Architecture

# Introduction

The Mileage Stats Reference Implementation (Mileage Stats) is a cross-browser, ASP.NET MVC application that focuses on features of modern browsers. The application offers two types of user experiences:

* 1. **A traditional website experience**, which performs a form post and reloads the page each time a button or hyperlink is clicked.
  2. **A rich website experience**, which performs an initial page load, and then only executes server requests when new data is required or updated, additionally the lack of a full page reload, enables the animation of client-side state changes.

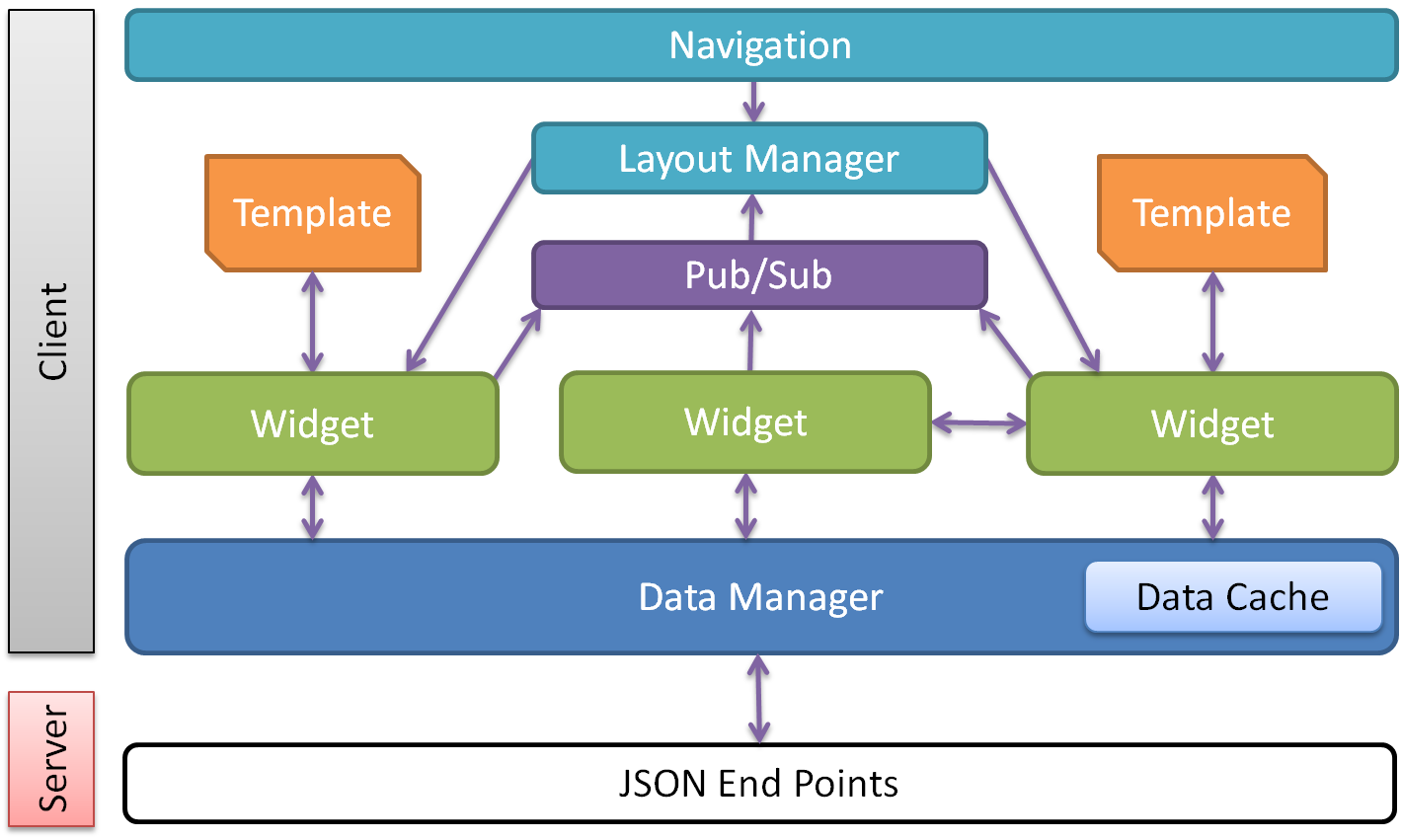
The rich website experience provides a superior experience for the end user as the application feels more responsive, and more like a desktop application to use. However, some users do not have script enabled or available on their User Agent (browser or accessibility tool such as a screen reader) and therefore require us to support the traditional website experience.

The traditional website experience 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, where 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 these two experiences introduces application complexity that requires careful planning on both the client-side and server-side to ensure the application is responsive, maintainable, has clean separation of concerns, and is testable.

Web applications run in web browsers. Developers need to determine early in the design phase, which experience the user should expect in each version of browsers the application will support. Support of older browsers may limit your technology choices and will affect the runtime experience of the application. Shims and polyfills are available for some technologies, such as providing HTML 5 support 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 these decisions early on allows establishing realistic expectations for users and project stakeholders.

This chapter provides a high level, mental map of the Mileage Stats client-side architecture 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 which is represented below as the Pub/Sub.
  + **Navigation** explains how to manage user gestures and coordinates animations, which is represented below as the Navigation and Layout Manager.
  + **Data** provides guidance for client-side data request and data caching which is represented below as the Data Manager.
  1. Mileage Stats Client Architecture
  2. 

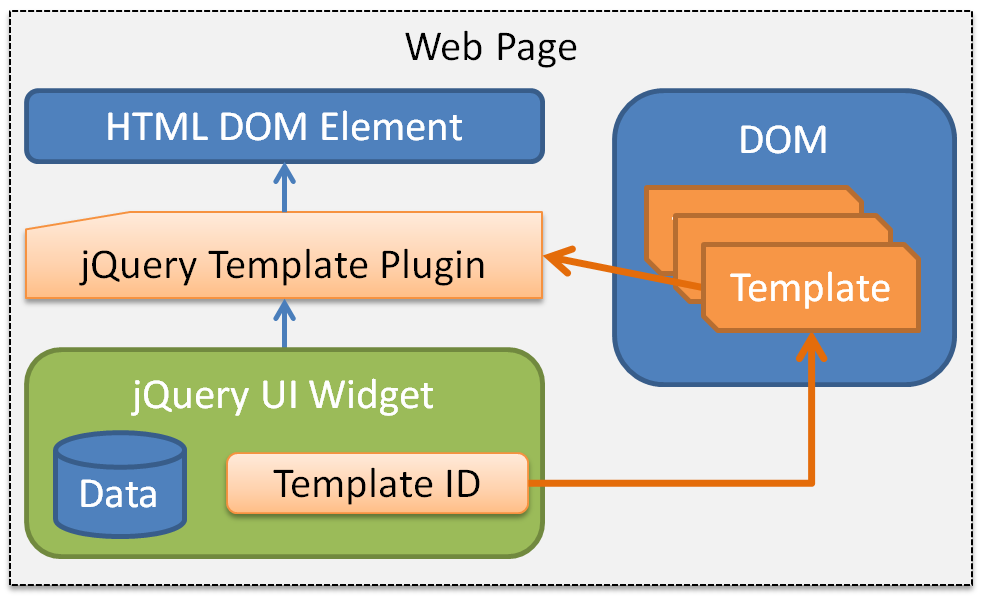
## What you will learn in this chapter

* + Options and strategies for getting the right HTML to the client
  + The advantages to modular code and an example 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 the whole client-side
  + How the Mileage Stats team solved a number of challenges related to structure, modularity, communication, navigation, and data
  1. The technologies and libraries discussed in this chapter are JavaScript, jQuery, jQuery UI Widgets, and jQuery BBQ: Back Button & Query Library (BBQ).

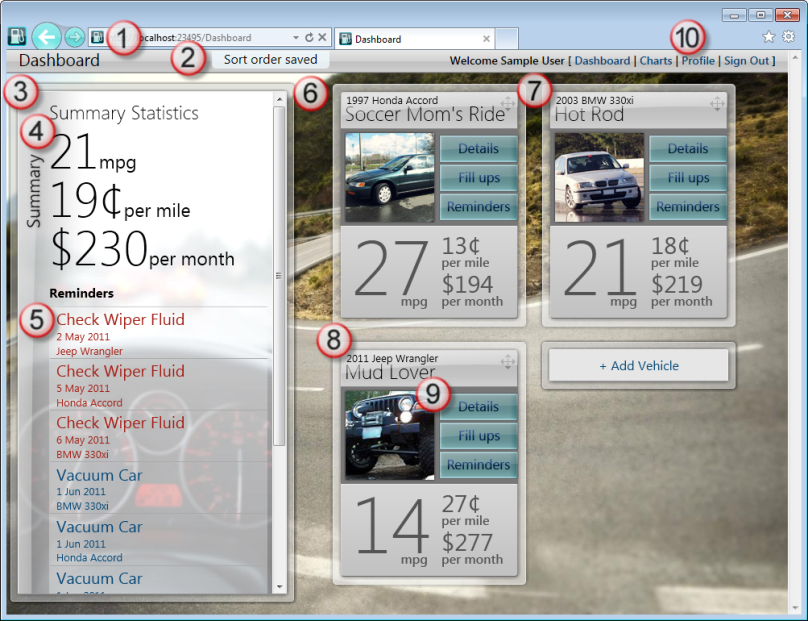
# Structure

* 1. Websites like Mileage Stats provide an engaging user experience when viewed with modern browsers with JavaScript enabled. The site can also be viewed without JavaScript enabled and will function when viewed with an older browser.
  2. 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.
  3. To support both rich website and traditional website user experiences, the Project Silk team chose to have the web server generate the initial HTML; then, using JavaScript to detect the browser capabilities, we enhance the user experience replacing the server generated HTML structure with a client-side version. Replacement includes 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 server-generated HTML is called Progressive Enhancement. Progressive Enhancement enables and adds features to the client-side experience based on browser capabilities.
  4. Client-side UI structure can be generated with JavaScript, loaded on-demand from the server, or rendered by a plugin or a library. Initially, the team tried on-demand 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.
  5. 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.

## jQuery Templates

* 1. jQuery templates are HTML markup with inline JavaScript expressions that are used to populate values in the markup. The jQuery Template plugin applies data to the template and renders the output into the DOM. Mileage Stats uses jQuery UI widgets to coordinate getting the data, applying it to the template using the plugin, and overwriting the DOM element.
  2. jQuery Template Rendering
  3. 
  4. 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.
  5. 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.
  6. 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.
  7. For more information on the jQuery Template plugin and authoring templates see the jQuery Templates in the Further Reading section. For more information on jQuery templates in Mileage Stats, see Chapter 5, "HTML Templates."

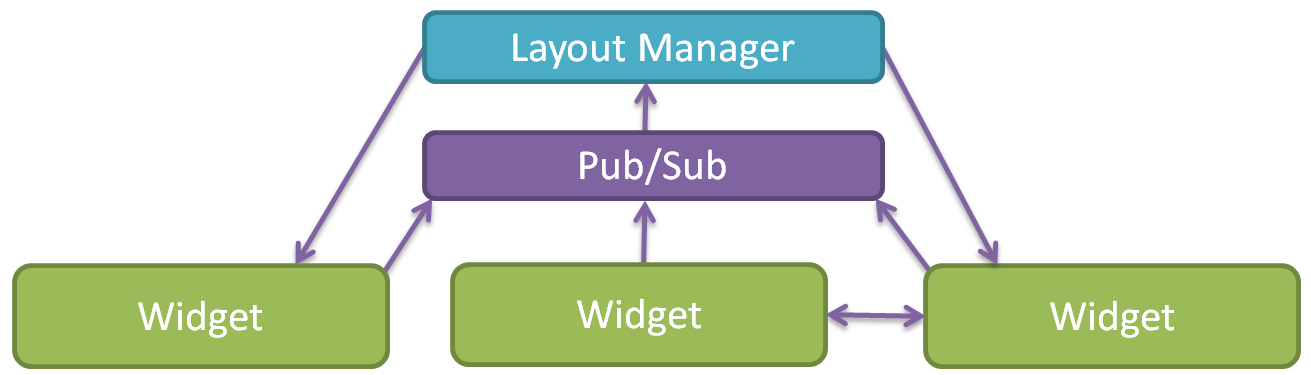
# Modularity

* 1. 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.
  2. 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.
  1. 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.
  2. **Pinned sites**. JavaScript object provides the pinned sites implementation.
  3. **Status widget**. Provides management and display of user notification messages.
  4. **Summary widget**. Acts as a container, managing its child registration, statistics, and reminders widgets.
  5. **Statistics widget**. Displays summary statistics for all vehicles.
  6. **Reminders widget**. Lists overdue and upcoming maintenance reminders. Manages the action of clicking on a reminder.
  7. **Layout manager widget**. Services navigation requests and coordinates UI layout changes.
  8. **Vehicle list widget**. Displays the vehicle tiles in one or two column listing. Invokes the child widget animation when required and controls when child widgets display in expanded or contracted view.
  9. **Tile widget**. Provides drag-and-drop capability for the child vehicle widget.
  10. **Vehicle widget**. Displays vehicle information in expanded or contracted view. Manages the actions of each button.
  11. **Header widget**. Provides top-level navigation and user name display. Manages actions when a hyperlink in the Header is clicked.
  12. Modularization in Mileage Stats
  13. 
  14. 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

* 1. 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, comprising the whole application. Well-defined communication between objects is critical not only from a functional view, but from an architectural view as well.
  2. 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).

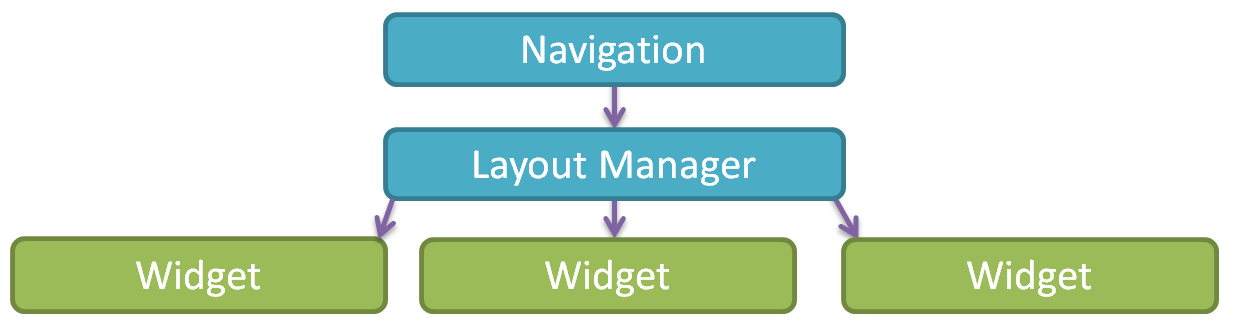
## Direct Communication

* 1. 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.
  2. Layout Manager and Pub/Sub
  3. 

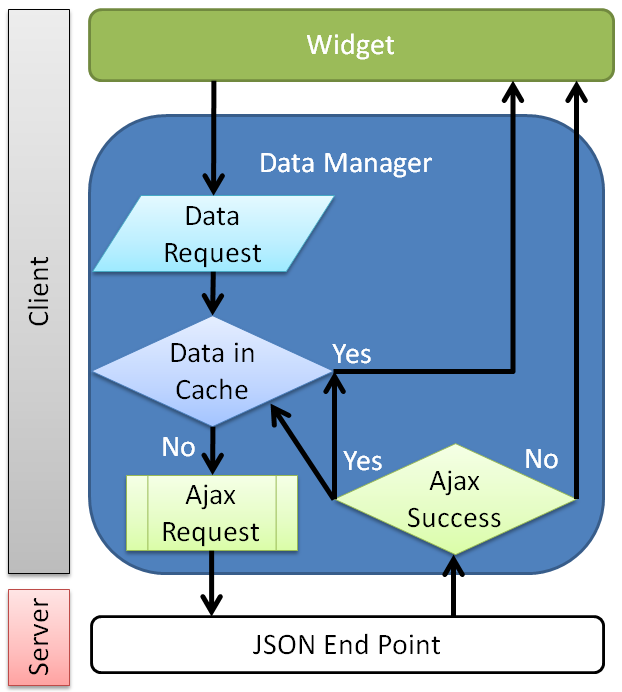
## Loose Communication

* 1. 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 from having direct knowledge of one another. Pub/sub messages are individually defined and can optionally contain a payload.
  2. 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.
  3. 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.
  4. Mileage Stats widgets have publish and subscribe functions passed in their options object during construction to decouple them from the pub/sub implementation.
  5. For more information about the pub/sub implementation in Mileage Stats, see Chapter 8, "Communication."

# Navigation

* 1. 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.
  2. The jQuery BBQ plugin (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.
  3. The Mileage Stats layout manager is a widget that works in conjunction with the BBQ plugin to service navigation requests. It subscribes to the BBQ plugin **hashchange** event, and initiates layout changes based on address bar URL changes.
  4. Navigation and Layout Manager
  5. 
  6. 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.
  7. 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.

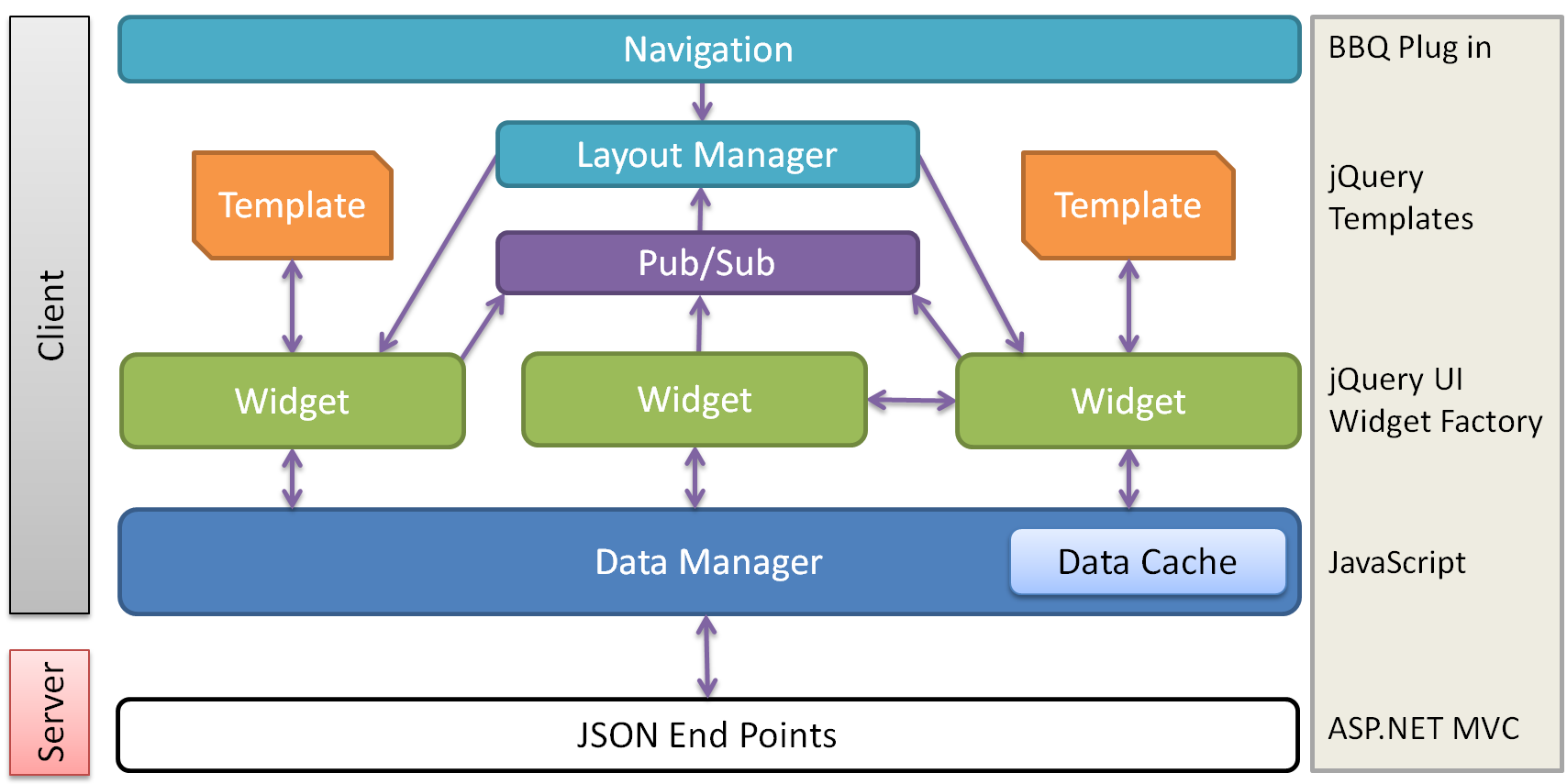
# Data

* 1. When designing your client-side data architecture, several key decisions will impact 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 pre-fetched, and if so, how much? Will data be cached, and if so, how much and where will the data cache live: in a JavaScript object, the DOM, the browser window object, or other browser data storage API? Answers to these questions are requirements specific to your application.
  2. In Mileage Stats, all data requests are via Ajax and are routed through the data manager. Having a single object handle data requests simplifies the calling client-side code, improves application testability, and facilitates cleaner application evolution when client-side libraries advance or change. The single, data manager object also affords the opportunity to implement client-side data caching in a central location. Data is cached in a JavaScript object, rather than using HTML 5 local storage or similar APIs, in order to meet the cross-browser requirements of the application.
  3. 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 provides a potential scalability boost to the application because repeated requests to the server for the same data are no longer necessary, requiring less server processing per user.
  4. Widgets are responsible for requesting their data from the data manager. The data manager services the request, checking 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.
  5. Data Request
  6. 
  7. For in-depth coverage on data management and caching, see Chapter 10, "Data, Caching, and Validation."

# Summary

* 1. 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 client-side objects and their implementation mapped to libraries or frameworks.

Mileage Stats Client Architecture Technology Map

* 1. 

# Further Reading

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