Application Architecture Guide - Chapter 16 - Rich Internet Applications (RIA)

From Guidance Share

Note - The patterns & practices Microsoft Application Architecture Guide, 2nd Edition is now live at http://msdn.microsoft.com/en-us/library/dd673617.aspx.

Contents

Chapter 16: Rich Internet Applications

Objectives

- Define a rich Internet application (RIA).
- Understand key scenarios where RIAs would be used.
- Understand the components found in an RIA.
- Learn the design considerations for RIAs.
- Learn the guidelines for performance, security, and deployment of RIAs.
- Learn the key patterns and technology considerations for designing RIAs.

Overview

Rich Internet applications (RIAs) support rich graphics and streaming media scenarios, while providing most of the deployment and maintainability benefits of a Web application. RIAs run in a browser plug-in, such as Microsoft® Silverlight®, as opposed to extensions that utilize browser code, such as Asynchronous JavaScript and XML (AJAX). A typical RIA implementation utilizes a Web infrastructure combined with a client-side application that handles the presentation. The plug-in provides library routines for rich graphics support as well as a container limiting access to local resources for security purposes. RIAs have the ability to run more extensive and complex client-side code than possible in a normal Web application, thus providing the opportunity to reduce load on the Web server.

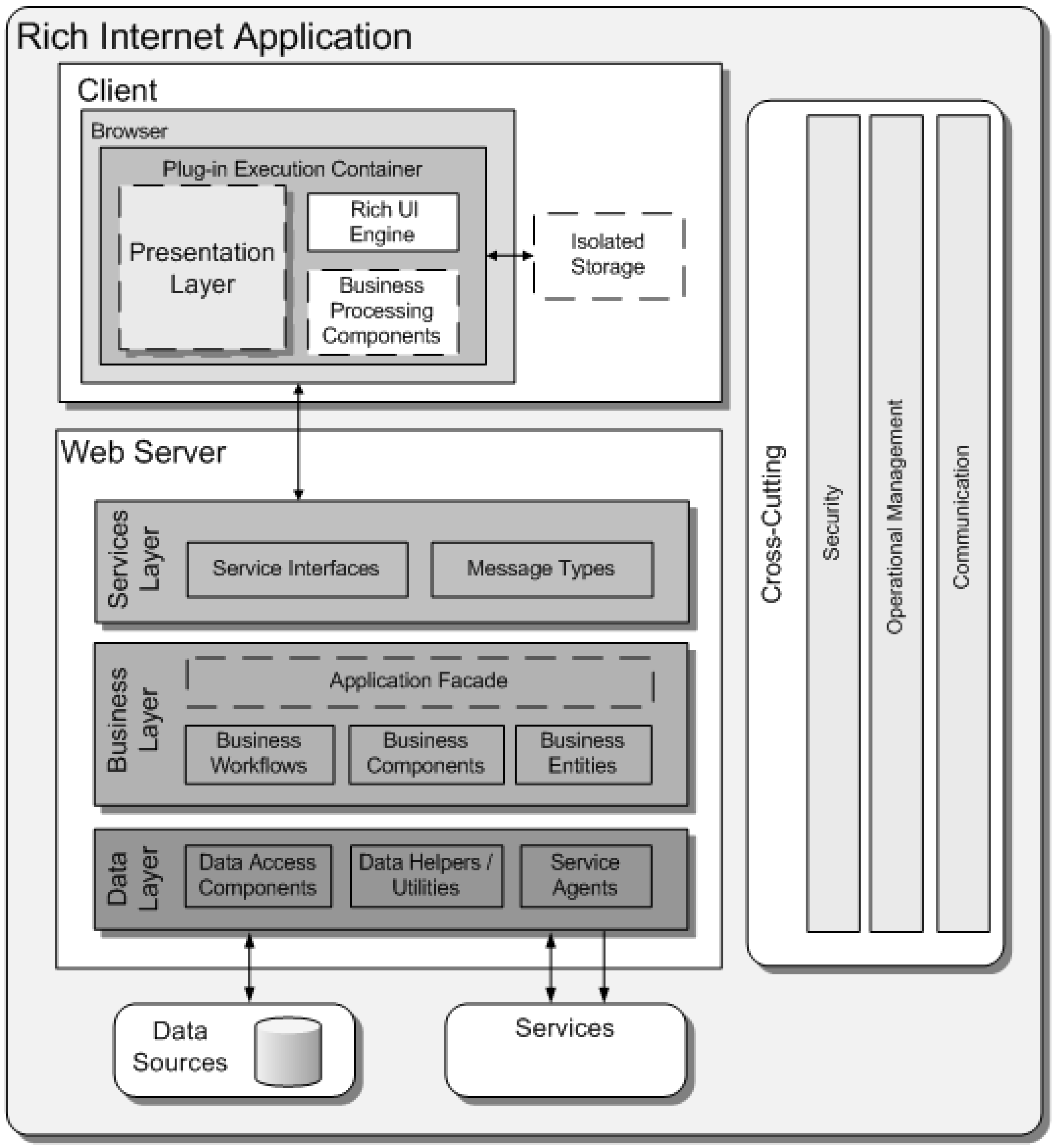

Figure 1 Architecture of a typical RIA implementation

Design Considerations

The following design guidelines provide information about different aspects you should consider when designing an RIA. Follow these guidelines to ensure that your application meets your requirements and performs efficiently in scenarios common to RIAs:

- **Choose an RIA based on audience, rich interface, and ease of deployment**. Consider designing an RIA when your vital audience is using a browser and operating system that supports RIAs. If part of your vital audience is on a non-RIA-supported browser, consider whether you can influence limiting browser choice to a supported version. If you cannot influence the browser choice, consider if the loss of audience is significant enough to require choosing another application type, such as a Web application using AJAX. With an RIA, the ease of deployment and maintenance is similar to that of a Web application, assuming that your clients have a reliable network connection. RIA implementations are well suited to Web-based scenarios where you need visualization beyond that

provided by basic HTML. They are likely to have more consistent behavior and require less testing across the range of supported browsers when compared to Web applications that utilize advanced functions and code customizations. RIA implementations are also perfect for streaming-media applications. They are less suited to extremely complex multi-page user interfaces (UIs).
- **Design to use a Web infrastructure utilizing services**. RIA implementations require an infrastructure similar to Web applications. Communication to the business layer of your application is usually through services, which allows reuse of existing Web application infrastructure. Transferring logic to the client should only be considered later in the design process. Only transfer logic for performance optimization and UI responsiveness reasons.
- **Design for running in the browser sandbox**. RIA implementations have higher security by default and therefore may not have access to all devices on a machine, such as cameras and hardware video acceleration. Access to the local file system is limited. Local storage is available, but there is a maximum limit.
- **Determine the complexity of your UI requirements**. Consider the complexity of your UI. RIA implementations work best when using a single screen for all operations. They can be extended to multiple screens, but this requires extra code and screen-flow consideration. Users should be able to easily navigate or pause, and return to the appropriate point in a workflow, without restarting the whole process. For multi-page UIs, use deep-linking methods. Also, manipulate the Uniform Resource Locator (URL), the history list, and the browser's back and forward buttons to avoid confusion as users navigate between screens.
- **Use scenarios to increase application performance or responsiveness**. List and examine the common application scenarios to decide how to intelligently divide and load modules, as well as how to cache or move business logic to the client. To reduce the download and start-up time for the application, intelligently segregate functionality into separate downloadable modules. Initially load only code stubs, which can lazy-load other modules. Consider moving or caching regularly used business layer processes on the client for maximum application performance.
- **Design for scenarios where the plug-in is not installed**. Because RIA implementations require a browser plug-in, a you should design for non-interruptive plug-in installation. Consider whether your clients have access to, have permission to, and will want to install the plug-in. Consider what control you have over the installation process. Plan for the scenario where users cannot install the plug-in, by displaying an informative error message, or by providing an alternative Web UI.

RIA Frame

There are several common issues that you must consider as your develop your design. These issues can be categorized into specific areas of the design. The following table lists the common issues for each category where mistakes are most often made.

Table 1 RIA frame

Category	Key issues
Business Layer	* Moving business operations to the client for reasons other than improved user experience or application performance ▪ Failing to use profiling to identify expensive business operations that should be moved to the client ▪ Trying to move all business processing to the client ▪ Failing to put business rules on the client into their own separate components to allow easy caching, updating, and replacement ▪ Using less powerful browser-supported languages instead of considering windowless RIA plug-ins written in rich programming languages to provide client side processing
Caching	* Failing to use isolated storage appropriately ▪ Failing to check and request increase of the isolated storage quota ▪ Failing to intelligently divide large client applications into smaller separately downloadable components ▪ Downloading and instantiating the entire application at start-up instead of intelligently and dynamically loading modules

Communication	* Trying to use a synchronous communication model ▪ Using an incorrect strategy to bind to the service interface ▪ Attempting to use sockets over unsupported or blocked ports
Controls	* Adding custom control behavior through sub-classing instead of attaching new behavior to specific instances of controls ▪ Incorrect use of controls for a UI type ▪ Implementing custom controls when not required
Composition	* Incorrectly implementing composition patterns, leading to dependencies that require frequent application redeployment ▪ Using composition when not appropriate for the scenario ▪ Not considering composition, and completely rewriting applications that could be reused with minimal or no changes
Data Access	* Performing data access from the client ▪ Failing to filter data at the server
Exception Management	* Failing to design an exception-management strategy ▪ Failing to trap asynchronous call errors and unhandled exceptions; for example, not using the OnError event handler supported by Microsoft Silverlight to trap exceptions in asynchronous calls
Logging	* Failing to log critical errors ▪ Failing to consider a strategy to transfer logs to the server ▪ Segregating logs by machine instead of by user
Media & Graphics	* Failing to take advantage of adaptive streaming for video delivery ▪ Assuming access to hardware acceleration on client
Presentation	* Not pixel-snapping UI elements, which results in degraded UI appearance ▪ Failing to handle the forward and back button events ▪ Not considering and designing for deep linking when necessary
Portability	* Failing to consider the cost of testing for each platform and browser combination in a Web application compared to using an RIA interface ▪ Using platform-specific APIs in code rather than portable RIA routines ▪ Using less powerful browser-based languages instead of more powerful portable RIA languages
State Management	* Failing to use isolated storage ▪ Using the server to store frequently changing application state ▪ Failing to synchronize state between the client and server when user configuration must be available on multiple clients
Validation	* Failing to identify trust boundaries and validate data that passes across them ▪ Failing to validate data on both the client and the server ▪ Failing to collate extensive client-side validation code into a separate downloadable module

Business Layer

RIA implementations provide the capability to move business processing to the client. Consider moving logic that improves the user experience or performance of the application as a whole.

Consider the following guidelines when designing the business layer:

- Consider starting with your business logic on the server exposed through services. Use scenario-based profiling to discover and target routines that cause the heaviest server load or have the most impact on UI responsiveness. Consider moving or caching only those routines to the client.
- When locating business logic on the client, consider putting business rules or routines in a separate assembly that the application can load and update independently.
- If your business logic is duplicated on the client and the server—for instance, if you are processing business rules on the client for performance but implement the same rules in the service for integrity—use the same code language on the client and server if your RIA implementation allows it. This will reduce any differences in language implementations and make it easier to be consistent in how rules are processed.
- If your RIA implementation allows creation of an instance without a UI, consider using it intelligently. You can keep your processing code in more structured, powerful, or familiar programming languages (such as C#) instead of using less flexible browser-supported languages.
- For security reasons, do not put highly sensitive unencrypted business logic on the client.

Caching

RIA implementations generally use the normal browser caching mechanism. Caching resources intelligently will improve application performance.

Consider the following guidelines when designing a caching strategy:

- Cache components of your application for improved performance and fewer network round trips. Allow the browser to cache objects that are not likely to change during a session. Utilize specific RIA local storage for information that changes during a session, or which should persist between sessions.
- Use installation, updates, and user scenarios to derive intelligent ways to divide and load application modules.
- Load stubs at start-up and then dynamically load additional functionality in the background. Consider using events to intelligently pre-load modules just before they may be required.
- To avoid unintended exceptions, check that isolated storage is large enough to contain the data you will write to it. Storage space does not increase automatically; you must ask the user to increase it.

Communication

RIA implementations must use the asynchronous call model for services to avoid blocking browser processes. Cross-domain, protocol, and service-efficiency issues should be considered as part of your design. If your RIA implementation allows it, consider using threading for background operations.

Consider the following guidelines when designing a communication strategy:

- If you have long-running code, consider using a background thread or asynchronous execution to avoid blocking the UI thread.
- If you are authenticating through services, design your services to use a binding that your RIA implementation supports.
- Ensure that the RIA and the services it calls use compatible bindings that include security information.
- If your RIA client must access a server other than the one from which it was downloaded, ensure that you use a cross-domain configuration mechanism to permit access to the other servers/domains.
- Consider using sockets to push data to the client if client polling causes heavy server load. Consider using sockets to push information to the server when this is significantly more efficient than using services; for example, real-time multi-player gaming scenarios utilizing a central server.

Controls

RIA implementations usually have their own native controls. You can often mix RIA-based and non-RIA-based controls in the same UI, but extra communication code may be required.

Consider the following guidelines when designing a strategy for controls:

- Use native RIA controls where possible.
- If the appropriate control is not supplied with your RIA package, consider third-party RIA-specific controls.
- If a native RIA control is not available, consider using a windowless RIA control in combination with an HTML or Windows Forms control that does have the necessary functionality.
- If your RIA controls support the ability to attach added behaviors, use that ability and avoid sub-classing the controls to extend functionality.

Composition

Composition allows you to implement highly dynamic UIs that you can maintain without changes to the code or redeployment of the application. You can compose an application using RIA and non-RIA components.

Consider the following guidelines when designing a composition strategy:

- Evaluate which composition model patterns best suit your scenario.
- If an interface must gather information from many disparate sources, and those sources are user-configurable or change frequently, consider using composition.
- When migrating an existing HTML application, consider mixing RIA and the existing HTML on the same page to reduce application reengineering.

Data Access

RIA implementations access data in a similar way to normal Web applications. They should request data from the Web server through services in the same way as an AJAX client. After data reaches the client, it can be cached to maximize performance.

Consider the following guidelines when designing a data-access strategy:

- Use client-side processing to minimize the number of round trips to the server, and to provide a more responsive UI.
- Filter data at the server rather than at the client to reduce the amount of data that must be sent over the network.
- For operation-based applications, utilize services to access data.

Exception Management

A good exception-management strategy is essential for correctly handling and recovering from errors in any application. In an RIA, you must consider asynchronous exceptions as well as exception coordination between the client and server code.

Consider the following guidelines when designing an exception-management mechanism:

- Do not use exceptions to control business logic.
- Only catch internal exceptions that you can handle. For example, catch data conversion exceptions that can occur when trying to convert null values.
- Design an appropriate exception-propagation strategy. For example, allow exceptions to bubble up to boundary layers where they can be logged and transformed as necessary before passing them to the next layer.
- Design an approach for catching and handling unhandled exceptions. Unhandled exceptions in RIAs are passed to the browser. They will allow execution to continue after the user dismisses a browser error message. Provide a friendly error message for the user if possible. Stop program execution if continued execution would be harmful to the data integrity of the application or could mislead the user into thinking the application is still in a stable state.
- Design an appropriate logging and notification strategy for critical errors and exceptions that does not reveal sensitive information.
- Design for both synchronous and asynchronous exceptions. Use **try/catch** blocks to trap exceptions in synchronous code. Put exception handling for asynchronous service calls in the separate handler designed specifically for such exceptions; for example, in Silverlight, this is the **OnError** handler.

Logging

Logging for the purpose of debugging or auditing can be challenging in an RIA implementation. For example, access to the client file system is not available in Silverlight applications, and execution of the client and the server proceed asynchronously. Log files from a client user must be combined with server log files to gain a full picture of program execution.

Consider the following guidelines when designing a logging strategy:

- Consider the limitations of the logging component in the RIA implementation. Some RIA implementations log each user's information in a separate file, perhaps in different locations on the disk.
- Determine a strategy to transfer client logs to the server for processing. Recombination of different users' logs from the same machine may be necessary if troubleshooting a client machine–specific issue.
- If using isolated storage for logging, consider the maximum size limit and the requirement to ask the user for increases in storage capacity.
- Consider enabling logging and transferring logs to the server when exceptions are encountered.

Media and Graphics

RIA implementations provide a much richer experience and better performance than ordinary Web applications. Research and utilize the built-in media capabilities of your RIA platform. Keep in mind the features that may not be available on the RIA platform compared to a stand-alone media player.

Consider the following guidelines when designing for multimedia and graphics:

- Design to utilize streaming media and video in the browser instead of invoking a separate player utility.
- To increase performance, position media objects on whole pixels and present them in their native size.
- Use adaptive streaming in conjunction with RIA clients to gracefully and seamlessly handle varying bandwidth issues.
- Utilize the native vector graphics engine for the best drawing performance.
- If programming an extremely graphics-intensive application, find out if your RIA implementation provides hardware acceleration. If it does not, create a baseline for what is acceptable drawing performance. Consider a plan to reduce load on the graphics engine if it falls below acceptable limits.

Mobile

RIA implementations provide a much richer experience than an ordinary mobile application. Utilize the built-in media capabilities of the RIA platform you are using.

Consider the following guidelines when designing for mobile device multimedia and graphics:

- When an RIA application needs to be distributed on a mobile client, research whether an RIA plug-in implementation is available for the device you want to support. Find out if the RIA plug-in has reduced functionality compared to non-mobile platforms.
- Attempt to start from a single or similar codebase. Branch code as required for specific devices.
- Re-examine UI layout and implementation for the smaller screen size.
- RIA applications work on mobile devices, but consider using different layout code on each type of device to reduce the impact of different screen sizes when designing for Microsoft Windows Mobile®.

Portability

One of the main benefits of RIAs is the portability of compiled code between different browsers, operating systems, and platforms. Similarly, using a single source codebase, or similar codebases, reduces the time and cost of development and maintenance, while still providing platform flexibility.

Consider the following guidelines when designing for portability:

- Design for the goal of "write once, run everywhere," but be willing to fork code in cases where overall project complexity or feature tradeoffs dictate that you do so.

- When comparing an RIA and a Web application, consider that differences between browsers can require extensive testing of ASP.NET and JavaScript code. With an RIA application, the plug-in creator, and not the developer, is responsible for consistency across different platforms.
- If your audience will be running the RIA on multiple platforms, do not use features available only on one platform; for example, Windows Integrated Authentication. Design a solution based on standards that are portable across different clients.
- When possible, use richer development languages that are supported for both rich clients and RIAs. See the Technology considerations in this section for recommendations.
- Make full use of the native RIA code libraries.

Presentation

RIA applications work best when designed as one central interface. Multi-page UIs require that you consider how you will link between pages. Positioning of elements on a page can affect both the look and performance of your RIA application.

Consider the following guidelines when designing for presentation:

- To avoid anti-aliasing issues that can cause fuzziness in RIAs, snap UI components to whole pixels. Pay attention to centering and math-based positioning routines. Consider writing a routine that checks for fractional pixels and rounds them to the nearest whole pixel value.
- Trap the browser's forward and back button events to avoid unintentional navigation away from your page.
- For multi-page UIs, use deep-linking methods to allow unique identification of and navigation to individual application pages.
- For multi-page UIs, consider the ability to manipulate the browser's address text box content, history list, and back and forward buttons in order to implement normal Web page–like navigation.

State Management

You can store application state on the client by using isolated storage if the state changes frequently. If application state is vital at start-up, synchronize the client state to the server.

Consider the following guidelines when designing for state management:

- Store state on the client in isolated storage to persist it during and between sessions.
- Store the client state on the server if loss of state on the client would be catastrophic to the application's function. Consider that isolated storage is deleted when the browser cache is cleared.
- Store the client state on the server if the client requires recovery of application state when using different accounts, or when running on other hardware installations
- Verify the stored state between the client and server at start-up, and intelligently handle the case when they are out of synchronization.
- Design for multiple concurrent sessions because you cannot prevent multiple RIA instances from initializing. Design either for concurrency in your state management, or to detect and prevent multiple sessions from corrupting application state.

Validation

Validation must be performed using code on the client or through services located on the server. If you require more than trivial validation on the client, isolate validation logic in a separate downloadable assembly. This makes the rules easy to maintain.

Consider the following guidelines when designing for validation:

- Use client-side validation to maximize the user experience, and server-side validation for security. Use isolated storage to hold client-specific validation rules.
- In general, assume that all client-controlled data is malicious. The server should re-validate all data sent to it. Design to validate input from all sources, such as the query string, cookies, and HTML controls.
- Design to constrain, reject, and sanitize data. Validate input for length, range, format, and type.
- For rules that require access to server resources, evaluate whether it is more efficient to use a single service call that performs validation on the server.

- If you have a large volume of client-side validation code that may change, consider locating it in a separate downloadable module so it can be easily replaced without downloading the entire RIA application again.

Performance Considerations

Properly using the client-side processing power for an RIA is one significant way to maximize performance. Server-side optimizations similar to those used for Web applications are also a major factor.

Consider the following key performance guidelines:

- Cache components of your application for improved performance and fewer network round trips. Allow the browser to cache objects that are not likely to change during a session. Utilize specific RIA local storage for information that changes during a session, or that should be persisted between sessions.
- Use installation, updates, and user scenarios to derive intelligent ways to divide and load application modules. Load stubs at start-up and then dynamically load additional functionality in the background. Consider using events to intelligently pre-load modules just before they may be needed. For example, in a merchant application, wait to load checkout functionality until sometime after a user has added an item to the shopping cart.
- Use scenario-based profiling to discover and target routines that cause the heaviest server load, or that have a major impact on UI responsiveness. Consider moving or caching these routines on the client. When locating business logic on the client, place business rules or routines in a separate assembly that the application can load and update independently.
- Position media objects on whole pixels and present them in their native size. Do not blend them with other objects, such as progress bar controls.
- Be aware of the size of your drawing areas. Only redraw parts of an area that are actually changing. Reduce overlapping regions when not necessary to reduce blending. Use profiling and debugging methods—for example, the “EnableRedrawRegions = true” setting in Silverlight—to see what areas are being redrawn. Note that certain effects, such as blurring, can cause every pixel in an area to be redrawn. Windowless and transparent controls can also cause unintended redrawing and blending.

Security Considerations

RIA applications mitigate a variety of common attack vectors because they run inside a sandbox in the browser. Access to most local resources is limited or restricted, which minimizes opportunities for attacks on the RIA and the client platform on which it runs.

Consider the following restrictions:

- Applications run inside a sandbox in the browser, within a memory space isolated from other applications.
- Browsing of the local client file system is restricted.
- Access to specialized local devices such as Webcams may be limited or not available.
- Access to domains other than the one that delivered the application is limited, protecting the user from cross-site scripting attacks

Protect sensitive information using the follow methods:

- The isolated storage mechanism provides a method for storing data locally, but does not provide built-in security. Do not store sensitive data locally unless it is encrypted using the platform encryption routines.
- Create an exception-management strategy to prevent exposure of sensitive information through unhandled exceptions.
- Be careful when downloading sensitive business logic used on the client because tools are available that can extract the logic contained in downloaded XAML Browser Application (XBAP) files. Implement sensitive business logic using Web services. If the logic must be on the client for performance reasons, research and utilize any available obfuscation methods.
- To minimize the amount of time that sensitive data is available on the client, utilize dynamic loading of resources and overwrite or clear components containing sensitive data from the browser cache.

Deployment Considerations

RIA implementations provide many of the same benefits as Web applications in terms of deployment and maintainability. Design your RIA as separate modules that can be downloaded individually and cached to allow replacement of one module instead of the whole application. Version your application and components so that you can detect the versions that clients are running.

Consider the following guidelines when designing for deployment and maintainability:

- Consider how you will manage the scenario where the RIA browser plug-in is not installed.
- Consider how you will redeploy modules when the application instance is still running on a client.
- Divide the application into logical modules that can be cached separately, and that can be replaced easily without requiring the user to download the entire application again.
- Version your components.

Installation of the RIA plug-in

Consider how you will manage installation of the RIA browser plug-in when it is not already installed:

- **Intranet**. If available, use application distribution software or the Group Policy feature of the Microsoft Active Directory® directory service to pre-install the plug-in on each computer in the organization. Alternatively, consider using Windows Update, where Silverlight is an optional component. Finally, consider manual installation through the browser, which requires the user to have Administrator privileges on the client machine.
- **Internet**. Users must install the plug-in manually, so you should provide a link to the appropriate location to download the latest plug in. For Windows users, Windows Update provides the plug-in as an optional component.
- **Plug-in updates**. In general, updates to the plug-in take into account backward compatibility. You may target a particular plug-in version, but consider implementing a plan to verify your application's functionality on new versions of the browser plug-in as they become available. For intranet scenarios, distribute a new plug-in after testing your application. In Internet scenarios, assume that automatic plug-in updates will occur. Test your application using the plug-in beta to ensure a smooth user transition when the plug-in is released.

Distributed Deployment

Because RIA implementations move presentation logic to the client, a distributed architecture is the most likely scenario for deployment.

In a distributed RIA deployment, the presentation logic is on the client and the business and data layers reside on the Web server or application server. Typically, you will have your business and data access layers on the same server, as shown in Figure 2.

Figure 2 Distributed deployment for an RIA

Consider the following guidelines:

- If your applications are large, factor in the processing requirements for downloading the RIA components to clients.
- If your business logic is shared by other applications, consider using distributed deployment.
- If you use sockets in your application and you are not using port 80, consider which ports you must open in your firewall.
- Ensure that you use a **crossdomain.xml** file so that RIA clients can access other domains where required.
- Design the presentation layer in such as way that it does not initiate, participate in, or vote on atomic transactions.
- Consider using a message-based interface for your business logic.

Load Balancing

When you deploy your application on multiple servers, you can use load balancing to distribute RIA client requests to different servers. This improves response times, increases resource utilization, and maximizes throughput. Figure 3 shows a load-balanced scenario.

Figure 3 Load balancing an RIA deployment Consider the following guidelines when designing your application to use load balancing:

- Avoid server affinity. *Server affinity* occurs when all requests from a particular client must be handled by the same server. It is most often introduced by using locally updatable caches or in-process or local session state stores.
- Consider storing all state on the client and designing stateless business components.
- Consider using network load balancing software to implement redirection of requests to the servers in an application farm.

Web Farm Considerations

Consider using a Web farm that distributes requests from RIA clients to multiple servers. A Web farm allows you to scale out your application, and reduces the impact of hardware failures. You can use either load balancing or clustering solutions to add more servers for your application.

Consider the following guidelines:

- Consider using clustering to reduce the impact of hardware failures.
- Consider partitioning your database across multiple database servers if your application has high I/O requirements.
- If you must support server affinity, configure the Web farm to route all requests for the same user to the same server.
- Do not use in-process session management in a Web farm unless you implement server affinity, because requests from the same user cannot be guaranteed to be routed to the same server otherwise. Use the out-of-process session service or a database server for this scenario.

Pattern Map

Key patterns are organized by the key categories detailed in the Rich Internet Applications Frame in the following table. Consider using these patterns when making design decisions for each category.

Table 2 Pattern map

Category	**Relevant patterns**
Business Layer	* Service Layer
Caching	* Page Cache
Communication	* Asynchronous Callback ▪ Command
Controls	* Chain of Responsibility
Composition	* Composite View ▪ Inversion of Control
Presentation	* Application Controller ▪ Model-View-Controller

- For more information on the Composite View pattern, see “Patterns in the Composite Application Library” at http://msdn.microsoft.com/en-us/library/cc707841.aspx
- For more information on the Model-View-Controller (MVC) and Application Controller patterns, see “Patterns of Enterprise Application Architecture (P of EAA)” at http://martinfowler.com/eaaCatalog/
- For more information on the Page Cache pattern, see “Enterprise Solution Patterns Using Microsoft .NET” at http://msdn.microsoft.com/en-us/library/ms998469.aspx
- For more information on the Chain of Responsibility and Command patterns, see “data & object factory” at http://www.dofactory.com/Patterns/Patterns.aspx
- For more information on the Asynchronous Callback pattern, see “Creating a Simplified Asynchronous Call Pattern for Windows Forms Applications” at http://msdn.microsoft.com/en-us/library/ms996483.aspx

- For more information on the Service Layer pattern, see “P of EAA: Service Layer” at http://www.dofactory.com/Patterns/Patterns.aspx

Pattern Descriptions

- **Application Controller**. An object that contains all of the flow logic and is used by other Controllers that work with a Model and display the appropriate View.
- **Asynchronous Callback**. Execute long-running tasks on a separate thread that executes in the background, and provide a function for the thread to call back into when the task is complete.
- **Chain of Responsibility**. Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request.
- **Command**. Encapsulate request processing in a separate command object that exposes a common execution interface.
- **Composite View.** Combine individual views into a composite view.
- **Inversion of Control**. Populate any dependencies of objects on other objects or components that must be fulfilled before the object can be used by the client application.
- **Model-View-Controller**. Separate the UI code into three separate units; Model (data), View (interface), and Presenter (processing logic), with a focus on the View. Two variations on this pattern include Passive View and Supervising Controller, which define how the View interacts with the Model.
- **Page Cache**. Improve the response time for dynamic Web pages that are accessed frequently, but that change less often and consume a large amount of system resources to construct.
- **Service Layer**. An architectural design pattern where the service interface and implementation is grouped into a single layer.

Technology Considerations

The following guidelines discuss Silverlight and Microsoft Windows Communication Foundation (WCF) and provide specific guidance for these technologies. At the time of writing, the latest versions are WCF 3.5 and Silverlight 2.0. Use the guidelines to help you to choose and implement an appropriate technology:

- At the time of this guide’s release, Silverlight for Mobile was an announced product and in development, but not released.
- Silverlight currently supports the Safari, Firefox, and Microsoft Internet Explorer browsers though a plug-in. Through these browsers, Silverlight 2.0 currently supports Mac, Linux, and Microsoft Windows®. Support for Windows Mobile was also announced in 2008.
- The local storage mechanism for Silverlight is named “Isolated Storage.” The current initial size is 1 megabyte (MB). The maximum storage size is unlimited. Silverlight requires that you ask the user to increase the storage size.
- Silverlight only supports Basic HTTP binding. WCF in .NET 3.5 supports Basic HTTP binding, but security is not turned on by default. Be sure to turn on at least transport security to secure your service communications.
- Silverlight does not obfuscate modules downloaded as XBAPs. XBAPs can be decompiled and the programming logic extracted.
- The .NET cryptography APIs are available in Silverlight and should be utilized when storing and communicating sensitive data to the server if not already encrypted using another mechanism.
- Silverlight contains controls specifically designed for it. Third parties are likely to have additional control packages available.
- Use Silverlight windowless controls if you want to overlay viewable HTML content and controls on top of a Silverlight application.
- If you want to provide rich control functionality in cases when WPF is not available (such as when Microsoft .NET Framework 3.5 is not installed on the machine), you can embed a Silverlight application in Windows Forms and run it in the Web Browser control.
- Silverlight allows you to attach additional behaviors to existing control implementations. Use this approach instead of attempting to subclass a control.
- Silverlight supports only asynchronous calls to Web services.
- Silverlight calls use the '***OnError***' event handler for an application when exceptions occur in services, or when synchronous exceptions are not handled.
- Silverlight does not currently support SOAP faults exposed by services due to the browser security model. Services must return exceptions to the client through a different mechanism.
- Silverlight supports two file formats to deal with calling services cross-domain. You can use either a ClientAccessPolicy.xml file specific to Silverlight, or a CrossDomain.xml file compatible with Adobe Flash. Place

the file in the root of the server(s) to which your Silverlight client needs access.
- In Silverlight, you must implement custom code for input and data validation. Check documentation to verify whether this is true for later versions.
- Silverlight performs anti-aliasing for all UI components, so consider the recommendations in the Presentation section about snapping UI elements to whole pixels.
- Consider using ADO.NET Data Services in a Silverlight application if large amounts of data must be transferred from the server.
- Silverlight logs to an individual file in the user store for a specific logged-in user. It cannot log to one file for the whole machine.
- Silverlight supports the languages of C#, Iron Python, Iron Ruby, and VB.NET. Most XAML code will also run in both WPF and Silverlight hosts.

Additional Resources

- For official information on Silverlight, see the official Silverlight Web site at http://silverlight.net/default.aspx
- For Silverlight blogs, see http://blogs.msdn.com/brada/ and http://weblogs.asp.net/Scottgu/