例にあげてみると、ユーザー数の多さや、アーキテクチャの拡張性の高さという観点では、大規模アプリケーションだと考えることができますが、一方で、機能の少なさから小規模なアプリケーションとして見ることも可能です。

そこで、このガイドでは、(カスタム ERP ソリューションのような) 機能的な複雑さや、(Twitter のような) アーキテクチャ、拡張性、およびサービス品質 (QoS) 上の複雑さなど、複雑さの度合いを基準としてアプリケーションの規模を表すこ

とにします。上記のような複雑性が最小限に留められているアプリケーションについては、小規模/中規模のアプリケーションとして扱います。

もちろん小規模/中規模のアプリケーションは、全社的な観点ではミッション クリティカルでなかったとしても、組織の特定の範囲 (部門レベル) で非常に重要なものであるはずです。そうでなければ構築された意味がありません。

小規模/中規模のアプリケーション (またはミッション クリティカルではないサブシステム) では通常、開発の生産性や開発に簡単に取り掛かれるかどうか、さらには迅速な開発によってビジネス価値がすばやくもたらされるかどうかが優先事項として扱われます。

一方、ミッション クリティカルな大規模コア ビジネス アプリケーションには (またはアプリケーション全体ではなく、サブシステムのみの場合も同様に)、別の考慮事項と長期目標があります。コア ドメイン アプリケーションを長期にわたって発展させていく場合、新たなテクノロジ トレンドとの間で摩擦が生じることがあります。このカテゴリで重視されるのは、主力業務の差別化につながるソフトウェアを作成することです。さらにこうしたシナリオでは、同時ユーザー数の増加に対処するため、きわめて高度な QoS が要求される場合もあります。短期的な開発の生産性は必ずしも重視されませんが、長期にわたる機動性の高い保守は不可欠です。ミッション クリティカルな大規模コア ビジネス アプリケーションは絶えず進化するシステムであり、長期的な持続可能性および保守容易性がきわめて重要になります。

開発にあたっては、図 5-2 に示した各種の優先事項に応じて、開発およびアーキテクチャのさまざまなアプローチを検討し、そのアプローチと相性の良いテクノロジを見極める必要があります。

開発の機動性 (変化に迅速に適応すること) は**いずれの領域でも不可欠**ですが、アーキテクチャ上のアプローチや選択すべきテクノロジは、カテゴリごとに異なるのが一般的です。

ただし、ほとんどのアプリケーションは、どちらかのカテゴリに完全に当てはめることができません。

そのため、優先事項による分類は、アプリケーション全体についてだけではなく、サブシステムについても行う方が適切です。図 5-3 に示すように、多くのアプリケーションには、いくつかのコア ビジネス サブシステムのほかに、よりシンプルな構造の付帯的なサブシステムが含まれます。

サブシステムの種類に応じて、適切な対応方法と設計方法はまったく異なります。

従来の従来型のアプリケーション パターンの分類

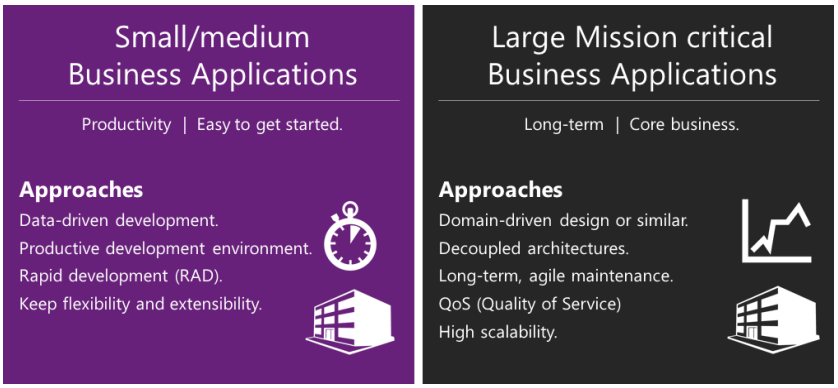

図 5-2

アプリケーション全体での分類とサブシステムごとの分類

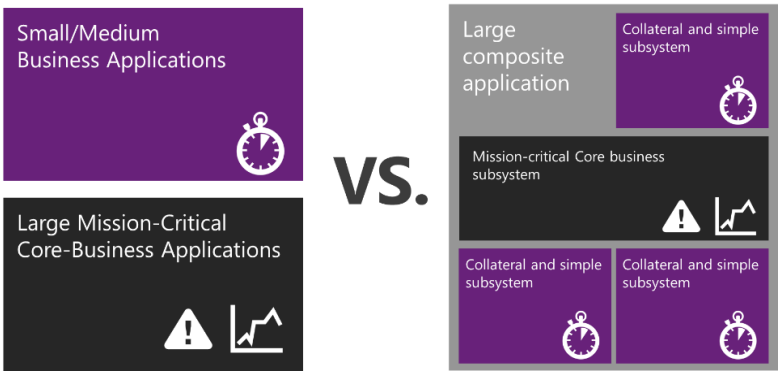

図 5-3

サブシステムは、"特定の境界によって区切られた、(専用のコードおよびモデル/データを持つ) アプリケーション内の差別化された領域" と定義できます。そのため、サブシステムはそれぞれ、多少調整を要するものの、別々の開発チームによって自律的に開発することが可能です。関係の整合性に基づいて明確に切り分けられたサブシステムは、ドメイン駆動設

計 (DDD) の境界づけられたコンテキストのコンセプトとよく似ています。このコンセプトについては、後ほど「ミッション クリティカルな大規模コア ビジネス アプリケーション」セクションで説明します。

このガイド内では、アプリケーションのカテゴリについて説明する場合、常にサブシステムのカテゴリも含まれている点に留意してください。実際の開発では、具体的なドメイン、コンテキスト、シナリオ、およびビジネス要件に応じて状況が異なります。

小規模/中規模ビジネス アプリケーション

社員や社内部門から成る管理された環境に対応する場合、ビジネス アプリケーションでは従来型のアプローチに従うのが通例です。このセクションでは、頻繁に変更され、ライフサイクルが比較的短い、小規模/中規模アプリケーションについて説明します。こうしたアプリケーションは主に、データ中心またはデータ駆動型のいわゆる CRUD (Create: 生成、Read: 読み取り、Update: 更新、Delete: 削除) シナリオに利用されるという特徴もあります。一般に、小規模/中規模ビジネス アプリケーションには図 5-4 で示すような優先事項があります。

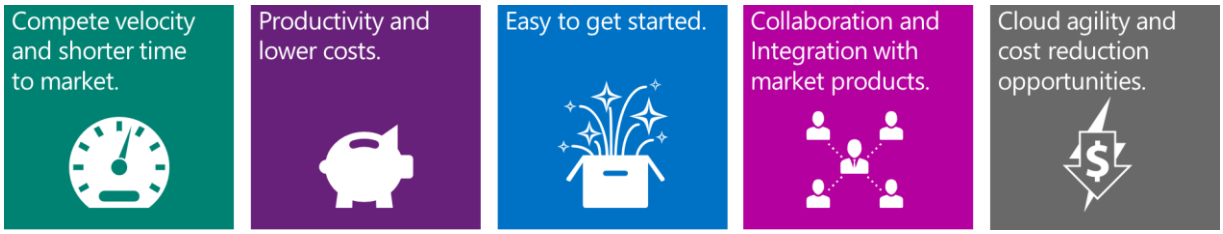

図 5-4

競争への迅速な対応と開発期間の短縮および継続的かつ段階的なエクスペリエンスの発展は、市場環境に見合った競争力あるビジネスを展開するうえで、IT 部門が対応するべき新しい基準です。

ビジネスの特定の分野で、アプリケーションを (わずか数か月などの) 短期間で作成して実稼働に移る必要がある場合、**生産性と低コスト性**が欠かせません。そのため、開発への**取り掛かりやすさ**を備え、操作の習得のコストを抑えられる開発プラットフォームを使用することが、最も有用性の高いアプローチになります。これによって、テクノロジ、アーキテクチャ、フレームワークに関して必要な意思決定の回数を減らすことができます。

開発ツールに関しては、実稼働用の ALM 環境 (アプリケーション ライフサイクル管理機能とソース コード リポジトリ) をすばやく作成できる、生産性が高く、シンプルでわかりやすい開発ツールを使用すると効果的です。たとえば、すぐに利用を開始できるクラウド/オンラインの ALM インフラストラクチャ (Visual Studio Team Foundation Service など) は、こうした種類のアプリケーションの開発に最適です。

Microsoft SharePoint および **Microsoft Dynamics** への統合、または埋め込みが可能なアプリケーションの構築にも高いニーズがあります。アプリケーションに、コラボレーション、共有、ドキュメント管理のトピック、または一般的なビジネス運営 (CRM および ERP) に関するその他の要件がある場合、その優先性はさらに高まります。

俊敏性とコストの削減というクラウドの利点は、アプリケーションを実稼働環境に発行する際、ビジネスの関係者によって機動性が求められる中規模のビジネス アプリケーションに最適な選択肢です。たとえば、2 か月で開発したアプリケーションが、実稼働環境で利用できるようになるまでに 1 か月もかかるようでは合理的とは言えません。そのため、きわめて多くの企業が、パブリック クラウドとプライベート クラウドを問わず、機動性の高い実稼働環境へとアプリケーションを移行しようとしています。図 5-5 は、小規模/中規模ビジネス アプリケーションのさまざまな背景をまとめたものです。

図 5-5

前述のように、**開発の生産性**は、小規模/中規模ビジネス アプリケーションにおいて通常最も重視される優先事項であり、**.NET** と **Visual Studio** が得意とする領域の 1 つでもあります。次に示す Forrester のレポートには、この点がわかりやすくまとめられています。

図 5-6

以降のセクションでは、次のシナリオについて説明します。

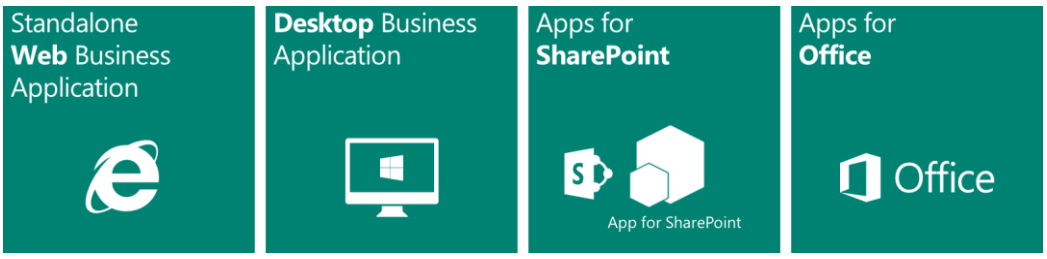

図 5-7

- **小規模/中規模のデータ中心のスタンドアロン ビジネス アプリケーション**
  - **データ中心の Web ビジネス アプリケーション**
    - *LightSwitch プロジェクトの HTML5 クライアント*
    - *HTML5 + ASP.NET (Web フォーム、MVC、SPA)*
  - **データ中心のデスクトップ ビジネス アプリケーション**
    - *WPF*
    - *Windows フォーム*
    - *LightSwitch Windows デスクトップ クライアント*

- **コラボレーション/生産性ビジネス アプリケーション**
  - *SharePoint* 用アプリケーション
  - *Office* 用アプリケーション

データ中心の Web ビジネス アプリケーション

データ中心のアプリケーションの大半は、マスターおよび詳細のシナリオを含む、CRUD 操作に対応しています。こうしたアプリケーションでは、ビジネス ロジックを実装する必要がありますが、対象範囲は広くありません。そのため、この種類の Web アプリケーションは、複雑なドメインと大量のビジネス ルールを扱う大規模アプリケーションとはならないのが一般的で、通常はきわめてデータ駆動性のきわめて高い、単純な構造をしています。こうしたアプリケーションで最も重視される優先事項は、生産性、コスト、およびビジネスへの価値です。開発ライフサイクルを短縮し、コストを比較的低く抑えながら、いかにビジネスに対して機動的に価値をもたらせるかが重要になります。

HTML5 は、Silverlight や Flash といった Web プラグインを介さず、Web でよく使用されるクライアント テクノロジです。HTML5 はあらゆるデバイス (PC、タブレット、スマートフォンなど) から使用でき、JavaScript (および jQuery を始めとする多くの強力な JavaScript ライブラリ) と CSS を多用します。

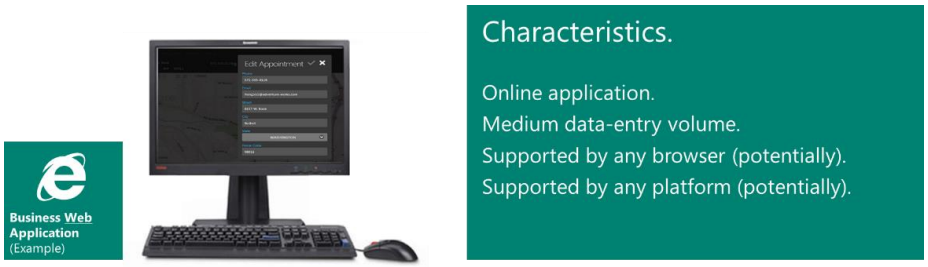

図 5-8

次の表では、各種のテクノロジ/アプローチを示し、作成する Web アプリケーションのコンテキストおよび優先事項に応じて、どのテクノロジをいつ使用するべきかを説明します。

ビジネス Web アプリケーションのプレゼンテーション層テクノロジのアプローチ

テクノロジ	選択すべき状況と理由
LightSwitch プロジェクトの HTML5 クライアント	**LightSwitch Web クライアント** ▪ 主にデータ駆動型 (CRUD) Web アプリケーション/モジュールに使用 ▪ すべての最新デバイス上で動作し、自動 HTML レンダリングの活用や、さまざまなフォーム ファクターへの適応が可能な、データを中心としたクロスブラウザーのモバイル Web クライアントを作成する最も容易な方法 ▪ jQuery Mobile などの OSS JavaScript ライブラリ、JavaScript、テーマ、および CSS3 を使用して拡張可能
HTML5 をサポートする ASP.NET **Web フォーム**	**HTML5 をサポートする ASP.NET Web フォーム** ▪ Web フォームを使い慣れた開発者が簡単に開発に着手でき、コードへの制御性も完全に維持されるアプローチ ▪ Modernizr などのライブラリを使用し、HTML5 の機能サポートの有無と回避策を検出 ▪ CSS3 を利用してスクリプトを短くし、保守しやすいコードを作成 ▪ クライアント側のプログラミングには JavaScript と jQuery を使用
ASP.NET **Web ページ**	**ASP.NET Web ページ** ▪ ASP.NET Web ページおよび Razor 構文は、サーバー コードと HTML を組み合わせて動的 Web コンテンツを作成するための、アプローチが容易で高速かつ軽量な手段を提供

HTML5 をサポートする ASP.NET **MVC**	**HTML5 をサポートする ASP.NET MVC** ▪ HTML のレンダリングを完全に制御でき、優れた単体テスト機能とフェイク機能を備えた、最も柔軟かつ強力なオプション ▪ Modernizr などのライブラリを使用し、HTML5 の機能サポートの有無と回避策を検出 ▪ CSS3 を利用してスクリプトを短くし、保守しやすいコードを作成 ▪ クライアント側のプログラミングには JavaScript と jQuery を使用 ▪ LightSwitch や Web フォームがターゲットとするプロジェクトと比べ、より複雑なプロジェクトに適する

表 5-1

参考資料

ASP.NET Web フォームの概要	http://www.asp.net/web-forms (英語)
LightSwitch アーキテクチャについて学ぶ	http://msdn.microsoft.com/en-us/vstudio/gg491708 (英語)

シンプルなデータ駆動型ビジネス Web アプリケーションのシナリオでは、開発の生産性が特に重視されます。要件が満たされる場合には、できるかぎり LightSwitch および HTML5 クライアントを組み合わせて使用することをお勧めします。データ モデルを作成するときや、マスターおよび詳細のシナリオを含む CRUD (Create: 生成、Read: 読み取り、Update: 更新、Delete: 削除) 操作の画面を作成するときには、LightSwitch を使用すると作業がきわめて容易になります。多くの場合にはコードを記述する必要もありませんが、LightSwitch が提供する多くの拡張ポイント (JavaScript および .NET の拡張ポイント) を使用することで、必要に応じてコードを追加し、スタイルのカスタマイズと操作の拡張を行えます。

ただし、LightSwitch を使用する場合は、アプリケーションの実行に LightSwitch ランタイムが必要になります。このランタイムは、HTML5 クライアントを使用する場合、サーバー上の ASP.NET および OData サービスと、クライアント上の JQuery Mobile をベースに構築された、エンドツーエンドの透過的なエンジンによって構成されます。LightSwitch では、表示対象のデータ コントロールに基づいて画面が自動的にレンダリングされるため、目的のアプリケーションがデータ駆動型ではなく、より複雑な UI が画面の大部分を占める場合、LightSwitch は最適な選択肢とは言えません。

このアプローチの特長は、LightSwitch の機能がアプリケーション要件に高い割合で合致している場合に、開発の生産性が飛躍的に向上する点です。

データ中心のシンプルな Web アプリケーションを作成するための、初期開発の生産性に優れたもう 1 つの手段として、**ASP.NET Web フォーム**が挙げられます。Web フォームでは、見てわかる Web コントロールをドラッグ アンド ドロップして位置を確認できるほか、使いやすいグリッド コントロールが提供されます。また、表形式と相性の良いシンプルなデータ バインドを使用して、ユーザーが簡単にレコードを更新できるようにする場合にも役立ちます。さらに、Windows フォーム や WPF を使用した開発など、デスクトップ アプリケーションの開発経験が豊富な開発者にとって、Web フォームは通常、きわめて簡単に使用可能です。

最も柔軟でパワフルな Web UI を追求したい (ただし、習得しやすさは追求しない) といったさらに高度なシナリオでは、**ASP.NET MVC** と HTML5/JavaScript の併用をお勧めします。JavaScript を多用するアプリケーションでは、**ASP.NET SPA** (Single Page Application) のような高度なアプローチを検討してもよいでしょう。

シナリオ: エンドツーエンドの小規模/中規模 Web ビジネス アプリケーション

図 5-9 は、小規模/中規模 Web ベース ビジネス アプリケーションの、典型的なテクノロジ連携パターンを示しています。LightSwitch や ASP.NET などいずれのテクノロジ アプローチを選択した場合にも、HTML5/JavaScript は、現代の Web 開発で常に使用されている Web クライアント テクノロジです。

シナリオ: エンドツーエンドの
小規模/中規模 Web ビジネス アプリケーション

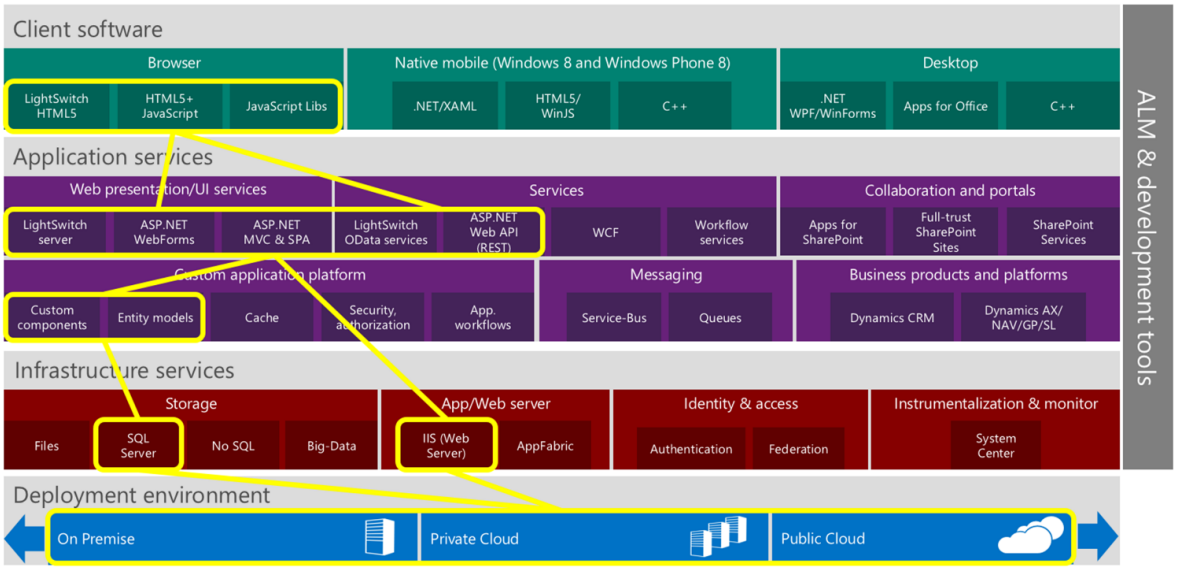

図 5-9

各テクノロジの詳細、およびサービスを構築する際にテクノロジを使い分ける基準は、既に説明した内容とほぼ同じです。サービス開発テクノロジの選択肢 (ASP.NET Web API、WCF など) の詳細について、ここでは再度紹介しないこととしますので、「次世代型のアプリケーション パターン」の「サービス」セクションを参照してください。

小規模/中規模 Web ビジネス アプリケーション向けの混合アプローチ

開発する Web アプリケーションの要件と優先事項によっては、選択すべきアプローチが明白な場合があります。たとえば、データ駆動型アプリケーションには LightSwitch が、より複雑なアプリケーションには ASP.NET MVC や一般的な HTML5/JS といったきめ細かい構築が可能なテクノロジが適しており、開発者の経験とスキルに応じて ASP.NET Web フォームも使用できます。

ただし、ここで思い出していただきたいことは、1 つのアプリケーションには多くの場合、複数のサブシステムが含まれているという点です。こうしたケースでは、同じアプリケーション内で、図 5-10 に示すような混合アプローチを使用するのが適切です。たとえば、LightSwitch で CRUD ベースの操作を開発する割合が 40%、よりきめ細かいテクノロジ (ASP.NET MVC など)

Web ビジネス アプリケーション向けの
混合 Web テクノロジ アプローチ

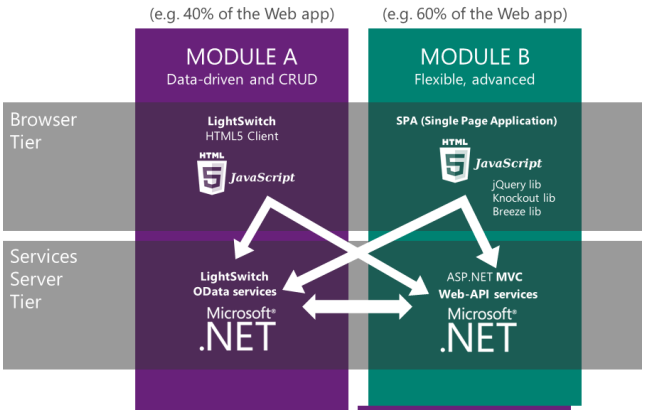

図 5-10

および高度なアプローチ (SPA など) を使用する割合が 60% のアプリケーションがあるとします。SPA を使用すると、HTML5、JavaScript、jQuery、およびその他の JavaScript ライブラリを活用し、アプリケーションの最も重要かつ動的な領域を構築できます。この 60% にあたる部分は、要件や開発者の得意分野に応じ、ASP.NET MVC や Web フォームを使用して開発することも可能です。LightSwitch の拡張ポイントを使用すれば、JavaScript (LightSwitch のクライアント層) または .NET サーバー コード (LightSwitch の中間層) を通じて外部サービスを利用することもできます。

データ中心のデスクトップ ビジネス アプリケーション

アプリケーションで大量のデータ入力を行う場合や、ローカル ストレージ、COM との相互運用性、および自動処理が関係する複雑なオフライン シナリオに対応する場合などは、Web アプリケーションではなくデスクトップ アプリケーションを構築する場合が多いでしょう。また、エンド ユーザーが業務上のニーズとスキルセットに基づいて、デスクトップ アプリケーションを好むケースもあるかもしれません。

図 5-11

デスクトップ アプリケーションのアーキテクチャに応じて、特定のテクノロジが必要となります。従来、デスクトップ アプリケーションには、図 5-12 に示す 2 階層および 3 階層のスタイルのアーキテクチャが最も一般的に使用されています。

図 5-12

小規模/中規模デスクトップ アプリケーションのクライアント層開発に使用されるテクノロジとして、ここでは以下の 3 つを取り上げます。

- .NET WPF
- .NET Windows フォーム
- デスクトップ用 LightSwitch

次の表では、開発するアプリケーションの優先事項および個別のコンテキストに応じて、どのテクノロジをいつ使用するべきかを示します。

小規模/中規模 デスクトップ ビジネス アプリケーションのプレゼンテーション層テクノロジのアプローチ

テクノロジ	選択すべき状況と理由
.NET WPF	**.NET Windows Presentation Foundation (WPF)** ▪ 複雑な UI や、スタイルのカスタマイズを必要とし、グラフィックスを多用するようなデスクトップ シナリオで推奨される、Windows ベース デスクトップ アプリケーション向けのテクノロジ。WPF では XAML のビューも活用。WPF の開発スキルは Windows ストア アプリの開発スキルと似通っているため、Windows ストア アプリへの移行は、Windows フォームからよりも WPF から行う方が容易 ▪ .NET 4.5 の新しいシンプルな非同期機能 (async/await) を活用 ▪ ASP.NET SignalR .NET クライアントを活用し、クライアントとサーバー間の双方向通信を実現
.NET Windows フォーム	**.NET Windows フォーム** ▪ デスクトップ アプリケーションの構築のため、.NET Framework で初めて提供された UI テクノロジであり、現在も多くのビジネス デスクトップ アプリケーションに活用可能。使いやすく、WPF よりも軽量なため、UI スタイルのカスタマイズを必要としないシンプルなシナリオに適する ▪ WPF よりもシンプルな UI 機能を提供し、XAML に基づいていないため、Windows ストア アプリへの移行には制約がある ▪ WPF と同様に、async/await を使用した非同期プログラミングなど、最新の .NET 機能の利用が可能
LightSwitch デスクトップ クライアント	**LightSwitch デスクトップ クライアント (ブラウザー不使用)** ▪ LightSwitch 中間層および HTML5 によるソリューションを既に採用している場合、LightSwitch が標準でサポートするデスクトップ エクスペリエンスを使用可能

表 5-2

参考資料

WPF デスクトップ アプリケーション	http://msdn.microsoft.com/ja-jp/library/aa970268.aspx (機械翻訳)
	http://msdn.microsoft.com/ja-jp/library/ms754130.aspx (機械翻訳)
Windows フォーム	http://msdn.microsoft.com/ja-jp/library/dd30h2yb.aspx (機械翻訳)
LightSwitch	http://msdn.microsoft.com/ja-jp/vstudio/ff796201

シナリオ: 小規模/中規模の 2 階層デスクトップ アプリケーション

図 5-13 は、WPF/Windows フォーム、カスタム コンポーネント、およびデータ アクセス コードが同一クライアント層 (デスクトップ PC) 内で実行される、**2 階層アプローチを使用した小規模/中規模デスクトップ ビジネス アプリケーション** の典型的なテクノロジ連携パターンを示しています。

シナリオ: 小規模/中規模の 2 階層デスクトップ アプリケーション

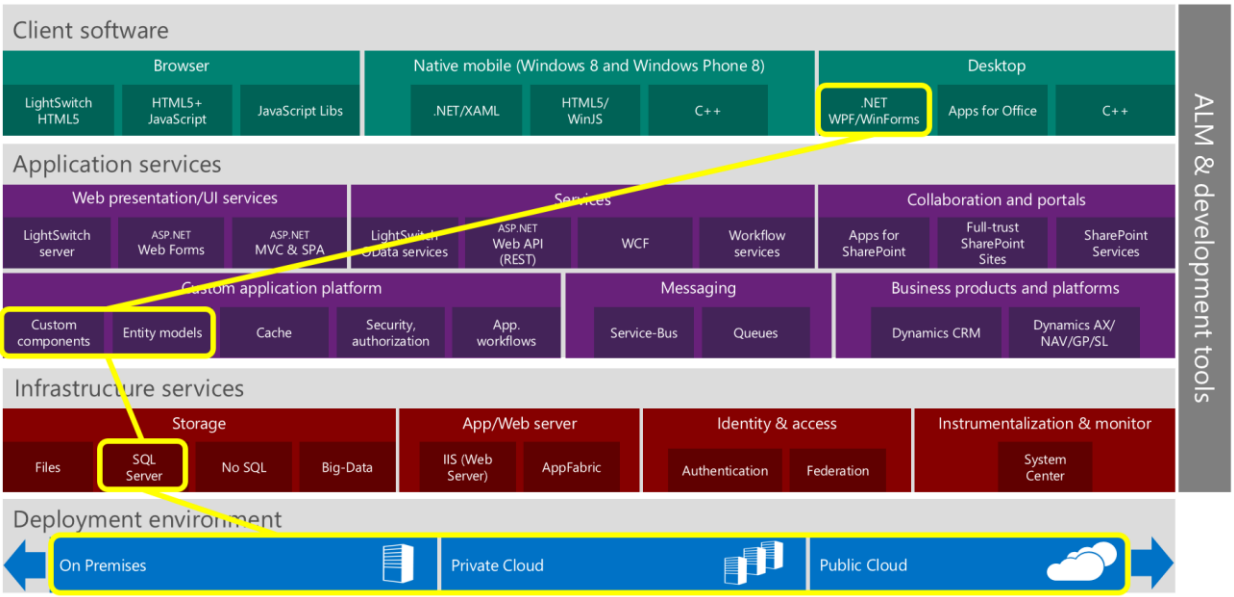

図 5-13

クライアント PC 内のクライアント データベース ドライバーとの直接的な依存性が生じる問題や、クライアント PC とデータベース サーバーの間にファイアウォールがある場合に生じる問題、データベース サーバーによって拡張性が制限されるといった問題があるため、多くの場合、**旧式の 2 階層アーキテクチャ アプローチ (従来型のクライアント/サーバー方式) の使用は推奨されていません**。それでもこのシナリオは、依然として従来型のアプリケーションで一般的に使用されているため、ここに取り上げています。このシナリオでは、デスクトップ クライアントに WPF を使用し、Entity Framework や ADO.NET といったデータ アクセス テクノロジを使用して、データ ソース (SQL Server などのデータベースが一般的) に直接アクセスします。

こうした 2 階層アプローチを適用する場合は、少なくともアプリケーションおよびビジネス ルール コードをデータ アクセス コードと分離し、別々の内部レイヤーに配置するなどの対策により、**.NET WPF コード内で関心の分離を実践するようお勧めします**。これを怠ると、デスクトップ アプリケーションがモノリシックに拡大していくため、保守が非常に難しくなります。

シナリオ: 小規模/中規模の 3 階層デスクトップ アプリケーション

図 5-14 は、3 階層アプローチを適用した、小規模/中規模デスクトップ ビジネス アプリケーションのテクノロジ連携パターンの例を示しています。

シナリオ: 小規模/中規模の 3 階層デスクトップ アプリケーション

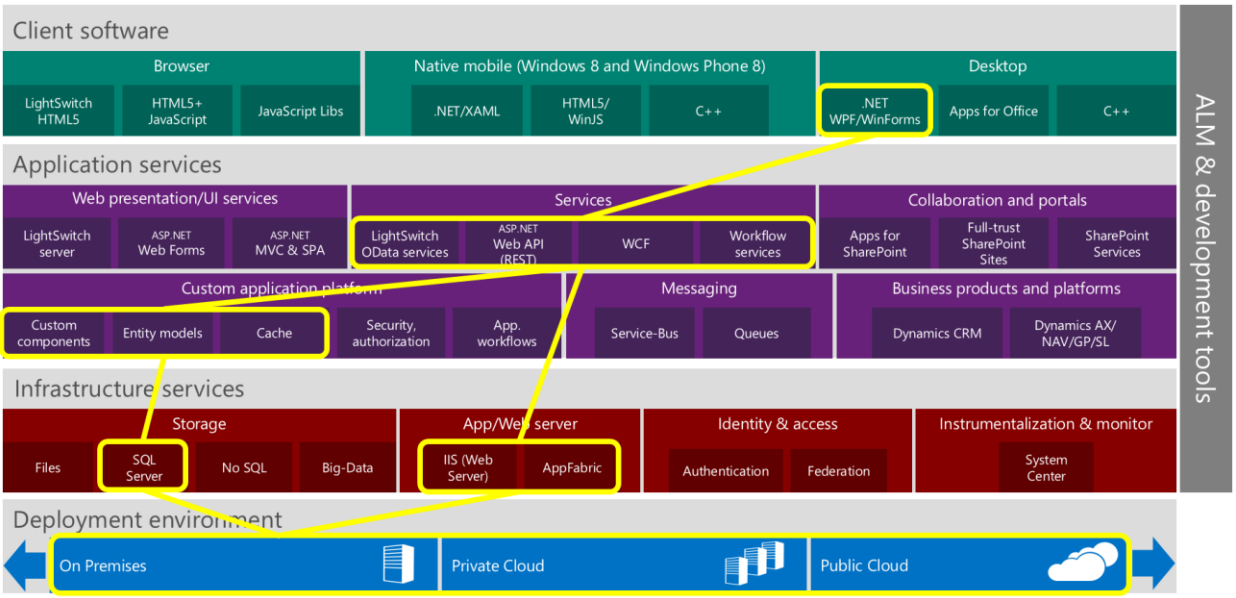

図 5-14

目的とするデスクトップ アプリケーションの要件および優先事項を考慮して、デスクトップ クライアントには WPF を、データ駆動性の高いアプリケーション (CRUD シナリオ) には OData サービスまたは LightSwitch OData サービスを使用します。また、RPC/コマンド指向のサービスには、WCF を使用できます。

サービスに応じてテクノロジを使い分ける基準は、既に説明した内容とほぼ同じです。サービス開発テクノロジの選択肢 (ASP.NET Web API、WCF など) の詳細については、「次世代型のアプリケーション パターン」の「サービス」セクションを参照してください。

特に、データ アクセス (CRUD) のみを主とするのではなく、ビジネス ルールも埋め込んだサービスを構築する場合は、ビジネス ルール コードとデータ アクセス コードを分離し、その各コードを、サービス層にある別々の内部レイヤーに (関心の分離の原則に従って) 配置することをお勧めします。

データ層へのアクセスに使用するテクノロジには、基本的に **Entity Framework** を推奨します。こうした比較的シンプルなデータ駆動型シナリオの場合、専用のビジュアル デザイナーと、**Model First** または Database first のアプローチを通じて Entity Framework を使用するのが最も一般的です。ただし、エンティティ クラスの完全な制御を必要とする場合は、シンプルに "Code First POCO エンティティ クラス" を使用することもできます。

デスクトップ ビジネス アプリケーションの刷新

デスクトップ上でのエクスペリエンスを構築する際は、利用者が寄せている新たな期待にも配慮するべきです。こうした期待に応えられるよう、.NET ではデスクトップ アプリケーションに関する複数の最新テクノロジを取り入れると共に、アーキテクチャを変更することなく新しいプラットフォームへとアプリケーションを拡張し、コードの再利用もできる機能を提供しています。次の推奨事項に留意してデスクトップ アプリケーションを構築すれば、アプリケーションの寿命が延び、新しいデバイスへの拡張も容易になるだけでなく、将来的にアプリケーション全体を移行することも可能になります。

- **Model-View-ViewModel (MVVM) 設計パターンを使用する:** マイクロソフトのクライアント プラットフォーム (WPF を含む) では、MVVM パターンによるアプリケーションの構築を簡単に行えます。このパターンを使用すると、状態および動作と画面表示とが明確に分離されるため、クリーンで保守容易性に優れたコードを作成し、複数のデバイス間で簡単に共有できるようになります。
- **クライアント ロジックに、ポータブル クラス ライブラリを使用する:** .NET のポータブル クラス ライブラリは、デスクトップ、Windows ストア アプリ、Windows Phone アプリケーションなど、複数のプラットフォーム間でバイナリを共有する機能を提供します。クライアント ロジックの実装に .NET のポータブル クラス ライブラリを使用すると、複数のプラットフォーム上で複数のエクスペリエンスを作成する場合でも、作業が格段に容易になります。
- **ユーザー エクスペリエンスを刷新する:** 今日のエンド ユーザーが求める各種のコンセプトは、.NET に取り入れられたデスクトップの最新テクノロジによって実装することができます。"軽快かつ柔軟"、"真のデジタル化を目指す"、および "より少ない要素でより大きな効果を上げる" といった設計原則を既存のデスクトップ アプリケーションに適用するには、XAML 画面デザインにモダン UI を採用し、アニメーションを慎重に使用すると共に、.NET の非同期プログラミングを広範に実装します。
- **ビジネス ロジックをサーバーに移行する:** 2 階層アプリケーション (クライアント/サーバー方式) は、新しいデバイス向けに拡張しようとするときわめて手間がかかります。この対応策として、ビジネス ロジックをサービスごとに明確に切り離し、後から他のデバイスおよびフォーム ファクターで再利用できるようにしておくことをお勧めします。
- **クラウドへの拡張を図る:** クライアントから分離したビジネス ロジックは、Windows Azure が提供する複数のソリューションを利用して、クラウドへと移行することができます。ロジックをクラウド サービスに変換することにより、既存ソリューションの弾力性と拡張性が著しく向上し、複数デバイス アプローチへの対応が可能になります。

Visual Studio パートナーからも、既存の .NET アプリケーションを刷新するのに役立つ各種のテクノロジが提供されています。

.NET アプリケーションの刷新に対応した Visual Studio パートナー製品

パートナー	選択すべき状況と理由
Xamarin	**Xamarin** は、Windows または Windows Phone をターゲットとした既存アプリケーションの C# コードを、iOS/Android デバイスと共有する手段を提供。基盤となる API にアクセスしてデバイス専用のビューを作成するほか、デバイス間でクライアント ロジックを再利用することが可能
ITR-Mobility の **iFactr** および **MonoCross**	**ITR-Mobility** は、iFactr や MonoCross など、C# でエンタープライズ モバイル アプリケーションを構築して主要モバイル プラットフォーム上で提供するためのソリューションを提供。抽象 UI やエンタープライズ データ同期といったサービスを提供し、広範なデバイスに対応するビジネス アプリケーションを実現
ArtinSoft の Mobilize.Net	**Mobilize.Net** は、既存のソース コードを新しいコードに変換することで、出力先アプリケーションのランタイムを使用することなく、Web、モバイル、クラウドといったモダン プラットフォームにレガシ アプリケーションを移行するためのソリューションとサービスを提供
Citrix の Mobile SDK for Windows Apps	**Citrix が提供する Mobile SDK for Windows Apps** を使用すると、リッチな開発ツール キットを活用して Windows 向け LOB アプリケーションをモバイル化できるほか、セントラル サーバー (Citrix XenApp/XenDesktop) 上で実行され、Citrix Receiver によって任意のモバイル デバイスからアクセスできる、タッチ操作対応の新しいアプリケーションを作成可能

表 5-3

参考資料	
Xamarin	http://xamarin.com/features (英語)
ITR-Mobility の iFactr および MonoCross	http://itr-mobility.com/products/ifactr (英語) http://monocross.net/ (英語)
Citrix の Mobile SDK for Windows Applications	http://www.citrix.com/mobilitysdk/ (英語)
Mobilize.Net	http://mobilize.net/ (英語)
VSIP パートナー ディレクトリ	https://vsipprogram.com/Directory (英語) の VSIP パートナー ディレクトリでは、Visual Studio のパートナーが提供する各種のソリューションを紹介

RIA コンテナー ベースのアプリケーションの刷新

わずか数年前、リッチ インターネット アプリケーション (RIA) のコンテナーおよびプラグインが流行していたころ、IT を取り巻く状況は今とはまったく違っていました。RIA によってほとんどの開発ニーズがまかなわれていた 5 年前、そのターゲットは Windows ベースの PC と Mac コンピューターのみでした。2010 年の "デバイス革命" 以降、さまざまなオペレーティング システム (Windows 8、Windows Phone 8、iOS、Android、Chrome OS など) を使用するさまざまなデバイス (タブレット、スマートフォン、およびコンピューター) が登場しましたが、その多くは RIA プラグインをサポートしていません。

同時に、以前ならプラグインを必要としていたような、よりリッチなシナリオが、HTML5 によってサポートされるようになりました。HTML5 は現在、あらゆるデバイスに広くサポートされており、従来型のプラグインに代わって、クロスプラットフォームのクライアントを開発する優れた手段となっています。

各デバイス固有の機能をフル活用して最も魅力的なカスタマー エクスペリエンスを実現できるため、市場ではネイティブ アプリケーションの提供も増加傾向にあります。

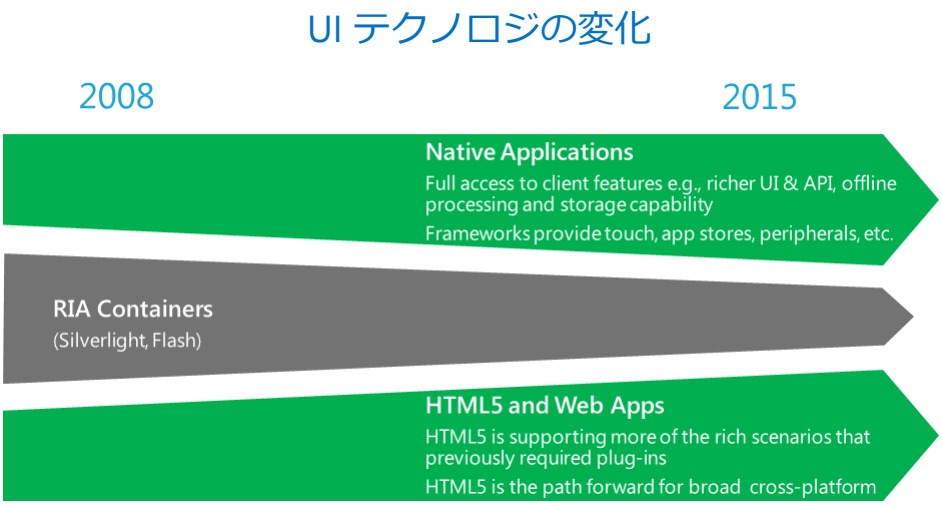

図 5-15

マイクロソフト プラットフォームは、ユーザー インターフェイスの 3 種類のアプローチ (Web、ネイティブ、および RIA) をすべてサポートしていますが、複数のデバイスに対応するモダン エクスペリエンスの開発には、主に Web テクノロジとネイティブ テクノロジが使用されることも考慮されています。どの段階でこうした移行に踏み切るかは、最終的にはそれぞれの要件とニーズに基づいて決定しますが、マイクロソフトは、お客様の選択された移行アプローチをプロセス全体にわたってご支援します。

- **ネイティブ アプリケーション**に移行する場合は、任意の Windows デバイス上で XAML/.NET をネイティブのターゲットとすることにより、既存のスキルはもちろん、コードも再利用できます。ポータブル クラス ライブラリを使用して、Silverlight を含めた複数のプラットフォーム間でバイナリを共有することも可能です。
- **ブラウザー ベースの HTML5 アプリケーション**については、最新の標準に基づいてあらゆるデバイスに対応したアプリケーションを作成できるよう、優れたツールとフレームワークを提供しています。Silverlight と HTML の相互運用性を活用すれば、ハイブリッド アプリケーションによる段階的な移行も可能です。
- 依然として **Silverlight によってのみサポートされる**シナリオ (ビデオ コンテンツの保護など) をターゲットとする場合や、今のところアプリケーションに次世代型のパターンを適用する必要のない場合は、Silverlight を引き続き使用できます。Silverlight は、安定性に優れる成熟したテクノロジです。既存の投資を最大限に活用し、HTML5 またはネイティブ ソリューションに段階的に移行できるよう、最新バージョン (Silverlight 5) には 10 年間の延長サポートが用意されています。

UI テクノロジの変化

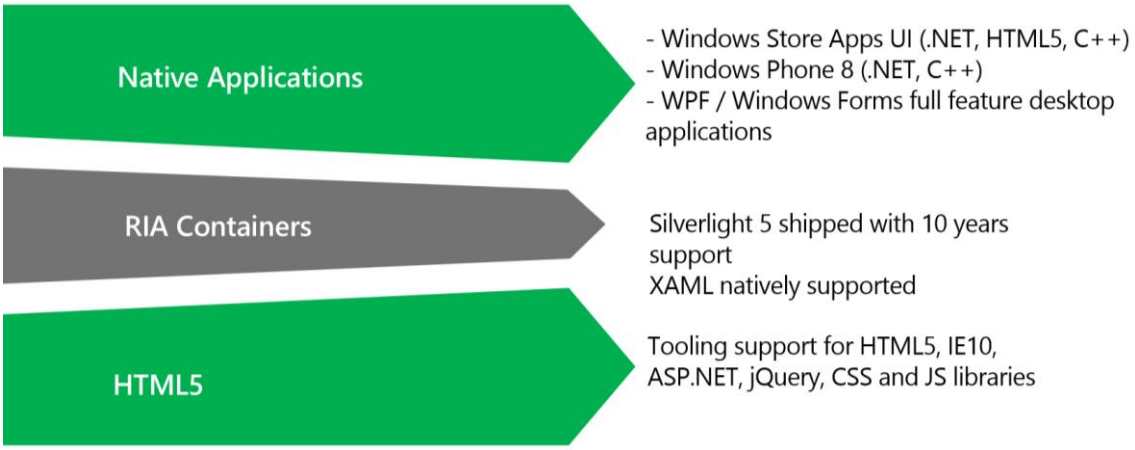

図 5-16

Silverlight の移行パスの詳細および推奨事項については、このガイドの末尾にある付録を参照してください。

Office および SharePoint のクラウド アプリケーション モデル

Office および SharePoint を軽量なアプリケーションにより拡張できるよう、Office 2013 にはクラウド アプリケーション モデルが導入されています。これにより、開発したビジネス アプリケーションの機能を、利用者が既に使用している Office アプリケーションを通じて提供できるようになりました。このクラウド アプリケーション モデルは、HTML、CSS、JavaScript、REST、OData、OAuth といったクライアント上で使用される標準の Web テクノロジと、ASP.NET などの、サーバー上で使用される任意のサーバー テクノロジに基づいて構築します。そのため、Web 開発者は、既存のスキルを使用してアプリケーションを構築し、使い慣れたツールや言語、ホスティング サービスを活用することができます。アプリケーションを開発した後は、クラウド内ですばやく展開、更新、および保守を行うことができ、Office ストアで公開して販売したり、内部のアプリケーション カタログを使用して IT 部門の承認済みアプリケーションを自社内に配布したりすることも可能です。

こうした統合アプリケーション モデルは、次の種類のアプリケーションに適用されます。

- **Office 用アプリ** (Office 2013、Office 365、Project Professional 2013、Word 2013、Excel 2013、PowerPoint 2013、Outlook 2013、Outlook Web App、Excel Web App、および Exchange Server 2013 に対応)
- **SharePoint 用アプリ** (SharePoint Server 2013、および Office 365 の SharePoint Online に対応)

Office 用アプリ (Apps for Office)

Office 用アプリ (Apps for Office) は、Office および SharePoint のクラウド アプリケーション モデルに基づいています。

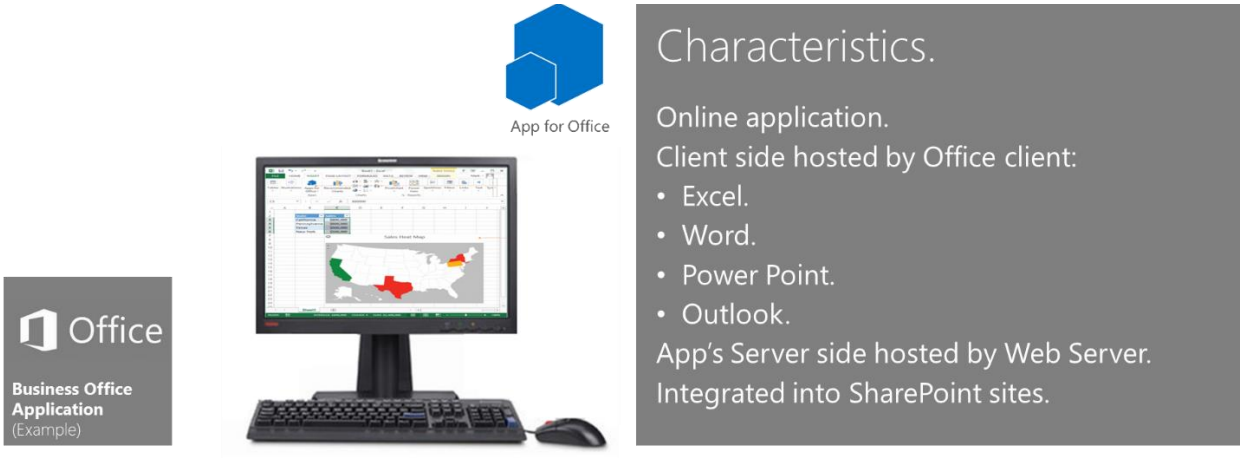

図 5-17

Office 用 アプリは、主に HTML Web ページと XML アプリケーション マニフェストで構成されています (図 5-18 を参照)。

Office 用アプリの構造

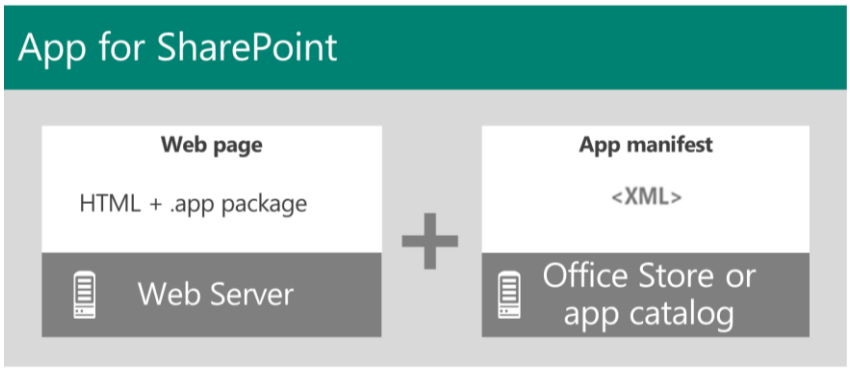

図 5-18

各種 Web 標準を使用した HTML コンテンツは、Office ドキュメント内の作業ウィンドウまたはコンテンツとして表示することができ、こうした Office コンテンツには、JavaScript からプログラム経由でアクセスできます。

アプリケーション マニフェストは、パブリックな Office ストア内またはプライベートなアプリケーション カタログ内でアプリケーションを公開し、発見しやすくするために必要です。

作成可能な Office 用アプリには、次の種類があります。

- **作業ウィンドウ アプリ:** Office クライアントの作業ウィンドウ内に表示されるアプリケーション
- **コンテンツ アプリ:** Office ドキュメントのコンテンツの内部に表示されるアプリケーション
- Outlook 2013 および Outlook Web Access 用の**メール アプリ:** 開いている Outlook アイテムの隣に表示されるアプリケーション (電子メール メッセージ、会議出席依頼、会議出席依頼への返信、会議の取り消し、予定など)

図 5-19 に、(Excel、Outlook、および Word での) Office 用コンテンツ アプリの例を示します。これらのアプリケーションに表示された特殊なコンテンツはすべて、IFrame 内でホストされている Office クライアントの要素に JavaScript コードを介してアクセスすることで、Office クライアント内に埋め込まれた HTML ページを使用して作成された要素です。

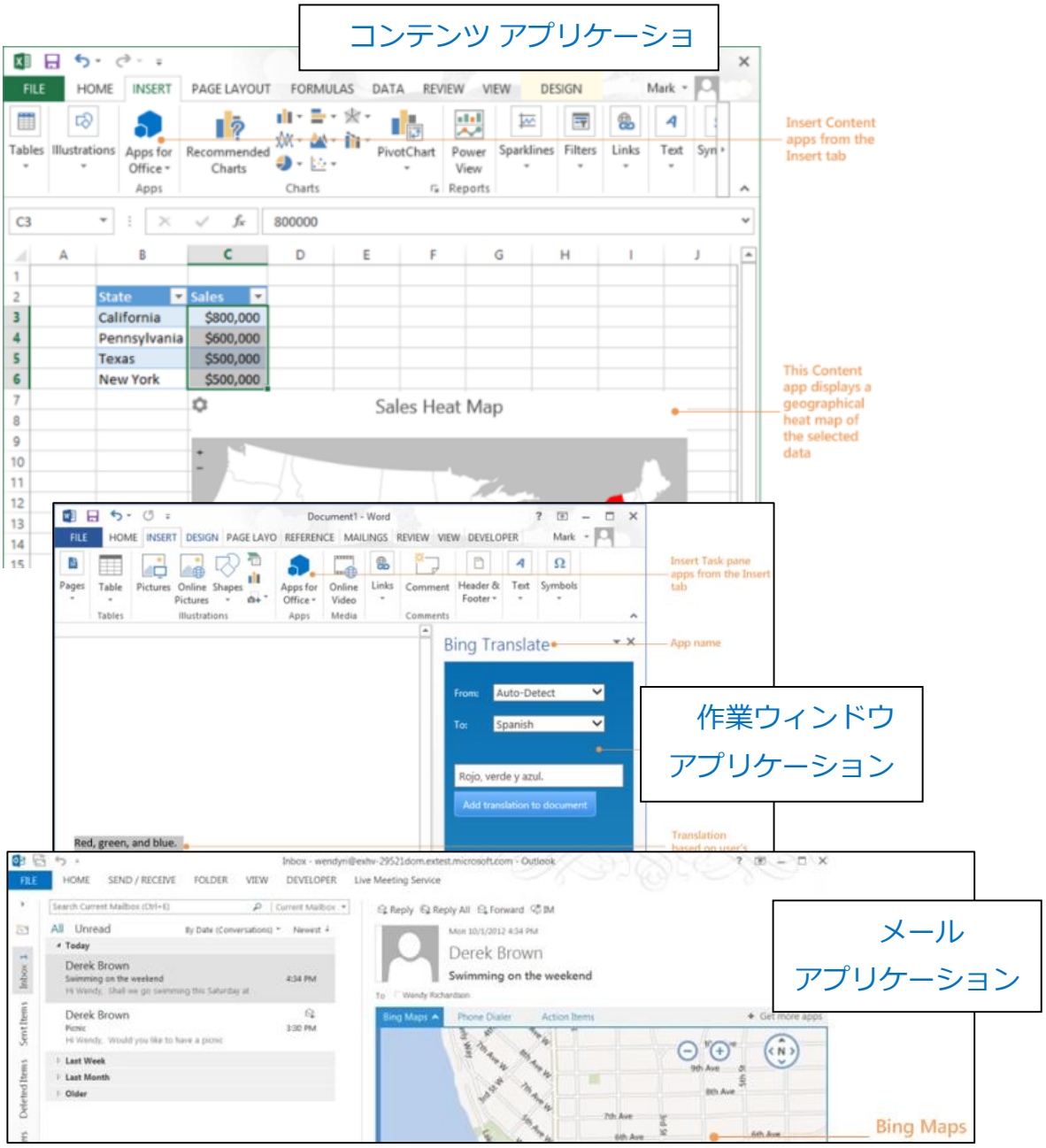

図 5-19

この新しいアプリケーション モデルを、従来型の VSTO (Visual Studio Tools for Office) アプリケーション モデル (アドインまたはドキュメント ベース) の代わりに使用すると、主に次のような利点があります。

- Office Web Apps がサポートされる
- Office RT がサポートされる
- 今後のバージョンの Office への移行が容易になる

さらに、この新しいアプローチを採用することで、Web 開発者は既存のスキルを活用して簡単に Office 用アプリを作成できます。

次の 2 つの表に、Microsoft Office クライアントに埋め込まれるアプリケーションを開発する際の、アプリケーションの優先事項および特定のコンテキストに応じた関連テクノロジを示します。ここでは、Office モードの新しいアプリケーションに関してだけでなく、レガシ テクノロジの情報も紹介しています。

Office 開発時のテクノロジの選択肢

テクノロジ	選択すべき状況と理由
Office 用アプリ モデル - Office Developer Tools - "Napa"	**Office に埋め込まれる HTML5/JavaScript Web クライアントの開発** ▪ 新しい Office 用アプリを作成し、将来的な Office の拡張に備えるうえで推奨されるテクノロジ ▪ Office 2013 用の開発では既定として使用 ▪ Web 経由または Office RT からのドキュメントへのアクセスが可能 ▪ アプリケーションをオンラインで実行し、アプリケーションをホストするサーバーにアクセスする場合に適する
Office アドイン - Visual Studio **Tools for Office (VSTO)**	**VSTO を使用した、Office アドインおよびドキュメント ベースのアプリケーション (クライアント内の .NET マネージ コード)** ▪ Office 2013 以前のバージョンの Office で使用する Office 用アプリの作成が必要とされる場合や、Office 用アプリ モデルでサポートされないシナリオに使用 ▪ あらゆる用途に対応 ▪ アプリケーションをオフラインで実行する、高度なシナリオに適する
Office VBA	**Visual Basic for Applications (VBA)** ▪ 自動処理や反復処理のほか、ユーザーによる対話式操作の拡張に使用 ▪ Office の自動処理およびシンプルなアプリケーション シナリオをサポート ▪ Office 用アプリと Office VBA によるハイブリッド アプローチでは、文書テンプレートに VBA を使用し、ドキュメントのコンテンツまたはカスタム XML パーツを介して Office 用アプリと情報をやり取りする

表 5-4

参考資料

Office 用アプリの概要	http://msdn.microsoft.com/ja-jp/library/office/apps/jj220082(v=office.15)
VBA (Office 開発者向け)	http://msdn.microsoft.com/ja-jp/office/ff688774

Office 用アプリ開発時のツールおよびテクノロジの選択肢

テクノロジ	選択すべき状況と理由
“Napa”	**"Napa" (ブラウザーでコードを作成できる、軽量な IDE)** ▪ Office 用/SharePoint 用アプリの開発を最も簡単に開始できる方法 ▪ 開発用マシンへの Visual Studio のインストールが不要。Office 365 開発者向けサイトで無償アプリケーションとして提供
Office Developer Tools for Visual Studio	**Office Developer Tools for Visual Studio (フル機能版)** ▪ 完全な機能と柔軟性が求められる、専門性の高い開発に使用 ▪ Office Developer Tools for Visual Studio は、Visual Studio Professional、Premium、または Ultimate の購入者に無償で提供

表 5-5

Office 用ビジネス アプリケーションは、ビジネス データの視覚化機能や分析機能を備えている場合が少なくありません。データ アクセスには、各種のサービスを利用するパターンが一般的です。こうしたサービスは JavaScript 経由で利用されるため、最も相性がよいのは ASP.NET Web API によるサービス (REST、JSON、OData など) です。シンプルなデータ駆動型サービスには、LightSwitch OData サービスまたは WCF Data Services を使用すると手間がかかりません。ただし、Office 用アプリは Web バックエンドを使用できるため、JavaScript 以外を選択することもできます。サーバー側から ASP.NET を使用しても、ビジネス データへのアクセスが可能です。

サービスを構築する際にテクノロジを使い分ける基準は、既に説明した内容とほぼ同じです。ASP.NET Web API や WCF などのテクノロジを使用し、Office 用アプリから利用するカスタム サービスを開発する場合に関する詳細については、「次世代型のアプリケーション パターン」の「サービス」セクションを参照してください。

シナリオ: Office 用アプリの連携

図 5-20 に、リモート サービスを利用してデータ ソースにアクセスする Office 用アプリの推奨連携パターンを示します。

SharePoint サービスの使用は任意です。Office 用アプリは、SharePoint から独立させて使用することもできます。

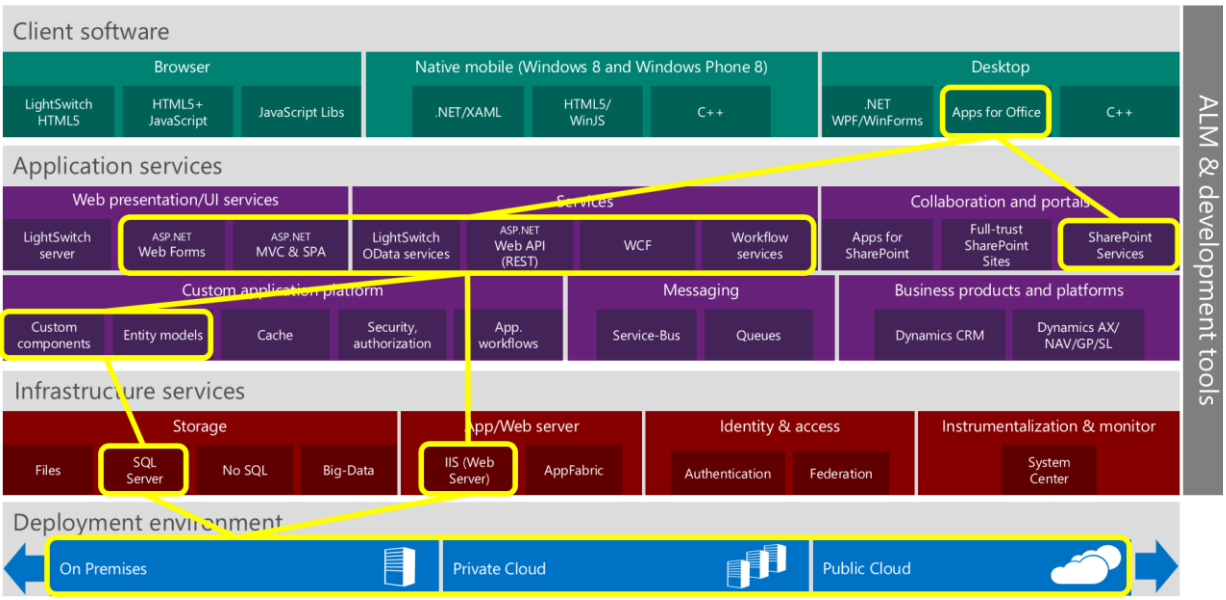

図 5-20

SharePoint 用アプリ (Apps for SharePoint)

SharePoint 用アプリ (Apps for SharePoint) は、既に説明した Office および SharePoint のクラウド アプリケーション モデルに基づいています。

SharePoint 用アプリケーション

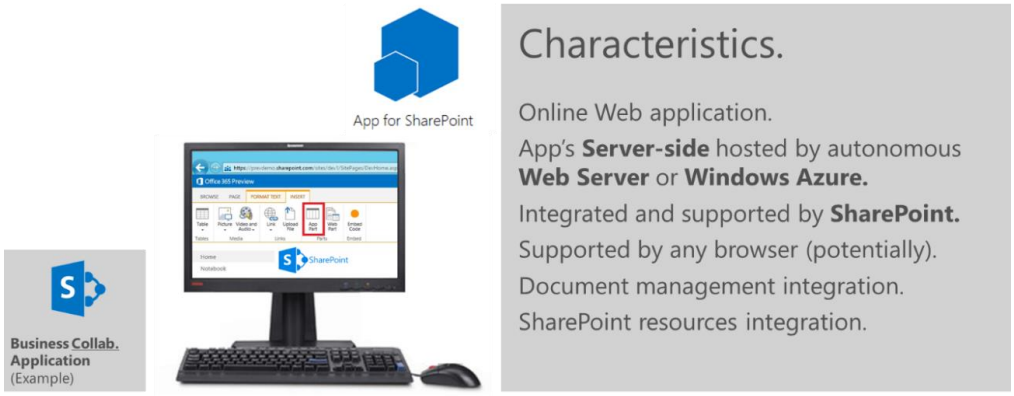

図 5-21

SharePoint 用アプリは、HTML Web ページである場合と、複数の HTML ページから成る (ASP.NET MVC などのいずれかのサーバー エンジン、または PHP、NodeJS などのサード パーティのテクノロジに基づいた) より複雑な Web アプリケーションである場合があります。SharePoint 用アプリは通常の Web アプリケーションなので、HTML ページでは任意の Web 標準 (HTML5/JS) テクノロジおよび jQuery のようなライブラリや、最新の SPA アプローチを使用できます。

また、リストや SharePoint Business Connectivity Services (BCS) モデルのような SharePoint リソースにアクセスして公開できるほか、SharePoint ワークフローの実装にも対応しています。

こうしたアプリケーションでは、アプリケーション マニフェストによる SharePoint への登録が必要になります。**アプリケーション マニフェストは、アプリケーションの基本プロパティを宣言した XML ファイルです**。また、アプリケーションを実行する場所と、起動時に行う処理も定義されます。このモデルは、Office 用アプリで使用されるモデルとまったく同じです。

SharePoint 用アプリを Web ブラウザーに表示する場合は、イマーシブなフル Web ページや、アプリケーション パーツ (SharePoint ページ内の IFrame を使用) として表示するほか、SharePoint UI のカスタム アクションを拡張して表示する方法があります。

SharePoint 用アプリケーションの構造

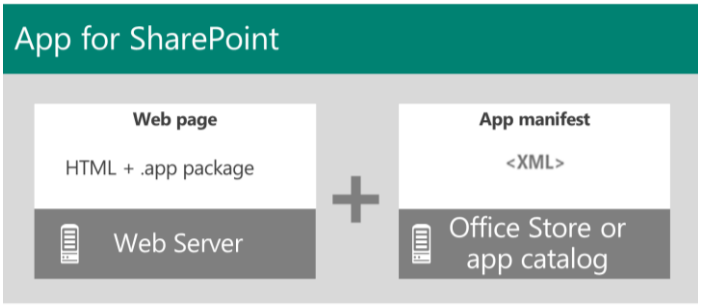

図 5-22

図 5-23

図 5-23 は、SharePoint 用アプリのさまざま実装オプションを示しています。SharePoint 用アプリを全画面表示の Web ページとして表示する場合、SharePoint とはまったく関係がなくなるように思えるかもしれませんが、クロム コントロール (Chrome Control) を使用することで、SharePoint と同じ外観を再現できます。また、OAuth を通じて、ユーザーのセキュリティ コンテキストを SharePoint とカスタム アプリケーションの間で共有することも可能です。アプリケーション内では、SharePoint 2013 によってサーバー側のコードが SharePoint サーバーから切り離されます。そのため、オンプレミス環境とクラウド環境のいずれの場合も、自社の専用 Web サーバー内でサーバー側コードを実行できます。

SharePoint サーバーにアプリケーションを展開する場合、アプリケーションにはクライアント側コード (HTML および JavaScript) のみで記述し、サーバー側コードは使用できません。一方、自動ホスト モデルおよびプロバイダー ホスト モデルを使用する場合は、サーバー側コードを SharePoint サーバーから切り離し、自社の専用サーバー/サービス内で実行できます。

SharePoint におけるアプリケーションのホスティング オプション

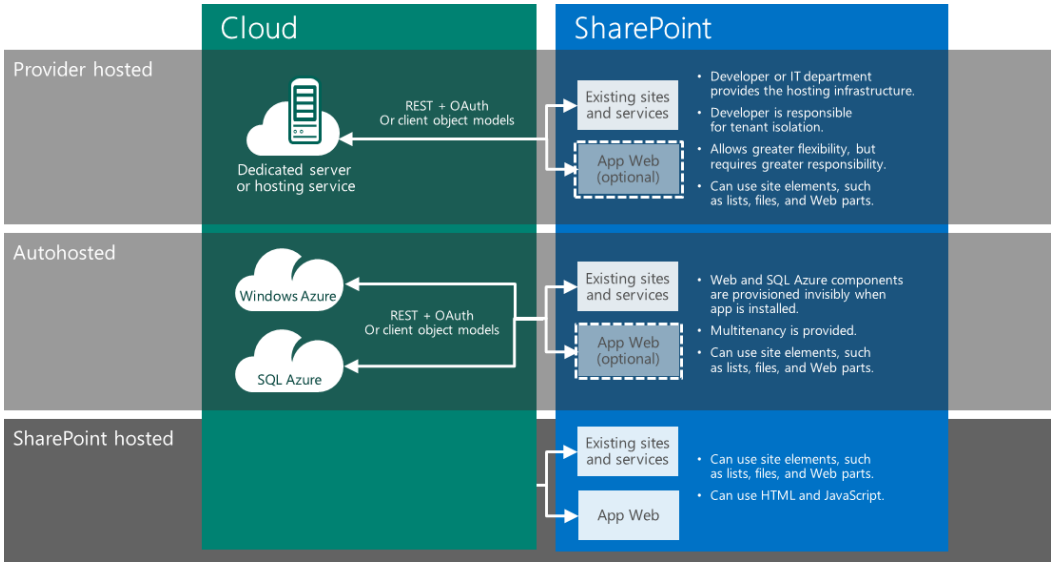

図 5-24

自動ホスト モデルとプロバイダー ホスト モデルは似通っていますが、前者の場合は、Windows Azure 上でのサーバー アプリケーションのプロビジョニングおよび展開が、透過的かつ自動的に実行されます。

以降の表では、SharePoint に関連したアプリケーション開発の際、各アプリケーションの優先事項および具体的なコンテキストに応じ、どのテクノロジを使用するべきかを示しています。このシナリオでは、新しい SharePoint 用アプリ モデルに加え、他のアプローチも紹介します。

SharePoint 開発時の選択肢

テクノロジ	選択すべき状況と理由
SharePoint 用アプリ モデル	**SharePoint 用アプリ (フル ページ アプリケーション、アプリケーション パーツ、および UI のカスタム アクション)** ▪ SharePoint と統合される、自律的なアプリケーションの既定のアプローチ ▪ SharePoint サーバーから切り離されたアプローチ
SharePoint サイト	**SharePoint サイトのカスタマイズ** ▪ SharePoint サイト自体のカスタマイズに使用 ▪ デザイン マネージャー機能および Visual Studio を使用してテンプレートとページをカスタマイズ
SharePoint の完全信頼ファーム ソリューション	**完全信頼での開発、管理上の拡張、従来型の Web パーツなど** ▪ SharePoint 用アプリで対応できない場合はファーム ソリューションを作成 ▪ 主な用途は、管理に関連した SharePoint のカスタム拡張や、SharePoint 内の内部アセットの開発 ▪ SharePoint 2013 以前のバージョンの SharePoint で必要とされるアプローチ
SharePoint のサンドボックス ファーム ソリューション (廃止)	**サンドボックス開発** ▪ SharePoint 2013 でサポートはされるが、削除された機能であるため、新たなサンドボックス ソリューションの構築は非推奨 ▪ SharePoint 2013 以前のバージョンの既存の開発物にのみ使用

表 5-6

参考資料	
SharePoint 2013: ファーム ソリューション、サンドボックス、アプリケーションの選択基準	http://social.technet.microsoft.com/wiki/contents/articles/13373.sharepoint-2013-what-to-do-farm-solution-vs-sandbox-vs-app.aspx (英語)
SharePoint アプリケーションの概要	http://msdn.microsoft.com/ja-jp/library/fp179930.aspx
新しい SharePoint	http://sharepoint.microsoft.com/blog/Pages/BlogPost.aspx?pID=1012 (英語)

SharePoint 用アプリ開発時のツールおよびテクノロジの選択肢

テクノロジ	選択すべき状況と理由
“Napa”	**"Napa" (ブラウザーでコードを作成できる、軽量な IDE)** ▪ Office 用/SharePoint 用アプリの開発を最も簡単に開始できる方法 ▪ 開発用マシンへの Visual Studio のインストールが不要
Office Developer Tools for Visual Studio	**Office Developer Tools for Visual Studio (フル機能版)** ▪ 完全な機能と柔軟性が求められる、専門性の高い開発に使用 ▪ Office Developer Tools for Visual Studio は、Visual Studio Professional、Premium、または Ultimate の購入者に無償で提供
LightSwitch	**LightSwitch で強化された SharePoint 用アプリ** ▪ SharePoint のリソース (リストおよびサービス) または SharePoint 以外のサービスとデータ ソースを利用する、データ駆動型 (CRUD) アプリケーションを作成する場合に使用 ▪ SharePoint と統合される、柔軟なフォームおよびデータ中心アプリケーションを作成する最も簡単な手段

表 5-7

SharePoint 用ビジネス アプリケーションはほとんどの場合、ビジネス データや SharePoint データなど、いずれかの種類のデータにアクセスする必要があります。こうしたデータ アクセスは、各種のサービスまたはローカル データ ソースを通じて提供されます。

SharePoint 用アプリのバックエンド サービス テクノロジ

テクノロジ	選択すべき状況と理由
SharePoint クライアントの .NET オブジェクト モデルまたは REST/OData エンドポイント	**SharePoint クライアントの .NET オブジェクト モデル (OM) または REST/OData エンドポイント** ▪ SharePoint のリスト、コンテンツ タイプ、リスト アイテム、ドキュメント ライブラリ、および SharePoint リソースに関連したいずれかの種類の操作を利用する場合に使用 ▪ SharePoint 用アプリを開発する際、SharePoint Server のマネージ OM クラス ライブラリは使用できないので注意 (SharePoint 用アプリは別のリモート サーバーで実行されるため)
SharePoint の JavaScript クライアント オブジェクト モデル	**SharePoint の REST/OData エンドポイントまたはクライアントの .NET マネージ API** ▪ アプリケーション内の JavaScript から、直接 SharePoint リソースを利用する場合に使用
カスタム ASP.NET Web API	**REST アプローチとリソース指向** ▪ サービスの利用者が、ネイティブ アプリケーションや Web クライアントを必要としている場合、またはインターネット経由の不特定のコンシューマーが対象となる場合に使用する、柔軟性に優れたアプローチ ▪ ASP.NET Web API は、ブラウザーやモバイル デバイスを含む幅広いクライアントから利用可能な、HTTP サービスを簡単に構築できるフレームワーク

	▪ 開発にすぐに取り掛かれる軽量なフレームワークであり、他のプラットフォームおよびフォーマット (JSON、OData など) との高度な相互運用性を提供 ▪ REST サービスに特化した設計。アプリケーションのドライバーとして HTTP のメソッドを使用し、リソース指向性が高い ▪ HTTP のメソッドに基づくインターネット キャッシュ (Akamai、Windows Azure CDN、Level3 など) を活用し、優れた拡張性を提供
LightSwitch OData REST サービス	**REST アプローチとリソース指向に基づく、最も簡単なサービス構築手段** ▪ シンプルなリソース指向 OData サービスを作成する場合の最も簡単な手段
カスタム WCF Data Services	**WCF Data Services** ▪ データ/リソース指向性が高く、主に CRUD およびデータ駆動型のサービスに使用 ▪ OData のみをサポート。使用方法は容易だが、柔軟性と制御性は ASP.NET Web API に劣る ▪ ASP.NET Web API と同じ OData コア ライブラリを使用

表 5-8

参考資料	
SharePoint 2013 での適切な API セットの選択	http://msdn.microsoft.com/ja-jp/library/jj164060(v=office.15).aspx

シナリオ: SharePoint 用アプリの連携

図 5-25 に、SharePoint 用アプリを作成する場合の、テクノロジ連携パターンの典型的な例を示します。

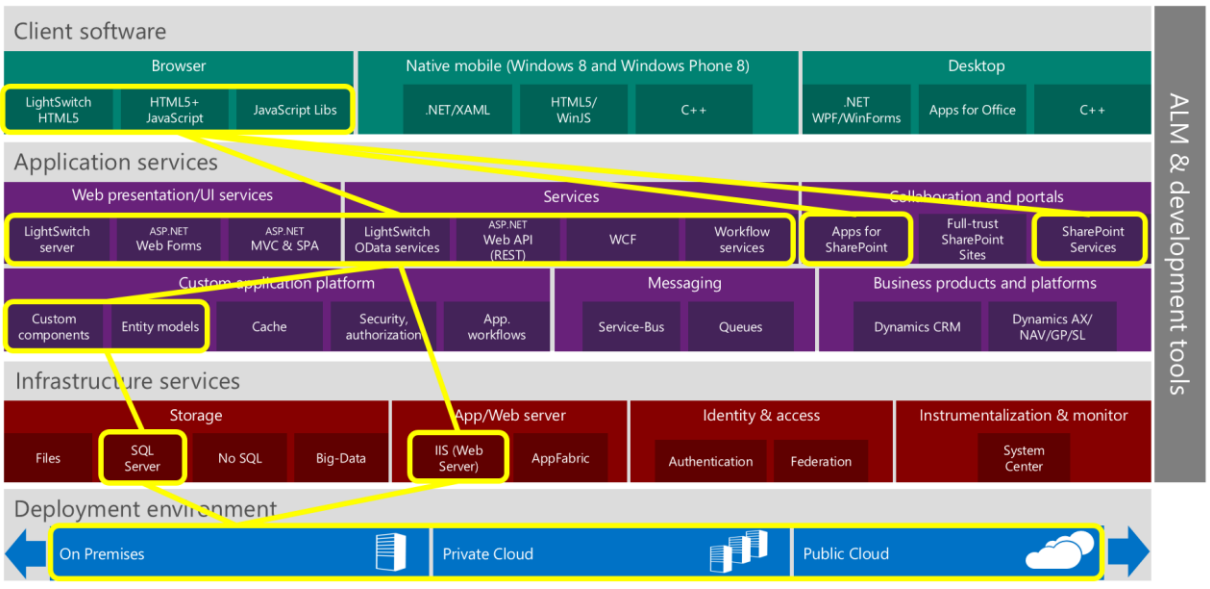

図 5-25

このシナリオでは、前述の Web アプリケーション開発テクノロジに関する推奨事項がほぼそのまま当てはまります。ただし、サービスを構築するテクノロジに関して 1 つ強調しておきたいのが、SharePoint では OData サービスのサポートが強く推進されているという点です。そのため、SharePoint 用カスタム アプリケーションでもそのパスを継承し、(ASP.NET Web API、WCF Data Services、LightSwitch OData サービスなど、OData をサポートする任意のテクノロジに基づいた) OData サービスの作成および利用をお勧めします。

ミッション クリティカルな大規模ビジネス アプリケーション

ミッション クリティカルなビジネス アプリケーションは、主力業務を適切に運営するうえで、欠くことのできない任意のビジネス アプリケーションだと定義できます。こうした性質を持つアプリケーションに、たとえ短時間でも障害が発生すれば、事業の失敗につながりかねません。

ミッション クリティカルなコア ビジネス アプリケーションには、新たな目標と固有の優先事項があります。絶えず進化する大規模アプリケーションには、長期的な持続可能性と保守容易性が不可欠です。

ミッション クリティカルな大規模アプリケーションの開発では、特に 2 つの点に注意する必要があります。

- **ビジネス アプリケーションのコア ドメインにおける複雑性を解消する**
  - ドメインに関する複雑な専門知識およびビジネス ルールのほか、これらをソフトウェアに効果的に反映するための方法と関連があります。
  - この点に配慮しながら、コンテキスト (ドメイン駆動設計アーキテクチャ アプローチ、依存性注入の手法およびコンテナーに基づく疎結合アーキテクチャなど) に応じたアーキテクチャ、設計、実装方法 (モノリシック アプローチにするか、構造化/複数層アプローチにするかなど) を決定し、最適なパターンとベスト プラクティスを判断する必要があります。また、アプリケーション設計では、アプリケーションの将来的な発展および保守しやすさを考慮する必要があります。
- **ミッション クリティカルな大規模アプリケーションの十分な QoS (サービス品質) を確保する**
  - 可用性、拡張性、セキュリティなどの問題と関連があります。
  - この点に配慮しながら、分野横断的な対策 (セキュリティ、運用、インストルメンテーションなど) の設計方法および実装方法を検討する必要があります。
  - また、インフラストラクチャ アーキテクチャも QoS と緊密にリンクさせます。たとえば、必要となる拡張性と弾力性、ユーザー タイプ、実稼働までの機動性のニーズに応じて、オンプレミス環境またはクラウド環境のインフラストラクチャ アーキテクチャを検討する必要があります。

図 5-26 に、こうした種類のアプリケーションで重視される事柄、選択基準、および対応方法を示します。

図 5-26

.NET によるミッション クリティカルな大規模コア ビジネス アプリケーション

.NET は何年もの時間をかけ、堅牢性、柔軟性、および拡張性を重視した、アプリケーション開発フレームワーク セットとして成長してきました。.NET はミッション クリティカルな大規模アプリケーションの構築に関し、マイクロソフトの多くのお客様から信頼を得ているほか、次に示す Forrester レポート (2012 年) の抜粋からわかるように、アナリストからも高く評価されています。

AD & D のリーダーたちは、大規模で複雑なアプリケーションに関し、.NET は Java に匹敵すると考えています。 Windows Server、SQL Server、および .NET Framework は、拡張が困難とされてきましたが、それは昔の話です。 Windows Server、SQL Server、.NET は、最大級のアプリケーションをサポートする機能を備え、企業の AD & D のリーダーたちから厚い信頼を得ています。 リーダーたちは、パフォーマンスと拡張性の障壁となるのは、アプリケーションのコードの不完全さであり、プラットフォーム ソフトウェア自体に問題はないと話しています。

「.NET は拡張性に優れています。 かなりの数のユーザーに対するロード テストを実施しましたが、 .NET ではキャッシングとセッション管理の適切なオプションが提供され、例外処理とデータ アクセスにも遜色はありませんでした。 類似した機能は Java の Spring にもありますが、.NET の方が統合性に優れています」 (金融サービス企業のアプリケーション開発担当 VP)

図 5-27

ミッション クリティカルな大規模コア ビジネス アプリケーションのテクノロジの選択

このガイドはカスタム開発の説明のみを目的としているため、ミッション クリティカルな市販製品 (SAP、Dynamics ERP などの具体的な製品) 自体については触れません。

> **データ駆動型の開発テクノロジに関するメモ:** ミッション クリティカルな大規模アプリケーションは、多くのサブシステムを含んでいることが少なくありません。こうしたサブシステムは、付帯的かつコア ビジネスに関係のない、データ駆動型のシンプルな領域の場合もあります。比較的シンプルなデータ中心のサブシステムには、「小規模/中規模ビジネス アプリケーション」セクションで説明した、データ駆動型テクノロジを使用できます。ここでは、こうしたアプローチとテクノロジに触れる代わりに、システムのコア ドメインを実装する、境界づけられたコンテキストのテクノロジについて説明します。

ミッション クリティカルな大規模アプリケーションには、幅広いテクノロジが使用されます。各種のプレゼンテーション層テクノロジに始まり、最終的には O/RM がベースとなることの多いドメイン モデルの実装、さらにはキャッシュ、ワークフロー、サービス バス、メッセージ キュー、さまざまな種類のデータ ソース (リレーショナルまたは NoSQL) を使用するためのテクノロジなど、その種類は多岐にわたります。この中からどのテクノロジを選択するかは、概して、具体的なシナリオとアプリケーションの優先事項によって決まります。

シナリオ: 大規模コア ビジネス アプリケーション
(多数のサブシステムを含む)

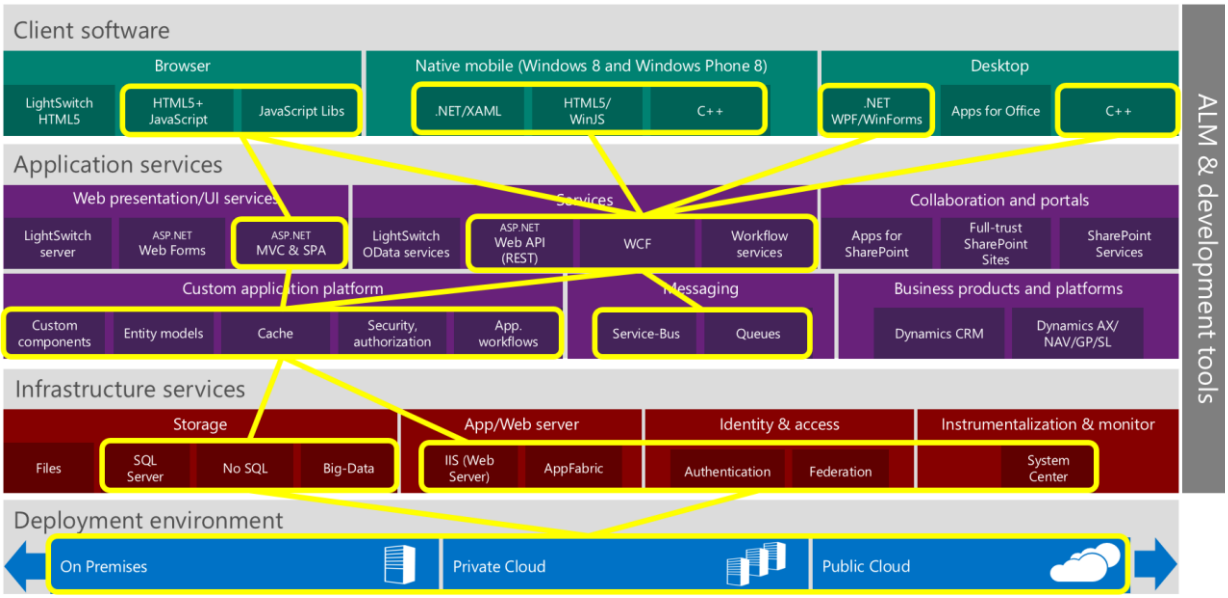

図 5-28

次の表では、開発するミッション クリティカルなアプリケーションの優先事項および具体的なコンテキストに応じ、ここに挙げたどのテクノロジをいつ使用するべきかを示します。

ミッション クリティカルなコア ビジネス サブシステムの Web UI テクノロジ

テクノロジ	選択すべき状況と理由
HTML5 をサポートする ASP.NET **MVC**	**HTML5 をサポートする ASP.NET MVC** ▪ **HTML のレンダリングを完全に制御**でき、優れた**単体テスト**機能と**フェイク**機能を備えた、最も柔軟かつ強力なオプションが必要な場合に**使用** ▪ **Modernizr** などのライブラリを使用し、HTML5 の機能サポートの有無と回避策を検出 ▪ **CSS3** を利用してスクリプトを短くし、保守しやすいコードを作成 ▪ クライアント側のプログラミングには **JavaScript** と **jQuery** を使用 ▪ MVC は**検索エンジンの最適化**と **RESTful URL** を標準サポート
ASP.NET **Single Page Application (SPA)**	**ASP.NET MVC、HTML5/JavaScript、Knockout、**および **Breeze** に基づく **Single Page Application (SPA)** ▪ SPA はフレームワークではなく、ASP.NET MVC を使用して実装できる設計パターン。Knockout (JavaScript 内で MVVM パターンをサポート) や Breeze (高度なデータ管理が可能) などの JavaScript ライブラリを多用 ▪ 対話式操作の多い Web アプリケーションやスムーズな UI の更新 (サーバー ページの再読み込みおよび可視性の切り替えに伴うラウンド トリップを最小限にする場合) のほか、クライアント側で HTML5、CSS3、および JavaScript による重要な対話式操作を行う場合にも使用 ▪ クライアントとサーバー間の双方向通信を実現するには、オープンソースの ASP.NET SignalR JavaScript クライアント ライブラリを活用 ▪ JavaScript から ASP.NET Web API サービスにアクセス

表 5-9

ミッション クリティカルなビジネス アプリケーションに関する Web 開発を行う際は、疎結合アーキテクチャをサポートし、最適な Web ユーザー エクスペリエンスを作成できるよう、完全な制御性ときめ細かい機能を備えたテクノロジを提供しなければなりません。そのためマイクロソフトでは、ASP.NET MVC (サーバーでの完全な制御性を実現) および ASP.NET SPA (Single Page Application) テンプレートを提供し、Knockout のような JavaScript ライブラリを多用して、JavaScript 内で MVVM パターンをサポートできるようにしています。また、高度なデータ管理に対応した Breeze ライブラリのほか、Durandal や RequireJS といった、便利な SPA JavaScript フレームワークやライブラリもあります。

参考資料	
Project Silk: (モダン ブラウザーのためのクライアント側 Web 開発)	http://silk.codeplex.com/ (英語) http://msdn.microsoft.com/en-us/library/hh396380.aspx (英語) http://www.amazon.com/Project-Silk-Client-Side-Development-Microsoft/dp/1621140105/ (英語)
ASP.NET MVC 4 モバイル機能のチュートリアル	http://www.asp.net/mvc/ (英語)
ASP.NET SPA の概要	http://www.asp.net/single-page-application (英語)
Breeze/Knockout テンプレートの概要	http://www.asp.net/single-page-application/overview/templates/breezeknockout-template (英語)
Durandal SPA フレームワーク	http://durandaljs.com/ (英語)
RequireJS ライブラリ	http://requirejs.org/ (英語)
Modernizr (HTML5/CSS3 の機能検出ライブラリ)	http://www.modernizr.com (英語)

ミッション クリティカルなコア ビジネス サブシステムのデスクトップ UI テクノロジ

テクノロジ	選択すべき状況と理由
.NET WPF	**.NET Windows Presentation Foundation (WPF)** ▪ 複雑な UI や、スタイルのカスタマイズを必要とし、グラフィックスを多用するようなデスクトップ シナリオで推奨される、Windows ベース デスクトップ アプリケーション向けのテクノロジ。WPF では XAML のビューも活用。WPF の開発スキルは Windows ストア アプリの開発スキルと似通っているため、Windows ストア アプリへの移行は、Windows フォームからよりも WPF からの方が容易 ▪ .NET 4.5 の新しいシンプルな非同期機能 (async/await) を活用 ▪ SignalR .NET クライアントを活用し、クライアントとサーバー間の双方向通信を実現
.NET Windows フォーム	**.NET Windows フォーム** ▪ デスクトップ アプリケーションの構築のため、.NET Framework で初めて提供された UI テクノロジであり、現在も多くのビジネス デスクトップ アプリケーションに活用可能。使いやすく、WPF よりも軽量なため、UI スタイルのカスタマイズを必要としないシンプルなシナリオに適する ▪ WPF よりもシンプルな UI 機能を提供し、XAML に基づいていないため、Windows ストア アプリへの移行には制約がある ▪ WPF と同様に、async/await を使用した非同期プログラミングなど、最新の .NET 機能の利用が可能
C++ (Win32 または MFC)	**C++ (Win32 または MFC)** ▪ UI のパフォーマンスに優れたアプリケーションや、きわめて大規模で複雑な、長期的に使用する製品 (Microsoft Office に相当するような製品) を構築する場合に使用 ▪ DirectX ベースのグラフィックスを多用するシナリオで、最良かつ最大限のパフォーマンスを実現する場合に使用

表 5-10

参考資料	
WPF の概要	http://msdn.microsoft.com/ja-jp/library/aa970268.aspx (機械翻訳)
Windows Presentation Foundation	http://msdn.microsoft.com/ja-jp/library/ms754130.aspx (機械翻訳)
Prism for WPF	http://msdn.microsoft.com/en-us/library/gg406140.aspx (英語)
Windows フォーム	http://msdn.microsoft.com/ja-jp/library/dd30h2yb.aspx (機械翻訳)
Win32 デスクトップ アプリケーション (Visual C++)	http://msdn.microsoft.com/ja-jp/library/vstudio/hh875053.aspx
MFC デスクトップ アプリケーション	http://msdn.microsoft.com/ja-jp/library/vstudio/d06h2x6e.aspx

デスクトップ アプリケーションは、大規模ビジネス アプリケーションの一部として、特に既存システム内の大量のデータ入力を行うシステムに多く使用されます。こうしたアプリケーションには、Windows ストア ビジネス アプリケーションへの移行パスを提供する WPF (Windows Presentation Foundation) のほか、UI スタイルのカスタマイズを必要としないシンプルなシナリオの場合には、WPF に比べ使いやすく軽量なソリューションを提供する Windows フォームの使用をお勧めします。

Windows ストア アプリのモダン UI の開発テクノロジ

(Windows ストア アプリおよび Windows Phone アプリケーション/サブシステム)

タッチ指向のモダン アプリケーション (Windows 8 または Windows Phone 向けの Windows ストア アプリ) には革新的かつ魅力的なユーザー エクスペリエンスが求められますが、企業ではさらに、そうしたユーザー エクスペリエンスを担当者の得意分野とスキルに沿って実現する必要があります。マイクロソフトはネイティブの Windows ストア アプリを作成するための広範なテクノロジおよび言語を提供しており、.NET/XAML、WinJS/HTML5、C++/XAML といった、開発チームの各種のスキルがサポートされます。

各テクノロジを使い分ける基準については、ここでは再度紹介しないこととしますので、「次世代型のアプリケーション パターン」セクションで紹介した推奨事項を参照してください。

ミッション クリティカルなコア ビジネス サブシステムの中間層テクノロジ

テクノロジ	選択すべき状況と理由
サービス	**サービス指向** 分散型エンタープライズ アプリケーションには継続的なサービスが不可欠であり、パフォーマンスや、軽量で相互運用可能な HTTP サービス、各種の標準 (REST、OData、SOAP、WS-*)、およびエンタープライズ サービスの各要件のサポートが優先事項となる。マイクロソフトでは、**ASP.NET Web API**、**WCF**、および **ASP.NET SignalR** に基づく広範なテクノロジを提供
ドメイン モデル	**ドメイン エンティティ モデル、集計、ドメイン ロジック** ドメイン モデルはアプリケーションの中核であり、大規模なドメインの複雑性解消、長期的な保守に対応した適切な設計の作成のほか、ドメイン コード (POCO エンティティ) とインフラストラクチャ テクノロジの分離が優先事項となる。マイクロソフトでは、ドメイン駆動設計パターンを適用する際に使用可能な、成熟したドメイン モデル アプローチおよび疎結合データ アクセス テクノロジ (Entity Framework、LINQ、ADO.NET など) を提供
複合および疎結合アーキテクチャ	**疎結合アーキテクチャ、複合、統合、ビジネス プロセス、ワークフロー** すべてのエンタープライズ アプリケーションには、たとえば、依存性の注入および IoC コンテナーに基づいた疎結合アーキテクチャ設計、時間のかかるプロセスおよびワークフロー、イベント駆動型サブシステムの統合、最新のクレームベース セキュリティの分離、トランザクションの実装などといった、大規模アプ

	リケーションの典型的なシナリオに対処するバックボーンが必要。マイクロソフトはこうした領域に対応する、制御の反転 (IoC) コンテナー (**Unity** および **MEF**)、Windows Workflow Foundation (**WF**)、Windows Identity Foundation (**WIF**)、NET Framework **System.Transactions** API、Reactive Extensions (**Rx**) といった各種の実績あるテクノロジを提供

表 5-11

参考資料	
ASP.NET Web API	http://www.asp.net/web-api (英語) http://msdn.microsoft.com/ja-jp/library/hh833994(v=vs.108).aspx
Windows Communication Foundation とは	http://msdn.microsoft.com/ja-jp/library/ms731082.aspx
ワークフロー サービス	http://msdn.microsoft.com/ja-jp/library/dd456788.aspx
ASP.NET SignalR	http://signalr.net/ (英語)
Unity	http://unity.codeplex.com/ (英語)
MEF	http://msdn.microsoft.com/en-us/library/dd460648.aspx (英語)
System.Transactions	http://msdn.microsoft.com/ja-jp/library/system.transactions.aspx (機械翻訳)
Reactive Extensions (Rx)	http://msdn.microsoft.com/en-us/data/gg577609.aspx (英語)

ミッション クリティカルなコア ビジネス サブシステムのインフラストラクチャ テクノロジ

テクノロジ	選択すべき状況と理由
メッセージング	**非同期メッセージングの調整** API による複合、統合、およびイベント駆動型のアプローチ (「ミッション クリティカルなコア ビジネス サブシステムの中間層テクノロジ」で前述) では、拡張に備えた強固な基盤インフラストラクチャが求められ、非同期メッセージング インフラストラクチャおよびメッセージ キュー インフラストラクチャに関連したアセットが必要。マイクロソフトは、**Windows Azure サービス バス**、**Windows Server サービス バス**、**Windows Azure キュー**、**メッセージ キュー (MSMQ)**、**Biztalk Server/サービス**といった、成熟した強固なインフラストラクチャ テクノロジを提供
セキュリティ	**ID、認証、および承認** マイクロソフトは、Active Directory (AD)、AD フェデレーション サービス (AD FS)、Windows Azure AD といった強固なセキュリティ インフラストラクチャ テクノロジを提供。モダン アプリケーションのセキュリティ関連の開発は、クレームベースのセキュリティおよび Windows Identity Foundation をベースとすることを推奨
キャッシュ	**アプリケーション内のデータ キャッシュ機能と、インターネットを使用した HTTP キャッシュ機能** Windows Server AppFabric キャッシュおよび Windows Azure キャッシュはインメモリの分散キャッシュであり、仮想マシン/ロールの外部の PaaS サービスとして使用するほか、展開したキャッシュを自社の Windows Server 間で共有することが可能 **Windows Azure CDN** は、HTTP 経由でアクセスするデータ (ビデオ、ファイルなど) をキャッシングできる "インターネット キャッシュ"
データ ソース	**リレーショナル SQL データベース、NoSQL 非構造化データベース、およびビッグ データ** ほぼすべての種類のビジネス アプリケーションは、ビジネス データを保持するためのデータ ソースを必要とするが、アプリケーションのコンテキストおよび優先事項に応じ、データ ソースを使い分けるのが適

	切。優先事項は、リレーショナル データの豊富さとトランザクション機能、データの可用性と拡張性、および膨大な非構造化データに対するアプローチが中心となる。マイクロソフトはこうした優先事項に対応するため、**SQL Server**、**Windows Azure SQL データベース**、**Windows Azure NoSQL テーブル**および **BLOB** のほか、Windows Server または Windows Azure 上での **HDInsight (ビッグ データ、Hadoop)** といった各種のテクノロジを提供
展開インフラストラクチャ	**オンプレミス、クラウド、およびハイブリッド クラウド** すべてのエンタープライズ アプリケーションは、将来的にコンテキストがどう変化するかにかかわらず、アプリケーションの可用性が確保されるような信頼性の高いインフラストラクチャを必要とする。優先事項として、オンプレミスまたはクラウドの選択に対する一貫性ある代替手段、信頼性、パフォーマンス、可用性、拡張性、弾力性、ハイブリッド IT、運用、および監視などが挙げられる。マイクロソフトはこうした要件に対し、**Windows Server**、**Windows Azure**、および **Microsoft System Center** が提供するインフラストラクチャを使用した、業界屈指のソリューションを提供 Windows Azure およびハイブリッド クラウドの詳細については、「次世代型のアプリケーション パターン」セクションを参照

表 5-12

複雑かつ大規模で、ミッション クリティカルなアプリケーションの開発に関しては、テクノロジのリストを示すだけでは十分ではありません。ターゲットとする特定のドメインおよびサービス品質要件に伴う複雑性を解消するためには、アーキテクチャおよび設計に関するアプローチについて、包括的に解説する必要があります。以降のセクションでは、ドメイン駆動設計、CQRS、イベント駆動型統合、レガシ コア ビジネス アプリケーションの刷新といった、コア ビジネス アプリケーションやミッション クリティカルなアプリケーションを開発する際に考慮すべき重要なアプローチを紹介します。

長期用コア ビジネス アプリケーションのアプローチと傾向

ミッション クリティカルな大規模コア ビジネス アプリケーションを構築するうえで一般的に考慮されるアプローチ、原則、およびパターンを挙げます (優先事項によっては、より小規模なアプリケーションも該当する場合があります)。

- **エンタープライズ アプリケーション アーキテクチャのパターン** (Martin Fowler 氏が提唱するパターンなど)
- **データ駆動設計 (DDD)** (Eric Evans 氏、Jimmy Nilsson 氏、Vaughn Vernon 氏の著書を参照)
- **CQRS (コマンド クエリ責務分離)** (Microsoft patterns & practices の「CQRS Journey」のほか、Greg Young 氏、Udi Dahan 氏などが提供する情報を参照)
- **イベント ソーシング** (Greg Young 氏の著書などを参照)
- **依存関係逆転の原則に基づく分離アーキテクチャ**
  - 依存性の注入、および制御の反転 (IoC) コンテナーの使用
    - Microsoft patterns & practices、Unity、Ninject、Castle Windsor、Spring.NET など
- **サービス指向** — REST サービス、WS-*、サービス バス、弾力性に優れたクラウド サービス
- **S.O.L.I.D. 原則** — http://en.wikipedia.org/wiki/SOLID_(object-oriented_design) (英語)
- **ビヘイビアー駆動設計 (BDD) および TDD** — (Dan North 氏、Chris Matts 氏などが提唱)
  BDD フレームワーク: SpecFlow、NSpec、Cuke4Nuke、NBehave、MSpec など

疎結合アーキテクチャと依存関係逆転の原則

疎結合アーキテクチャは、依存関係逆転の原則、手法、およびテクノロジや、S.O.L.I.D. オブジェクト指向設計と共に、ミッション クリティカルなコア ビジネス アプリケーションの多くのシナリオに適用されます。注目したいポイントは、こうしたアプローチが**小規模/中規模ビジネス アプリケーション**の開発にも役に立ち、将来的にアプリケーションの複雑性やボリュームの増大が見込まれる場合は特に有効だということです。ただし、**大規模コア ドメイン アプリケーションの開発では、こうしたアプローチが絶対不可欠**となります。

アプリケーションを構成するコンポーネントを明確に区分された構造化レイヤーに配置することで、"関心の分離" を始めとした各種原則を順守し、保守の効率化などのメリットがもたらされます。また、特にオブジェクトどうしの関係に注意を払うことが重要です。つまり、オブジェクトがどのように利用され、各オブジェクトがどういった状況で、他のオブジェクトによってインスタンス化されるかに注意する必要があります。

"疎結合オブジェクト" の主な手法は、依存関係逆転の原則 (S.O.L.I.D. 原則の 1 つ。http://en.wikipedia.org/wiki/SOLID_(object-oriented_design) を参照) に基づいています。この原則は、オブジェクト指向で使用される従来型の依存関係 (上位のレイヤーが下位のレイヤーに依存する) を逆転させるものです。このアプローチにより、上位のオブジェクトをインフラストラクチャの実装から独立させ、その結果、下位のテクノロジからも切り離すという重要な目的が達成されます。

依存関係逆転の原則では、次のことが定められています。

A. 上位のレイヤーは下位のレイヤーに依存してはならない。どちらのレイヤーも、抽象 *(コントラクト/インターフェイス)* に依存するべきである。

B. 抽象は実装の詳細に依存してはならない。実装の詳細が抽象 *(コントラクト/インターフェイス)* に依存するべきである。

この原則は、上位のオブジェクトを下位のオブジェクトから切り離すことを目的としています。これによって、上位クラスの依存関係を設定したインターフェイスの実装が置き換えられ、上位クラスのコードを一切変更することなく、上位クラスによって調整された最終的なアクションを拡張または変更できます。たとえば、上位アプリケーション オブジェクトのまったく同じコードを (コードを変更せずに) 実行することで、さまざまな下位インフラストラクチャ/テクノロジ オブジェクトを利用して、依存関係を定義した各種の共通インターフェイス (抽象) を実装する機能を実現します。これは、S.O.L.I.D. のリスコフの置換原則 (http://ja.wikipedia.org/wiki/%E3%83%AA%E3%82%B9%E3%82%B3%E3%83%95%E3%81%AE%E7%BD%AE%E6%8F%9B%E5%8E%9F%E5%89%87) にも関係しています。

コントラクト/インターフェイスは下位コンポーネントで必要となる動作を定義します。こうしたインターフェイス/コントラクトは、上位のレイヤーに配置するようにします (つまり、DDD の場合であれば、通常は抽象/インターフェイスのほとんどをドメイン モデル層で定義します。このため、インフラストラクチャ層がシステムと連携する際、必要となる仕様を規定したコントラクトはドメイン モデル層に所有されることになります)。対象となるインフラストラクチャ層は将来的に、S.O.L.I.D. の開放/閉鎖原則 (http://ja.wikipedia.org/wiki/%E9%96%8B%E6%94%BE/%E9%96%89%E9%8E%96%E5%8E%9F%E5%89%87) に従って置き換えられます。

下位のコンポーネントが (上位のレイヤーに配置された) インターフェイスを実装する場合、この下位のコンポーネント/レイヤーは上位のコンポーネントに依存していることになるため、従来型の依存関係が逆転します。これが、**"依存関係逆転の原則"** と呼ばれる理由です。

この目的で使用される手法およびパターンはいくつかあります。

- 制御の反転 *(IoC)* コンテナー
- 依存性の注入 *(DI)*

制御の反転 (IoC) コンテナー: クラスの依存関係向けに具体的実装の種類を選択する機能を、外部コンポーネントまたは外部ソース (いずれかのベンダーが提供する任意の IoC コンテナーの実体) に委任します。特定のオブジェクトが、自身が

依存している、または必要としている他のオブジェクトのインスタンスを要求できる、"プラグイン" タイプのアーキテクチャをサポートする手法です。

依存性の注入 (DI): IoC の特殊なケース。必要なオブジェクト/依存関係をクラス自身が作成する代わりに、オブジェクト/依存関係が外部からクラスに提供されます。上位オブジェクトのコンストラクターによって依存関係を挿入する方法が、最も一般的に使用されます。

IoC コンテナーおよび依存性の注入を使用すると、プロジェクトの柔軟性、包括性、および保守容易性が高まり、実際のコードに加える変更を最小限に抑えることができます。その代わりに、プロジェクト進行中には、新たな実装/クラスを定義する場合を除いて、できるだけコードを追加しないことが想定されています。

DI では、依存関係の実装のみを変更することで、クラスの内部コードを変更することなく、クラスの動作を変更できます。これは、S.O.L.I.D. の開放/閉鎖原則 (http://ja.wikipedia.org/wiki/%E9%96%8B%E6%94%BE/%E9%96%89%E9%8E%96%E5%8E%9F%E5%89%87) に関係しています。オブジェクト インスタンスの挿入手法には、"インターフェイス挿入"、"コンストラクター挿入"、"プロパティ セッター挿入"、および "メソッド呼び出し挿入" があります。

アーキテクチャの品質向上手段としての依存性注入

DI および IoC を使用すると、優れた設計手法を簡単に活用できるようになり、オブジェクト指向システムのアーキテクチャの品質が向上されます。また、クラスとその依存関係の疎結合を実現するほか、クラスがテストしやすくなり、汎用的な柔軟性メカニズムを提供します。また、クラスと構成メカニズムとの直接的な結び付きが最小限に抑えられ、コードを再利用できるチャンスが大幅に拡大します。

S.O.L.I.D. の**単一責任の原則**によれば、各オブジェクトは一意の責任を持つべきであるとされています (http://en.wikipedia.org/wiki/Single_responsibility_principle (英語) を参照)。この原則は、単一の責任とはコードを変更する単一の理由である (そして、コードの変更はバグを発生させる要因となる) とし、1 つのクラスが変更されるには、1 つの理由が必要である (複数個あってはならない) と結論付けています。この原則は、単一の責任のみを持つ小規模なクラスを設計および開発するよう推奨するもので、業界で広く受け入れられています。このことは依存関係の数、言い換えると各クラスが依存するオブジェクトの数に直結します。通常、1 つのクラスの責任が 1 つであれば、クラスに含まれるメソッドと他のオブジェクトとの依存関係は、実行時にほとんど発生しません。1 つのクラスが多くの依存関係 (たとえば 15 件の依存関係など) を持つ場合、このコードは深刻な問題の兆候が現れた、いわゆる "悪臭" のする状態だと言えます。なぜなら、コンストラクターで依存関係を挿入するときには、コンストラクター内でオブジェクトの依存関係をすべて宣言しなくてはならず、この場合には 1 つのみの責任を持つクラスのために 15 の依存関係を宣言することになってしまいます。このクラスが単一責任の原則従っていないことは明らかでしょう。そのため、DI は優れた設計および実装を実現するためのガイドとしての役割も果たし、さまざまな実装を明確に挿入するための分離アプローチを提供します。

加えて、依存性の注入および制御の反転コンテナーは、単体テストまたはフェイク/モックの利便性のみを目的として設計されたものではありません。それを目的だとするならば、インターフェイスの主な目標が、テスト実施を可能にすることになってしまいます。実際には、依存性を減らして柔軟性を高め、アプリケーションの保守を 1 か所から簡単に行えるようにすることが DI および IoC の目的です。テストも重要ですが、依存性の注入および IoC コンテナーを使用する第一の理由または最も重要な理由にはなりません。

DI/IoC コンテナー

DI と IoC は、さまざまなベンダーや提供元による、さまざまなテクノロジおよびフレームワークを通じて実装できます。次の表にフレームワークの一例を挙げます。どの実装方法を選択しても、その方法は非常に似通っています。ここで重要なポイントは、正しい理解に基づいて、適切なパターンと原則を適用することです。

IoC (制御の反転) コンテナー – フレームワーク

テクノロジ	選択すべき状況と理由
Microsoft Unity	**Microsoft patterns & practices** ▪ IoC と DI の実装に使用される、現時点で最も完全かつ軽量なマイクロソフト提供のフレームワーク ▪ Microsoft Public License (Ms-PL) によって提供されるオープンソース プロジェクト **詳細情報:** http://msdn.com/unity (英語)、http://unity.codeplex.com/ (英語)
Castle プロジェクト (Castle Windsor)	**Castle Stronghold** ▪ Castle はオープン ソースのプロジェクトで、最もよく使用されている IoC フレームワークの 1 つ **詳細情報:** http://www.castleproject.org/ (英語)
Microsoft MEF	**Managed Extensibility Framework (MEF)** ▪ MEF は Microsoft .NET Framework の一部で、UI の拡張性とプラグインに比重を置いた、ツールおよびアプリケーションの自動拡張を可能にするためのフレームワーク ▪ 従来型の IoC コンテナーではなく、動的かつ自動的な拡張を実現するフレームワークとしてとらえる必要があるが、一部の機能は典型的な IoC コンテナーと重複 ▪ Microsoft Public License (Ms-PL) によって提供されるオープンソース プロジェクト **詳細情報:** http://msdn.microsoft.com/ja-jp/library/dd460648.aspx、http://mef.codeplex.com/ (英語)
Ninject	**Nate Kohari 氏が開発** ▪ 依存性注入ツール。コードを理解しやすく、変更しやすくすると共に、エラーの発生を抑える、オープンソースのプロジェクト **詳細情報:** http://www.ninject.org/ (英語)
Spring.NET	**SpringSource (VMware の子会社)** ▪ エンタープライズ アプリケーション構築のフレームワークであり、IoC の機能に加え、AOP (アスペクト指向プログラミング) 機能もサポート ▪ オープン ソースのプロジェクト **詳細情報::** http://www.springframework.net/ (英語)
StructureMap	**.NET OSS コミュニティの開発者数名が開発** ▪ 2004 年 6 月以来公開されている、.NET 用の最も古い IoC/DI ツール ▪ オープン ソースのプロジェクト **詳細情報:** http://structuremap.sourceforge.net/Default.htm (英語)
Autofac	**.NET OSS コミュニティの開発者数名が開発** ▪ アプリケーションのサイズや複雑性が増大しても、変更が困難にならないよう、クラス間の依存関係を管理 ▪ オープン ソースのプロジェクト **詳細情報:** http://code.google.com/p/autofac/ (英語)
LinFu	**Philip Laureano 氏が開発** ▪ IoC、AOP、DbC、およびその他の機能を提供することで、CLR を拡張する一連のライブラリから成るフレームワーク ▪ オープン ソースのプロジェクト **詳細情報:** https://github.com/philiplaureano/LinFu (英語)

Funq	**Daniel Cazzulino 氏を始めとする .NET OSS コミュニティの開発者たちが開発** ▪ ラムダと汎用関数をファクトリとして使用することで、実行時のリフレクションをすべて解消する高パフォーマンスな DI フレームワーク ▪ オープン ソースのプロジェクト **詳細情報:** http://funq.codeplex.com/ (英語)

表 5-13

IoC/DI コンテナーのほとんどは、インターセプト パターンもサポートしています。これは、さらに別の間接指定レベルを使用したパターンです。この手法では、クライアントと実際のオブジェクトの間にオブジェクト (プロキシ) を配置します。このとき、クライアントの動作は、実際のオブジェクトと直接やり取りする場合と同じですが、プロキシは実際のオブジェクトおよび他のオブジェクトと随時連携し、処理をインターセプトして実行を代行します。インターセプトは AOP (アスペクト指向プログラミング) の実装手段として利用できます。

任意のアーキテクチャ スタイルへの依存性の注入

もう 1 点強調したいのが、DI/IoC を使用した分離設計は、ほぼすべてのアーキテクチャ アプローチによるアプリケーション開発に良い影響を与えるということです。分離設計は元来、ドメイン駆動設計 (DDD) やコマンド クエリ責務分離 (CQRS) を基礎としているため、当然ながら分離を必要とするアーキテクチャに使用するのが最も適しています。さらに、2 レイヤー アプローチや CRUD アプローチのように比較的シンプルなアーキテクチャ スタイル (たとえば、ASP.NET MVC および 2 つのレイヤーのみに基づいたシンプルなアプローチを使用する場合) でも、分離設計は優れた効果を発揮します。こうしたシンプルなコンテキストに DI/IoC および関連する原則を適用した場合でも、得られるメリットに変わりはありません。一方で、規模が大きく複雑なアプリケーションほど、分離アーキテクチャからより多くのメリットが得られることも確かです。

コア ビジネス アプリケーションのアーキテクチャ スタイル

ソフトウェア アーキテクトや開発者が使用するアーキテクチャ スタイルには、数多くの種類があります。次に、**.NET Framework** で実装できる標準的なアプローチの一部を示します。

- **従来型の複数層アーキテクチャ** (トップダウン式の論理レイヤー)
- **ドメイン駆動設計の複数層アーキテクチャ** (ドメイン層を中心とした論理レイヤー)
- **CQRS アーキテクチャ アプローチ** (論理レイヤーと物理階層)
- **3 階層アーキテクチャ** (物理階層)
- **N 階層アーキテクチャ** (物理階層)
- **サービス指向アーキテクチャ (SOA)** (高レベルのオーケストレーション)
- **イベント駆動型アーキテクチャ (EDA)** (高レベルのオーケストレーション)

特定の N 階層物理アーキテクチャを構築すれば、複数の層から成る論理アーキテクチャを展開できるため、上記の多くは補完的に使用されます。また、内部的に複数層アーキテクチャとして設計され、多くのサービスを組み合わせて提供される、サービス指向アーキテクチャ (SOA) や高レベルのイベント駆動型アーキテクチャ (EDA) なども使用できます。

ただし、忘れてはならないのは、1 つのアーキテクチャで、あらゆる状況に対応することはできないという点です。さらに言うなら、**大規模アプリケーションを扱う場合には通常、1 種類のトップレベル アーキテクチャのみを使用するのではなく**、多くのサブシステム (ドメイン駆動設計の用語で言うならば "境界づけられたコンテキスト") から成る大規模な複合アプリケーションを設計するようお勧めします。この場合は、サブシステムの性質および優先事項に基づき、さまざまなアーキテクチャ スタイルを適用することが正しいアプローチとなります。大規模アプリケーション内の付帯的サブシステム (境界づけられたコンテキスト) については、シンプルなデータ駆動型 CRUD アプローチを選択し、コア ドメイン (コア ビジネス) に関してのみ、ドメイン駆動型のアプローチおよびパターンのような、より高度な分離アーキテクチャを必要に応じて適用することもできます (図 5-29 を参照)。

さまざまなアーキテクチャを適用したサブシステム/コンテキスト境界

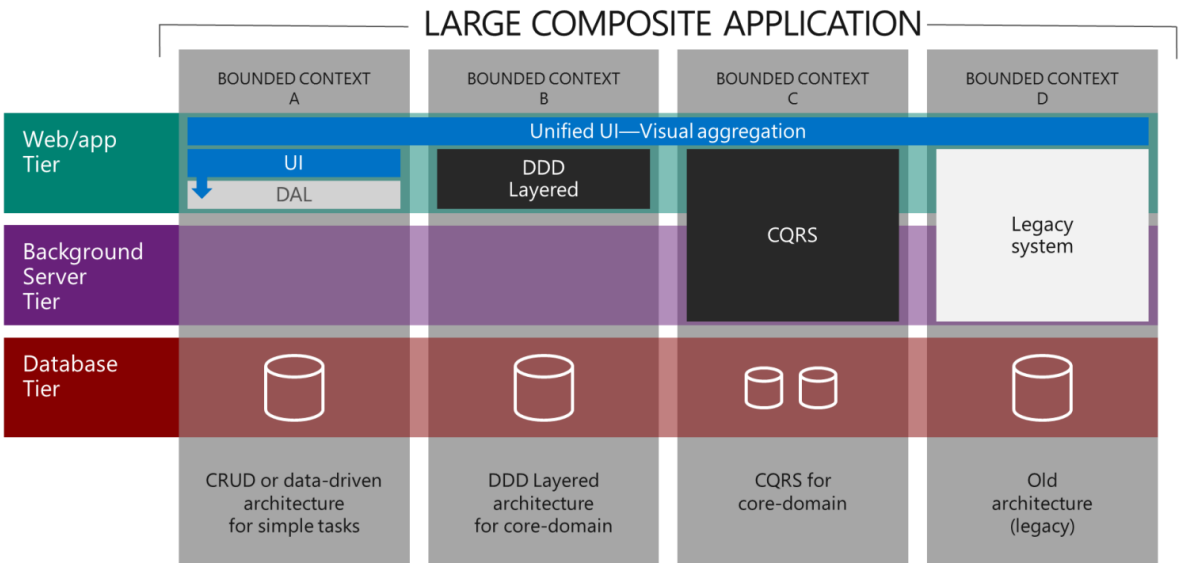

図 5-29

境界づけられたコンテキスト (BC) はサブシステムよりも厳密な用語です (同じ意味で使用される場合もあります)。**ドメイン駆動設計**における**本来の定義**では、「特定のモデルの適用可能な範囲。境界づけられたコンテキストを定めることで、一貫性の必要なものや、独立して開発できるものについて、チームのメンバーは明確な共通認識を持つことができる」と説明されています (Eric Evans 氏による定義)。ここで重要となる特徴は、各境界づけられたコンテキストには固有のドメイン モデルがあり、多くの場合、固有のデータベース スキーマおよびデータベースも備えているという点です。

たった 1 つのアーキテクチャであらゆるケースに対応できるといった万能の解決策は存在しません。"すべてのアプリケーションやサブシステム (境界づけられたコンテキスト) に有効な単一のアーキテクチャ" を手に入れることは不可能です。それぞれのサブシステム (境界づけられたコンテキスト) の優先事項に応じ、異なるアプローチを選択する必要があります。何年にもわたってビジネス ロジックが発展および成長するようなアプリケーションに、データ駆動型のきわめてシンプルなアプローチ (CRUD および RAD) を使用してしまうと、大きな失敗につながりかねません。こうしたアプリケーションはいずれ "大きな泥だんご" (乱雑に構築された無秩序な状態) となり、長期的に保守できないためです。その逆の場合にも同じことが言えます。たとえば、すべての開発に共通の "標準アプローチ" を使用したいがために、非常に高度な最先端アーキテクチャを実装して、きわめてシンプルなデータ駆動型のテーブル メンテナンス アプリケーションを構築することは合理的ではありません。これでは "ロケットの部品で自転車を作る" ようなことになり、非常にコストがかかります。大規模ビジネス アプリケーションには、"分割統治" こそが最適なアプローチです。

つまりは、**アプローチおよびアーキテクチャ スタイルは、トップレベルの規範的アーキテクチャとしてではなく、大規模アプリケーション内の特定の領域に対して適用することが推奨**されるということです。

すると当然ながら、「そうしたサブシステムまたは境界づけられたコンテキストをどうやって区別すればいいのか」や、さらに進んで、「そうした境界づけられたコンテキストをどうやって統合すればいいのか」という新たな疑問が湧いてくるでしょう。このガイドではこうした疑問点の多くに答えていますが、ドメイン駆動設計の一般的な内容およびその他の関連アプローチの詳細については、次の参考資料を参照してください。

参考資料	
ドメイン駆動設計関連のコミュニティとコンテンツ	http://dddcommunity.org/uncategorized/bounded-context/ (英語)
	http://dddcommunity.org (英語)
	http://dddcommunity.org/books/ (英語)
	http://www.agilification.com/file.axd?file=2010/8/CQRS+DDD+Notes+Greg+Young.pdf (英語)
スノーマン アーキテクチャ: Vertically Aligned Synergistically Partitioned (VASP) アーキテクチャ (境界づけられたコンテキストに相当するコンセプト) Roger Sessions 氏のブログ記事より	http://simplearchitectures.blogspot.com/2012/12/the-snowman-architecture-part-three.html (英語)
境界づけられたコンテキスト間の連携 (Microsoft patterns & practices ガイド内「CQRS Journey」)	http://msdn.microsoft.com/en-us/library/jj591572.aspx (英語)
DDD 境界づけられたコンテキストでの Entity Framework ドメイン モデル (純粋な分離 DDD BC を境界づけられたコンテキストのビューで代替)	http://msdn.microsoft.com/ja-jp/magazine/jj883952.aspx (機械翻訳)

ミッション クリティカルなエンタープライズ アプリケーションの刷新

ミッション クリティカルな大規模アプリケーションを新たに作成する場合は、拡張性と弾力性に優れた最新アプローチの適用が推奨されます。ただし、既存のエンタープライズ アプリケーションのほとんどは非同期エンジンおよびイベント駆動型アプローチに基づいて構築されておらず、スケールアウト アーキテクチャや弾力性のあるアーキテクチャをサポートしていません。多くの場合、こうしたアプリケーションをレガシ システムととらえ、最新の (Web ベースまたはサービスベースの) ファサードに統合する方がよいでしょう。こうした統合は、さまざまなチャネルを集約するメッセージ指向の非同期統合システムを新たに構築する際に、同時に実施することをお勧めします。従来型のアプリケーションを変更および更新する際は、イノベーションの推進 (モダン アプリケーションの場合も同様)、また拡張性およびパフォーマンス関連の領域に重点的に取り組みます。ただし、レガシ システムを扱う場合は、ミッション クリティカルなアプリケーション全体をゼロから再構築するのではなく、クラウド内にフロントエンド サービスとキャッシュ (上記では "ファサード" として言及) を構築することで、より簡単に拡張性を向上できます。そのうえで、予算と時間が許すのであれば、並行してファサードの内部のシステムを再設計することも可能です。

こうした刷新は、まさにこのガイドが目的とするシナリオの 1 つであり、以降で詳しく説明します。

ミッション クリティカルな大規模カスタム アプリケーションのシナリオ

以降のセクションで、ミッション クリティカルな大規模カスタム アプリケーションの典型的なシナリオについて説明します。

- **シナリオ: ドメイン駆動型サブシステム (境界づけられたコンテキスト)**
- **シナリオ: CQRS サブシステム (境界づけられたコンテキスト)**
- **シナリオ: さまざまな境界づけられたコンテキストの連携**
- **シナリオ: ミッション クリティカルなレガシ エンタープライズ アプリケーションの刷新**

シナリオ: ドメイン駆動型サブシステム (境界づけられたコンテキスト)

このシナリオでは、複雑なコア ビジネス サブシステムが対象となります。ここで言う "複雑" とは、何年にもわたって発達し、絶えず変化するドメインまたはビジネス ロジックを大量に備えたシステムを意味します。

プレゼンテーション層の形式はさまざま (Web、デスクトップ、または最新のモバイル デバイス) でも、資金、設計作業、および開発作業の大半は、一般にビジネス/ドメイン ロジックが配置されるサーバー/サービス側に費やされます。

この種類のアプリケーション (またはサブシステム) の特徴は、ライフサイクルが比較的長く、発展に伴う相当量の変化に適応しなければならないという点です。こうしたアプリケーションには継続的な保守が不可欠となるため、長期的な保守と変更を容易に行えるアーキテクチャ、設計、および開発のために投資を進めることが、最も重要な優先事項の 1 つとなります。アプリケーションがかなりの規模に拡大した場合にも、実際のコードへの影響は最小限に抑えたいはずです。こうした理由から、モノリシック アプローチはこのようなアプリケーションには不適合だとされています。

既に述べたとおり、複雑なアプリケーションでは、ビジネス ルール (ドメイン ロジック) の動作を変更することが珍しくありません。そのため、持続的な保守を可能にするには、ドメイン ロジック層上で変更や構築を行い、隔離された環境でテスト (単体テストおよびモック テスト) を実行できるような、容易かつ依存性のない方法が不可欠となります。この重要な目標を達成するには、ドメイン モデル (ロジックおよびビジネス ルールを含んだエンティティ モデル) と、システム内の他のレイヤー (プレゼンテーション層、インフラストラクチャ層、データ永続化層など) との結合を最小限に抑える必要があります。そのための手段として、このコンテキストでは通常、抽象に基づく永続化非依存モデルおよび分離設計が求められます。

加えて、こうした長期用アプリケーションのライフサイクルは一般的に非常に長い (少なくとも数年単位の) ため、いずれは利用可能なテクノロジが発展したり置き換わったりします。そのため、長期用システムにとっての重要な目標となるのは、インフラストラクチャやテクノロジを変更したときに、アプリケーションの他の部分 (ドメイン モデル、ドメイン ルール、他の階層と連携しているデータ構造 (DTO) など) への影響を最小限に抑えながら、変更に対応できるようにすることです。アプリケーションのインフラストラクチャ テクノロジ (データ アクセス、サービス バス、セキュリティ操作、インストルメンテーションなど) の変更は、アプリケーション内の上位レイヤー、特にドメイン モデル層に、できるだけ影響を与えないように実施しなくてはなりません。

長期用アプリケーション アーキテクチャでは、こうしたクラス/オブジェクト間および垂直的領域 (境界づけられたコンテキスト) 間の依存性の軽減がますます促進される傾向にあります。**ドメイン駆動設計 (DDD)** は、こうした傾向に後押しされた、複雑な機能を持つソフトウェア システムの開発アプローチの代表例です。

DDD アプローチでは、各種の手法を駆使してドメインを分析し、分析結果を反映した概念モデルを構築します。このモデルは、構築するソリューションの基礎として使用できます。DDD アプローチの分析およびモデルは、大規模で複雑なドメインのソリューションを構築する場合に特に適しています。また DDD は、ソフトウェア開発プロセスの他の側面にも影響を及ぼし、複雑性への対処に効果を発揮します。

ドメイン駆動設計の基本原則は次のとおりです。

- **コア ドメイン**の**重視**
- **"ドメインのエキスパート"** と**開発チーム**との間の**直接的なコラボレーション**
- 明示的な境界づけられたコンテキスト (サブシステム) 内での**ユビキタス言語**の使用
- 所定の DDD **アーキテクチャおよび実績ある設計パターン**の適用

つまり、DDD は、単なるアーキテクチャの提案なのではなく、プロジェクトを管理すると共に、チームで協力して **"知識を咀嚼"** し、だれもが理解できる **"ユビキタス言語"** を特定するための手段だと言えます。これにより、**開発チームとドメインのエキスパートの間で継続的な情報交換が行われ**、ドメインの重要な部分がモデルに反映されると共に、より高度な抽象、リファクタリング、およびモデル改善の持続的な追求が可能となります。また、時間が経つにつれて、開発者はドメインに関してより多くの知識を蓄え、ドメインのエキスパートはドメインに関する知識を様式化したり、より完全なものにしたりする能力を高めることができます。

DDD の論理アーキテクチャを設計する際は、複雑な境界づけられたコンテキストを複数層に分割することをお勧めします。この設計では各レイヤーを 1 つのまとまりと考えますが、当然ながらシステム内のさまざまなレイヤーを定義する必要があります。オブジェクト間の依存関係は、抽象に基づいて決定されるようにします。**さらに、ドメイン モデル層を明確に定義し、特定のテクノロジ** (データ アクセス テクノロジなど) **に基づいたインフラストラクチャ層のような、他のレイヤーから切り離します**。コア ドメイン アプリケーションでは、ドメイン層がソフトウェアの中核なので、ドメインを中心としてあらゆる事柄を構成する必要があります。

したがって、具体的な境界づけられたコンテキスト内では、ドメイン モデルに関係するすべてのコード (ドメイン エンティティ、ドメイン サービスなど) を単一レイヤーに配置するようにします。ドメイン モデルの目的は、それ自体の保持または保存でも、下位のテクノロジとの直接的な依存関係を定義することでもありません。重視すべきは、ドメイン モデル (データおよびロジック) をわかりやすく示すことのみです。"永続化非依存の原則" に基づく Code First POCO エンティティを使用すると、**.NET Framework** および **Entity Framework** によって、このモデルの実装が大幅に簡略化されます。

このレイヤーは、図 5-30 に示すような構成になり (DDD アーキテクチャ スタイルの傾向を反映)、加えて境界づけられたコンテキストなどのさまざまなアーキテクチャ アプローチを併用できます。

大規模アプリケーション内のシンプルな DDD 複数層アーキテクチャ

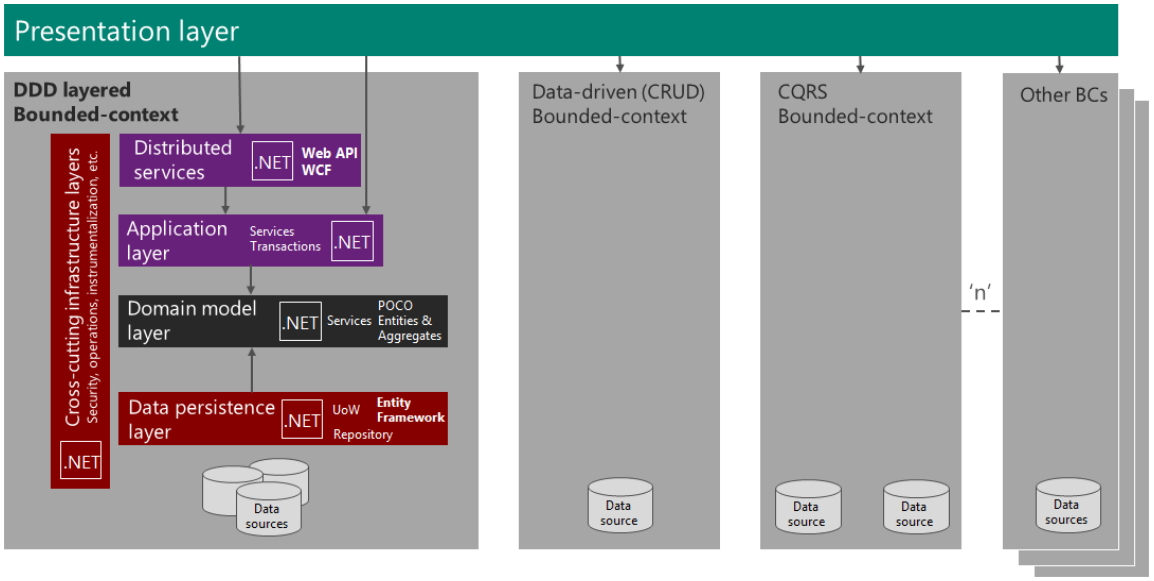

図 5-30

この図では、さまざまなコンポーネント間やレイヤー間の依存関係を矢印で表しています。DDD の複数層アプローチと従来型のトップダウン式複数層アプローチが主に異なるのは、ドメイン モデル層に他のレイヤーとの依存関係が存在しない点です。このアプローチでは、抽象 (ドメイン モデル層内で定義されたインターフェイス。関連する実装クラスは他のレイヤーに配置) および POCO エンティティ モデル (**Microsoft Entity Framework** POCO エンティティおよび Code First アプローチを使用する場合など) が基礎となっています。そのため、ドメイン モデル層は、使用されるデータ アクセス テクノロジに関し、"永続化非依存の原則" に準拠しています。

高度なアプローチ (DDD や CQRS など) は、アプリケーションのコア ドメインに重点を置いた境界づけられたコンテキスト内のみに適用します。その他の、よりシンプルで付帯的な境界づけられたコンテキストやサブシステムは、図 5-30 に示したデータ駆動型の CRUD アプローチで実装することを検討してください。

図 5-30 で示した論理アーキテクチャでは、内部アーキテクチャ アプローチを使用しています。トップレベル アーキテクチャを適用し、さまざまなサービスや、そうしたサービスを展開する他の特定の物理/インフラストラクチャ アーキテクチャ (階層、およびプライベートまたはパブリック クラウド内の弾力的なサービスなど) を組み合わせることもできます (図 5-31 および図 5-32 を参照)。

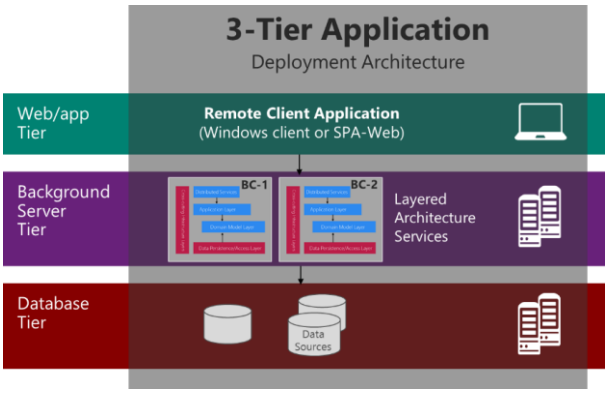

図 5-31

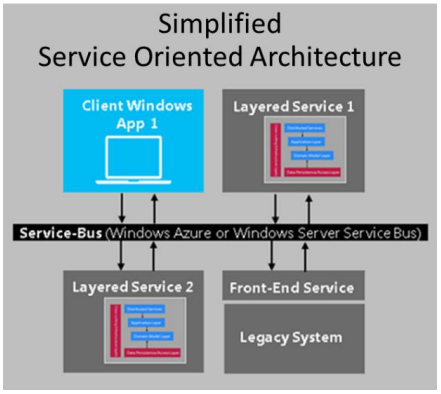

図 5-32

3 階層アプリケーションで使用されるサービスは、多数の境界づけられたコンテキストによって構成することができます。各境界づけられたコンテキストに固有のデータ ソースを用意することも、複数の境界づけられたコンテキストで 1 つのソースを共有することも可能です。重要なポイントは、境界づけられたコンテキストにはそれぞれ固有のドメイン エンティティ モデルがあるということで、これは通常、固有のデータベース スキーマを指します。

SOA では、必須ではありませんが、1 つのサービスを 1 つの境界づけられたコンテキストと対応させることができます。また、境界づけられたコンテキストは複数のサービスで構成することも可能ですし、反対に大規模なサービスを複数の境界づけられたコンテキストで構成することもできます。

境界づけられたコンテキストとドメイン モデル

境界づけられたコンテキストでは、実社会の (ドメインの) 特有のコンセプトを表わす、ドメインのさまざまな側面を扱います。境界づけられたコンテキストは自律的なコンポーネントであり、それぞれ固有のドメイン モデルと、固有のユビキタス言語 (具体的なドメイン用語) が存在します。境界づけられたコンテキスト間では、実行時に一切の依存関係が生じないようにし、単独で動作可能にすることが適切です。

境界づけられたコンテキストは、一貫性の必要なものや、独立して開発できるものについて、チームのメンバーに明確な共通認識をもたらします。そのため、**特定のモデルの適用可能範囲および一貫性に基づいて、境界づけられたコンテキストごとに境界を定義する必要があります**。こうした理由から、各境界づけられたコンテキストには、それぞれのモデルに影響を及ぼす固有のユビキタス言語がある場合が一般的です。類似したコンセプトやエンティティであっても、別の境界づけられたコンテキストに存在する場合には属性や、使用されている名前および用語が異なることもあります。

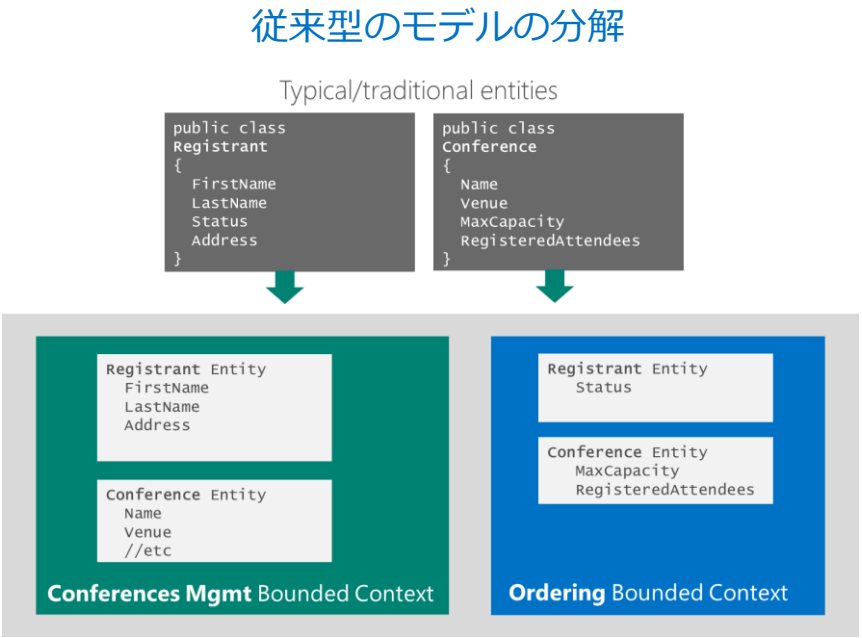

図 5-33

図 5-33 は、従来型の "アップフロント型エンティティ" (アンチパターン) を複数の境界づけられたコンテキスト (BC) に分解するための適用可能なアプローチを示した疑似エンティティ ダイアグラムです。境界づけられたコンテキストにはそれぞれ異なるモデルが必要です。複数のモデル内のエンティティに (図 5-33 のように) 同名のコンセプトが存在する場合も少なくありませんが、名前は同じでも、属性が異なっている可能性があります。

この例では、エンティティ間で一貫させる必要のある属性と、各エンティティに与える影響に基づいてエンティティを分割しています。たとえば、ConferencesManagementBC という境界づけられたコンテキストの Registrant エンティティ クラスに、名前、住所、勤務先データといった個人データが含まれているとします。ただし、Status 属性は、会議の RegisteredAttendees 属性と一致している必要があるため、OrderingBC という別の BC 内の Registrant エンティティ クラスに含める方が適切です。そのため、BC の境界の区別では、データ間の一貫性のほか、他の属性に影響を及ぼす属性やまったく影響を及ぼさない属性も確認する必要があります。たとえば、登録者の FirstName を変更しても、会議登録の属性には影響を及ぼしません。

時には、複数の境界づけられたコンテキストに、同じ属性を重複して設定しなければならないこともあります (ただし、最小限に抑えるようにします)。こうした場合は、"結果整合性 (英語)" を採用して対処する必要があります。結果整合性は、異なる集約 (一貫性あるエンティティのセット) 間でも使用します。

また、エンティティ間で必要となる関係を考慮 (グループ化して集約 (英語) を作成) するのも、BC を区別する効果的な手段です。初期モデルの一部の領域と、計画中のアプリケーションの他の領域とに一切関わりがない場合、対象の領域は境界づけられたコンテキストにするべきだと言えます。このプロセスを図 5-34 に示します。

各境界づけられたコンテキストおよびドメイン モデルは、同じ状況 (ドメイン) を異なる観点から見たものですが、使用方法と目的がそれぞれ異なります。

さらにわかりやすい表現として、"ドメインが世界 (現実) だとすれば、BC とそのドメイン モデルはその世界の地図である" という Eric Evans 氏の有名なたとえもあります。

エンティティを設計および実装する際は、DDD の **"ドメイン モデル貧血症"** (アンチパターン) が発生しないようにすることも重要です。エンティティ クラスでは、そのエンティティ クラスのメソッド内に固有のドメイン ロジックを含めるようにします。

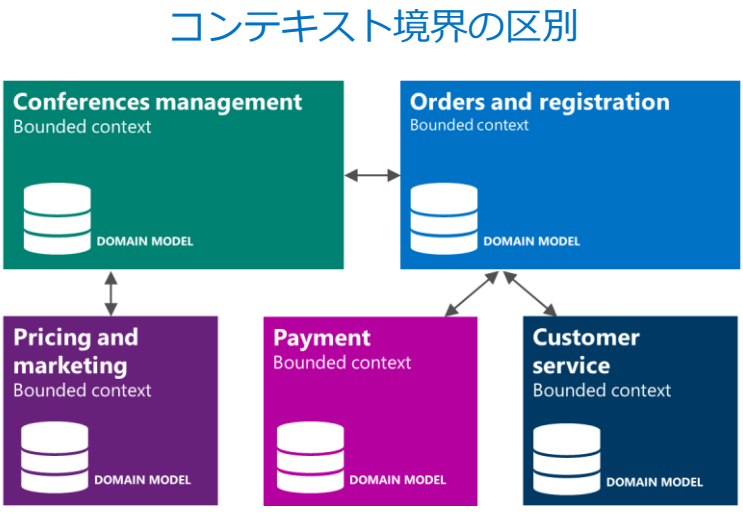

図 5-34

マイクロソフトの開発テクノロジを使用して、ドメイン モデル エンティティを実装する方法にはさまざまな種類があります。ただしマイクロソフトでは、**"永続化非依存"** の原則に常に準拠した (エンティティ自体がインフラストラクチャ テクノロジに依存していない)、POCO (Plain Old CLR Object) エンティティ (プレーンなクラス) を使用した方法をお勧めします。また、モデルは、**Microsoft Entity Framework** および **Code First アプローチのマッピング**のようなインフラストラクチャのデータ永続化テクノロジと疎結合されていなければなりません。その他の選択肢としては、NHibernate のような別の O/RM を使用する方法や、プレーンな ADO.NET に基づいて、カスタム アプローチを "ゼロから構築" する方法があります。ただし、どの方法を選んだ場合にも、最終的な依存関係 (エンティティおよびリポジトリのコントラクトまたはインターフェイス) は、インフラストラクチャ テクノロジ内ではなくモデル内で定義する必要があります。この点は、DDD の複数層アーキテクチャと通常の複数層アーキテクチャとの決定的な違いです。

.NET による、ドメイン駆動設計パターンおよびその他の機能の実装

ドメイン駆動設計を適用する際は、各レイヤーに異なるパターンを実装するのが一般的です。こうしたパターンには、ドメイン エンティティ、集約、集約ルート、集約間の疎結合、集約ストレージ、値/オブジェクト、リポジトリ、作業単位、ドメイン サービス、ファクトリ、ドメイン イベント、結果整合性などがあります (これらのパターンの詳細については、以降の DDD に関する参考資料を参照)。

図 5-35 に、**マイクロソフトのテクノロジ**を使用した、**典型的な DDD パターンおよびインフラストラクチャ機能の実装パス**の推奨例を示します。

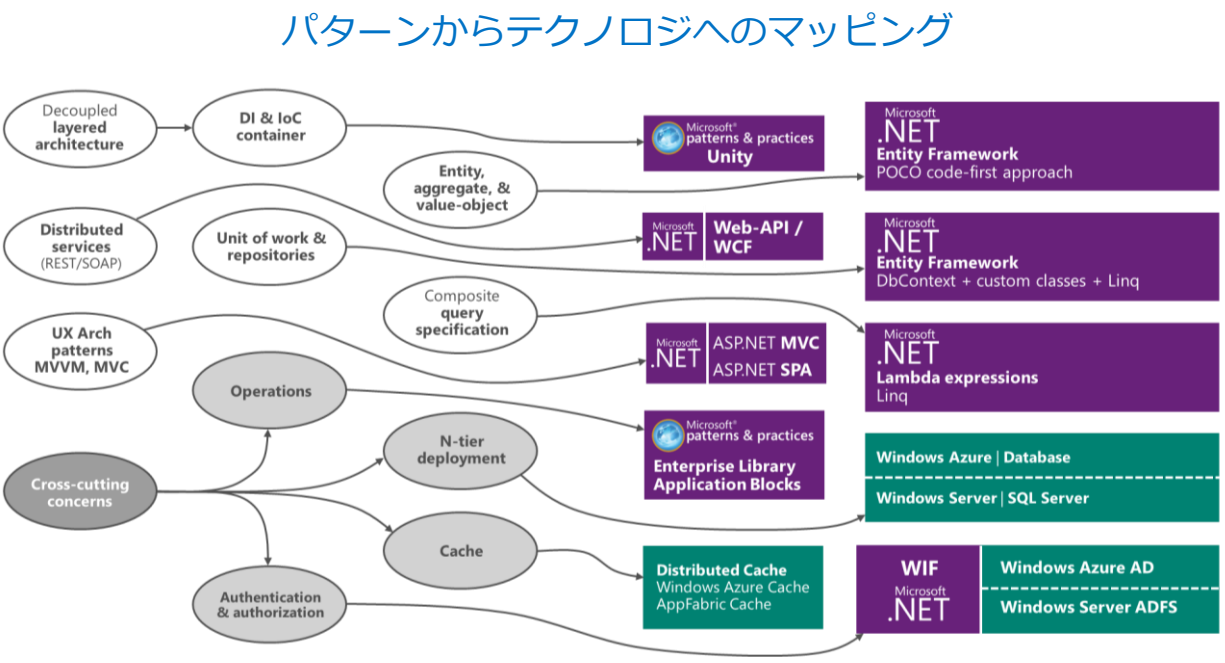

図 5-35

.NET を通じてリレーショナル データベースを利用し、DDD パターンを実装する場合は、**Microsoft Entity Framework** などの O/RM で中継することにより、ドメイン エンティティ、集約、集約ルート、リポジトリ、および作業単位の各パターンを簡単に実装できます。ただし、この方法は、アプリケーションの優先事項および要件に応じて適宜使用します。NoSQL データベース、つまりドキュメント データベース (リレーショナルではないため、O/RM を使用しない) に基づいた集約ストレージ パターンを使用する方が適切なケースもあります (イベント ソーシング (英語) および集約イベント ストレージを利用する場合など)。

参考として、MSDN.com と Microsoft patterns & practices では、分野横断的な注意事項、運用、セキュリティ、およびインストルメンテーションをテーマとした資料を豊富に提供しています。

参考資料	
ドメイン モデル貧血症	http://www.martinfowler.com/bliki/AnemicDomainModel.html (英語)
POCO (Plain Old CLR Object) エンティティ クラス	http://en.wikipedia.org/wiki/Plain_Old_CLR_Object (英語)
ドメイン モデル パターンを使用する	http://msdn.microsoft.com/ja-jp/magazine/ee236415.aspx
Entity Framework	http://msdn.microsoft.com/en-us/library/bb399567.aspx (英語)
イベント ソーシング	http://codebetter.com/gregyoung/2010/02/20/why-use-event-sourcing/ (英語)

シナリオ: CQRS サブシステム (境界づけられたコンテキスト)

このシナリオでは、CQRS (コマンド クエリ責務分離) に基づいたコア ビジネス サブシステムについて説明します。通常、CQRS は DDD と関連があり、アプリケーション リソースが十分でない環境や、アプリケーションに高い拡張性が求められる環境 (インターネットを介してユーザー数を制限せずに提供される、ミッション クリティカルなアプリケーションに多く見られる環境) と特に深い結び付きがあります。

"コマンド クエリ分離" 原則は Betrand Meyer 氏が提唱した用語で、オブジェクトのメソッドは、コマンドまたはクエリのいずれかにするべきであるという原則です。クエリはデータを返しますが、オブジェクトの状態は変化させません。一方、コマンドはオブジェクトの状態を変更しますが、データを返すことはありません。システム内の状態を変化させるもの、変化させないものをよく理解しやすくなることが、この原則のメリットです。

CQRS では、この原則をさらに一歩進めたシンプルなパターンを定義しています。

「CQRS とは要するに、以前は 1 つだったオブジェクトを 2 つのオブジェクトとして作成することである。メソッドは、コマンドであるかクエリであるかに応じて分離される (コマンド クエリ分離原則でも同じ定義が使用されており、提唱者の Meyer 氏は『コマンドとは状態を変化させるメソッドであり、クエリは値を返すメソッドである』と説明している)」
—Greg Young 氏—

図 5-36 は、エンタープライズ システムの一部 (BC) に対して CQRS パターンを適用する場合の典型例を示しています。この図では、この範囲において読み取り側と書き込み側を分離するには、いつ、どうやって CQRS パターンを適用すればよいかわかるようになっています。**CQRS は、Windows Azure による多くのクラウド シナリオにも適しています。**

大規模アプリケーション内での CQRS アプローチ

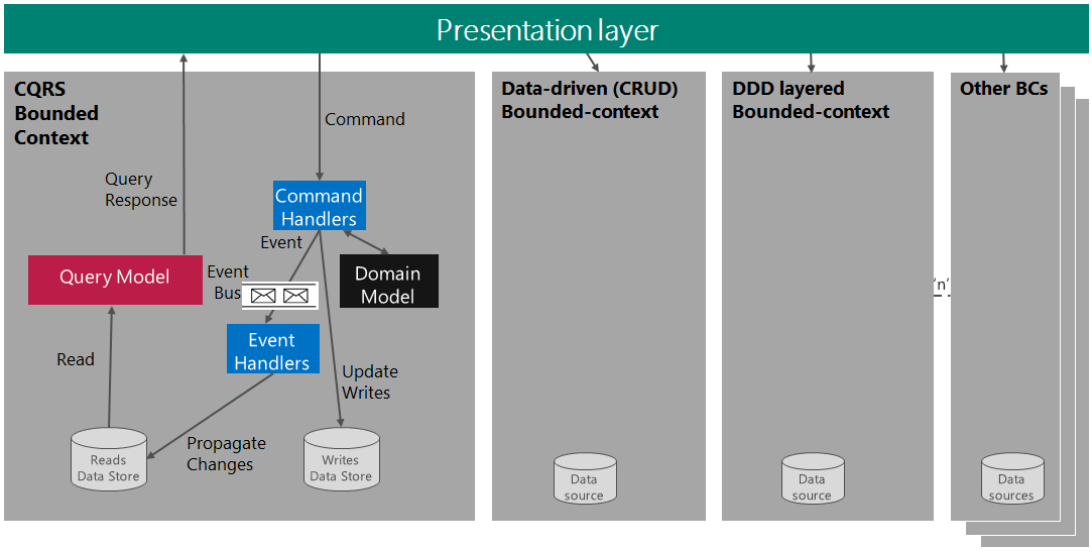

図 5-36

参考資料	
CQRS Journey (Microsoft patterns & practices ガイド)	http://msdn.microsoft.com/en-us/library/jj554200.aspx (英語)
Windows Azure での CQRS	http://msdn.microsoft.com/ja-jp/magazine/gg983487.aspx
Greg Young 氏の CQRS に関するドキュメント	http://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf (英語)
Udi Dahan 氏の CQRS に関するブログ記事	http://www.udidahan.com/2009/12/09/clarified-cqrs/ (英語)

シナリオ: さまざまな境界づけられたコンテキストの連携

境界づけられたコンテキスト間では、実行時に一切の依存関係が生じないようにし、単独で動作可能にすることが適切です。大規模アプリケーションや複雑なアプリケーションには、相互連携された境界づけられたコンテキスト (BC) がいくつか含まれる場合があり、**各 BC は固有のドメイン モデルを持っています** (図 5-37 を参照)。大規模アプリケーションでは、BC の数が数十に上ることもあります。

多くのサブシステムを持つ大規模アプリケーション

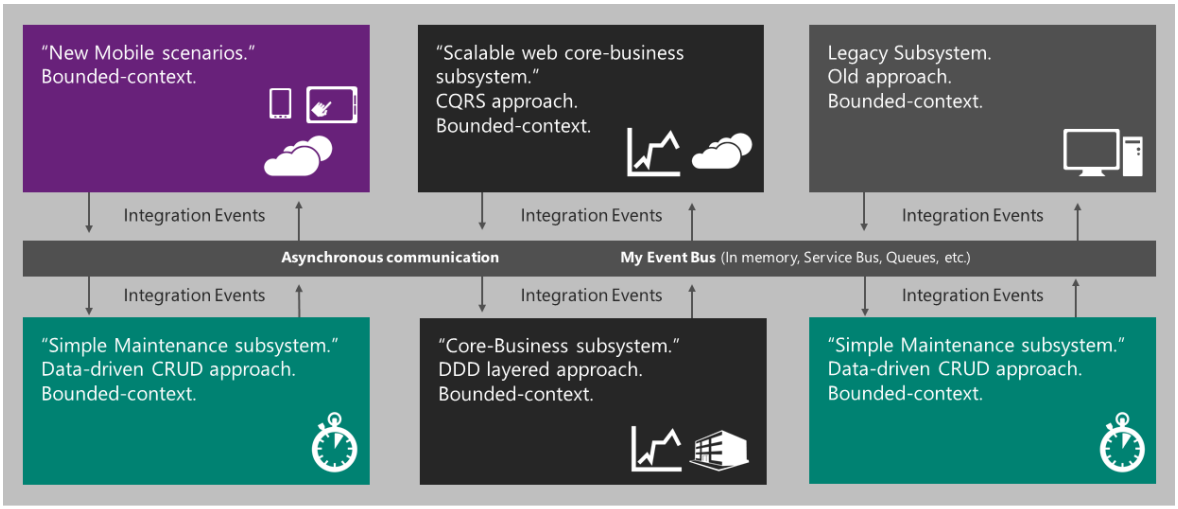

図 5-37

ただし、それぞれの BC は、1 つの包括的なシステム、またはより大きなアプリケーションの一部であるため、互いにデータをやり取りできなくてはなりません。BC どうしを切り離す必要があっても、1 つの BC 内で発生した情報の変更やイベントは、他の BC にも影響を与えます。こういった種類の BC 間通信には、イベントに基づいた非同期通信を使用します。境界づけられたコンテキストは、その BC の外部で発生したイベントに応答することができます。また、他の BC がサブスクライブ可能なイベントの発行にも対応しています。イベント (既に発生した事柄に関して情報を発行する、一方向の非同期メッセージ) を使用すると、境界づけられたコンテキスト間の疎結合の維持が可能になります。こうしたイベントは、境界づけられたコンテキスト間を行き来するイベントという意味で、統合イベントと呼ぶことができます。

境界づけられたコンテキスト間の統合イベントおよび非同期通信を使用することは、必ずしも SOA やサービス バスの使用につながるわけではありません。こうした通信は、各種のインフラストラクチャ テクノロジによって実装可能な、いずれかのタイプのイベント バスを通じて行うこともできます。**イベント バス**の実装方法としては、**シンプルなインメモリ バス**、名前付きパイプによるプロセス間通信イベント バス (同一サーバー内のイベント バス。目的を果たすうえで十分役立つ場合、または概念実証のために使用)、MSMQ **(Microsoft メッセージ キュー)** のようなメッセージ キューのほか、実稼働用システムの場合なら、**Windows Server サービス バス**や **Windows Azure サービス バス**のようなフル機能のサービス バス (または NServiceBus などのサード パーティ製サービス バス) を選択することもできます。

BC 間の統合イベントへの対処方法は、CQRS でのイベントへの対処方法とほぼ同じです。イベント (いずれかのタイプのシンプルなメッセージ) は、ドメイン ストレージ内で何か (ドメイン ストレージでの更新処理など) が発生した後で作成される必要があります。イベントは、過去の事柄を表わす名前を付けて、イベント バスに発行されます。このとき、イベント バスがパブリッシュ / サブスクライブ メカニズムをサポートしていなければなりません。イベント バス内にメッセージが発行されると、そのイベント バスをサブスクライブするすべての BC は、発行されたイベントを取得して、自身のドメイン モデルに反映するためのアクションを実行します。このアプローチにより、**すべての境界づけられたコンテキスト間で "イベントの一貫性" が維持されます**。

コンテキスト マップ

システムが大規模で、何十という境界づけられたコンテキストと何百種類もの統合イベントがある場合、関係性の把握は容易ではありません。コンテキスト マップは、どの境界づけられたコンテキストがどの統合イベントを発行し、どの境界づけられたコンテキストがどの統合イベントをサブスクライブしているかを示す、大切なレコードです。

加えて、境界づけられたコンテキストの数や統合イベントの種類が大幅に増加した場合は、パブリッシュ / サブスクライブ メカニズムをサポートするサービス バスがきわめて有用になります。

コンテキスト間の関係

2 つ以上のコンテキスト間に存在するさまざまな関係は、各コンテキストに関わる複数のチーム間でどの程度情報をやり取りするか、他チームに対する管理権限をどの程度持っているかが大きく関係します。たとえば、実稼働環境のシステムや運用停止になったシステムのような、特定のコンテキストでは変更処理を実行できない場合があります。

共有カーネル

このアプローチは、エンドツーエンドのアプリケーションを構築する場合のような、"グリーン フィールド" シナリオに適しています。こうしたシナリオで、2 つ以上のコンテキストがあり、それぞれのコンテキストを利用するチーム間で情報が頻繁にやり取りされる場合は、すべてのコンテキストに共通する特定のオブジェクトに対して、共有の責務を作成してもよいでしょう。たとえば、1 つのドメイン モデル内にある特定の共有領域を、2 つ以上の境界づけられたコンテキストで共有する場合が考えられます。こうしたオブジェクトは、全コンテキストで使用する、いわゆる共有カーネルです。共有カーネル内のいずれかのオブジェクトに変更を加えるには、関係するコンテキストを所有するすべてのチームから承認を得る必要があります。ドメイン モデルの同じ領域を共有する際は、チーム間の良好なコミュニケーションを促進することが重要です (ただし、該当する境界づけられたコンテキストでは、別途独自のドメイン モデルを保持できます)。

シナリオ: ミッション クリティカルなレガシ エンタープライズ アプリケーションの刷新

これは、ライフサイクルが非常に長く、絶えず発展するミッション クリティカルな大規模システムでのきわめて重要なシナリオです。企業内には、古いけれども重要性の高い、刷新を要するアプリケーションが少なからず存在しているでしょう。通常、こうしたアプリケーションについては、最初のステップとして、システムの外観または一部のみを新しくすることをお勧めします。**Microsoft のモバイル アプリケーション (Windows ストア または Windows Phone 向け)** を追加すれば、新しいエクスペリエンスとシナリオ への対応が可能になります。わかりやすい例として、ホスト アプリケーションや大規模なレガシ ERP システムを想像してください。多くの場合、こうしたシステムは、全体を一度に移行したり刷新したりすることはできません。

レガシ システムを刷新するには、まず**サービス ファサード**の作成から始めます。これには、レガシ システムをラップし、クラウド内の永続性キャッシュのような拡張性に優れた新しいシステムに対して公開することで、スケールアウトを可能にすると共に、多くの新しいモバイル クライアントからサービスを利用できるようにする目的があります。このアプローチによって、レガシ システムでは制限されていた拡張性がクラウド システム上では理論上無制限となり、新しいチャネルやユーザー (顧客の直接アクセスなど) に対応できるようになります。

ただし、長期的に使用される大規模コア ビジネス アプリケーションには、従来型のレガシ システムとの一貫性が必要な新しい境界づけられたコンテキスト (サブシステム) が含まれる場合があります。このときにも、一貫性を維持するために、新しい境界づけられたコンテキストと従来型のレガシ サブシステムとを連携させ、統合する必要があります。これには、腐敗防止層のような所定のアプローチの使用が求められます。

腐敗防止層

特にチーム間のやり取りが少ない場合など、状況によっては、境界づけられたコンテキストの統合に関して最も重要なパターンとなります。また、統合対象の境界づけられたコンテキストの 1 つが旧式のものであり、異なるユビキタス言語が使用されている (そのため、モデルがまったく異なる) 可能性がある場合、新しい境界づけられたコンテキストへの影響を回避するためにも重要となるパターンです。こうした状況では、**腐敗防止層**の実装が効果を発揮します。腐敗防止層は、

複数のコンテキストの間に配置され、(モデル オブジェクトおよび統合イベントに関連した) 必要な変換処理を実行する中間層です。

腐敗防止層を使用した、従来システムの刷新

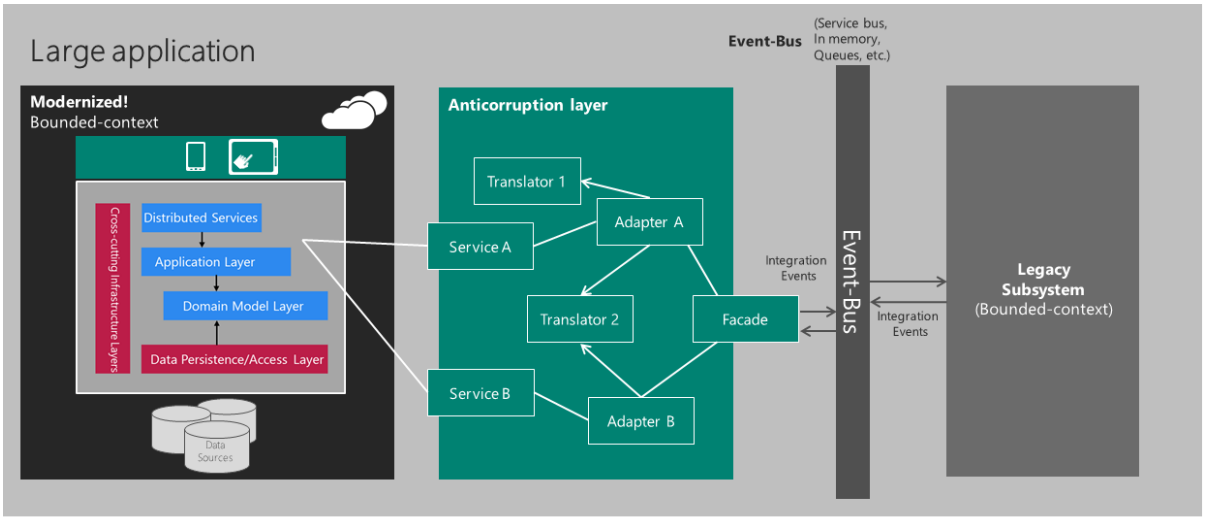

図 5-38

腐敗防止層は、アダプター、変換サービス、およびファサードという、3 種類のコンポーネントで構成されます。まず、ファサードは、他のコンテキストとの情報のやり取りを簡略化し、公開する機能を最小限に抑える役割を果たします。ファサードに関して重要なのは、他のコンテキストのモデル要素の観点から定義すべきであるという点です。これを忘れると、変換する際に、他のシステムへのアクセスと混同してしまう恐れがあります。ファサードの設計が終わったら、他のコンテキストのインターフェイス/イベントを変更し、所有するコンテキストで求められるインターフェイスに適応させるため、アダプターを配置します。最後に、変換サービスを使用して、所有するコンテキストの各要素と、他のコンテキスト ファサードの対応する要素とのマッピング (またはその逆のマッピング) を行います。

この腐敗防止層は、統合イベントの管理にも不可欠です。境界づけられたコンテキストはそれぞれ独立しているため、変更や更新が個別に行われたり、こうした変更によって境界づけられたコンテキストの発行するイベントが変更されたりすることもあります。該当する変更としては、新しいイベントの導入、イベントの使用中止、イベント名の変更、ペイロード内の情報の追加または削除に伴うイベント定義の変更などが挙げられます。境界づけられたコンテキストは、別の境界づけられたコンテキストに変更が加えられても、できるだけ影響を受けないようにする必要があります。腐敗防止層には、受信する統合イベントの妥当性を検証する役目があります。また、受信する統合イベントを変換するために使用することもできます。

腐敗防止層の実装はカスタマイズできます (プレーンな .NET)。これは初期段階でのアプローチです。より複雑なコンテキストの場合や、拡張対象のレガシ サブシステムが市販製品 (SAP をはじめとする ERP など) の場合、統合と変換の方法は絶えず新しくなり、きわめて複雑化する可能性があります。こうした状況では、特定の統合プラットフォームの利用を検討します。たとえば、Windows Server や Windows Azure (IaaS) 上で実行できる Biztalk Server は、市販のパッケージ製品および堅牢なメッセージング インフラストラクチャとの接続を可能にする 25 以上のアダプターを備え、Windows Azure サービス バスとの接続機能も提供します。

まとめ

ビジネス アプリケーションに適用するアプローチ、テクノロジ、アーキテクチャ、およびパターンには、実に多種多様な組み合わせがあります。このガイドでは、主流なアプリケーションの優先事項に基づいて、多くのアプローチを紹介するよう努めていますが、ビジネス アプリケーションには膨大な数のコンテキストが存在するため、厳密かつ規範的なガイドとしてそのまま適用することはできません。このガイドには、論理的な例外が存在する可能性があります。たとえば、データ駆動性がそれほど高くなく、むしろドメイン ルールが常に変化するためにドメイン駆動性の高い小規模/中規模ビジネス アプリケーションもあれば、反対に、ミッション クリティカルな大規模アプリケーションの一部であっても、明らかにデータ駆動型アプローチが適している付帯的サブシステム (境界づけられたコンテキスト) もあります。

このガイドの主な目的は、アプリケーションおよびビジネスの優先事項から導き出される、さまざまなアプローチに対応したマイクロソフトのテクノロジを紹介することです。アプリケーションをどのように分類するかは、それぞれのアプリケーションのコンテキストおよび具体的なドメインに応じてまったく異なります。各種のアプローチおよびテクノロジの位置付けをイメージし、整理するために分類しているにすぎません。

カスタム アプリケーションの開発は多くの場合、科学よりも芸術に似ています。従うべき絶対的ルールなど存在しません。使用するアプローチ、手法、およびテクノロジは、目的のアプリケーションのコンテキスト、対処すべき実際のドメイン、および開発チームの置かれた環境とスキル セットに応じて決まります。

6. 付録 A – Silverlight の移行パス

前述したように、RIA コンテナーは、より多くのデバイスをサポートする手段 (HTML5) や、デバイスの統合性に優れた手段 (ネイティブ開発) に徐々に取って代わられつつあります。ただし、どの選択肢をどういった期間で実行するかは、現在の投資状況とニーズに応じてそれぞれ異なるでしょう。長期的に見ると、今後は HTML5 およびネイティブ アプリケーションの利用がさらに増え、RIA コンテナーは減少していくと考えられますが、移行方法や期間を最終的に決定するのは皆さん自身です。

現在 Silverlight を使用している場合は、都合のよい期間で段階的に移行を進めることができます。アプリケーションのコードの一部に、HTML5 および Silverlight を使用することも可能です。Silverlight はプラグインなので、HTML コンテンツの内部で使用できます。Silverlight は、ネイティブ アプリケーションの方が適しているシナリオの場合でも、デスクトップ アプリケーションの WPF や、Windows ストア アプリの .NET モデル、(このガイドで既に紹介したソリューションを併用する) マイクロソフト以外のサード パーティ製デバイスといった、.NET/XAML をサポートする各種プラットフォームとの優れた仲介役を務めます。最新バージョンの Silverlight は 10 年間サポートされるため、Silverlight が提供する成熟かつ安定したソリューションを引き続き使用しながら、移行を段階的に進めることが可能です。

主要アプリケーションにモノリシック アーキテクチャを採用しており、バックエンドがフロントエンドと緊密に統合されている場合は、システムを構成するこれらの 2 つの部分を分離し、複数のフロントエンド (HTML4、HTML5、Silverlight、ネイティブ モバイル) から共通のバックエンドに接続できるようにします。

また、アプリケーションによっては、依然として RIA コンテナー ソリューションでしか要件 (ビデオ コンテンツの保護など) を満たせないため、既存の投資を維持し続けた方がよい場合もあります。

幅広いシナリオや、多様なアプリケーションの種類、コンテキスト、および優先事項に対処するため、以下に推奨する移行パスを示します。

現在のアプリケーションの種類と Silverlight の使用	**アプリケーションのコンテキストと優先事項**	**推奨パス**
ブラウザー内で実行される、Silverlight ビデオ ストリーミング システムなどの Web メディア ソリューション	できるだけ広範なユーザーおよびマルチプラットフォーム デバイスに対応する 通常の HTML5 メディア機能で十分	HTML5 Web 開発アプローチに移行し、すべてのマルチベンダー デバイス (タブレット、コンピューター、スマートフォンなど) をサポートします。ただし、任意のデバイスをサポートすると、サポートする必要のあるメディアの形式とコーデックも変わります。 重要: 移行アプローチとしては、フォールバック アプローチが最適な選択だと考えられます。クライアントが Silverlight をサポートしている場合、Silverlight の現行の機能 (スムーズ ストリーミングや DRM) は HTML5 のメディア機能よりも強力です。さらに、一部の旧式ブラウザーでは、Silverlight はサポートされても、HTML5 はサポートされない場合があります。
ブラウザー内で実行される、プライベート ビデオの Silverlight スムーズ ストリーミング システムのような高度な Web メディア ソリューション	HTML5 のメディア機能でまだサポートされていない、3D や DVR などの機能を使用した高度なメディア ソリューションを実行する 任意の種類のモバイル デバイスでは	Silverlight 5 (10 年間のサポート付き) で現在のメディア アプリケーションを維持することを推奨します。 DRM が必要な場合も、Silverlight では引き続きサポートされます。

	なく、対象のほとんどが Windows ベースの PC または Mac OS X コンピューターから成る、制御された環境内のシステム	必要とする高度な機能のサポート状況に合わせ、HTML5 への段階的移行を計画します。
ブラウザー内で Silverlight を使用する、パブリック インターネット Web アプリケーション/サイト	できるだけ広範なユーザーおよびマルチプラットフォーム デバイスに対応する	HTML5 Web 開発アプローチに移行し、すべてのマルチベンダー デバイス (タブレット、コンピューター、スマートフォンなど) をサポートします。
ブラウザー内で Silverlight を使用する、プライベート/内部用ビジネス Web アプリケーション	絶えず成長、発展する重要なコア ビジネス アプリケーションではなく、中期的に使用する付帯的なアプリケーション 任意の種類のモバイル デバイスではなく、対象のほとんどが Windows ベースの PC または Mac OS X コンピューターから成る、制御された環境でアプリケーションを引き続き実行	Silverlight 5 (10 年間のサポート付き) で現在のアプリケーションを維持すると同時に、将来的な HTML5/JS への段階的移行について計画しておくことを推奨します。 使用するブラウザーは、デスクトップで実行される従来型のブラウザーでなくてはならない点を考慮に入れます。IE の "モダン ブラウザー" (Windows ストアアプリ仕様) は、Silverlight プラグインをサポートしません。
	何年にもわたって絶えず成長、発展する、重要なコア ビジネス Web アプリケーション テクノロジの刷新を図る新しいアプローチに基づいてアプリケーションを発展させると共に、特定のシナリオでは、いずれかの種類のモバイル デバイス (いずれかのタブレットなど) への対応も検討する	ASP.NET Web API サービスを利用して、コア ビジネス アプリケーションのプレゼンテーション層を、ASP.NET MVC、MVVM 対応の JavaScript ライブラリ (Knockout ライブラリ) などのライブラリに基づいた、ASP.NET SPA (Single Page Application) のような HTML5/JavaScript ベースの Web アプローチへと段階的に移行することをお勧めします。 現在の分散型サービス層は、多額のコストをかけずに Web API サービスに移行することも、再利用することも可能です。 使用するブラウザーは、デスクトップで実行される従来型のブラウザーでなくてはならない点を考慮に入れます。IE の "モダン ブラウザー" (Windows ストアアプリ仕様) は、Silverlight プラグインをサポートしません。
ブラウザー外実行で Silverlight を使用する、プライベート/内部用ビジネス デスクトップ アプリケーション	絶えず成長、発展する重要なコア ビジネス アプリケーションではなく、中期的に使用する付帯的なアプリケーション	Silverlight 5 (10 年間のサポート付き) で現在のアプリケーションを維持することを推奨します。 将来的には、段階的な移行を計画します。
	何年にもわたって絶えず成長、発展する、重要なコア ビジネス デスクトップ アプリケーション 従来型のデスクトップ アプローチを廃止し、タッチ シナリオが望ましい場合もある	現在のプレゼンテーション層を (短期的に) 維持しながら、タッチ シナリオが望ましい場合や新しいアプリケーション開発のアプローチが現在の MVVM による Silverlight/XAML アプリケーションときわめて似通っている場合などは、没入型な Windows ストア アプリ (C#/XAML) への移行計画を開始することをお勧めします。
	何年にもわたって絶えず成長、発展する、重要なコア ビジネス デスクトップ アプリケーション タッチ シナリオよりも従来型のデス	現在のプレゼンテーション層を (短期的に) 維持しながら、新しいコンテキストへの移行計画を開始することをお勧めします。これには、次のような選択肢が考えられます。

	クトップ シナリオが必要	- Web シナリオがすべての要件を満たす場合は、Web HTML5/JavaScript プレゼンテーション層アプローチへの移行を検討する - Silverlight の使用を止める場合は、Windows Presentation Foundation (WPF) に移行 (現在の分離アーキテクチャからのスムーズな移行が可能)。WPF アプローチでは、現在の MVVM による Silverlight/XAML アプリケーションとの高い類似性が期待できる

表 6-1

参考資料	
Silverlight または WPF XAML/コードの Windows ストア アプリへの移行	http://msdn.microsoft.com/library/windows/apps/br229571.aspx
Silverlight アプリケーションの Windows ストア アプリへの移行	http://blogs.msdn.com/b/win8devsupport/archive/2012/11/12/port-a-silverlight-application-to-windows-8.aspx (英語)
Silverlight のサポート ポリシー	http://support.microsoft.com/lifecycle/?LN=en-us&c2=12905

7. 付録 B –データ アクセス テクノロジの位置付け

マイクロソフトは何年にもわたり、より柔軟なデータ アクセス テクノロジの構築と発展に取り組んできました。きわめて専門性の高いものから汎用的なものまで、どのテクノロジにも共通した 2 つの目標があります。1 つは、アプリケーションが必要な情報にアクセスできるようにすること、もう 1 つは、データ ストレージの込み入った詳細事項への対応に、開発者ができるだけ時間を割かずに済むようにすることです。これにより、開発者はデータを利用するソフトウェアに注力できるため、ユーザーに真の価値がもたらされます。

データにアクセスするための適切なテクノロジを選択する際は、使用することになるデータ ソースの種類と、対象のデータをアプリケーション内で処理する方法を考慮します。テクノロジによっては、特定のシナリオに適合する場合もあります。以降のセクションで、考慮すべき主なテクノロジおよび特徴について説明します。

Entity Framework

Entity Framework (EF) は、リレーショナル データベースへのアクセスが必要な新しいアプリケーションに適した、マイクロソフトが推奨するデータ アクセス テクノロジです。EF はオブジェクト リレーショナル マッパー (O/RM) として、.NET 開発者がドメイン固有のオブジェクトを使用し、リレーショナル データを扱えるようにするための機能を提供します。EF を使用する場合には、通常なら開発者が記述しなくてはならないデータ アクセス コードのほとんどが必要ありません。EF でモデルを作成するには、コード (クラス) を記述する方法と、Entiry Data Model デザイナーのビジュアル ダイアグラムを使用する方法があります。いずれのアプローチも、既存のデータベースまたは新たに作成するデータベースに使用できます。

リレーショナル データベースとマッピングするエンティティ モデルを作成する場合には、EF の使用をお勧めします。通常、1 つのエンティティ クラスは、上位のレベルで、1 つのテーブルまたは複雑なエンティティを構成する複数のテーブルとマッピングされます。EF の最も優れたメリットは、多くの点で、連携するデータベースを意識せずに済むところです。これは、各データベース管理システム (DBMS) で必要となるネイティブ SQL ステートメントが EF モデルによって生成され、Microsoft SQL Server、Oracle、DB2、MySQL のいずれを使用しているかにかかわらず、透過的にデータにアクセスできるためです。DBMS ごとに対応する EF プロバイダーを変更する作業のみが必要となります (ほとんどの場合、接続文字列を変更して EF モデルを再生成するだけで完了です)。そのため、EF は、オブジェクト モデルをベースとしたオブジェクト ロ

ール モデリング (ORM) 開発モデルを使用し、柔軟な方法を通じてリレーショナル モデルへとマッピングする場合に適しています。また、EF は一般的に LINQ to Entities と併用されます。LINQ to Entities は、LINQ のようなオブジェクト指向の構文を使用し、厳密に型指定されたクエリをエンティティに対して実行する場合に適しています。

サード パーティの O/RM テクノロジ: マイクロソフトが提供およびサポートしているもの以外にも、優れたテクノロジ (NHibernate、LinqConnect、DataObjects.net、BLToolkit、OpenAccess、Subsonic といった O/RM など) が多数存在し、それらも効果的なアプローチとして使用できます。

ADO.NET: ADO.NET ベースのクラスの使用は、低い API レベルへのアクセスを必要とする場合に適しています。ADO.NET により、完全な制御性 (SQL ステートメント、データ接続など) が提供されますが、リレーショナル データベース管理システム (RDBMS) に対する EF の透過性は失われます。既存の反転パターンの再利用 (既存のストアド プロシージャや、ADO.NET を使用して実装された既存の Data Access Building Block の大規模な利用) が必要な場合も、ADO.NET を使用しなければならないことがあります。

Microsoft Sync Framework: 接続が途切れることのあるシナリオや、異なるデータベース間での連携が必要なシナリオをサポートするアプリケーションの設計に適しています。

LINQ to XML: アプリケーション内で XML ドキュメントを多用し、そのドキュメントを LINQ 構文を通じて照会する場合に適しています。

NoSQL データベースおよびテクノロジ: NoSQL は、広く使用されている RDBMS モデルに準拠していない DBMS の総称です。これに該当するデータ ストアでは、固定テーブル スキーマが必須ではありません。一般的には結合操作を行わず、データ操作に SQL 文を使用しません。

この種類のアーキテクチャは通常、水平的な拡張性や、取得操作および追加操作への最適化を重視する場合に適しており、多くの NoSQL システムは、レコード ストレージ (キー/バリュー型ストアなど) とほぼ同等の機能しか備えていません。フル機能の SQL システムと比べると実行時の柔軟性に劣りますが、特定のデータ モデルではその点を補うほどの優れた拡張性とパフォーマンスが実現されます。NoSQL データベースは、リレーショナル モデルを必要としない大量のデータを扱う場合に効果を発揮します。NoSQL は構造化データも扱えますが、要素間の関係を扱うことより、膨大なデータを格納および取得できることが最優先される状況に適しています。使用例としては、1 つまたは少数の連想配列内に、キーと値の数百万件に上るペアを格納する場合が挙げられます。

現在、マイクロソフトが NoSQL 向けに提供している主なデータ ソースは、Windows Azure テーブルと Windows Azure BLOB です。そのほかに、MongoDb、Casandra、RavenDb、CouchDB といったサード パーティ製の NoSQL データ ソースも使用できます。各種の NoSQL データベース/テクノロジは通常、長所と短所、使用方法、および API の点でそれぞれまったく異なるため (たとえば、これまで紹介した NoSQL データベース システムは、いずれもまったく性質が違う)、そのうちのどれを選択するかは、必要とするアプリケーションおよびデータ アクセスの種類によって決まります。これは、NoSQL がリレーショナル データベースと大きく異なる点です。リレーショナル データベースは、使用方法 (リレーショナル テーブルと結合) や API (SQL がベース) に関し、どれもきわめて似通っています。

NoSQL API: ほとんどの NoSQL 実装はそれぞれまったく異なるため、API の実装も個々に独立している場合が一般的です。ただし好都合なことに、NoSQL の多くには .NET API があり、Windows Azure のテーブルや BLOB のように、リモート アクセス API (REST サービスに基づくものなど) が提供される場合もあります。

ビッグ データ: ビッグ データとは、きわめて大量かつ複雑なデータ セットの集合体であり、従来型のデータ処理システムでは処理しきれなくなりつつあります。データの大容量化が進む背景には、関連データをひとまとめにした大容量データは、総量が等しい細切れのデータ セットに比べ、分析によって引き出せる情報量が多く、相関性を見つけて "ビジネス トレンドを特定" したり、調査の品質を判断したりできるという点があります。

現在のビッグ データのサイズは、単一のデータ セットで数十テラバイトから数ペタバイトにもなります。こうしたデータセットの処理は非常に煩雑な作業となるため、大量のデータを扱える "ビッグ データ" テクノロジの新しいプラットフォームが必要とされています。わかりやすい例を挙げるなら、Apache Hadoop ビッグ データ プラットフォームでしょう。

Hadoop は、膨大な量のデータ セットを一括して並列処理するために設計されたオープン ソース ライブラリです。Hadoop 分散ファイル システム (HDFS) を基盤とし、クラスター内に格納されたデータを処理するための各種ユーティリティおよびライブラリで構成されます。こうしたバッチ処理は、Map/Reduce ジョブを含む、さまざまなテクノロジを通じて実行します。

ビッグ データに関するマイクロソフトのエンドツーエンドのロードマップでは、エンタープライズ クラスの Hadoop ベース ソリューションを Windows Server 上と Windows Azure 上の両方に分散させることで、Apache Hadoop を取り入れています。こうしたマイクロソフトは、**HDInsight** というエンタープライズ向け Hadoop サービスを提供しています。ビッグ データに関するマイクロソフトのロードマップの詳細については、マイクロソフトのビッグ データのページ (英語) を参照してください。