Microsoft .NET

.NET Technology Guide for Business Applications

June 2013, Microsoft Corporation

Microsoft

Table of contents

1. Key takeaways

Select your architecture approaches and development technology based on your specific application's priorities and requirements.

A single architecture and approach won't work for every type of application. The Microsoft development stack and .NET are extremely flexible and offer many possibilities, but it's essential that you choose specific approaches and technologies based on the kind of application—or even subsystem—you build. Each application will have very different priorities and tradeoffs that must be taken on different paths.

Business application modernization goes further than simply building mobile apps. Mobile applications must rely on and extend your foundational business applications.

To be successful, mobile apps must be built with deep integration into your current foundational business applications. Mobile business apps should be part of the larger enterprise ecosystem and substantially extend foundational business applications, whether the foundational systems are established legacy applications or new, large, mission-critical applications built with innovative, scalable, and elastic services.

Positioning your application or subsystem within a global patterns segmentation will help you to choose the right approaches and technologies.

It is fundamental to position your application/subsystem in the right segmentation area. The right approaches and technologies for each of the following application types could potentially be very different:

- Emerging applications patterns
  - Devices and services
- Established applications patterns
  - Small and medium business-applications
  - Large, mission-critical business-applications

2. Purpose of this guide

This guide will help you effectively select the right Microsoft development technologies and approaches for your .NET custom application development, depending on the priorities you have for your application and for your business domain.

This guidance does not cover *Application Lifecycle Management* (ALM) practices. For additional guidance on this topic, you can visit the Visual Studio ALM website at www.microsoft.com/visualstudio/alm.

Who should use this guide

This guide will be useful to decision makers, software architects, development leads, and developers who are involved in selecting the technologies to use for their applications and projects based on Microsoft development platforms.

Specifically, it covers custom enterprise application development, although ISVs might also find the information and recommendations useful.

This paper does not cover development solutions based on Microsoft business-market products, such as vertical solutions based on Dynamics CRM or Dynamics ERP.

How to use this guide

This guide covers a broad spectrum of software development options that focus on business applications. It is written as a reference document, so you can go directly to an area you're interested in, such as "Emerging application patterns" or "Large mission-critical business-applications" within the "Established application patterns" section.

We do recommend that you read the Overview for context before you dive deeper into the individual sections.

3. Overview

Today, technology use is in the midst of a shift toward multi-device experiences powered by services in the cloud. Usage patterns are increasingly dependent on local hardware capabilities such as touch, sensors, and mobility, combined with the power of web connectivity and back-end services such as data storage, media streaming, and social connectivity.

The devices-services nexus spans both business and consumer scenarios. In the consumer space, mobile computing initially created a wave of devices focused on consumption, which continues to grow as hardware capabilities and technologies advance. Within the enterprise, the twin phenomena of the consumerization of IT and bring-your-own-device (BYOD) have created a dynamic in which consumer experiences are driving the future of business computing and line-of-business (LOB) applications.

The next generation of device- and service-dependent applications is not emerging in isolation. These applications have to work in an extremely well-integrated fashion with existing applications, unlocking their value to new audiences and new modes of interaction. This creates two different patterns that every application developer must now face:

- **Established application patterns:** These are applications developed using technology patterns such as client/server or web applications optimized for desktop browsers. They act as foundational applications and are heavily centered in existing **business processes.**
- **Emerging application patterns:** Patterns such as multi-devices and the cloud are emerging as technology enablers for new applications. They complement the established patterns by extending the applications to be centered on the **end user.**

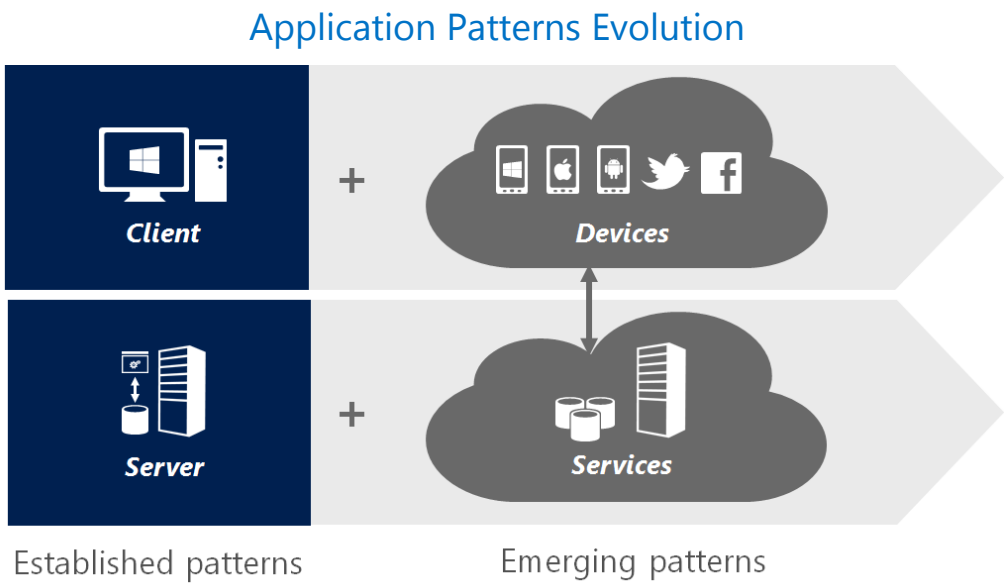


Figure 3-1

This extension of established patterns to meet the end user is a key opportunity for developers to drive new innovation and differentiation vs. competitors. Retail, communications, finances, logistics, customer services—every company is a software company in today's business world. Each company's ability to fulfill customer needs and compete effectively is only as good as their ability to deliver software innovation.

However, extending existing applications to embrace these new needs is a challenging transformation process. Current development technologies are deeply rooted in the established pattern and are difficult to integrate with the emerging patterns needed for modern software. Existing tools do not provide an obvious path from the existing client/server world to the emerging device/cloud world.

The Microsoft platform enables developers to address these challenges. It builds upon **existing applications**, extending them to emerging application patterns. It embraces **multiple development technologies**, so developers can choose the option that best fits their skills or the technologies used by their existing applications. For services development, Microsoft Windows Azure supports a multitude of technologies that any developer can use, such as Java, Node.js, PHP, Python, Ruby, and first-class support for .NET. Client development for the Microsoft platform also supports a broad range of technologies natively, such as .NET, HTML5/JavaScript, and C++.

This document focuses on .NET development, and specifically on business applications. It covers how to use .NET to develop for the established patterns that shape existing applications and also how to embrace the emerging patterns that are enabling the **modern business applications** of the future.

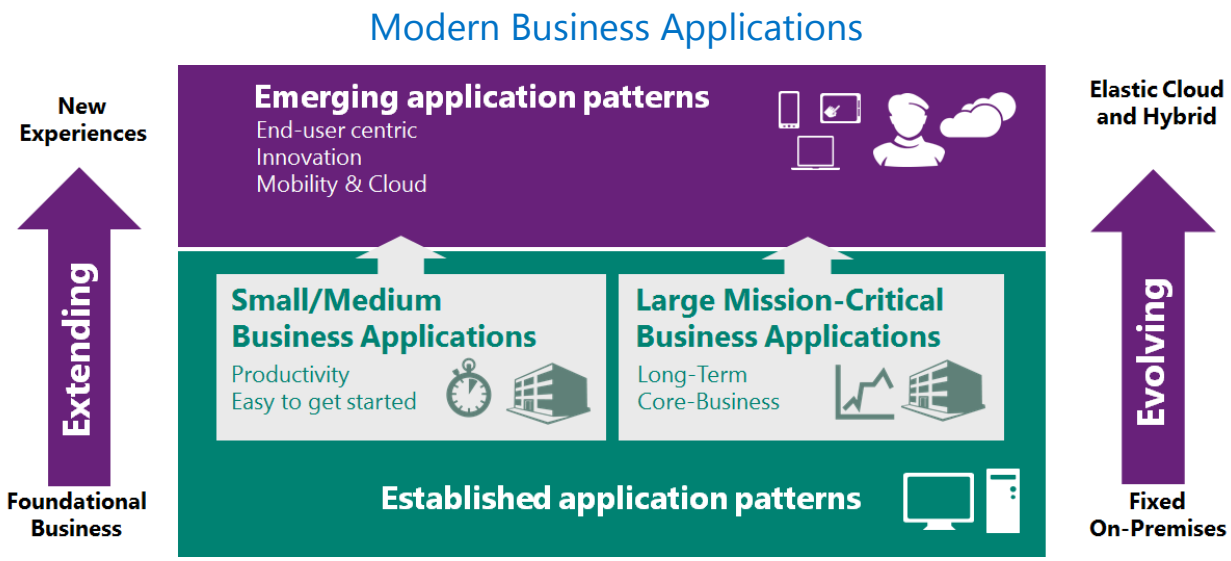


Figure 3-2

The .NET Framework and the future of development

The Microsoft .NET Framework was built to enable developers to create compelling applications on the Microsoft platform and, by all accounts, it has been a huge success in the market. Today, millions of developers across companies of all sizes and segments rely on .NET to create applications. It provides the core services required to build consumer applications, small business-applications, and large mission-critical applications, all with unprecedented quality, performance, and productivity.

.NET was also built with these now-emerging patterns in mind. At Forum 2000, Bill Gates said that the goal for .NET was *"to move beyond today's world of stand-alone websites to an Internet of interchangeable components where devices and services can be assembled into cohesive, user-driven experiences."* The original vision for .NET is remarkably well aligned with today's developer

landscape including cross-device, service-powered experiences that are changing how the industry thinks about software development.

Enabling multi-device experiences empowered by services was a key attribute for .NET from the beginning. .NET has kept evolving since then, providing a first-class development experience for the new needs of applications:

- **On the server side,** .NET provides a common platform for developers to target services that run on-premises or in the cloud. Its close integration with Windows Server and Windows Azure allows applications to be gradually extended to the cloud, taking the best of each platform and enabling hybrid applications that sit between the two worlds. The fast delivery cadence in the .NET Framework libraries also provides continuous innovation that addresses the new needs of cloud-based applications in areas such as lightweight services, real-time communications, mobile web applications, and authentication.
- **On the client side,** .NET provides a consistent, first-class development experience across all Microsoft devices—desktop experiences, Windows Phone apps, and Windows Store apps. It allows .NET developers to keep developing foundational applications on the desktop and add exciting new experiences, all while using their existing skills and reusing code between devices. For scenarios where the reach goes beyond Microsoft devices, HTML5 browser-based solutions are the norm. .NET, in conjunction with Visual Studio, provides a modern solution for creating standard-based web applications that run across various devices. For developers looking to create more tailored, native experiences on any device, Visual Studio industry partners (VSIP) provide solutions that enable reusing C# skills and code with non-Windows devices.

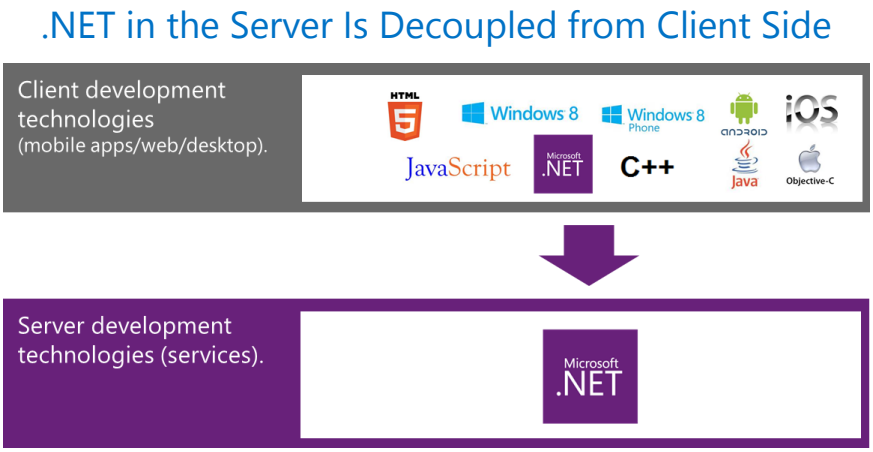


Figure 3-3

This document addresses all these .NET development options, so you can make the right decision for your current skills and application requirements as you move your applications forward. It is structured to address the two application patterns:

- "Emerging application patterns" focuses on how to build applications using the emerging patterns that are shaping the new applications across devices and services.
- "Established application patterns" covers the technologies available for creating foundational business applications, as well as recommendations on how to modernize them.

Figure 3-4 shows a global diagram of Microsoft development platform technologies. The upcoming sections will analyze and recommend when to use which technologies, depending on the application patterns and priorities.

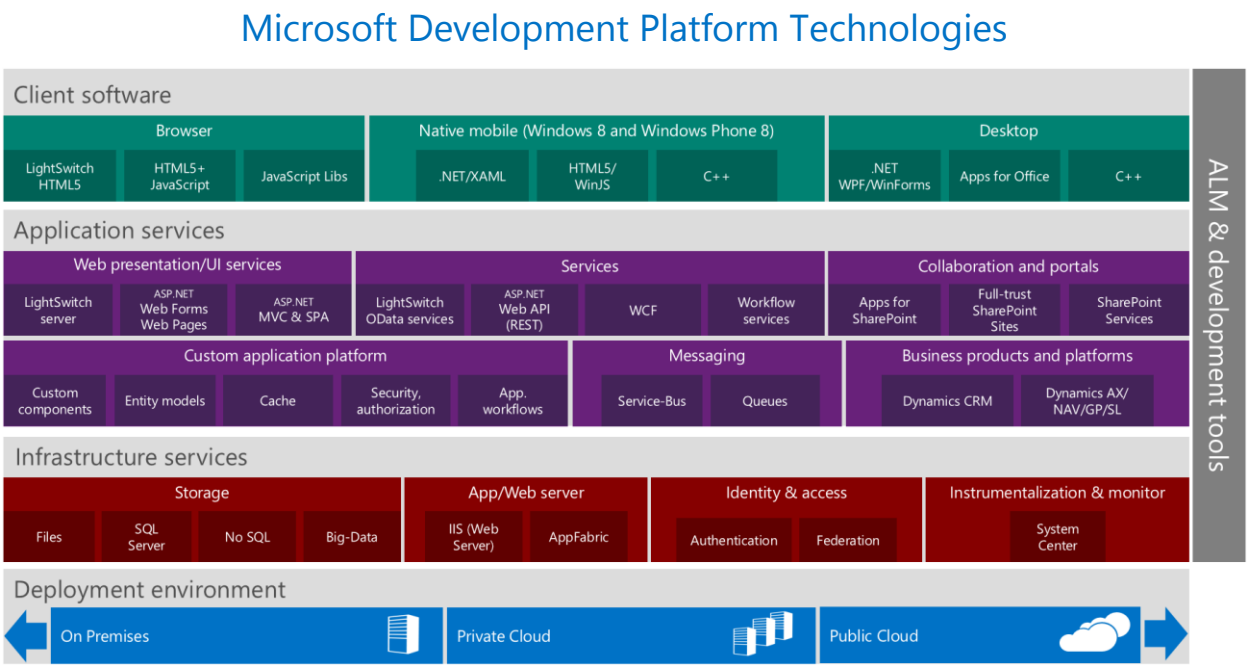


Figure 3-4

4. Emerging application patterns

As introduced before, emerging application patterns are shaping the applications of the future. Customers and employees now demand applications that deliver a more personal experience. They want to stay continuously connected to the services they need.

This section is structured in the two main components that need to be addressed when developing this new breed of applications:

- Creating experiences across heterogeneous devices.
- Creating standard, lightweight services that extend through the cloud.

The emerging applications patterns are, in many ways, comparable to the *"systems of engagement"* (term originally coined by Geoffrey Moore and also frequently used by Forrester), but additionally they must be supported by the cloud, and rely and extend upon actual *"systems of record"* (foundational business applications).

The term "systems of engagement" doesn't mean "applications targeting just consumers". In fact, there are many new scenarios in the enterprise (internal apps with mobility requirements like dashboards)—along with scenarios targeting end customers (such as online banking and e-commerce)—where this concept is perfectly valid.

Figure 4-1

Therefore, Microsoft's vision of emerging application patterns includes the necessity of continuous and elastic services, in addition to mobile needs and direct-to-consumer connection as shown in figure 4-2.

In most modern business applications, the client application will need a remote service in order to work with data and provide a centralized orchestration. This is where you need service orientation based on Internet standards for remote services. Finally, these services will be deployed in the public cloud infrastructure and services, which are the pillars for the emerging patterns in regards to deployment environments.

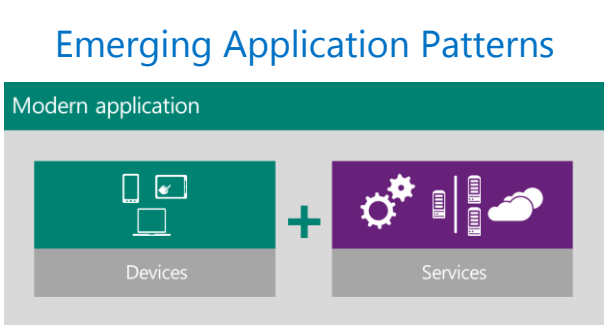


Figure 4-2

As we drill further into the emerging application patterns, you'll need to take into account all the possible mobile devices and social networks as a means for extending your business applications to new use cases and for building solid, continuous services that can flexibly scale to meet any demand.

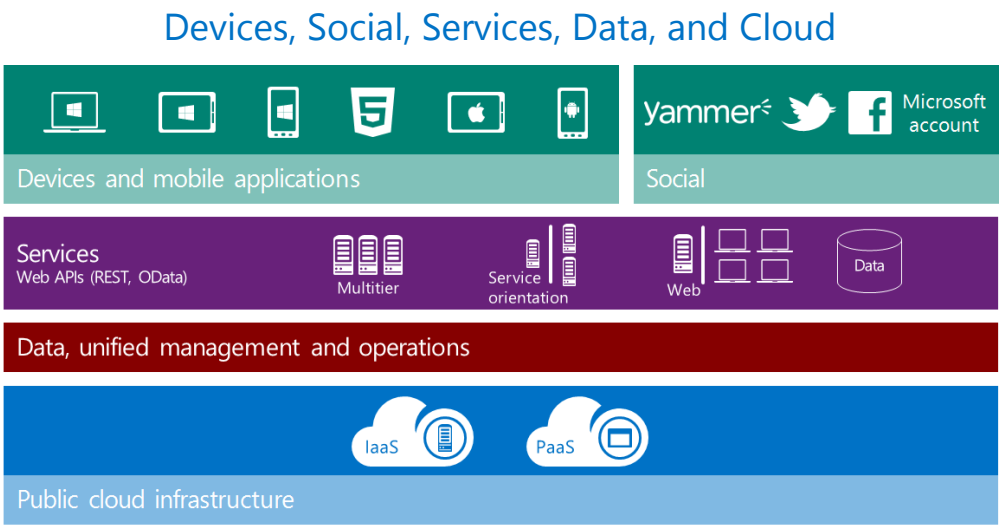


Figure 4-3

Devices

The ability to deliver new experiences tailored for devices is the key attribute for the emerging application patterns. Choosing the technology to create those applications can be difficult and involves many factors, including:

- Your previous skills and technology preference.
- The ability to create tailored experiences that integrate with local hardware capabilities.
- The diversity of devices your application will target.
- The technology used by your existing applications that need to be migrated or extended to devices.

The two alternatives commonly established in the industry are based on very different approaches:

- **Native applications,** which can get the most from every device but require unique skills and code for each platform.
- **Web applications,** which can be created with a common set of skills and code, but cannot provide a tailored experience for each device.

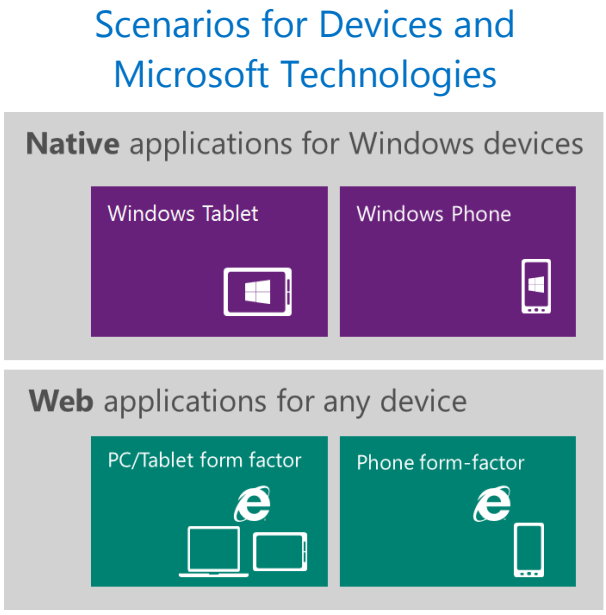


Figure 4-4

The Microsoft platform fully supports both approaches, but reduces the disadvantages of each significantly. First, Windows devices do not enforce a unique native development model. You can use the technology that makes the most sense for your skills and your existing applications. By bringing first-class device integration to .NET, HTML/JavaScript, and C++, you can make the decision that best fits your needs without compromising the experience. Second, .NET and Visual Studio greatly simplify creating web applications that can run across any device. ASP.NET fully embraces the modern standards and, in conjunction with the latest unique innovations in Visual Studio, enables a new breed of web applications that takes full advantage of modern browsers across devices.

Native applications for Windows devices

A native application is an application that runs on the client device and takes full advantage of that device's specific features in order to provide the most compelling experience for customers. As explained before, the Windows platform extends this concept to technologies beyond C++, which greatly expands the potential to reuse your code and skills to target new form factors.

The following tables explain the differences between these technologies and when they are appropriate to use, depending on your application priorities and concrete context.

Figure 4-5

UI development technologies for *native* Windows 8 Store applications (Windows runtime/WinRT)

Technologies	When to use and why
.NET/XAML	▪ Appropriate if you are already familiar with .NET and XAML or when extending existing .NET/XAML applications. ▪ It also allows you to share .NET Framework code and libraries between Windows Store apps, Windows Phone apps, Windows desktop apps, and other Microsoft platforms by using portable class libraries. ▪ Take advantage of open source .NET libraries such as sqlite-net for local light SQL database engines, or SignalR .NET client for bi-directional communication between the client and server.
JavaScript/ HTML5	▪ Appropriate if you are already familiar with HTML and JavaScript technologies or you are creating a Windows Store tailored experience for your existing web application. ▪ It allows you to reuse custom JavaScript or HTML/CSS assets from existing browser-based web applications and use the new WinJS library to get access to the native capabilities in the Windows Store apps/API. ▪ Take advantage of open source JavaScript libraries such as SQLite3-WinRT for local light SQL database engines, or SignalR JavaScript client for bi-directional communication between the client and server.
C++	▪ Appropriate if you are already familiar with C++. It lets you optimize and achieve the best performance for graphics-intensive apps or games. ▪ This option also allows you to share C++ code between Windows, Windows Phone, and other platforms, as well as target Direct3D for low-level graphics access. ▪ Take advantage of open source C/C++ code such as SQLite.

Table 4-1

References

Maximize code reuse between Windows Phone 8 and Windows 8	http://msdn.microsoft.com/en-us/library/windowsphone/develop/jj681693
Prism for Windows Runtime from Microsoft Patterns & Practices	http://prismwindowsruntime.codeplex.com/
ASP.NET SignalR: Incredibly simple real-time web for .NET	http://signalr.net/
SQLite: System.Data.SQLite	http://system.data.sqlite.org/index.html/doc/trunk/www/downloads.wiki
Visual C++ and WinRT: Some fundamentals	http://www.codeproject.com/Articles/262151/Visual-Cplusplus-and-WinRT-Metro-Some-fundamentals

UI development technologies for *native* Windows Phone 8 applications

Technologies	When to use and why
.NET/XAML	▪ Appropriate if you are already familiar with .NET and XAML or you are extending existing .NET/XAML applications. ▪ It allows you to share .NET Framework code and libraries between Windows Phone, Windows Store, Windows desktop, and other Microsoft platforms by using portable class libraries. ▪ Take advantage of open source .NET libraries, such as sqlite-net-wp8.
C++	▪ Appropriate if you are already familiar with C++. It lets you optimize and achieve the best performance for graphics-intensive apps or games. ▪ This option also allows you to share C++ code between Windows Phone, Windows Store, and other platforms, as well as target Direct3D for low-level graphics access in those platforms. ▪ Take advantage of open source C/C++ code, such as SQLite.

Table 4-2

References

Windows Phone 8 developer platform highlights	http://blogs.windows.com/windows_phone/b/wpdev/archive/2012/11/05/windows-phone-8-developer-platform-highlights.aspx
Windows Phone 8 and Windows 8 app development	http://msdn.microsoft.com/en-us/library/windowsphone/develop/jj714089(v=vs.105).aspx
Maximize code reuse between Windows Phone 8 and Windows 8	http://msdn.microsoft.com/en-us/library/windowsphone/develop/jj681693
Direct3D app development for Windows Phone 8	http://msdn.microsoft.com/en-US/library/windowsphone/develop/jj207052(v=vs.105).aspx

Web applications for any device

Microsoft provides best-of-breed tools and technologies for creating browser-based HTML5 applications that run on any device. Because Windows also supports HTML5 as a native technology, you can target multiple devices with HTML5 web apps, then reuse and optimize that code for a native Windows Store app.

The automatic switch for different HTML rendering or sizing can be achieved through different mechanisms provided by ASP.NET, JavaScript, and LightSwitch, as well as libraries such as Modernizr, which will detect the browser and adapt your HTML/JavaScript code, as shown below in Figure 4-6.

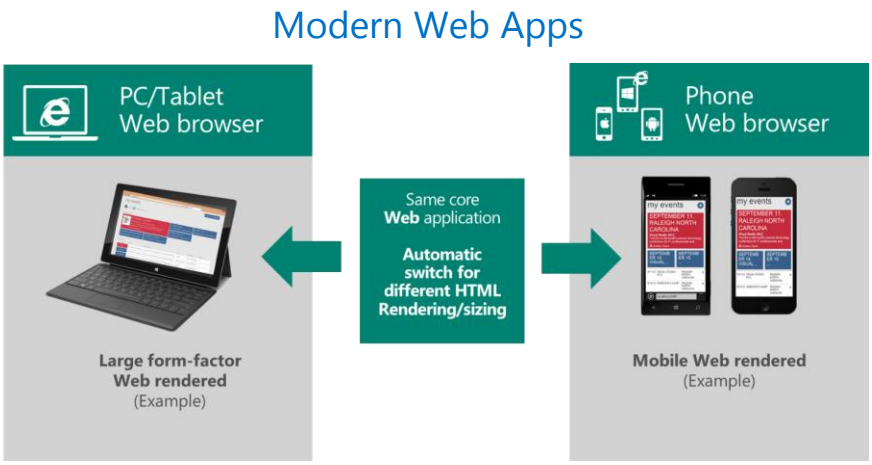


Figure 4-6

This approach is particularly advantageous when targeting different operating systems (such as Windows, iOS, and Android). After attaining compatibility with the most common web browsers, your application will be cross-platform compatible with the major mobile operating systems.

The next technology table enumerates the breadth of web-development technologies available, and recommends when to use each technology, depending on your application priorities and context.

UI development technologies for *web* applications

Technologies	When to use and why
ASP.NET **MVC** with HTML5 support	**ASP.NET MVC with HTML5 support** ▪ Use for flexible modern and mobile web apps, cross-browser compatibility, mobile web clients that can run on any modern device, and to take advantage of the ASP.NET MVC mobile features (such as using different pages and rendering based on detection of the current browser user agent). ▪ Offers a full HTML rendering control, better unit testing support, and faking capabilities. ▪ HTML5: Use libraries like *Modernizr* for detecting HTML5 features support and workarounds. ▪ Use CSS3 for less script and more maintainable code. ▪ Use JavaScript and jQuery for client-side programming. ▪ MVC allows Search Engine Optimizations (SEO) customizations. ▪ RESTful URLs are an integral part of MVC.
ASP.NET **SPA**	**Single Page Application (SPA)** based on **ASP.NET MVC, HTML5/JavaScript, Knockout,** and **Breeze** ▪ SPA is not a framework but a design pattern that can be implemented by using ASP.NET MVC and heavy use of JavaScript libraries like *Knockout* (for supporting the MVVM pattern within JavaScript) and like the *Breeze* JavaScript library for advanced data management and JavaScript frameworks like *Durandal*. ▪ Use it for highly interactive web applications and smooth UI updates (minimum server page reload and visible round trips) and when including significant client-side interactions using HTML5, CSS3, and JavaScript. ▪ Take advantage of the open-source *SignalR* JavaScript client library for bi-directional communication between the client and server. ▪ Consume ASP.NET Web API services from JavaScript.
HTML5 client for **LightSwitch** projects	**LightSwitch HTML5 client** ▪ Use if your web application or module is mostly data-driven (CRUD). ▪ The LightSwitch HTML5 client is the easiest way to create data-centric, cross-browser, and mobile web applications that can run on any modern device, take advantage of automatic HTML rendering, and adapt to different form factors.

	▪ Take advantage of CSS3, JavaScript, and OSS JavaScript libraries like jQuery Mobile and themes. ▪ Coupled to the end-to-end LightSwitch runtime server engine which is built on top of ASP.NET.
ASP.NET **Web Pages**	**ASP.NET Web Pages** ▪ ASP.NET Web Pages and the Razor syntax provide a fast, approachable, and lightweight way to combine server code with HTML to create dynamic web content.

Table 4-3

References

ASP.NET MVC mobile features	http://www.asp.net/mvc/tutorials/mvc-4/aspnet-mvc-4-mobile-features
ASP.NET SPA (Single Page Application)	http://www.asp.net/single-page-application
Breeze/Knockout template	http://www.asp.net/single-page-application/overview/templates/breezeknockout-template
Durandal SPA framework	http://durandaljs.com/
RequireJS library	http://requirejs.org/
BootStrap	http://twitter.github.io/bootstrap
Modernizr: the feature detection library for HTML5/CSS3	http://www.modernizr.com
LightSwitch Architecture	http://msdn.microsoft.com/en-us/vstudio/gg491708
Enhance Your LightSwitch Applications with OData	http://blogs.msdn.com/b/bethmassi/archive/2012/03/06/enhance-your-lightswitch-applications-with-odata.aspx
HTML Client Screens for LightSwitch Apps	http://msdn.microsoft.com/en-us/library/vstudio/jj674623.aspx

Services

The process of targeting multiple devices starts on the back end. Applications need to expose services that can be consumed on any device and scaled for the Internet.

Internet services are meant to have **high availability and continuity** with the back-end services that allow modern applications to work. They must also be agile and have a continuous, smooth evolution, adhering to the velocity of business change.

Within this context, **the ".NET ecosystem" and frameworks are essential when building services.**

As shown in figure 4-7, the .NET Framework supports a full range of approaches for building applications: ASP.NET approaches for web development (Single-page application [SPA], Model-view-controller [MVC], and WebForms); ASP.NET Web API for building HTTP/REST services; and using the entity framework to access data in relational databases. **Most server-side development is usually based on a .NET technology.**

The following technologies cover the different approaches you can take to build services:

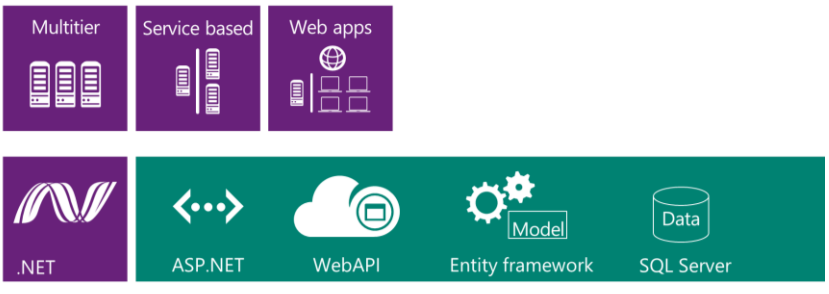


Figure 4-7

Back-end service technologies

(To be consumed by native or web applications)

Technologies	When to use and why
ASP.NET **Web API**	**HTTP based, REST approach, resource oriented, focus on OData and JSON** ▪ It is the preferred technology for flexible service development with **REST** approaches, **OData**, or **JSON** requirements. Try to use Web API as your first choice when evaluating which technology to use. Use any of the other technologies if Web API does not meet your needs. ▪ Especially made for REST services. Embraces HTTP verbs (PUT, GET, POST, DELETE...) as the application drivers. Resource oriented. ▪ High scalability thanks to Internet caches (Akamai, Windows Azure CDN, Level3, and so on) based on HTTP verbs.
Windows Communication Foundation (WCF)	**Decouple and flexible approach** ▪ Use it when you need SOAP interoperability or you want to use a non-HTTP transport. ▪ WCF can use any protocol (such as HTTP, TCP, or named pipes), data formats (such as SOAP, binary, JSON, or others), and hosting processes. ▪ This technology works best for RPC-style (task/command-oriented) services and for enterprise inter-application communications. REST is possible but not preferred here. ▪ Good fit to use with the Microsoft Service-Bus (in Windows Azure or Windows Server) and AppFabric hosting and monitoring.
WCF Data Services	**Data-driven services** ▪ Use when your services are data/resource-oriented and mostly CRUD and Data-Driven . ▪ It only supports **OData**. It is straightforward to use, but offers less flexibility and control than using ASP.NET Web API. ▪ Shares the same OData core libraries with ASP.NET Web API.
Workflow Services	**Workflow-based approach for building services** ▪ Use if your service logic is internally a Windows Workflow Foundation (WF) workflow. Externally, this is really a WCF service. ▪ Workflow Services has all the benefits and characteristics of WCF and WF, but is coupled to WCF.
SignalR	**ASP.NET SignalR library** ▪ Use for real-time functionality on the client side. ▪ This approach enables your server-side code to push content to connected clients in real time and at high scale, even to millions of users. ▪ It is HTTP and WebSockets based. It can be consumed from JavaScript in browser clients, .NET native Windows clients and server side events, and long polling.
LightSwitch OData Services	**OData and REST approach, resource oriented, easy way to build data-driven services** ▪ Use it if your client app is a LightSwitch app or for simple standalone and data-driven OData services. Visually model your data and LightSwitch will automatically create the services. ▪ It is OData-based. Services consumption client libraries are the same. ▪ Coupled to LightSwitch server engine, OData (Open Restful Data Protocol), XML, JSON, and HTTP. ▪ Less flexible than Web API and WCF, but requires no coding.

Table 4-4

References

ASP.NET Web API	http://www.asp.net/web-api
WCF	http://msdn.microsoft.com/en-us/library/ms731082.aspx
WCF Data Services Tools for Windows Store Apps	http://www.microsoft.com/en-us/download/details.aspx?id=30714
Creating and Consuming LightSwitch OData Services	http://blogs.msdn.com/b/bethmassi/archive/2012/03/09/creating-and-consuming-lightswitch-odata-services.aspx
ASP.NET SignalR	http://signalr.net/

Cloud and hybrid-cloud

Modern business applications typically support many Internet users (such as end customers and partners), so maintaining your back-end services within your company's internal datacenters may not make sense. For this reason, services are likely to be deployed in the cloud. Services also benefit from the core capabilities of the cloud, such as elasticity and a quick and cost-efficient production setup.

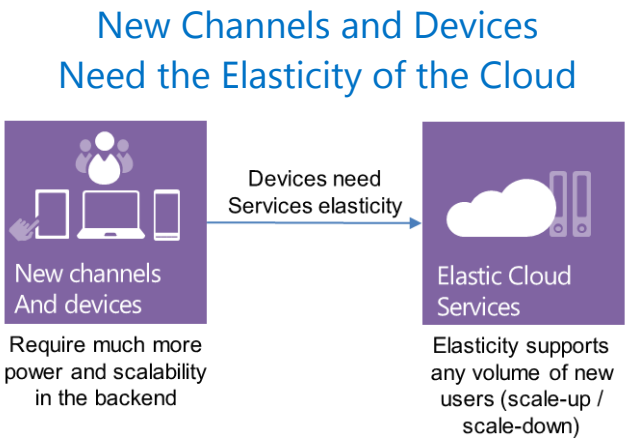


Figure 4-8

Additionally, many modern business applications are the result of new ideas, new channels, and new opportunities. You never know if you'll attract a significant number of new users, thus demanding more resources than what's available in your internal datacenters. The cloud is agile and scalable, so you can develop new concepts and quickly move them to production without hardware investments or setups. Therefore, elastic platform clouds, like Windows Azure, are the best options for modern business applications.

In a diverse ecosystem where business applications live in different environments (both in the cloud and on-premises), there are many new needs to meet. Because you are also extending foundational applications that are currently on-premises, you need a link between private datacenters and the cloud, and you need to link those worlds in a secure way through a hybrid-cloud approach, as shown in Figure 4-9.

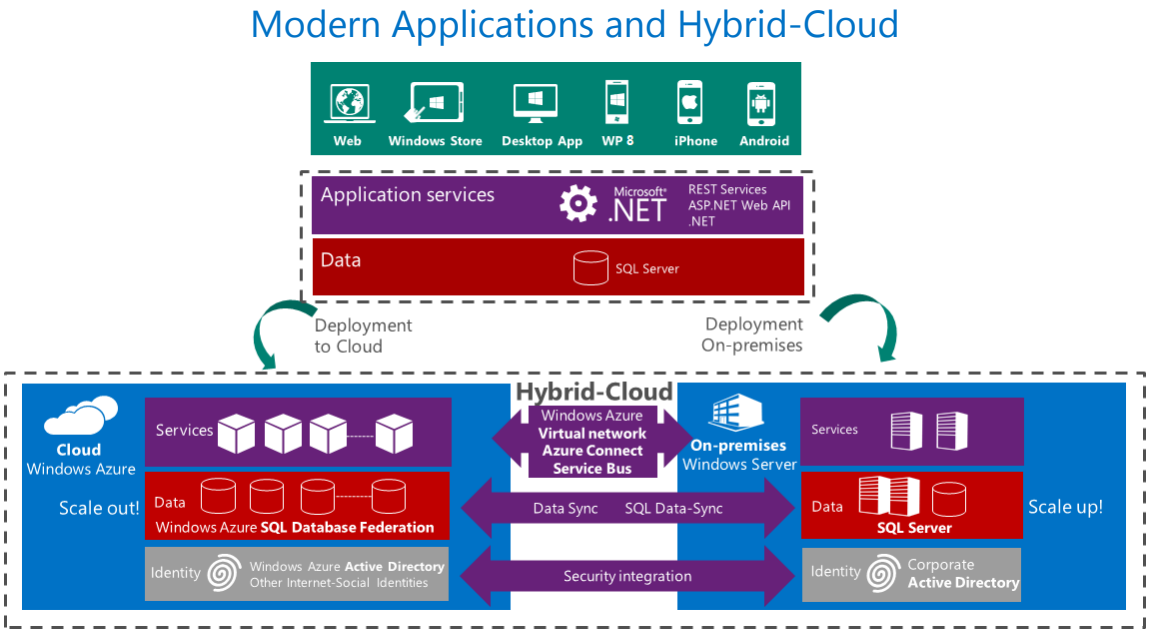


Figure 4-9

The Microsoft cloud platform offers symmetrical architecture, technologies, and products to support the cloud and on-premises infrastructures—and provides common services to manage applications that span across the two infrastructures in hybrid environments. This platform enables you to migrate and build your applications easily and gradually.

As shown in figure 4-10, Microsoft provides one consistent platform, whether your application is targeting the cloud or on-premises infrastructure. For development practices, it is best to use the same development platform (e.g. Visual Studio or .NET) for both cloud and on-premises environments. Similarly, having a single system to control and manage infrastructure (such as System Center) for any on-premises or cloud system is fundamental for efficient IT governance.

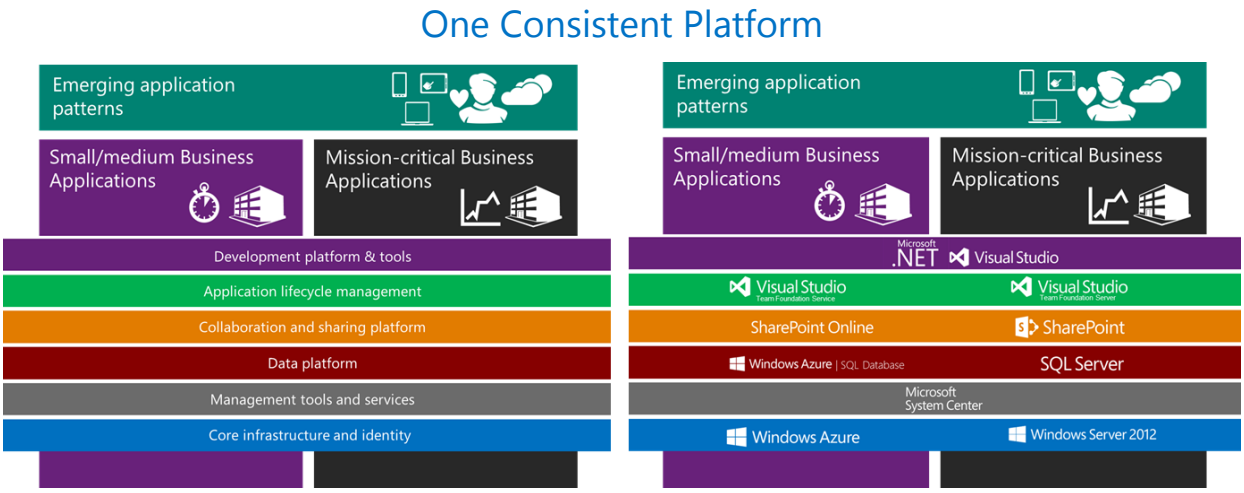


Figure 4-10

References	
Microsoft Public Cloud	http://www.microsoft.com/en-us/server-cloud/public-cloud/default.aspx
Microsoft Advances the Cloud OS with New Management Solutions	http://www.microsoft.com/en-us/news/press/2013/jan13/01-15OSMomentPR.aspx
Hybrid Cloud	http://www.microsoft.com/enterprise/en-nz/solutions/hybrid-cloud.aspx#fbid=CmzuvecYY9J
Cloud OS Vision	http://www.microsoft.com/en-us/server-cloud/cloud-os/

The next technology tables expose the main Windows Azure technologies related to application development.

Windows Azure cloud technologies

Technologies	When to use and why
Execution Models	**Execution models for web applications and services** Windows Azure supports multiple execution models that you can choose depending on your needs. ▪ **Infrastructure Services (Infrastructure as a Service/IaaS):** Use it when you need a traditional and flexible approach, where you are responsible for the internal Virtual machine infrastructure, and software maintenance, and administration (operating system, services, etc.). It supports traditional installations of software, like full SQL Server. ▪ **Web sites** (web hosting): Use it as an easy way to get started by simply deploying web applications on a managed IIS web site environment, and no infrastructure administration work is needed. It provides a low-cost, initially scalable, and broadly useful platform. ▪ **Cloud Services** or **Platform as a Service (PaaS):** It follows the same idea of the websites, but with a much more scalable and flexible platform suited for higher quality-of-service requirements and greater control. It also handles most of the work required for reliability and administration as PaaS. ▪ **Windows Azure Mobile Services:** It provides a scalable cloud back end, adding structured storage, user authentication, push notifications, and back-end jobs and services to your Windows Store, Windows Phone, Apple iOS, Android, and HTML/JavaScript applications.

Data Management	**Data sources in the cloud and tools** ▪ *Windows Azure SQL Database* for rich and relational database usage and a high parity with SQL Server on-premises, making it easy to move on-premises SQL Server databases to the cloud or to synchronize the different environments. Its biggest benefit is high availability out of the box and great simplification of the maintenance/administration, as it is delegated to the Windows Azure infrastructure. ▪ *SQL Server in Windows Azure Virtual Machines* is a specific scenario within the Windows Azure Infrastructure Services. For applications that need full SQL Server functionality, Virtual Machines is an ideal solution. ▪ *Windows Azure Tables* based on simple unstructured data with a NoSQL approach, suitable for very highly scalable data source requirements. ▪ *Windows Azure Blobs,* designed to store unstructured binary data, like video, files, or backup data or other binary information.
Business Analytics	**Big Data and reporting** ▪ *SQL Reporting* for functions similar to SQL Reporting Services when you need a platform to create reports. ▪ *Hadoop on Windows Azure,* which enables Big Data to be hosted as PaaS within Windows Azure.
Networking	**Hybrid-cloud and networking features** ▪ *Windows Azure Virtual Network* to connect your own on-premises local network to a defined set of Windows Azure VMs—a similar approach to VPNs, but oriented to servers and sub-networks. ▪ If your Windows Azure application is running in multiple datacenters, you can use *Windows Azure Traffic Manager* to intelligently route requests from users across multiple instances of the application. ▪ *Windows Azure Connect* lets you configure IPsec-protected connections (1:1 relationships) between certain computers or VMs in your organization's network and roles running in Windows Azure.
Messaging	**Messaging and asynchronous communication** ▪ *Queues,* suitable for asynchronous communication between different applications or processes, accessing the same persistent queues. ▪ *Service Bus* supports persistent messaging too, but it is suitable for more advanced scenarios like event-driven applications and integrations, publish/subscription patterns, and to easily access on-premises datacenters from the cloud through the relay-feature in Service Bus (which is also useful for peer-to-peer applications with firewalls in between). On-premises parity is available with Windows Server Service Bus.
Caching	**Internal application data caching and Internet HTTP caching** ▪ *Windows Azure Cache* is an in-memory distributed cache. It can be used as an external service or sharing a deployment between your own virtual machines. ▪ *Windows Azure CDN* is an "Internet cache" capable of caching HTTP-accessed data, like video, files, etc.
Identity	**Identity management services** ▪ *Windows Azure Active Directory* (AD) integrates your on-premises identity-management services for single sign-on across your cloud applications. It is a modern, REST-based service that provides identity management and access control capabilities for your cloud applications. Now you have one identity service across Windows Azure, Microsoft Office 365, Dynamics CRM Online, Windows Intune, and other third-party cloud services. ▪ *Windows Azure AD Access Control Service* provides an easy way to authenticate users who need to access your web applications and services without having to factor complex authentication logic into your code. It supports Windows Identity Foundation (WIF); web identity providers (IPs), including Windows Live ID, Google, Yahoo, and Facebook; Active Directory Federation Services (AD FS) 2.0; and Open Data Protocol (OData). ▪ *Active Directory Federation Services (AD FS) (Windows Server)* simplifies access to on-premises datacenters, systems, and applications using a claims-based access (CBA) authorization mechanism to maintain application security. AD FS supports web single-sign-on (SSO) technologies that help information technology (IT) organizations collaborate across organizational boundaries. AD FS on-premises can be integrated/extended through Windows Azure AD and ACS.
HPC	**High-Performance Computing (HPC)** ▪ It allows you to run many virtual machines simultaneously, all working in parallel on certain tasks. Windows Azure provides the HPC Scheduler in order to distribute the work across these instances. This component can work with HPC applications built to use the industry-standard Message Passing Interface (MPI). The goal is to make it easier to build HPC applications running in the cloud.
Media	**Windows Azure Media Services** ▪ It provides a set of cloud components for media ingest, encoding, content protection, ad insertion, streaming, etc. that greatly simplify the process of creating and running applications using video and other media.

Table 4-5

Windows Azure and Cloud References	
Building Hybrid Applications in the Cloud on Windows Azure (Microsoft Patterns & Practices guide)	http://msdn.microsoft.com/en-us/library/hh871440.aspx
Moving Applications to the Cloud (Microsoft Patterns & Practices guide)	http://msdn.microsoft.com/en-us/library/ff728592.aspx
Claims-Based Identity and Access Control (Microsoft Patterns & Practices guide)	http://msdn.microsoft.com/en-us/library/ff423674.aspx
Developing Multi-Tenant Applications for the Cloud, 3rd Edition (Microsoft Patterns & Practices guide)	http://msdn.microsoft.com/en-us/library/ff966499.aspx
Moving Your Applications to Windows Azure (Microsoft Patterns & Practices guide)	http://msdn.microsoft.com/en-us/magazine/jj991979.aspx
Enterprise Library 5.0 Integration Pack for Windows Azure (Microsoft Patterns & Practices guide)	http://msdn.microsoft.com/en-us/library/hh680918.aspx
Windows Azure cloud	http://www.windowsazure.com/

End-to-end scenarios in emerging application patterns

Scenario: Connected Native Windows Store Applications

Using the map of technologies introduced in Section 3, figure 4-11 below shows a typical scenario and its technology connection pattern for a modern native Windows app (including both tablet and mobile phone form factors).

Scenario: Connected Native Windows Store Apps

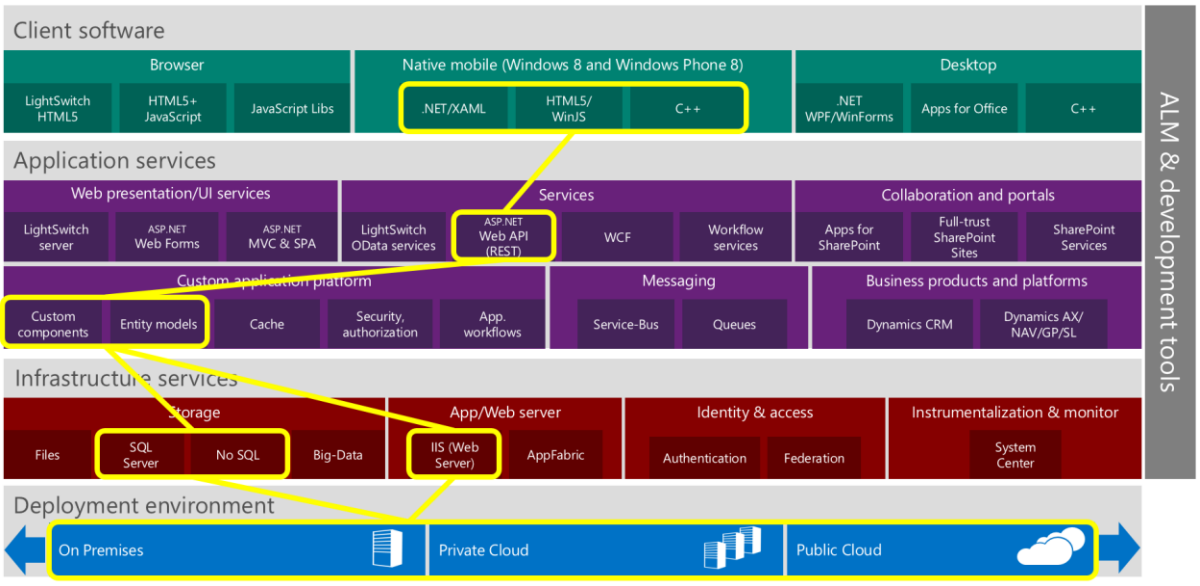


Figure 4-11

This scenario would be similar whether our UI client application is a native Windows Store app (e.g. a Windows Tablet) or a native Windows Phone app. Depending on your development skills and application requirements and priorities, we could choose between .**NET/XAML, HTML5/WinJS**, or **C++,** although only .NET and C++ cover both Windows Store and Windows Phone native app development.

For custom services development situated at the back end of modern apps, the recommended technology for this scenario is **ASP.NET Web API**. This provides a high degree of flexibility while having a light and performing framework for Internet services, usually heading approaches like REST and OData or JSON.

Then, the ASP.NET Web API services would wrap middle-tier logic in the form of .NET custom-class libraries and entity models (using entity framework if accessing relational databases, or any other API if accessing NoSQL data sources).

Finally, we could deploy our services and server components in any deployment environment, though in emerging application patterns, we would usually deploy it into a public cloud like Windows Azure.

As stated previously, a typical characteristic of modern applications is about **having different apps for different contextual/personal scenarios** (as shown in Figure 4-12). In most cases, these scenarios will have different priorities and requirements, which will drive your technologies decision. For instance, when developing a Windows Store app, for a concrete scenario quite close to Internet feeds and social networks APIs or even when you want to reuse skills and JavaScript code, HTML5/WinJS development would be the best choice. In other cases, if you want to reuse your C# or XAML skills while having good performance, you might use the .NET/XAML approach. And sometimes having the best possible graphics performance is a priority, in which case C++ would be the right choice.

Developing for Windows Phone takes similar recommendations into account, whether you choose .NET or C++.

Another important point is that back-end services can be the same for—and be re-used by—all the different clients/scenarios you choose. At a minimum, you would share the same tier and server technologies. In other cases, you could have different data models, services, and subsystems optimized for each client app.

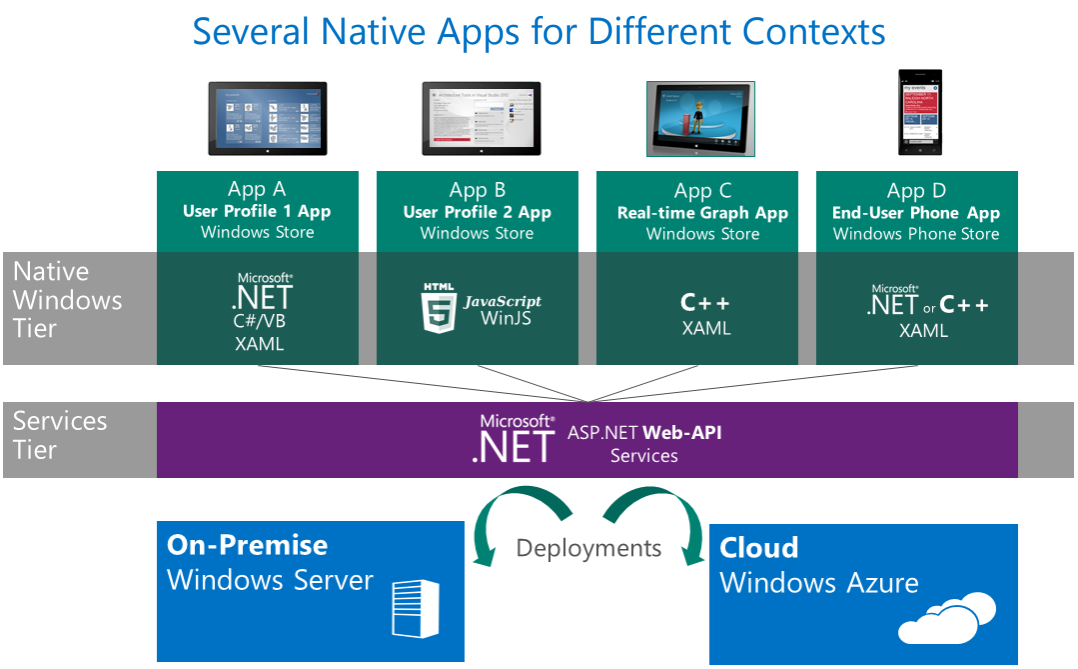


Figure 4-12

Scenario: Modern Web Applications for Any Mobile Device (Tablets and Phone)

Figure 4-13 shows a typical modern web application scenario and its technology connection pattern when targeting any form factor and any client platform (Windows or non-Microsoft OS).

For modern web application development supporting mobile devices, the recommended technologies are **ASP.NET MVC** with HTML5 support for flexible and full-control web development, and **HTML5 client for LightSwitch** projects when dealing with simpler data-driven scenarios.

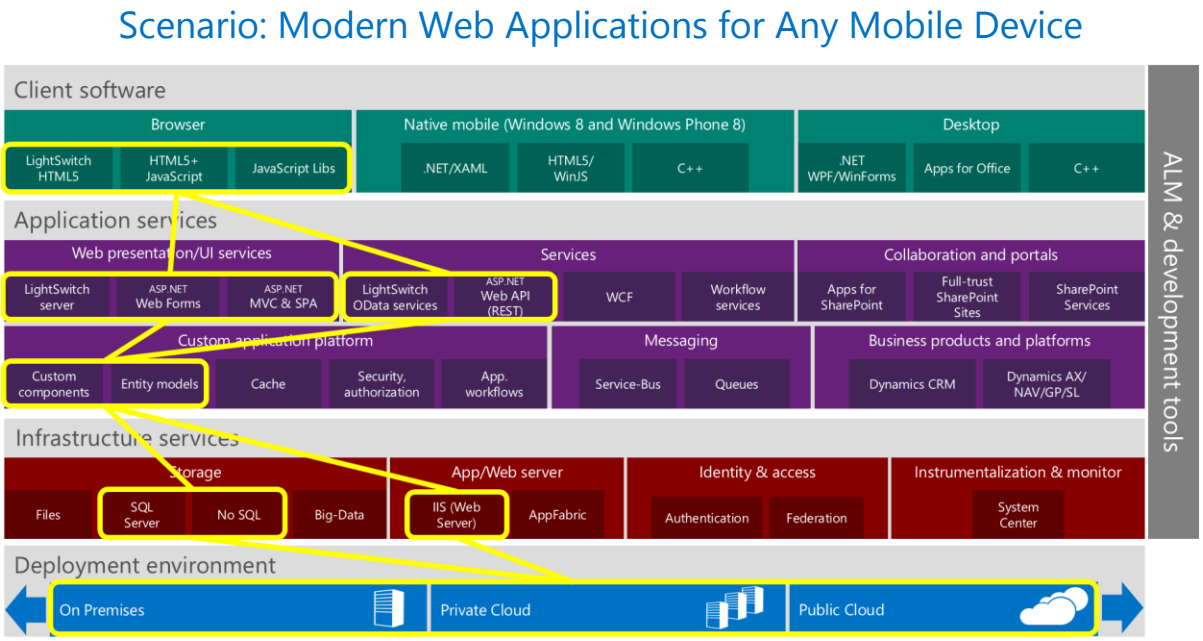


Figure 4-13

Simplified sample architecture for modern web applications targeting any web browser and operating system platform

Depending on the requirements and priorities of the web app, you would use different approaches. For highly data-driven apps, using LightSwitch is recommended, while more complex scenarios call should use finer-grain technologies like ASP.NET MVC and plain HTML5/JS.

You might have several modules with different priorities and characteristics (data-driven vs. more complex UI webs) within the same application. In such cases, you will likely adopt a mixed approach, as illustrated in Figure 4-14.

For this scenario, because you are using standard web technologies, you could be targeting any device and any operating system, not just a subset of mobile devices.

If the application is targeting different form-factors (phone and tablet), you should consider if each form factor would benefit from a different layout or view.

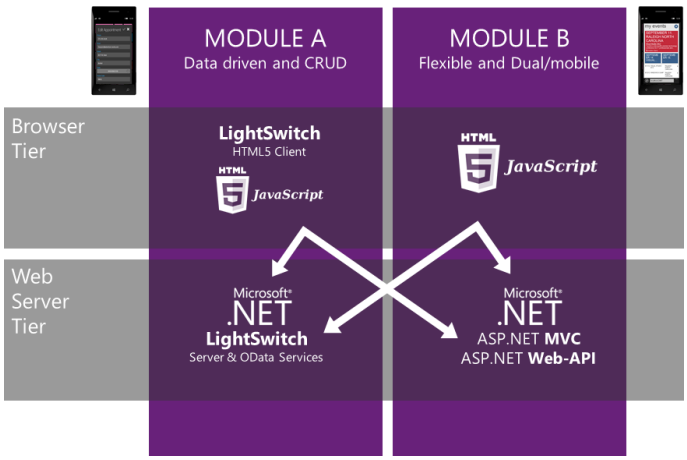


Figure 4-14

In figure 4-15, the server rendering services (whether they are ASP.NET MVC or LightSwitch) is capable of rendering HTML code to support multi-device alternatives. HTML can be rendered for each browser based on the web user agent. In the case of ASP.NET MVC, different files must be created for each user agent, because it is lower-level crafting. With LightSwitch, the different renderings are generated automatically, which dramatically simplifies multi-channel application development.

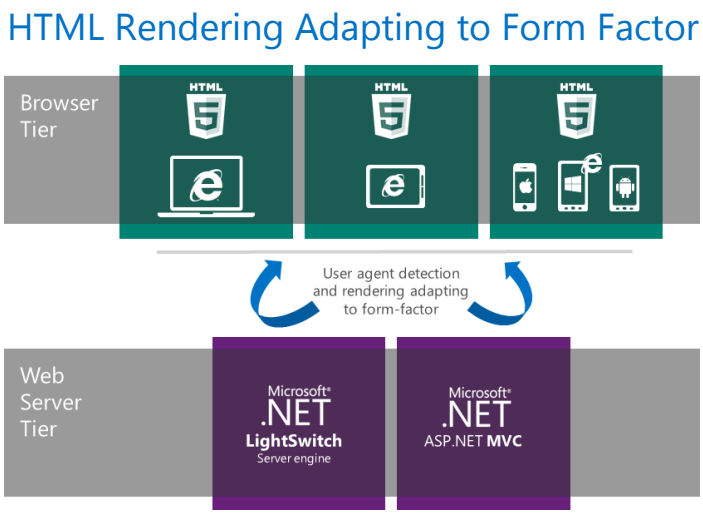


Figure 4-15

5. Established application patterns

Building modern business applications is not only about creating new mobile applications. The new experiences demanded by users have to be extremely well integrated with the business processes so they can unlock the value already provided by the core applications of any enterprise.

This section will cover the technologies commonly used for those established patterns, as well as the recommended approaches for extending these applications to embrace the concepts of modern business applications. It is structured by the two main categories of applications found in any company: small and medium applications or large, mission-critical applications.

Business-applications segmentation by priorities

In this section, the established application patterns are organized by application priorities. For instance, is it a small, departmental application? Or are you working on a long-term, core-business, mission-critical application? These categories have very different priorities.

From these mentioned business priorities, this section segments business applications into two categories:

- *Small/medium applications*
- *Large, mission-critical applications*

As illustrated in Figure 5-1, business application priorities can be used to determine which category your application falls into. From here, you can refine your needs or come to realize that subsystems in your application fall into separate categories.

Referring to applications just by their size can be tricky. For example, should Twitter be considered a large application based on its number of users and highly scalable architecture? Or does the sparseness of its functionality mean it's a small one?

So, by "application size," we're referring to the volume of complexity, whether it is related to functional complexity (like a custom Enterprise Resource Planning solution) or architectural, scalability, and quality-of-service (QoS) complexity (like Twitter). If the application in question has minimal complexity in the above, for our purposes, it would be considered a small/medium application.

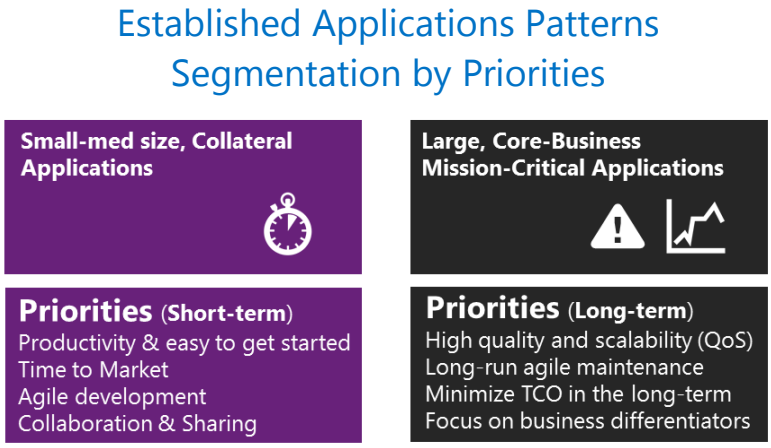


Figure 5-1

Even when small/medium applications are not mission-critical for the whole enterprise, they are still crucial to a certain area of the organization (e.g. a department). Otherwise, why build it?

The priorities of small/medium applications (or non-mission-critical subsystems) are generally development productivity, getting started with ease, and rapid development to deliver business value quickly.

On the other hand, large mission-critical and core-business applications (or even just subsystems rather than a whole application) have additional considerations and long-term goals. When evolving your core-domain applications over a long period of time, frictions with new technology trends may arise. In this case, you want to create software that implements your mainstream business differentiators. Also, in this scenario, you might have more concurrent users, so you might want to have a very high degree of quality of service (QoS). The short-term development productivity might not be as important, but long-run agile maintenance is a necessity. Long-term sustainability and maintainability are crucial for large mission-critical and core-business applications that are "ever-growing" systems.

As shown in Figure 5-2, depending on these priorities, you must consider different development and architectural approaches, as well as the technologies that better align with those approaches.

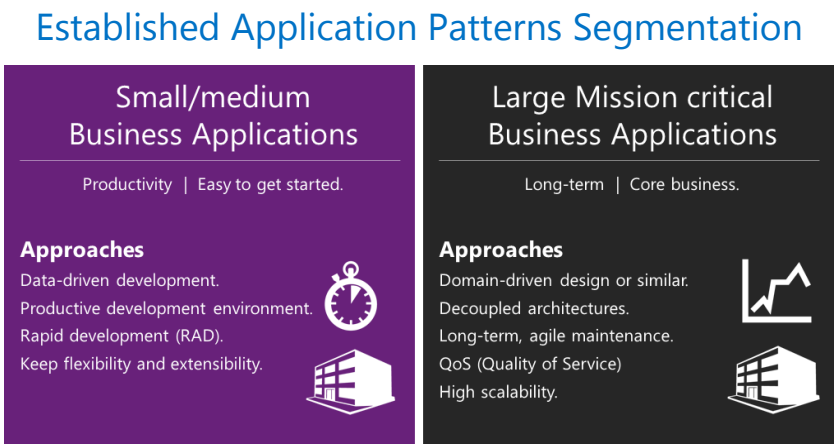


Figure 5-2

Development agility (adapting rapidly to changes) **is crucial for both areas**, but their architectural approaches and even the chosen technologies are usually different for each category.

Most of the time, though, no application falls 100% into one category or the other.

As such, segmenting by priorities should be made not just about applications as a whole, but also about subsystems. Many applications can have some core-business subsystems, but also other collateral subsystems, which can be much simpler, as shown here in Figure 5-3.

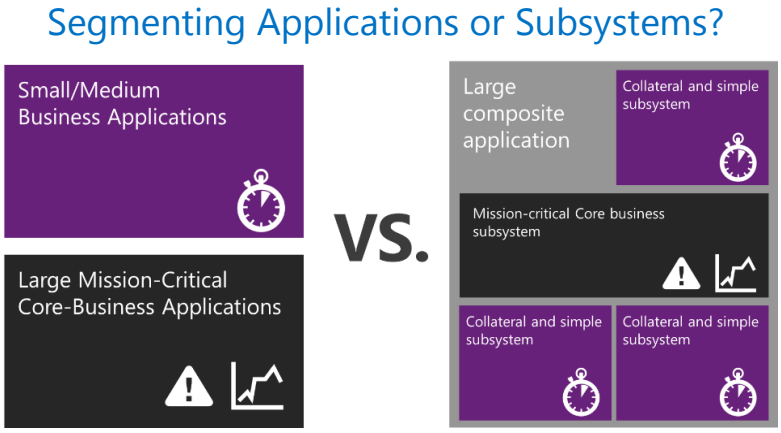


Figure 5-3

Each type of subsystem should be treated and designed in a very different way.

A **subsystem** can be defined as *a differentiated area of your application delimited by a certain boundary (owning code and its own model/data),* so different subsystems can be developed autonomously by different development teams, with small friction. Limited by clear boundaries based on consistency relationships, these types of subsystems are very similar to the *Bounded-Context* concept in *Domain-Driven Design* (DDD) jargon. This concept will be covered in the "Large mission-critical and core-business applications" section later in this white paper.

Therefore, **whenever this guide discusses application categories, it also refers to subsystem categories.** In your case, it will depend on your concrete domain, context, scenarios, and business requirements.

Small and medium business-applications

Business applications generally follow an established approach when supporting controlled environments such as employees and company departments. This section focuses on small and medium-sized business applications that have a relatively short, evolving lifespan. These applications also primarily support data-centric or data-driven scenarios, often referred to as CRUD (Create, Read, Update, and Delete) scenarios. Small and medium business-applications usually have the priorities shown in Figure 5-4.

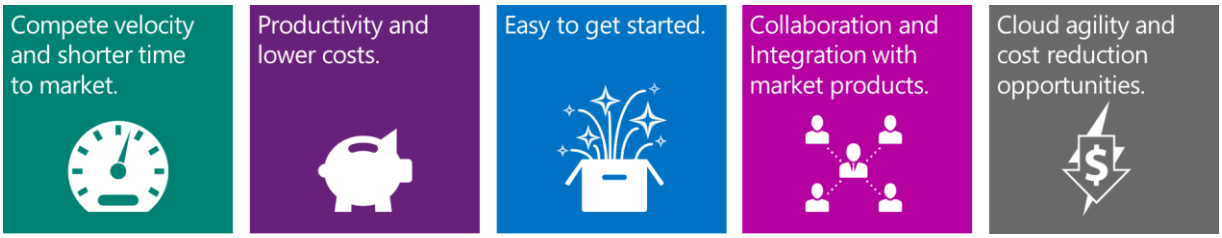


Figure 5-4

The **"compete velocity and shorter time to market"** and a continuous, incremental evolution of experiences is the new norm that IT organizations need to embrace to make their businesses relevant and competitive.

"Productivity and lower costs" is critical for certain areas of the business when applications must be developed and be up and running in short periods of time (likely in a matter of a few months). Therefore, the most valued approach uses a development platform that makes it **easy to get started** and does not require an initial high-cost learning curve. This means you'll want to make a limited number of decisions regarding technology, architecture, and framework.

With regards to development tools, it's beneficial to have a productive, simplified, and clear development tools that provide high agility when creating the production ALM environment (Application Lifecycle Management and source code repository). For instance, a cloud/online and easy-to-get-started ALM infrastructure (like Visual Studio Team Foundation Service) is a perfect fit for developing these types of applications.

Building applications that integrate with or are embedded within **Microsoft SharePoint** and **Microsoft Dynamics** are also in high demand. These scenarios are even more compelling when the application has additional requirements related to collaboration, sharing, document management topics, or common business operations (CRM and ERP).

The **cloud benefits of agility and cost reduction** fit perfectly with medium business-applications where the business stakeholders require agility when publishing the application to production. For instance, it doesn't make sense to develop an application in two months and then wait another month before the application is up and running in the production environment. This is why so many businesses are eager to move applications to agile production environments, whether they are public or private cloud environments. Figure 5-5 summarizes the context of small/medium business-applications.

What is it about?	**When?**: Priorities & Req.	**Examples**	**How?**
Low business-rules changes Short-Medium life Small-Medium app. Mostly isolated CRUD oriented app. (Create, Read, Update, Delete) **Data-Driven app.**	Enterprise agility Fast time to market Rapid initial development Balanced quality with dev.speed Short-term Productivity	Departmental apps Collaboration Apps Collaboration Sites Web-Portals Dash-Boards Contacts-App Mash-up apps	RAD dev & tools Collaboration platform CMS platform Simplified architectures Data-Driven approaches

Figure 5-5

As stated, **development productivity** is usually the most important priority for small/medium business-applications, and it is one of the areas where **.NET** and **Visual Studio** shine. The following excerpt from a Forrester report highlights this topic.


FORRESTER®

FOR: Application Development & Delivery Professionals

The Future Of Microsoft .NET: New Options, New Choices, New Risks

by John R. Rymer and Jeffrey S. Hammond, August 24, 2012

- **AD&D leaders choose .NET for the productivity of an integrated environment.** For many years, AD&D pros have almost unanimously praised Microsoft for producing a highly productive development environment. (Those with experience in Java as well almost always give .NET the advantage.) .NET's productivity rises from three sources: the Visual Studio integrated development environment's (IDE's) design; tight integration of Visual Studio, API libraries, and the underlying platform; and Microsoft's quality implementation of new development concepts.[17]

"Microsoft has done a great job on the [developer] tooling — and integrated the tools and the platform very nicely." (VP of application development, financial services company)

Figure 5-6

The next sections focus on the following scenarios:

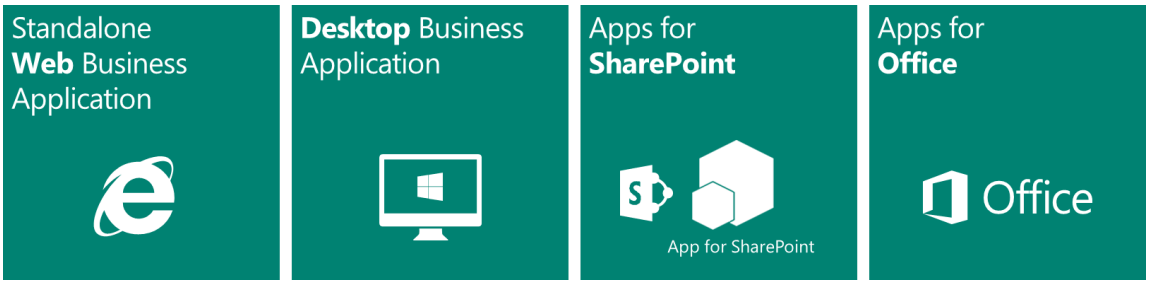


Figure 5-7

- **Small/medium standalone data-centric business applications**
  - **Data-centric web business-applications**
    - *HTML5 Client for LightSwitch projects*
    - *HTML5 + ASP.NET (WebForms, MVC, SPA)*
  - **Data-centric desktop business-applications**
    - *WPF*
    - *Windows Forms*
    - *LightSwitch Windows Desktop Client*
- **Collaboration and productivity business-applications**
  - *Apps for SharePoint*
  - *Apps for Office*

Data-centric web business applications

The majority of data-centric applications cover CRUD operations, including master-detail scenarios. These applications also need to implement business logic, but not extensively. Terefore, these kind of web applications generally aren't large applications with complex domains and a high volume of business rules; instead, they are usually straightforward, and heavily data-driven. The most important priorities for these applications are productivity, cost, and value. Their goal is to achieve short development lifecycles at with a relatively low cost while providing value to the business in an agile manner.

HTML5 is the preferred client technology for the web, over web plug-ins like Silverlight and Flash. HTML5 can be consumed from any device (PC, tablet, smartphone, and more) and heavily uses JavaScript (and many powerful JavaScript libraries, such as jQuery) and CSS.

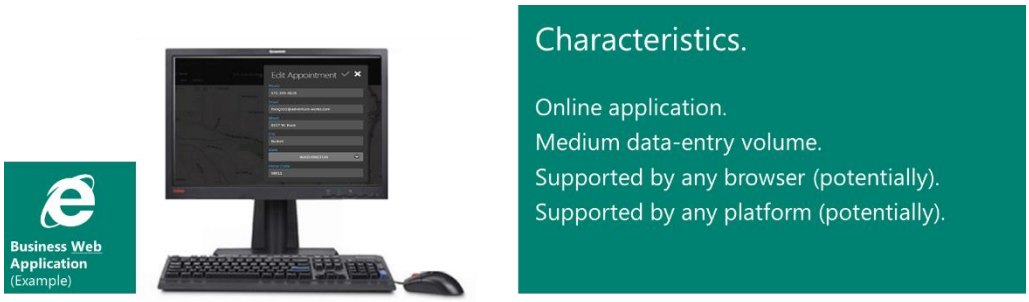


Figure 5-8

The following technology/approach tables are intended to recommend when to use which technology, depending on the context of your web application and its priorities.

Presentation layer technology approaches for business web applications

Technologies	When to use and why
HTML5 client for **LightSwitch** projects	**LightSwitch web client** ▪ Use it when your web application/module is mostly data-driven oriented (CRUD). ▪ This is the easiest way to create data-centric, cross-browser, mobile web clients that can run on any modern device, take advantage of automatic HTML rendering, and adapt to different form factors. ▪ Extend with JavaScript, OSS JavaScript libraries like jQuery Mobile, themes, and CSS3.
ASP.NET **Web Forms** with HTML5 support	**ASP.NET Web Forms with HTML5 support** ▪ For developers who are familiar with Web Forms, this approach provides an easy way to start development while maintaining full control over the code. ▪ Use libraries like Modernizr for detecting HTML5 features support and workarounds. ▪ Use CSS3 for less script and more maintainable code. ▪ Use JavaScript and jQuery for client-side programming.
ASP.NET **Web Pages**	**ASP.NET Web Pages** ▪ ASP.NET Web Pages and the Razor syntax provide a fast, approachable, and lightweight way to combine server code with HTML to create dynamic web content.
ASP.NET **MVC** with HTML5 support	**ASP.NET MVC with HTML5 support** ▪ This is the most flexible and powerful option, with full-HTML rendering control, strong unit testing support, and faking capabilities. ▪ Use libraries like Modernizr for detecting HTML5 features support and workarounds. ▪ Use CSS3 for less script and more maintainable code. ▪ Use JavaScript and jQuery for client-side programming. ▪ Suitable for more complex projects than the projects targeted by LightSwitch or Web Forms.

Table 5-1

References	
ASP.NET Web Forms	http://www.asp.net/web-forms
LightSwitch HTML	http://msdn.microsoft.com/en-us/vstudio/gg491708

For business web applications that are simplified, data-driven scenarios, the main priority is development productivity. Microsoft recommends using LightSwitch with its HTML5 client whenever it covers your requirements. Using LightSwitch is very straightforward when creating data models and screens for CRUD (Create, Read, Update, Delete) operations, including master-detail scenarios. In many cases, you won't even need to write code, although you may want to customize styles and extend the operations with code through the many extensibility points that LightSwitch offers (JavaScript and .NET extensibility points).

However, when using LightSwitch, the application will be completely coupled to the LightSwitch runtime, which is comprised by an end-to-end, transparent engine built upon ASP.NET and OData Services on the server, and on JQuery Mobile on the client when using the HTML5 client. It automatically renders screens based on the data controls to be shown. If the application is not data-driven and has a more complex UI for the majority of its surface, LightSwitch is probably not the best fit.

The upside of this approach is you will get unprecedented development productivity when a high percentage of LightSwitch features map to the application requirements.

Another possibility for simple data-centric web applications with a good initial development productivity is **ASP.NET Web Forms.** They support visual drag-and-drop of web controls and have grid controls that are easy to use. Web Forms is also useful when you have simple data binding that fits nicely in a tabular format, and you want to provide a simple way for users to update records. Additionally, Web Forms is generally quite easy if the developer's background is about desktop applications, like developing using WinForms, or WPF.

For more advanced scenarios, where you are looking for the most flexible and powerful Web UI (but not the easiest learning path), we recommend using **ASP.NET MVC** plus HTML5/JavaScript, or even considering advanced approaches like **ASP.NET SPA** (Single Page Application) for heavily JavaScript-centric applications.

Scenario: End-to-End Small/Medium Web Business Applications

Figure 5-9 shows a typical technology connection pattern for small/medium web-based business applications. Independent of the technology approach you take (such as LightSwitch or ASP.NET), HTML5/JavaScript is the web client technology that is always present in modern web development.

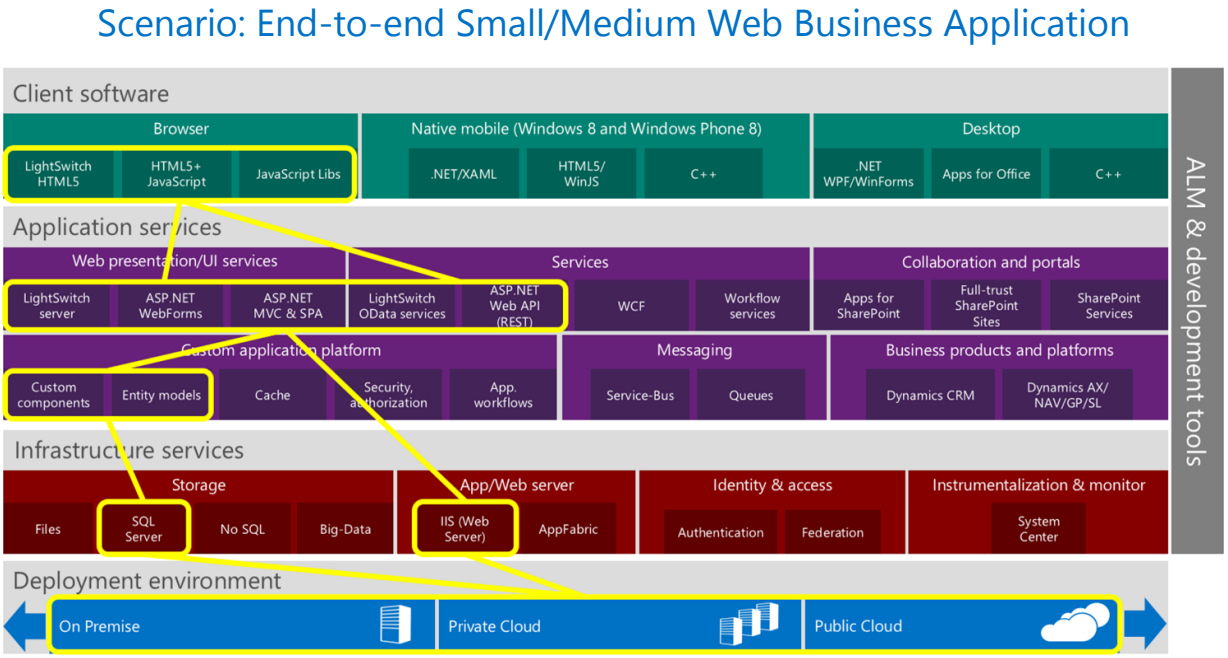


Figure 5-9

The technologies and reasons why you should choose one or another technology for building services are quite similar to what was described previously. For more details regarding the services development technology choices (like ASP.NET Web API, WCF, etc.), and to avoid redundancy, please refer to the Services section within the "Emerging application patterns" section.

Mixed approach for small/medium business web applications

Depending on the requirements and priorities of your web app, some approaches are clear, like using LightSwitch for data-driven apps, finer-grain technologies like ASP.NET MVC and plain HTML5/JS for more complex apps, or ASP.NET Web Forms if it suits your experience and skills.

But again, consider that you may have several subsystems within the same application. In these cases, you will likely use a mixed approach within the same application, like the one shown in Figure 5-10. This is an example of an application with 40 percent CRUD-based operations developed using LightSwitch, and 60 percent finer-grained technologies (like ASP.NET MVC) and advanced approaches (like SPA). SPA allows you to leverage HTML5, JavaScript, jQuery, and other JavaScript libraries to build the most critical and dynamic areas of the application. That 60 percent could also be developed using ASP.NET MVC or Web Forms, depending on your requirements and your expertise. Using the LightSwitch extensibility points, you can consume external services from JavaScript (LightSwitch Client Tier) or from .NET server code (LightSwitch Middle Tier).

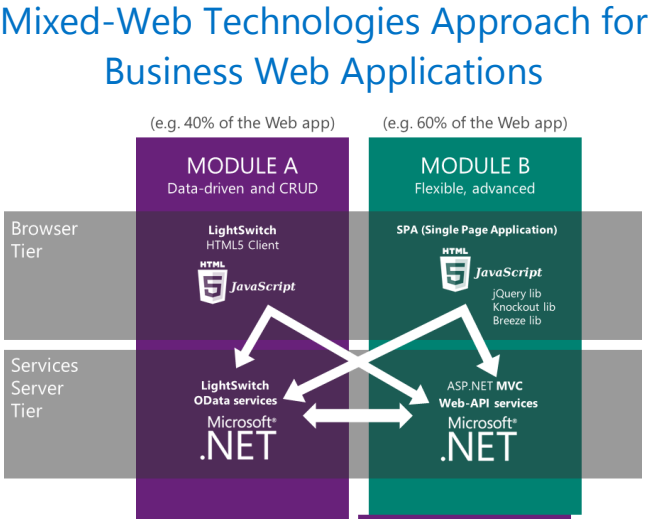


Figure 5-10

Data-centric desktop business applications

You might prefer to build desktop applications more than web applications because your application involves heavy-duty data entry; because you have complex offline scenarios that involve local storage, COM interoperability, and automation; or perhaps because your end users prefer desktop applications based on their work needs and skillsets.

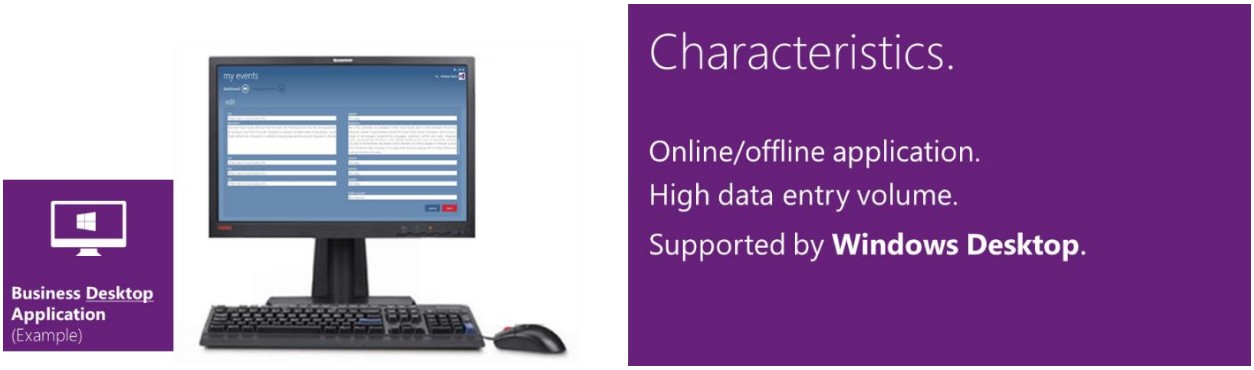


Figure 5-11

Depending on the architecture of your desktop application, you will need certain technologies. Historically, the most common architectures for desktop applications are the 2-tier and 3-tier architectural styles shown in Figure 5-12.

Figure 5-12

The highlighted technologies that can be used when developing the client tier in small/medium desktop applications are the following:

- *.NET WPF*
- *.NET Windows Forms*
- *LightSwitch for desktop*

The following tables recommend when to use which technology, depending on your application priorities and particular context.

Presentation layer technology approaches for small/medium business *desktop* applications

Technologies	When to use and why
.NET WPF	**.NET Windows Presentation Foundation** ▪ This is the preferred technology for Windows-based desktop applications that require UI complexity, styles customization, and graphics-intensive scenarios for the desktop. WPF also takes advantage of XAML views. And WPF development skills are similar to Windows Store development skills, so migration from WPF to Windows Store apps is easier than migration from Windows Forms. ▪ Take advantage of new simplified asynchronous capabilities in .NET 4.5 (async/await). ▪ Take advantage of SignalR .NET client for bi-directional communication between the client and server.

.NET WinForms	**.NET Windows Forms** ▪ This was the first available UI technology in the .NET Framework for building desktop applications. It is still a good fit for many business desktop applications. Windows Forms is easier to use and lighter than WPF for simple scenarios where you don't need UI styles customization. ▪ It provides simpler UI capabilities than WPF and it is not based on XAML, limiting its path forward to Windows Store Apps. ▪ Can also take advantage of the latest .NET capabilities such as asynchronous programming with async/await.
LightSwitch desktop client	**LightSwitch desktop client (out of browser)** ▪ If you have already adopted the LightSwitch middle-tier and HTML5 solutions, you can use the desktop experience that LightSwitch supports out of the box.

Table 5-2

References

WPF desktop applications	http://msdn.microsoft.com/en-us/library/aa970268.aspx
	http://msdn.microsoft.com/en-us/library/ms754130.aspx
Windows Forms	http://msdn.microsoft.com/en-us/library/dd30h2yb.aspx
LightSwitch	http://msdn.microsoft.com/en-us/vstudio/ff796201

Scenario: Small/Medium 2-Tier Desktop Application

Figure 5-13 shows the typical possibilities with the technology connection pattern for a **small/medium desktop business application with a 2-tier approach,** where the WPF/WinForms and custom components and data-access code would be running within the same client tier (desktop PC).

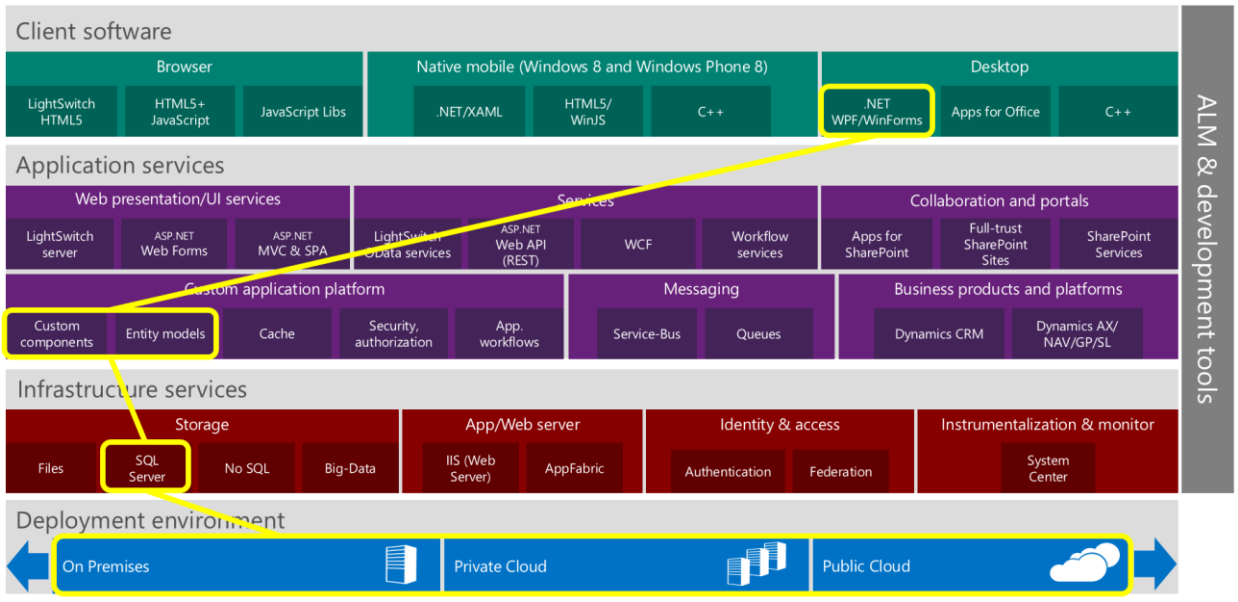


Figure 5-13

The old-fashioned 2-tier architectural approach (traditional client/server) is not recommended in most scenarios because of issues such as direct dependency on client database drivers in the client PC, issues if there's a firewall in between the client PC and the database server, and scalability limited by the database server. However, we included this scenario as it is still a very common approach in established applications. In this scenario, you would use WPF as a desktop client and directly access the data source (typically a database like SQL Server) using data-access technologies, like Entity Framework or ADO.NET.

When applying this 2-tier approach, at the very least, **it is recommended to have separation of concerns within your .NET WPF code**, like separating the application and business rules code from the data-access code in different internal layers. If you don't do this, your desktop application will grow monolithically and be very hard to maintain.

Scenario: Small/Medium 3-Tier Desktop Applications

Figure 5-14 below shows an example using a technology connection pattern for a small/medium desktop business application with a 3-tier approach.

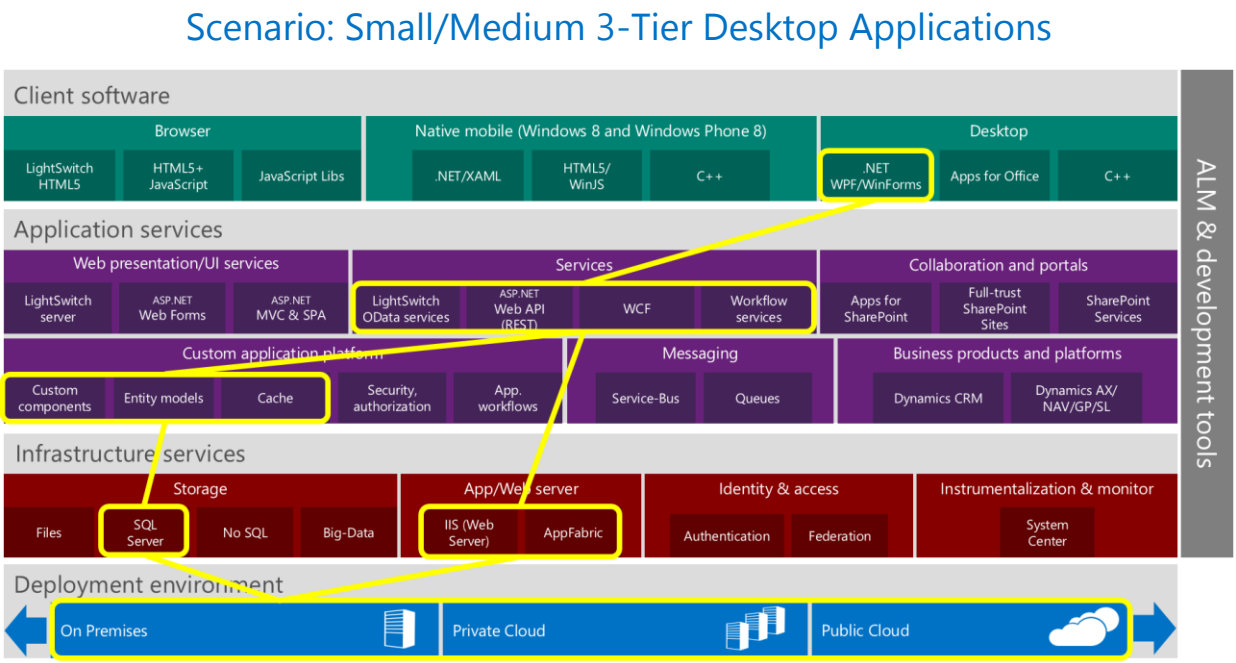


Figure 5-14

Depending on the requirements and priorities of the desired desktop application, you would use WPF as desktop client and OData Services or LightSwitch OData Services for very data-driven applications (CRUD scenarios). Alternatively, you could use WCF if your services are RPC/command-oriented.

The reasons why you should choose one or another technology for services are quite similar. For more details regarding the services-development technology choices (like ASP.NET Web API, WCF, etc.), please refer to the Services section within the "Emerging application patterns" section.

Especially when your services are not just about data access (CRUD), but also have business rules embedded within services, it is recommended to separate the business-rules code from the data-access code and position each type of code in different internal layers (following *the separation of concerns* principle), having those layers within the services tier.

When accessing the data tier, the recommended technology, by default, is Entity Framework. In this simpler and data-driven scenario, the most common usage of **Entity Framework** is through its visual designers and **model-first** or database-first approaches. But you can also use plain "Code First POCO entity classes" if you require full control of the entity classes.

Modernizing desktop business applications

Experiences running on the desktop should also be built considering the new customer's expectations. .NET provides multiple innovations for desktop applications to better address these expectations, as well as capabilities that allow your application to be extended to new platforms without changes in the architecture and you can even reuse code. Building your desktop applications with these recommendations in mind will extend their lifespans and make it easier for you to extend to new devices—or even migrate the entire application in the future.

- **Use the Model-View-ViewModel design pattern (MVVM):** Microsoft client platforms (including WPF) make it easy to build applications using the MVVM pattern. With this pattern you will get a strong separation of display from state and behavior, which will help you to create clean and maintainable code that can be easily shared between multiple devices.
- **Use portable class libraries for the client logic:** .NET portable libraries allow binaries to be shared between multiple platforms such as the desktop, Windows Store apps, Windows Phone apps, and others. Implementing your client logic with .NET portable libraries will greatly simplify the creation of multiple experiences on multiple platforms.
- **Modernize your user experience:** Concepts that are demanded by today's end users can be implemented with the latest innovations on .NET for the desktop. Design principles such as "fast and fluid," "authentically digital," and "do more with less" can be applied to your existing desktop application by employing a modern UI for your XAML design, carefully using animations, and implementing .NET asynchronous programming extensively.
- **Move the business logic to the server:** Two-tier applications (client/server) are significantly harder to extend to new devices. The recommended approach is to clearly separate the business logic into services that can be reused later on other devices and form factors.
- **Extend to the cloud:** Once separated from the client, Windows Azure provides multiple solutions to move the business logic to the cloud. Transforming that logic into cloud services greatly improves the elasticity and scalability of existing solutions, making them ready to embrace multi-device approaches.

Visual Studio partners provide a set of technologies that will also help you modernize your .NET applications.

Visual Studio partners for .NET application modernization

Partners	When to use and why
Xamarin	**Xamarin** provides a means to share C# code from your applications targeting Windows or Windows Phone with iOS and Android devices. It provides access to the underlying API to create tailored views while reusing the client logic code between devices.
ITR-Mobility **iFactr** and **MonoCross**	**ITR-Mobility** offers a solution for building enterprise mobile applications in C# for delivery on the major mobile platforms. It provides services such as Abstract UI and Enterprise Data Synchronization to enable business applications across a range of devices.
Mobilize.NET by Art in Soft	**Mobilize.NET** provides solutions and services for migrating legacy applications to modern platforms—including the web, mobile, and the cloud—by transforming the existing source code into new code without runtimes for the output application.
Citrix	**Citrix Mobile SDK for Windows Applications** provides a rich tool kit for developers to mobilize LOB Windows applications or write new touch-friendly applications executing on central servers (Citrix XenApp/XenDesktop) and accessed from any mobile device with Citrix Receiver.

Table 5-3

References	
Xamarin	http://xamarin.com/features
ITR-Mobility iFactr and **Monocross**	http://itr-mobility.com/products/ifactr http://monocross.net/
Citrix	http://www.citrix.com/mobilitysdk/
Mobile.NET	http://mobilize.net/
VSIP partner directory	Visit the VSIP partner directory for more solutions provided by Visual Studio industry partners: https://vsipprogram.com/Directory

Modernizing applications based on RIA containers

A few years ago, when Rich Internet Application (RIA) containers and plug-ins were popular, the context in IT was quite different from today. Five years ago, RIA covered most deployment needs by just targeting Windows-based PCs and Mac computers. After the "device revolution" in 2010, you now have different devices (tablets, smartphones, and computers) with different operating systems (including Windows 8, Windows Phone 8, iOS, Android, and Chrome OS)—and many of them don't support RIA plug-ins.

At the same time, HTML5 has been evolving to support richer scenarios that previously required plug-ins. Currently, HTML5 is broadly supported across all devices and offers a better alternative for cross-platform client development than traditional plug-ins.

Native applications are also becoming more popular in the market, as they take full advantage of each device's specific features to provide the most compelling experience for customers.

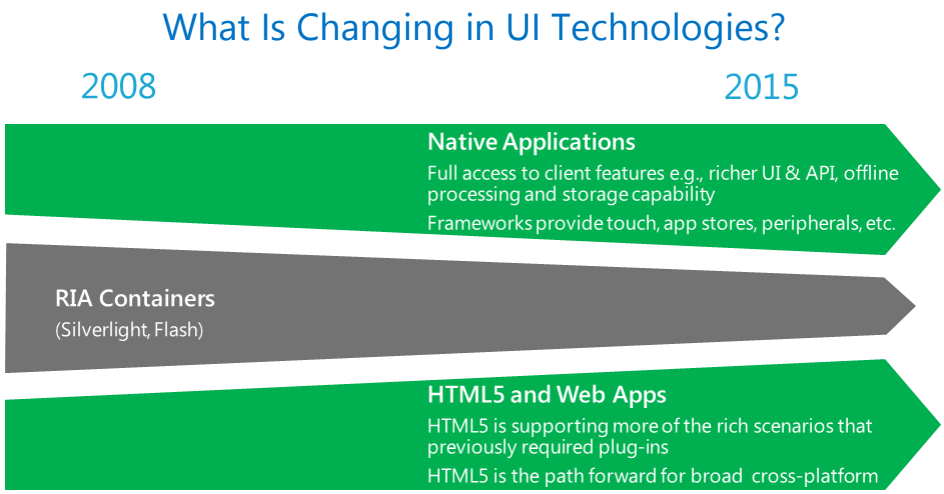


Figure 5-15

The Microsoft platform supports all three approaches for user interfaces (web, native, and RIA), but it also takes into account that modern experiences across devices are primarily developed with web and native technologies. Deciding when to make that transition will ultimately depend on your requirements and needs. Microsoft is committed to supporting your choice and helping you throughout the process:

- If you are transitioning to **native applications**, you can leverage your existing skills and even code by targeting XAML/.NET natively on any Windows device. Portable libraries will also allow you to share your binaries between different platforms, including Silverlight.
- For **browser-based HTML5 apps**, Microsoft provides leading tools and frameworks to help you create applications for any device based on the latest standards. Silverlight's interoperability with HTML also enables a gradual transition through hybrid applications.

- If your application's targeted scenario is still **only supported by Silverlight** (for example, video content protection) or emerging patterns are not a requirement for your applications yet, you can continue to use Silverlight. Silverlight is a mature and stable technology, and its latest version (Silverlight 5) was released with extended support for 10 years to ensure you get the most from existing investments and to allow you to gradually transition to HTML5 or native solutions.

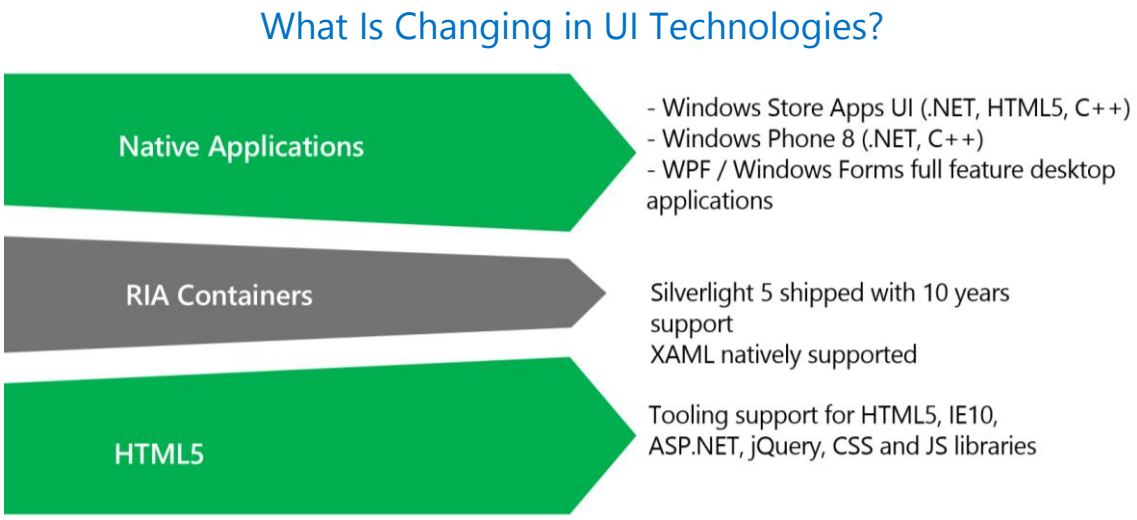


Figure 5-16

The Silverlight migration paths appendix at the end of this guide provides additional details and recommendations.

Cloud app model for Office and SharePoint

Office 2013 introduced the cloud app model for extending Office and SharePoint through lightweight apps. This delivers the value of your business applications through the Office productivity applications your customers already use. The cloud app model is built on standard web technologies such as HTML, CSS, JavaScript, REST, OData, and OAuth on the client—along with any server technology, including ASP.NET, on the server. If you're a web developer, you can use your existing skills to build apps and take advantage of familiar tools, languages, and hosting services. You can deploy, update, and maintain your apps faster in the cloud, then publish and sell your apps in the Office Store, or distribute IT-approved apps within your organization by using an internal app catalog.

The unified app model applies to the following types of applications:

- **Apps for Office** (applies to Office 2013, Office 365, Project Professional 2013, Word 2013, Excel 2013, PowerPoint 2013, Outlook 2013, Outlook Web App, Excel Web App, and Exchange Server 2013).
- **Apps for SharePoint** Server 2013 and SharePoint Online in Office 365.

Apps for Office

Apps for Office are based on the cloud app model for Office and SharePoint.

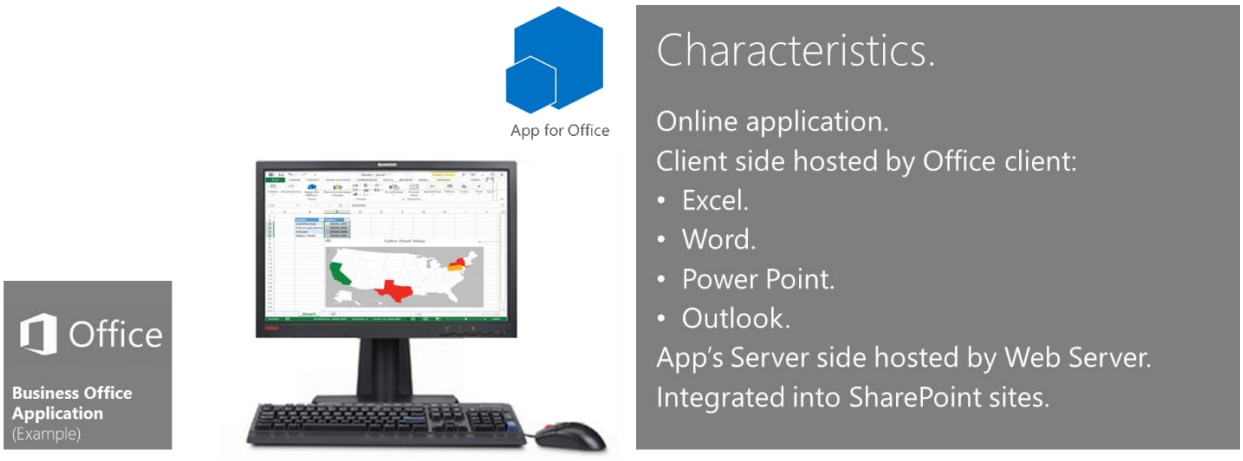


Figure 5-17

An app for Office is simply an HTML web page plus an XML application manifest as shown in Figure 5-18.

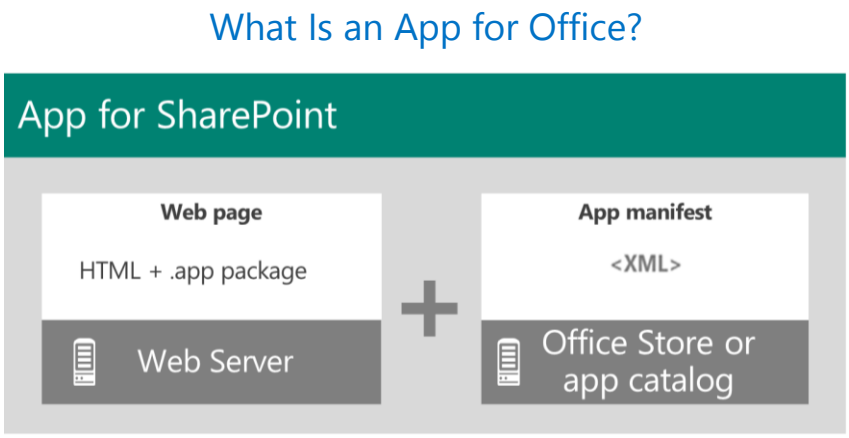


Figure 5-18

HTML content using web standards can be exposed as Task Panes or Content within an Office document, and you can programmatically access the Office content from JavaScript.

The application manifest is required in order to publish and have your app discoverable in the public Office Store or in any private app catalog.

You can create the following types of apps for Office:

- **Task pane apps:** These apps appear in the Task Pane of an Office client.
- **Content apps:** These apps appear inside of the Office document's content.
- **Mail apps** for Outlook 2013 and Outlook Web Access: These apps appear next to the Outlook item that's open. (This could be an email message, meeting request, meeting response, meeting cancellation, or appointment.)

Figure 5-19 shows a few examples of content apps for Office (in Excel, Outlook, and Word). All the special content shown in these apps are elements created using HTML pages embedded in the Office client by hosting it within an IFrame and accessing to Office client elements from the JavaScript code.

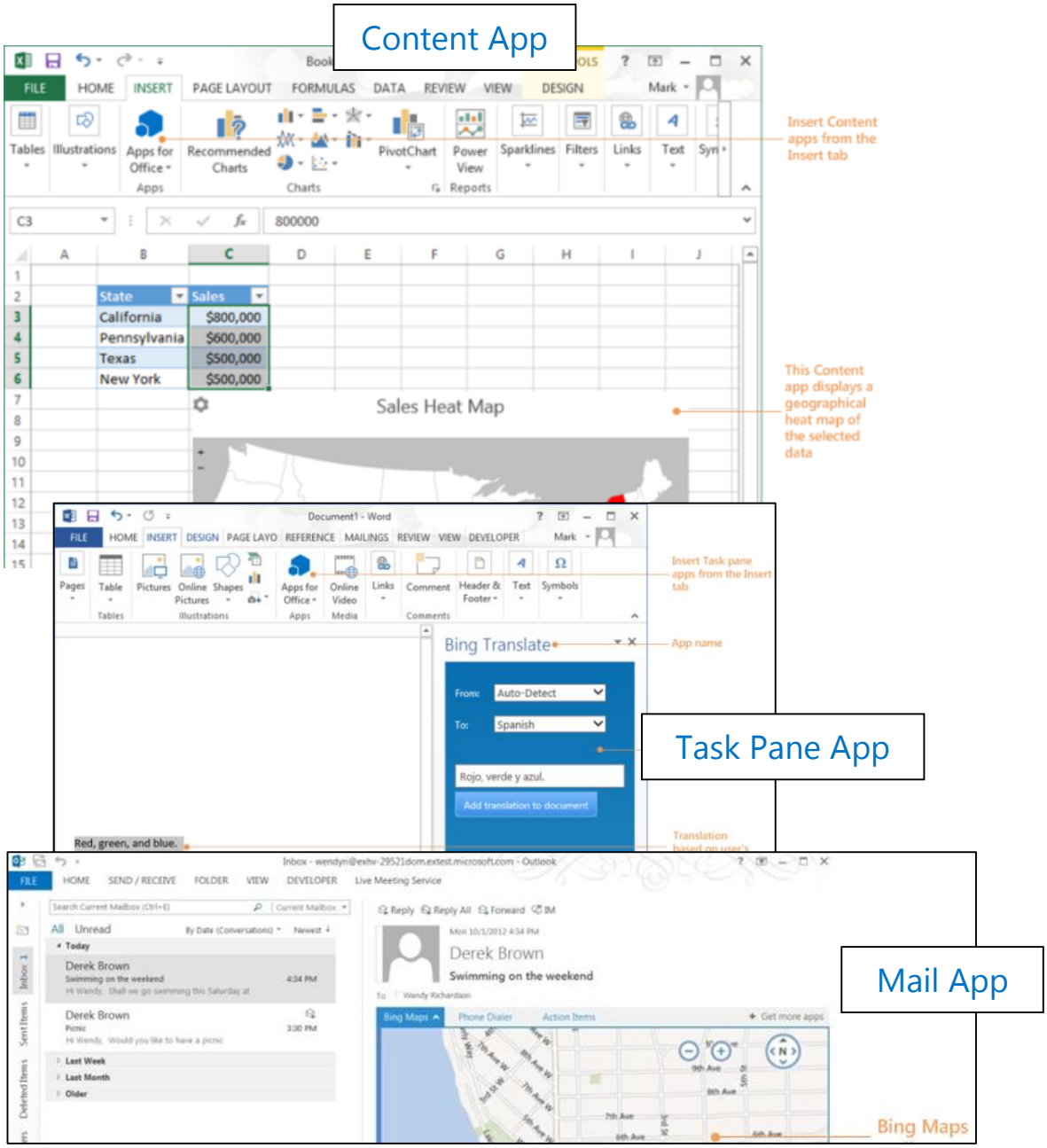


Figure 5-19

The main advantages of the new-apps model versus the legacy VSTO (Visual Studio Tools for Office) application model (add-ins, or document-based) are:

- Support Office Web Apps
- Support for Office RT
- Easier to migrate to future versions of Office

Additionally with this new approach, web developers can take advantage of their skills to easily create apps for Office.

The following tables recommend related technologies for developing applications embedded in a Microsoft Office client, depending on the application's priorities and specific context. In addition to the new apps for Office mode, legacy technologies are also included.

Technology choices for Office development

Technologies	When to use and why
Apps for Office model - Office Developer Tools - "Napa"	**HTML5/JavaScript web client development embedded in Office** ▪ It is the preferred technology for new apps for Office and the future direction for Office extensibility. ▪ It should be the "by-default" choice when developing for Office 2013. ▪ Allows accessing documents via the web or from Office RT. ▪ Better suited for online application execution with access to servers hosting the apps.
Office Add-ins - Visual Studio **Tools for Office (VSTO)**	**Office add-ins and document-based applications with VSTO (.NET-managed code in the client)** ▪ Use it if you need to create applications for Office with older versions than Office 2013 or scenarios not covered by the apps for Office model. ▪ It is fully supported. ▪ Better suited for offline application execution and advanced scenarios.
Office VBA	**Visual Basic for Applications (VBA)** ▪ Use VBA for automation and repetitive solutions or extensions to user interaction. ▪ Supported for automation of Office and simple application scenarios. ▪ Hybrid approach of apps for Office and Office VBA. Where the VBA is in a document template and it interacts with an app for Office via the content of the document/custom XML parts.

Table 5-4

References

Overview of apps for Office	http://msdn.microsoft.com/library/office/apps/jj220082(v=office.15)
VBA for Office developers	http://msdn.microsoft.com/en-US/office/ff688774

Tools and technology choices when developing apps for Office

Technologies	When to use and why
"Napa"	**"Napa" (Lightweight IDE, coding in the browser)** ▪ It is the easiest way to start developing apps for Office and SharePoint. ▪ It does not require installation of Visual Studio in the development machines. It's available as a free app for your Office 365 Developer Site.
Office Developer Tools for Visual Studio	**Office Developer Tools for Visual Studio (full power)** ▪ Use it for professional development when you demand full power and flexibility. ▪ Office Developer Tools for Visual Studio is free, but requires Visual Studio Professional, Premium, or Ultimate.

Table 5-5

In many scenarios, business apps for Office provide visualizations or an analysis of business data. A typical pattern for accessing data is through services. Because the consumption of these services will be from JavaScript, the best fit is ASP.NET Web API Services (REST/JSON/OData, and so on). If the services are simple and data-driven, an easier approach is to use LightSwitch OData services or WCF Data Services. However, apps for Office can have a web back end, so JavaScript is not the only option. You can also access business data from the server side using ASP.NET.

The reasons why you should choose one or another technology for building services are quite similar to those previously mentioned. When developing custom services to be consumed from apps for Office using technologies like ASP.NET Web API, WCF, etc., please refer to the Services section within the "Emerging application patterns" section for more information.

Scenario: Connected Apps for Office

Figure 5-20 shows the recommended possibilities for a connected app for Office that is consuming remote services and accessing a data source.

Using SharePoint services is optional. An App for Office can be isolated from SharePoint, as well.

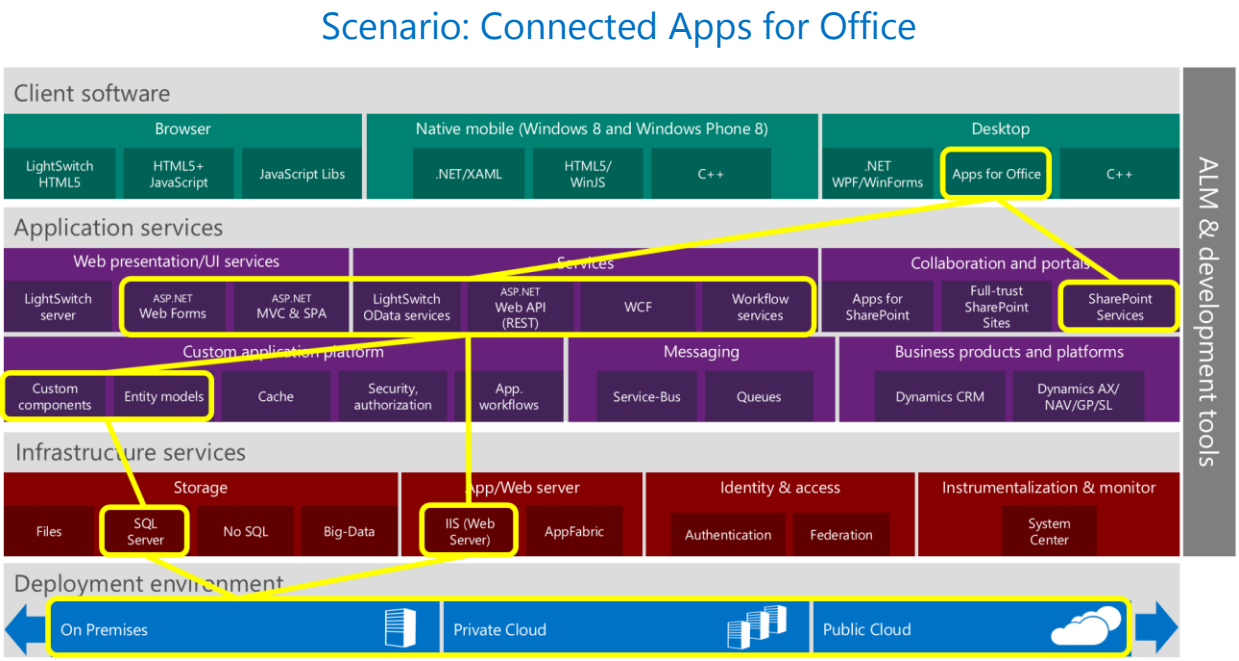


Figure 5-20

Apps for SharePoint

Apps for SharePoint are based on the introduced cloud app model for Office and SharePoint.

Figure 5-21

An app for SharePoint is an HTML web page or a more complex web application (based on any server engine, like ASP.NET MVC, or even non-Microsoft technologies like PHP, NodeJS, and so on) that provides HTML pages. Because it is a regular web application, the HTML page can use any web standard (HTML5/JS) technology and library, like jQuery or modern SPA approaches.

It can also access and expose SharePoint resources like a List, SharePoint Business Connectivity Services (BCS) models, and implement a SharePoint Workflow.

The application will then need to be registered into SharePoint using an app manifest. An **app manifest is an XML file that declares the basic properties of the app** along with where the app will run and what to do when the app is started. The model is really the same as the one used for apps for Office.

Apps for SharePoint can be shown in the web browser as immersive full webpages, as app-parts (based on IFrames within SharePoint pages), and extending SharePoint UI custom actions.

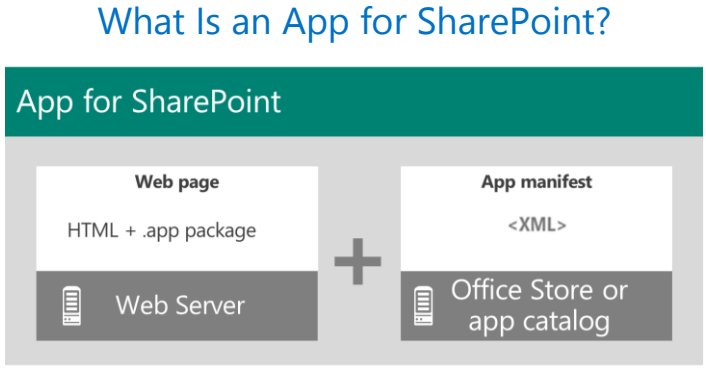


Figure 5-22

Figure 5-23

Figure 5-23 shows the different implementation options for apps for SharePoint. Although using the full webpage approach may seem too isolated from SharePoint, it isn't, because apps for SharePoint can use the same look and feel using Chrome Control. Through OAuth, the user's security context can also be shared between SharePoint and your custom app. Within apps, SharePoint 2013 decouples your server-side code from the SharePoint server, enabling you to run your server-side code in your own web server, whether it is on-premises or in the cloud.

When deploying an app into the SharePoint server, that app can have just client-side code (HTML and JavaScript) rather than server-side code. But, auto-hosted and provider-hosted models allow server-side code in your own server/services, decoupled from the SharePoint servers.

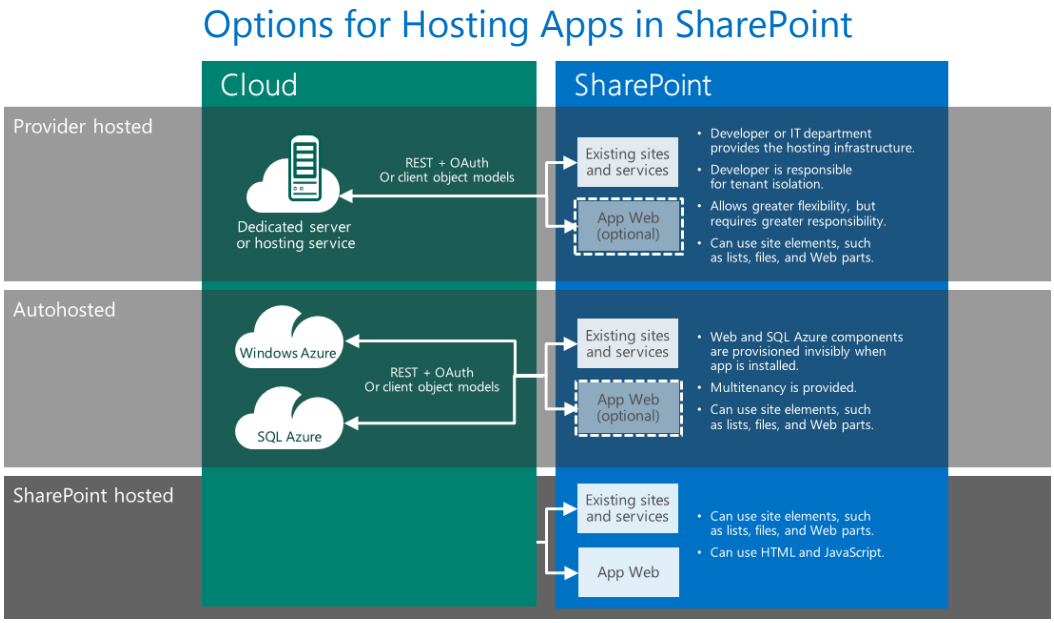


Figure 5-24

The auto-hosted model is similar to the provider-hosted model, but it provides a transparent and automatic provisioning and deployment of your server application on Windows Azure.

The following tables show and recommend which technology to use when developing applications related to SharePoint, depending on your application priorities and specific context. For this scenario you will also find other approaches in addition to the new apps for SharePoint model.

Choices for SharePoint development

Technologies	When to use and why
Apps for SharePoint model	**Apps for SharePoint (full-page app, app parts, and UI custom actions)** ▪ Default approach for autonomous apps integrated with SharePoint. ▪ Approach decoupled from SharePoint servers.
SharePoint sites	**SharePoint sites customization** ▪ Use it when customizing SharePoint sites themselves. ▪ Templates and pages customization through the Design Manager Feature and Visual Studio.
SharePoint full-trust farm solutions	**Full-trust development, admin extensions, traditional Web Parts, etc.** ▪ Create farm solutions when you can't do it in an app for SharePoint. ▪ Primarily for custom administrative extensions of SharePoint or developing internal assets within SharePoint. ▪ Required approach for SharePoint versions older than SharePoint 2013.
SharePoint sandboxed farm solutions (deprecated)	**Sandbox development** ▪ This approach has been deprecated since SharePoint 2013 (although it is supported in SharePoint 2013), therefore, it's unadvisable to build new sandboxed solutions. ▪ Use it only in existing developments for versions older than SharePoint 2013.

Table 5-6

References

SharePoint 2013: What to Do? Farm Solution vs. Sandbox vs. App	http://social.technet.microsoft.com/wiki/contents/articles/13373.sharepoint-2013-what-to-do-farm-solution-vs-sandbox-vs-app.aspx
Apps for SharePoint overview	http://msdn.microsoft.com/en-us/library/fp179930.aspx
The New SharePoint	http://sharepoint.microsoft.com/blog/Pages/BlogPost.aspx?pID=1012

Tools and technology choices when developing apps for SharePoint

Technologies	When to use and why
"Napa"	**"Napa" (Lightweight IDE, coding in the browser)** ▪ It is the easiest way to get started developing apps for Office and SharePoint. ▪ It does not require you to have Visual Studio installed on the development machines.
Office Developer Tools for Visual Studio	**Office Developer Tools for Visual Studio (full power)** ▪ Use it for professional development when you demand full power and flexibility. ▪ Office Developer Tools for Visual Studio is free with Visual Studio Professional, Premium, and Ultimate.
LightSwitch	**Apps for SharePoint powered by LightSwitch** ▪ Use it when creating data-driven apps (CRUD) either consuming SharePoint resources (lists and services) or non-SharePoint services and data sources. ▪ The easiest way to create flexible forms and data-centric applications integrated to SharePoint.

Table 5-7

In most scenarios, business apps for SharePoint require access to some kind of data, whether it's business data or SharePoint data. That data access is provided through services or local data sources.

Back-end service technologies for apps for SharePoint

Technologies	When to use and why
SharePoint client .NET Object Model or REST/OData endpoints	**SharePoint client .NET object model or REST/OData endpoints** ▪ Use it when working with SharePoint lists, content types, list items, document libraries, and any type of operation related to SharePoint resources. ▪ Note that when developing apps for SharePoint, it is not possible to use the managed SharePoint Server OM class library because the app for SharePoint runs on a different remote server.
SharePoint JavaScript client object model	**SharePoint REST/OData endpoints or client .NET-managed API** ▪ Use it when consuming SharePoint resources directly from JavaScript in your app.
Custom ASP.NET Web API	**REST approach, resource oriented** ▪ Use it if the service's consumers are about native apps, web clients, or unknown Internet consumers. It is a flexible approach. ▪ ASP.NET Web API is a framework that makes it easy to build HTTP services that reach a broad range of clients, including browsers and mobile devices. ▪ Light framework, easy to get started, and high degree of interoperability with other platforms and formats (JSON, OData, and so on). ▪ Especially made for REST services. Embraces HTTP verbs as app drivers. Resource oriented. ▪ High scalability, thanks to Internet caches (Akamai, Windows Azure CDN, Level3, and so on) based on HTTP verbs.
LightSwitch OData REST Services	**REST approach, resource oriented, easiest way to build services** ▪ Use it when creating simple resource-oriented OData services. It is the easiest way!
Custom WCF Data Services	**WCF Data Services** ▪ Use when your services are data/resource-oriented and mostly CRUD and Data-Driven. ▪ It only supports OData. It is straightforward to use, but offers less flexibility and control than using ASP.NET Web API. ▪ Shares the same OData core libraries with ASP.NET Web API

Table 5-8

References

Choose the right API set in SharePoint 2013	http://msdn.microsoft.com/en-us/library/jj164060(v=office.15).aspx

Scenario: Connected Apps for SharePoint

Figure 5-25 shows a technology connection pattern with the typical possibilities when creating an app for SharePoint.

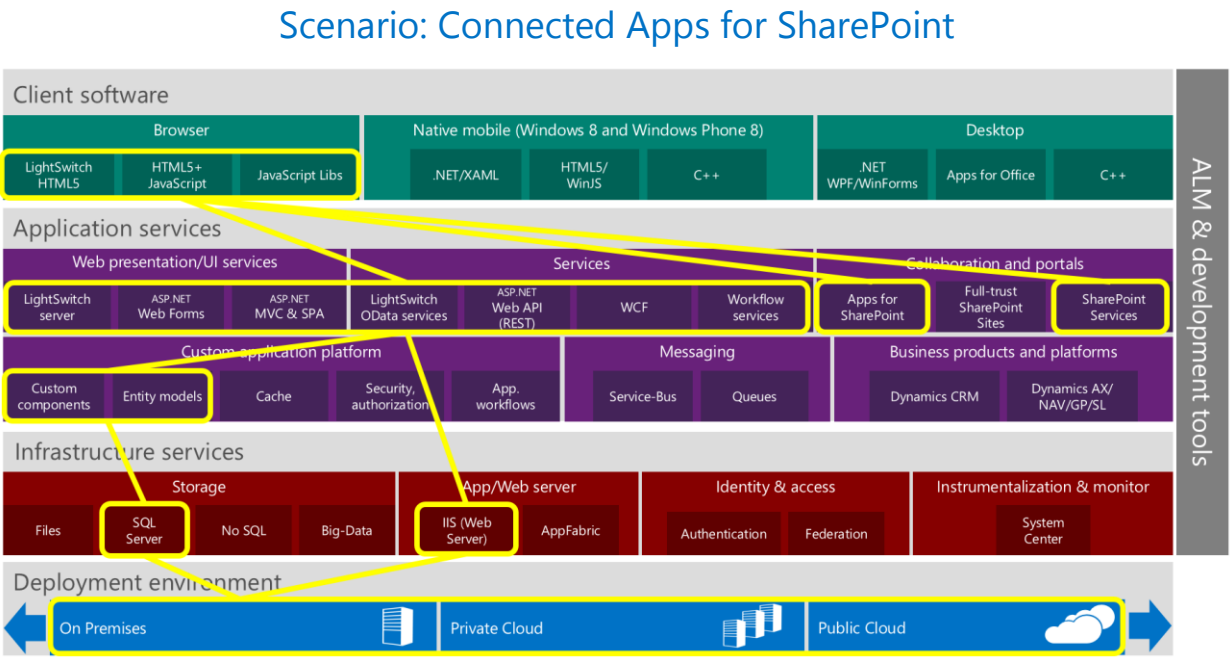


Figure 5-25

Within this scenario, most of the recommendations on web application development technologies already described in this document apply in a similar way, but it is worth highlighting regarding technologies to build services, SharePoint is heading strongly to OData services support and custom apps for SharePoint should also follow that path creating and consuming OData services (based on any technology that supports OData, like ASP.NET Web API, WCF Data Services and LightSwitch OData Services).

Large, mission-critical business-applications

A mission-critical business application can be defined as any business application that is critical to the proper running of mainstream business operations. If an application of this nature fails for any length of time, you may be out of business.

Mission-critical and core-business applications have additional goals and specific priorities. Long-term sustainability and maintainability are crucial for large, ever-evolving applications.

If you're working with large, mission-critical applications, you have to deal with two main concerns:

- **Tackling complexity in the core domain of the business application.**
  - This involves complex domain expertise and business rules, and how to effectively project these into the software.
  - This concern must drive architecture, design, and implementation decisions (such as monolithic versus structured/layered approaches) and best patterns and practices, depending on the context (such as *Domain-Driven Design* architectural approaches, loosely coupled architectures based on dependency injection techniques and containers, and so on). The application design must take the future evolution and maintainability of the application into consideration.
- **Ensuring enough QoS (quality of service) for large, mission-critical applications.**
  - This involves availability, scalability, security, and similar subjects.
  - This concern must drive how cross-cutting measures (security, operations, instrumentation, and so on) are designed and implemented.
  - Infrastructure architecture is also closely linked to QoS. For instance, you must consider infrastructure architecture on-premises or in the cloud depending on required scalability and elasticity, type of users, and go-to-production agility needs.

Figure 5-26 below summarizes what, when, and how to handle these types of applications.

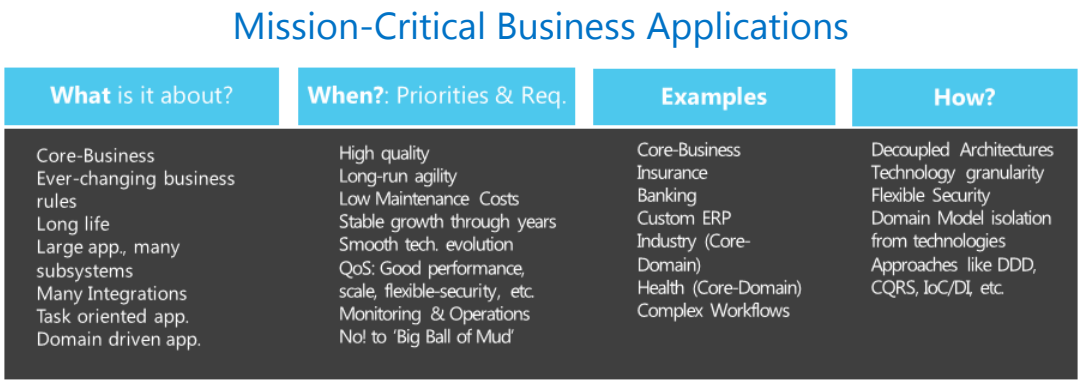
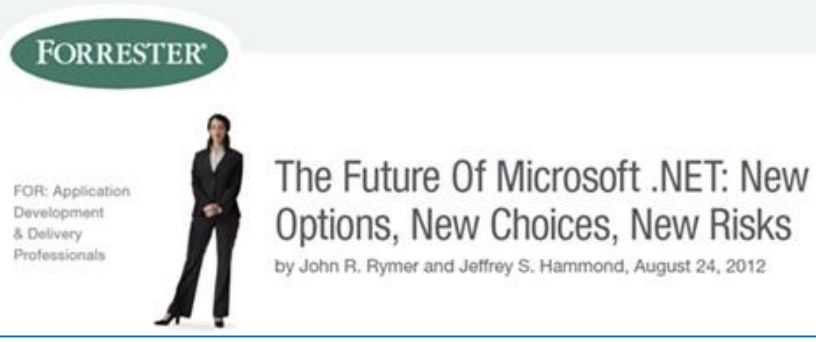


Figure 5-26

.NET in large, mission-critical and core-business applications

Over the years, .NET has evolved as a set of application development frameworks, focused on robustness, flexibility, and extensibility. A large base of Microsoft's customers has expressed confidence in .NET for large, mission-critical applications, and it also has recognition from analysts, as shown in the following excerpt from a recent Forrester report (2012).



AD&D leaders see .NET as Java's equal for large, complex applications. Windows Server, SLQ Server, and the .NET Framework were once challenging to scale, but not anymore. Enterprise AD&D leaders expressed great confidence in their ability to support their largest applications on Windows/SQL Server/.NET. The barrier to performance and scaling, they said, is poorly written application code, not the platform software itself.

".NET scales well. We've done load tests with pretty significant user populations. The platform gives us good options for caching and session management, and we like the exception handling and data access as well. You have some of these things in Java Spring, but in .NET they are better integrated." (VP of application development, financial services company)

Figure 5-27

Technology selection for large mission-critical and core-business applications

This guide does not cover mission-critical market-products themselves, like SAP, Dynamics ERP or other specific products, as it just focuses only on custom development.

> **Note on Data-Driven Development Technologies:** *Any large, mission-critical application can have many subsystems, and some of them could be collateral, non-core-business, and even simple data-driven areas. For simpler, data-centric subsystems you can use the data-driven technologies outlined in the small/medium business-applications section. This section does not cover those approaches/technologies, instead focusing on technologies for bounded contexts that implement the core-domain of your system.*

Any large, mission-critical application can use a wide range of technologies, starting from different presentation layer technologies, to domain model implementations ultimately based on O/RMs in many cases, use of cache, workflow, service bus, message queues, different types of data-sources (relational vs. NoSQL), etc. All in all, technology choices will depend on the specific scenarios and priorities that the application may have.

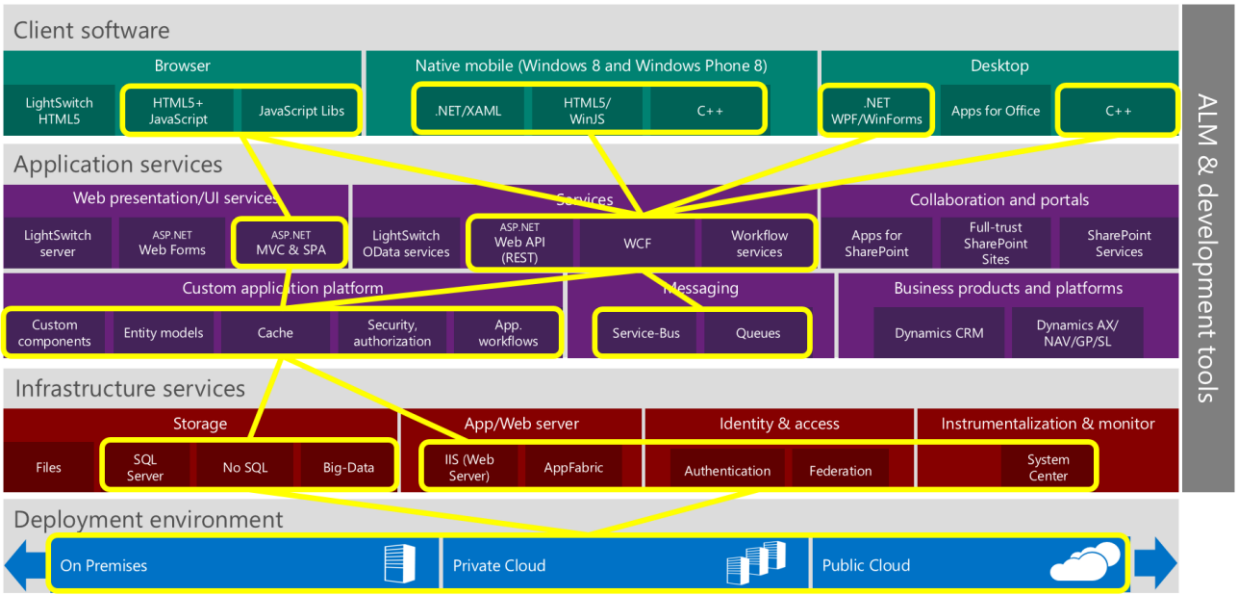


Figure 5-28

The following tables show and recommend when to use a given technology depending on your mission-critical application priorities and specific contexts.

Web UI technologies for core-business and mission-critical subsystems

Technologies	When to use and why
ASP.NET **MVC** with **HTML5** support	**ASP.NET MVC with HTML5 support** ▪ **Use it if** you want the most flexible and powerful option, **with full HTML rendering control**, better **Unit Testing** support, and **Faking** capabilities. ▪ HTML5: Use libraries like **Modernizr** for detecting HTML5 features support and workarounds. ▪ Use **CSS3** for less script and more maintainable code. ▪ Use **JavaScript** and **jQuery** for client-side programming. ▪ **Search engine optimizations** and **RESTFul URLs** are integral part of MVC.
ASP.NET **Single Page Application (SPA)**	**Single Page Application (SPA)** based on **ASP.NET MVC, HTLM5/JavaScript, Knockout,** and **Breeze** ▪ SPA is not a framework but a design pattern that can be implemented by using ASP.NET MVC and heavy use of JavaScript libraries like *Knockout* for supporting the MVVM pattern within JavaScript and like the *Breeze* JavaScript library for advanced data management handling. ▪ Use it for highly interactive web applications and smooth UI updates (minimum server page reload and visible round trips) and when including significant client-side interactions using HTML5, CSS3, and JavaScript. ▪ Take advantage of the open source *SignalR* JavaScript client library for bi-directional communication between the client and server. ▪ Consume ASP.NET Web API services from JavaScript.

Table 5-9

The web development for mission-critical business-applications must offer full-control and fine-grained technologies, so they can support loosely coupled architectures and create the best web UX. For that purpose, Microsoft provides ASP.NET MVC (full control in the server) and ASP.NET SPA (Single Page Application) templates, which make heavy use of JavaScript libraries like Knockout to support the MVVM pattern within JavaScript. There's also the Breeze library for advanced data management handling. There are other useful SPA JavaScript frameworks and libraries, like Durandal and RequireJS.

References

Project Silk: Client-side Web Development for Modern Browsers	http://silk.codeplex.com/ http://msdn.microsoft.com/en-us/library/hh396380.aspx http://www.amazon.com/Project-Silk-Client-Side-Development-Microsoft/dp/1621140105/
ASP.NET MVC	http://www.asp.net/mvc/
ASP.NET SPA (Single Page Application)	http://www.asp.net/single-page-application
Breeze/Knockout template	http://www.asp.net/single-page-application/overview/templates/breezeknockout-template
Durandal SPA framework	http://durandaljs.com/
RequireJS library	http://requirejs.org/
Modernizr: The feature detection library for HTML5/CSS3	http://www.modernizr.com

Desktop UI technologies for core-business and mission-critical subsystems

Technologies	When to use and why
.NET WPF	**.NET Windows Presentation Foundation** ▪ This is the preferred technology for Windows-based desktop applications that require UI complexity, styles customization, and graphics-intensive scenarios for the desktop. WPF also takes advantage of XAML views. WPF development skills are similar to Windows Store development skills, so migration from WPF to Windows Store apps is easier than migration from Windows Forms. ▪ Take advantage of new simplified asynchronous capabilities in .NET 4.5 (async/await). ▪ Take advantage of SignalR .NET client for bi-directional communication between the client and server.
.NET WinForms	**.NET Windows Forms** ▪ This was the first available UI technology in the .NET Framework for building desktop applications. It is a good fit for many business desktop applications. Windows Forms is easier to use and lighter than WPF for simple scenarios where you don't need UI styles customization. ▪ It provides simpler UI capabilities than WPF and it is not based on XAML, limiting its path forward to Windows Store Apps. ▪ Can also take advantage of the latest .NET capabilities such as asynchronous programming with async/await.
C++ (Win32 or MFC)	**C++ (Win32 or MFC)** ▪ Use it for building high-performing UI applications and very large and complex long-term products (such as products comparable to Microsoft Office). ▪ Use it for building the best and most performing graphics-intensive scenarios based on DirectX.

Table 5-10

References	
Introduction to WPF	http://msdn.microsoft.com/en-us/library/aa970268.aspx
Windows Presentation Foundation	http://msdn.microsoft.com/en-us/library/ms754130.aspx
Prism for WPF	http://msdn.microsoft.com/en-us/library/gg406140.aspx
Windows Forms	http://msdn.microsoft.com/en-us/library/dd30h2yb.aspx
C++ Win32 desktop applications	http://msdn.microsoft.com/en-us/library/vstudio/hh875053.aspx
C++ MFC desktop applications	http://msdn.microsoft.com/en-us/library/vstudio/d06h2x6e.aspx

Desktop applications are used often in large business applications, especially for heavy data-entry systems within existing systems. The recommended technologies for such applications are WPF (Windows Presentation Foundation), which provides a migration path to Windows Store business applications, and Windows Forms, which provides an easier to use and lighter solution than WPF for simple scenarios where you don't need UI styles customization.

Modern UI development technologies for native Windows Store applications
(Windows Store and Windows Phone apps/subsystems)

Modern touch-oriented applications (Windows Store applications for Windows 8 or Windows Phone) require innovative and compelling UX experiences, but the enterprise also requires that the UX support and align with their teams' preferences and skills. Microsoft offers a broad array of technologies and languages to create native Windows Store applications that support your development team's skills, including .NET/XAML, WinJS/HTML5, and C++/XAML.

To reduce redundant information, the recommendations on whether to use one or another technology are similar to the recommendations available in the "Emerging applications patterns" section.

Middle-tier technologies for Core-Business and Mission-critical Subsystems

Technologies	When to use and why
Services	**Service Orientation** Continuous services are fundamental for distributed enterprise applications. The priorities are performance and support of light inter-operable HTTP services, standards (REST, OData, SOAP, WS-*), and enterprise service requirements. Microsoft provides a broad range of technologies based on **ASP.NET Web API, WCF,** and **ASP.NET SignalR.**
Domain Model	**Domain-entity models, aggregates, domain-logic** The domain model is the core of the application, where the priorities include tackling complexity in large domains, creating the right design for long-term maintenance, and isolating the domain code (POCO entities) from infrastructure technologies. Microsoft provides mature domain-model approaches and loosely-coupled data-access technologies, like Entity Framework, LINQ, and ADO.NET, which can be used when applying *Domain-Driven-Design* patterns.
Composition and Loosely Coupled Architecture	**Loosely coupled architecture, composition, integration, business processes, workflow** Every enterprise application needs a backbone that addresses scenarios typically found in these large applications, such as loosely coupled architecture design based on dependency injection and IoC containers, long-running processes and workflows, event-driven subsystems integration, modern decoupled claim-based security, and transactions implementation. Microsoft provides proven technologies that cover those areas, such as inversion of control (IoC) containers (**Unity** and **MEF**), Windows Workflow Foundation (**WF**), Windows Identity Foundation (**WIF**), and .NET Framework **System**.**Transactions** APIs, Reactive Extensions (**Rx**), etc.

Table 5-11

References	
ASP.NET Web API	http://www.asp.net/web-api http://msdn.microsoft.com/en-us/library/hh833994(v=vs.108).aspx
WCF	http://msdn.microsoft.com/en-us/library/ms731082.aspx
Workflow Services	http://msdn.microsoft.com/en-us/library/dd456788.aspx
ASP.NET **SignalR**	http://signalr.net/
Unity	http://unity.codeplex.com/
MEF	http://msdn.microsoft.com/en-us/library/dd460648.aspx
System.Transactions	http://msdn.microsoft.com/en-us/library/system.transactions.aspx
Reactive Extensions (Rx)	http://msdn.microsoft.com/en-us/data/gg577609.aspx

Infrastructure technologies for core-business and mission-critical subsystems (BCs)

Technologies	When to use and why
Messaging	**Asynchronous messaging coordination** Composition, integration, and event-driven approaches using APIs (as explained above in the mid-tier technologies) need a solid foundation infrastructure in order to scale. The assets needed are related to asynchronous messaging infrastructure and message queuing infrastructure. Microsoft provides mature, solid infrastructure technologies such as **Windows Azure Service Bus, Windows Server Service Bus, Windows Azure Queues, Message Queuing (MSMQ),** and **Biztalk Server/Services.**
Security	**Identity, authentication, and authorization** Microsoft provides solid security infrastructure technologies such as Active Directory (AD), AD Federation Services (AD FS), Windows Azure AD, etc. The recommended security related development in modern applications is based on "claims-based security" and Windows Identity Foundation.
Cache	**Internal application data caching and Internet HTTP caching** Windows Server AppFabric Cache and Windows Azure Cache is an in-memory distributed cache. Can be used as a PaaS service external to your virtual machines/roles, or sharing a cache deployment between your own Windows Servers. **Windows Azure CDN** is an "Internet cache" capable to cache HTTP-accessed data, like video, files, etc.
Data Sources	**Relational SQL databases, NoSQL unstructured databases, and Big Data** Almost any kind of business application needs data sources in which to persist business data. Depending on the application's context and priorities, different data-source types should be used. The priorities revolve around relational data richness and transactional capabilities, data availability and scalability, and massive unstructured data approaches. Microsoft provides technologies that cover these priorities, such as **SQL Server, Windows Azure SQL Database, Windows Azure NoSQL tables** and **blobs,** and **HDInsight (Big Data, Hadoop)** on Windows Server or Windows Azure.
Deployment Infrastructure	**On-Premises, Cloud, and Hybrid-Cloud** All enterprise applications need a reliable infrastructure that ensures application availability, no matter how the context changes in the future. The priorities include consistent alternatives for on-premises or cloud choice, reliability, performance, availability, scalability, elasticity, hybrid IT, operations, and monitoring. Microsoft has the best-of-breed solutions for these requirements with infrastructure provided by **Windows Server, Windows Azure,** and **Microsoft System Center.** For further information on Windows Azure and hybrid-cloud, please refer to the "Emerging applications patterns" section.

Table 5-12

In complex and large, mission-critical application development, positioning a list of technologies is not enough. Comprehensive guidance on architectural and design approaches is needed to tackle the complexity involved with specific target Domain and Quality of Service requirements. The following sections in this guide introduce several important approaches to take into account in core-business and mission-critical applications, like Domain-Driven Design, CQRS, Event-Driven integrations, modernizing legacy core-business applications, and more.

Approaches and trends for long-term core-business applications

The following approaches, principles, and patterns are usually considered when building core-business and large, mission-critical applications (and even smaller applications, depending on your priorities):

- **Patterns of Enterprise Application Architecture** (such as those by Martin Fowler)
- **Domain Driven Design (DDD)** (such as the material provided by Eric Evans, Jimmy Nilsson, Vaughn Vernon, etc.)
- **CQRS (Command & Query Responsibility Segregation)** (such as "CQRS Journey" from Microsoft patterns and practices, and other sources provided by Greg Young, Udi Dahan, etc.)
- **Event sourcing** (such as the material provided by Greg Young)
- **Decoupled architectures based on the dependency inversion principle**
  - Use of Dependency Injection and Inversion of Control (IoC) containers
    - Microsoft patterns and practices; Unity, Ninject, Castle Windsor, Spring.NET, and so on
- **Service orientation** — REST Services, WS-*, Service Bus, elastic cloud services
- **S.O.L.I.D. principles** — http://en.wikipedia.org/wiki/SOLID_(object-oriented_design)
- **Behavior Driven Design (BDD) and TDD** — (i.e., Dan North, Chris Matts, etc.)
  BDD Frameworks: SpecFlow, NSpec, Cuke4Nuke, NBehave and MSpec, etc.

Loosely coupled architecture and the dependency-inversion principle

Loosely coupled architecture, along with dependency-inversion principles, techniques, and technologies, as well as S.O.L.I.D. object-oriented design, apply to many scenarios for core-business and mission-critical applications. It is worth noting that when developing **small/medium business-applications** these approaches are also useful, especially when the applications may grow in complexity and volume in the future. But **when developing large core-domain applications, these approaches are completely fundamental**.

The components of an application should be placed in distinct structured layers so you are compliant with principles like "separation of concerns" and get benefits like more efficient maintenance. It is also important to pay special attention to how the objects interact with each other; that is, how they are consumed and especially how some objects are instantiated from others.

Main 'loosely coupled objects' techniques are based on the dependency inversion principle (one of the principles in S.O.L.I.D.: http://en.wikipedia.org/wiki/SOLID_(object-oriented_design), where the traditional dependency relationship used in object orientation ("high-level layers should depend on low-level layers"), is inverted. With this approach, you reach an important goal, which is to have high-level objects independent from the infrastructure implementation, and therefore, independent from the underlying technologies.

The **dependency inversion principle** states the following:

A. *High-level layers should not depend on low-level layers. Both layers should depend on abstractions (contracts/interfaces).*

B. *Abstractions should not depend upon details. The details (implementation) should depend on abstractions (contracts/interfaces).*

The purpose of this principle is to decouple the high-level objects from the low-level objects so that it is possible to extend or change the final actions coordinated by a high-level class by swapping out the implementation of the interfaces where you set the high-level class dependencies, but, without making any change to the high-level class code. For example, being able to execute

the same code of a high-level application object (with no code changes) and consume different underlying infrastructure/technology objects that implement the same interfaces (abstractions) where you base your dependencies. This is also related to the Liskov substitution principle in S.O.L.I.D. (http://en.wikipedia.org/wiki/Liskov_substitution_principle).

The contracts/interfaces define the required behavior of the low-level components. These interfaces/contracts should exist in the high-level layers (that is, in DDD, you usually define most of the abstractions/interfaces within the Domain Model Layer, so the Domain Model Layer owns the contracts which are the specifications required to the infrastructure layers in order to work with your system). Those infrastructure layers could be swapped in the future following the Open/Close principle in S.O.L.I.D. (http://en.wikipedia.org/wiki/Open/closed_principle).

When the low-level components implement interfaces (that are placed in the high-level layers), this means that the low-level components/layers are the ones that depend on the high-level components. Thus, the traditional dependency relationship is inverted; that is the why it is called the **"dependency inversion" principle**.

There are several techniques and patterns used for this purpose:

- *Inversion of Control (IoC) containers*
- *Dependency Injection (DI)*

Inversion of Control (IoC) containers: This delegates the duty of selecting concrete implementations of our class dependencies to an external component or source (any concrete IoC container from any vendor). This pattern describes techniques to support a "plug-in" type architecture, where the objects can request for instances of other objects they require and on which they depend.

Dependency Injection (DI) pattern: This is actually a special case of IoC. It is a pattern where objects/dependencies are provided to a class instead of the class itself creating the objects/dependencies needed. The most common way to do that is providing the dependencies through the constructor of the high-level object.

IoC containers and dependency injection add flexibility, comprehension, and maintainability to the project and will result in changing the actual code as little as possible. Instead, it involved trying to add code only in new implementation/classes as the project goes forward.

DI allows altering the behavior of a class without changing the internal code of the class by just changing the dependencies implementations. This is related to the open/close principle in S.O.L.I.D. (http://en.wikipedia.org/wiki/Open/closed_principle). The object instance injection techniques are "interface injection," "constructor injection," "property injection (setter)," and "injection of method calls."

Dependency injection as a way for architectural quality

DI and IoC are used to improve the architectural qualities of an object-oriented system by making it easier to utilize good design techniques. DI and IoC enable loose coupling between classes and their dependencies, improve the testability of a class, and provide generic flexibility mechanisms. They can also greatly enhance the opportunities for code reuse by minimizing direct coupling between classes and configuration mechanisms.

The **single responsibility principle** in S.O.L.I.D. states that each object should have a unique responsibility (see http://en.wikipedia.org/wiki/Single_responsibility_principle). It establishes that one responsibility is one reason to change code (and code changes potentially introduce bugs) and concludes by saying that a class must have one—and only one—reason to change. This principle is widely accepted by the industry and favors designing and developing small classes with only one responsibility. This is directly connected to the number of dependencies; that is, objects that each class depends on. If one class has one responsibility, its methods will normally have few dependencies with other objects in its execution. If there is one class with many dependencies (for example, 15 dependencies), this indicates what is commonly known as "bad smell" of the code. In fact, by doing dependency injection in the constructor, you are forced to declare all the object dependencies in the constructor. In this example, you would clearly see that this class in particular does not seem to follow the single responsibility principle, because it is unusual for a class with one single responsibility to declare 15 dependencies in the constructor. Therefore, DI also serves as a

guide for us to achieve good designs and implementations, and it offers a decoupling approach that you can use to inject different implementations clearly.

Additionally, the Dependency Injection and Inversion of Control containers are not designed to only promote unit testing, or fakes/mocks. To say they were would be like saying that the main goal of interfaces is to enable testing. DI and IoC are about decoupling, more flexibility, and having a central place that enables maintainability of our applications. Testing is important, but it is not the first reason or the most important reason to use the Dependency Injection and IoC containers.

DI/IoC containers

DI and IoC can be implemented with different technologies and frameworks from different vendors or sources. The following table lists a few frameworks, but you can use any implementation you want in a very similar way. The important point here is to understand and apply the right patterns and principles.

IoC (Inversion of Control) containers—frameworks

Technologies	When to use and why
Microsoft Unity	**Microsoft patterns & practices** ▪ This is currently the most complete lightweight Microsoft framework to implement IoC and DI. ▪ It is an open-source project with Microsoft Public License (Ms-PL) licensing. **Extended info:** http://msdn.com/unity ; http://unity.codeplex.com/
Castle Project (Castle Windsor)	**Castle Stronghold** ▪ Castle is an Open Source project. It is one of the most used IoC frameworks. **Extended info:** http://www.castleproject.org/
Microsoft MEF	**Managed Extensibility Framework** ▪ MEF is a part of the Microsoft .NET Framework. It is a framework for automatic extensibility of tools and applications, focusing more on UI extensibility and plug-ins. ▪ MEF cannot be considered as a traditional IoC container; instead it must be considered a framework for dynamic and automatic extensibility. But some of its features overlap to a typical IoC container. ▪ It is an open-source project with Microsoft Public License (Ms-PL) licensing. **Extended info:** http://msdn.microsoft.com/en-us/library/dd460648.aspx ; http://mef.codeplex.com/
Ninject	**By Nate Kohari** ▪ Ninject is a dependency injector that makes code easier to understand, easier to change, and less error-prone. It is an open-source project. **Extended info:** http://www.ninject.org/
Spring.NET	**SpringSource (a division of VMware)** ▪ Spring.NET is a framework for building enterprise applications. It supports IoC capacities but also AOP (Aspect Oriented Programming) capabilities. ▪ It is an open-source project. **Extended info:** http://www.springframework.net/
StructureMap	**By several developers of the .NET OSS community** ▪ StructureMap is the oldest IoC/DI tool for .NET, available since June 2004. ▪ It is an open-source project. **Extended info:** http://structuremap.sourceforge.net/Default.htm
Autofac	**By several developers of the .NET OSS community** ▪ Autofac manages the dependencies between classes so that applications stay easy to change as they grow in size and complexity. ▪ It is an open-source project. **Extended info:** http://code.google.com/p/autofac/
LinFu	**By Philip Laureano**

	▪ The LinFu Framework is a set of libraries that extend the CLR by providing IoC, AOP, DbC, and other features. ▪ It is an open-source project. **Extended info:** https://github.com/philiplaureano/LinFu
Funq	**By Daniel Cazzulino and other developers from the .NET OSS community** ▪ Funq is a high-performance DI framework that eliminates all runtime reflection through the use of lambdas and generic functions as factories ▪ It is an open-source project. **Extended info:** http://funq.codeplex.com/

Table 5-13

Most of the IoC/DI containers also support the Interception pattern. This pattern introduces another level of indirection. This technique places an object (a proxy) between the client and the real object. The behavior of the client is the same as if it interacted directly with the real object, but the proxy intercepts it and solves its execution by collaborating with the real object and other objects as required. Interception can be used as a way to implement AOP (Aspect Oriented Programming).

Dependency injection for any architectural style

Finally, it is important to highlight that a decoupled design using DI/IoC is also beneficial when developing applications of almost any type and of almost any architectural approach. Of course, it fits perfectly into architectures that must be decoupled because of their original design such as Domain-driven Design (DDD) and Command Query Responsible Segregation (CQRS). However, it also works great in simpler architectural styles like 2-Layered and CRUD approaches (for instance, using a simpler approach based just on ASP.NET MVC and 2 layers). If you apply DI/IoC and the related principles in those simpler contexts, you will also get similar benefits. On the other hand, there's no doubt that the larger and more complex an application can be, the more benefits you will get from decoupled architectures.

Architectural styles for core-business applications

There are many architectural styles used by software architects and developers. The following are a few typical approaches that can be implemented with **.NET Framework:**

- **Traditional Layered architecture** (logical top-down layers)
- **Domain-Driven Design Layered architecture** (logical layers where the heart is the domain layer)
- **CQRS architectural approach** (logical layers and physical tiers)
- **3-Tier architecture** (physical tiers)
- **N-Tier architecture** (physical tiers)
- **Service-Oriented Architecture (SOA)** (higher level of orchestration)
- **Event-Driven Architecture (EDA)** (higher level of orchestration)

Many of them are complementary, because you can deploy a layered, logical architecture by following a specific N-Tier physical architecture. You can also have a service-oriented architecture (SOA) or high-level Event Driven Architecture (EDA) that orchestrates many services that are internally designed as a layered architecture, etc.

The important point is that no particular architecture is right for all situations. Even more, **when dealing with large applications, you shouldn't usually apply a single top-level architecture approach.** On the other hand, you should design large composite applications composed by many subsystems (or bounded contexts in Domain-Driven Design lingo), where the right approach will be to apply different architecture styles based on the subsystems' nature and priorities. You could even decide that for certain collateral subsystems or bounded contexts within your large application, the right approach is a simple data-driven and CRUD approach; and only for your core-domain (core-business), you might need to apply a more advanced and decoupled architecture like domain-driven approaches and patterns, as illustrated below in Figure 5-29.

Subsystems/Bounded Contexts with Different Architectures

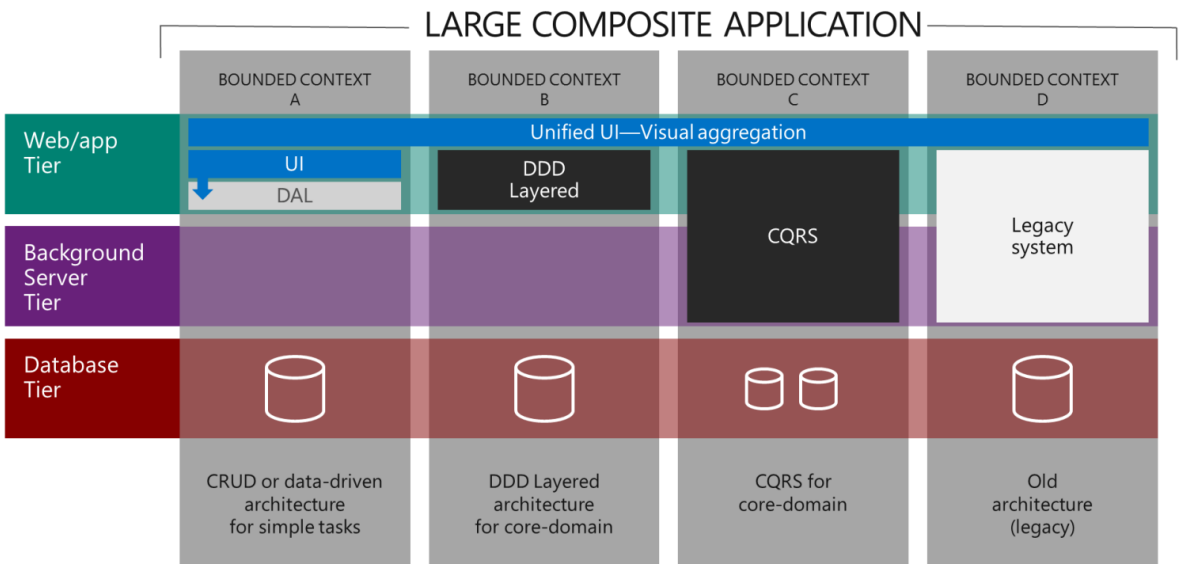


Figure 5-29

A **bounded context (BC)** is more precise than the subsystem term (although sometimes they could coincide). The **original definition** in **Domain-Driven Design** is, *"The delimited applicability of a particular model. Bounded Contexts gives team members a clear and shared understanding of what has to be consistent and what can develop independently"* (Eric Evans definition), where an important characteristic is that each bounded context has its own domain model, and in many cases even its own database schema and database.

There is no "silver bullet" or a unique right architecture for any given case. You cannot have "a single architecture to rule all the applications or all the subsystems or bounded contexts." Depending on the priorities of each subsystem (bounded context), you must choose a different approach. It will be a disaster if you use a very simple data-driven approach (CRUD and RAD) for an application that will be evolving and growing its business logic for many years, because eventually the application will get converted into a "big ball of mud," and that application will be unmaintainable in the long term. However, the opposite is also true. For instance, it doesn't make sense to build a very simple data-driven table maintenance application implementing a very advanced and "state of the art" architecture, just because you want to use the same "standard approach" for all of your developments. It will be very costly and comparable to "building a bicycle with rocket parts!" Therefore, the right approach for large business applications is "divide and conquer!"

In summary, **approaches and architectural styles should be applied to certain areas within large applications, rather than as top-level prescriptive architectures.**

Then, of course, the focus now moves to these questions: "How do I identify those subsystems or bounded contexts?" and even further, "How do I integrate those bounded contexts?" We address many of these questions in this guide, but for further detailed information about common Domain-Driven Design content and other related approaches, please refer to the following references.

References	
Domain-Driven Design Community and Content	http://dddcommunity.org/uncategorized/bounded-context/
	http://dddcommunity.org
	http://dddcommunity.org/books/
	http://www.agilification.com/file.axd?file=2010/8/CQRS+DDD+Notes+Greg+Young.pdf
Snowman Architecture: Vertically Aligned Synergistically Partitioned (VASP) Architecture (Comparable concept to Bounded-Context) By Roger Sessions	http://simplearchitectures.blogspot.com/2012/12/the-snowman-architecture-part-three.html
Communicating Between Bounded Contexts (Microsoft patters & practices CQRS Journey)	http://msdn.microsoft.com/en-us/library/jj591572.aspx
Entity Framework Domain-Models in DDD Bounded Contexts (Bounded-Context-View rather than pure decoupled DDD BC)	http://msdn.microsoft.com/en-us/magazine/jj883952.aspx

Modernizing mission-critical enterprise applications

When creating new, large mission-critical applications, current scalable and elastic approaches should be applied. However, most existing enterprise applications were not built with asynchronous engines and event-driven approaches, and they don't support scale-out and elastic architectures. In many cases, it is better to treat these applications as legacy systems to be integrated into updated (web-based or service-based) façades. This integration should be done while building a new asynchronous message-oriented integration system as a cross-channel organizer. The changes and updates to the established applications should focus on areas related to innovation (as in modern applications) or areas related to scalability and performance. However, when dealing with legacy systems, scalability can be achieved more easily by building front-end services and caches in the cloud (those previously mentioned façades) instead of rebuilding the whole mission-critical application from scratch. Then, in parallel, you can re-architect the internal systems behind the façades as budget and time allow.

This modernization is precisely one of the scenarios this guide focuses in the next sections.

Scenarios for custom large, mission-critical applications

The following sections discuss typical scenarios for custom large mission-critical applications:

- **Scenario: Domain-Driven Subsystem (Bounded Context)**
- **Scenario: CQRS Subsystem(Bounded Context)**
- **Scenario: Communicating Different Bounded Contexts**
- **Scenario: Modernizing Legacy Mission-Critical Enterprise Applications**

Scenario: Domain-Driven Subsystem (Bounded Context)

This scenario covers complex core-business subsystems. In this case, the word "complex" means systems that have a high volume of ever-changing domain/business logic that will be evolving for years.

While their presentation layer can vary (web, desktop, or modern mobile devices), the major volume of investment, design, and development is in the server/services side where the business/domain logic is usually placed.

This type of application (or subsystem) is characterized by having a relatively long life span and may need to absorb a considerable amount of evolutionary changes. Ongoing maintenance critical in these applications, so one of the most important priorities is to

invest on architecture, design, and development principles that will ease maintenance and changes in the long term. If the application grows substantially, you want to ensure the minimum possible impact to the actual code. This is why monolithic approaches are the worst approach for these types of applications.

As mentioned earlier, in complex applications, the behavior of business rules (domain logic) is often subject to change, so the ability to modify, build, and perform isolated tests (unit testing and mocking) on domain logic layers in an easy and independent way is critical for a sustainable maintenance. Achieving this important goal requires minimal coupling between the domain model (entity model including logic and business rules) and the rest of the system's layers (presentation layers, infrastructure layers, data persistence, and so on). For that purpose, persistence-ignorant models and decoupled designs based on abstractions are usually required in this context.

Additionally, because these long-term applications will usually have a very long life (several years at least), eventually, the technologies available will evolve or be replaced. Therefore, an important goal for long-term systems is to be able to adapt to infrastructure/technological changes with minimal impact to the rest of the application (domain model, domain rules, data structures communicated to other tiers [DTOs], etc.) Changes in infrastructure technology of an application (technologies for data access, service bus, security, operations, instrumentation, etc.) should have a minimal impact on higher-level layers in the application, particularly on the application domain model layer.

The trends in long-term application architectures are increasingly oriented to promote this decoupling between classes/objects and also between vertical areas (bounded contexts). **Domain-Driven Design (DDD)** is a prime example of this trend as an approach to developing complex functional software systems.

The DDD approach uses a set of techniques to analyze your domain and to construct a conceptual model that captures the results of that analysis. You can then use that model as the basis of your solution. The analysis and model in the DDD approach are especially well suited to building solutions for large and complex domains. DDD also extends its influence to other aspects of the software development process to help manage complexity.

The core principles in Domain-Driven Design are:

- **Focus** on the **core domain**
- **Direct collaboration** between **"domain experts"** and the **development team**
- Speak an **ubiquitous language** within an explicit bounded context (subsystem)
- Apply certain DDD **architecture and proven design patterns**

Therefore, DDD is much more than just a proposed architecture; it is a way of managing projects, team-work to identify an **"ubiquitous language"** based on **"knowledge crunching."** This leads to **continual interaction between the development team with domain experts** to capture the important parts of the domain in the model, the ongoing search for better abstractions, refactoring, and improvement of the model. Developers' knowledge of the domain grows over time, along with the domain experts' ability to formalize or distill their domain knowledge.

When designing DDD logical architectures, a complex bounded context should be partitioned into layers. This design must be cohesive within each layer, but clearly must define the different layers within the system. Dependencies between objects should be based on abstractions. **In addition to clearly delineate and isolate the domain model layer from the rest of the layers, like the infrastructure layers based on specific technologies** (like data-access technologies). Everything must revolve around the domain, because **"*the domain layer is the heart of the software*"** in core-domain applications.

Hence, within a concrete bounded context, all the code related to the domain model (e.g. domain entities and domain services) should be placed in a single layer. The domain model should not persist/save itself, nor should it have direct dependencies to underlying technologies. It must exclusively focus on expressing the domain model (data and logic). The .**NET Framework** and **Entity Framework** ease this model implementation significantly when using POCO Code-First entities which are "persistent ignorant".

This layering is similar to Figure 5-30 (following the trends of DDD architectural styles), where you could have additional bounded contexts with other different architectural approaches:

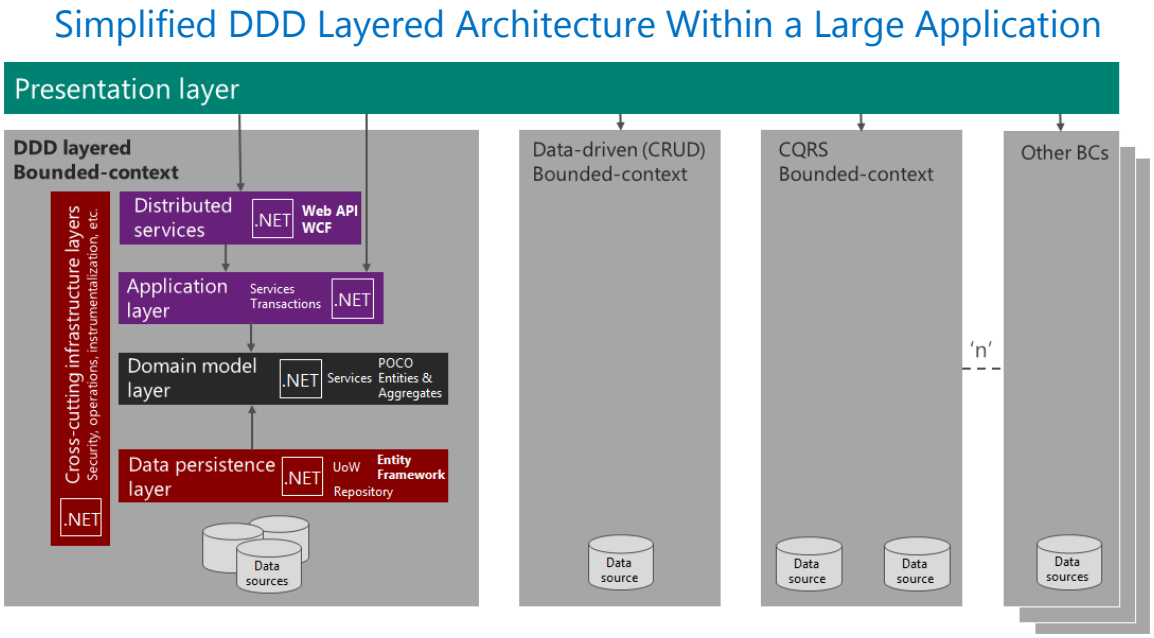


Figure 5-30

In this figure, the arrows represent dependencies between the different components and layers. The main difference between a DDD layered approach and a traditional top-down layered approach is that the domain model layer has no dependency to any other layer. This approach is based on abstractions (interfaces defined within the domain model layer and the related implementation classes are placed in other layers) and POCO entity models (like using **Microsoft Entity Framework** POCO entities and Code-First approach). That is why the domain model layer complies with the *persistence ignorant principle* regarding the data-access technologies you may use.

You should apply advanced approaches (like DDD or CQRS) only within the bounded contexts that focus on the core-domain of the application. Other collateral and simpler bounded contexts or subsystems may be implemented following data-driven and CRUD approaches, as shown in Figure 5-30.

The logical architecture shown in Figure 5-30 is the internal architectural approach. You may have top-level architectures orchestrating different services and other specific physical and infrastructure architectures (tiers and elastic services in private or public clouds, and so on) where you deploy those services, as shown in Figure 5-31 and Figure 5-32.

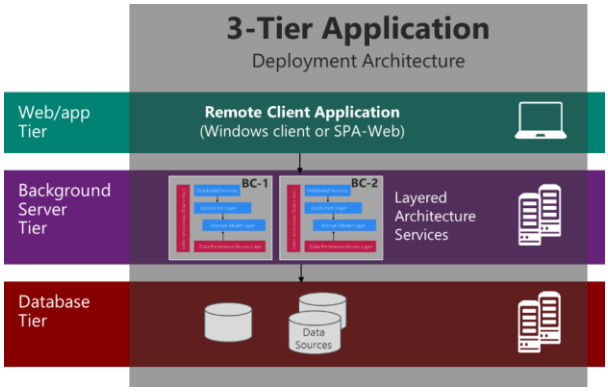


Figure 5-31

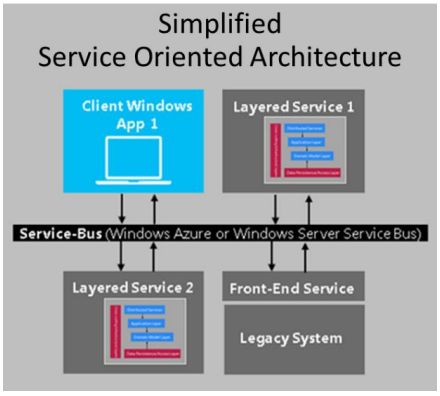


Figure 5-32

The services used by a 3-tier application can be composed of a number of bounded contexts. Each bounded context could have a unique data source, or they could share the same sources. It's important that each bounded context has its own domain entity model, which usually means its own database schema.

In SOA, a service could match a bounded context, but it is not mandatory. A bounded context could also be composed by several services, or vice versa—a large service could be composed of several bounded contexts.

Bounded contexts and domain models

Bounded contexts deal with different aspects of the domain that represents a singular real-world (domain) concept. Bounded contexts are autonomous components, with their own domain models and their own ubiquitous language (concrete domain terms). They should not have any dependencies on each other at runtime and should be capable of running in isolation.

Bounded contexts give team members a clear and shared understanding of what has to be consistent and what can develop independently. Therefore**, for each bounded context, you need to define a boundary around the delimited applicability and consistency of a particular model.** Because of this, each bounded context usually has its own ubiquitous language that impacts its own model, so when you are talking about similar concepts and entities in several bounded contexts, they probably will have different attributes and potentially even different names and terms.

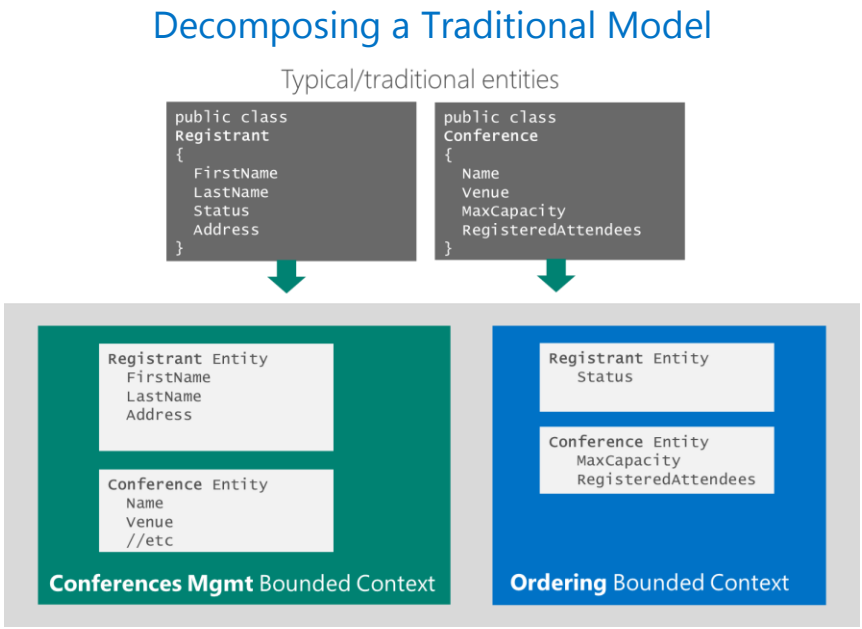


Figure 5-33

The pseudo entity diagram in Figure 5-33 shows a possible approach for a decomposition of traditional "upfront entities" (anti-pattern) into several bounded contexts (BCs). We must have a different model for each bounded context, but in many cases (like Figure 5-33), you have the same "concept" of entity in several models, using the same entity name, but probably with some different attributes.

In that example, we are splitting entities based on what attributes have to be consistent between them and impact to each other. For instance, the *Registrant* entity class in a hypothetical *ConferencesManagementBC* might have personal data like name, address, company data, and so on. But it is likely that the "Status" attribute makes more sense within the *Registrant* entity class in a different BC called, for instance, *OrderingBC*, because that attribute must be consistent with the conference *RegisteredAttendees* attribute. Therefore, BC boundaries must also be identified based on consistency between the data and which attributes impact other attributes and which ones don't impact at all. For example, if you change the registrant's *FirstName*, that won't affect the conference registration attributes.

Sometimes you'll need to have redundant attributes in different bounded contexts (try to minimize that). In those cases, you'll have to deal and embrace "eventual consistency". You also use eventual consistency between different Aggregates (consistent set of entities).

Also, taking a look at the required relationships between entities (grouping them in aggregates) is a good way to identify BCs. If some area of the initial model is really isolated from other areas in the potential application, that area should probably be a bounded context. Figure 5-34 illustrates this process.

Each bounded context and domain model is a different vision of the same reality (the domain), but for a different use and purpose.

In order to understand it, there is a famous metaphor from Eric Evans saying that the world (the reality) is the domain and any kind of world-map is a different BC with its own domain model.

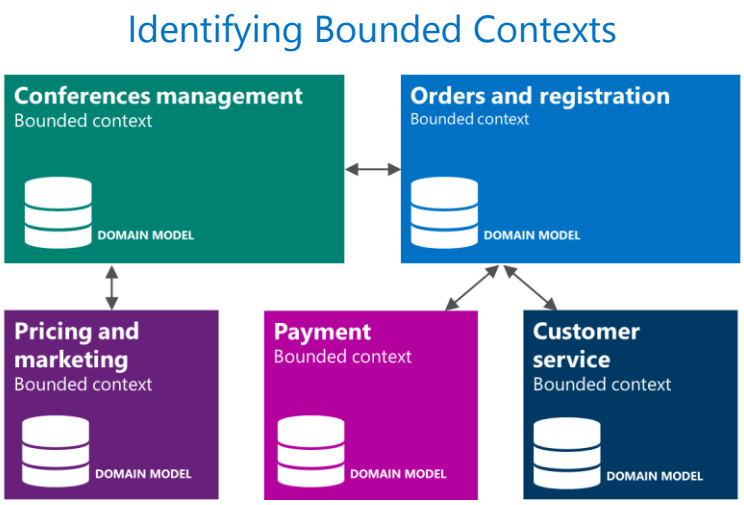


Figure 5-34

When designing and implementing entities, it is also fundamental not to have an "**anemic-domain model**" (anti-pattern) in DDD. Entity classes should have their own domain logic within the entity-class methods.

The implementation of Domain Model Entities using Microsoft development technology can be achieved through different ways, though the way recommended by Microsoft is by using POCO (Plain Old CLR Objects) entities (plain classes) that must comply with the **"Persistence ignorant"** principle (no dependencies from the entities themselves to infrastructure technology). Then, the model must be loosely-coupled to infrastructure data persistence technologies like **Microsoft Entity Framework** and **Code-First approach mappings**. Other alternatives include different O/RMs, like NHibernate, or custom approaches "from scratch" based on plain ADO.NET. But, in any case, the final dependencies (entities and repository contracts/interfaces) have to be within the model rather than in the infrastructure technologies. That is a crucial difference between DDD-layered architectures compared to regular layered architectures.

Implementing Domain-Driven Design patterns and other features with .NET

When applying Domain-Driven Design, each layer will usually implement different patterns like *Domain Entity, Aggregate, Aggregate-Root,* loose coupling between aggregates, *Aggregate storage, Value-Object, Repository, Unit of Work, Domain Service, Factory, Domain events,* eventual consistency and so on. (For detailed explanations of these patterns, please refer to the DDD references cited below.)

Figure 5-35 shows a suggested **implementation path of typical DDD patterns and infrastructure features** using **Microsoft technologies**.

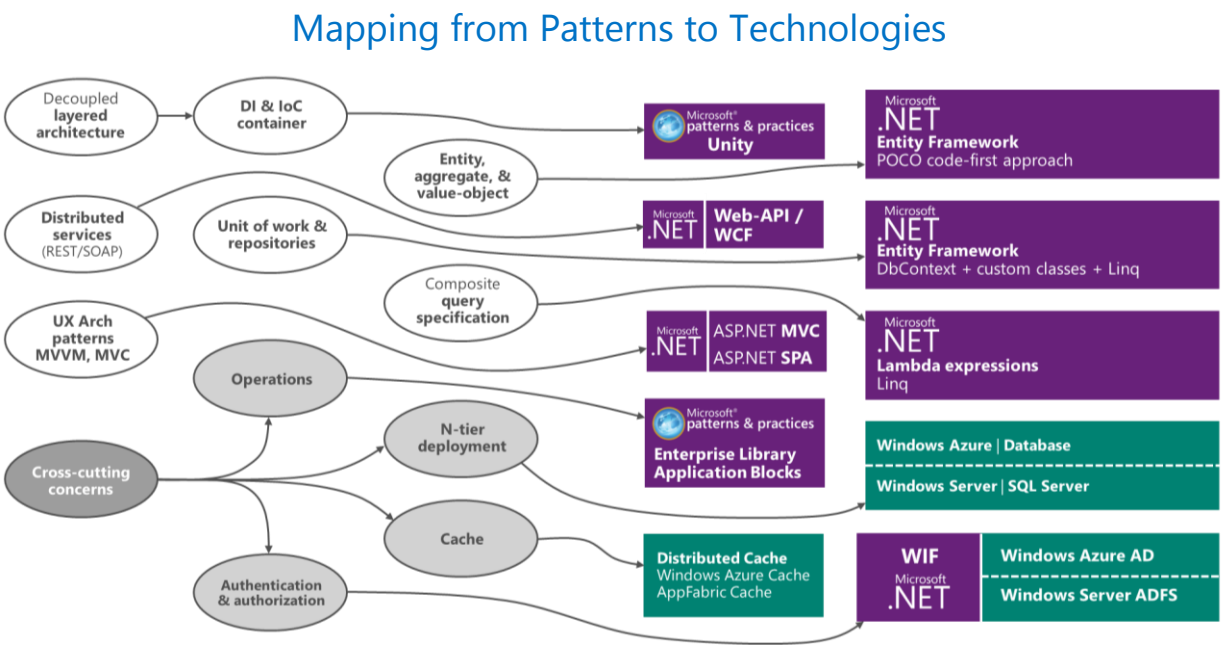


Figure 5-35

When using relational databases and implementing DDD patterns using .NET, the *Domain-Entity, Aggregate, Aggregate-Root, Repository and Unit of Work* patterns can be easily implemented relaying those patterns on any O/RM like **Microsoft Entity Framework** or any other O/RM. But, it depends on your application priorities and requirements. Other times, an *Aggregate storage* based on a *NoSQL* or *document databases* (non-relational, so no O/RM usage) is a better approach (e.g. for *Event-Sourcing* and aggregate events storage).

There is a wealth of resources regarding cross-cutting concerns, operations, security, and instrumentation subjects on MSDN.com and Microsoft Patterns and Practices documentation if you'd like to learn more.

References

The Anemic Domain Model	http://www.martinfowler.com/bliki/AnemicDomainModel.html
POCO Entity classes (POCO: Plain Old CLR Objects)	http://en.wikipedia.org/wiki/Plain_Old_CLR_Object
Employing the Domain Model Pattern	http://msdn.microsoft.com/en-us/magazine/ee236415.aspx
Entity Framework	http://msdn.microsoft.com/en-us/library/jj591559.aspx
Event Sourcing	http://codebetter.com/gregyoung/2010/02/20/why-use-event-sourcing/

Scenario: CQRS Subsystem (Bounded Context)

This scenario covers core-business subsystems based on CQRS (Command and Query Responsibility Segregation). CQRS is usually related to DDD, in particular, with contexts where users are competing for application resources and when the application requires high scalability (common in mission-critical applications, especially when facing an unlimited number of Internet users).

Betrand Meyer introduced the term "Command Query Separation" to describe the principle that an object's methods should be either commands or queries. A query returns data and does not alter the state of the object; a command changes the state of an object but does not return any data. The benefit is that you have a better understanding what does, and what does not, change the state in your system.

CQRS takes this principle a step further to define a simple pattern.

> *"CQRS is simply the creation of two objects where there was previously only one. The separation occurs based upon whether the methods are a command or a query (the same definition that is used by Meyer in Command and Query Separation: a command is any method that mutates state and a query is any method that returns a value)."*
> —Greg Young—

Figure 5-35 shows a typical application of the CQRS pattern to a portion (BC) of an enterprise system. This approach shows how, when you apply the CQRS pattern, you can separate the read and write sides in this portion of the system. **CQRS can be a good fit for many Windows Azure cloud scenarios.**

CQRS Approach Within a Large Application

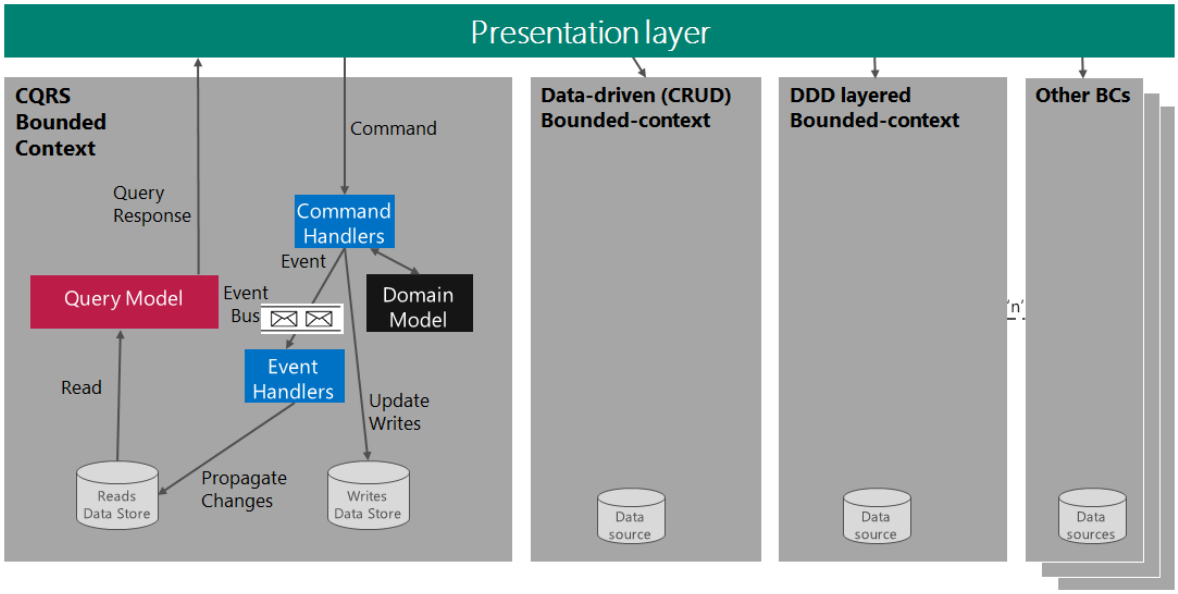


Figure 5-36

References	
CQRS Journey guidance – by Microsoft patterns & practices	http://msdn.microsoft.com/en-us/library/jj554200.aspx
CQRS on Windows Azure	http://msdn.microsoft.com/en-us/magazine/gg983487.aspx
Greg Young	http://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf
Udi Dahan	http://www.udidahan.com/2009/12/09/clarified-cqrs/

Scenario: Communicating Different Bounded Contexts

Bounded contexts should not have dependencies on each other at runtime—they should be capable of running in isolation. A large/complex application can have any number of interrelated bounded contexts, **each one with its own domain model,** as shown in Figure 5-37. In a large application, there can be dozens of bounded contexts.

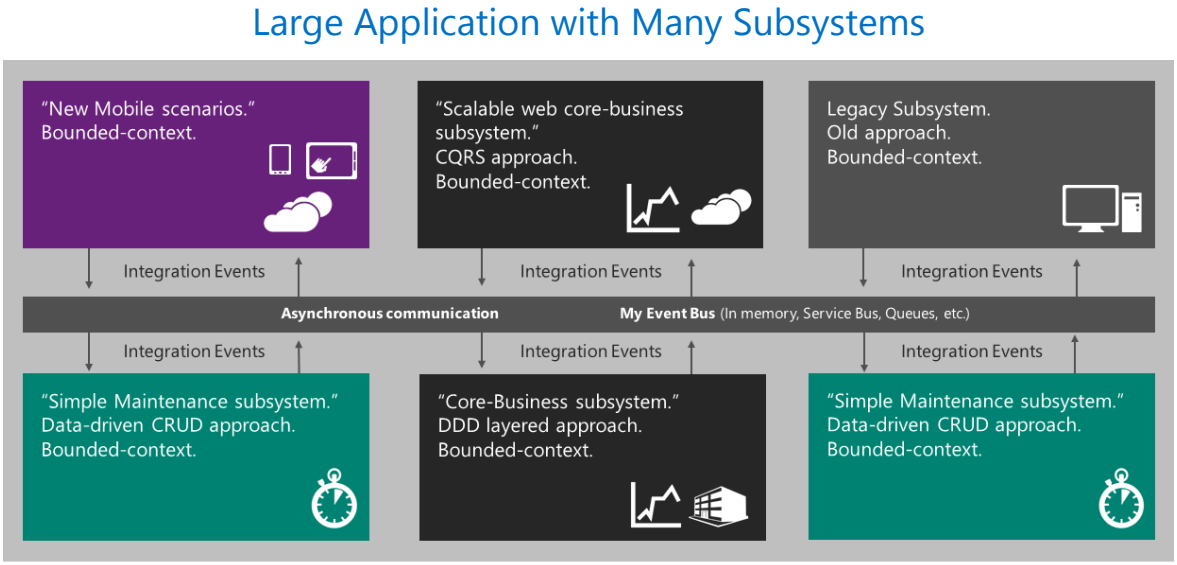


Figure 5-37

However, BCs are a part of the same overall system or larger application and they need to exchange data with one another. Even when they should be isolated from other BCs, information changes, and events that happen in one BC will impact others. Use asynchronous communication base on events for this type of inter-BC communication. The bounded context can respond to events that are raised outside of the bounded context. Bounded context can publish events that other bounded contexts may subscribe to. Events (one-way, asynchronous messages that publish information about something that has already happened) enable you to maintain the loose coupling between your bounded contexts. These events can be referred to as integration events to indicate an event that crosses bounded contexts.

When using integration events and asynchronous communication between bounded contexts, it doesn't necessarily mean a SOA or using a service bus. This communication could also be based on any type of event-bus that can be implemented using different infrastructure technologies. Possibilities for an **event-bus** implementation include **a simple in-memory bus,** an inter-process communication event bus using named-pipes (within the same server, if that is enough for our objectives or for Proof of Concept purposes), message queues like MSMQ **(Microsoft Message Queuing),** or going for a full service bus like **Windows Server Service Bus** or **Windows Azure Service Bus** (or a third party service bus, like *NServiceBus*) for production systems.

The way of dealing with BCs integration events is almost identical to the way you deal with events in CQRS. The events (simple message of any type) must be created after something has occurred in the domain storage (e.g. an update in the domain storage). The event will be named in past tense and will be published into the event bus. It is fundamental that the event bus must support a publish/subscription mechanism, so after publishing a message event in the event bus, any BC who was subscribed to the event bus will get that event and perform an action in order to propagate it to its own domain model. Based on that approach, you will have ***eventual consistency*** **between all the bounded contexts.**

Context maps

A large system, with dozens of bounded contexts and hundreds of different integration event types, can be difficult to understand. Context maps are valuable records of which bounded contexts publish which integration events, and which bounded contexts subscribe to which integration events.

Additionally, once you get to a large number of bounded contexts or integration event types, a service bus capable of publish/subscription mechanisms becomes very convenient.

Relationships between contexts

The different relationships that exist between two or more contexts depend greatly on the degree of communication between the various teams in each context and the degree of control you have over them. For example, you may not be able to perform changes in certain contexts, such as in the case of a system in production or a discontinued system.

Shared Kernel

This approach fits better for 'Green field' scenarios, where you are actually building an end-to-end application. In this case, when you have two or more contexts where the teams who are working on them can communicate fluently, it is interesting to establish a shared responsibility for certain objects that all the contexts use in common. An example might be certain shared areas of the same Domain Model shared by two or more bounded contexts. These objects become what is known as a shared kernel for all the contexts. In order to make a change on any object of the shared kernel, it is necessary to have the approval of all the teams who own the contexts involved. Promoting a good communication between the different teams is critical as they are sharing the same Domain model area (although those bounded contexts can also have their own different domain models, additionally).

Scenario: Modernizing Legacy Mission-Critical Enterprise Applications

This scenario is very important in large mission-critical systems, as the systems have a very long, evolving life. There can be many legacy and/or critical applications in the enterprise that need to be modernized. Usually the first step should be modernizing the external face of those systems or parts of it. Adding a **Microsoft Windows Store mobile application (Windows 8 or Windows Phone)** opens the application to *new experiences and scenarios*. A good example is a Host Application or a large legacy ERP system—it often isn't possible to migrate or modernize the whole system.

To modernize legacy systems, start creating **service façades** that wrap those legacy systems and expose them to new scalable systems like persistent caches in the cloud in order to scale out and consume services from many new mobile clients. This approach moves limited scalability in a legacy system to hypothetically unlimited scalability in cloud systems for new channels and users (e.g. direct customer access).

However, in a long-term, large, core-business application there might be new bounded contexts (subsystems) that need to be consistent with the older legacy systems. Here too, because of consistency required, it's necessary to communicate and integrate new bounded contexts with older legacy subsystems. This requires certain approaches like the anti-corruption layer.

The Anti-Corruption Layer

This is probably the most important pattern regarding bounded contexts integration, especially when faced with teams that are poorly communicating. This pattern is also important if one of the bounded contexts to integrate is a legacy bounded context, which can have a different ubiquitous language (and therefore completely different models), and you do not want that to affect your newer bounded contexts. For this type of situation you can implement an **anti-corruption layer**, an intermediate layer between the different contexts that performs the required translations (related to model objects and also to integration events).

Modernizing Established Systems Using the **Anti-Corruption Layer**

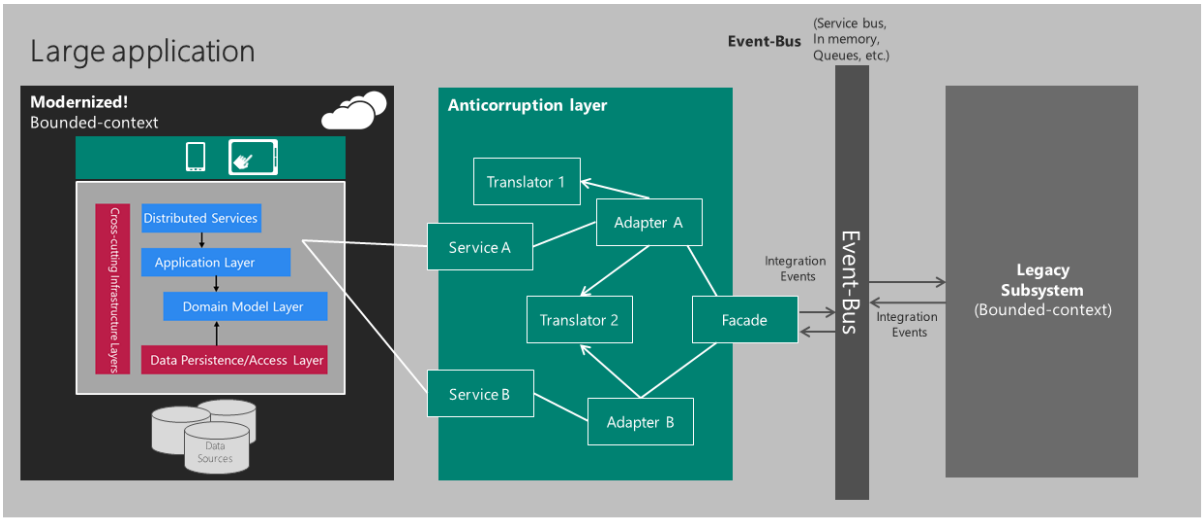


Figure 5-38

An anti-corruption layer consists of three types of components*: adapters, translators*, and *façades*. First, a façade is designed to simplify the communication with the other context and exposes the minimum functionality as possible. It is important to understand that the façade should be defined in terms of the other context's model elements; otherwise you would be mixing the translation with the access to the other system. After designing a façade, an adapter is placed to modify the interface/events of the other context and adapt it to the interface expected by our context. Finally, you use a translator to map the elements of our context to the expected elements of the other context façade (or vice versa).

The anti-corruption layer is also critical for managing the integration events. Bounded contexts are independent of each other and may be modified or updated independently. Such modifications may result in changes to the events that a bounded context publishes. These changes might include introducing a new event, dropping the use of an event, renaming an event, or changing the definition of event by adding or removing information in the payload. A bounded context must be robust in the face of changes that might be made to another bounded context. The anti-corruption layer is responsible for verifying that incoming integration events make sense. You can also use the anti-corruption layer to translate incoming integration events.

The implementation of an anti-corruption layer can be custom (plain .NET). This is the initial approach. For more complex contexts, or when the legacy subsystem to be extended is a market-product (like SAP or any other ERP, etc.), the integration and translations can be continuously evolving and become very complex. In this situation, you might consider using a specific integration platform (like *Biztalk Server*)—which offers more than 25 adapters linking to market-packages and a robust messaging infrastructure—on top of Windows Server or Windows Azure (IaaS), and can also link to Windows Azure Service Bus.

Conclusions

There are too many combinations of approaches, technologies, architectures, and patterns to apply in business applications. This guidance tries to position a number of approaches based on mainstream application priorities, but it should not be taken as rigid, prescriptive guidance precisely because of the large number of contexts in business applications. There may be logical exceptions to this guidance, like small/medium business-applications that are not very data-driven but should be more domain-driven due to ever-changing domain rules. Or vice versa—collateral subsystems or bounded contexts that are clearly following a data-driven approach though they might be a part of a large mission-critical application.

The main goal of this guide is to position Microsoft technologies in different approaches driven by application and business priorities. How you categorize your application completely depends on your application context and specific domain. The main purpose of this segmentation is simply to provide a way to envision and organize different approaches and technology positions.

Custom application development is oftentimes more of an art than a science—that's why there's no prescriptive truth to follow. The approaches, techniques, and technologies to use depend on the application context, the real domain to solve, and on the developer team context and skill set.

6. Appendix A – Silverlight migration paths

As explained in this guide, RIA containers are being replaced gradually by other alternatives with broader device support (HTML5) or better device integration (native development). However, your choice and timeframe may be different based on your current investments and needs. Over time, this will involve more HTML5 and native applications and less RIA containers, but you are in control of the final decision and timeframe.

If you are currently using Silverlight, you can make this transition gradually at the timeframe of your choice. It is also possible that some portions of the application need to be coded in HTML5 and/or Silverlight. Since Silverlight is a plug-in, it can be used within HTML content. If your scenario is more suited for a native application, Silverlight also provides a good bridge for platforms supporting .NET/XAML, such as WPF in the desktop, Windows Store Apps .NET model, or even (with the solutions explained previously in this guide) non-Microsoft devices. Because the latest Silverlight version is supported for 10 years, you can make that transition gradually while still using the mature and stable solution Silverlight provides.

If you have a major application that has a monolithic architecture—i.e., the back end tightly integrated with the front end—you should decouple these two parts of the system, so that multiple front ends (HTML4, HTML5, Silverlight, native mobile) can connect to the same back end.

Additionally, there are some applications whose requirements are still only covered by RIA containers solutions (for example, video content protection), and therefore you should continue with the existing investment.

Because of the breadth of different scenarios, and the variety of application types, context, and priorities, the following migration paths are recommended.

Current application type and Silverlight use	Application context and priorities	Recommended path
Web media solution, like Silverlight video streaming system running in-browser.	Maximize reach of users and multi-platform devices. Regular HTML5 media features would be enough	Migrate to HTML5 web development approaches to support all the multi-vendor devices (such as tablets, computers, and smartphones). Although supporting any device also impacts on the media format and codecs you need to support. IMPORTANT: As transitional approach, a fallback approach might be the best choice. If the client supports Silverlight, current capabilities in Silverlight (like smooth-streaming or DRM) are more powerful than media in HTML5. Even more, some old browsers support Silverlight but they don't support HTML5.
Advanced web media solution, like a private video Silverlight Smooth-Streaming system running in-browser.	Run an advanced media solution with capabilities still not supported by HTML5 media features, like 3D and DVR. System in a controlled environment where the majority is based on Windows-based PCs or Mac OS X computers rather than reaching any kind of mobile devices.	The recommended path is to maintain current media app in Silverlight 5, which shipped with 10 years of support. If DRM is required then Silverlight continues to be a supported option. Plan a progressive migration once HTML5 supports the required advanced capabilities.
Public Internet web application/site using Silverlight in-browser.	Maximize reach of users and multi-platform devices.	Migrate to HTML5 web development approaches, to support all the multi-vendor devices (such as tablets, computers, and smartphones).

Private/internal business web application using Silverlight in-browser.	The application is a collateral medium-range type rather than a critical core-business application (constantly growing/evolving). The application will continue running in a controlled environment where the majority is based on Windows-based PCs or Mac OS X computers rather than reaching any kind of mobile devices.	The recommended path is to maintain current app in Silverlight 5, which shipped with 10 years of support, while planning a progressive migration for the future to HTML5/JS. Take into account that the browser to be used must be a traditional browser running in the desktop. IE "modern browser" (Windows Store style) does not support the Silverlight plug-in.
	The application is a critical core-business web application, constantly growing/evolving over the years. The application evolution is heading a new modernizing approach and any kind of mobile devices (such as any tablet) is also a possibility for certain scenarios.	The recommended path is to progressively migrate the presentation layer of the core-business application to HTML5/JavaScript-based web approaches like ASP.NET SPA (Single Page Application) based on ASP.NET MVC and JavaScript libraries for MVVM (Knockout lib) and other libraries, consuming ASP.NET Web API services. The current distributed services layer can be reused or migrated to web-API services with not a huge investment. Take into account that the browser to be used must be a traditional browser running in the desktop. IE "modern browser" (Windows Store style) does not support the Silverlight plug-in.
Private/internal business desktop application using Silverlight out-of-browser	The application is a collateral medium-range type rather than a critical core-business application (constantly growing/evolving).	The recommended path is to maintain current app in Silverlight 5 which shipped with a 10 years length of support. In the future, plan a progressive migration.
	The application is a critical core-business desktop application, constantly growing/evolving over the years. And touch-scenarios are possibly desired, abandoning a traditional desktop approach.	The recommended path is to maintain the current presentation layer (in the short term), but start planning a migration to an immersive Windows Store application (C#/XAML) if touch-scenarios are desired and where the new application development approach will be quite similar to a current Silverlight/XAML application with MVVM, etc.
	The application is a critical core-business desktop application, constantly growing/evolving over the years. Traditional desktop scenario is required rather than touch-scenarios.	The recommended path is to maintain the current presentation layer (in the short term), but start planning a migration to new contexts. There are several possibilities here: - Evaluate a migration to a web HTML5/JavaScript presentation layer approach if a web scenario fits all the requirements. - Migrate to Windows Presentation Foundation (WPF) if you want to step out of Silverlight; the migration would be natural because of current decoupled architecture. The WPF approach could be quite similar to a current Silverlight/XAML application with MVVM.

Table 6-1

References

Porting Silverlight XAML/code to a Windows Store app (Windows)	http://msdn.microsoft.com/en-us/library/windows/apps/xaml/br229571.aspx
Migrate Silverlight applications to Windows Store apps	http://blogs.msdn.com/b/win8devsupport/archive/2012/11/12/port-a-silverlight-application-to-windows-8.aspx
Silverlight support policy	http://support.microsoft.com/lifecycle/?LN=en-us&c2=12905

7. Appendix B — Positioning data-access technologies

Over the years, Microsoft has been creating and evolving more flexible data-access technologies. Some are very specialized, others are general purpose, but all of them share two common goals: to help applications access the information they need, and to help developers spend less time dealing with intricate details of data storage, which allows more time focusing on their own software that utilizes that data, thus bringing real value to their customers.

The selection of a proper technology to access data should consider the type of data source you will have to work with and how you want to handle the data within the application. Some technologies are better adapted to certain scenarios. The next sections describe the main technologies and characteristics to be considered.

Entity Framework

Entity Framework (EF) is Microsoft's recommended data-access technology for new applications that need to access relational databases. EF is an object-relational mapper (O/RM) that enables .NET developers to work with relational data using domain-specific objects. It eliminates the need for most of the data-access code that developers usually need to write. EF allows you to create a model by writing code (classes) or using visual diagrams in the EF Designer. Both of these approaches can be used to target an existing database or create a new database.

EF is recommended when you want to create an entity model mapped to a relational database. At a higher level, one entity class is usually mapped to one or multiple tables that comprise a complex entity. The most outstanding advantage of EF is that the database it works with will be transparent in many ways. This is because the EF model generates native SQL statements required for each Database Management System (DBMS), so it would be transparent whether you are working over Microsoft SQL Server, Oracle, DB2, or MySQL. You simply need to change the EF provider related to each DBMS. (In most cases, this involves nothing more than changing a connection string and regenerating the EF model). So, EF is suitable when the intention is to use an Object Role Modeling (ORM) development model based on an object model, then mapped to a relational model through a flexible scheme. If you use EF, you will also commonly use LINQ to Entities, too. Consider LINQ to Entities if the intention is to execute strongly typed queries against entities using an object-oriented syntax such as LINQ.

Third-party O/RM technologies: There are many other good technologies (such as O/RMs like *NHibernate, LinqConnect, DataObjects.net, BLToolkit, OpenAccess*, and *Subsonic*) that are not provided and supported by Microsoft, but they are good approaches, too.

ADO.NET: Consider using ADO.NET base classes if access to a lower API level is required. This will provide complete control (SQL statements, data connections, etc.) but relinquish the transparency toward Relational Database Management System (RDBMS) systems provided by EF. You may also need to use ADO.NET if you need to reuse existing inversions (massive use of existing stored procedures or legacy Data Access Building Blocks implemented using ADO.NET).

Microsoft Sync Framework: Consider this technology if you are designing an application that should support scenarios that are occasionally disconnected/connected or that require cooperation between the different databases.

LINQ to XML: Consider this technology when there is an extensive use of XML documents within your application and you want to query them through LINQ syntax.

NoSQL databases and technologies: NoSQL is an umbrella term for a DBMS identified by non-adherence to the widely used RDBMS model. Data stores that fall under this term may not require fixed-table schemas; they usually avoid join operations and generally do not use SQL sentences for data manipulation.

The motivation for this type of architecture is usually high horizontal scalability and high optimization for retrieval and appending operations, and often NoSQL systems offer little functionality beyond record storage (for example, key/value stores). The reduced runtime flexibility compared to full SQL systems is compensated by marked gains in scalability and performance for certain data models. NoSQL databases are useful when working with a huge quantity of data when the nature of that data does not require a

relational model. The data can be structured, but NoSQL is used when what really matters is the ability to store and retrieve huge quantities of data, rather than using relationships between the elements. Usage examples might be to store millions of key/value pairs in one or a few associative arrays.

Currently, Microsoft's main NoSQL data sources are Windows Azure tables and Windows Azure blobs. There are also other recommended third-party NoSql data sources like *MongoDb, Casandra, RavenDb*, and *CouchDB*. The selection of a NoSql database/technology fully depends on the type of application and data access you need, because they are usually very different regarding pros, cons, usage, and API (for instance, all the previously mentioned NoSQL database systems are very different between them). That is one of the biggest differences when comparing NoSQL to relational databases, which are pretty similar regarding usage (relational tables and joins) and APIs (based on SQL).

NoSQL APIs: Because most NoSQL implementations are quite different from one another, they usually each have a concrete and independent API implementation. Conveniently, however, many of them have a .NET API and even remote access API (for example, based on REST services), like Windows Azure tables and blobs.

BigData: Big data is a collection of data sets so large and complex that it becomes difficult to process using traditional data processing systems. The trend to larger data sets is due to the additional information derivable from analysis of a single large set of related data, compared to separate smaller sets with the same total amount of data, allowing correlations to be found to "spot business trends," determine quality of research, and so on.

Big data sizes are currently ranging from a few dozen terabytes to many petabytes of data in a single data set. With this difficulty, a new platform of "big data" technologies is required to handle large quantities of data. Currently, the best example is probably the Apache Hadoop big data platform.

Hadoop is an open source library designed to batch-process massive datasets in parallel. It's based on the Hadoop distributed file system (HDFS), and consists of utilities and libraries for working with data stored in clusters. These batch processes run using different technologies, including Map/Reduce jobs.

Microsoft's end-to-end roadmap for big data embraces Apache Hadoop by distributing enterprise class, Hadoop-based solutions on both Windows Server and Windows Azure. This Microsoft Hadoop enterprise distribution is named as **HDInsight.** To learn more about the Microsoft roadmap for big data, see the Microsoft Big Data page (http://www.microsoft.com/bigdata/).