Silk 1.0 - June 2011 Project Silk: Web Guidance Table of Contents

Here is the Table of Contents. At this point, we have drafts of the chapters that are linked below. These chapters are ready for your review and feedback.

- <u>Chapter 1: Introduction</u>
- <u>Chapter 2: Architecture</u>
- Chapter 3: jQuery UI Widgets
- Chapter 4: Design and Layout
- Chapter 5: HTML Templates
- Chapter 6: Application Notifications
- <u>Chapter 7: Modularity</u>
- Chapter 8: Communication
- Chapter 9: Navigation
- Chapter 10: Client Data Management and Caching
- Chapter 11: Server-Side Implementation
- <u>Chapter 12: Security</u>
- Chapter 13: Unit Testing Web Applications
- Glossary
- Appendix A: How-to Topic Listing
- Appendix B: Widget QuickStart
- <u>Appendix C: How to: Check UIElement Properties with Coded UI Test</u>
- <u>Appendix D: How to: Create Automation Negative case with Coded UI</u> <u>Test</u>
- Appendix E: How to: Create Web Client UI Test using Coded UI Test

Known issue: The graphics are not in their final format. They will be fixed before publication.

Community

To report documentation errors or provide feedback on this documentation, please send email to pagdoc@microsoft.com

Copyright © 2011 by Microsoft Corporation. All rights reserved.

Project Silk provides guidance for building secure cross-browser web applications that are characterized by rich graphics, interactivity, and a responsive user interface (UI) which enables an immersive and engaging user experience (UX). These applications leverage the latest web standards like HTML5, CSS3 and ECMAScript version 5, and modern web technologies such as jQuery, and ASP.NET MVC3.

The application performance is in-part due to the fast JavaScript engines of the modern standards based web browsers which enable developers to deliver applications whose experience and execution speed rivals that of a desktop application. Windows Internet Explorer 9 further enriches the user experience by taking full advantage of the modern PC hardware providing hardware accelerated graphics. High-definition videos are smooth, graphics are clearer and more responsive, colors are truer, and web applications are more interactive.

The Project Silk written guidance is demonstrated in the Mileage Stats Reference Implementation (Mileage Stats), a real-world customer facing web application that provides users the ability to track and compare their vehicles: fuel efficiency, usage, operation costs, as well as the scheduling of vehicle maintenance reminders.

Getting Started

This section describes how to install and start exploring the Project Silk guidance. You can download Project Silk from MSDN.

Prerequisites

This guidance is intended for web developers and assumes you have hands-on experience with ASP.NET MVC 3, CSS, HTML, JavaScript, and the jQuery libraries. There are a few important JavaScript concepts that Project Silk uses heavily that you need to be familiar with. If you are unsure of these concepts or have not used them in your own projects, please see the Further Reading section at the end of this chapter which provides links to resources for learning these.

- **Closures**. JavaScript closures are a powerful feature of the language that ensures an inner function always has access to the variables and parameters of its outer function, even after the outer function has returned.
- **jQuery selectors**. jQuery selectors and attribute selectors allow you to select all DOM elements or groups of elements for a given tag name, id, attribute name or value and allow you to manipulate them as a group or a single node.

System Requirements

This guidance was designed to run on the Microsoft Windows 7 or Windows Server 2008 operating system. It has been smoke tested on Windows Vista and XP.

Before you can compile and run the Mileage Stats application, the following must be installed:

- Microsoft Visual Studio 2010 Professional, Premium, or Ultimate edition
- Microsoft Visual Studio 2010 SP1
- Microsoft .NET Framework 4.0 (installed with Visual Studio 2010)
- ASP.NET MVC 3
- <u>Microsoft SQL Server Compact 4.0</u>
- <u>ADO.NET Entity Framework 4.1</u>. For more information about this Entity Framework release, see <u>EF 4.1 Released</u>.
- <u>NuGet</u>

Downloading and Installation

To download and install Project Silk's guidance which includes: source code for the Mileage Stats Reference Implementation and jQuery UI Widget QuickStart, this written guidance, and additional How To appendixes in .chm and .pdf formats, please following these steps:

- 1. Download Project Silk.
- 2. To extract the download, right-click the .exe file and then click **Run as administrator**. This will extract the source code and documentation into the folder of your choice.
- 3. Read the ReadMe file which contains instructions on installing external dependencies from NuGet and downloading the required external JavaScript libraries.

Spectrum of Web Applications

Before diving deep into the design and coding of these rich and interactive web applications, it would be prudent on our part to first survey the spectrum of web applications being built today. This exercise will better equip you to make technology and design choices that provide solutions for your scenarios.

The below four general types of web applications are categorized by their full page reload requirements and the client-side interactivity they provide. Each application type provides a richer experience than the one listed above it.

- **Static**. Web sites consisting of static HTML pages, CSS and images. As each page is navigated to, the browser performs a full page reload.
- Server rendered. Web application where the server dynamically assembles the page from one or more source files and can inject data values during the rendering. This type of application typically has some but not a lot of client-side code. As each page is navigated to, the browser performs a full page reload. ASP.NET MVC or ASP.NET Web Forms applications that don't make heavy use of client-side JavaScript are examples of server rendered web applications.
- **Hybrid design**. Web application has the same characteristics as the server rendered web application, except that it relies heavily on client-side code to deliver an engaging experience. This type of application has islands of interactivity within the site that do not require full page reloads to change the user interface, as well as some pages that require a full page reload. Mileage Stats is an example of a web application using a hybrid design.
- **Single page interface**. Web application where the user is only required to perform a full page load once. From that point on, all page changes and data loading is performed without a full page reload. Hotmail, Office Live, and Twitter are examples of single page interface web applications.

Characteristics of Modern Web Applications

While there are many types of modern web applications that address very different needs and scenarios, these applications tend to have the below characteristics in common.

- **Standards-based**. To have the broadest reach across multiple platforms, standards-based applications don't rely on proprietary implementations of HTML5 or CSS3.
- Interactive. Modern web applications keep the user engaged by providing constant, yet subtle feedback. This feedback can be in the form of user messages, animations to hide or show elements, mouse over UI element effects, drag and drop feedback, or the automatic refreshing of screen data. Interactive applications strive to provide feedback during application state or page changes. This feedback can be in the form of animating elements into new positions or animating of their opacity to provide fade in or fade out visual effects. Interactive applications leverage the fast JavaScript engines in modern browsers to perform their client-side tasks.
- Limit full page reloads. Modern web applications seek to limit the number of full page reloads. Reloads are much slower than a localized Ajax call to update a portion of the UI. Full page reloads also limits the ability to animate state or page changes. By not performing a full page reload; the user can be kept in context, providing a fluid experience as they navigate from one task to another.
- Asynchronous. Modern web applications use Ajax to dynamically load data, templates, or partial views as required instead of performing a full page reload to acquire data or HTML structure. The asynchronous loading of data allows the UI to stay responsive and keep the user informed while the data request is being fulfilled. This asynchronous on-demand loading also reduces application response time because requests can be tuned to only return exactly what is needed to fulfill the request.
- **Data management**. When applicable, modern web applications provide client-side data caching and pre-fetching to boost client-side performance. This enables the user interface to immediately respond to user input gestures because it does not have to make a call to the server for data. Data caching also serves to minimize the impact on server

resources increasing application scalability because fewer calls to the server are required.

Considerations for Building Modern Web Applications

Building a rich modern web application can be rewarding and fun. For developers or web software firms that have typically delivered server-centric applications, possibly with small amounts of JavaScript, embarking on a modern web application project will involve a paradigm change that should not be minimized or overlooked.

In the next two sections we will examine the skillsets a project team will need and the technologies used when building a modern web application.

Team Skillsets

Developing modern web applications requires a broad range of skills. Depending on your application requirements, your team will need expertise provided by people in the following roles:

- **Designer roles**: user experience, user interface, graphics
- Client-side developer roles: user interface, user interaction, test
- Server-side developer roles: website, business objects & logic, database, test

The composition of the web project team will vary from project to project based on the application's requirements and the team's resources. For example, on a large project each of the roles would probably be filled by a different person or possibly a team, while on a small project team members will likely fill multiple roles with augmentation from consultants as required to fill in the gaps.

On Project Silk the all of the above roles were filled by a lead developer, designer, server-side developer, and two client-side developers.

Project Silk also had a test team that consisted of a test manager, test program manager, and two software test engineers. The test team was responsible for cross-browser, deployment, performance, stress, and security testing. To accomplish this, the test team set up lab with Windows Server 2008 web servers that had different configurations, and client computers configured with different operating systems and installed browsers. These system where then used to perform daily automated and manual tests.

Technologies

This section will familiarize you with technologies and patterns used in building the Mileage Stats application. If any of these are new to you please review the Further Reading topics so that you will get the most from the guidance and will be able to understand the Mileage Stats JavaScript, HTML5, CSS3 and C# code.

Ajax

The web has been benefiting from the ability to replace full page reloads with Ajax calls for over 10 years now. But given the advances in standards, browsers adherence to those standards, and the arrival of powerful, cross-browser JavaScript frameworks, we have all the tools necessary to build highly engaging client-side experiences using the latest web standards and technologies.

The Ajax technology facilitates a paradigm change in web development from the traditional full page reload model of server-centric applications to rich and responsive client-centric applications. The client receives data and updates the user interface (UI) using JavaScript. Bandwidth requirements are minimized because the server responds to requests by returning just the requested data instead of HTML pages with data. The application runs faster because the data requests take less time to complete, and the UI is quickly updated without a full page reload. Asynchronous calls are essential to keeping interactive and immersive applications responsive from the user's perspective.

JavaScript

This section will be completed soon.

jQuery

This section will be completed soon.

Modernizr

This section will be completed soon.

ASP.NET MVC Razor

Razor is a codename for the new view engine in ASP.NET MVC 3. A view engine is a mechanism that is plugged into the ASP.NET MVC framework by which MVC views are rendered. Razor is a view engine that is optimized for HTML rendering that is compact, expressive, and easy to learn.

The Razor syntax is clean and concise, easy to learn, and Visual Studio includes IntelliSense and code colorization for Razor syntax. Additionally, Razor views can be unit tested without requiring that you run the application or launch a web server.

For a more detailed overview about Razor, see "*ASP.NET MVC 3 Overview, The Razor View Engine*" in the further reading section.

Dependency Injection

Dependency injection is a variant of the Inversion of Control design pattern. Dependency injection containers reduce the dependency coupling between objects by providing a facility to instantiate instances of classes and manage their lifetime based on the configuration of the container. During the creation of objects, the container injects any dependencies that the object requires into it. If those dependencies have not yet been created, the container creates and resolves their dependencies first.

Dependency injection provides applications several benefits:

- Reduced coupling as classes and components don't have to locate dependencies or manage their lifetimes.
- Improved testability because dependencies can be easily substituted with mocked implementations
- Improvedflexibility and maintainability as dependencies can be swapped out easily.

ASP.NET MVC 3 provides better support for applying Dependency Injection and for integrating with Dependency Injection or Inversion of Control containers. For a complete coverage of Dependency Injection in ASP.NET MVC 3 see, "*ASP.NET MVC 3 Overview, Dependency Injection Improvements*" in the further reading section.

Exploring this Guidance

The purpose of this guidance is to show you how to plan, design, and build a rich interactive web application that your users will enjoy using. This guidance includes written documentation in the form of topical chapters, Mileage Stats Reference Implementation, and jQuery UI Widget QuickStart.

What is a reference implementation? Reference implementations are a means to an end, with a singular goal of demonstrating solutions to real-world scenarios and challenges. They are by nature not packed with a lot of features, but focus on clearly communicating coding patterns, techniques, application architecture, and unit testing necessary to demonstrate and support the accompanying written guidance.

- **Documentation**. This guidance provides an architectural overview of rich web applications and the accompanying detailed chapters that cover the design, concepts, patterns, security, testing, and implementation of the Mileage Stats ASP.NET MVC application.
- **Reference Implementation**. Comprehensive sample application demonstrating a real-world, interactive, cross-browser, consumer facing, rich web application. The reference implementation is intentionally incomplete but does illustrate the core concepts, design patterns, security requirements, web technologies, and unit testing necessary to be successful.
- **QuickStart**. The guidance includes the source code and documentation for understanding how to use and develop jQuery UI Widgets.

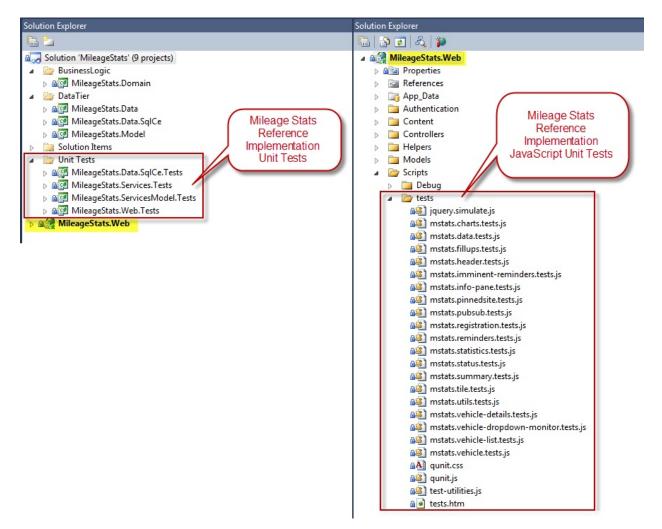
Exploring the Mileage Stats Reference Implementation

The Mileage Stats Reference Implementation (Mileage Stats) is an ASP.NET MVC application that enables users to track and compare various metrics about their vehicles including fuel consumption, fuel costs, miles driven, and maintenance reminders. The application is a multi-page interactive web application where the pages are rendered without requiring a full page reload. This creates the illusion of a desktop application. The lack of full page reloads enables rich UI transitions between states (pages) and the application runs very fast because of the client-side data caching and some data pre-fetching.

Much of the effort in building Mileage Stats was applied to the usability and rich interactivity of the experience. Animations were included to enhance the enjoyment of the site and Ajax is used to keep the interface responsive and immersive. A great deal of care was also taken to ensure the client-side JavaScript was modularized for maintainability. To accomplish these design goals, the JavaScript code was structured using the jQuery UI Widget Factory. Widgets allow breaking the UI into small discrete stateful objects; providing a clean separation of responsibilities and concerns.

Mileage Stats is partitioned into three layers: data, business, and web spread across five Visual Studio projects pictured on the left side of the below image. In addition, Mileage Stats includes four unit test projects for testing the five C# projects and a suite of JavaScript unit tests, pictured on the right side of the below image.

Mileage Stats unit tests



The design and implementation of the above Mileage Stats solution is the subject of this guidance and will be covered in the remaining chapters. Now let's walk through the Mileage Stats application from a user's perspective.

Using the Mileage Stats

Unauthenticated users accessing the Mileage Stats website will be redirected to the landing page to sign in. Mileage Stats uses 3rd party OpenID providers for user authentication. Mileage Stats supports deep linking so that previously authenticated users returning to the website can go directly to any page.

The Mileage Stats introduction tutorial video can also be viewed from the landing page.

For Internet Explorer 9 users, Mileage Stats provides a customized pinned sites experience that is accessed from the landing page. The below image shows the site in its pinned state running in the pinned sites customized chrome of Internet Explorer 9. The menu, favorites bar, command bar, and status bar have been removed from view. The back and forward buttons are rendered with a custom color and the site's favicon is displayed to the left of the back button.

Landing page



The first time a new user logs into the Mileage Stats application the summary pane will display the Complete your Registration form pictured below. This form will continue to be displayed in the summary pane until the user clicks the Save button. Further edits to the users profile can be made by clicking the Profile link at the top right of the browser window.

3rd party authentication providers do not uniformly expose their user data to applications requesting authentication services. For example, a user may have an OpenID account, but the Mileage Stats application may not be able to request information from the provider like the users first and last name to populate the below Display Name field. Our team's UX designer did not want to force a new user to complete a form, just to use Mileage Stats after authenticating. So we implemented a non-intrusive form for collecting the new users name, country and postal code. The new user can immediately use the Mileage Stats application and can complete the registration information at their leisure.

First time logging in



The Dashboard provides a summary view of the user's vehicles. From here the user can add a new vehicle, drill down to more detail for a vehicle, and can see maintenance reminders that are overdue or due soon.

There is a navigation bar at the top of the browser window that provides toplevel navigation to the Dashboard, Charts, or Profile pages and a Sign Out link to sign out of application.

Dashboard



A high value scenario for this guidance was to demonstrate fluid and rich UI transitions, and animations.

The below image shows the application transitioning from the Dashboard (two

column vehicle listing) to the Details view (single column vehicle listing) in response to the user clicking on the Details button in Hot Rod's vehicle tile.

The below image demonstrates the synchronization of animated opacity changes and UI element movement as the summary pane, vehicle tiles, and info pane animate into their respective positions.

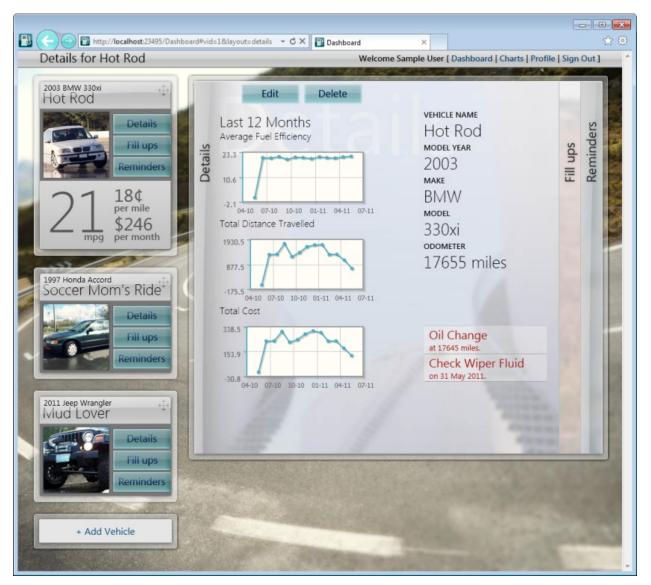


Transitioning from the Dashboard to Details

The Details view displays aggregated monthly values for: fuel efficiency, distance travelled, and fuel cost. The user is able to quickly see trends in their

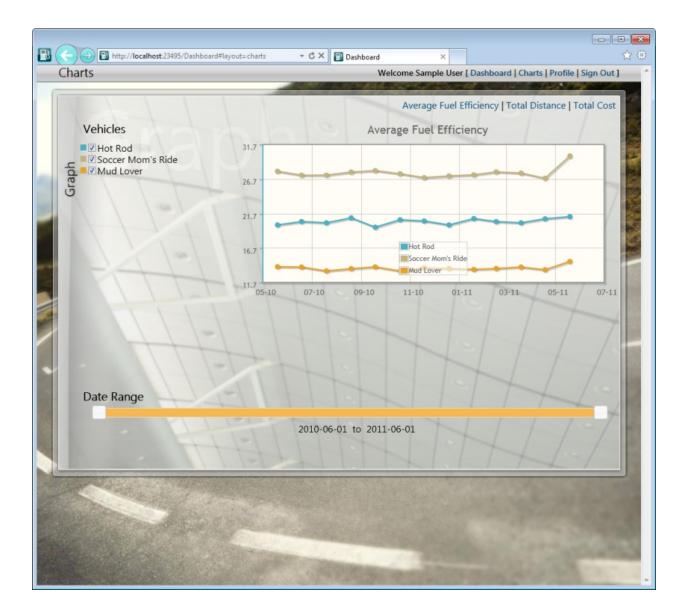
vehicle usage as well as overdue maintenance reminders. The Details view allows the user to edit or delete the vehicle, as well as navigate to the fill ups and reminders views.

Details



The Charts page provides three charts which allow the user to easily compare their vehicles fuel efficiency, distance travelled, and cost. The displayed data can be filtered by vehicle and date range. The data displayed in these charts is pre-fetched and cached, providing a very fast user experience.

Charts



Exploring the Documentation

This guidance covers a wide range of topics that includes: planning and designing your application, understanding and writing jQuery UI Widgets, writing server-side code that supports the client-side application, patterns and concepts used in JavaScript code, data and caching, securing, and testing your application.

The printed book contains all of the below chapters and Appendixes A and B. The remaining appendixes can be read online and are also included with the Project Silk download.

The written and online documentation includes the following:

- **Chapter 2, "Architecture."** This chapter explains the Mileage Stats client-side architecture by studying how its structure, modularity, communication, navigation, and data relate to one another.
- **Chapter 3,** "**jQuery UI Widgets.**" An understanding of jQuery UI Widgets is critical to comprehending this guidance and the Mileage Stats application because Mileage Stats makes heavy use of widgets to modularize its JavaScript. This chapter provides ample instruction on widget fundamentals, lifetime, creation, events, properties and methods, and inheritance.
- **Chapter 4, "Design and Layout."** This chapter explains the importance of an iterative design process and the roles different team members fulfill. After a survey of user experience and user interface design considerations, we will walk through the design and building of the Mileage Stats application and how these considerations influenced the application.
- **Chapter 5, "HTML Templates."** This chapter discusses how an interactive application like Mileage Stats can manage client-side HTML changes with having to fully reload the page each time the user navigates or completes a task.
- **Chapter 6,** "**Application Notifications.**" Web applications that users consider responsive have one thing in common; they provide appropriate and timely feedback to the user. In this chapter we will show how to provide unobtrusive feedback to the user and how to implement notifications on the desktop with the Pinned Sites API.
- Chapter 7, "Modularity." Rich and interactive web applications can

require a fair amount of JavaScript coding. Modularizing your JavaScript makes your code easier to maintain and evolve. In this chapter we will explain how the Mileage Stats JavaScript was modularized using jQuery UI Widgets and JavaScript objects.

- **Chapter 8, "Communication."** This chapter explains how communication between widgets and JavaScript objects was implemented in Mileage Stats. Topics such as loose communication that use the, "publish and subscribe" metaphor, events, and inter-widget communication are covered.
- **Chapter 9, "Navigation."** Rich web applications support client-side transitions, animations, as well as deep linking. Web users expect their browser back-button to function as expected. This chapter explains the challenges client-side web developers face maintaining the browser history when using Ajax calls instead of full page reloads. In addition, the Mileage Stats state change management is fully explained.
- Chapter 10, "Client-Side Data Management and Caching." This chapter covers how Mileage Stats JavaScript objects request, and send data as well as the data manager façade that performs the Ajax calls to the server and provides transparent data caching.
- **Chapter 11, "Server-Side Implementation."** This chapter covers the Mileage Stats ASP.NET MVC (module view controller) application and the other server-side components and the services they provide to support the client-side JavaScript objects. Coverage takes you from the database, through the repositories, to the business objects that provide data validation and data shaping services to the controllers that consume their data and render it to the client.
- **Chapter 12, "Security."** Web security is critical to consumer confidence. Poor security can result in compromising your customer's data, your own data and intellectual property. This chapter covers some of the security features of the ASP.NET MVC platform and security features in Mileage Stats that provide countermeasures against the relevant threats for authentication, input validation, anti-forgery, and JavaScript Object Notation (JSON) hijacking.
- **Chapter 13, "Unit Testing Web Applications."** Unit tests are long-term investments that provide the development team confidence when refactoring or evolving the application, and when external dependencies such as versions of jQuery or a 3rd party jQuery UI plug-in are updated.

This is a getting started chapter for JavaScript and ASP.NET MVC code.

- **Appendix A, "Glossary."** This appendix provides a concise summary of the terms, concepts, and technologies used in the Mileage Stats application and this guidance.
- **Appendix B, "How-do-I Topics."** This appendix is an *at-a-glance* list of topics to facilitate locating a section of guidance by topic.
- Appendix C, "Widget QuickStart."
- Appendix D, "How to Check UIElement Properties with Coded UI Test."
- Appendix E, "How to Create Automation Negative case with Coded UI Test."
- Appendix F, "How to Create Web Client Test Automation with Coded UI Test."

Exploring the QuickStart

The Widget QuickStart illustrates the way Mileage Stats uses the jQuery UI Widget Factory to create maintainable widgets that implement client-side behavior.

Further Reading

For information on the designer role, see Chapter 4, "Design and Layout."

For information on unit testing, see Chapter 13, "Unit Testing Web Applications."

Stefanov, Stoyan. JavaScript Patterns, O'Reilly Media, 2010

Crockford, Douglas. JavaScript: The Good Parts, O'Reilly Media, 2008

ASP.NET MVC 3 Overview, The Razor View Engine http://www.asp.net/mvc/mvc3#BM_TheRazorViewEngine

ASP.NET MVC 3 Overview, Dependency Injection Improvements http://www.asp.net/mvc/mvc3#BM_Dependency_Injection_Improvements

Community

Project Silk's community site is: <u>http://silk.codeplex.com</u>. On this community site, you can post questions, provide feedback, or connect with other users for sharing ideas. Community members can also help Microsoft plan and test future offerings.

The community site also has links to tutorial videos, MSDN content, the Readme, and this documentation in .chm and .pdf formats.

| Community

To report documentation errors or provide feedback on this documentation, please send email to pagdoc@microsoft.com

Copyright $\ensuremath{\mathbb{C}}$ 2011 by Microsoft Corporation. All rights reserved.

Silk 1.0 - June 2011 Architecture

Introduction

The Mileage Stats Reference Implementation (Mileage Stats) is a cross-browser, ASP.NET Model View Controller (MVC) application that takes advantage of the features of modern browsers. The application offers two types of user experiences:

- 1. **A traditional website experience**. In this approach, a form post and page reload are executed each time a button or hyperlink is clicked.
- 2. **A rich website experience**. In this approach, the initial page is loaded once, and server requests are only made when new data is required or updated. In addition to other user-friendly features, the lack of a full-page reload enables the animation of client-side state changes.

The rich website approach provides a superior experience for the user, as the application feels more responsive and more like a desktop application. But because some users do not have scripting enabled or available on their user agent (web browser or accessibility tool, such as a screen reader), which is necessary for the partial-page reloads, we must support the traditional website experience for them.

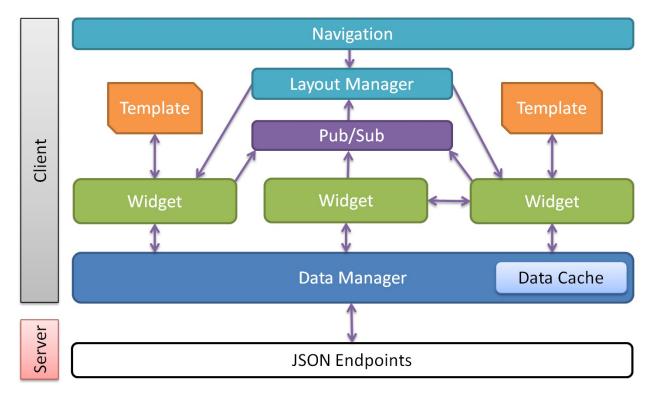
In the traditional approach, the ASP.NET MVC controllers are responsible for acquiring data and returning a built-up view that consists of HTML structure and data. In the case of the rich website experience, we perform asynchronous data requests and the controller returns only data. The client then renders the data in the user interface (UI) without reloading the whole page.

Supporting both these experiences introduces complexity that requires careful planning on both the client and server sides to ensure that the application is responsive, maintainable, has a clean separation of concerns, and is testable.

You should determine early in the design phase which experience the user should expect in each browser and browser version the application will support. If you choose to support older browsers, you may limit your technology choices and affect the run-time experience of the application. Shims and polyfills, such as those that provide HTML5 support, are available for adding support for some technologies in older browsers, but these come at the cost of additional dependencies (see "Further Reading" at the end of the chapter to learn more about shims and polyfill solutions). Making decisions about which technologies you will need to support early on allows you to establish realistic expectations for users and project stakeholders.

This chapter provides a high-level map of the Mileage Stats client-side architecture, and is divided into five areas of discussion: structure, modularity, communication, navigation, and data.

- **Structure** refers to client-side HTML structure and manipulation which is represented below as the Template.
- **Modularity** refers to how a clean separation of JavaScript objects helps create a more maintainable application which is represented below as the Widget.
- **Communication** defines how JavaScript objects communicate, and is represented by the Pub/Sub.
- **Navigation** explains how to manage user gestures and coordinate animations, and is represented by the Navigation and Layout Manager.
- **Data** provides guidance for client-side data requests and data caching and is represented below as the Data Manager.



Mileage Stats client architecture

In this chapter you will learn:

- Options and strategies for getting the right HTML to the client.
- The advantages of modular code and techniques for using jQuery UI widgets.
- How the pub/sub pattern can be used for loosely coupled communication.
- How to solve browser history and back-button problems when the site doesn't perform full-page reloads.
- How a loosely coupled data layer can simplify caching for client side data requests.
- How the Mileage Stats team solved a number of challenges related to structure, modularity, communication, navigation, and data.

The technologies and libraries discussed in this chapter are JavaScript, jQuery, jQuery UI Widgets, and jQuery Back Button & Query Library (jQuery BBQ).

Structure

Websites like Mileage Stats provide an engaging user experience when viewed using modern browsers with JavaScript enabled. The site can also be viewed without JavaScript enabled and will function when viewed in an older browser.

To provide an engaging, responsive, and interactive experience, the application needs to manage client-side structure changes without performing full-page reloads. This requires client-side loading, creation, and replacement of HTML fragments or pages.

To support both rich and traditional user experiences, the Project Silk team chose to have the web server generate the initial HTML; then, after using JavaScript to detect the browser capabilities, we enhance the user experience by replacing the server-generated HTML structure with a client-side version in capable browsers. Elements replaced include portions of HTML, button actions, and CSS classes. Enhancement can mean adding animation, page transitions, or Ajax functionality to client-side elements. Client-side enhancement of servergenerated HTML is called progressive enhancement. Progressive enhancement enables and adds features to the client-side experience based on browser capabilities.

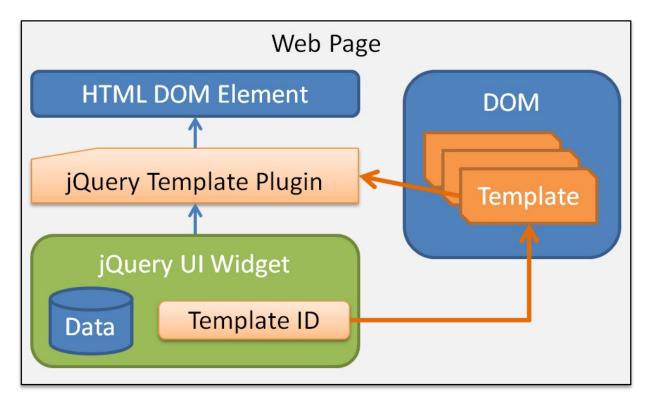
After the initial enhancement of the server-generated HTML, the client-side JavaScript responds to user gestures, requests data, and initiates UI changes without posting back to the server.

Client-side UI structure can be generated with JavaScript, loaded on demand from the server, or rendered by a plug-in or a library. Initially, the team tried ondemand loading of granular HTML fragments from the server. This approach was motivated by the team's desire to limit the creation of HTML to a single location. However, this approach failed to provide the desired result, so the team changed tactics and used jQuery templates instead. See Chapter 5, "HTML Templates" for a full explanation of this choice.

jQuery Templates

jQuery templates are HTML markup with inline JavaScript expressions that are used to populate values in the markup. The jQuery Template plug-in applies data to the template and renders the output into the DOM. Mileage Stats uses jQuery UI widgets to coordinate the data retrieval, applying it to the template using the plug-in, and overwriting the DOM element.

jQuery template rendering



The data can be a single object or an array of objects. jQuery templates separate structure and data, making the application easier to code, test, and maintain.

If you use ASP.NET MVC or ASP.NET Web Forms, you can use the rendering engine to dynamically create or modify the jQuery template while it's being rendered. Mileage Stats uses this capability to inject URLs and data-dash attributes into the templates at render time.

Mileage Stats loads all jQuery templates as part of the initial page load. Preloading templates simplifies the client-side application and provides much faster client-side rendering than on-demand loading of templates provides. For more information on the jQuery Template plug-in and authoring templates, see "jQuery Templates" in the "Further Reading" section. For more information on jQuery templates in Mileage Stats, see Chapter 5, "HTML Templates."

Modularity

Modularized code simplifies the overall application, establishes clear boundaries of responsibility, provides separation of concerns, increases testability, eases maintenance, and enables reuse. The modularization of code in Mileage Stats is achieved by composing client-side JavaScript into jQuery UI widgets and JavaScript objects.

jQuery widgets are objects attached to page elements that supply services for managing lifetime, state, inheritance, theming, and communication with other widgets or JavaScript objects. Objects in Mileage Stats belong to one of the following functional categories:

- **UI**. Includes these jQuery UI widgets: vehicle, vehicle list, information pane, vehicle details, vehicle fill ups, vehicle reminders, registration, statistics, summary, status, header, and charts.
- **Behavior**. Includes the tile and layout manager widgets, and JavaScript objects for pinned sites and validation.
- **Infrastructure**. Includes JavaScript objects for data access, caching, and pub/sub messaging.

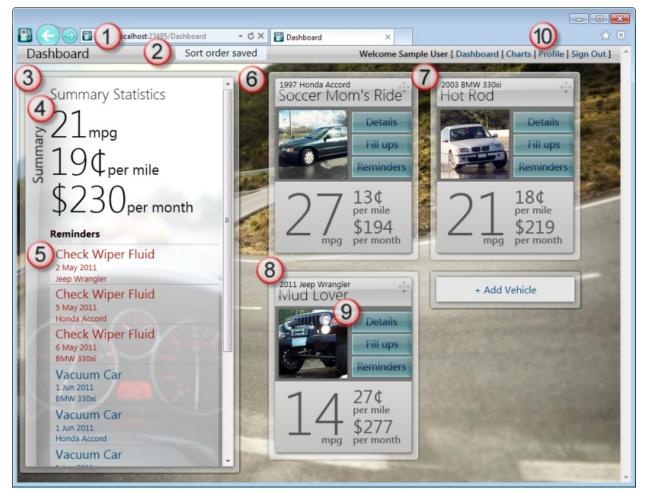
The jQuery widgets that compose the Mileage Stats Dashboard are pictured in the image below. The complexity of the application demonstrates the need for modularization. By breaking the implementation into discrete, loosely coupled objects, the client-side code is much easier to understand, author, maintain, test, and debug.

- 1. **Pinned sites**. JavaScript object provides the pinned sites implementation for Windows Internet Explorer 9.
- 2. **Status widget**. Provides management and display of user notification messages.
- 3. **Summary widget**. Acts as a container, managing its child registration, statistics, and reminders widgets.
- 4. Statistics widget. Displays summary statistics for all vehicles.
- 5. **Reminders widget**. Lists overdue and upcoming maintenance reminders. Manages the action of clicking on a reminder.
- 6. **Layout manager widget**. Services navigation requests and coordinates UI layout changes.
- 7. Vehicle list widget. Displays the vehicle tiles in a one-column or two-

column listing. Invokes the child widget animation when required and controls when child widgets are displayed in expanded or contracted view.

- 8. **Tile widget**. Provides drag-and-drop capability for the child vehicle widget.
- 9. **Vehicle widget**. Displays vehicle information in expanded or contracted view. Manages the actions of each button.
- 10. **Header widget**. Provides top-level navigation and user name display. Manages actions when a hyperlink in the header is clicked.

Modularization in Mileage Stats



For more information on modularity in Mileage Stats, see Chapter 7, "Modularity." For more information on jQuery UI widgets see Chapter 3, "jQuery UI Widgets" and Chapter 7, "Modularity." For more information on pinned sites, see Chapter 6, "Application Notifications."

Communication

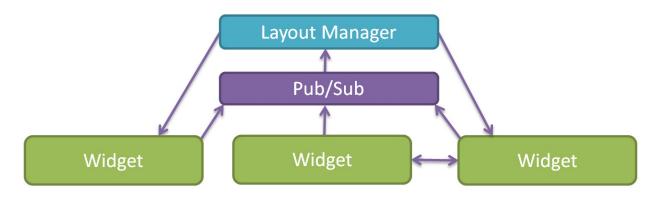
jQuery widgets and JavaScript objects help you modularize your code, but these objects are not isolated solitary islands; rather they are small objects that work together to form the complete application. Well-defined communication between objects is critical not only from a functional view, but from an architectural view as well.

If not carefully planned, communication between objects can lead to tight coupling and undesirable dependencies. Mileage Stats objects communicate *directly* with one another, or *loosely* by using a publish and subscribe pattern (pub/sub).

Direct Communication

Direct widget communication is typically reserved for high-level widgets controlling lower-level widgets, such as when the layout manager tells a widget to hide or show itself.

Layout manager and pub/sub



Loose Communication

Pub/sub is a messaging pattern that enables loose communication between publishers and subscribers. When a message is published, zero or more subscribers will be notified. A pub/sub object manages communication, relieving the publishers and subscribers of needing direct knowledge of one another. Pub/sub messages are individually defined and can optionally contain a payload.

The pub/sub pattern provides clean separation between the object invoking the action and the object that handles the action. This separation allows the publisher or subscriber's internal implementation to evolve without affecting the other.

Mileage Stats has its own pub/sub implementation that provides for loose communication. For example, the **Status** widget subscribes to the **status** message. The **status** message has a payload that contains message, type, duration, and priority values. Publishers of the **status** message provide these values when publishing this message.

Mileage Stats widgets have publish and subscribe functions passed in their options object during construction to decouple them from the pub/sub implementation.

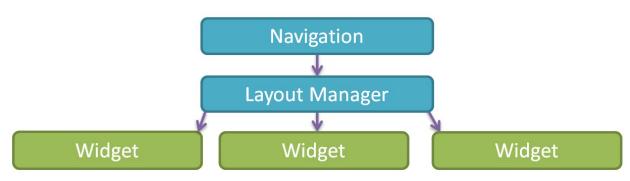
For more information about the pub/sub implementation in Mileage Stats, see Chapter 8, "Communication."

Navigation

Rich client-side web applications like Mileage Stats do not perform full-page reloads each time a button or hyperlink is clicked. Instead, client-side application code handles these events.

The jQuery BBQ plug-in (Back Button & Query Library) is responsible for providing address bar URL changes. Changing the address bar URL performs two functions. First, it allows users to bookmark addresses into the application so that they can return directly to that state. This is also known as deep linking. Second, it enables the browser history and back button to perform as the user expects.

The Mileage Stats layout manager is a widget that works in conjunction with the BBQ plug-in to service navigation requests. It subscribes to the BBQ plug-in **hashchange** event, and initiates layout changes based on address bar URL changes.



Navigation and layout manager

Along with hiding and showing UI elements, the layout manager is also responsible for initiating UI animations during navigation. The layout manager does not perform the animation, but sequentially calls methods on one or more lower-level widgets, resulting in an engaging UI transition.

As part of the layout manager's top-level widget responsibilities, it subscribes to several pub/sub messages and invokes lower-level widget data refresh methods when those messages are published.

Data

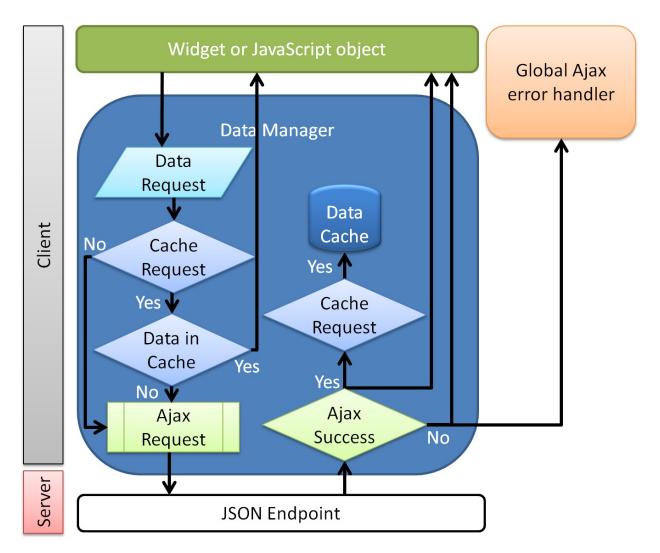
When designing your client-side data architecture, several key decisions will impact application performance, maintainability, and browser support. Will data requests flow through a central object or will objects make direct calls to the server? Will data be cached, and if so, how much? Will data be pre-fetched, and if so, how much? Answers to these questions will vary based on your application's specific requirements.

In Mileage Stats, all data requests are made via Ajax and are routed through the data manager. Having a single object handle data requests simplifies the client-side calling code, improves application testability, and facilitates cleaner application evolution when client-side libraries advance or change. The single data manager object also affords you the opportunity to implement client-side data caching in a central location. Data is cached in a JavaScript object, rather than using HTML5 local storage or similar APIs, in order to meet the cross-browser requirements of the application.

Mileage Stats pre-fetches chart data during the initial page load, enabling instant application response when the user navigates to the charts page. Whenever data is returned from the server, it's cached. This can make the application more scalable because repeated requests to the server for the same data are no longer necessary, requiring less server processing per user.

Widgets and JavaScript objects request their data from the data manager. The data manager services the request, first checking if the request should be cached, and if so, checks the cache before making a call to the server. Upon successful completion of the request, the returned data will be added to the cache, and then passed to the calling widget. If an error occurs, the error will be returned to the calling widget.

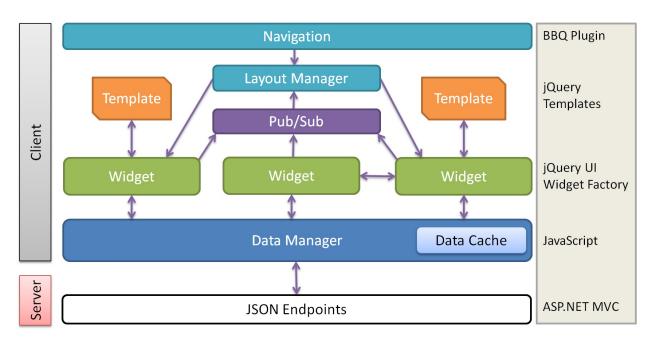
Data request



For in-depth coverage of data management and caching, see Chapter 10, "Data, Caching, and Validation."

Summary

Building a rich web application that reduces the number of full-page loads, includes animations, and is responsible for updating the UI dynamically requires a thoughtful approach to managing structure, modularity, communication, navigation, and data. This chapter provided a high-level view of the Mileage Stats client-side application architecture. The following image shows the clientside objects and their implementation mapped to libraries or frameworks.



Mileage Stats client architecture technology map

Further Reading

For more information on jQuery UI widgets see Chapter 3, "jQuery UI Widgets" and Chapter 7, "Modularity."

For more information on jQuery templates in Mileage Stats, see Chapter 5, "HTML Templates."

For more information on pinned sites, see Chapter 6, "Application Notifications."

For more information on modularity in Mileage Stats, see Chapter 7, "Modularity."

For more information about the pub/sub implementation in Mileage Stats, see Chapter 8, "Communication."

For more information about the libraries and guidelines discussed in this chapter, see the following:

- jQuery: http://jquery.org
- jQuery Templates: <u>http://api.jquery.com/category/plugins/templates/</u>
- "jQuery BBQ: Back Button & Query Library" on Ben Alman's blog: <u>http://benalman.com/projects/jquery-bbq-plugin/</u>
- "Filling the HTML5 Gaps with Polyfills and Shims" from Rey Bango's MIX11 session: http://channel9.msdn.com/Events/MIX/MIX11/HTM04

| Community

To report documentation errors or provide feedback on this documentation, please send email to pagdoc@microsoft.com

Copyright © 2011 by Microsoft Corporation. All rights reserved.

Silk 1.0 - June 2011 **jQuery UI Widgets**

Introduction

When building rich client-side web applications, some of the visual elements on the page will naturally take on roles, responsibilities, and state. As more of these elements are added to the page, complexity will increase, so it's important for the design to support a maintainable codebase. Maintainable solutions have at least two important characteristics: they have unit tests, and they have an intentional design that plays to the strengths of the platform, language, and key parts of the environment.

The web browser is the platform. JavaScript represents the language and various JavaScript libraries represent key parts of the solution. Among other benefits, libraries like jQuery and jQuery UI are used to:

- Address typical challenges like browser compatibility
- Provide consistency for AJAX interactions, animations, and events
- Assist in creating a maintainable codebase through modularity

According to the official jQuery UI project, "[it] provides abstractions for lowlevel interaction and animation, advanced effects and high-level, themeable widgets, built on top of the jQuery JavaScript Library, that you can use to build highly interactive web applications." A central concept in the visual parts of jQuery UI is the *widget*. Widgets are objects attached to page elements that supply services for managing lifetime, state, inheritance, theming, and communication with other widgets or JavaScript objects.

Even though they have a number of additional features on top of typical jQuery plugins, it's important to know a widget *is* a jQuery plugin. This may not be obvious because they are defined differently, but widgets are used the same way you use official jQuery methods and most custom plugins. Sometimes a plugin is sufficient and other times a widget is more appropriate. When you need to apply behavior or state to *individual* elements and need to communicate between elements, widgets provide a number of capabilities you would otherwise have to write yourself. This chapter illustrates the use of these capabilities.

In this chapter you will learn:

- How to define and apply widgets
- How to manage the lifetime of widgets

- How to define default options that permit overrides and change notifications
- How to use options for decoupling behavior and facilitating event subscriptions
- How to use private methods to improve the readability of the code
- How to define and use public methods, properties, and events
- How to inherit from a base widget

The technologies discussed in this chapter are jQuery Plugins and the jQuery UI Widget Factory. The code examples used in this chapter come from the Widget QuickStart included with Project Silk. For more information, see Appendix B: Widget QuickStart.

Widget Fundamentals

If you know how to use jQuery, you know how to use a widget. However, before you can use a widget, it has to be defined. Once it has been defined it can be applied to elements. Widgets are defined using the widget factory. When the widget factory is invoked, it creates a *widget method* on the jQuery prototype, **\$.fn** – the same place plugins and other jQuery functions are located. The widget method represents the primary interface for applying the widget to elements and using the widget after it's applied.

Defining a Widget

The dependencies for a widget can be fulfilled with script references to the <u>CDN locations</u> for jQuery and jQuery UI. Widgets often reside in their own .js file and are wrapped in a self-executing function. This wrapper creates a JavaScript closure which prevents new variables from being globally scoped. A single solution should prevent any more than one global object according to well-accepted JavaScript practices. The **\$** and **undefined** arguments reestablish their default expectations inside the closure in case another script previously defined them as something else.

JavaScript

Copy Code

```
// Contained in jquery.qs.tagger.js
(function($, undefined) {
    $.widget('qs.tagger', {
        // definition of the widget goes here
    });
}(jQuery));
```

The call to **\$.widget** invokes the widget factory which makes the widget available to use. The first argument, **qs.tagger**, is the widget's namespace and name separated by a period. The name is used as the name of the widget method placed on the jQuery prototype. The second argument, called the *widget prototype*, is an object literal that defines the specifics of the widget. The widget prototype is stored directly on the jQuery object under the namespace provided: **\$.qs.tagger**.

Using a Widget

Once a widget has been defined, it's ready to be applied to elements. To apply the widget to the matched elements, invoke the widget method just like you would other jQuery methods. The following code shows how to apply the tagger widget to all **span** elements with a **data-tag** attribute.

JavaScript

Copy Code

```
// Contained in startup.widget.js
$('span[data-tag]').tagger();
```

Because the widget method is used as the primary interface to the widget, it's not only called when initially applying the widget to the element, it's also used for calling methods and reading and writing options and properties on the widget. When widgets are applied to elements, an instance of the widget prototype is created and stored inside each element. This is how the widget factory knows if a widget has already been attached to an element so it can take the correct action in subsequent calls.

Managing Lifetime

There are three phases of a widget's lifetime that you can control: creation, initialization, and destruction.

Creation

The first time the widget is applied to an element, the factory calls the widget's **_create** function. Method names preceded with an underscore have private scope by convention, which means they only expect to be invoked from inside the widget. The following code shows the **_create** method in the **infobox** widget.

JavaScript

```
// Contained in jquery.qs.infobox.js
_create: function () {
   var that = this,
      name = that.name;
   that.infoboxElement = $('<div class="qs-infobox" /
   that.infoboxElement.appendTo('body')
   .bind('mouseenter.' + name, function () {
      mouseOverBox = true;
   })
   .bind('mouseleave.' + name, function () {
      mouseOverBox = false;
      that.hideTagLinks();
   });
},
```

- The **_create** method is the most appropriate place to perform a number of common tasks including the following:
- Adding classes to various elements in the widget is the recommended way to apply styling, layout theming and more to the widget.
- **Storing references** can increase performance when a particular set of elements are used from a number of methods. Simply create object-level variables for them once and all other methods can use them. This is an accepted jQuery performance best practice.
- **Creating elements** is common for widgets that have requirements like animations, effects, styling, accessibility, and cross-browser compatibility. As an example, consider the **div.qs-infobox** element created by the **infobox** widget.

• **Applying other widgets** is recommended during creation when you need them available as soon as possible. Even if your widgets don't require each other, consider using the official jQuery UI widgets from inside yours to add useful behaviors and interactions.

Initialization

• While the **_create** method is only called when the widget is first applied to the element, the **_init** method is called each time the widget method is called with no arguments or with options. When the widget is applied to the element the first time, **_init** is called after **_create**. When the widget method is called after the widget has been attached, only **_init** will be called. The **_init** method is the recommended place for setting up more complex initialization and is a good way to give the widget a way to reset. Although, it's not uncommon for widgets to not implement an **_init** method.

Destruction

• The widget's **destroy** method is used to detach a widget from an element. The goal of the **destroy** method is to leave the element exactly like it was before the widget was attached. Therefore, it's not surprising the common tasks are to remove any added classes, detach any added elements and destroy any initialized widgets. Here is the **destroy** method for the tagger widget.

JavaScript

Copy Code

```
// Contained in jquery.qs.tagger.js
destroy: function () {
    this.element.removeClass('qs-tagged');
    // if using jQuery UI 1.8.x
    $.Widget.prototype.destroy.call(this);
    // if using jQuery UI 1.9.x
    //this._destroy();
}
```

The last part calls the widget's base implementation of **destroy** and is a recommended practice. The base **destroy** will remove the instance of the widget from the element and unbind all namespaced event bindings, which are covered later in the chapter.

Defining Options

Options give widgets the ability to have state that is public, readable, writable, and callable. Options are automatically merged with the widget's default options during creation and the widget factory supports change notifications when option values change. In principle, you should be able to save the options on a widget, remove the widget from memory, recreate the widget with the saved options and have the same widget you started with.

Options are defined in the **options** property of the widget prototype as shown below in the **infobox** widget.

JavaScript

Copy Code

```
// Contained in jquery.qs.infobox.js
$.widget('qs.infobox', {
    options: {
        dataUrl: ''
        maxItems: 10,
    },
    ...
```

To override default options during the creation of the widget, pass them in as an object literal to the widget method as shown below in the startup code.

JavaScript

Cc

```
// Contained in startup.widget.js
var infobox = $('body').infobox({
    dataUrl: 'http://feeds.delicious.com/v2/json/popu]
});
```

This can be done as many times on an element as needed. The options will always be merged with the options already in the widget.

To read the options from inside the widget, use the **options** property directly as shown on the last line below.

JavaScript

```
// Contained in jquery.qs.infobox.js
displayTagLinks: function (event, tagName) {
    var i,
        that = this,
        options = that.options,
        url = options.dataUrl + tagName + '?count=' +
        ...
```

Reading the values directly off of **options** is acceptable when reading values from inside the widget, but you should not use this approach when changing the value of options. Instead, use the **option** method (without an 's').

JavaScript

Copy Code

```
// Code illustration: not in QuickStart
var max = this.option('maxItems');
this.option('maxItems', max + 4);
```

The **option** method is called with one argument when reading the option's value, two arguments when setting a value and a single object hash when setting more than one option. The **option** method should always be used to change the value of options so change notifications will work as expected. Changing the option directly on the **options** property bypasses the notification mechanism.

When Options Change

If the widget needs to react to an option's value being changed, it should use the **_setOption** method. This method is called by the widget factory just after the value has been set on the **options** property. The Widget QuickStart doesn't have a need for **_setOption**, but if the number of links in the **infobox** widget were configurable by the user, as an example, the widget might need to adjust the size of the box when **maxItems** changes.

JavaScript

Cop:

```
// Code illustration: not in QuickStart
_setOption: function (name, value) {
    if(name === 'maxItems') {
        this._resizeBoxForMaxItemsOf(value);
    }
    $.Widget.prototype._setOption.apply(this, argument
},
```

If **maxItems** is the name of the option being provided, the

_resizeBoxForMaxItemsOf method will be called. The last line is calling the base widget's **_setOption** method. This will set the value of the option and will aid in supporting a **disabled** state.

∠Note:

All widgets support the notion of being disabled whether they choose to implement it or not. The Boolean value is stored at **this.options.disabled** or **\$(selector).widget('option', 'disabled')** if you're asking from the outside. In return for honoring this option (whatever that would mean for the UI and behavior of your widget) the widget factory will default it to false and manage some CSS classes related to theming and accessibility.

The **_setOption** method is not called for the options passed in during the creation of the widget. When a widget has changed some of its options, inside **_create** for example, and wants **_setOption** to be called on each option, a convenient approach is to use the **_setOptions** method (with an 's') as in the following example.

JavaScript

// calls this._setOption on all options this._setOptions(this.options);

If options are passed to the widget method after it has been created, **_setOption** will be called on each passed option just before **_init** is called.

Functions as Options

Defining functions as options is a powerful way to decouple the widget from functionality better located elsewhere.

⊠Note:

The widgets in Mileage Stats use this approach for publishing and subscribing to global events by using their **publish** and **subscribe** options and getting data from the **dataManager** using their **sendRequest** option. To learn more about the pub/sub engine, see the Communication chapter and the Data chapter for more on the **dataManager**.

For example, rather than forcing the tagger widget to know how to invoke the public methods on the **infobox** widget, they can be kept free of any knowledge of each other by passing in the functions from the startup script since it already knows about both widgets. To set this up, the tagger widget defines **activated** and **deactivated** options.

JavaScript

Copy Code

```
// Contained in jquery.qs.tagger.js
$.widget('qs.tagger', {
    options: {
        activated: null,
        deactivated: null
    },
```

Just like normal options, these can either define defaults or not. The startup script will provide these options when it applies the tagger widget to the **span** elements.

JavaScript

Copy Code

```
// Contained in jquery.qs.tagger.js
$('span[data-tag]').tagger({
    activate: function (event, data) {
        // call displayTagLinks() on infobox here
    },
    deactivate: function () {
```

// call hideTagLinks() on infobox here } });

In the above code examples, the options are being set and read from inside the widget's implementation or passed in during creation or initialization. These options can also be read and written to from outside the widget through a public interface. Later in the chapter you'll see how function-based options are used as callbacks for events.

The Widget Method

Well-designed objects have public interfaces that are intentional, intuitive, and focused. Widgets go one step further and provide a single method that represents the entire public interface of the widget. The action the widget performs when you call this method depends on the number and type of arguments provided in the call. In addition to creating and initializing the widget as shown earlier, the widget method is also used to do the following:

- Invoke public methods
- Read and write public properties
- Read and write options

Public Methods

Public methods are defined on the widget prototype as you can see here in the **infobox** widget. The public methods are **hideTagLinks** and **displayTagLinks**.

```
JavaScript
```

Copy Code

```
// Contained in jquery.qs.infobox.js
$.widget('qs.infobox', {
    hideTagLinks: function() {
        ...
    },
    displayTagLinks: function(event, tagName) {
        ...
    }
```

Widgets must be created before their methods can be called. So the following calls to the **infobox** widget assume the widget method has already been called once to apply the widget to the **body** element. To call **hideTagLinks** from outside the widget, use a jQuery selector to match the element and pass the name of the method to the widget method as its only argument.

JavaScript Copy Code

```
// Code illustration: not in QuickStart
$('body').infobox('hideTagLinks');
```

When you have to pass arguments into the call, like **displayTagLinks**, simply add the arguments after the method name.

JavaScript

🖻 Сору Сс

```
// Code illustration: not in QuickStart
$('body').infobox('displayTagLinks', event, data.name)
```

The **option** method covered earlier in Defining Options (not to be confused with the **options** property) is an example of a public method. When one argument is

passed to it, the method will return the value of that option. When two arguments are passed, it will set the option specified in the first argument to the value of the second argument. When calling this method from outside the widget, pass the method name, **option**, as the first argument, the name of the option as the second, and the value as the third argument as shown here.

JavaScript

Copy Code

```
// Code illustration: not in QuickStart
var max = $('body').infobox('option', 'maxItems', 12);
```

As you can see above, public methods can also return values by putting the statement on the right hand side of a variable declaration. Returning a value from methods on **infobox** is reasonable because it is only attached to a single element. But be aware if you call a method on a wrapped set that contains more than one element, the method will only be called on and return from the first element.

In the examples so far, each time the widget method is invoked, it is being called on the instance returned by the jQuery function, **\$(selector)**, which requires accessing the DOM. The next section recommends a couple of alternatives.

Reusing an Instance

Each time the jQuery function uses a selector to invoke the widget method it must search the DOM. This has a negative impact on performance and is unnecessary because widget methods return a jQuery object, which includes the wrapped set of matched elements.

JavaScript

Copy (

```
// Code illustration: not in QuickStart
var ib = $('body').infobox(); // queries the DOM
ib.infobox('displayTagLinks'); // does not query the C
```

Rather than use a selector with the jQuery method each time you need to call a method on a widget, create a variable when the widget is initially attached to the elements. This will access the DOM, but it should be the only time you need to. In subsequent calls, like the second line in the snippet above, you can call the widget method on the variable you created and it won't access the DOM.

Using the Pseudo Selector

In a situation where neither the selector nor the instance is available, there is still a way to obtain all instances of a particular widget. As long as you know the name of the widget you can use a pseudo selector to get all instances that have been applied to elements.

JavaScript

The pseudo selector begins with a colon, followed by the widget's namespace and name separated by a hyphen. This selector has the potential to increase coupling between widgets so be aware of this if you intend to use it.

Private Members

Private methods and properties have private scope, which means you can only invoke these members from inside the widget. Using private members is a good idea because they improve the readability of the code.

Methods

Private methods are methods that start with an underscore. They are expected to be accessed directly using **this**. Private methods are common and recommended.

Private methods are only private by convention. This means if a widget isn't called according to the convention for calling public methods, described later, then its private methods can still be accessed. The convention is easy and consistent, and the underscore makes it easy to distinguish between the public and private interface.

Properties

Unlike methods, properties on the widget prototype are not made private by prepending an underscore – they are private by default. Only methods are made private with underscores. Properties don't need underscores because they cannot be accessed through the widget method.

JavaScript

```
// Code illustration: not in QuickStart
$.widget('qs.infobox', {
    dataUrl: '', // should only be accessed using th
    _maxItems: 10 // unnecessary, properties are alre
});
```

Because each element contains its own instance of the widget, the **dataUrl** property can be different for each element.

Clearly **dataUrl** is best exposed as an option, but if this was not a configurable option, you would likely want to define it so only one copy of the value was available to all instances of the widget. Let's call these static members.

Static Members

To define a variable that's available to all instances of the widget but nowhere else, place them inside the self-executing function wrapper, but above the call to the widget factory as shown in the tagger widget.

```
JavaScript
// Contained in jquery.qs.tagger.js
(function ($) {
    var timer,
        hideAfter = 1000; // ms
    $.widget('qs.tagger', {
        ...
```

Because the **timer** variable is defined outside of the widget prototype, only a single timer will be created and shared across all instances of the tagger widget. Functions that don't rely on the instance of the widget can also be defined here.

If you need access to static members from outside the widget, they can be added to the widget after the widget's definition. A fictitious change can be made to the infobox widget to illustrate this. Inside the **displayTagLinks** method in the infobox widget, a function variable called **displayResult** is defined.

```
JavaScript
```

Copy Code

Copy Code

```
// Contained in jquery.qs.infobox.js
displayResult = function () {
    // don't need to pass in elem, html, top, or left
    // since they are in scope
    elem
    .html(html);
    .css({top: top, left: left});
    .show();
};
```

It is defined in **displayTagLinks** because it's the only method that uses it. If the infobox widget needs to make AJAX calls from other methods, the **displayResult** function might need to be moved so it is available to all methods that need it. Defining it outside the scope of the widget is a way to make this happen.

JavaScript

Copy Code

```
// Code illustration: not in QuickStart
$.widget('qs.infobox', {
    ...
});
$.extend($.qs.infobox, {
    displayResult: function(elem, html, top, left) {
        elem
            .html(html);
            .css({top: top, left: left})
            .show();
    }
});
```

The **\$.extend** method is used to merge the object passed as the second argument into the object passed as the first argument. Therefore, the **displayResult** method is merged into the prototype of the widget, **\$.qs.infobox**. With **displayResult** defined here, the infobox widget can use it from anywhere as shown here.

JavaScript

```
// Code illustration: not in QuickStart
// assume elem, html, top, and left variables were alr
$.qs.infobox.displayResult(elem, html, top, left);
```

Events

Events are an effective way to communicate between widgets without forcing them to be tightly coupled. jQuery supports and extends the DOM event model and provides the ability to raise and handle custom events that are not defined in the DOM.

Binding Handlers

Event handlers bind to widget events the same way they bind to other events.

JavaScript

Notice how the name of the event being bound to has had the name of the widget prepended. This is the default behavior for event names. If you would prefer a different name so your code is more readable, this behavior can be changed.

Event Naming

The **widgetEventPrefix** property defines what will be prepended to the names of the events the widget raises. By default the value is the name of the widget and is set by the widget factory. If you want to use something other than the widget name, simply define this property and provide an alternative value.

JavaScript

Copy Code

```
// Contained in jquery.qs.tagger.js
$.widget('qs.tagger', {
    widgetEventPrefix: 'tag',
    options: {
        activated: null,
        deactivated: null
    },
```

When **widgetEventPrefix** has a value, it will be used instead of the widget name.

Raising the Event

The widget naming convention described above is only applicable to the event handler. Inside the widget, the original event name is used to raise the event. The following code sample shows one way the tagger widget might raise the **activated** event when the mouse enters the element.

JavaScript

```
// Code illustration: not in QuickStart
_create: function () {
   var that = this,
      tag = that.infoboxElement.text();
   that.infoboxElement
      .bind('mouseenter', function (event) {
        that._trigger('activated', event, {name: t
      });
},
```

When **trigger** is called, the event will be raised and any bindings will be invoked. The problem with binding directly from inside a widget is that it creates more coupling than is needed for event handlers. If the widget is following well-accepted widget design practices, the widget will have callbacks defined in its options.

Relationship to Options

When options are defined as functions and their names correspond to an event name without the prefix, they are referred to as callbacks. The **_trigger** method on the base widget will automatically invoke the callback whose name matches the event being raised.

JavaScript

```
// Contained in jquery.qs.tagger.js
widgetEventPrefix: 'tag',
options: {
    activated: null,
    deactivated: null
},
_create: function () {
    var that = this,
        name = this.name(),
        tag = this.element.text();
    this.element
        .bind('mouseenter.' + name, function (event) {
            that._trigger('activated', event, {name: t
            });
},
```

The JavaScript that creates the tagger widget can now define the handler for the **activated** and **deactivated** events when it creates the widgets.

JavaScript

```
$('span[data-tag]').tagger({
    activated: function (event, data) {
        infobox.infobox('displayTagLinks', event, data
    },
    deactivated: function () {
```

```
infobox.infobox('hideTagLinks');
});
```

This allows the two widgets to interact without explicitly knowing about each other. Using this approach causes the script that invokes the widgets to act as a connective tissue that describes a lot about the solution in a succinct readable format.

Inheritance

Sometimes when building a widget, another widget already has a lot of what the new widget needs. The widget factory's inheritance support is designed for this case. For illustration purposes, consider the following widget.

JavaScript

Copy Code

```
// Code illustration: not in QuickStart
(function ($) {
    $.widget('a.container', {
        ...
        resize: function() {
            // resize width and height
        },
        ...
    });
}(jQuery));
```

If this widget was built elsewhere and you want to change its resizing behavior to animate, a reasonable approach would be to inherit from **a.container** and override its resize method. Inheritance is accomplished by passing three arguments into the widget factory. The first argument is the namespace and name of the widget, the second is the base widget being inherited from, and the third argument is the object prototype of the derived widget.

JavaScript

```
// Code illustration: not in QuickStart
(function ($) {
    $.widget('an.animatedContainer', $.a.container.prc
    ...
    resize: function() {
        // override with animations
        },
      });
}(jQuery));
```

The only difference between the signature above and the signature usually used for defining widgets is addition of the second parameter.

Inheritance is a useful tool when you are using a widget that almost does what you want it to do. In version 1.9 of jQuery UI, widgets will be able to inherit from themselves, which makes it even easier to extend the functionality of widgets.

Summary

Using the jQuery UI widget factory is a great way to add modularity to clientside web applications. Their lifetimes can be managed with **_create**, **_init**, and **destroy**. Options should have intelligent defaults but can be overridden at any time by passing them to the widget method or calling the **option** method directly. Functions are a powerful way to decouple functionality and using them for callbacks makes raising and handling events straight-forward. Widgets can have public methods and properties and uses a prepended underscore for private methods. Define functions and variables outside of the widget prototype but inside the self-executing function wrapper when it's appropriate for all instances of the widget to use a single function or variable. Widgets can also be inherited when base functionality can be shared across different widgets. See the next chapter for more places to learn about jQuery plugins and jQuery UI.

Further Reading

Appendix B: Widget QuickStart

Widget Factory documentation on the jQuery UI wiki: <u>http://wiki.jqueryui.com/w/page/12138135/Widget-factory</u>

jQuery Documentation for Plugins/Authoring: http://docs.jquery.com/Plugins/Authoring

jQuery UI Developer Guidelines: http://jqueryui.com/docs/Developer_Guide

jQuery UI source code: https://github.com/jquery/jquery-ui

Resources

Content Delivery Network (CDN) Addresses: http://www.asp.net/ajaxlibrary/cdn.ashx

Community

To report documentation errors or provide feedback on this documentation, please send email to pagdoc@microsoft.com

Copyright © 2011 by Microsoft Corporation. All rights reserved.

Silk 1.0 - June 2011
Application Notifications

Introduction

All web applications that users consider responsive have one thing in common: they provide appropriate and timely feedback to the user. This feedback can come in many forms, including a save or success message following a completed task, subtle animations in response to a user interface (UI) gesture, a progress message for long-running tasks or input error messages displayed before a page is submitted.

How the application surfaces the notifications to the user is almost as important as the information itself. Intrusive message boxes, modal dialogs, and overlays (floating messages) that require the user to dismiss messages, can interrupt the user's workflow, get in the way, and degrade the overall user experience.

In addition to providing feedback during normal application use, the website must also provide quality feedback when a non-recoverable error occurs. Quality feedback means providing understandable information about what has occurred, along with clear instructions on how to proceed.

What You Will Learn in This Chapter

In this chapter you will discover:

- How to provide unobtrusive user notification messages.
- How to handle multiple simultaneous notification messages raised by the application.
- The benefits of encapsulating the display and management of user notification in a single JavaScript object.
- How to display a helpful global error page.
- How to set up a global error handler for Ajax requests.
- Alternatives to modal dialogs for prompting users.
- How to enable application notifications on the desktop with the Pinned Sites API.

The technologies discussed in this chapter are jQuery UI Widgets and Pinned Sites in Windows® Internet Explorer® 9.

For a comprehensive look at input validation error messages, see Chapter 10, "Data, Caching, and Validation."

Notifying the User

Providing a high-quality application notification experience requires careful planning with emphasis on where notifications will be displayed, what events initiate a message, how potential multiple simultaneous messages will be handled, and how to decouple the message originator from the object tasked with displaying the message.

During the design phase of the Mileage Stats application, the Project Silk team discussed *where* and *how* notification messages would be displayed. We spent time prototyping several different notification designs.

Where notification messages are displayed is an essential part of the overall application user experience (UX) and user interface design. Our initial design called for messages and progress bars to be displayed within the boundaries of each jQuery UI widget. After building several prototypes and performing usability testing, the team determined this design was unnecessary because the UI loads very fast, alleviating the need for a loading progress bar in this application. The team decided that displaying user messages in a single location made for a much better experience than having messages displayed within individual widgets.

Throughout application development, the team tailored application events that triggered user messages based on usability testing. Initially, the team displayed messages each time an Ajax request was invoked. This caused the UI to be too busy, so we associated a time delay with the message so that it would only display if the request took longer than the time delay. This too, got messy, requiring a good deal of code with little or no value added to the application. In the end, the "less is more" principle triumphed, resulting in a good balance of informative messages.

Interactive and engaging applications such as Mileage Stats can execute multiple, asynchronous operations, such as the Dashboard page that loads data for several jQuery UI widgets in addition to the chart widget. Each of these operations loads data for a region of the UI. Any of these operations is a potential point of failure requiring an error message. It's important that the application notification implementation be able to manage multiple simultaneous or nearly simultaneous messages.

From an architectural design perspective, it's critical that message initiators not

be responsible for determining how to coordinate the display of messages in the UI. Decoupling the message initiator from the rendering object allows both of them to evolve independently and to be tested in isolation.

The above section provides a glimpse into how the team worked together to maintain the delicate balance of UX, UI, and engineering concerns. It's this type of designer-developer collaboration that enabled the team to deliver a successful notification feature.

Where Notification Messages are Displayed

Mileage Stats is composed of widgets. The decision to create and use a widget for displaying notification messages is a natural architectural design fit for this application. Widgets have flexible and powerful UI capabilities, provide for encapsulation of behavior, and can have external dependencies like publish and subscribe (pub/sub) injected into their options object during creation.

Mileage Stats uses a single widget called **status** for displaying messages to the user. The **status** widget subscribes to the Mileage Stats **status** pub/sub message. It also handles the placement and rendering of messages as well as the coordination of multiple simultaneous messages.

Location of the status widget



The **status** widget is rendered within the **header** widget UI, as pictured above. This top, semi-central location was chosen because it's easier for the user to notice the message in this location, as opposed to a message area along the bottom of the browser window. The balance of easily noticed, easy-to-read, yet unobtrusive user notifications, took time, patience, and usability testing, but the multiple design iterations were worth the extra investment of time.

How Notification Messages are Initiated

Mileage Stats notification messages are initiated by widgets and communicated to the **status** widget using the pub/sub JavaScript object. Like other pub/sub messages, the status message has an associated payload object that is passed with the message.

Notification messages passed using Pub/Sub



The code snippet below is from the **vehicleDetails** widget. The **_publishStatus** method is responsible for making the pub/sub call. It's called internally by other widget methods to initiate the display of a message. The **status** argument is the message payload and is forwarded in the **publish** call. The **publish** method was passed in the widget options object when the widget was created and points to the **pubsub** JavaScript object. The jQuery **isFunction** method verifies that **publish** is a valid JavaScript function object before it's called.

```
JavaScript Copy Co
// Contained in mstats.vehicle-details.js
_publishStatus: function (status) {
    this.options.publish(mstats.events.status, status)
},
```

As stated earlier, Mileage Stats does not bother the user with data request messages. However, when initiating an Ajax operation such as a save or delete, it's important to keep the user informed by updating the UI as the request proceeds and concludes.

The following functions show how easy it is to initiate the display of a user message:

• The **_showDeletingMessage** function is called after the user confirms his intent to delete the vehicle. This message is intended to inform the user that the vehicle deletion has been submitted to the server.

- The **_showDeletedMessage** function is called after a successful deletion of the vehicle, informing the user that the deletion was successful.
- The **_showDeleteErrorMessage** function is called if an error occurred while deleting the vehicle.

```
JavaScript
```

```
// contained in mstats.vehicle-details.js
_showDeletingMessage: function () {
  this. publishStatus({
    type: 'saving',
    message: 'Deleting the selected vehicle ...',
    duration: 5000
  });
},
_showDeletedMessage: function () {
  this._publishStatus({
    type: 'saved',
    message: 'Vehicle deleted.',
    duration: 5000
  });
},
_showDeleteErrorMessage: function () {
  this._publishStatus({
    type: 'saveError',
    message: 'An error occurred deleting the selected
    duration: 10000
  });
}
```

Each function creates an object literal containing a **type**, **message**, and **duration** property. The **type** property is used by the **status** widget to prioritize multiple or overlapping display message requests. The **message** is the text of the message to display and the **duration** is how long the message should display.

For detailed information on the inner working of the Mileage Stats pub/sub implementation, see Chapter 8, "Communication."

How Individual or Multiple Notification Messages are Displayed

In the following **_create** method, the **status** widget subscribes to the **status** event. When this event is raised, the **_statusSubscription** method is invoked.

The **_statusSubscription** method is responsible for displaying and hiding messages as well as managing multiple simultaneous messages. If a message is being displayed and another message with a higher priority arrives, the higher priority message will be shown.

JavaScript

```
// contained in mstats.status.js
_create: function () {
  // handle global status events
  this.options.subscribe(mstats.events.status, this._s
},
. . .
statusSubscription: function (status) {
  var that = this;
  status.priority = that._getPriority(status);
  // cancel displaying the current message if its pric
  // the new message. (the lower the int, the higher r
  if (that.currentStatus && (status.priority < that.cu
    clearTimeout(that.currentStatus.timer);
  }
  that.currentStatus = status;
  that.element.text(status.message).show();
  // set the message for the duration
  that.currentStatus.timer = setTimeout(function () {
    that.element.fadeOut();
    that.currentStatus = null;
```

```
}, status.duration || that.options.duration);
},
```

User Session Timeout Notification

Mileage Stats uses forms authentication, with a session timeout threshold of 20 minutes. If the session has timed out, the request (Ajax or non-Ajax) is redirected to the page specified by the forms authentication **loginUrl** in the web.config file.

In traditional websites that perform page reloads between pages, it's common to redirect the user to a sign-in page when their session times out. Applications like Mileage Stats that make heavy use of Ajax calls to retrieve data, perform few full-page reloads. Consequently, if a session timeout occurs, it's usually during an Ajax request. Let's examine what happens when an Ajax request is redirected because of an authentication session timeout:

- Ajax JavaScript Object Notation (JSON) data request initiated.
- Forms authentication runtime detects an expired session and redirects the request to the sign-in page.
- A parsing error occurs because the Ajax handler is expecting JSON data and not HTML. The HTML is the content of the sign-in page to which the request was redirected.
- An Ajax error callback is invoked.
- A global Ajax error callback is invoked.

Errors that can occur anywhere in the application can often be handled in a centralized location so that individual objects don't need to repeat the same error handling code. Mileage Stats implements the global **ajaxError** method handler shown below to catch errors occurring during an Ajax request. The primary purpose of this method in Mileage Stats is to identify whether the initiating Ajax request caused a session timeout error and, if so, redirect the user to the sign-in page.

When looking at the code below, "jqXHR.status === 200" appears out of place or incorrect. Remember, this method is only executed when an Ajax error occurs. If the session times out and the request is redirected to the sign-in page, the response status code will be 200 because the redirect succeeded. In addition to checking for the response status code, this method also verifies that the returned HTML contains the sign-in page's title. If both conditions are met, the browser is redirected to the sign-in page.

JavaScript

⊠Note:

If the originating Ajax calling code also implements an error handler, the originating Ajax caller's error handler will be called first, then the above global Ajax error handler will be called.

Website Error Notification

ASP.NET provides you with the ability to specify a default error page for their website that the ASP.NET runtime will redirect to when an unhandled exception occurs. This error page is configured in the web.config file **customErrors** section.

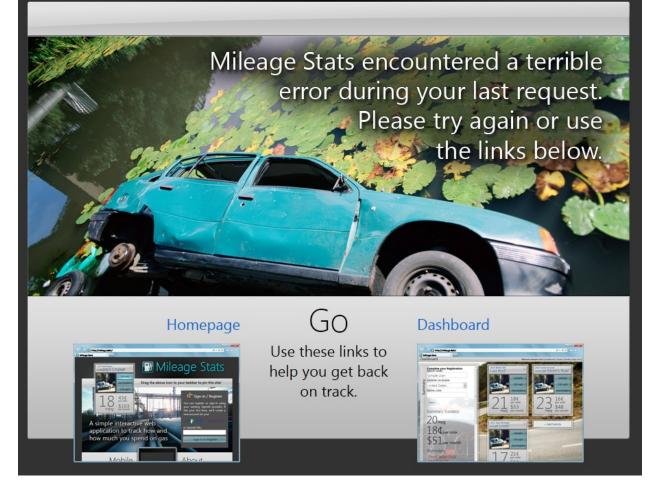
C#

```
// Contained in web.config
<customErrors defaultRedirect="GenericError.htm" mode=</pre>
```

The error page should look and feel like it is part of the website, contain a brief explanation of why the user has been redirected to this page, and provide links to continue using the site.

Mileage Stats GenericError.htm page

Mileage Stats



Prompting Users

During the design phase of Project Silk, the team had a goal of not prompting users with modal dialogs. Website UX designers are getting away from modal dialogs that ask the user questions like, "Are you sure?" Instead, designers prefer an undo system, allowing users to undo the previous task. The undo feature also enhances the application by extending undo capabilities to tasks that did not require a confirmation dialog.

Since Mileage Stats is only a sample application, it has limited functionality. A production application could implement this undo feature. The team preferred the undo feature, but other features took priority.

The code below uses the JavaScript **confirm** function to validate the user's request to fulfill a maintenance reminder.

JavaScript

```
// contained in mstats.reminders.js
fulfillReminder: function (fulfillmentUrl) {
    var shouldfulfill = confirm('Are you sure you want
    if (shouldfulfill) {
        this._fulfillReminder(fulfillmentUrl);
    }
},
```

⊠Note:

The jQuery UI dialog provides an alternative to using the JavaScript **confirm** dialog. If you are leveraging jQuery UI plug-ins, you should consider using the jQuery UI dialog for consistency in your UI.

Desktop Notifications

Given that modern web applications can provide excellent user experiences that rival desktop applications, the team wanted to take the next logical step and integrate the Mileage Stats application with the user's desktop to provide appropriate dynamic user notifications. This integration was made possible by the Internet Explorer 9 Pinned Site API.

Websites that implement the Pinned Site API can feel more like a native Windows application. They can take advantage of the Microsoft® Windows® 7 taskbar capabilities and, when launched, the browser window is customized specifically for the site. The full Pinned Sites experience requires Internet Explorer 9 running on Windows 7. Windows Vista® provides fewer Pinned Sites features—site pinning, customized reduced chrome, and the disabling of browser add-ons.

Mileage Stats uses Pinned Sites to provide Windows 7 taskbar notifications that indicate whether the user has one or more overdue maintenance reminders. In addition, a dynamic jump list provides a direct link to each overdue maintenance reminder.



Mileage Stats taskbar integration

∠Note:

Jump list items will be available whether the site is opened in a browser or not. However, the notification icons are only displayed when the site is

opened in the browser.

The two images below contrast Mileage Stats running in a normal browser window and a customized Pinned Sites browser window. The lower image shows the clean, pared down browser window with potentially distracting browser features removed from view, allowing the user to focus on the application features. Applications run in the customized browser window when they are launched from a taskbar or Start Menu Pinned Sites icon.

Mileage Stats without using Pinned Sites



Mileage Stats using Pinned Sites



In addition to a cleaner browser window, Pinned Sites also allows the developer to customize the color of the browser back and forward buttons and displays the website favicon to the left of the back button. This favicon is also a link to the website home page.

Implementing Pinned Sites

Microsoft provides documentation for implementing Pinned Sites in their web applications on MSDN®. The title of this topic is, "Pinned Sites Developer Documentation," and is located at <u>http://msdn.microsoft.com/en-us/library/gg491731(v=VS.85).aspx</u>.

Pinned Sites in Mileage Stats

The following sections will not attempt to duplicate the MSDN documentation just mentioned nor cover every line of code pertaining to Pinned Sites. Instead, the Mileage Stats implementation will be explained, enabling you to understand pieces, requirements, capabilities, and value of the Pinned Sites API.

The Pinned Sites implementation in Mileage Stats includes feature detection, site pinning, dynamic jump list updating, and display of notification icons. These features are encapsulated in the **mstats.pinnedSite** JavaScript object that is contained in the mstats.pinnedsite.js file. The **pinnedStite** object is initialized differently depending on whether or not the user is signed in. This initialization will be described below.

Feature Detection

Pinned Sites feature detection is provided by the Internet Explorer 9 **msIsSiteMode** function. Verifying that the page is opened as a pinned site before executing Pinned Site API methods prevents unnecessary JavaScript errors.

The **msIsSiteMode** function returns **true** if the current page is launched as a pinned site; **false** if it is not. The below **isPinned** function wraps the **msIsSiteMode** call and returns **false** if the page is not launched as a pinned site, or the browser is not Internet Explorer 9.

JavaScript

Copy Code

```
// Contained in mstats.pinnedsite.js
isPinned: function () {
    try {
        return window.external.msIsSiteMode();
    }
    catch (e) {
        return false;
    }
}
```

Enabling Website Pinning

Unauthenticated users visiting the site are directed to the landing page, which is shown below. This page allows users to sign in, pin the site, and view the Mileage Stats video (not pictured). The Pinned Sites icon will glow when it is draggable, allowing the user to pin the site to the taskbar or Start Menu. The callout text displays for 5 seconds when the page loads. It will also show and hide the text as the user moves her mouse over or away from the Pinned Sites icon.

☑ Note:

Developers are not required to implement a draggable site icon as Mileage Stats does to enable site pinning. Providing a draggable icon allows the website more control over the pinning experience.

Without a draggable icon, sites can still be pinned by dragging the tab or the favicon to the taskbar.

Landing page



The Pinned Sites JavaScript object is initialized when the above page loads with the below JavaScript function.

CSHTML

Copy Code

```
// Contained in Index.cshtml
<script>
    $(function () {
        mstats.pinnedSite.intializePinndedSiteImage();
    });
</script>
```

If the browser is Internet Explorer 9 and the website is not currently pinned, the **intializePinndedSiteImage** method will attach appropriate event handlers for hiding and showing the callout text. It also adds the **active** CSS class to the Pinned Sites icon so that the icon appears to glow.

JavaScript

```
// Contained in mstats.pinnedsite.js
intializePinndedSiteImage: function () {
  try {
    // Do not enable site pinning for non-Internet Exp
    // Do not show the callout if the site is already
    if (!(document.documentMode === undefined || mstat
      $('#pinnedSiteImage')
            .bind('mousedown mouseout', mstats.pinnedS
            .bind('mouseover', mstats.pinnedSite.showC
            .addClass('active');
      $('#pinnedSiteCallout').show();
      setTimeout(mstats.pinnedSite.hideCallout, 5000);
    }
  }
  catch (e) {
    // Fail silently. Pinned Site API not supported.
  }
},
```

The HTML snippet below shows the required **msPinSite** class applied to the Pinned Sites icon. This class is used by Internet Explorer 9 to enable the user to drag the Pinned Sites icon to the taskbar or Start Menu and pin the site.

CSHTML

Copy Code

```
// Contained in Index.cshtml
<img id="pinnedSiteImage" class="msPinSite" ... />
```

To call the user's attention to the draggable Pinned Sites icon, the **active** CSS class below adds an attractive outer glow to it.

CSS

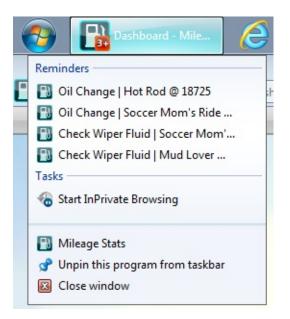
```
// Contained in static.css
#pinnedSiteImage.active
{
    cursor: pointer;
    box-shadow: 0px 0px 15px #6Dffff, inset 0px 0px 10
    border-radius: 12px;
}
```

The user can pin a website by dragging the Pinned Sites icon, browser tab, or favicon to the taskbar or Start Menu. Internet Explorer 9 integrates with the Windows shell to accomplish the pinning.

Dynamic Jump List Updating and Notification Icons

Mileage Stats uses the jump list and notification icons to notify users of overdue maintenance reminders. When users click on the jump list entry, they will be taken to that reminder. The notification overlay icon displays 1, 2, 3, or 3+ to provide a taskbar indication of outstanding reminders.

Jump list and notification icon



On the initial page load after the user authenticates, the client-side widgets and JavaScript objects are invoked by code in the mileagestats.js file. The **pinnedSite** object is initialized by passing it a delegate to the data manager's **sendRequest** method.

JavaScript

```
// Contained in mileagestats.js
mstats.pinnedSite.intializeData(mstats.dataManager.ser
```

The **initializeData** function saves the **sendRequestFunc** in the **sendRequest** property for future calls to the data manager by the **requeryJumpList** function.

JavaScript

Copy Code

```
// Contained in mstats.pinnedsite.js
intializeData: function (sendRequestFunc) {
   sendRequest = sendRequestFunc;
   mstats.pinnedSite.requeryJumpList();
},
```

The below **requeryJumpList** function is called when the **pinnedSite** object is initialized and called by the **layoutManager** widget when a reminder is fulfilled. It's the layout manager's call that initializes the dynamic updating of

the jump list and notification icon.

∠Note:

Only the essential lines of code that demonstrate the loading of the jump list and updating of the notification icon are listed below.

All of the below **msSite** functions are provided by Internet Explorer 9. After using feature detection to determine if the site is pinned, the jump list and overlay icon are cleared, and a new jump list is created.

Not shown below is the Ajax call to the data manager to get the array of overdue reminders. If that Ajax request is successful and the **data.Reminders** array has data, a URL will be constructed for each data item and added to the jump list. Next, the appropriate overlay icon is set. Finally, **msSiteModeShowJumpList** is called to update the jump list.

JavaScript

```
// Contained in mstats.pinnedsite.js
requeryJumpList: function () {
  try {
    if (mstats.pinnedSite.isPinned()) {
      var g_ext = window.external,
      . . .
      g_ext.msSiteModeClearJumpList();
      g_ext.msSiteModeCreateJumpList("Reminders");
      g_ext.msSiteModeClearIconOverlay();
      if (data.Reminders) {
        for (i = 0; i < data.Reminders.length; i += 1)</pre>
          reminderUrl = mstats.getRelativeEndpointUrl(
            data.Reminders[i].Reminder.ReminderId.toSt
          g_ext.msSiteModeAddJumpListItem(data.Reminde
            "./favicon.ico", "self");
        }
        if (data.Reminders.length > 0) {
          iconOverlayUrl = '/content/overlay-' + data.
          iconOverlayMessage = 'You have ' + data.Remi
```

```
' maintenance tasks that are ready to be a
if (data.Reminders.length > 3) {
    iconOverlayUrl = '/content/overlay-3plus.i
    }
    g_ext.msSiteModeSetIconOverlay(iconOverlayUr
    }
}
g_ext.msSiteModeShowJumpList();
...
```

The above code demonstrates that with a small investment, you can deliver dynamic desktop notifications in your websites.

Requirement for Jump List Items to Appear

The Windows 7 taskbar jump list items can be disabled by your users, preventing them from displaying even though the website has been pinned to the taskbar.

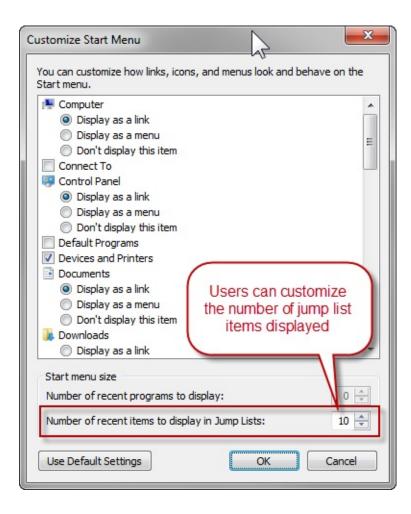
If your website implements the jump list feature, you should provide this information to your users and advise them that the "Store and display recently opened items in the Start Menu and the taskbar" property setting needs to be checked for the jump list items to show up.

Taskbar and Start Menu properties

L Taskbar and Start Menu Properties	
Taskbar Start Menu Toolbars	
To customize how links, icons, and menus look and behave in the Start menu, dick Customize.	Dashboard - Mile
Power button action: Log off	Reminders
Privacy	📔 🛅 Oil Change Hot Rod @ 18725 🕴
Store and display recently opened programs in the Start menu	🔡 Oil Change Soccer Mom's Ride
Store and display recently opened items in the Start menu and the taskbar	Check Wiper Fluid Soccer Mom' Check Wiper Fluid Mud Lover
	Tasks
Required!	🔝 Mileage Stats
	📌 Unpin this program from taskbar
	Close window
How do I change the way the Start menu looks?	
OK Cancel Apply	

In addition be being able to disable jump list items, users can customize the number of jump list items displayed on their computers. The default value is 10 and can be changed in the Customize Start Menu dialog below. This dialog is opened by clicking the **Customize** button in the Taskbar and Start Menu Properties dialog shown above.

Customizing Start Menu Properties



Summary

Providing timely feedback that is uniformly displayed, context sensitive, and understandable to your users without breaking their workflow takes planning by designers and developers alike. Your users will appreciate this extra effort, which results in a polished user experience. By encapsulating the display and management of user notifications in a single JavaScript object, your application will be easier to code, maintain over time, and test. You have also learned about integrating your website with the Windows 7 desktop to provide users with dynamic notifications and jump list items, as well as browsing your site using a customized browser window.

Further Reading

For a comprehensive look at input validation error messages, see Chapter 10, "Data, Caching, and Validation."

For detailed information on the inner working of the Mileage Stats pub/sub implementation, see Chapter 8, "Communication."

For more information about the **isFunction** method, see jQuery.isFunction(): <u>http://api.jquery.com/jQuery.isFunction/</u>

For more information about Pinned Sites, see the Pinned Sites developer documentation:

http://msdn.microsoft.com/en-us/library/gg491731(v=VS.85).aspx

| Community

To report documentation errors or provide feedback on this documentation, please send email to pagdoc@microsoft.com

Copyright $\ensuremath{\mathbb{C}}$ 2011 by Microsoft Corporation. All rights reserved.

Silk 1.0 - June 2011 Modularity

Introduction

Applying a modular design makes solutions more maintainable. By partitioning complexity into modules named after concepts in the solution domain, the client-side code will be easier to read, understand, maintain, test, and troubleshoot. When the source code is difficult to follow, these tasks can be unreasonably time consuming. While not all websites contain enough complexity to warrant a deliberate modular design, web applications with immersive experiences certainly do.

Immersive experiences use modern user interface (UI) design approaches to keep the user *in context* while inside the defined borders. *In context* means the user is never confused about where they are in the solution. Breadcrumbs are one way to help the user know where they are, but they don't match the visual intuitiveness of an immersive UI. To create these UIs, you must keep the interface fluid and avoid jarring or flickering. Flickering is occurs when a whole page has to load in a browser for the first time. The user expects either instant responsiveness or some indication of progress when using a responsive web application. This requirement alone places a number of non-trivial responsibilities on the client-side. Some of these responsibilities have to do with Ajax data retrieval and caching, updating UI content, state management, and animating layout transitions. For these applications to be maintainable, they must be well-composed of objects that have clear boundaries and responsibilities.

Fortunately, there are good libraries available today that can help make your application more modular. This chapter uses Mileage Stats to illustrate how to define the boundaries, responsibilities, and interactions in an intentionally modular design that supports portions of an immersive UI. Mileage Stats uses jQuery as its library for DOM manipulation and jQuery UI as its library for helping achieve modularity. As a result, the rest of the chapter will refer to modules either as a JavaScript object, a jQuery plugin, or a jQuery UI widget.

In this chapter you will learn:

- Strategies for defining the boundaries of widgets
- How to define the interface based on its interactions
- An understanding of the different types of widgets
- When to use a JavaScript object, a jQuery plugin, or a jQuery UI widget

Defining Widget Boundaries

Widgets typically have a one-to-one mapping with a single or repeating element on the page. These elements define the boundary of the widget. The widget essentially attaches behavior and state to matching elements. For example, the following HTML shows a condensed structure of the Mileage Stats dashboard layout. The elements not relevant to this discussion are removed.

HTML

Copy Code

```
<div id="dashboard-page">
    <div id="vehicles">
        <div id="vehicle-list-content">
            <div class="wrapper">
                <div class="vehicle"/>
                <div class="vehicle"/>
                <div><a>Add Vehicle</a></div>
            </div>
        </div>
    </div>
    <div id="main-chart"/>
    <div id="fixed">
        <div id="summary">
            <div id="registration"/>
            <div id="statistics"/>
            <div id="reminders"/>
        </div>
        <div id="info"/>
    </div>
</div>
```

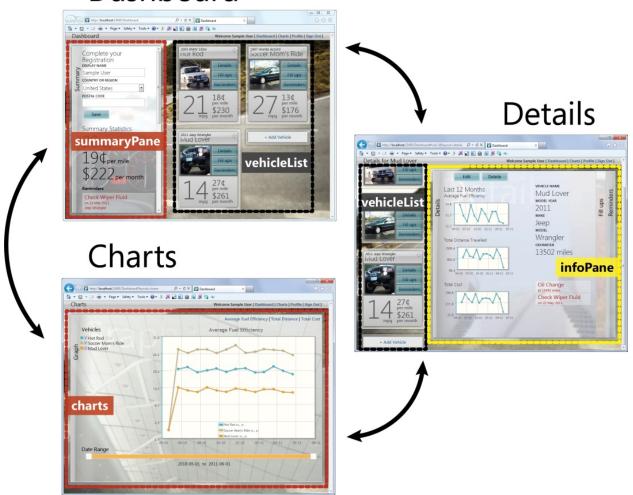
This shows the structural relationship these regions have with each other. These are the elements the associated widgets are attached to. But it's unlikely you would start with this. The process of defining the boundaries of widgets can depend on multiple factors such as application layouts, animations, and data refreshes.

Influences from the Layout

The UI layout design can provide good hints about the boundaries of potential widgets. A top priority in Mileage Stats was for the user to not see any page refreshes while the user is reading vehicle statistics at various levels for different vehicles. This means the user is navigating between the dashboard, details , and charts layouts. In other words, this is the boundary of the immersive experience.

The three main layouts in the immersive experience

Dashboard

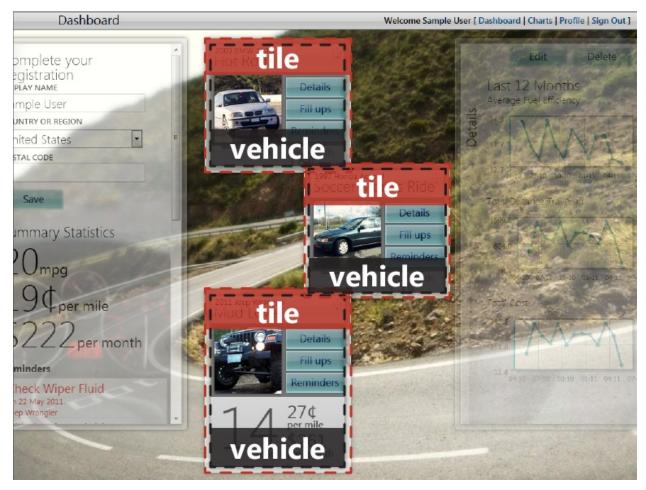


These layouts are made up of four regions: summaryPane, vehicleList, infoPane, and charts. To keep the user in context, the vehicleList is used in both the Dashboard and Details layouts, and transitions between these layouts as they are animated. Because the vehicleList is used in both Dashboard and Details layouts, the user never loses sight of the selected vehicle. The summaryPane and the vehicleList regions enter and exit from the left side of the screen and the infoPane and charts enter and exit from the right.

Because navigation between these layouts isn't causing full page refreshes, these regions should know how to respond to show, hide, and animate messages. Their need for these behaviors is a good indication they should be widgets. Also, something must be responsible for telling each of these widgets to show, hide, or animate. This is the role of the layoutManager, a widget that doesn't have any UI, but controls the operation of other widgets. The layoutManager widget will be covered later in the chapter.

Influences from Animations

The vehicleList widget contains two kinds of boxes: vehicle boxes and a box to hold the *add vehicle* link. When transitioning to and from Dashboard and Details layouts, all of these boxes animate between one and two columns in a two-step process. At the same time, each of the vehicle boxes that were not selected shrink to a compact size. The logic for these animations is included in two widgets: the tile widget, and the vehicle widget. The tile widget is responsible for animating the position of all boxes horizontally and vertically since both of these boxes need that behavior. The vehicle widget is responsible for expanding and collapsing the boxes since only vehicle widgets have that behavior.



Animating from Dashboard to Details

Modularizing these animations into widgets provides clear boundaries and responsibilities. Another, perhaps more appropriate, approach might be to place

all of the logic for how to animate items in a container into a single, applicationagnostic widget. Then the vehicleList widget could apply the animation behavior widget to itself.

Influences from Refreshing Data

When all data updates are happening through Ajax calls, various parts of the UI will have to know how and when to request updates from the server and apply any necessary changes to the UI. The statistics and imminent reminders regions of the summaryPane are good examples of this.

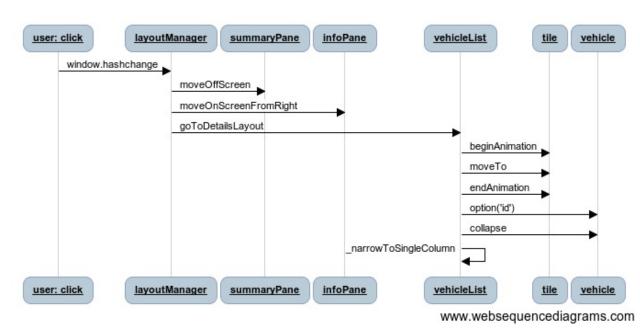
Statistics widget is responsible for refreshing content



The statistics and imminent reminders regions know how to independently request their data and update their content accordingly when changes in vehicle, fill up, or reminder data are detected. Many of the other widgets in Mileage Stats are also responsible for retrieving and applying updated content. However, the code that actually makes the requests, and adds some caching functionality, is implemented in a separate module. To learn more about the data abstraction in Mileage Stats, see Chapter 10, "Client Data Management and Caching." This section illustrated how a widget's boundaries can be influenced by UI layout, behaviors for animation and interaction, and content updates for the UI. While the boundary defines the scope of the widget, the interface defines the widget's responsibility and how it interacts with other modules in the solution. Exploring these interactions can provide clues into how to define the widget's interface.

Defining the Interactions

The interface of each widget is designed so the widget can work with other parts of the UI without any unnecessary coupling. The interface of a widget is made up of three things: the options it accepts, the public methods it exposes, and the events it raises. As an example, the following sequence diagram shows the public methods and options called when the **Details** button on a vehicle tile is selected while on the dashboard layout.



Sequence when transitioning from Dashboard to Details

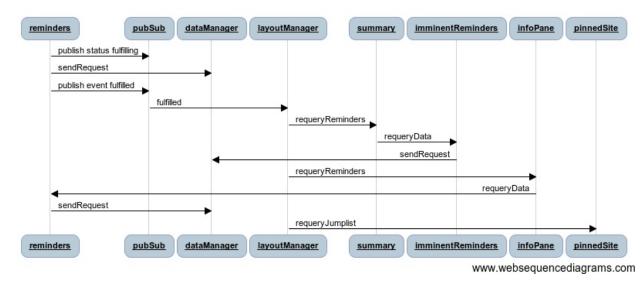
The diagram illustrates how the layoutManager orchestrates the transition by calling methods on the widgets. Notice how the diagram doesn't start with the vehicle widget even though it contains the button that was clicked. This is because rather than the vehicle widget having the responsibility of capturing the click, it's responsible for modifying the button to trigger the **window.hashchange** event.

The tile widget is responsible for animating tiles. Because of the specifics of the tile animation, it exposes a **beginAnimation** method to prepare for the animation that happens when the **moveTo** method is called. Not surprisingly, **endAnimation** completes the process. To learn more about how the tile widget performs its animation, see Chapter 9, "Navigation ."

The vehicleList has the responsibility of collapsing all vehicle tiles that are not

the selected vehicle. To do this, the vehicleList loops over each vehicle comparing its **id** with the selected vehicle **id**. It uses the **option** method to get the **id** of the current vehicle in the loop.

The previous diagram illustrated interactions related to layout changes and animations. The following diagram illustrates an example of an interaction related to refreshing data in response to a global event. When the **Fulfill** button on the reminders pane is selected, it publishes its fulfilling status, makes the Ajax to save the reminder, and finally publishes an **mstats.events.vehicle.reminders.fulfilled**.



At this point, the reminders widget has not yet updated its UI with an updated list of reminders. It does have this responsibility, but only when it is told to do so by the infoPane.

These interactions illustrate the flexibility of a modular design. The remainder of the diagram shows the specific responsibilities of all widgets involved in the global event of fulfilling a reminder.

So far in this chapter you have seen modules have a variety of responsibilities. These responsibilities can be categorized into types.

Types of Modules

There are three common types of modules: the ones you see, the ones you only see the effects of, and the ones you don't see. These types directly relate to the responsibility the module. UI widgets are responsible for visual elements, behavior widgets add functionality, and infrastructure modules aren't associated with elements.

UI

 UI widgets take responsibility for the UI for an element. They can be general purpose such as date and time pickers, combo boxes, or tab controls. They can also be application specific such as the Mileage Stats widgets. The UI widgets in Mileage Stats include the vehicleList, infoPane, vehicleDetails, fillups, reminders, registration, statistics, summary, status, header, and charts. They may rely on solely on the HTML and CSS for the appearance of the widget. Alternatively, the widget may be applied to a single element that doesn't have any child elements, like in the case of the infoPane widget.

If you have a large application with many different views, coordinating a large number of widgets at the application level can lead to complex, monolithic logic. You can prevent this by designating a widget as a container for other widgets. An example of a container in Mileage Stats is the summary widget that coordinates the registration, statistics, and imminentReminders widgets. It is generally acceptable for containers to have knowledge of their children as they are often responsible for creating their children, attaching children to the correct elements, and responding to events from their children. However, you should avoid creating components that have knowledge of their parent because this makes it more difficult to compose and test.

Behavior

• Behavior widgets add functionality to an existing element. The jQuery UI project calls these *interactions* and it includes widgets such as draggable, droppable, resizable, selectable, and sortable. In Mileage Stats, the behavior widgets include tile, pinnedSite, and layoutManager.

Infrastructure

Infrastructure widgets provide commonly needed functionality that isn't related to the visual aspects of the application. They don't interact with the UI. Examples include data access, communication, logging, or caching strategies. The infrastructure modules in Mileage Stats include dataManager. dataStore, and pubsub.

Module Implementations

This chapter is primarily about how to use jQuery UI widgets to compose the client-side, but widgets aren't the only option when it comes to writing modular code. You can also use generic JavaScript objects or, if you're using jQuery and don't need the facilities of a widget, plugins are a good choice.

Overall Approach

Once you have decided to support modularity, you must decide how to use each of these types and you have two options. You can choose the most appropriate type for each module in the solution as described in the next section or you can take the approach Mileage Stats did by using widgets for everything that is associated with an element and objects for everything else.

The benefits of the first choice should be obvious. Each module will only have as much functionality as it needs. This will likely increase performance and portability. The disadvantage of this approach is it requires the development team to understand how to choose between objects, plugins, and widgets, as well as how to implement each one. Based on the size and experience level of the development team, consistency could suffer with this choice.

Mileage Stats uses widgets for all modules attached to elements and JavaScript objects for all other modules.

Objects

JavaScript objects are the most basic implementation of a module. This is a good choice when the module isn't associated with any elements on the page. In Mileage Stats, modules implemented as plain JavaScript objects include dataManager, dataStore, mstats.events, pinnedSite, pubsub, and vehicleDropDownMonitor.

Plugins

One of the characteristics of a high quality framework, such as jQuery, is a robust extensibility mechanism. Creating a jQuery plugin is *the* recommended way to extend jQuery. In fact, a plugin that follows the recommendations in the jQuery Plugin Authoring Guidelines is indistinguishable from the core library. Many methods in jQuery began life as an external plugin and were added later.

Plugins manifest themselves as functions alongside the other jQuery functions. As a result, they can be invoked on elements using the full power of jQuery selectors. As an added advantage, the **this** keyword is the jQuery wrapped set inside the function body.

```
JavaScript

(function($){
    $.fn.doubleSizeMe = function() {
        return this.each(function() {
            var $this = $(this),
                width = $this.width(),
                height = $this.height();
        $this.width(width * 2);
        $this.height(height * 2);
        });
    };
})(jQuery);
```

The closure above adds the **doubleSizeMe** method to the jQuery prototype so it's available when operating on a wrapped set. For example, to invoke it on all elements with a class of **icon**, you would use the following call.

JavaScript	Copy Code
------------	-----------

\$('.icon').doubleSizeMe();

There is much more functionality you can add to your plugins. However, if you need more than additional functions to partition the logic in your plugin, widgets may be more appropriate for your module.

For more information on authoring plugins, see the "jQuery Plugin Authoring Guidelines" in the "Further Reading" section at the end of the chapter.

Widgets

jQuery UI widgets are jQuery plugins that include many of the features that are common to a lot of plugins. Specifically, widgets include features for managing lifetime, storing state, merging options, and exposing public methods. Depending on the features you need, if you are already using jQuery UI, it might be more appropriate to define the module as a widget rather than add these features to a plugin. To learn more about how to build widgets, see Chapter 3, "jQuery UI Widgets."

Summary

Choosing a modular design allows your codebase to be more maintainable by making it easier to understand. This makes troubleshooting and applying future changes much less costly. JavaScript objects are a good choice for implementing these modules when they aren't associated with elements on the page. When the boundaries of the module are defined by elements, jQuery and jQuery UI widgets provide a robust environment for modularizing your solution. When choosing these boundaries in your solution, consider the various layouts in the application and the regions in those layouts. Also, consider animations and content that must be updated via Ajax when identifying boundaries. Once identified, the modules will fall into being related to UI, behaviors, or the underlying infrastructure. Independent of the types of modules you use, your solution will benefit from being composed with widgets.

Further Reading

jQuery Plugin Authoring Guidelines: <u>http://docs.jquery.com/Plugins/Authoring</u>

To learn more about how to build widgets, see Chapter 3, "jQuery UI Widgets."

To learn more about how the tile widget performs its animation, see Chapter 9, "Navigation ."

To learn more about the data abstraction in Mileage Stats, see Chapter 10, "Client Data Management and Caching."

| Community

To report documentation errors or provide feedback on this documentation, please send email to pagdoc@microsoft.com

Copyright $\ensuremath{\mathbb{C}}$ 2011 by Microsoft Corporation. All rights reserved.

Silk 1.0 - June 2011 Client Data Management and Caching

Introduction

Web applications are in the business of presenting data to the user. In rich, interactive, client-centric applications like Mileage Stats, users expect the application to respond quickly to mouse gestures, page transitions, or saving form data. Delays caused by data retrieval or saving can negatively impact the users experience and enjoyment of the site.

A sound client-side data management strategy is critical to the success of a web application and will address fundamental concerns such as:

- **Maintainability**. Writing clean maintainable JavaScript code requires skill, discipline, and planning. The Mileage Stats data implementation addresses maintainability by providing a simple object for other application objects to use to execute data requests and the cache results.
- **Performance**. Client-side caching and prefetching of data, plays a key role in achieving application responsiveness from the user's perspective. Eliminating unnecessary data calls to the server enables the browser to process other tasks quicker such as animations or transitions. In addition, maintaining application responsiveness while its retrieving data from the server is a key factor in perceived responsiveness. Mileage Stats addresses these performance concerns by caching data returned from the server, uses prefetching to acquire data that a user is likely to view, and uses Ajax to perform asynchronous data calls to the server.
- Scalability. Client-side objects should avoid making repeated requests to the server for the same data. Unnecessary, calls to the server require additional resources which can impact the scalability of your application. Mileage Stats uses client-side data caching to increase the scalability of the application.
- **Browser support**. The data cache implementation can influence which browsers the application can support. Mileage Stats caches data using a generic JavaScript object so that older browsers such as Windows Internet Explorer 7 can be used to view the application.

In this chapter you will learn:

- Benefits of a client-side data manager and the abstraction of data requests
- How to improve application performance by caching and prefetching

data

The technologies and libraries discussed in this chapter are Ajax, JavaScript, jQuery, and jQuery UI Widgets.

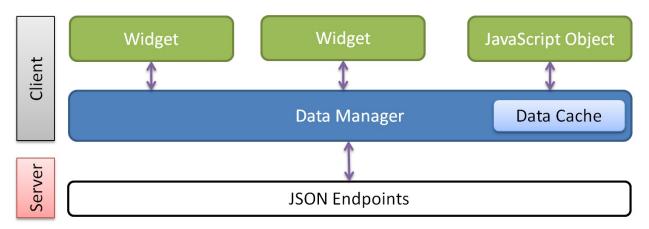
⊠Note:

Data validation on the client and server is covered in the next chapter, "Server-Side Implementation."

Client-Side Data Design

The Mileage Stats data solution centers on the data manager which handles client-side data requests and manages the data cache. The diagram below shows the relationship of the client-side JavaScript objects to one-another and the server JSON (JavaScript Object Notation) endpoints.

Mileage Stats client-side data architecture



Mileage Stats objects use URLs when requesting data from the server. URLs were chosen because their use simplified the data manager's design by providing a mechanism to decouple the server JSON endpoints from the data manager's implementation.

The URL contains the JSON endpoint, and optionally a data record key value corresponding to the requested object. The URL typically aligns with the UI elements the object is responsible for. For example the reminders widget uses "/Reminder/JsonList/1" to retrieve the reminders for the vehicle with the ID of 1.

When data is requested from the data manager it returns the data to the caller and optionally caches the data. The caching of data provides a performance boost to the application because repeated requests to the server for the same data are no longer necessary.

In addition to data caching, Mileage Stats also prefetches the chart data. The chart data is prefetched on the initial page load because there is a reasonable expectation that the user will use the Charts page to compare their vehicles. The prefetching of this data enables instant application response when the user navigates to the Charts page.

In your applications, the amount of data you elect to prefetch should be based on volatility of the data, the likelihood of the user accessing that data, and the relative cost to get that data when the user requests it. Of course, the number of concurrent website users and the capabilities of your web server and database server also play a role in this decision.

Data Manager

All Ajax data requests are routed through the **dataManager** JavaScript object contained in the mstats-data.js file. The data manager is responsible for performing data requests and managing interactions with the data cache. The data manager has a simple public interface that exposes two methods, **sendRequest** which processes Ajax calls to the server, and **resetData** which removes a requested item from the data cache.

The next three sections examine the benefits of the data manager abstraction, look at how data requests are initiated by jQuery UI widgets, and show how those requests are executed by the data manager.

Benefits of the Data Manager Abstraction

Abstracting the data request implementation to a data manager object provides an injection point for the cross-cutting concern, data caching. Data requestors get the full benefit of data caching without taking another dependency or implementing additional code. Isolating the data cache also makes changing the data cache implementation much easier because only the data manager has a direct dependency on it.

The data manager improves application testability by being able to unit test data requests and caching in a single place by having the data request code in a single object.

The data manager also facilitates changing the application over time. Evolution of an application is required not only after release but during development as well. For example, the team added a feature to the data manager that would have required modifying all the Ajax request code. Had the data manager not been implemented, the change would have had more risk and potential cost.

This added feature was the result of deployment testing in various server configurations. The team discovered when the website was deployed to a virtual directory as opposed to the root directory of the web server, that URLs in the JavaScript code had not taken the virtual directory into account. The fix for this problem only had to be applied to the data manager, which saved the team development and testing resources. This feature is discussed in the "Performing Ajax Request" section below.

Data Request

Client-side data requests in Mileage Stats are initiated by jQuery UI widgets and JavaScript objects and performed by the data manger. The data manager **sendRequest** method has the same method signature as the jQuery **ajax** method. Widgets making requests set their calls up as if they are calling jQuery **ajax**, passing an **options** object that encapsulates the URL, **success** callback, and optionally an **error** callback or other callbacks such as **beforeSend** or **complete**.

Data Request Options

When a widget is constructed, the options provided supply the methods needed to execute a data request or remove an item from the data cache. Externally configuring widgets removes tight coupling and hard-coded values from the widget. Widgets can also pass their options, like **sendRequest**, to other widgets that they create.

This technique of external configuration enables Mileage Stats widgets to have a common data request method injected during widget construction. In addition to run-time configuration, this technique also enables the ability to use a mock implementation for the data request methods at test-time.

Below, the **summaryPane** widget is constructed, setting its **sendRequest** and **invalidateData** options to corresponding data manager methods. The summary widget does not make any data requests; instead these two methods will be passed into child widgets created by the summary widget.

JavaScript

Copy Code

```
// contained in mileagestats.js
summaryPane = $('#summary').summaryPane({
   sendRequest: mstats.dataManager.sendRequest,
   invalidateData: mstats.dataManager.resetData,
   publish: mstats.pubsub.publish,
   header: header
});
```

In the below code snippet, the summary widget is constructing the child widget

statisticsPane and passes the above **sendRequest** and **invalidateData** data manager methods as options. Setting these options replaces the default implementation defined in the statistics widget for making data requests. Now, when the statistics widget performs a data request, the method defined in the data manager will be executed.

```
JavaScript
```

```
Copy Cc
```

```
// contained in mstats.summary.js
_setupStatisticsWidget: function () {
  var elem = $('#statistics');
  mstats.summaryPane.statistics = elem.statisticsPane(
     sendRequest: this.options.sendRequest,
     dataUrl: elem.data('url'),
     invalidateData: this.options.invalidateData,
     templateId: '#fleet-statistics-template'
  });
},
```

The **dataUrl** option is the URL or endpoint for the data request. The **url** value is stored in the below data dash attribute in the HTML. The **statisticsPane** widget is attached to and is queried by the above **elem.data** method call. Externally configuring data endpoints avoids hard-coding knowledge about the server URL structure within the widget.

CSHTML

Performing a Data Request

Specifically, the **sendRequest** method has the same method signature as the jQuery **ajax** method that takes a settings object as the only argument. The below **__getStatisticsData** method passes the **sendRequest** method an object literal

that encapsulates the **url**, **success**, and **error** callbacks. When the Ajax call completes, the appropriate callback will be invoked and its code will execute.

JavaScript

Copy Code

```
// contained in mstats.statistics.js
_getStatisticsData: function () {
   var that = this;
   that.options.sendRequest({
      url: that.options.dataUrl,
      success: function (data) {
        that._applyTemplate(data);
        that._showStatistics();
      },
      error: function () {
        that._hideStatistics();
        that._showErrorMessage();
      }
   });
},
```

The above pattern simplified the Mileage Stats data request code because this code does not need to know about the data caching implementation or any other functionality that the data manager handles.

Now that you understand how widgets and JavaScript objects initiate a data request, let's examine how the data manager makes the Ajax request to the server and see how data caching is implemented.

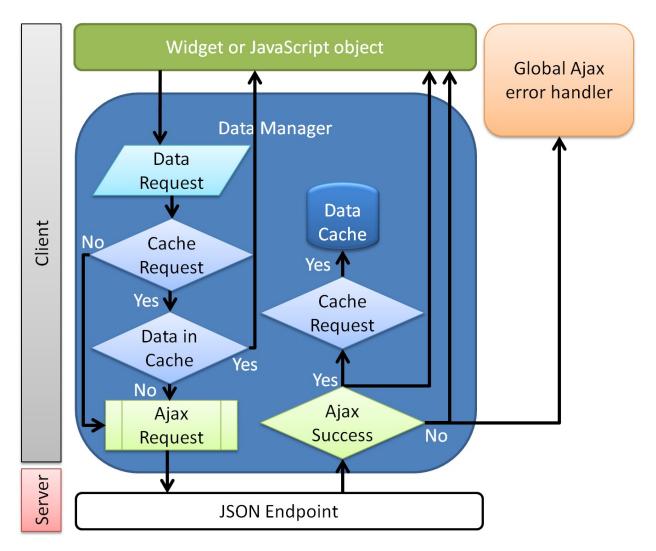
Performing Ajax Request

The data manager **sendRequest** method is used to request data from the server. Since the jQuery **ajax** method signature is the same for requesting as well as posting data, the team chose to implement a single method for Ajax calls to the server. In addition to **success** and **error** callbacks, the **sendRequest** method has an option to cache the request or not. By default, requests are cached.

Mileage Stats has two use cases where data is not cached: data requests that only post data to the server, and the Pinned Sites requests. The Pinned Sites requests are not cached because these requests are only initiated by events after data has changed. Since Pinned Sites only refreshes its data after a change, the data request needs to get fresh data from the server.

The diagram below illustrates the logical flow of a data request. The data manager services the request by first checking if the request should be cached and if so, checks the cache before making a call to the server. Upon successful completion of the request, the resulting data will be returned to the user and added to the cache according to the option.

Data request



Now let's look at the code that implements the functionality of the above diagram. The below **sendRequest** method first modifies the URL to account for the virtual directory the website is deployed to by calling the **getRelativeEndpointUrl** function. Using the modified URL, it attempts to retrieve the requested data from the data cache. The options are then merged with the data manager's default options. If the caller wants the data cached, and data was found in the cache, it's immediately returned to the caller. If the data is not found, the jQuery **ajax** call is made. If successful and the caller requested the data to be cached, it is added to the cache and the caller's **success** callback is invoked. If an error occurs and the caller implemented an **error** callback, it will be invoked after the **error** callback.

⊘Note:

jQuery **ajax** method can be configured at the global level to define default options as well as default event handlers. Mileage Stats defines the global Ajax error handler shown above.

For more information about how Mileage Stats implements the global Ajax error handler, see the "User Session Timeout Notification" section in Chapter 6, "Application Notifications."

JavaScript

```
// contained in mstats.data.js
sendRequest: function (options) {
 // getRelativeEndpointUrl ensures the URL is relativ
  var that = mstats.dataManager,
            normalizedUrl = mstats.getRelativeEndpoint
            cachedData = mstats.dataStore.get(normaliz
            callerOptions = $.extend({ cache: true },
                                        that.dataDefaul
                                        options,
                                        { url: normaliz
 if (callerOptions.cache && cachedData) {
    options.success(cachedData);
    return;
 }
 callerOptions.success = function (data) {
    if (callerOptions.cache) {
      mstats.dataStore.set(normalizedUrl, data);
    }
    options.success(data);
 };
 $.ajax(callerOptions);
},
```

⊠Note:

getRelativeEndpointUrl is a utility method in the mstats.utils.js file that is

used to modify the URL passed in the argument, inserting the virtual directory the website is installed under. This is necessary since the virtual directory is not known until run-time.

Data Cache

The Mileage Stats data manager uses an internal data cache for storing request results; the data cache is only accessed by the data manager. Making the data caching internal to the data manager allows the caching strategy to evolve independently without affecting other JavaScript objects that call the data manager.

The data cache is implemented using a JavaScript object named **dataStore** that is contained in the mstats-data.js file. Other data cache storage locations could include the DOM, browser data storage API or 3rd party library. The **dataStore** JavaScript object was implemented because Mileage Stats supports Internet Explorer 7, which does not support the HTML5 web storage specification and the team chose not to use a shim or polyfill.

Adding and Retrieving Cached Data

Mileage Stats integrates client-side data caching into the data manager's **sendRequest** method implementation that was described in the previous section.

Internally, the **dataStore** is implemented using a name value pair strategy. It exposes three methods: **get** to retrieve data by a name, **set** to cache data by a name, and **clear** to remove data corresponding to a name.

JavaScript

Copy Code

```
// contained in mstats.data.js
mstats.dataStore = {
    __data: {},
    get: function (token) {
        return this._data[token];
    },
    set: function (token, payload) {
        this._data[token] = payload;
    },
    clear: function (token) {
        this._data[token] = undefined;
    },
};
```

Removing a Data Cache Item

In addition to the data manager retrieving and adding data to the cache, the data manager also provides the **resetData** method for removing cached data by URL.

```
JavaScript
// contained in mstats.data.js
resetData: function (endpoint) {
    mstats.dataStore.clear(mstats.getRelativeEndpointUr]
}
```

Mileage Stats objects call the **resetData** method when client-side user actions make the cached data invalid. For example, when a maintenance reminder is fulfilled, the below **requeryData** method will be called by the layout manager widget. When designing your data architecture, it is important to consider which client-side actions should invalidate the cache data.

```
JavaScript
```

Copy Code

```
// contained in mstats.statistics.js
refreshData: function () {
   this._getStatisticsData();
},
requeryData: function () {
   this.options.invalidateData(this.options.dataUrl);
   this.refreshData();
},
```

The **requeryData** method first invokes the **invalidateData** method passing the URL of the cache item to remove. **invalidateData** is an option on the statistics widget which was passed the data manager's **resetData** method when the widget was created. Now that the data cache item has been removed, the next call to **refreshData** will result in the data manager not locating the cached data keyed by the URL, and will then execute a request to the server for the data.

Summary

In this chapter, we have examined the design, benefits, and implementation of a centralized client-side data manager that executes all Ajax requests and manages the caching of data. We have seen how this approach simplifies testing, facilitates application or external library changes over time, and provides a consistent pattern for objects to execute data requests.

We also learned how Mileage Stats keeps its widgets free from external dependencies and the hard-coding of server URLs by constructing and configuring them externally. This approach of injecting external dependencies increases the flexibility and maintainability of the JavaScript code, and the absence of hard-coded server URLs averts brittle JavaScript code.

Further Reading

For detailed information on jQuery UI widgets, see Chapter 3, "jQuery UI Widgets."

For detailed information on the global Ajax error handler, see Chapter 6, "Application Notifications."

For information on data validation, see Chapter 11, "Server-Side Implementation."

HTML 5 Web Storage: http://dev.w3.org/html5/webstorage/

jQuery: http://jquery.com/

jQuery ajax() method: http://api.jquery.com/jQuery.ajax/

jQuery data() method: http://api.jquery.com/data/

jQuery ajaxError() method: http://api.jquery.com/ajaxError/

Ajax Programming on Wikipedia: <u>http://en.wikipedia.org/wiki/Ajax_(programming)</u>

| Community

To report documentation errors or provide feedback on this documentation, please send email to pagdoc@microsoft.com

Copyright $\ensuremath{\mathbb{C}}$ 2011 by Microsoft Corporation. All rights reserved.

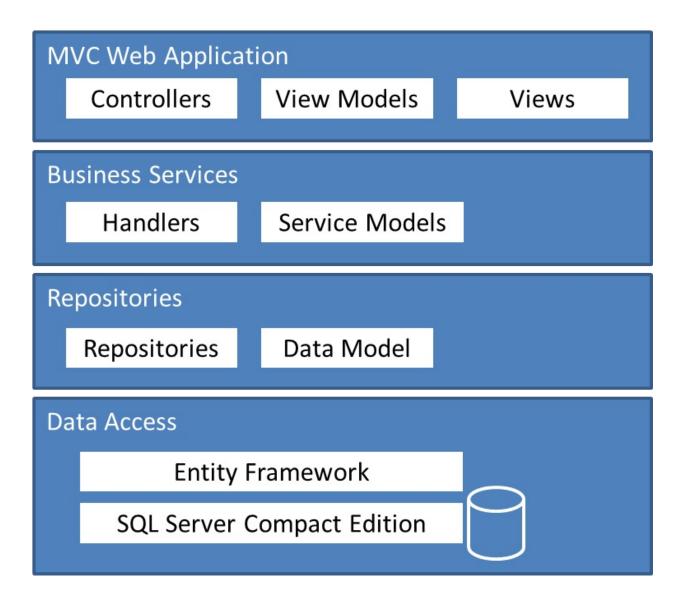
Silk 1.0 - June 2011 Server-Side Implementation

Introduction

Crafting a well-architected web server application requires meeting the needs of the web client while properly factoring the web server .NET Framework code. A web server application is responsible for more than just returning HTML content given a URL. Data models, data access and storage, security, communication, resource management, and internationalization are all part of creating a web server application. This chapter covers ways you can integrate technologies in the Microsoft web platform into a coherent architecture that is reliable, testable, and capable of handling demanding web client applications.

The following diagram shows the architecture of the Mileage Stats Reference Implementation (Mileage Stats). The data access and repository layers are covered first, then the MVC and business services layers are discussed. In the context of these layers, we will also discuss the distinctions between data models, domain models, and view models. Lastly, we'll show you how to provide asynchronous data and validation to web clients.

Mileage Stats High Level Architecture



What you will learn in this chapter

- Leveraging Entity Framework and SQL Server Compact to create a data model.
- Techniques to separate concerns between your data model, business logic, and user interface.
- How to support interactive web clients with asynchronous data.
- Managing data validation at each level of the stack.

The technologies discussed in this chapter are ASP.NET MVC 3, Entity Framework 4, SQL Server Compact Edition 4, and Unity Application Block 2.0.

Creating a Data Access Layer

Data access is a key part of your application. The choice of storage technology and data access patterns can affect the entire application. This section covers an approach using rapid modeling techniques and tools while allowing you to migrate to high-scale data storage in the future.

A well-designed data access layer captures key truths about the data and avoids conditional logic specific to the application. When you separate concerns that are specific to data access from those that specific to the application's logic, the application remains robust and maintainable as you add features over time. The typical concerns of a data access layer include the type of data, the relationships between entities, and constraints.

The data you store is often in a format that is optimized for the storage technology, such as a relational database. Frequently, this format is not convenient for consumption by the application. For example, duration may be stored as a number representing the number of computer clock ticks, but having an instance of a **TimeSpan** would be easier for the application to use. In this case the data access layer should encapsulate the translation between the storage and in-memory formats. Ideally, the data access layer should not contain any user interface or application logic. It should fully abstract the underlying storage implementation.

In Mileage Stats, the **MileageStats.Model** project contains the data model. The data model is part of the data access layer. The structure and strong-typing of the classes in this project express the data types, relationships, and constraints inherent to the data. For example, the **PricePerUnit** property of the **FillupEntry** class is a **double** to allow for dollar and cents, the **Fillups** property of the Vehicle class is an **ICollection<FillupEntry**> to express a one-to-many relationship, and the **DueDate** property of the **Reminder** class is a nullable **DateTime** to allow it to be optional.

When your application has significant complexity or conditional interaction with the data, you should consider creating a separate domain model that is distinct from your data model. See the "Composing Application Logic" section for guidance about whether or not to create a separate domain model.

Rapid Data Modeling using the Entity Framework and SQL Server Compact

The Entity Framework provides three ways for you to rapidly create a data model. You can use the code-first approach to author standard classes that the Entity Framework uses to generate a database schema. Alternatively, you can use the database-first approach where the Entity Framework generates data model classes from an existing database. Finally, you could chose use the model-first approach where an Entity Data Model (.EDMX) can be used to generate the code and database.

The code-first approach is well suited for scenarios like Mileage Stats where developers are defining a new data model that will likely evolve as the application is written and there is not an existing database. If you have an existing database, prefer to use stored procedures, or have a data architect on your team then you may prefer a more traditional database modeling techniques that lets you generate the data model code from the database.

Using SQL Server Compact with the Entity Framework allows you to use an ondisk database that can easily be recreated whenever your schema changes. It can be seeded with a small dataset useful for debugging and unit testing. SQL Server Compact provides a minimal footprint and can be migrated to SQL Server Express, SQL Server, or SQL Azure when the application is deployed.

⊠Note:

SQL Server Compact provides query and update functionality, but does not support conditional syntax (such as IF EXISTS) nor stored procedures. Consider other SQL Server editions as your starting point if you need database-centric logic.

Mileage Stats uses the code-first approach with the Entity Framework and SQL Server Compact. This allowed the data model to be built quickly, adapt to changes, and minimized the day-to-day cost of database setup for the development team.

The Entity Framework lets you easily seed your database with sample data each time the database is rebuilt. This gives you the opportunity to use realistic sample data while you develop the application. We discovered many issues early in the development process of Mileage Stats because the sample data forced the user interface and application logic to work with realistic data.

To use the code-first approach, you first create plain old CLR object (POCO) classes. The Entity Framework then infers the database schema from your class structure and your property types. In the following example, the **FillupEntry** class defines properties that the Entity Framework can map to a database schema.

C#

```
// contained in FillupEntry.cs
public class FillupEntry
ł
  . . .
public int FillupEntryId { get; set; }
  public int VehicleId { get; set; }
  public DateTime Date { get; set; }
  public int Odometer { get; set; }
  public double PricePerUnit { get; set; }
 public string Vendor { get; set; }
 public double TotalCost
Ł
get { return (this.PricePerUnit*this.TotalUnits) + thi
}
. . .
}
```

The Entity Framework maps property types like **double**, **string**, and **int** to their equivalent SQL data type. Fields that represent unique entity identifiers such as **FillupEntryId** and **VehicleId** are automatically populated. Calculated properties like **TotalCost** can be added that are not saved to the database.

The Entity Framework has three mechanisms for determining the database schema from the class definition:

• Inspection of the classes to create a schema. Some of the decisions the Entity Framework makes are based on convention. For example, property names that end in **Id** are considered unique identifiers. They are

auto-populated with database generated values when inserted in the database.

- Inspection of data annotation attributes attached to properties. These attributes are found in the **System.ComponentModel.DataAnnotations** namespace. For example, the **KeyAttribute** indicates a unique entity identifier. Attributes such as **RequiredAttribute** and **StringLengthAttribute** cause the Entity Framework to create column constraints in the database.
- Calls to the **DbModelBuilder** as part of database creation. These methods directly determine the data types, entity relationships, and constraints for the database schema.

⊠Note:

Using the data annotation attributes with the Entity Framework affects how the Entity Framework generates the database schema as well as performing validation of values when the data is saving using **DbContext.SaveChanges**. However, using the **DbModelBuilder** only changes the database schema. Which approach you choose can change the error messages you see when invalid data is submitted as well as whether a database call is made.

See the "Further Reading" section for the Entity Framework documentation. It contains the detailed API reference and steps to apply each of these techniques.

Mileage Stats used the **DbModelBuilder** approach to define the storage schema and did not apply any data annotation attributes to the data model classes. This kept database-specific concerns from polluting the data model and allowed changing the database schema, if necessary, for other kinds of database deployments. This approach was part of the decision to create a separate data model and domain model. See the "Creating a Business Services Layer" section for more information on this decision.

The domain model in Mileage Stats uses data annotation attributes extensively. See the "Data Validation" section for details on using attributes for validation.

Using the DbModelBuilder to Create a Data Model

In Mileage Stats, the **MileageStats.Data.SqlCE** project contains the **MileageStatsDbContext** class. A data model built using the Entity Framework has at least one class derived from **DbContext**. This class provides the starting point for accessing the data model. It is also used for defining the model which

results in a database schema.

The **MileageStatsDbContext** class overrides the **OnModelCreating** virtual method and uses the **DbModelBuilder** parameter to provide the Entity Framework more information about the schema. Defining each entity is factored out into separate methods that **MileageStatsDbContext** .**OnModelCreating** invokes. The following example is one of those methods. It builds the model for the **Vehicle** class.

	ш
	Ħ

```
// contained in MileageStatsDbContext.cs
private void SetupVehicleEntity(DbModelBuilder modelBu
{
modelBuilder.Entity<Vehicle>().HasKey(v => v.VehicleIc
modelBuilder.Entity<Vehicle>().Property(v => v.Vehicle
.HasDatabaseGeneratedOption(
DatabaseGeneratedOption.Identity);
modelBuilder.Entity<Vehicle>().Property(v => v.Name)
.IsRequired();
modelBuilder.Entity<Vehicle>().Property(v => v.Name)
.HasMaxLength(100);
modelBuilder.Entity<Vehicle>().Property(v => v.SortOrc
modelBuilder.Entity<Vehicle>().Property(v \Rightarrow v.MakeNan
.HasMaxLength(50);
modelBuilder.Entity<Vehicle>().Property(v => v.ModelNa
.HasMaxLength(50);
modelBuilder.Entity<Vehicle>().HasOptional(v => v.Phot
modelBuilder.Entity<Vehicle>().HasMany(v => v.Fillups)
modelBuilder.Entity<Vehicle>().HasMany(v \Rightarrow v.Reminder
}
```

DbModelBuilder provides a fluent API that allows you to chain calls together because each method returns an object that can be used in subsequent calls. The calls above use the **Entity**<**T**> method to locate the entity based on the type of the class. The chained **Property** method locates a property for that entity. Lambda expressions like **v** => **v.VehicleEntryId** allow the **Property** method to work without having to provide the name of the property as a string. The last method call defines the data model type, relationship, or constraint.

It is possible for you to use data annotation attributes in conjunction with calls to **DbModelBuilder**. Data annotation attributes provide a decentralized approach where relationships and constraints are attached to the class properties. The **DbModelBuilder** approach provides you centralized control of the data model and a more powerful set of modeling options. You should be careful to keep the constraints in sync when mixing approaches. For this reason, it is recommended to choose either using data annotation attributes *or* the **DbModelBuilder**, and avoid mixing approaches.

☑ Note:

There is an order of precedence in the Entity Framework when all three mechanisms are used: **DbModelBuilder** calls override data annotation attributes which override convention by inspection.

Creating the Database

Once you define the data model in code, you need to create the database. When you use the code-first approach, the Entity Framework doesn't create your database until the first request for data occurs. You should create the database on application start up rather than on the first request so that the first user isn't forced to wait. Initializing during application startup also reduces the chance of a race condition during database creation.

∠Note:

Many web applications built using the Entity Framework contain the autogenerated **WebActivatorAttribute** code. This attribute automatically calls the database creation and initialization code. Mileage Stats forgoes this approach because the Unity dependency injection container controls the lifetime of the **MileageStatsDbContext** instance.

In **Global.asax.cs**, the **Application_Start** method initializes the dependency injection container and then initializes the database. The **InitializeDatabase** method uses the dependency injection container to resolve an instance of the **IRepositoryInitializer** and then calls the **Initialize** method. In the following example, the constructor of the **RepositoryInitializer** configures the database connection and initializer and the **Initialize** method requests some data to ensure the database is created.

C#

```
// contained in RepositoryInitializer.cs
publicRepositoryInitializer(IUnitOfWorkunitOfWork)
{
...
Database.DefaultConnectionFactory =
newSqlCeConnectionFactory("System.Data.SqlServerCe.4.@
Database.SetInitializer(
newDropCreateIfModelChangesSqlCeInitializer<MileageSta
}
public void Initialize()
{
this.Context.Set<Country>().ToList().Count();
...
}
```

Initializing the Database

The Entity Framework lets you control how your database is created and initialized through the **IDatabaseInitializer**<**T**> interface and the **Database.SetInitializer** method. You can write the initializer with the logic you need in order to create and populate your database.

In Mileage Stats, the **MileageStats.Data.SqlCe** project contains three classes that can initialize the database: **CreateIfNotExistsSqlCeInitializer**, **DropCreateAlwaysSqlCeInitializer**, and **DropCreateIfModelChangesSqlCeInitializer**. All three inherit from the **SqlCeInitializer** base class that implements the **IDatabaseInitializer**<**T**> interface.

⊠Note:

When you use **NuGet** to add the Entity Framework to your project, the package manager will generate some default initializers similar to those found in Mileage Stats. Mileage Stats classes are modified versions of the

original generated classes. The modifications allow each initializer to share the database seeding code used to start the application with some useful sample data.

Each class implements a different strategy for creating the database. Mileage Stats defaults to the **DropCreateIfModelChangesSqlCeInitializer** to drop and create the database anytime the model schema changes. This can be very useful during product development when the data model is evolving and the database doesn't contain real data.

When deployed to a production environment, you should change the default initializer to **CreateIfNotExistsSqlCeInitializer**. If you deploy a new version of the application where the schema needs to be upgraded you would either need to author an initializer that upgrades the database, or run upgrade scripts pre-deployment of the newer version. Otherwise, you would lose all the data stored in the database.

Optimized Data Access

Many application data models are hierarchical and connected with one-to-many and many-to-many relationships between entities. On the other hand, web applications are connectionless and stateless; they take a request and produce a response. You should avoid loading large model hierarchies for requests that only need a subset of the data. Overloading data places additional processor, memory, and bandwidth pressure on the server and that can limit scalability and performance.

Fortunately, the Entity Framework provides powerful querying support in the **DbSet** class that allows you to return just the data you need. In Mileage Stats, the **VehicleRepository** class uses the **Where** and **Include** methods on the **DbSet** to control the data retrieved as shown in the following code.

C#

D) (

```
// contained in VehicleRepository.cs
public Vehicle GetVehicle(int userId, int vehicleId)
{
return this.GetDbSet<Vehicle>()
.Include("Fillups")
.Include("Reminders")
```

```
.Where(v => v.VehicleId == vehicleId && v.UserId == us
.Single();
}
```

The Entity Framework requires **ICollection**<**T**> properties like **Vehicles**, **Fill-ups**, and **Reminders** be explicitly included. Properties marked optional by dataannotation attributes or **DbModelBuilder** calls such as the **Vehicles.Photo** property must also be explicitly included to be retrieved.

The Entity Framework has additional features to support lazy loading properties and change tracking. When using the code-first approach, lazy loading is done by applying the **virtual** keyword, and change tracking is done through having standard **get** and **set** methods along with using **ICollection**<**T**> for one-to-many relationships. The Entity Framework also supports these features through implementing the **IEntityWithChangeTracker** or **IEntityWithRelationships** interfaces.

Implementing the Repository Pattern

The Repository pattern assists the data model in separating data storage concerns from the application logic. This pattern is especially beneficial when you use the Entity Framework because it allows you to hide Entity Framework-specific classes such as **DbContext** and **DbSet**, to optimize the shape of the data returned to the application, to coordinate updates, and to unit test your application without requiring access to physical data storage. See the "Further Reading" section for a formal definition of the repository pattern.

In the repository pattern, a repository is a set of interfaces and implementations providing methods for data access. The interfaces do not expose any types specific to data storage. You can choose how many repositories to create based on how granular you want to factor the methods and the expected data access pattern from your application.

In Mileage Stats, the **MileageStats.Data** project contains the repository interfaces and the **MileageStats.Data.SqlCe** project contains the implementation. The Mileage Stats repositories map closely to the data entities to match the usage pattern from the business services layer. The following code shows the **IReminderRepository** interface.

C#
// contained in IReminderRepository.cs
public interface IReminderRepository
{
 void Create(int vehicleId, Reminder reminder);
 Reminder GetReminder(int reminderId);
 void Update(Reminder reminder);
 void Delete(int reminderId);

IEnumerable<Reminder> GetRemindersForVehicle(int vehic IEnumerable<Reminder> GetOverdueReminders(int vehicle] DateTime forDate, int forOdometer);

```
IEnumerable<Reminder> GetUpcomingReminders(int vehicl€
DateTime forStartDate, DateTime forEndDate,
int odometer, int warningOdometer);
```

IEnumerable<Reminder> GetFulfilledRemindersForVehicle(}

⊠Note:

Note that the **IReminderRepository** interface returns collections as **IEnumerable**<**T**>, rather than **IList**<**T**> or **ICollection**<**T**>. This was an intentional design choice to prevent the addition of entities to the collections directly. To create a new reminder, the developer must use the **Create** method.

Also note that our actual implementation of the **IReminderRepository** calls **ToList**() before returning the **IEnumerable**<**T**>. This is to ensure that the query is executed inside the repository. If **ToList**() was not called, then the repository would return an **IQueryable**<**T**> and the database would not be hit until something iterated over the **IQueryable**<**T**>. The problem with returning an **IQueryable**<**T**> is that a developer consuming the API is likely to assume that query has already executed and that they are working with the results. If they iterate the query more than once, it will result in multiple calls to the database.

If you specifically want your repository to return queries instead of results, use the **IQueryable**<**T**> on the interface in order to make the intention explicit.

Because web applications and services follow a request/response pattern, incoming data is built from the POST form data. This means the incoming object class was not retrieved from the **DbContext** and cannot be updated because it is not attached to the context. Using the repository pattern with the Entity Framework provides the proper place to deal with attached and detached entities, as well as setting entity state.

In the following example, the **Update** method in the **VehicleRepository** is passed an entity that is not attached to the Entity Framework context. The **Update** method locates the corresponding attached entity, updates the attached entity, and ensures the attached entity state is set correctly.

C#

// contained in VehicleRepository.cs

```
public void Update(Vehicle updatedVehicle)
{
Vehicle vehicleToUpdate =
this.GetDbSet<Vehicle>().Where(v => v.VehicleId ==
updatedVehicle.VehicleId).First();
```

```
vehicleToUpdate.Name = updatedVehicle.Name;
vehicleToUpdate.Year = updatedVehicle.Year;
vehicleToUpdate.MakeName = updatedVehicle.MakeName;
vehicleToUpdate.ModelName = updatedVehicle.ModelName;
vehicleToUpdate.SortOrder = updatedVehicle.SortOrder;
vehicleToUpdate.PhotoId = updatedVehicle.PhotoId;
```

```
this.SetEntityState(vehicleToUpdate,
vehicleToUpdate.VehicleId == 0 ? EntityState.Added : E
this.UnitOfWork.SaveChanges();
}
```

Composing Application Logic

Web client applications built for rich user interactivity are often more complex than those built for clients that post back synchronously on each mouse click and always displaying static HTML in response. Web applications that provide interactive behavior on a single page (via AJAX method calls, JavaScript templates, and secondary resource requests) require thoughtful composition of server application code. This section covers several techniques and considerations to help you create maintainable applications that provide a rich set of services to interactive clients.

Factoring application code with ASP.NET MVC

Because ASP.NET MVC is a web platform technology built around a design pattern, following the MVC pattern is a key step in properly factoring your application logic. Well-designed MVC applications have controllers and actions that are small, and views that are simple. Keeping your application code DRY (Don't Repeat Yourself) as the application is built is far easier than trying to clean it up later.

☑ Note:

The routes you create in global.asax.cs define the URL hierarchy of your application. Defining your URL strategy, routes, and controller topology early in a project can help prevent having to change your client application code later.

Since the majority of the application logic is contained within the models, many MVC applications contain different kinds of models:

- View models are built solely for a view to data-bind against. These models are contained within the MVC application and often follow the same composition hierarchy as the views. They are focused on presentation concerns. That is, they are only concerned with presenting data in the user interface. Sometimes a special type of view model is also used, called a form model, to represent the data coming into an application from the user.
- **Domain models** are built based on the problem domain. They are focused on handling the business logic of the application. They represent the logical behavior of the application independent of the user interface and the storage mechanism. They may be annotated or extended to support some application features such as validation or authentication. Because these models are easy to round-trip to the client browser, they are sometimes contained within view models. Domain models are sometimes referred to as application models or service models as well.
- **Data models** are built for data services and storage. These are not exposed by the application and are often encapsulated behind a services layer.

Organizing your application into these categories of models is a way of separating concerns in your code. This separation is increasingly important as

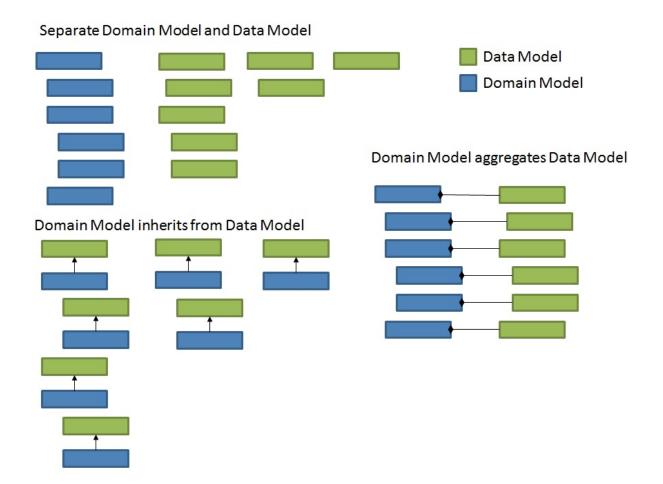
an application grows in complexity. If you find that changes to your application logic are affecting storage or presentations concern (or vice versa), this is an indication that you should factor the code into separate models.

In some cases the models may be very similar to one another. In other cases, the models may radically diverge. If your domain model and your data model are very similar you can consider aggregation to contain an instance of your data model class within your application model class. If your application and data models have a matching hierarchy and compatible interfaces, you can also consider inheritance to derive your application model classes from your data model classes. The following illustration shows the three approaches to domain and data model design.

⊘Note:

The inheritance approach provides the advantage of less coding as you reuse your data model as your domain model, but at the cost of tighter coupling. If you can ensure you will not need to substitute a different data model and that the domain and data models will not deviate, the inheritance approach can be effective.

Three Approaches to Domain and Data Model Design



As you're writing your controller actions, you should factor complex methods into helper methods or classes in your models and services layer. You should prefer action filter attributes such as the **HttpPostAttribute** to avoid writing conditional logic in each action that inspects the **HttpContext**. Also, use action filters for cross-cutting concerns such as authentication (e.g.

AuthorizeAttribute) and error handling (e.g. HandleErrorAttribute). Ideally methods that handle GET should contain only a few method calls and not contain much conditional logic; methods that handle POST should validate the incoming data, perform the update when the data is valid, and conditionally return a view depending on the success of the update. The following examples from Mileage Stats show two versions of the Add method (first the GET version and then the POST version). In these examples, the generic method Using<T> is a helper method used to delegate logic to the classes in the services layer.

C	#
U	#

```
// contained in FillupController.cs
[Authorize]
public ActionResult Add(int vehicleId)
{
var vehicles = Using<GetVehicleListForUser>()
.Execute(CurrentUserId);
var vehicle = vehicles.First(v => v.VehicleId == vehic
var newFillupEntry = new FillupEntryFormModel
{
Odometer = (vehicle.Odometer.HasValue) ? vehicle.Oc
};
var fillups = Using<GetFillupsForVehicle>()
.Execute(vehicleId)
.OrderByDescending(f => f.Date);
var viewModel = new FillupAddViewModel
{
VehicleList = new VehicleListViewModel(vehicles, vehic
FillupEntry = newFillupEntry,
Fillups = new SelectedItemList<Model.FillupEntry>(fill
};
ViewBag.IsFirstFillup = (!fillups.Any());
return View(viewModel);
}
```

C#

```
// contained in FillupController.cs
[Authorize]
[HttpPost]
public ActionResult Add(int vehicleId, FillupEntryForm
{
```

```
var vehicles = Using<GetVehicleListForUser>()
.Execute(CurrentUserId );
if (ModelState.IsValid)
{
var errors = Using<CanAddFillup>()
.Execute(CurrentUserId, vehicleId, model);
ModelState.AddModelErrors(errors, "AddFillup");
if (ModelState.IsValid)
{
Using<AddFillupToVehicle>()
     .Execute(CurrentUserId, vehicleId, model);
TempData["LastActionMessage"] = Resources
                                               .Vehicle
return RedirectToAction("List", "Fillup",
                                              new { ve
}
}
var fillups = Using<GetFillupsForVehicle>()
.Execute(vehicleId)
.OrderByDescending(f => f.Date);
var viewModel = new FillupAddViewModel
{
VehicleList = new VehicleListViewModel(vehicles, vehic
 { IsCollapsed = true },
FillupEntry = model,
Fillups = new SelectedItemList<Model.FillupEntry>(fill
};
ViewBag.IsFirstFillup = (!fillups.Any());
return View(viewModel);
}
```

Injecting dependences through the controller's constructor is also beneficial when unit testing. Since the controller depends on interfaces and not concrete implementations, we can easily replace the actual dependencies with mock implementations. This allowed us test just the code for the action and not the entire functional stack.

After factoring your models and controller actions, your views will use the models to produce the HTML. When building views, you should keep the amount of code to an absolute minimum. Code contained in views is not easily testable. Errors in views are harder to debug because the exception occurs during the rendering pass by the view engine. Some very simple logic in views is acceptable. For example, looping over items to build a repeating section of the user interface or conditional logic for toggling the visibility of specific sections. However, if you find that you need something more complicated then try to push that logic into the view model. If the logic is a cross cutting concern then consider placing the logic inside an HTML helper extension method. Examples of built-in HTML helper extension methods in MVC include **BeginForm, RenderPartial**, and **ActionLink**. Any HTML that is repeated in multiple views is a candidate for being factored into a partial view.

☑ Note:

The MVC Razor syntax allows you to write code more compactly as well as easily mix code and markup. Don't let this powerful view engine tempt you into writing a lot of code within your views. Instead, let it help you keep the code you do write clear and maintainable.

Design checklist for MVC applications

The following checklist is useful when reviewing your MVC web application code.

Check	When reviewing your MVC web application code
	Each controller handles a common set of concerns, either for particular model type or a related set of interactions with the user.
	Actions methods consist of a linear set of calls to helper methods, helper classes, or model classes. They do not contain complex branching conditional logic. They should be easy to unit test and self-documenting.
	The same code is not repeated in multiple action methods. Action filter attributes are used to handle cross-cutting concerns.
	The majority of the application logic is contained within the model or service layer.
	The hierarchy of model classes used by controller actions and views is built to be effective for the application. If required, separate data model classes are contained within another assembly.
	Views contain only small conditional statements and calls to HTML helper methods.
	The same HTML is not repeated in multiple views. Commonly used HTML is factored into partial views.
• See the "Eurther Deading" section for links to more MVC best practices	

• See the "Further Reading" section for links to more MVC best practices.

Creating a Business Services Layer

As you factor your application code from your controllers' action methods into helper methods and classes, you may find there are a set of classes and methods that help to properly retrieve, validate, and update data in your data model. This business logic code is distinguished from the controller code because it encapsulates logical operations on the data model and is *not* specific to any view.

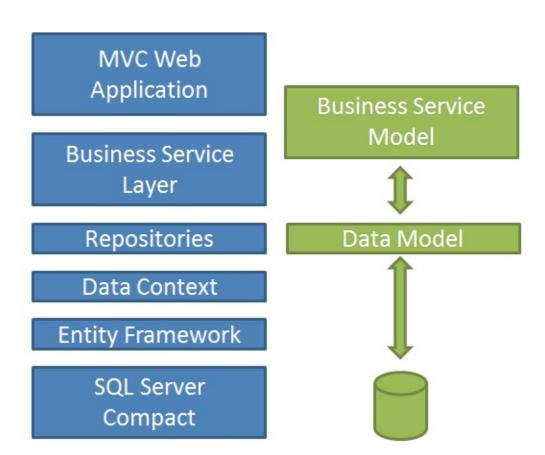
When you have a significant amount of business logic, you may need to create a business services layer. The business services layer is another layer of abstraction, and there is a cost to adding the layer to the application. However, adding this layer allows you to test the business logic in isolation, as well as simplifying the tests for your controllers. Since the service layer is unaware of the user interface, you can also reuse it when exposing additional interfaces in the future such as a web service (using service technologies like Windows Communication Foundation). This can allow you to support both desktop and mobile clients without needing to modify the business service layer.

When deciding whether or not to create a business services layer, you should also consider the decision of whether or not to create a separate domain model. See the "Factoring application code with ASP.NET MVC" section for details on the different kinds of models and techniques for separating a domain model from a data model. Creating a separate domain model along with a business services layer is most beneficial when you need to fully encapsulate your data model, your data model does not perform validation, and the domain model functionality will make it easier for you to write your controllers and views. However, a separate domain model and data model does incur cost transforming between the two models.

The services layer in Mileage Stats consists primarily of handlers and models. The handlers are a set of classes that implement the core behavior of the application. They are completely independent from and unaware of any concerns related the user interface. Reading over the names of the handler classes is like reading a list describing the features of Mileage Stats. The models are a second set of classes in the services layer. They differ from both the data models and the view models. The data models in Mileage Stats are primarily concerned with persisting data to the database. The view models are very specific to needs of the user interface. However, the models in the services layer are not concerned with either persistence or the user interface. The handlers and the models in the services layer represent the business logic of the application. Together they provide validation, calculation of statistics, and other similar services.

For more information on data validation, see the "Data Validation" section.

The following illustration shows the high level design of the service layer and data model.



Mileage Stats Service Layer and Data Model

In following example from Mileage Stats, we see the **Execute** method from the **AddFillupToVehicle** handler. This handler is represented as a single class with a single public method. We chose the convention of naming the method **Execute**. The general dependencies of the handler are injected into the constructor of the handler. Any specific values that may be needed to invoke the handler are passed as arguments into the **Execute** method. Unity is responsible

for managing and injecting the dependencies for the handler's constructor. Whereas the **Execute** method will be invoked by some consumer and it is expected that the consumer will provide the necessary arguments. In the case of Mileage Stats the consumer is a controller action.

Also note that the handler has two private helper methods **AdjustSurroundingFillupEntries** and **ToEntry**. These helper methods are responsible for calculating the statistics and converting the data to the form needed by the data layer respectively. C#

```
// contained in AddFillupToVehicle.cs
public virtual void Execute(int userId, int vehicleId,
ł
if (newFillup == null) throw new ArgumentNullExceptior
try
Ł
var vehicle = _vehicleRepository.GetVehicle(userId, ve
if (vehicle != null)
{
newFillup.VehicleId = vehicleId;
var fillup = newFillup;
var entity = ToEntry(fillup);
AdjustSurroundingFillupEntries(entity);
_fillupRepository.Create(userId, vehicleId, entity);
// update calculated value
newFillup.Distance = entity.Distance;
}
}
catch (InvalidOperationException ex)
throw new BusinessServicesException(Resources.UnableTc
}
}
```

}

In Mileage Stats, the handlers are responsible for implementing the core business logic of the application. The controllers have the responsibility of accepting the users input and invoking the handler. Controllers then take the results of invoking handlers and compose any data necessary for rendering views. This data frequently takes the form of classes that we call view models.

Overall, the business services layer provide functionality that makes writing controllers, actions, and views much easier.

Supporting Interactive Web Clients

Interactive web clients asynchronously communicate with the server and manipulate the document object model (DOM). Because multiple interactions can occur simultaneously, managing state and tracking events can be difficult. This section outlines ways the web application server can support web clients by providing services that reduce the complexity of the JavaScript.

Providing HTML Structure

Traditionally, the server in a web application returns HTML as content that the browser client directly renders. Because interactive web clients manipulate the HTML structure, you will need to focus less on the appearance of the HTML and more on providing a useful hierarchy for the client. You should think of the HTML structure as part of the contract between the client and the server.

In order to modify the content, web clients first need to locate elements in the DOM. The popular jQuery library provides a powerful selector syntax that can be used to locate elements in many ways (e.g. by ID, class, relative position, etc.). If the web client depends on the hierarchical structure of the HTML you produce, you will likely break the client application when you modify the structure. See the "Client-Side Architecture" chapter for more information on the usage of jQuery within Mileage Stats.

To avoid tightly coupling the client JavaScript with the HTML structure, you can use **data-** (pronounced "data dash") attributes on your HTML. The **data-** attributes are attributes whose names are prefixed with "data-".

Many JavaScript developers use the **id** and **class** attributes to locate elements. The **id** attribute is limited since there can be only one per element and their values are generally expected to be unique within a page. The **class** attributes cause confusion because they are also used to apply layout and style to the element through Cascading Style Sheets (CSS).

Since **data-** attributes are orthogonal to the HTML structure, they allow you to restructure the HTML without impacting the client. See the "Client-Side Data Management, Caching, and Validation" chapter for more information on how clients can consume and use **data-** attributes.

Below are two **data-** attribute examples from Mileage Stats. In the first example the **data-vehicle-id** attribute allows the client to locate the associated element. Notice that we are rendering the value for the **data-** attribute on the server and that it will be consumed by JavaScript on the client.

HTML

```
// contained in Views\Vehicle\List.cshtml
<a class="list-item@(item.Reminder.Is0verdue ?"overdue</pre>
```

```
href="@Url.Action("Details","Reminder", new { id = ite
data-vehicle-id="@item.Vehicle.VehicleId">
    ...
</a>
```

In the second example, the **data-chart-url** attribute provides the client a URL to use in an AJAX call.

HTML

```
// contained in Views\Vehicle\List.cshtml
<div id="main-chart"class="article framed"
   data-chart-url="@Url.Action("JsonGetFleetStatisticSe</pre>
```

Ideally, your JavaScript should use only **data-** attribute to locate elements and to discover contextual data from the server. However, there are cases where using a selector to manipulate all elements of a given element name, unique ID, or class is a more practical approach. In these cases, you should author the JavaScript code to allow for the case where the set of selected elements is empty.

⊠Note:

If you have developers writing the web client independent of the web application, we *strongly* recommend you ensure agreement on the expected HTML structure before coding the web client JavaScript.

Using View Model and View Bag

ASP.NET MVC 3 introduced the **ViewBag. ViewBag** is a dynamic object that wraps the **ViewData** property that you might recognize from previous versions of ASP.NET MVC. **ViewBag** is a name/value keyed collection that lets you store any loosely-typed data you like. This differs from the **Model** property on view which contains strongly-type data. Having two ways to provide the view data can cause confusion about when to use **View.Model** vs. **ViewBag**.

The strongly-typed **View.Model** has several benefits over **ViewBag**. It enables IntelliSense auto-complete in the view. It provides type safety when generating the view model from a controller action. Additionally, many of the helpers are specifically designed to work with a strongly-type model and they can extract metadata from the model to help automatically construct a view.

When you use the **View.Model** in a **form** element, you will have an associated controller action (marked with the **HttpPostAttribute**) that accepts the model as a parameter. When the form is submitted, the MVC model binder will use the posted form data to construct and populate an instance of your view model class.

Often the view model representing the form that is passed into a controller action will be significantly different from the view model returned from the controller action. In those cases you may choose to create a *form model* that embodies just the data from the form. An example of this in Mileage Stats is the **Add** action on **VehicleController**. It has a parameter of type **VehicleAddViewModel**. The **VehicleAddViewModel** contains data such as the current user and a list of vehicles, as well as the original form model.

You should prefer to create a view model specific to each of your views. This provides you complete control over the data sent to and from the client. It also reduces confusion by making the relationship between views and view models explicit. Likewise, using form models that are specific to views prevents the ASP.NET MVC model binder from setting properties that you didn't expect to receive from the client. In many cases, if you follow this practice you will never need to use **ViewBag**.

However, there can be cases when your view needs additional data that doesn't belong in your view model and you don't want to round-trip to the client. In

these cases, consider placing the data in ViewBag.

In Mileage Stats, the **_ProfileForm** partial view uses the **User** class as the **View.Model**. Part of the view is a drop-down list of countries. The following example shows the **ViewBag** used to populate the drop-down list of countries.

CSHTML

```
//contained in Views\Shared\ ProfileForm.cshtml
@modelMileageStats.Domain.Models.User
<divclass="editor-label">
@Html.LabelFor(model => model.Country)
</div>
<divclass="editor-field">
@Html.DropDownListFor(model => model.Country, ViewBag.
"-- Select country --", new{ @class ="editor-textbox"})
@Html.ValidationMessageFor(model => model.Country)
</div>
. . .
<divclass="editor-commands">
<buttondata-action="profile-save"class="button generic</pre>
<imgsrc="@Url.Content(
      "~/Content/button-save.png")"
      title="Save Profile"alt="Save"/>
</button>
</div>
<divstyle="clear: both;">
</div>
@Html.ValidationSummary(true)
```

Mileage Stats could have had a separate view model class containing the **User** and an **ICollection<Country>**. However, doing so would reduce the reusability of the partial view because every view model up the hierarchy of views and partial views would have to contain this new view model.

Providing Data Asynchronously

Requesting data asynchronously is at the heart of a responsive, interactive web client. You could use web services, Windows Communication Framework (WCF) services, or even write an **HttpHandler** to serve data to the client. Fortunately, ASP.NET MVC web applications are a great endpoint for serving data to web clients. You can use the same routing, controllers, security, and models that you do for returning HTML structure when returning data. This allows the web client to use the relative URLs you provided in the **data**-attributes as well as some knowledge of the site's URL structure to create requests for data.

Choosing a Data Format

Web clients typically request data as HTML, JavaScript Object Notation (JSON), XML, or as binary (i.e. images, video, etc.) from the server. Each of these formats helps the web client in different ways. You can think of the initial request as just the first data request in a series of requests the client will make.

The JSON format is the recommended choice when the web client needs to bind data to existing HTML elements, generate new HTML from the data, transform the data, or make conditional logic decisions about the data. JSON is a very concise format that has serialization support on the client and in ASP.NET MVC. Because JSON contains no markup, it helps separate user interface and data service concerns.

The HTML format is useful when the client will make minimal or no changes to the returned content and likely place the entire HTML result into a predetermined area of the page. This can work well for scenarios like advertisements, content aggregators, and content management systems.

The XML format is useful when the client receives data based on a pre-defined schema. XML is also used when working open-standards formats such as Really Simple Syndication (RSS), Atom, and oData. Web clients can use the known schema structure to process the XML into HTML (often using XSLT).

Binary formats are generally employed for media. Images are the most common example; the server returns an **img** element with a **src** attribute, the browser makes a secondary request to the server and then renders the binary result as an image.

⊠Note:

Not all browsers send the same data to the server when requesting images and other resources. Some browsers will send authentication headers and cookies while others will not. If you have secondary requests that must be authenticated you will need to verify those requests work on the browsers you intend to support. In addition, you should test both in the ASP.NET development server and in an IIS deployed web application.

Supporting a particular format in ASP.NET MVC consists of returning a **JsonResult**, **ContentResult**, or **FileResult** instead of a **ViewResult** from your action methods.

The following example from Mileage Stats returns a **JsonResult**. The view model is created and then the **Controller.Json** method is called to convert the object into JSON for the response.

C#

```
// contained in VehicleController.cs
[Authorize]
[HttpPost]
publicJsonResultJsonDetails(int id)
{
VehicleModel vehicle = Using<GetVehicleById>()
.Execute(CurrentUserId, vehicleId: id);
IEnumerable<ReminderSummaryModel> overdue = Using<Get(
.Execute(id, DateTime.UtcNow, vehicle.Odometer ?? 0);
JsonVehicleViewModel vm = ToJsonVehicleViewModel(vehic
return Json(vm);
} }</pre>
```

⊠Note:

Controller actions that return a **JsonResult** are easy to unit test because you can directly inspect the **JsonResult.Data** property. Conversely, debugging a serialization issue with a **JsonResult** is harder because it requires inspecting the returned data from the web service in the web client.

Factoring Controller Actions for AJAX

You should continue following the design guidance for factoring your application logic when you incorporate actions that provide asynchronous data.

If you decide to create a separate set of URLs for returning data (i.e. create an independent data service API), you may choose to create separate controllers and routes. This is beneficial when you expect multiple types of clients (e.g. web client, Silverlight, etc.) to use the data actions, but only the web client to use the view-based actions.

If your data actions are closely related to your view-based actions, you may choose to put data actions in the same controller as the view-based actions. Mileage Stats is an example of this scenario because the data actions focus on the same domain models as the view-based actions.

If your web client needs to use the same URL and request different data formats, you may extend your controller action methods by using the **HttpRequestBase.IsAjaxRequest** extension method to determine which format of result to call. This is beneficial when you can reuse your view model as your JSON model. If you find that you have large **if-else** blocks in your controller actions, you should factor the view-based and JSON actions into different helper methods. Alternatively, you could author a custom **AjaxAttribute** action filter that uses **IsAjaxRequest** and provide overloaded action methods similar to how the **HttpPostAttribute** works.

When errors occur, your data actions can throw exceptions just like view-based actions. The jQuery method supports **beforeSend**, **send**, **success**, **error**, and **complete** handlers you can use to handle server responses and failures. If you don't want the friendly error page HTML content returned when a JSON data action throws an exception, you may need to apply a different **HandleErrorAttribute** to your data actions.

As you design your data actions, you should consider how many round trips to the server will be required for each interaction with the user. Every AJAX request requires separate threading on the client as well as resources for the connection, server response, data download, and client processing. If you create overly granular data actions, your web client may suffer performance issues managing a large number of requests to satisfy a user action. If you create monolithic data actions, your web client and server may suffer performance issues because of both the creation and processing of data that isn't required.

⊠Note:

If you haven't already, you may find it helpful to use web browser debugging and tracing tools such as Internet Explorer developer tools, Fiddler, and FireBug to see the relative cost of the different parts of each round trip to the server. Depending on the connection speed and distance between your users and your server, creating a connection can be much more costly than downloading the data once the connection is made. Many web applications created for users across the globe favor requesting larger chunks of data when data cannot be cached closer to the user.

Data Validation

Interactive web applications need to let the user know when they have provided data that is invalid. Data validation checks need to happen on the client to inform the user in the context of what they are trying to accomplish, on the server to protect from untrustworthy callers, and in the database to ensure data integrity. Having data validation occur at multiple levels in the stack makes creating common and consistent validation logic important to the user experience. This section covers data validation techniques you can use to validate your data on both the server and the client.

Data Annotation Attributes

Applying data annotation attributes to your model allows ASP.NET MVC and the Entity Framework to provide data validation at the server level. As mentioned in the "Creating a Data Model" section, the Entity Framework also inspects data annotation attributes on your entity classes to create the database schema. You can find the standard data annotation attributes in the **System.ComponentModel.DataAnnotations** namespace. In this section, data annotation attributes that provide validation are referred to as validation attributes.

In Mileage Stats, data annotations are most commonly found on the form models. The following example shows the validation attributes applied to the **VehicleFormModel** class in the **MileageStats.Domain** project. The attributes applied to the **VehicleFormModel.Name** property validate the name is not null, is not an empty string, is no more than 20 characters, and does not contain script injection characters. Validation attributes also support localization. By using the resource names the error messages are loaded from a RESX file.

C#

```
// Contained in VehicleFormModel.cs
[StringLength(20,
ErrorMessageResourceName = "VehicleNameStringLengthVa]
ErrorMessageResourceType = typeof(Resources))]
[TextLineInputValidator]
[Required(AllowEmptyStrings = false,
ErrorMessageResourceName = "VehicleNameRequired",
ErrorMessageResourceType = typeof(Resources))]
public string Name { get; set; }
```

Validating Data in MVC

The ASP.NET MVC default model binder uses the **Validator** and **ValidationContext** classes when parsing incoming data into an instance of your model class. These two classes work together to validate the data based on the validation attributes you have applied.

If any of the validation fails, **AddModelError** is called on the **ModelState** class. **ModelState.IsValid** returns false when **ModelState** has one or more errors. Because all this happens before your action is called, validating data in your controller actions is that much easier. The following example shows the **FillupController** using **ModelState.IsValid** before making the update.

C#

```
// contained in FillupController.cs
[Authorize]
[HttpPost]
. . .
{
var vehicles = Using<GetVehicleListForUser>()
.Execute(CurrentUserId );
if (ModelState.IsValid)
{
var errors = Using<CanAddFillup>()
.Execute(CurrentUserId, vehicleId, model);
ModelState.AddModelErrors(errors, "AddFillup");
if (ModelState.IsValid)
{
Using<AddFillupToVehicle>().Execute(CurrentUserId, veh
TempData["LastActionMessage"] = Resources.VehicleContr
return RedirectToAction("List", "Fillup", new { vehic]
}
}
```

```
...
var viewModel = new FillupAddViewModel
{
...
};
return View(viewModel);
}
```

⊠Note:

In the previous example, invoking the handler **CanAddFillup** returns a collection **ValidationResult**. These are validation results returned from the business services layer. The **AddModelErrors** extension method iterates over the **ValidationResult** collection and calls **ModelState.AddModelError** for each. This level of indirection keeps the business services layer from depending on ASP.NET MVC.

Creating Custom Validation Attributes

When the standard validation attributes don't provide what you need, you can write your own. All the standard validation attributes derive from the **ValidationAttribute** class containing the abstract **IsValid** method to implement.

The following example shows the implementation to validate a postal code. The implementation is simpler than would be used in many applications, but it shows cross-field validation on a model object.

```
C#
```

```
// contained in PostalCodeValidatorAttribute.cs
protected override ValidationResult IsValid(object val
Ł
varuserToValidate = context.ObjectInstance as User;
var memberNames = new List<string>() { context.MemberN
if (userToValidate != null)
{
if (string.IsNullOrEmpty(userToValidate.Country) &&
string.IsNullOrEmpty(userToValidate.PostalCode))
ł
return ValidationResult.Success;
if (string.IsNullOrEmpty(userToValidate.PostalCode))
{
return ValidationResult.Success;
if (userToValidate.Country == Resources.UnitedStatesDi
{
if (USPostalCodeRegex.IsMatch(userToValidate.PostalCoc
Ł
return ValidationResult.Success;
}
return new ValidationResult(Resources.USPostalCodeVali
memberNames);
}
else
```

```
{
if (InternationalPostalCodeRegex.IsMatch(userToValidat
{
  return ValidationResult.Success;
}
return new ValidationResult(
Resources.InternationalPostalCodeValidationErrorMessaç
}
return ValidationResult.Success;
}
```

Handling Complex Data Validation

You may have noticed in the earlier example that the **FillupController.Add** method calls **ModelState.IsValid** twice. The **CanAddFillup** handler in the following example contains validation logic that uses multiple objects in the domain model and requires additional database access. This validation logic is not suited for a single custom **ValidationAttribute**. It returns a collection of validation results that the controller uses to call **ModelState.AddModelError**. In cases like these, you should factor complex validation logic into helper methods or a business services layer.

```
C#
// contained in CanAddFillup.cs
public virtual IEnumerable<ValidationResult> Execute()
Ł
var foundVehicle = _vehicleRepository.GetVehicle(user]
if (foundVehicle == null)
ł
yield return new ValidationResult(Resources.VehicleNot
}
else
ł
var fillups = _fillupRepository.GetFillups(vehicleId);
if (!fillups.Any()) yield break;
var priorFillup = fillups.Where(f => f.Date < fillup.[</pre>
if ((priorFillup != null) && (priorFillup.Odometer >=
{
yield return new ValidationResult(
"Odometer",
string.Format(CultureInfo.CurrentUICulture,Resources.C
priorFillup.Odometer));
}
}
```

}

Supporting Validation on the Client

ASP.NET MVC supports client-side validation of data by sharing validation information from the server. This is done by implementing the **IClientValidatable** interface on your validation attributes. **IClientValidatable** contains only the **GetClientValidationRules** method that returns a **ModelClientValidationRule** collection.

In the following example, the **PostalCodeValidatorAttribute** implements **GetClientValidationRules** by returning a single **ModelClientValidationRule**. By setting the **ValidationType** property to "**postalcode**" the client will use the validation routine with the same name registered on the client. The validation parameters are added to provide the client-side code the information it needs to implement the validation rule.

C#

```
public IEnumerable<ModelClientValidationRule> GetClier
ModelMetadata metadata, ControllerContext context)
{
  var rule = new ModelClientValidationRule()
  {
  ErrorMessage = Resources.InvalidInputCharacter,
  ValidationType = "postalcode"
  };
```

```
rule.ValidationParameters.Add("internationalerrormessa
Resources.InternationalPostalCodeValidationErrorMessaç
rule.ValidationParameters.Add("unitedstateserrormessaç
Resources.USPostalCodeValidationErrorMessage);
rule.ValidationParameters.Add("internationalpattern",
Resources.InternationalPostalCodeRegex);
rule.ValidationParameters.Add("unitedstatespattern",
Resources.USPostalCodeRegex);
```

```
return new List<ModelClientValidationRule>() { rule };
}
```

When MVC HTML helper extension methods such as **TextBoxFor** and **EditorFor** are called, MVC inspects the property definition for validation attributes. When a validation attribute implements **IClientValidatable**, MVC uses the client validation rules to include **data-val** attributes. The following HTML fragment shows the **data-val** attributes present on the postal code field in the registration form.

HTML

```
<input data-val="true"
  data-val-length="Postal code must be less than 10 ch
 data-val-length-max="10"
 data-val-postalcode="Only alpha-numeric characters a
 data-val-postalcode-internationalerrormessage=
    "Postal codes must be alphanumeric and ten charact
 data-val-postalcode-internationalpattern="^[\d\w]{0,
 data-val-postalcode-unitedstateserrormessage=
    "United States postal codes must be five digit num
 data-val-postalcode-unitedstatespattern="^[\d]{5}$"
  data-val-textlineinput=
    "Only alpha-numeric characters and [.,_-']
 data-val-textlineinput-pattern="^(?!.*--)[A-Za-z0-9)
 id="PostalCode" maxlength="10" name="PostalCode" siz
  />
 <span class="field-validation-valid" data-valmsg-for</pre>
    data-valmsg-replace="true">
  </span>
```

The MVC validation JavaScript (**jquery.validate.js** and **jquery.validate.unobtrusive.js**) provides for the standard validation attributes and provides you methods to register validation routines and unobtrusive validation adapters. The following example shows the registration of the **postalcode** client-side validation routine. Notice how it uses the **params** object to access the **data-val** attributes.

JavaScript

// contained in mstats.validation.js

```
s.validator.addMethod('postalcode', function (value, <math>\epsilon
if (!value) {
return true; // not testing 'is required' here!
}
try {
var country = $('#Country').val(),
postalCode = $('#PostalCode').val(),
usMatch = postalCode.match(params.unitedStatesPattern)
internationalMatch = postalCode.match(params.internati
message = '',
match;
if (country.toLowerCase() === 'united states') {
message = params.unitedStatesErrorMessage;
match = usMatch;
} else {
message = params.internationalErrorMessage;
match = internationalMatch;
}
$.extend($.validator.messages, {
postalcode: message
});
return (match && (match.index === 0) &&
(match[0].length === postalCode.length));
} catch (e) {
return false;
}
});
```

IClientValidatable helps you to share validation information, but you still have two copies of your validation logic to maintain. You may choose remote validators (i.e. implementing validation actions in your controller) and call them using AJAX from the client. However, the round trip to the server will not be as responsive as validating directly on the client.

☑ Note:

It is important to remember that client-side validation only helps improve the user experience and is not a substitute for proper validation and security on the server. Hackers won't use your web client JavaScript or even the browser when maliciously posting data to the web application on the server, so you *must* ensure that any client-side validation is repeated on the server before any data changes occur.

Validating Validation

Because validation occurs at multiple levels of the stack, you may end up with duplicate validation attributes and validation logic to keep in sync. While proper factoring of your application logic, your models, and data validation information can help, you should always unit test each layer in isolation to make sure the validation works as expected.

While you don't need to unit test the standard validation attributes, you should unit test that the validation attributes are properly applied to your model and validate as expected (just as if you had written code inside the setter of your model property). The following example shows a unit test verifying the **Title** of the **Reminder** is required.

C#

```
// contained in ReminderFixture.cs
[Fact]
public void WhenTitleSetToNull_ThenValidationFails()
{
    Reminder target = new Reminder();
    target.Title = null;
    var validationContext = new ValidationContext(target,
    var validationResults = new List<ValidationResult>();
    bool actual = Validator.TryValidateObject(target, validationResult);
    Assert.False(actual);
    Assert.Equal(1, validationResults.Count);
    Assert.Equal(1, validationResults[0].MemberNames.Count
    Assert.Equal("Title", validationResults[0].MemberNames
}
```

⊠Note:

The **true** parameter at the end of the **TryValidateObject** call is important – it causes the validation of all properties. It means that your unit test has to ensure that all other properties are set to valid values when you try to verify

that one invalid property fails validation.

Other Considerations

This section briefly covers some other areas of server architecture you may want to consider.

Dependency Injection

Mileage Stats uses Unity for dependency injection. The unity.config file in the web application maps interfaces to concrete classes. It also determines the lifetime for each mapping. For example, Unity ensures the **VehicleController** constructor provides implementations of the **IUserServices**, **ICountryServices**, **IServiceLocator** and **IChartDataService** interfaces.

In an effort to manage dependencies and improve testability in the MVC pattern, ASP.NET MVC also provides a dependency resolver. This allows ASP.NET MVC applications a suitable place to resolve dependencies for framework created objects like controllers or action filters. In the following example, Mileage Stats registers Unity as the MVC dependency resolver as part of initializing the dependency injection container for the application.

C#

```
// contained in global.asax.cs
privatestaticvoidInitializeDependencyInjectionContaine
{
    IUnityContainer container = new UnityContainerFactor
    var serviceLocator = new UnityServiceLocator(contair
    ServiceLocator.SetLocatorProvider(() => serviceLocat
    DependencyResolver.SetResolver(new UnityDependencyRe
}
```

See the "Further Reading" section for more information on dependency injection and Unity.

Unit Testing

One of the key reasons ASP.NET MVC follows the MVC pattern is to allow for unit testing of the application logic. The **System.Web.Abstractions** assembly was introduced primarily to allow substitution of mocked instances of classes like **HttpContextBase** during unit testing. You should unit test as much of your application logic as possible; it will not only help ensure the quality of your application, but will also help identify design issues early when they are less expensive to fix.

Mileage Stats uses the xUnit unit test framework as well as Moq for mocking interfaces. The application is unit tested at the data model, business services, and controller layers. As mentioned in the "Composing Application Logic" section, keeping controller actions simple and factoring application logic into a business services layer makes unit testing much easier. Mileage Stats is an example of reaping those benefits. Unit testing was much easier because interfaces like **IUserServices** could be mocked.

For more information on quality assurance see the chapter "Testing Web Applications."

Error Management

Web clients expect proper HTTP status code responses when a web application cannot fulfill a request. This means you should avoid hiding errors like a resource not being found (404), failure to authorize (403), and server errors (500+). ASP.NET MVC will respond with the correct HTTP status code when no valid route is found and based on the type of exception when an exception is thrown from a controller action. There may be cases where you need to catch an exception from a call and throw a different exception type.

Generally, users don't want to see all the developer details for an exception. ASP.NET MVC provides a **HandleErrorAttribute** that provides a friendly error page when an exception occurs. The friendly page displayed is determined in the web.config **customErrors** section. Mileage Stats applies the **HandleErrorAttribute** to all controller actions in the **RegisterGlobalFilters** methods.

Although friendly errors are an improvement, the user experience shouldn't be interrupted with HTTP error code if the user enters an invalid value. Use the Post/Redirect/Get pattern (PRG) when handling a POST action. When the user has submitted invalid data, you should return the same view as the GET action populated with the incoming data. When a POST action succeeds, it can redirect.

In the following example, if a **ReminderFormModel** doesn't pass data validation, the **Add** view result is returned populated with the reminder data that was passed into the action method.

C#

```
//contained in ReminderController.cs[HttpPost]
public ActionResult Add(int vehicleId, ReminderFormMoc
{
    if ((reminder != null) && ModelState.IsValid)
    {
        var errors = Using<CanAddReminder>().Execute(CurrentUs
        ModelState.AddModelErrors(errors, "Add");
    }
}
```

```
if (ModelState.IsValid)
```

```
{
Using<AddReminderToVehicle>().Execute(CurrentUserId, \
return RedirectToAction("Details", "Reminder", new { i
}
}
var vehicles = Using<GetVehicleListForUser>()
.Execute(CurrentUserId);
var vehicle = vehicles.First(v => v.VehicleId == vehic
var reminders = Using<GetUnfulfilledRemindersForVehicl</pre>
.Execute(CurrentUserId, vehicleId, vehicle.Odometer ??
.Select(r => new ReminderSummaryModel(r, r.IsOverdue ?
var viewModel = new ReminderAddViewModel
{
VehicleList = new VehicleListViewModel(vehicles, vehic
Reminder = reminder,
Reminders = new SelectedItemList<ReminderSummaryModel>
};
return View(viewModel);
}
```

Concurrency

Because Mileage Stats tracks vehicles per user account, concurrency conflict detection and management was not a scenario for the application. Even though we choose not to make use of it, the Entity Framework does supports optimistic concurrency by adding time stamps to the data model and taking appropriate action when handling the **DbUpdateConcurrencyException**.

Summary

Hopefully you now have a frame of reference for architecting your server-side web application. There are many different choices you will make to structure your site, factor code, and model data. Successful architectures properly balance the layers of abstraction required to solve the problem at hand while keeping an eye on future features and technologies.

Key takeaways:

- Understand your web client needs and build a contract for the HTML structure, URL structure, and data formats between the client and server early in the process.
- Decide on whether or not to create a business services layer along with deciding whether or not to create separate domain and data models.
- Create slim controllers by placing the majority of your application logic in your domain models, a services layer, or helper classes and methods.
- Keep application logic simple and partitioned.
- Provide a data API that allows web clients to consume data asynchronously in the right format and granularity for the application.
- Structure your validation logic to support validation both on the client and on the server.

Further Reading

ADO.NET Entity Framework on MSDN: http://msdn.microsoft.com/library/bb399572.aspx

The Repository Pattern on MSDN: http://msdn.microsoft.com/en-us/library/ff649690.aspx

Unit of Work pattern: http://www.martinfowler.com/eaaCatalog/unitOfWork.html

Catalog of Patterns of Enterprise Application Architecture: <u>http://martinfowler.com/eaaCatalog/</u>

Evans, Eric. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2003.

Nilsson, Jimmy. *Applying Domain-Driven Design and Patterns: With Examples in C# and .NET*. Addison-Wesley Professional, 2006.

Understanding Models, Views, and Controllers on ASP.NET: <u>http://www.asp.net/mvc/tutorials/understanding-models-views-and-controllers-</u><u>cs</u>

Best Practices for ASP.NET MVC: <u>http://blogs.msdn.com/b/aspnetue/archive/2010/09/17/second_2d00_post.aspx</u>

Dependency Injection on MSDN Magazine: http://msdn.microsoft.com/en-us/magazine/cc163739.aspx

Unity Application Block on MSDN: http://www.msdn.com/unity

Post/Redirect/Get pattern: http://en.wikipedia.org/wiki/Post/Redirect/Get

| Community

To report documentation errors or provide feedback on this documentation, please send email to pagdoc@microsoft.com

Copyright $\ensuremath{\mathbb{C}}$ 2011 by Microsoft Corporation. All rights reserved.

Silk 1.0 - June 2011 Security

Introduction

This chapter addresses security-related topics for the Mileage Stats Reference Implementation (Mileage Stats) and is divided into three sections. The first section introduces security threats relevant to Mileage Stats. The second section provides a guided tour of Mileage Stats security features that provide countermeasures against the relevant threats. The third section describes possible security modifications to adjust for changes in the deployment environment and security requirements for the application. After reading this chapter, you should have an understanding of how relevant security threats are mitigated in Mileage Stats and of some of the extensibility points for its security.

In this chapter you will learn:

- Key security threats that you should address in any web application, including unauthorized access, malicious input, content injection, cross-site scripting, eavesdropping, message tampering, message replay, and cross-site request forgery.
- Security features in Mileage Stats that provide countermeasures against the relevant threats for authentication, input validation, anti-forgery, and JavaScript Object Notation (JSON) hijacking.
- Security modifications to adjust for changes in the deployment environment and security requirements for the application.

This chapter will cover some security features of ASP.NET and ASP.NET MVC and OpenID.

Security Threats

The section describes a few key security threats that need to be addressed in any web application. If you're already familiar with the security threats below and how to mitigate them, skip to the next section where the security features for Mileage Stats are described.

Unauthorized Access

To prevent just anyone from casually accessing your website and changing data, you will need to limit who can access it. This is typically accomplished by requiring users to authenticate. The most common form of authentication requires a user to provide his user name and password as credentials. Once verified, the user is permitted access. If the credentials are not recognized by the website, the user is not allowed access.

Malicious Input – Content Injection and Cross-Site Scripting

There are a variety of methods whereby malicious users can attempt to corrupt content by uploading malicious input to your website. Such attacks can result in data corruption or even make your website unusable. If links can be uploaded to a website, a malicious user can potentially execute a cross-site scripting (XSS) attack, enabling them to collect potentially sensitive form data and security information for later exploitation. A common way to prevent the uploading of malicious input to a website is to limit the length and type of input that users are allowed to provide. It is important to limit the range of characters that a user is able to provide for text input. If you remove the ability to submit tags by filtering out tag characters ("<" and ">"), it goes a long way towards preventing malicious users from submitting scripts or HTML tags. As a rule of thumb, input allowed for submission to the website should be as limited as possible based on expected length, content, and data type for a particular data field.

Eavesdropping, Message Tampering, and Message Replay

Eavesdropping, message tampering, and message replay are grouped together because they are often encountered and mitigated by similar measures. A common and relatively simple way to exploit a web application through eavesdropping is to use a network data capture utility to find and record HTTP requests and responses between a website and a client. Without protection from eavesdropping and tampering, an attacker can alter the contents of a captured HTTP request and re-submit it to the website. This type of attack is commonly referred to as a message replay attack. Even if the website requires authentication, it processes the request as if it came from the client since it contains a legitimate security token. HTTP requests can be altered to cause the website to behave undesirably, deleting data, changing data, or causing large numbers of transactions to be executed. A common way to mitigate message replay in web applications using HTTP is by requiring communication via Secure Sockets Layer (SSL). When you use SSL in a non-anonymous mode, you prevent the ability to replay messages back to the server. Two additional and very important benefits of using SSL are that it prevents any sensitive content in the HTTP traffic from being disclosed to eavesdroppers and prevents messages from being tampered with.

Cross-Site Request Forgery

Cross-site request forgery (CSRF, often pronounced as "sea surf",) is an attack whereby malicious commands are sent to a website from the browser of a trusted user. An attacker constructs a seemingly harmless HTML element on a different website that surreptitiously calls the target website and attempts to do something malicious while posing as a trusted user. CSRF has great potential to damage the website being exploited; an attacker can potentially tamper with or delete data, or execute large numbers of unwanted transactions on the targeted website.

This section is intended as an introductory overview and is not a substitute for more comprehensive guidance or a threat model. For more information on ASP.NET security, see the <u>ASP.NET Web Application Security</u> reference at the end of this chapter.

Web Platform Security

This section describes some of the out-of-the-box security built into the various components of the web application platform.

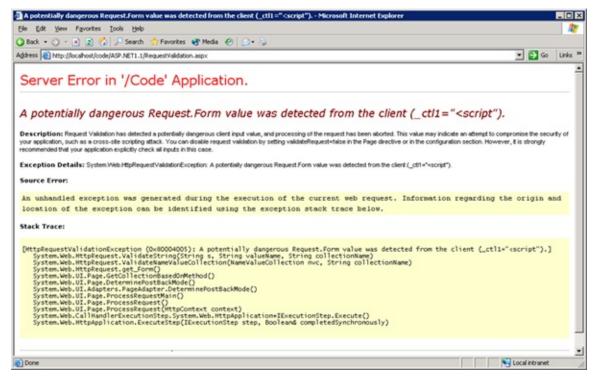
MVC View Encoding

The Razor syntax uses the *@* operator in MVC views to specify Microsoft® .NET Framework code. Any output written to the client in the view from code using the *@* operator is automatically HTML encoded for you. This traps malicious content by preventing it from being rendered back to the client. Trapping malicious content goes hand-in-hand with input filtering, because you should never assume that any data that comes from users or other applications is safe.

ASP.NET Input Filtering

Starting with version 1.1, ASP.NET started providing input filtering out of the box as a secure default. Any attempt to submit a request containing bracketed tags "<" or ">" in any of its form data, query string parameters, or cookies results in an error page indicating that malicious input has been detected. The figure below is a screenshot of the default ASP.NET input validation failure page:

Input validation failure



While this is good to have as a secure default, you should still be validating user input in your application because this is only a very basic means of filtering input.

Protecting Application-Specific Data

Sometimes applications host their own data sources instead of accessing them from a centrally hosted location. Whenever you create an application-specific data source, it should be hosted in the App_Data subdirectory. For example, if you add a SQL membership provider for a specific application, the Microsoft SQL Server® Express .mdf file should be created in the App_Data directory. ASP.NET protects application-specific data in a few different ways. Files stored in the App_Data folder cannot be accessed directly by clients because the folder is protected from browsing by default.

An additional layer of protection for application-specific data files is configured using HTTP handlers in the machine's web.config. Requests for certain file types are routed to the ASP.NET **HttpForbiddenHandler** by default. The code below shows the configuration to handle HTTP requests for .mdf and .mdb files used by SQL Server Express and Microsoft Access®, respectively, via the **HttpForbiddenHandler** in the machine's web.config file:

XML

```
<!--contained in machine's default web.config-->
<addpath="*.mdf"verb="*"type="System.Web.HttpForbidder
<addpath="*.ldf"verb="*"type="System.Web.HttpForbidder
```

Note that the HTTP handlers defined above are for standard ASP.NET applications. Handlers for an ASP.NET MVC application are defined via route mappings in the application's Global.asax.cs file. These files would normally be inaccessible when using ASP.NET MVC unless you explicitly create a route mapping for them or you configure ASP.NET MVC to ignore routing for the file type and let ASP.NET handle them using the settings described above. Neither of them is recommended for ASP.NET MVC applications because the files should be protected from client browsing by routing all requests to the MVC controllers. For more information on MVC routing best practices, see the reference on "<u>Best Practices for ASP.NET MVC</u>" on the ASP.NET and Web Tools Developer Content Team's blog at the end of this chapter.

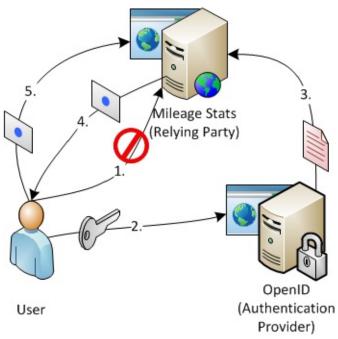
Mileage Stats Application Security

This section explains security measures implemented in Mileage Stats that mitigate security threats against it.

Authentication

Mileage Stats implements authentication using a third-party authentication provider (OpenID) and ASP.NET forms authentication tickets to prevent unauthorized access. Whenever an unauthenticated user attempts to access the site, she is redirected to the login URL via ASP.NET forms authentication. The diagram below depicts the logical flow for user authentication in Mileage Stats.

Third-party user authentication



- 1. The user attempts to access the site without authenticating first and is redirected to the Mileage Stats home page.
- 2. The user navigates from the Mileage Stats home page to the OpenID sign-in page to authenticate.
- 3. An authentication result is attached to the response by OpenID and picked up by Mileage Stats for processing.
- 4. Mileage Stats converts a successful authentication result from OpenID to an encrypted ASP.NET forms authentication ticket and caches it client-side as a session cookie in the web browser.
- 5. The user accesses the site successfully with a valid forms ticket. Whenever a request is processed with a valid authentication ticket, the expiration time of the ticket is reset to provide a sliding expiration window.

The significance of this authentication model is that a third party (OpenID) is responsible for managing and validating the user's credentials as opposed to more commonly encountered authentication models in which the owner of the website is the same party maintaining and validating the user's credentials.

While out-of-the box ASP.NET forms authentication is not being used, encrypted forms authentication tickets and forms ticket validation are being leveraged programmatically. Forms ticketing is an effective mechanism that you can leverage to work with OpenID authentication instead of creating a security ticketing mechanism from the ground up.

There are several configuration settings and segments of code responsible for implementing authentication end-to-end in Mileage Stats. Let's take a look at the configuration and major components.

Configuration

While the forms tickets are being generated manually in Mileage Stats, the code still leverages the configuration settings used for configuring out-of-the-box forms authentication. The following configuration snippet in Mileage Stats web.config is responsible for configuring the application for forms authentication and setting the login redirect URL:

XML

Copy Code

```
<!--contained in web.config-->
<authenticationmode="Forms">
<formsloginUrl="~/Auth/SignIn"timeout="20"/>
</authentication>
```

The relying party class validates a user's credentials with an authentication provider, then uses them to create a forms authentication ticket. The relying party implementation is configured in the unity.config file:

XML

```
<!--contained in unity.config-->
<!--NOTE: This is a real openId authentication mechani
<typetype="MileageStats.Web.Authentication.IOpenIdRely
mapTo="MileageStats.Web.Authentication.DefaultOpenIdRe</pre>
```

MileageStats.Web"> <lifetimetype="perRequest"/> </type>

Note that there are two unity.config files, and the one used by Mileage Stats depends on which solution configuration is selected when you compile the application. The one shown above is used when the Release solution configuration is selected. When Debug is selected, a different unity.config is used that contains a mock authentication relying party class that can be used to get the application up and running. The mock authenticator will put the user through a mock authentication workflow that does not validate any credentials. To deploy the application, we strongly recommend that you compile the application using the Release solution configuration to use the actual relying party class that actually validates user credentials.

Both the mock authenticator and the OpenID relying party class implement the **IOpenIdRelyingParty** interface, which acts as a wrapper around the DotNetOpenAuth interfaces to expose only what is required to interact with the authentication provider and to process the results. Implementing **IOpenIdRelyingParty** enables you to configure a different relying party implementation if your application requirements change.

AuthController

AuthController is an Model View Controller (MVC) controller in Mileage Stats that is responsible for handling the user redirect for authentication and converting the response from a successful authentication attempt into an ASP.NET forms authentication ticket. The AuthController uses the relying party implementation specified in the unity.config file. AuthController's SignInWithProvider method is invoked to redirect the user to the authentication provider's sign-in page. The code snippet below shows the SignInWithProvider method:

C#

```
// contained in AuthController.cs
publicActionResultSignInWithProvider(stringproviderUr]
{
```

```
if(string.IsNullOrEmpty(providerUrl))
{
  returnthis.RedirectToAction("SignIn");
}
varfetch =newFetchRequest();
varreturnUrl =this.Url.Action("SignInResponse","Auth",
this.Request.Url.Scheme);
try
{
  returnthis.relyingParty.RedirectToProvider(providerUr]
}
  catch(Exception)
{
  this.TempData["Message"] =
  Resources.AuthController_SignIn_UnableToAuthenticateWi
  returnthis.RedirectToAction("SignIn");
}
```

The **AuthController**'s **SignInResponse** method is invoked to process the response from the user's authentication attempt with the authentication provider. **SignInResponse** calls the **GetResponse** method on the relying party class and processes the result. If the result is a successful authentication, **AuthController** creates an ASP.NET forms authentication ticket and attaches it to the response. If anything other than a successful authentication result is returned from the authentication provider, the user is redirected back to the authentication provider's sign-in page.

```
C#
```

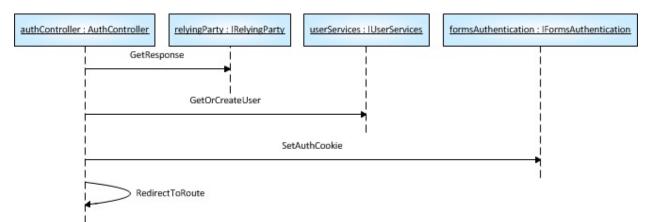
```
// contained in AuthController.cs
publicActionResultSignInResponse(stringreturnUrl)
{
varresponse =this.relyingParty.GetResponse();
```

```
switch(response.Status)
Ł
caseAuthenticationStatus.Authenticated:
varuser =this.userServices.GetOrCreateUser(response.Cl
this.formsAuthentication.SetAuthCookie(this.HttpConte>
UserAuthenticationTicketBuilder.CreateAuthenticationTi
returnthis.RedirectToRoute("Dashboard");
caseAuthenticationStatus.Canceled:
this.TempData["Message"] ="Cancelled Authentication";
returnthis.RedirectToAction("SignIn");
caseAuthenticationStatus.Failed:
this.TempData["Message"] = response.Exception.Message;
returnthis.RedirectToAction("SignIn");
default:
this.TempData["Message"] =
Resources.AuthController_SignInResponse_Unable_to_auth
returnthis.RedirectToAction("SignIn");
}
}
```

The below sequence diagram shows the calls made in

AuthController.SignInResponse to authenticate the user with the relying party and to attach the encrypted forms ticket as a cookie if the authentication attempt was successful.

Authentication sequence diagram



- 1. **AuthController** calls the **GetResponse** method of its referenced **IOpenIdRelyingParty** implementation to get the authentication result from the authentication provider.
- 2. AuthController calls the GetOrCreateUser method of its referenced IUserServices implementation
- 3. **AuthController** calls the **SetAuthCookie** method of its referenced **IFormsAuthentication** implementation.
- 4. **AuthController** invokes its own **ReDirectToRoute** method and sends the user to its landing page after a successful authentication.

Let's take a closer look at the relying party implementation used in Mileage Stats.

DefaultOpenIdRelyingParty

DefaultOpenIdRelyingParty implements **IOpenIdRelyingParty**, which is a facade for the **OpenIdRelyingParty** class provided as part of DotNetOpenAuth that validates the user's credentials with OpenID. The code snippet below shows the **RedirectToProvider** method on the **DefaultOpenIdRelyingParty** class, which is responsible for redirecting the user to the authentication provider's login page:

```
C#

// contained in DefaultOpenIdRelyingParty.cs

publicActionResultRedirectToProvider(stringproviderUrJ

FetchRequestfetch)

{

IAuthenticationRequestauthenticationRequest =

this.relyingParty.CreateRequest(providerUrl,Realm.Auto
```

```
newUri(returnUrl));
authenticationRequest.AddExtension(fetch);
return
newOutgoingRequestActionResult(authenticationRequest.F
}
```

Forms Authentication Sliding Expiration

Sliding expiration of the forms authentication ticket in Mileage Stats is accomplished by resetting the expiration of the ticket every time the user makes a new request to the server. Normally this would be enabled by setting the **slidingExpiration** attribute to true on the forms security configuration in the web.config; however, since the ticket is being manually created and attached to the response in the **AuthController**, it needs to be refreshed manually. A custom handler for the **HttpApplication.PostAuthenticateRequest** event implements the sliding expiration for the forms ticket:

C#

```
// contained in Global.asax.cs
privatevoidPostAuthenticateRequestHandler(objectsender
{
    HttpCookieauthCookie =
    this.Context.Request.Cookies[FormsAuthentication.Forms
    if(IsValidAuthCookie(authCookie))
    {
      varformsAuthentication =
      ServiceLocator.Current.GetInstance<IFormsAuthenticatic
    varticket = formsAuthentication.Decrypt(authCookie.Val
    varmileageStatsIdentity =newMileageStatsIdentity(ticke
    this.Context.User =newGenericPrincipal(mileageStatsIde
    // Reset cookie for a sliding expiration.
    formsAuthentication.SetAuthCookie(this.Context, ticket</pre>
```

} }

The advantage of using a forms ticket with a sliding expiration is that it does not force the user to re-authenticate if he maintains a reasonable level of frequent activity in the application. Otherwise, the user would be redirected to authenticate after a fixed amount of time had elapsed after authenticating. While this greatly enhances the usability of the application, it is also a potential security risk since the user's authenticated session can be kept alive indefinitely by submitting requests to the server before the sliding expiration time on the forms ticket has passed. This can be mitigated by introducing an additional timeout value that does not slide, after which the ticket will expire regardless of user activity. While this approach is effective, it was not implemented in Mileage Stats because it would add complexity to the forms ticket handling that is beyond the scope of the application.

Input Validation

One of the key methods of preventing an application from accepting malicious content is validating any input before it is accepted by the application. While out-of-the-box ASP.NET input validation does a good job of preventing script or HTML injection, it is not always practical to use this mechanism in an ASP.NET MVC application. In Mileage Stats, this mechanism is disabled to handle input validation directly within the MVC model classes. If you don't implement your own input validation and rely on the built-in ASP.NET input validation instead, two things will happen: first, the input will not be validated until *after* the controller has processed it and before the view has rendered. Next, you will get the default "yellow screen of death" input validation page, which is not a pleasant user experience. When the out-of-the box ASP.NET input validation is disabled, you also lose HTML encoding on your input. To account for this, the *@* operator in Razor syntax automatically HTML-encodes output that is rendered in an MVC view.

Although input validation can be done on the client side to reduce round trips to the server, it must also be performed on the server since client-side validation can be bypassed by an attacker. One of the advantages of ASP.NET MVC is that it can be configured to render client-side validation based on server-side validation attributes defined on MVC model properties. This provides a single point in the application to define and maintain data validation rules. For example, in Mileage Stats, the _ProfileForm view is configured to use the **MileageStats.ServicesModel.User** class as its MVC model via the **@model** directive. If you look at the **DisplayName** property on the class, you will see attributes to limit the length of the value, require a value for the property, and use a custom text input validator that filters input using a regular expression:

C#

```
// contained in User.cs
[StringLength(15,
ErrorMessageResourceName ="UserDisplayNameStringLength
ErrorMessageResourceType =typeof(Resources))]
[TextLineInputValidator]
[Required(AllowEmptyStrings =false,
ErrorMessageResourceName ="UserDisplayNameRequired",
```

```
ErrorMessageResourceType =typeof(Resources))]
[Display(Name ="UserDisplayNameLabelText", ResourceTypublicstringDisplayName {get;set; }
```

Take a look at the custom input validator class behind the **TextLineInputValidator** attribute:

C#

```
//contained in TextLineValidatorAttribute.cs
public class TextLineInputValidatorAttribute : Regular
                                                 IClier
{
 public TextLineInputValidatorAttribute()
    : base(Resources.TextLineInputValidatorRegEx)
  {
    this.ErrorMessage = Resources.InvalidInputCharacte
 }
 public IEnumerable<ModelClientValidationRule>
   GetClientValidationRules(ModelMetadata metadata, (
  {
   var rule = new ModelClientValidationRule()
    {
      ErrorMessage = Resources.InvalidInputCharacter,
      ValidationType = "textlineinput"
    };
    rule.ValidationParameters.Add("pattern", Resources
    return new List<ModelClientValidationRule>() { rul
 }
}
```

The **TextLineInputValidatorAttribute** class uses a set of regular expressions it loads from the resources file. If any of the regular expression patterns are matched in the input, it fails validation. The regular expression pattern used to validate text input is **^(?!.*--)[A-Za-z0-9\.,'_ \-]*\$**. This limits the text assigned

to the property to alphanumeric characters and allows only a limited range of punctuation characters. Notice that the regular expression pattern is matching on known characters that are legitimate rather than excluding invalid characters. By default, everything else outside of what's been explicitly defined in the regular expression is not allowed in application data. The list that limits input by only allowing what is known to be valid is commonly referred to as a safe list. The advantage of safe lists is that anything that falls outside of the valid set of characters is not allowed; you don't have to worry about accidentally omitting a character from the regular expression that shouldn't be allowed in the application input.

What's nice about this is that the client-side validation for these properties is generated for you based on the server-side validation defined in the MVC model when the HTML for the MVC view is rendered. The following code shows you what the rendered client-side validation looks like for the user display name form field when you view the HTML source of the page.

HTML

```
<!-- rendered HTML in the client browser -->
<input data-val="true"
   data-val-length="Display name must be less than 15 c
   data-val-length-max="15" data-val-required="Display
   data-val-textlineinput="Only alpha-numeric character
   data-val-textlineinput-pattern="^(?!.*--)[A-Za-z0-9`\
   id="DisplayName"
   maxlength="15"
   name="DisplayName"
   type="text"
   value="Sample User" />
```

⊠Note:

If you want to accept HTML tags or other types of input that would normally be rejected by simple input validation, you can leverage the same mechanism to do it. However, you will need to create more sophisticated patterns to allow the input you want to accept while still excluding the input that you don't want submitted to your application.

Anti-Forgery

As previously explained in the Security Concepts section, Cross-Site Request Forgery (CSRF) is a security threat that has a high damage potential and should be protected against. ASP.NET MVC has a simple, yet effective mechanism for mitigating CSRF attacks. In your MVC view content, add an **@Html.AntiForgeryToken** directive to the view, as below:

CSHTML

Copy Code

```
<!-- contained in _ProfileForm.cshtml -->
@modelMileageStats.ServicesModel.User
@Html.AntiForgeryToken()
```

When the view HTML is rendered to the client, MVC places a unique value for the user in a hidden form field and in a session cookie. The form field in the rendered HTML might look like this:

HTML

```
<!-- rendered HTML in the client browser -->
<input name="___RequestVerificationToken"
   type="hidden"
   value="H4zpQFvPdmEdGCLsFgeByj0xg+B0DBjIMvtSl5anoNa0f
   zF0LxP5SzVR4cM9XZeV78IPi8K4ewkM3k2oFkplrXL4uoAqy+aoS
   h2jAwn3w0MwmhkcDQiJfJK7hGvN0jXA4d7S8x7rbLxp4Y8IJZS9v</pre>
```

There's another piece to implementing the anti-forgery tokens in ASP.NET MVC. You must also put a **ValidateAntiForgeryToken** attribute on corresponding MVC controller actions that you want to protect. Below is an example of the attribute decorating the edit action on the **ProfileController** class:

C#

Copy Code

```
// contained in ProfileController.cs
[HttpPost]
[ValidateInput(false)]
```

[ValidateAntiForgeryToken] publicActionResultEdit(UserupdatedUser)

When ASP.NET MVC checks for a request forgery, it verifies that the request verification token form field and cookies are present and that the values match each other. If either the cookie or the form field values are missing, or the values don't match, ASP.NET MVC does not process the action and returns an authorization failure instead.

☑ Note:

MVC anti-forgery tokens do not work using HTTP GET requests because the **_RequestVerificationToken** value from the client needs to be sent as a posted value. Therefore, it is important to make sure that you only accept client requests that use HTTP POST when you want to implement antiforgery tokens. This shouldn't be an issue because you should already be using only HTTP POST for data updates and using HTTP GET exclusively for read-only operations.

If you want to implement more tightly controlled anti-forgery measures so that the tokens will only validate to a particular set of MVC views and actions, you can pass a salt as a parameter to the **@Html.AntiForgeryToken** directive in the MVC model. Make sure that you are using the same salt values in the **@Html.AntiForgeryToken** directive in the MVC model and in the parameter for the **ValidateAntiForgeryToken** attribute on the MVC controller action; otherwise, the anti-forgery token will not validate properly when the MVC action is called. For more information about using a salt with an MVC antiforgery token, see the reference to "<u>Prevent Cross-Site Request Forgery (CSRF</u>) <u>using ASP.NET MVC'sAntiForgeryToken()helper</u>" at the end of the chapter.

JSON Hijacking Prevention

In some situations, it may be possible for an attacker to get at data via a JSON request using an attack that closely resembles a CSRF attack. If an attacker can get a user to click on a malicious link that makes a JSON request via HTTP GET and returns a JSON array, it may dump the contents of the array in the response, making it accessible to the attacker. The mitigation for JSON hijacking is fairly straightforward: either you can make sure to never return JSON arrays in a response, or you can restrict JSON requests to responding only to requests using the HTTP POST action. To configure a JSON action on an MVC controller to respond to requests only via HTTP POST, add the **HttpPost** attribute to it. The code snippet below shows the **HttpPost** attribute on the **ProfileController's JsonEdit** action.

C# Copy Code // contained in ProfileController.cs [HttpPost] [ValidateInput(false)] publicActionResultJsonEdit(UserupdatedUser)

This may not be an issue if you don't care whether or not someone can gain unauthorized access to data that is not sensitive; however, as a general practice you should protect your JSON calls against hijacking. For more information, see the reference to <u>JSON Hijacking</u> in the "Further Reading" section at the end of the chapter.

Additional Security Considerations

This section covers additional security considerations when deploying and configuring Mileage Stats. Securing communication between the client and the server is a key feature for protecting the data that goes back and forth between them. You may want to use a different method of authenticating users for the application, whether that means using a different relying party for the existing authentication mechanism or switching to an ASP.NET membership provider. If you expand the functionality of your application, you may want to restrict different levels of functionality to a limited subset of application users. In some cases, you may want to deploy the data tier to SQL Server or SQL Server Express instead of running it on SQL Server Compact Edition.

Securing Communication between Client and Server

When properly configured, Secure Sockets Layer (SSL) is a very effective mechanism to prevent eavesdropping, session hijacking, message replay and tampering between the client and the server. Mileage Stats does not use SSL out of the box because it depends on your infrastructure and not on the application itself. There's already a lot of documentation on how to set up SSL on your server, so the specifics won't be covered here. For more information, see the reference on "How to Set Up SSL on IIS 7" in the "Further Reading" section at the end of this chapter. Make sure that when you configure SSL on your server that you do **not** configure anonymous SSL. Otherwise, communication between the client and server will still be susceptible to several different attacks.

⊘Note:

Warning: When communicating with the server via SSL, a client verifies the identity of the server with which it is communicating by checking the host name provided in the SSL certificate against the URL of the server. Sometimes the client does not have a certificate that can be used to identify the server, and sometimes the server is configured to use SSL protocols that do not incorporate identification of the server; these are examples of anonymous SSL. Anonymous SSL can be configured on some web servers or by using an SSL certificate on the server that cannot be verified against the host's identity. While this provides protection against tampering and eavesdropping, it does not protect against message replay or spoofing attacks where one party can pose as the other during communication.

Once SSL has been set up in your environment, there are two changes to the web.config that you'll need to make in Mileage Stats. The first configuration change is to require ASP.NET forms authentication to use SSL.

The **requireSSL** attribute needs to be added to the forms element:

XML

```
<!--contained in web.config-->
<authenticationmode="Forms">
<formsloginUrl="~/Auth/SignIn"timeout="20"requireSSL='
</authentication>
```

The default timeout for a forms authentication ticket is 20 minutes, though you should feel free to adjust this to a reasonable time to achieve the usability of the sliding expiration while minimizing the security risk of indefinitely authenticated users.

The second configuration change is for OpenID to require an SSL communication with the relying party.

XML	Copy Code

```
<!--contained in web.config-->
<openid>
<relyingParty>
<securityrequireSsl="true"/>
```

The value of **requireSsl** needs to be set to true. That way you can ensure that the authentication provider will only attempt to communicate with your application via HTTPS.

Any use of the **UrlHelper.Action** method specifying a protocol (HTTP) in your application for constructing URLs will need to be updated to HTTPS for any URL references that are now secured via SSL. For more information on the **UrlHelper.Action** method, see the reference on the **UrlHelper.Action** method at the end of the chapter.

Protecting Connection Strings

If you change data sources for the application, you will likely end up changing the connection string in the web.config. As a good practice, you should protect connection strings in the web.config from disclosure. For more information on protecting connection strings, see the reference on <u>Protecting Connection</u> <u>Information (ADO.NET)</u> at the end of the chapter.

Deploying to a Shared Environment

In situations where you are deploying your web application to a server that is being shared by multiple parties, you may want to protect your application from others that are hosted on the server. In addition to protecting sensitive information in the web.config file, ensure that the application pools on the server are configured to run per web application and that the temporary ASP.NET files cached on the server are in a location that is not shared with the other web applications on the server. For more information, see "<u>How To:</u> <u>Secure an ASP.NET Application on a Shared Server</u>" on MSDN®.

Summary

This chapter provided an overview of security threats that impact Mileage Stats: unauthorized access, malicious input (content injection and cross-site scripting), eavesdropping, message tampering, message replay, and cross-site request forgery and how they are mitigated through various security features. Also, a few ideas were provided on how to extend or change security for Mileage Stats to accommodate different deployment environments and security requirements.

Further Reading

"ASP.NET Web Application Security" on MSDN: <u>http://msdn.microsoft.com/en-us/library/330a99hc.aspx</u>

"How to set up SSL on IIS 7:" http://learn.iis.net/page.aspx/144/how-to-set-up-ssl-on-iis-7/

"UrlHelper.Action Method" on MSDN: http://msdn.microsoft.com/en-us/library/dd505232.aspx

"Understanding the Forms Authentication Ticket and Cookie" on Microsoft Support:

http://support.microsoft.com/kb/910443

"Authenticating Users with Forms Authentication" on ASP.NET: <u>http://www.asp.net/mvc/tutorials/authenticating-users-with-forms-authentication-cs</u>

Protecting Connection Information (ADO.NET) on MSDN: http://msdn.microsoft.com/en-us/library/89211k9b(v=VS.100).aspx

"How To: Secure an ASP.NET Application on a Shared Server" on MSDN: <u>http://msdn.microsoft.com/en-us/library/ms228096.aspx</u>

"A Guide to Claims–based Identity and Access Control" on MSDN: <u>http://msdn.microsoft.com/en-us/library/ff423674.aspx</u>

"OWASP Top 10 Project" on OWASP.org: https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

"JSON Hijacking" on Phil Haack's blog: <u>http://haacked.com/archive/2009/06/25/json-hijacking.aspx</u>

"Best Practices for ASP.NET MVC" on the ASP.NET and Web Tools Developer Content Team's blog:

http://blogs.msdn.com/b/aspnetue/archive/2010/09/17/second_2d00_post.aspx

"Prevent Cross-Site Request Forgery (CSRF) using ASP.NET MVC's AntiForgeryToken() helper" on Steve Sanderson's blog: http://blog.stevensanderson.com/2008/09/01/prevent-cross-site-request-forgerycsrf-using-aspnet-mvcs-antiforgerytoken-helper/

| Community

To report documentation errors or provide feedback on this documentation, please send email to pagdoc@microsoft.com

Copyright © 2011 by Microsoft Corporation. All rights reserved.

Silk 1.0 - June 2011 **Unit Testing Web Applications**

Introduction

Unit testing, sometimes referred to as developer testing, focuses on testing small pieces of code a developer is writing, such as a class. These tests are critical for developers to ensure the pieces they build work as expected and should operate correctly when put together with other parts of the application. This helps support management of the application over time by giving you the confidence that changes you make don't inadvertently affect other parts of the system.

This chapter shows you how to get started unit testing JavaScript as well as server-side code, but does not cover all aspect of unit testing. References to more detailed discussions about unit testing can be found at the end of this chapter.

This chapter does not cover other important aspects of testing, such as performance, stress, security, automation, deployment, localization, and globalization. Nor does it discuss other important aspects to consider when testing the client-side such as cross-browser compatibility or usability. However, these areas are important for you to consider when testing your web application.

What you will learn in this chapter

In this chapter you will discover:

- How to get started unit testing your JavaScript and ASP.NET MVC code.
- The arrange-act-assert unit test structure.
- Techniques to isolate your tests and components.
- What to think about when testing your jQuery UI widgets.

The technologies discussed in this chapter are QUnit to test your JavaScript and jQuery client-side code and xUnit and Moq to test your server-side, ASP.NET MVC code.

JavaScript Unit Testing

Testing the client-side portion of a web application offers different challenges for unit, functional, and performance testing than that of server-side testing. In addition to testing the structural layout of a page and basic application functionality, you may want to verify that animations properly execute, that a page with a large amount of JavaScript has no memory leaks, and that the application maintains its functional and performance expectations across multiple browsers.

As a developer, you will use JavaScript to handle user interface (UI) logic in your application to dynamically build the structure, enable or disable portions of your UI, or load data in the background. Portions of these pieces you build may rely on libraries you adopt, such as jQuery, or those that you build yourself. You want to be sure that each of these pieces—on their own—operate as you expect so that, when put into the overall application, work as you want.

Unit testing is a way to verify that individual pieces work as you expect them to and provides a way for you to verify that they continue to work as libraries or tools evolve. For example, you may build a jQuery UI widget to manage a piece of your UI. When the next version of jQuery comes out you can quickly and easily verify that your widget is still working by executing the unit tests using the new jQuery libraries.

While unit testing isn't hard, there is a learning curve for those unfamiliar with it. One common objection to adopting unit testing is the perceived extra time it takes to write unit tests. While it is true that it will take longer to build something with unit tests than without (after all, there is more code in the form of unit tests), what is often not reflected is the time it will save later in tracking down bugs or verifying that things still work after changes to the code or by upgrading to new versions of a library. For the uninitiated, it can also be difficult to determine what should be tested or how to approach testing for a particular behavior. Unit testing can be a complicated topic. This section seeks to provide you with the basics to get started. It will give you an idea of what you want to test and provide some approaches to solving common challenges in unit testing JavaScript in your application.

Getting Started With Unit Testing

The Project Silk team decided to use QUnit for unit testing their JavaScript components since they heavily rely on jQuery and the jQuery UI widget framework, which also use QUnit . The QUnit unit testing framework can be located on the jQuery website at <u>http://docs.jquery.com/QUnit</u>. The site provides examples, documentation, and links to the download.

Setting up QUnit typically involves creating an HTML page with specific QUnit elements with certain class attributes specified, and including the **qunit.js** and **qunit.css** files. In Mileage Stats, these were added to the tests folder under the Scripts folder.

QUnit Files In Project

4	🗁 Scripts
	Debug
	🔺 🧁 tests
	斗 jquery.simulate.js
	斗 mstats.charts.tests.js
	斗 mstats.data.tests.js
	😂 mstats.fillups.tests.js
	斗 mstats.header.tests.js
	斗 mstats.imminent-reminders.tests.js
	斗 mstats.info-pane.tests.js
	😂 mstats.pinnedsite.tests.js
	斗 mstats.pubsub.tests.js
	mstats.registration.tests.js
	😂 mstats.reminders.tests.js
	mstats.statistics.tests.js
	😂 mstats.status.tests.js
	😂 mstats.summary.tests.js
	😂 mstats.tile.tests.js
	ല്ല്ല് mstats.utils.tests.js
	a mstats.vehicle-details.tests.js
	a mstats.vehicle-dropdown-monitor.tests.js
	a
	a mstats.vehicle.tests.js
	a 🛃 qunit.css
	👜 qunit.js
	a test-utilities.js
	a tests.htm
	excanvas.min.js
	AR ignlot canvasAvisTickRenderer min is

Once this is in place you will create a test JavaScript file for each set of tests

you want to run. This set is typically focused around a JavaScript object. For example, in Mileage Stats there is a JavaScript test file for each of the jQuery UI widgets that the application implements.

Each of these JavaScript test files and the JavaScript file of the item being tested are referenced from test.htm file so the QUnit framework can locate and execute the tests.

HTML

Copy Code

```
<!-- contained in test.htm -->
<!-- Code under test -->
<scriptsrc="../Debug/mstats.utils.js"></script>
<scriptsrc="../Debug/mstats.events.js"></script>
<scriptsrc="../Debug/mstats.events.js"></script>
...
<!-- Unit tests -->
<scriptsrc="mstats.utils.tests.js"></script>
<scriptsrc="mstats.utils.tests.js"></script>
...
<!-- Unit tests -->
<scriptsrc="mstats.utils.tests.js"></script>
...
</scriptsrc="mstats.utils.tests.js"></script>
...
</scriptsrc="mstats.utils.tests.js">...
</scriptsrc="mstats.utils.tests.js"</scriptsrc="mstats.utils">...
</s
```

These unit tests can be run by viewing the test HTML file in a browser. From Visual Studio, you can right click the test HTML file and select **View in Browser**. For the Mileage Stats tests the output would look like this while executing the tests.

QUnit Test Run Output

,O - C × B Mileage Stats Widget Testing ×

QUnit - Widget Testing

Hide passed tests

Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; WOW64; Trident/5.0; SLCC2; .NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR 3.0.30729; Met CLR

1. MileageStats Vehicle Details Widget: When created, then adds mstats-vehicle-details class to attached element (0, 1, 1)

- 2. MileageStats Vehicle Details Widget: When created, then adds mstats-vehicle-details-content class to attached element (0, 1, 1)
- 3. MileageStats Vehicle Details Widget: while loading data, then the widget ensures the contents are hidden (0, 1, 1)
- 4. Running MileageStats Vehicle Details Widget: when loading data is complete, then the contents are shown
- 5. Running MileageStats Vehicle Details Widget: when loading data errors out, then the widget ensures the contents are hidden
- 6. Running MileageStats Vehicle Details Widget: when loading data errors out, then triggers error status
- 7. Running MileageStats Vehicle Details Widget: when selected vehicle is set, then widget calls getData with dataUrl and selected vehicle id

Creating Unit Tests

There are typically multiple unit tests in one file and they are often grouped around a particular topic or type of test. Using QUnit, the **module** function is used to denote a group of tests.

```
JavaScript
```

Copy Code

```
module('Test Group');
test('Test one', function () {
    // Test logic goes here
});
test('Test two', function () {
    // Test logic goes here});
```

Let's look at the structure of a typical test. This is a test from the **mstats.data.test.js** to test a data caching component within the solution.

JavaScript

```
// contained in mstats.data.test.js
test('When data is saved, then it can be retrieved',fu
expect(1);
    // Arrange
varvalue ='some-data';
    // Act
mstats.dataStore.set(' /some/url', value);
    // Assert
    equal(
mstats.dataStore.get(' /some/url'),
value,
'mstats.datastoresaved and returned'+ value);
});
```

☑ Note:

The **test**, **expect**, **equal**, **start** and **stop** methods are specific to qUnit.

The basic structure of most unit tests follow an arrange-act-assert pattern, where the setup for the test is clearly grouped together, the action that should be tested is executed, followed by the verification of the desired results.

In the above example, the 'act' ensures the value can be set in the store and 'assert' verifies that the value was appropriately set. The QUnit framework provides a number of functions to help with assertions. The **equal** assertion is shown in the example, but **ok** (which just performs a Boolean check) is also typically used.

You'll see we execute a single assertion in this test and keep the number of assertions small, which typically results in smaller, more focused unit tests. Writing unit tests in this manner encourages you to write code that is also small and focused. This tends to lead to code that is more composable because then it will have a single responsibility. They should generally take one action and make one assertion. However, there are cases where a group of similar assertions will be made, such as when verifying the property values on a deserialized JSON object.

The QUnit framework provides a mechanism to ensure that the proper number of expected assertions were run using the **expect** function. At the beginning of the test you will see that **expect(1)** was called to let QUnit know how many assertions should be run. If QUnit does not encounter that number of assertions, then it will produce an error in its output when the tests are run.

What to Test

Now that you know how to write a unit test, perhaps the more important question is *what* should be tested? Generally in a unit test you are trying to verify the functionality of a relatively small component, this could be a JavaScript object or something like a jQuery UI widget. Each test verifies independent pieces such as verifying a calculation happened correctly or that the proper DOM modification occurred.

When testing UI widgets, it can be unclear what should be tested. The basic rule of thumb is to test anything a designer would not change. Logic that drives the UI might be tested, such as the right navigation was invoked, an element had the proper class attribute applied (or removed), or that the right event was raised. But, you would not test that a specific font value was set or the specific background color of an element.

Isolating Your Tests

Often your object under test will rely on other objects, functions, or libraries. You may have an object that makes Ajax calls to retrieve data. If you attempt to make Ajax calls when running the unit tests you might get unpredictable results because the server responding to the calls may be unavailable when you run your tests. Generally, you want to isolate your component from these types of problems.

You will also want to isolate your component from other objects you build within your system when testing. In Mileage Stats, many jQuery UI widgets rely on a publish-subscribe object for communication. During the testing of objects with dependencies, you do not want to invoke the *actual* dependencies. If you did, you would be testing more than one thing at a time. Instead, it is important to test that the component *attempts* to invoke a dependency. The typical strategy for isolating your component under test in these scenarios is to supply an alternative component or function that the component calls instead of the real component during tests. These alternatives may also be referred to fakes, doubles, stubs or mocks. As it turns out, the ability to isolate your component in this manner also helps the overall design of your application by tending to create smaller, more focused components.

With a substitute object employed, you can then verify that the correct calls were made with the right values. For example, when testing that the Mileage Stats data cache component makes an Ajax call with the appropriate URL an alternate jQuery Ajax function is supplied for testing. In this alternate, we verify that the expected URL is invoked by the component.

JavaScript

```
// contained in mstats.data.tests.jstest('whensendRequ
expect(1);
    // Arrange
$.ajax=function(options) {
        // Assert
equal(options.url,'/url','Urlwas properly set');
};
    // Act
```

```
mstats.dataManager.sendRequest({
url:'/url'
});
});
```

Note also that this somewhat changes the typical arrange-act-assert order of the test structure because the assertion is in the supplied Ajax function. This is why it is important to use the **expect** function at the beginning of your tests to help ensure that all the expected assertions are made.

When providing these alternative functions or components, it is also a good idea to capture and restore the original values to avoid interfering with any other test that may have relied on these values. In QUnit this can be done when defining the test module where the **setup** and **teardown** functions can be supplied.

JavaScript

Copy Code

```
// contained in mstats.data.tests.jsmodule(
'MileageStatsDataManagersendRequestTests',
{
  setup:function() {
  this.savedAjax= $.ajax;
  ...
},
  teardown:function() {
  $.ajax=this.savedAjax;
  ...
}
}
```

jQuery UI Widget Testing

When unit testing jQuery UI widgets there are some additional considerations. Since widgets are attached to a DOM element, you will need to create these elements either in the test or, if they are more complicated, in the setup for a module. In Mileage Stats, since many of the widgets interact with a section of the DOM, some of that structure needs to be created during test setup. For example, the header widget test creates the structure over the DOM it manipulates in the setup for the test.

JavaScript

```
// contained in mstats.header.tests.jsmodule('Header W
setup:function() {
$('#qunit-fixture').append(
'<div class="header" id="header">'+
'<div><div><h1>Dashboard</h1>'+
'<div id="notification"></div>'+
'<div class="nav">'+
'<span id="welcome">Welcome <b>Sample User</b></span>'
'[ <a id="dashboard-link" href="/Dashboard">Dashboard<
'| <a id="charts-link" href="/Chart/List">Charts</a>'+
'| <a id="profile-link" href="/Profile/Edit">Profile<//profile</pre>
'| <a id="login-link" href="/Auth/SignOut">Sign Out</ε
'</div>'+
'</div>'+
'</div>'
);
}
});
```

In QUnit, you add these elements to the element with the ID of 'qunit-fixture'. You should only add the minimal amount of structure needed to appropriately simulate your test needs as this will make the structural dependencies of the test clearer.

When testing jQuery UI widgets you will also often need to supply alternate implementation on dependent functions or objects. Since you don't control the

creation of the jQuery UI widgets directly, you will typically do this as part of the options object passed into the widget (see the section jQuery UI widgets for more details about the use of an option object). For example, when testing the Mileage Stats vehicle-details widget an alternative implementation for the Ajax method and the event publisher are supplied as part of the options.

JavaScript

```
// contained in mstats.vehicle-details.tests.jstest('v
expect(3);
vareventType='loadError',
details = $('#details-pane').vehicleDetails({
templateId:'#testTemplate',
getData:function(options) {options.error({}); },
eventPublisher:function(event, status) {
if(status.type==eventType) {
ok(status,'status object passed to publisher');
equal(status.type, eventType,'status is of type :'+ ev
equal(status.origin,'vehicleDetails','status has corre
}
}
});
// force a data refresh
details.vehicleDetails('option','selectedVehicleId', 1
});
```

Server-Side Unit Testing

Unit testing code on the server typically involves many more interactive pieces than what you encounter when testing client-side JavaScript. In an ASP.NET MVC application, controllers will interact with services or repositories to handle each request. These interactions and expectations can be tested using unit tests as each piece is built to instill confidence that the system will continue to work as new features are added or new versions of dependent libraries are supplied.

This section is intended to provide you with enough information to get started unit testing your server side application. Since each application is different, testing all scenarios is out of scope for this chapter. To find out more about unit testing your applications, see the "Further Reading" section.

Getting Started Unit Testing

There are a number of unit testing frameworks to choose from when unit testing server-side .NET components. Most unit tests frameworks are similar and any one of them can be a reasonable choice. Microsoft offers two technologies that can be used for writing unit tests: Microsoft Test and xUnit. Microsoft Test is supplied with certain versions of Visual Studio and xUnit.net is a Microsoft developed, open-sourced unit testing framework available on Codeplex or NuGet.

Regardless of your unit test framework choice, unit tests are placed in a separate assembly that the unit test framework can discover and use to execute the tests. A typical Visual Studio solution organization has the unit test projects included in the solution with the projects under test. For example, the Mileage Stats solution has its test projects in a Unit Tests solution folder.

Unit test location in MileageStats project

Solution 'MileageStats' (9 projects)

- BusinessLogic
- DataTier
- Script Documents
- Solution Items
- 4 📄 Unit Tests
 - MileageStats.Data.SqlCe.Tests
 - MileageStats.Services.Tests
 - MileageStats.ServicesModel.Tests
 - MileageStats.Web.Tests
- MileageStats.Web

There is one unit test project for the Services, ServicesModel, SqlCe, and Web. Some projects don't have a corresponding unit tests project primarily because these projects contain only shared interfaces and data transfer classes that do not have significant logic to test.

The team choose xUnit as the unit testing framework for the Mileage Stats project. While you can accomplish unit testing with either Microsoft Test or xUnit, the team felt that since xUnit was built specifically for developer unit testing that it would better fit their needs. The remainder of this section

discusses unit testing using examples in xUnit, but you can readily apply the same approaches with Microsoft Test, although some of the exact mechanics may be different.

To create a new unit test project, you add a C# or Visual Basic Class Library project and reference the xUnit assemblies. In the test project, there will be a class to contain all the related tests for a particular component. For example, the **MileageStats.Web.Tests** project contains a test class for each controller in the Web project. They generally are named the same as the controller name with the term 'Fixture' appended.

To write a test, create a method with the attribute **Fact** specified. The xUnit.net framework searches for these attributes and executes these methods. Each test should follow the arrange-act-assert pattern. In this pattern all the setup for the test (e.g. arrangement) is done first, then the action to be tested is executed (e.g. act), and then the validation is done (e.g. assert).

C#

```
//contained in ReminderFixture.cs[Fact]
publicvoidWhenReminderIsNotOverdue_ThenIsOverdueReturr
ł
// Arrange
varreminder = newReminderFormModel()
{
Title ="future reminder",
DueDate=DateTime.UtcNow.AddDays(2),
DueDistance= 10000
};
reminder.UpdateLastVehicleOdometer(10);
// Act
boolisOverdue=reminder.IsOverdue;
// Assert
Assert.False(isOverdue);
}
```

Generally tests should be small and focused, with only one or two asserts. At times, there may be more asserts in a single test if they are verifying in a logical group.

In addition to **Assert.False**, xUnit.net supplies a number of other built-in asserts available on the **Assert** static class.

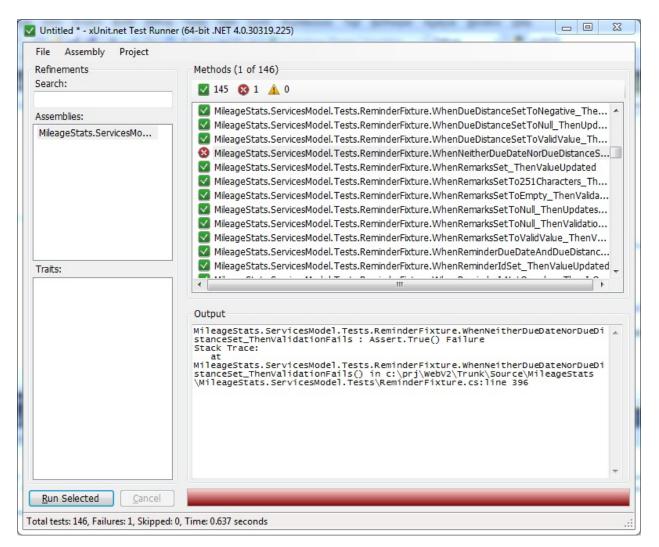
Once the tests have been built you can execute them using the xUnit.net test runner to see that they pass. The test runner can be found where you unpackaged the xUnit.net contents retrieved from Codeplex. After you add the test assembly into the runner you can run all the tests to see if they succeed.

Running Unit Tests

File Assembly Project	
Refinements	Methods (149)
Search:	🔽 149 😵 0 🛕 0
Assemblies: MileageStats.ServicesMo	 MileageStats.Services.Tests.Validators.TextMultilineInputValidatorAttributeFixture.WhenT MileageStats.Services.Tests.Validators.TextMultilineInputValidatorAttributeFixture.WhenT MileageStats.Services.Tests.Validators.TextMultilineInputValidatorAttributeFixture.WhenT MileageStats.Services.Tests.Validators.TextMultilineInputValidatorAttributeFixture.WhenT MileageStats.Services.Tests.Validators.TextMultilineInputValidatorAttributeFixture.WhenT MileageStats.Services.Tests.Validators.TextMultilineInputValidatorAttributeFixture.WhenT MileageStats.ServicesModel.Tests.FleetStatisticsFixture.WhenConstructed_ThenCalculates
Traits:	MileageStats.ServicesModel.Tests.FleetStatisticsFixture.WhenConstructed_ThenCalculates MileageStats.ServicesModel.Tests.FleetStatisticsFixture.WhenConstructed_ThenCalculates Output
Run All Cancel	

If there is a failing test it will show up in the console.

Running Unit Tests With Errors



Alternatively, you can run tests by using TestDriven.Net or Resharper which would run the tests from within Visual Studio. For more details on setting this up, see <u>http://xunit.codeplex.com</u>.

What to Test

On the server side, you should create unit tests for any classes and components that contain logic or must interact with other components. You should not write unit tests for generated code or code you don't own. The team wrote unit tests for classes in each of the major layers.

Repository Layer. The repository layer provides the basic persistence for information throughout the system. In Mileage Stats, this relies heavily on Entity Framework Code-First and SQL Server Compact Edition. Much of the tests written against this layer verifies that the persistence and retrieval implementations for the various repositories produce the correct results. These tests, since they are writing to an actual database, cannot strictly be considered unit tests, but are useful in verifying that the persistence mechanism for all the models occurs as expected. Often, these are referred to as integration tests.

These tests were also useful because the Entity Framework Code-First library was adopted before actual release, so these tests helped demonstrate that the expectations around Entity Framework were maintained between releases.

Business Logic Layer. The business services layer is invoked by the controllers in the Web Layer in order to execute business rules and store data in the repository. Unit tests for the business services layer focus on verifying the business rules and its interaction with the repository layer. The tests do not actually store data in the repository but use a fake repository and verify that the business services layer uses it correctly. The models in this layer are what the web application retrieves from and supplies to the business services layer. These models often contain validation logic that are verified in unit tests.

Web Layer. The actual controllers that respond to requests have unit tests to verify that they interact with the services and models appropriately and return correctly built View Models for the Views or jQuery template.

Isolating Your Components

It is common for the classes you are testing to rely on other classes. For example, a class may rely on a repository to persist a model. During testing you want to isolate your class' interaction with these other objects to ensure that only the behavior of the class in question is tested. Additionally, it can sometimes be painful to setup these other classes appropriately. For example, if the class calls a web service it would be difficult to expect that the web service be up when you want to run your unit test.

Instead of trying to create the actual context for the class under test we supply it with alternative implementations of the object it depends on. These alternatives may also be called fakes, doubles, stubs or mocks. Using these alternative implementations has the side-effect of also helping separate the responsibilities of our classes.

To provide this separation, instead of creating a class that depends on a specific technology we provide an abstraction for the class to depend on. This allows us to provide different implementations of the dependency at different times, such as at unit test time. Often this abstraction could be an interface definition but it could also be a base or abstract class.

For example, suppose we had a class to test that needed to store values somewhere. Instead of tying the class directly to a specific store implementation, it can depend on an **IStore** abstraction.

```
C# Copy Code
publicinterfaceIStore
{
voidPersist(stringitem);
}
publicclassClassToTest
{
privateIStorestore;
publicClassToTest(IStorestore)
{
}
```

```
this.store= store;
}
publicvoidSave(stringvalue)
{
...
store.Persist(value);
...
}
}
```

When we write a test for this class that depends on **IStore**, we can then provide an alternative implementation.

C# Copy Code

```
[Fact]
public voidWhenSaving_ThenSendsValueToStore()
{
varmockStore= newStoreMock();
varclassToTest= newClassToTest(mockStore);
classToTest.Save("Test");
Assert.Equal(mockStore.ValueSaved, "Test");
}
private classStoreMock:IStore
{
public stringValueSaved{ get; set; }
public void Persist(string item)
{
ValueSaved= item;
}
```

The StoreMock captures the saved item to verify that ClassToTest sends the

correct value to the store. Instead of making these mocks by hand, as shown above, the team relied on Moq—a mocking framework—when writing Mileage Stats tests. This allows us to supply mock objects without requiring us to create mock classes by hand. The same test above would look like this using Moq.

```
[Fact]
publicvoidWhenSaving_ThenSendsValueToStore()
{
varmockStore=newMock<IStore>();
varclassToTest=newClassToTest(mockStore.Object);
classToTest.Save("Test");
mockStore.Verify(s =>s.Persist("Test"));
}
```

C#

C#

Moq dynamically builds the objects needed for testing and in the case of the **Verify** method, can automatically verify that methods or properties were called with the correct values. See the Moq Codeplex site for more information about using Moq at <u>http://moq.codeplex.com</u>.

There are times when you don't control the class that you want to be able to mock. For instance, if you use a static class built into the .NET Framework library like **FormsAuthentication**. In these cases, you will often create an interface for just the functionality you use and provide a default implementation for run-time and a mock implementation at test-time. This was the approach employed with Mileage Stats when using the DotNetOpenAuth library. This library helps with implementing the various authentication protocols used in Mileage Stats. To isolate the components and make them more testable, the **IOpenIdRelyingParty** interface was created.

Copy Code

Copy Code

```
// IOpenIdRelyingParty.cs
publicinterfaceIOpenIdRelyingParty
{
ActionResultRedirectToProvider(
```

```
stringproviderUrl,
stringreturnUrl,
FetchRequestfetch);
IAuthenticationResponseGetResponse();
}
```

Mileage Stats has a default implementation that uses the real DotNetOpenAuth library at run-time and a mock implementation when testing the **AuthController**.

At run-time, all these pieces are connected using a technique known as dependency injection. See the "Dependency Injection" section in Chapter X to better understand how this works.

Testing your ASP.NET MVC Controllers

ASP.NET MVC was designed to support the testability of the controllers, filters, and actions that developers typically write when developing an MVC application. Since each controller is responsible for handling a request and MVC automatically maps input from the query string or from form data to the data types on your controller's methods, you can easily write tests for your controllers and simply supply them with the necessary inputs. For instance, the ReminderController's **Add** method takes an integer value for the vehicle identifier and **Reminder** object.

C#

```
// contained in ReminderController.cs
publicActionResultAdd(intvehicleId,ReminderFormModelre
{
...
returnView(viewModel);
}
```

In a unit test, it is very simple to just provide these values for testing. The example below demonstrates supplying the reminder and vehicle ID directly in the test.

```
C#
```

```
// contained in ReminderControllerFixture.cs[Fact]
publicvoidWhenAddReminderWithValidReminder_ThenReturns
{
    ...
varresult =(RedirectToRouteResult)controller.Add(vehic
;
    Assert.NotNull(result);
Assert.Equal("Details", result.RouteValues["action"]);
Assert.Equal("Reminder", result.RouteValues["controller
}
```

While many unit tests for controllers can use this approach, there are still cases where the controllers require access to the **HttpContext**. Usually providing alternate implementations of **HttpContext** is very difficult, thus making certain scenarios very hard to test. But since the MVC base controller class **Controller** relies on **HttpContextBase** instead of **HttpContext**, it can much more easily be substituted. Mileage Stats uses this on many of its controller tests to ensure the **User** property is set appropriately on **HttpContextBase**.

To do this, the Mileage Stats uses an **MvcMockHelpers** class that wraps the building of a Moq object that substitute **HttpContext** information. This controller context is then set on the controller under test by calling the static **SetFakeControllerContext** method in the **MvcMockHelpers** class. The **RemindersControllerFixture** sets this when it builds a testable controller.

```
C#
// contained in RemindersControllerFixture.cs
privateReminderControllerGetTestableReminderController
{
 varreminderController =newReminderController(this.mock
this.mockBusinessServices.Object);
reminderController.SetFakeControllerContext();
reminderController.SetUserIdentity(
newMileageStatsIdentity(this.DefaultUser.Authorization
this.DefaultUser.DisplayName,
this.DefaultUser.UserId));
returnreminderController;
}
```

The fake context creates a series of Moq objects that the controller will interact with under test. If you want to adjust what they're doing you can recover the mock and change its behavior. The static **SetUserIdentity** above does this for a controller to set an identity context for the test into the **HttpContext**.

C#

```
// contained in ControllerMockHelpers
publicstaticvoidSetUserIdentity(thisControllercontrol]
{
```

```
Mock.Get(controller.HttpContext)
.Setup(x =>x.User)
.Returns(newGenericPrincipal(identity,null));
}
```

The types of tests you typically write around your controller include:

- **View Models**. You will want to test that the controller provides the correct model data for a specific view.
- **Navigation**. You will want to test that the controller will provide the correct redirection when it is finished processing the request or when there is an error processing the request.
- **Interaction**. You will want to test that the controller makes the appropriate calls to your repository or services layers (which will be mocked in the tests). You will also want to test that your controller appropriately handles the situation when the model data supplied to a controller is in an invalid state.
- **JSON Endpoints**. If you have JSON data endpoints you want to make sure these return appropriate JSON results for the call.

Summary

You should make a conscious decision about whether or not you are going to unit test your code. Unit testing is not hard but it does require an investment of time to learn and apply. The time spent initially writing unit tests will save time over the life of your project and deliver better quality code. There are frameworks that help you write unit tests available for most languages and platforms. Visual Studio includes unit test support for C# and Visual Basic .NET, among other languages, and you can readily find them for languages such as JavaScript.

Further Reading

Meszaros, Gerard. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, 2007.

Cem Kaner, Jack Falk, and Hung Q. Nguyen. *Testing Computer Software*, 2nd Edition. Wiley, 1999.

patterns & practices Acceptance Test Engineering Guidance on CodePlex: http://testingguidance.codeplex.com/

Performance Testing Guidance for Web on MSDN: http://msdn.microsoft.com/en-us/library/bb924375.aspx

Guidance for Build, Deploy and Test Workflows on MSDN: <u>http://msdn.microsoft.com/en-us/library/ff972305.aspx</u>

QUnit unit testing framework on the jQuery website: <u>http://docs.jquery.com/QUnit</u>

xUnit.net on CodePlex: <u>http://xunit.codeplex.com</u>.

Moq on CodePlex: <u>http://moq.codeplex.com</u>

"Testing the User Interface with Automated UI Tests" on MSDN: <u>http://msdn.microsoft.com/en-us/library/dd286726.aspx</u>

"How to: Create a Coded UI Test" on MSDN: <u>http://msdn.microsoft.com/en-us/library/dd286681.aspx</u>

Resharper: http://www.jetbrains.com/resharper/

TestDriven.NET: http://testdriven.net

| Community

To report documentation errors or provide feedback on this documentation, please send email to pagdoc@microsoft.com

Copyright $\ensuremath{\mathbb{C}}$ 2011 by Microsoft Corporation. All rights reserved.

```
Silk 1.0 - June 2011
Widget QuickStart
```

This Widget QuickStart illustrates the way Project Silk uses the <u>jQuery UI</u> <u>Widget Factory</u> to create maintainable widgets that implement client-side behavior.

Business Scenario

Our team has been asked to enable cross-browser keyword lookup capabilities in our web pages by hyperlinking select keywords to popular websites. This feature will need to be added dynamically to all company web pages.

Another team has been tasked with tagging the keywords in the web pages. The words will be tagged dynamically, based on server-side business logic driven by agreements with third parties.

The focus of this QuickStart is to enable the client-side behavior for the tagged keywords. When a user hovers over a keyword, the browser will display a popup list of popular links for that keyword from the Delicious.com bookmarking service.

Walkthrough

To interact with the completed scenario, ensure you have an Internet connection and follow the steps below:

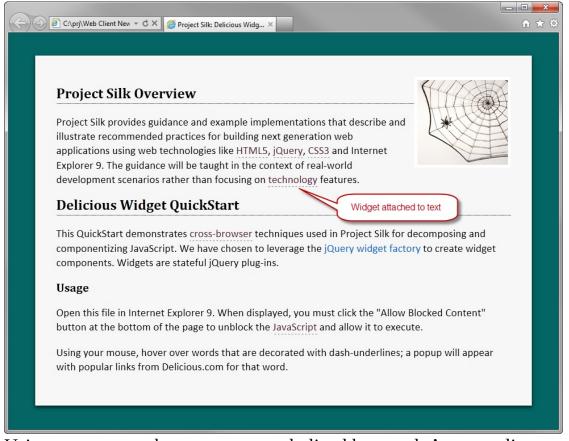
1. Open the **default.htm** file using Windows Internet Explorer 9. After the file's content is displayed, you'll need to click on the **Allow blocked content** button at the bottom of the browser window to enable scripts to run. Blocking active content by default is a security feature of Internet Explorer 9.

Widget QuickStart (default.htm)

Project Silk Overv	iew	
	nce and example implementations that descri	be and
	ractices for building next generation web	
	hnologies like HTML5, jQuery, CSS3 and Interr vill be taught in the context of real-world	let
	ther than focusing on technology features.	
	tes cross-browser techniques used in Project s We have chosen to leverage the jQuery widg	
Usage		
	xplorer 9. When displayed, you must click the e page to unblock the JavaScript and allow it t	
Using your mouse hover a	over words that are decorated with dash-unde	rlines: a nonun will annear

2. After allowing blocked content, you'll notice that the keywords are displayed in a new color and have been underlined with a dashed line, as pictured below.

Widget QuickStart after scripts are unblocked



- 3. Using your mouse, hover over an underlined keyword. A pop-up list with the top-10 most popular links for that keyword will be displayed. Notice that the keyword has been repeated in the title of the pop-up list.
 - a. One second after moving your mouse away from the keyword, the pop-up list will close unless your mouse is within the boundaries of the pop-up list.
 - b. If the keyword is on the left side of the page, the pop-up list will open to the right of the cursor. If the keyword is on the right side of the page, the pop-up list will open to the left side of the cursor, as in the image below.

Pop-up list for the keyword "jQuery"

Project Silk Overview



Move your mouse over the pop-up list. You can now click on a link, which will open in a new browser window.
 Links from Delicious.com in the pop-up list

Project Silk Overview Project Silk provides guidance and example implementations that describe and illustrate recommended practices for building next generation web applications using web technologies like HTML5, jQuery, CSS3 and Internet Explorer 9. The guidance will be taught in the context of real-world development scena Popular Links for jQuery v features. Useful JavaScript and jQuery Tools, Libraries, P... Delicious Wid Animated Scroll to Top A Better Way To AJAX - jqueryahm.com Isotope This QuickStart dem Circular Discography Template with jQuery | C... sed in Project Silk for decomposing and defunkt/jquery-pjax - GitHub he jQuery widget factory to create widget componentizing Jav The jQuery Divide Slider Kit, sliding contents with jQuery components. Widge jStat : a JavaScript statistical library

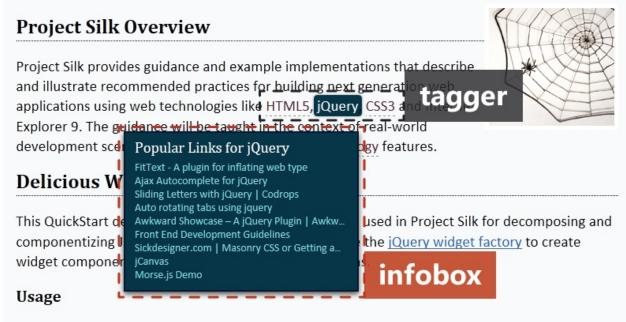
5. Moving your mouse outside the boundaries of the pop-up list will cause the pop-up list to close.



Conceptual View

This section illustrates the relationship of the jQuery UI widgets to the HTML page. A single **infobox** widget is attached to the page's **body** element. After it's attached, it creates a <div> element and dynamically adds it to the page's <body> element. Additionally, a **tagger** widget is attached to each keyword.

Relationship of the jQuery UI widgets to the HTML page



Open this file in Internet Explorer 9. When displayed, you must click the "Allow Blocked Content" button at the bottom of the page to unblock the JavaScript and the page to unblock the page to

Using your mouse, hover over words that are decorated with dash-underlines; a popup will appear with popular links from Delicious.com for that word.

The HTML below reveals a keyword tagging strategy that takes advantage of HTML5 data attributes. Each of the keywords has been wrapped in a **span** tag with the **data-tag** attribute applied. For this scenario, the keyword wrapping was accomplished on the server side.

HTML

```
<!-- Contained in default.htm -->
<!DOCTYPE html>
<html>
```

Attaching Widgets

Once created, the widget is attached to an HTML element and its **options** can be set.

JavaScript

```
// Contained in startup.js
(function ($) {
   var infobox = $('body').infobox({
      dataUrl: 'http://feeds.delicious.com/v2/json/p
   });
   $('span[data-tag]').tagger({
      activated: function (event, data) {
         infobox.infobox('displayTagLinks', event,
        },
        deactivated: function () {
            infobox.infobox('hideTagLinks');
        }
    });
} (jQuery));
```

The code above demonstrates the **infobox** widget being attached to the **body** element. The **dataUrloption** value will be used when performing popular keyword link lookups.

The jQuery selector **span[data-tag]** returns a jQuery wrapped set that contains all **span** tags with a **data-tag** attribute. A **tagger** widget will be attached to each of the **span** tags in the returned collection. The **tagger** widget has **activated** and **deactivated** options that are used as callbacks. These callbacks are used to handle events raised when the mouse hovers over the tag.

Widget Initialization

When a widget is created (attached), the jQuery UI widget factory will call the private method **_create**. This method provides the developer an opportunity to perform widget setup actions. Examples include building and injecting markup, adding CSS classes, binding events, and so forth.

JavaScript

Co

```
// Contained in jquery.qs.infobox.js
_create: function () {
   var that = this,
      name = that.name;
   that.infoboxElement = $('<div class="qs-infobox" /
   that.infoboxElement.appendTo('body')
   .bind('mouseenter.' + name, function () {
      mouseOverBox = true;
   })
   .bind('mouseleave.' + name, function () {
      mouseOverBox = false;
      that.hideTagLinks();
   });
},
```

The code snippet above first creates a variable for **this** called **that** within the closure, so the widget can be referenced within the **mouseenter** and **mouseleave** event handlers.

Recall that the **infobox** widget is attached to the **body** element. The element **div.qs-infobox** will contain the UI for this widget. It is stored in **that.infoboxElement**, attached to the **body** element, and bound to some events. The **name** variable holds the name of the widget and is appended to the name of the event it's binding to. This is a recommended practice when using jQuery; the reasons why will be explained later in the QuickStart.

⊠Note:

Note: Most of the time, widgets are attached to the element that they will control; however, there are times when a widget will need to create additional elements.

In the above **_create** function, the **infobox** widget creates a **div** to hold the list of links. The default.htm HTML page could have been modified to include the **div** in the first place, making it unnecessary for the widget to add an additional structure. However, the code was written this way to illustrate a widget adding UI elements to an existing HTML structure.

Widget Interactions

An interesting challenge in this scenario is giving the user enough time to click the links without showing the pop-up list longer than needed. The implementation requires coordination between the two widgets.

Mouse Entering a Keyword Span

When the mouse enters the keyword span, the **mouseenter** event handler in the **tagger** widget is invoked. The **name** being appended to the event name is the name of the widget and is used as a namespace for the event binding. This is a recommended practice. Any string can be used as the namespace, but using the name of the widget allows you to tap into a feature of the widget factory described later in the QuickStart.

JavaScript

Copy Code

```
// Contained in jquery.qs.tagger.js
.bind('mouseenter.' + name, function (event) {
    clearTimeout(timer);
    that._trigger('activated', event, {name: tag});
})
```

The **clearTimeout** call uses the **timer** variable, which is defined outside of the widget prototype and set in the handler for **mouseleave**, discussed in the next section. This means there will be only one timer created and shared among all instances of the **tagger** widget.

The next line raises the **tagactivated** event. It doesn't raise the **taggeractivated** event because the widget sets the **widgetEventPrefix** property, as shown in the next code snippet. It also doesn't raise the **activated** event as you may have suspected because the widget factory changes the name of raised events by prepending the name of the widget to the name of the event being triggered.

JavaScript

Copy Code

```
// Contained in jquery.qs.tagger.js
$.widget('qs.tagger', {
    widgetEventPrefix: 'tag',
    options: {
```

When the **tagactivated** event is raised, the **displayTagLinks** method is called

on the **infobox** widget. As you will notice from having a look at **jquery.qs.infobox.js**, it never binds to this event. Doing so would create a dependency between the widgets. A better option is to follow a recommended pattern and take advantage of a related jQuery UI feature. It is recommended that a widget provide callback options for all of the events it raises.

```
JavaScript
```

Copy Code

```
// Contained in jquery.qs.tagger.js
options: {
    activated: null,
    deactivated: null
},
```

The jQuery UI widget factory will automatically call any option with the same name as the event being raised. This feature allows the event handlers to be associated by setting the value of the option. The QuickStart does this in the startup file.

JavaScript

```
// Contained in startup.js
$('span[data-tag]').tagger({
    activated: function (event, data) {
        infobox.infobox('displayTagLinks', event, data
    },
    deactivated: function () {
        infobox.infobox('hideTagLinks');
    }
});
```

This approach is also a nice way to avoid having to know if the event is called **tagactivated** or **taggeractivated** or something else. The **displayTagLinks** method accepts a browser event and the name to look up. The first part of the method sets up enclosed variables to be used in the second part of the method.

JavaScript

```
// Contained in jquery.qs.infobox.js
displayTagLinks: function (event, tagName) {
    var i,
        html,
        that = this,
        options = that.options,
        elem = that.infoboxElement,
        top = event.pageY + offsetY,
        left = event.pageX + offsetX,
        url = options.dataUrl + tagName + '?count=' +
        displayResult = function () {
                elem.html(html);
                elem.css({top: top, left: left});
                elem.show();
            };
        if (event.pageX > window.screenWidth / 2) {
            left = event.pageX + leftSideAdjustment;
        }
```

After the closure is prepared, **left** is adjusted in case the tag is on the right-hand side of the page. The second part of the **displayTagLinks** method is an Ajax call to the **url**, constructed above, for the Delicious bookmarking service.

JavaScript

```
}
html += '';
} else {
    html = '<h1>Data Error</h1>[snipped]Ajax Error</h1>' +
        'The Ajax call returned the followir
        jqXHR.statusText + '.';
    displayResult();
});
```

The local **displayResult** function is scoped only to the **displayTagLinks** method since it was needed for both **success** and **error** conditions and nowhere else. This is the method that applies the result to the element for the user to see.

Mouse Leaving a Keyword Span

When the mouse leaves the tag's **span**, a similar coordination occurs. The **tagger** widget has a namespaced event bound to the span's **mouseleave** event.

```
// Contained in jquery.qs.tagger.js
.bind('mouseleave.' + name, function () {
    timer = setTimeout(function () {
        that._trigger('deactivated');
      }, hideAfter);
});
```

The **timer** is set to raise the **tagdeactivated** event after 1000 milliseconds, which is the value of **hideAfter**.

When the **tagger** widget was applied to the span elements, a function was supplied to the **deactivated** callback, as you also saw earlier in the QuickStart.

JavaScript

JavaScript

```
// Contained in startup.js
$('span[data-tag]').tagger({
    activated: function (event, data) {
        infobox.infobox('displayTagLinks', event, data
    },
    deactivated: function () {
        infobox.infobox('hideTagLinks');
    }
});
```

The function invokes the **hideTagLinks** method on the **infobox** widget.

JavaScript

Copy Code

Copy Code

```
// Contained in jquery.qs.infobox.js
hideTagLinks: function () {
    !mouseOverBox && this.infoboxElement.hide();
```

},

The **infobox** is only hidden if the mouse is not over it. Effectively, the 1000 ms delay provides the user time to move the mouse from the keywords to the links.

Mouse Entering the Pop-up List

Internally, the **infobox** widget uses the **mouseOverBox** variable to maintain state indicating whether or not the mouse is over the pop-up list. This variable is defined in the closure created by the self-executing anonymous function wrapping the file.

JavaScript

Copy Code

```
// Contained in jquery.qs.infobox.js
(function ($) {
    var offsetX = 20,
        offsetY = 20,
        mouseOverBox = false,
        leftSideAdjustment = -270;
    $.widget('qs.infobox', {
```

When the mouse enters the **infobox**, **mouseOverBox** is set to **true**.

```
JavaScript
```

Copy Code

```
// Contained in jquery.qs.infobox.js: _create
.bind('mouseenter.' + name, function () {
    mouseOverBox = true;
})
```

Mouse Leaving the Pop-up List

When the mouse leaves the pop-up list, **mouseOverBox** is set to **false** and **hideTagLinks** is invoked.

```
JavaScript Code

// Contained in infobox.js
.bind('mouseleave.' + name, function () {
	mouseOverBox = false;
	that.hideTagLinks();

});

hideTagLinks: function () {
	!mouseOverBox && this.infoboxElement.hide();

},
```

Further Reading

You may find the following links useful in your investigation of the jQuery UI widget factory:

- jQuery UI API Developer Guide
- <u>Widget factory</u> on the jQuery UI wiki
- <u>Tips for Developing jQuery UI 1.8 Widgets</u> on Eric Hynds' blog
- <u>Understanding jQuery UI widgets: A tutorial</u> on bililite.com

| Community

To report documentation errors or provide feedback on this documentation, please send email to pagdoc@microsoft.com

Copyright $\ensuremath{\mathbb{C}}$ 2011 by Microsoft Corporation. All rights reserved.

```
Silk 1.0 - June 2011
How to: Check UIElement Properties with Coded UI Test
```

The following How-to topic will walk you through the creation of an automated test that checks for **UIElement** properties in a web application by using Visual Studio 2010 Coded UI Test. The Coded UI test performs actions on the user interface (UI) controls and verifies that the **UIElement** properties are displayed with the correct values. For this topic, the Mileage Stats Reference Implementation (Mileage Stats) will be the targeted application used for testing.

The automated test we will create in this topic will navigate to the Mileage Stats home page and verify that each of the images contain an **Alt** property with the expected values. This is important to ensure that the site is accessible and usable to all visitors.

Prerequisites

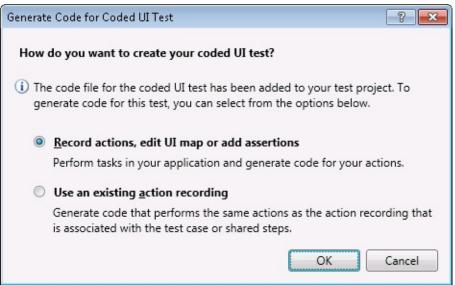
This topic requires you to have the same prerequisites required by Mileage Stats:

- Microsoft Visual Studio 2010 Professional, Premium, or Ultimate edition
- Microsoft Visual Studio 2010 SP1
- Microsoft .NET Framework 4.0 (installed with Visual Studio 2010)
- ASP.NET MVC 3
- Microsoft SQL Server Compact 4.0
- ADO.NET Entity Framework 4.1
- <u>NuGet</u>
- Internet Explorer 9

It is assumed that the Mileage Stats debug model web application has been deployed to a server running Microsoft Internet Information Services (IIS), and that the test site is <u>http://localhost/mileagestats</u>.

Steps

- In Visual Studio, create a new Test Project named CheckUIElementProperty. To do this, point to New on the File menu, and then click Project. In the New Project dialog, select Test Documents under Test Projects. Set the project's name to CheckUIElementProperty, specify a valid location, and then click OK.
- 2. Add a Coded UI Test. To do this, in Solution Explorer, right-click the CheckUIElementProperty project, point to Add, and select New Test. In the Add New Test dialog, select the Coded UI Test. Name the Coded UI Test VerifyImageAltProperty and click OK. In the Generate Code for Coded UI Test dialog select Record actions, edit UI map or add assertions, and click OK, as shown in the screenshot below.

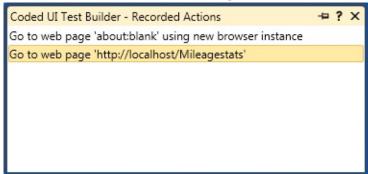


- 3. Record the UI Test as follows:
 - a. Click the **Record** button to start recording.



- c. Open Internet Explorer.
- d. Navigate to the Mileage Stats home page.
- e. Click the **Record** button to stop recording. Click the **Show Recorded Steps** button to check if the steps were recorded correctly. The coded UI test builder – Recorded Actions dialog

should look like the following screenshot.



- f. If there are unexpected steps, you can remove them by rightclicking the step you want to delete and selecting **Delete**.
- g. Click the Generate Code button. Name the method
 GoToHomePage. Click the Add and Generate button. The code will be generated in a file called UIMap.Designer.cs. This code can be customized according to your needs.

⊿Note:

Each time you generate the code from a recorded method, the code in the UIMap.Designer.cs file will be overwritten.

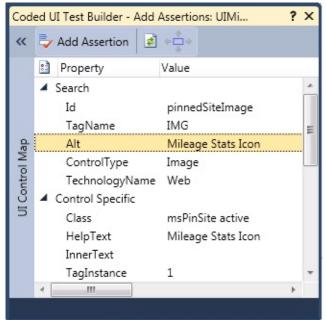
- 4. Use the **Coded UI Test Builder** to create a validation method to validate properties of the target UI control. For this example, you will verify that the property of each image on the home page is set to expected values by following these steps:
 - a. Add an assertion to the UI control. To do this, drag the crosshairs onto the UI control in your application that you want to test. When the box outlines your control, release the mouse. For example, drag the crosshairs to the mileage status icon, as shown below, on the home page.

☑ Note:

It will be easier to select the UI elements if the browser is maximized to 100%.



b. The properties for this control are now listed in the **Coded UI Test Builder - Add Assertions** dialog box. c. Right-click the **Alt** property, and select the **Add Assertion**. Leave all values set to the defaults and click **OK**.



- d. Repeat the above three steps for all images on the home page, such as MyOpenId, Yahoo, and the HTML5 icons. For this test, collect all multi-assertions into one **Assert** method.
- e. Click the **Generate Code** button. Name the method **AssertImageAltProperty**. The following code snippet will be auto-generated in the **VerifyImageAltProperty.cs** file.

C#

C#

```
[TestMethod]
public void CodedUITestMethod1()
{
   // To generate code for this test, select "Generate
   // For more information on generated code, see   /
   this.UIMap.VerifyAltPropertyMethod();
   this.UIMap.AssertImageAltProperty();
}
```

1. Verify the assertion method generated in **UIMap.Designer.cs**.

```
// <summary>
  // AssertImageAltProperty - Use 'AssertImageAltProp
// </summary>
public void AssertImageAltProperty()
ł
#region Variable Declarations
 HtmlImage uIMileageStatsIconImage = this.UIBlankPage
 HtmlImage uIMyOpenIDImage = this.UIBlankPageWindows1
 HtmlImage uIYahooImage = this.UIBlankPageWindowsInt\epsilon
 HtmlImage uISignInorRegisterImage = this.UIBlankPag€
#endregion
// Verify that 'Mileage Stats Icon' image's 'Alt' prop
Assert.AreEqual(this.AssertImageAltPropertyExpectedVa
// Verify that 'My OpenID' image's 'Alt' property equa
Assert.AreEqual(this.AssertImageAltPropertyExpectedVa
// Verify that 'Yahoo' image's 'Alt' property equals
Assert.AreEqual(this.AssertImageAltPropertyExpectedVa
// Verify that 'Sign In or Register' image's 'Alt' pro
Assert.AreEqual(this.AssertImageAltPropertyExpectedVa
}
```

- 1. Modify the generated code as follows:
 - a. Copy the code in UIMap.Designer.cs and paste it into UIMap.cs.
 - b. In UIMap.cs, if not already present, add the following **using** statement:

C#

using Microsoft.VisualStudio.TestTools.UIT

c. If you want to close the browser window automatically after each test case runs, add a **CloseBrowserWindow** function in the UIMap.cs partial class, as follows:

```
public partial class UIMap
{
    ...
    public void CloseBrowserWindow()
    {
        #region Variable Declarations
        BrowserWindow currentBrowserWindow =
        #endregion
        currentBrowserWindow.Close();
    }
    ...
}
```

 d. Add the following code snippet to the VerifyImageAltProperty Class in the VerifyImageAltProperty.cs file. The TestCleanup attribute in this method marks this method to be executed every time a test method completes its run.

C#

C#

```
//Use TestCleanup to run code after each te
[TestCleanup()]
public void MyTestCleanup()
{
  // To generate code for this test, select
  // For more information on generated code
  this.UIMap.CloseBrowserWindow();
  }
```

2. To run the test, Close all browser windows. Right-click inside the **VerifyImageAltProperty.cs** file and click **Run Tests**. The CodedUI

Test begins to execute; this will open a browser and will run the application programmatically based on the recorded steps and will assert if the conditions are met. If they are met, the test will pass. Otherwise, they will fail. Once the test completes, the results will be shown in the Test Results window.

Outcome

Here we created the Automation test project, which can be used to automate testing of your web application's UIElement **Alt** property.

Further Reading

- Testing the User Interface with Automated UI Tests on MSDN
- How to: Create a Coded UI Test on MSDN

| Community

To report documentation errors or provide feedback on this documentation, please send email to pagdoc@microsoft.com

Copyright $\ensuremath{\mathbb{C}}$ 2011 by Microsoft Corporation. All rights reserved.

```
Silk 1.0 - June 2011
How to: Create Automation Negative Case with Coded UI Test
```

The following How-to topic will walk you through the creation of an automated test for your web application by building a coded UI test using Visual Studio Premium or Visual Studio Ultimate. The coded UI test performs actions on the user interface (UI) controls and verifies that the UI controls display the expected values. For this topic, the Mileage Stats Reference Implementation (Mileage Stats) will be the targeted application used for testing.

The automated test created in this topic will assert a negative test case scenario. These tests will help to test the error handling capability of the application at the user level.

Prerequisites

This topic requires you to have the same prerequisites required by Mileage Stats:

- Microsoft Visual Studio 2010 Professional, Premium, or Ultimate edition
- Microsoft Visual Studio 2010 SP1
- Microsoft .NET Framework 4.0 (installed with Visual Studio 2010)
- ASP.NET MVC 3
- Microsoft SQL Server Compact 4.0
- ADO.NET Entity Framework 4.1
- <u>NuGet</u>
- Internet Explorer 9

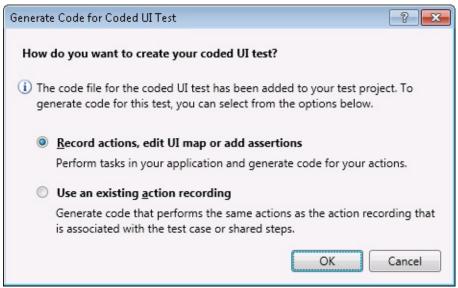
It is assumed that the Mileage Stats web application has been deployed to a server running Microsoft Internet Information Services (IIS) in debug mode, and that the test site is <u>http://localhost/mileagestats</u>.

Steps

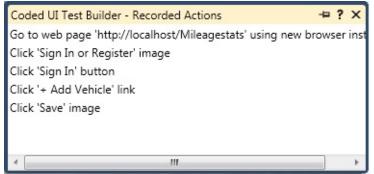
1. In Visual Studio, create a new Test Project named **NegativeTest**. To do this, point to **New** on the File menu, and then click **Project**. In the New Project dialog, select **Test Documents** under **Test Projects**. Set the project's name to **NegativeTest**, specify a valid location, and then click **OK**.

Recent Templates		.NET Framework 4 Sort by: Default	• 6 🕕	Search Installed Templates
Installed Template				Type: Visual C#
 Visual C# Windows Web Office Cloud Reporting SharePoint Shereight Test WCF Workflow Other Project Ty Database Modeling Project Test Projects Online Templates 	pes	Test Project	Virual C#	A project that contains tests.
Name:	NegativeTest			
Location:	C:\Users\v-haw	ei\Documents\Visual Studio 2010\Projects\	•	Browse
Solution name:	NegativeTest			Create directory for solution

2. Add a coded UI test. To do this, in Solution Explorer, right-click the NegativeTest project, point to Add, and select New Test. In the Add New Test dialog, select Coded UI Test. Enter the Test name as NegativeCodedUITest and click OK. In the Generate Code for Coded UI Test dialog, select Record actions, edit UI map or add assertions, and click OK.



- 3. Record the UI Test as follows:
 - a. Click the **Start Recording** button to start recording.
 - b. Open Internet Explorer.
 - c. Navigate to the Mileage Stats home page.
 - d. Click the Sign in or Register button.
 - e. On the Mock Authentication page, click the **Sign In** button.
 - f. On the Dashboard, click **Add Vehicle** button.
 - g. Click the **Save** button on the **Add Vehicle** form without entering any data.
- 4. Click the **Record** button to stop recording. Click the **Show Recorded Steps** button to check if the steps were recorded correctly. The coded UI test builder dialog should look like the following screenshot.



- 5. If there are unexpected steps, you can remove them by right-clicking the step you want to delete and selecting **Delete**.
- Click the Generate Code button. Name the method AddVehiclewithNullData. Click the Add and Generate button. The code will be generated in a file called UIMap.Designer.cs. This code

can be customized according to your needs.

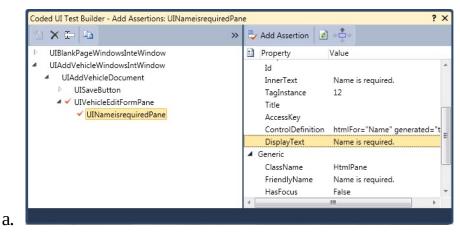
⊠Note:

Note: Each time you generate the code from a recorded method, the code in the UIMap.Designer.cs file will be overwritten.

- 7. Use the **Coded UI Test Builder** to create a validation method to validate properties of the target UI control. For this example, you will verify whether the error message is displayed after the **Save** button is clicked by following these steps:
 - a. Add an assertion to a UI control. To do this, drag the crosshairs onto the UI control in your application that you want to test. When the box outlines your control, release the mouse. For this example, drag the crosshairs to the **UINameisrequiredPane** UI element, which displays the validation message "Name is required," and release the mouse.

e	VEHICLE NAME
Add Vehicle	Name is required. MODEL YEAR
PA	Select a Year
	MAKE
	Select a Make
	MODEL
	Select a Model
	Photo
	Browse
	Save
	Note: Year, Make, Model and Photo are optional, but by providing them you get access to vehicle performance reports.

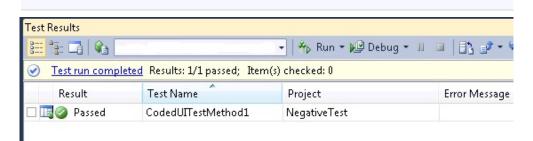
b. The properties for this control are now listed in the Coded UI
 Test Builder - Add Assertions dialog box.



- c. Right-click the **Display Text** property, and select the **Add Assertion** command. Keep the other default values and click **OK**.
- d. Click the **Generate Code** button. Name the method **VerifyErrorMessage**. The **VerifyErrorMessage** method will be auto-generated and added to the test method in the NegativeCodedUITest.cs file.

C#

```
[CodedUITest]
public class NegativeCodedUITest
ł
  public NegativeCodedUITest()
   ł
   }
   [TestMethod]
  public void CodedUITestMethod1()
   {
  // To generate code for this test, sele
  // For more information on generated cod
      this.UIMap.AddVehiclewithNullData();
      this.UIMap.VerifyErrorMessage();
   }
. . .
. . . .
}
```



- 8. Modify the generated code as follows:
 - a. Copy the code in UIMap.Designer.cs and paste it into UIMap.cs.
 - b. In UIMap.cs, if not already present, add the following **using** statement:

C#

```
using Microsoft.VisualStudio.TestTools.UIT(
```

c. If you want to close the browser window automatically after each test case runs, add a **CloseBrowserWindow** function in the UIMap.cs partial class as follows:

C#

```
public partial class UIMap
{
    ...
    public void CloseBrowserWindow()
    {
        #region Variable Declarations
        BrowserWindow currentBrowserWindow =
        #endregion
        currentBrowserWindow.Close();
    }
    ...
}
```

d. Add the following code snippet to the **NegativeCodedUITest** class in **NegativeCodedUITest.cs**. The **TestCleanup** attribute in this method marks this method to be executed every time a test method completes its run.

```
C#
// Use TestCleanup to run code after each
[TestCleanup()]
public void MyTestCleanup()
{
  // To generate code for this test, selec
  // For more information on generated code
  // http://go.microsoft.com/fwlink/?LinkIo
  this.UIMap.CloseBrowserWindow();
}
```

- 9. 9. Run the test method as follows:
 - a. Close all browser windows. Right-click inside the NegativeCodedUITest.cs file and click **Run Tests**. The coded UI test begins to execute; this will open a browser and will run the application programmatically based on the recorded steps and will assert if the conditions are met. If they are met, the test will pass. Otherwise, they will fail. In the test below, it passes.
 - b. Once the test completes, the results are shown in the Test Results window.

📰 💱 📑 🕼 🔄		🝷 🔺 Run 👻 😥 Debug	- 11 💷 🗄 🗗 - 9
Test run comple	eted Results: 1/1 passed; Item	n(s) checked: 0	
Result	Test Name	Project	Error Message
🗆 🔢 🥥 Passed	CodedUITestMethod1	NegativeTest	

Outcome

Here we created the Automation test project, which can be used to automate UI testing of your web application for negative test cases.

Further Reading

- <u>Testing the User Interface with Automated UI Tests</u> on MSDN
- How to: Create a Coded UI Test on MSDN

| Community

To report documentation errors or provide feedback on this documentation, please send email to pagdoc@microsoft.com

Copyright $\ensuremath{\textcircled{C}}$ 2011 by Microsoft Corporation. All rights reserved.

Silk 1.0 - June 2011 How to: Create Web Client UI Test using Coded UI Test

Overview

The following How-to topic will walk you through the creation of an automated test for your web application by creating a coded UI test using Visual Studio Premium or Visual Studio Ultimate. The coded UI test performs actions on the user interface (UI) controls and verifies that the correct controls are displayed with the correct values. For this topic, a default ASP.NET MVC 3 web application will be the targeted application used for testing.

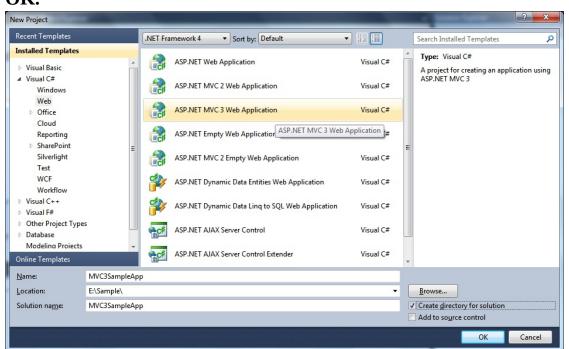
Prerequisites

This topic requires you to have the following prerequisites:

- 1. Microsoft Visual Studio 2010 Premium or Ultimate edition
- 2. Microsoft Visual Studio 2010 SP1
- 3. <u>Internet Explorer 9</u>
- 4. <u>ASP.NET MVC 3</u>

Steps

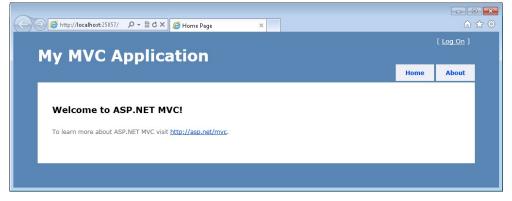
 In Visual Studio, create a new project. To do this, point to New on the File menu, and then click Project. In the New Project dialog, select Visual C# under Installed Templates, and then select ASP.NET MVC 3 Web Application. You can create applications using either Visual Basic or Visual C#. Name your project MVC3SampleApp, then click OK.



a. In the New ASP.NET MVC 3 Project dialog box, select
 Internet Application. Select Razor as the view engine. Click OK.

	Description:
Empty Internet Application	A default ASP.NET MVC 3 project with a account controller that uses forms authentication.
⊻iew engine: Razor ▼	
<u> </u>	
<u>C</u> reate a unit test project Test <u>p</u> roject name: MVC3SampleApp.Tests	
Test <u>p</u> roject name:	

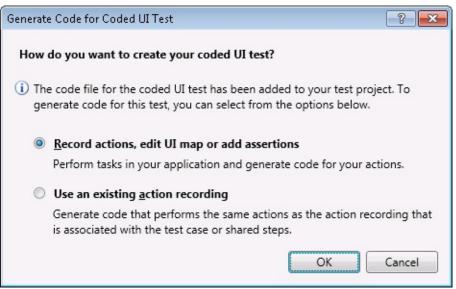
c. Select **Start Debugging** from the **Debug** menu. Visual Studio will launch a browser and open the application's home page. Notice that the address bar of the browser says localhost. That's because localhost points to your own local computer, which is running the application you just built. When Visual Studio runs a web project, a random port is used for the web server. In the image below, the random port number is 25857. When you run the application, you'll probably see a different port number.



- d. Click the **Log On** link on the Home page.
- e. Click the **Register** link and register an account by entering a **User name** and **Password**. Click the **Register** button.

	plication			
			Home	About
Create a New Ac	count			
Use the form below to creat	e a new account.			
Decowords are required to h	e a minimum of 6 characters in length.			
Account Informati	on			
User name				
Test Account				
Email address				
test@website.com				
Password				
•••••				
Confirm password				
Register				

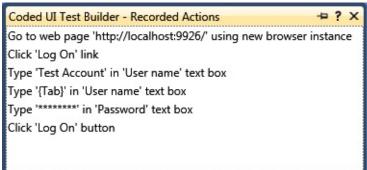
- Add a test project to the solution using the Test Project template. Rightclick the MVC3SampleApp solution in Solution Explorer, point to Add, and select New Project. In the New Project dialog, select Test Documents under Test Projects. Set the project's name to TestProject, specify a valid location, and then click OK.
- 3. Add a coded UI test. To do this, in Solution Explorer, right-click the **TestProject** project, point to **Add**, and select **New Test**. In the **Add New Test** dialog, select the **Coded UI Test**. Name the coded UI test **MyCodedUITest** and click **OK**.
- 4. In the **Generate Code for Coded UI Test** dialog, select **Record actions, edit UI map or add assertions**, and click **OK**.



The **Coded UI Test Builder** dialog box appears. You can use the **Coded UI Test Builder** to add a UI control to the <u>UIMap</u> for your test, or to generate code for a validation method that uses an assertion for a UI control. Click the red button to start recording steps.



- 5. Record the UI Test as follows:
 - a. Navigate to the home page
 - b. Click the **Log on** link on the top-right corner.
 - c. Enter the **user name** and **password**.
 - d. Click the **Log on** button.
- 6. Click the record button to stop recording. Click the **Show Recorded Steps** button to check if the steps are recorded correctly. The result should look like the screenshot below:



7. Click the **Generate Code** button in UIMap - Coded UI Test Builder. In the **Coded UI Test Builder-Generate Code** dialog box, change the

method name to **LogOn** and click the **Add and Generate** button.

8. The code, which will be generated to the UIMap.Designer.cs file, is shown in the following code snippet.

⊠Note:

If you record a new method, the generated code will override the code in UIMap.Designer.cs.

C#

```
[GeneratedCode("Coded UITest Builder", "10.0.402
public partial class UIMap
{
/// <summary>
/// LogOn - Use 'LogOnParams' to pass parameter
/// </summary>
public void LogOn()
 {
 #region Variable Declarations
  HtmlHyperlink uILogOnHyperlink = this.UIBingW
  HtmlEdit uIUsernameEdit = this.UIBingWindowsI
  HtmlEdit uIPasswordEdit = this.UIBingWindowsI
  HtmlInputButton uILogOnButton = this.UIBingWi
 #endregion
 // Go to web page 'http://localhost:9926/' usi
 this.UIBingWindowsInternetEWindow.LaunchUrl(ne
 // Click 'Log On' link
 Mouse.Click(uILogOnHyperlink, new Point(8, 11)
 // Type 'Test Account' in 'User name' text box
 uIUsernameEdit.Text = this.LogOnParams.UIUsern
 // Type '{Tab}' in 'User name' text box
 Keyboard.SendKeys(uIUsernameEdit, this.LogOnPa
 // Type '******' in 'Password' text box
  uIPasswordEdit.Password = this.LogOnParams.UIP
```

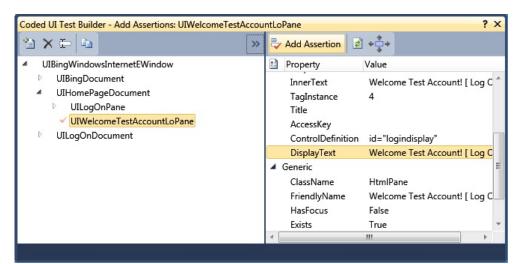
```
// Click 'Log On' button
Mouse.Click(uILogOnButton, new Point(17, 16));
}
...
}
```

- 9. Use the **UIMap** -**Coded UI Test Builder** to create an assert method to validate properties of the UI control. In this case, you will verify if logging on to the default site was successful by following these steps:
 - a. Add an assertion to the UI control. To do this, drag the crosshairs onto the UI control in your application that you want to test. When the box outlines your control, release the mouse. In this case, draw a box around the Logon portion of the home page.

	Welcome Test Account! [Log Off
My MVC Application	
	Home About
Welcome to ASP.NET MVC!	
To learn more about ASP.NET MVC visit http://asp.net/mvc.	
TO learn more about ASP, NET MYC VISIL http://asp.net/myc.	
	Coded UI Test Builder - Add Assertions: UIWelcomeTestAccountLoPane
	🖄 🗙 🏧 🚵 🔅 🖓 Add Assertion 🖻 + 🖕 +
	 UIBingWindowsInternetEWindow Property Value
	UIBingDocument InnerText Welcome Test Accord 4 UIHomePageDocument TagInstance 4
	▷ UILogOnPane Title
	UIWelcomeTestAccountLoPane AccessKey
	▷ UILogOnDocument ControlDefinition id="logindisplay" DisplayText Welcome Test Acco
	displayiext welcome restracto d Generic
	ClassName HtmlPane
	FriendlyName Welcome Test Acco
	HasFocus False
	Evists True
	Exists True

The properties for this control are now listed in the **Coded UI Test Builder - Add Assertions** dialog box.

b. Right-click the **Display Text** property and select **Add Assertion**. Keep all values as default and click **OK**.



 c. Click the Generate Code button in the Coded UI Test Builder. Name the assertion method VerifyLogOn. This method will be auto generated and added to the CodedUITestMethod1 in the MyCodedUITest.cs file, as shown below:

```
C#
```

```
[TestMethod]
public void CodedUITestMethod1()
 {
   // To generate code for this test, select "Gene
   // For more information on generated code, see
    this.UIMap.LogOn();
    this.UIMap.VerifyLogOn();
}
```

- 10. Modify the generated code as follows:
 - a. Copy the code in UIMap.Designer.cs and paste it into UIMap.cs.
 - b. In UIMap.cs, if it is not already present, add the following **using** statement:

```
C#
```

using Microsoft.VisualStudio.TestTools.UIT

c. If you want to close the browser window automatically after each test case runs, add a **CloseBrowserWindow** function in the UIMap.cs partial class, as follows:

```
C#
public partial class UIMap
{
    ...
    public void CloseBrowserWindow()
    {
        #region Variable Declarations
        BrowserWindow currentBrowserWindow =
        #endregion
        currentBrowserWindow.Close();
    }
    ...
}
```

d. Add the following code snippet to the **MyCodedUITest** Class in **MyCodedUITest.cs** file. The **TestCleanup** attribute in this method marks this method to be executed every time a test method completes its run.

```
C#
//Use TestCleanup to run code after each to
[TestCleanup()]
public void MyTestCleanup()
{
  // To generate code for this test, select
  // For more information on generated code
  this.UIMap.CloseBrowserWindow();
}
```

11. To run the test, close all browser windows. Right-click inside the **MyCodedUITest.cs** file and click **Run Tests**. The coded UI test will begin to execute; this will open a browser, will run the application programmatically based on the recorded steps, and will assert if the conditions are met. Once the test completes, the results will be shown in the Test Results window.

Outcome

The Automation test project is created and can be used to automate UI testing of your web application.

Further Reading

- <u>Testing the User Interface with Automated UI Test</u> on MSDN.
- How to: Create a Coded UI Test on MSDN.

| Community

To report documentation errors or provide feedback on this documentation, please send email to pagdoc@microsoft.com

Copyright $\ensuremath{\mathbb{C}}$ 2011 by Microsoft Corporation. All rights reserved.

This document is provided "as-is". Information and views expressed in this document, including URL and other Internet Web site references, may change without notice. You bear the risk of using it. Some examples depicted herein are provided for illustration only and are fictitious.No real association or connection is intended or should be inferred.

© 2011 Microsoft. All rights reserved.

Microsoft, Windows, Windows Server, Windows Vista, Silverlight, Expression Blend, MSDN, IntelliSense, Visual C#, Visual C++, and Visual Studio are trademarks of the Microsoft group of companies.

All other trademarks are the property of their respective owners.

| Community

To report documentation errors or provide feedback on this documentation, please send email to pagdoc@microsoft.com

Copyright $\ensuremath{\mathbb{C}}$ 2011 by Microsoft Corporation. All rights reserved.