Modularity

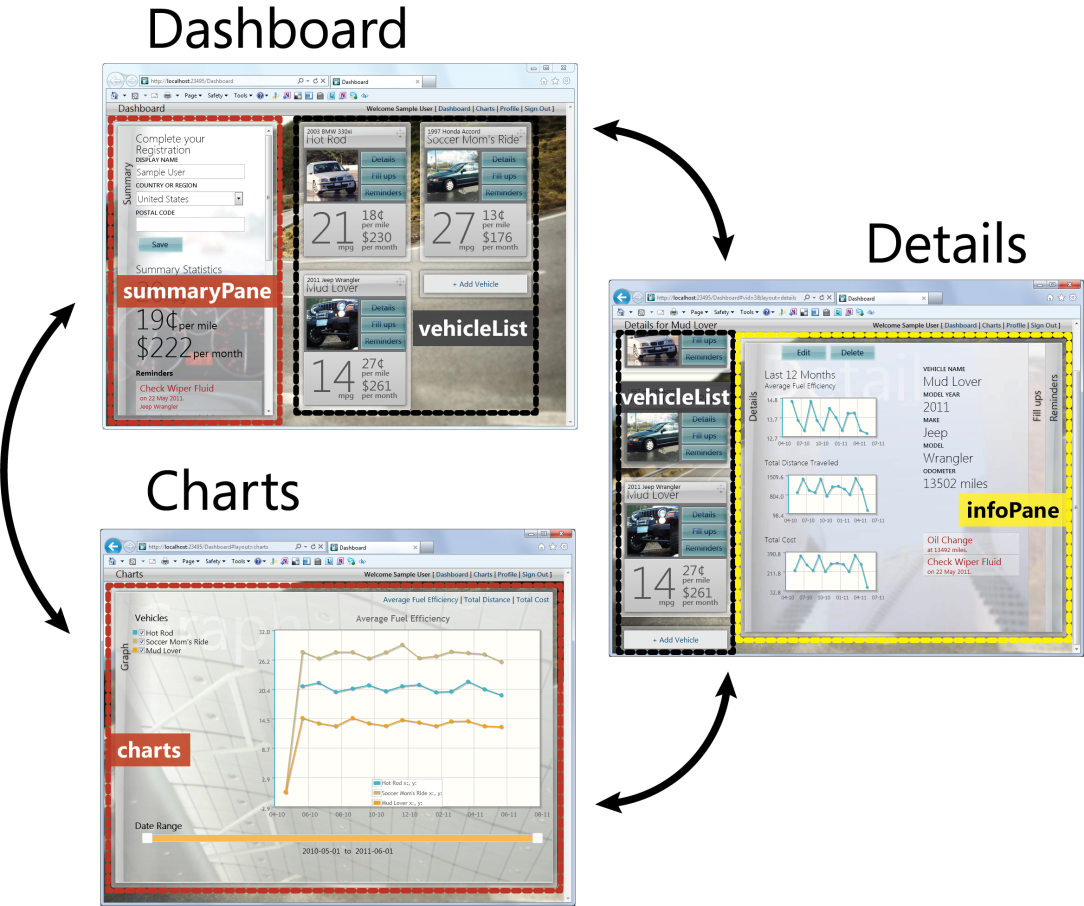
# Introduction

* 1. 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.
  2. 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.
  3. 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.
  4. 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

* 1. 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.
  2. HTML
  3. <div id="dashboard-page">
  4. <div id="vehicles">
  5. <div id="vehicle-list-content”>
  6. <div class="wrapper">
  7. <div class="vehicle"/>
  8. <div class="vehicle"/>
  9. <div><a>Add Vehicle</a></div>
  10. </div>
  11. </div>
  12. </div>
  13. <div id="main-chart"/>
  14. <div id="fixed">
  15. <div id="summary">
  16. <div id="registration"/>
  17. <div id="statistics"/>
  18. <div id="reminders"/>
  19. </div>
  20. <div id="info"/>
  21. </div>
  22. </div>
  23. 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.

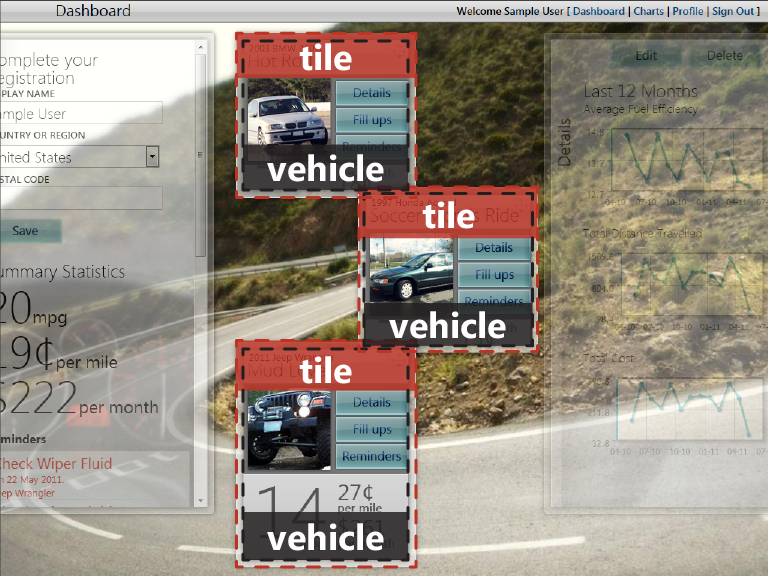
## Influences from the Layout

* 1. 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.
  2. The three main layouts in the immersive experience
  3. 
  4. 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.
  5. 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.

* 1. Animating from Dashboard to Details

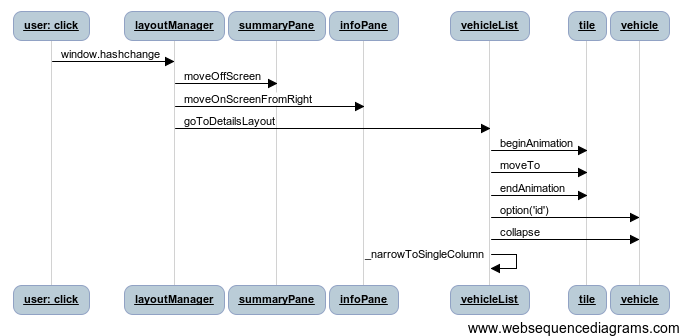
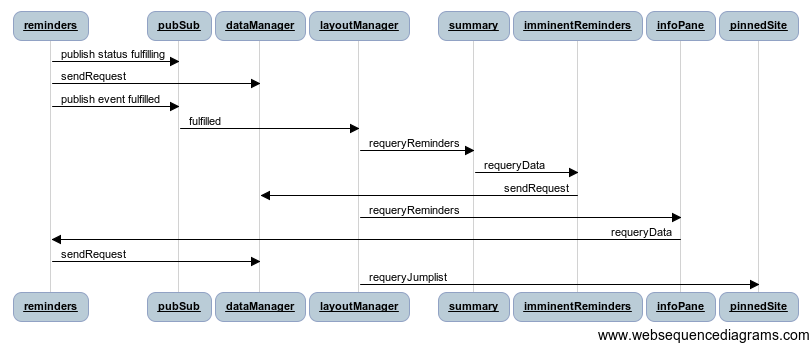
**

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, application-agnostic widget. Then the vehicleList widget could apply the animation behavior widget to itself.

## Influences from Refreshing Data

* 1. 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.
  2. Statistics widget is responsible for refreshing content
  3. 
  4. 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."
  5. 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

* 1. 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.
  2. Sequence when transitioning from Dashboard to Details
  3. 
  4. 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.
  5. 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 ."
  6. 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.
  7. 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**.
  8. 
  9. 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.
  10. 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.
  11. So far in this chapter you have seen modules have a variety of responsibilities. These responsibilities can be categorized into types.

# Types of Modules

* 1. 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

1. 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.
   1. 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

1. 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

* 1. 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 includedataManager. 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.

* 1. JavaScript
  2. (function($){
  3. $.fn.doubleSizeMe = function() {
  5. return this.each(function() {
  6. var $this = $(this),
  7. width = $this.width(),
  8. height = $this.height();
  9. $this.width(width \* 2);
  10. $this.height(height \* 2);
  11. });
  12. };
  13. })(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.

* 1. JavaScript
  2. $('.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

* 1. jQuery Plugin Authoring Guidelines: <http://docs.jquery.com/Plugins/Authoring>
  2. To learn more about how to build widgets, see Chapter 3, "jQuery UI Widgets."
  3. To learn more about how the tile widget performs its animation, see Chapter 9, "Navigation ."
  4. To learn more about the data abstraction in Mileage Stats, see Chapter 10, "Client Data Management and Caching."