Client Data Management and Caching

# Introduction

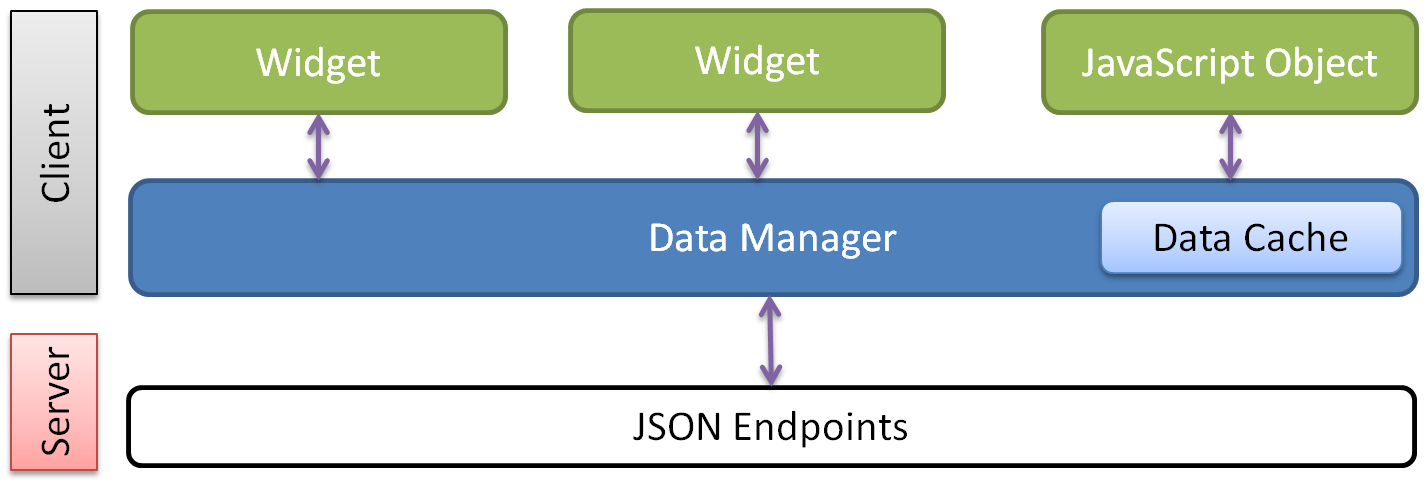
Web applications are in the business of presenting data to the user. In rich, interactive, client-centric applications like Mileage Stats, users expect the application to respond quickly to mouse gestures, page transitions, or saving form data. Delays caused by data retrieval or saving can negatively 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.
  1. 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
  1. The technologies and libraries discussed in this chapter are Ajax, JavaScript, jQuery, and jQuery UI Widgets.
  2. **Note:** Data validation on the client and server is covered in the next chapter, "Server-Side Implementation."

# Client-Side Data Design

* 1. 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.
  2. 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.

* 1. 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.
  2. 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

* 1. 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.
  2. 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

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

* 1. 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.

* 1. JavaScript
  2. // contained in mileagestats.js
  3. summaryPane = $('#summary').summaryPane({
  4. sendRequest: mstats.dataManager.sendRequest,
  5. invalidateData: mstats.dataManager.resetData,
  6. publish: mstats.pubsub.publish,
  7. header: header
  8. });
  9. 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.
  10. JavaScript
  11. // contained in mstats.summary.js
  12. \_setupStatisticsWidget: function () {
  13. var elem = $('#statistics');
  14. mstats.summaryPane.statistics = elem.statisticsPane({
  15. sendRequest: this.options.sendRequest,
  16. dataUrl: elem.data('url'),
  17. invalidateData: this.options.invalidateData,
  18. templateId: '#fleet-statistics-template'
  19. });
  20. },

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.

* 1. CSHTML
  2. // contained in \Views\Vehicle List.cshtml
  3. <div id="statistics" class="statistics section"
  4. data-url="@URL.Action("JsonFleetStatistics","Vehicle")">
  5. ...
  6. </div>

### Performing a Data Request

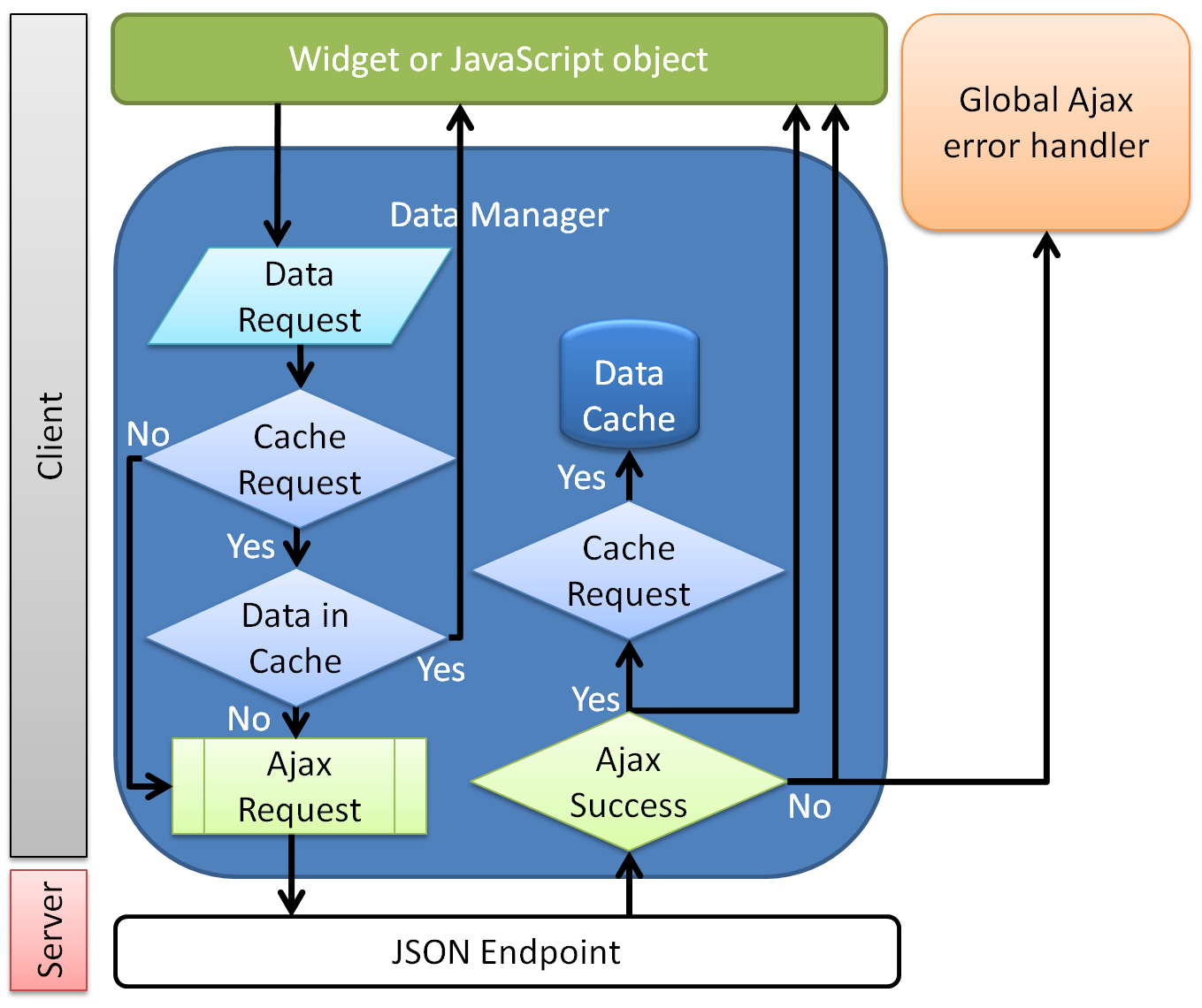
* 1. Specifically, the **sendRequest** method has the same method signature as the jQuery **ajax** method that takes a settings object as the only argument. The below **\_getStatisticsData** method passes the **sendRequest** method an object literal that encapsulates the **url**, **success**, and **error** callbacks. When the Ajax call completes, the appropriate callback will be invoked and its code will execute.
  2. JavaScript
  3. // contained in mstats.statistics.js
  4. \_getStatisticsData: function () {
  5. var that = this;
  6. that.options.sendRequest({
  7. url: that.options.dataUrl,
  8. success: function (data) {
  9. that.\_applyTemplate(data);
  10. that.\_showStatistics();
  11. },
  12. error: function () {
  13. that.\_hideStatistics();
  14. that.\_showErrorMessage();
  15. }
  16. });
  17. },
  18. 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.
  19. 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

1. 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.
2. 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



* 1. 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. If a global Ajax error handler has been defined, it will be invoked after the **error** callback.
  2. **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.
  3. 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."
  4. JavaScript
  5. // contained in mstats.data.js
  6. sendRequest: function (options) {
  7. // getRelativeEndpointUrl ensures the URL is relative to the website root.
  8. var that = mstats.dataManager,
  9. normalizedUrl = mstats.getRelativeEndpointUrl(options.url),
  10. cachedData = mstats.dataStore.get(normalizedUrl),
  11. callerOptions = $.extend({ cache: true },
  12. that.dataDefaults,
  13. options,
  14. { url: normalizedUrl });
  15. if (callerOptions.cache && cachedData) {
  16. options.success(cachedData);
  17. return;
  18. }
  19. callerOptions.success = function (data) {
  20. if (callerOptions.cache) {
  21. mstats.dataStore.set(normalizedUrl, data);
  22. }
  23. options.success(data);
  24. };
  25. $.ajax(callerOptions);
  26. },
  27. **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

* 1. 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.
  2. 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.

* 1. JavaScript
  2. // contained in mstats.data.js
  3. mstats.dataStore = {
  4. \_data: {},
  5. get: function (token) {
  6. return this.\_data[token];
  7. },
  8. set: function (token, payload) {
  9. this.\_data[token] = payload;
  10. },
  11. clear: function (token) {
  12. this.\_data[token] = undefined;
  13. },
  14. };

## Removing a Data Cache Item

* 1. 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.
  2. JavaScript
  3. // contained in mstats.data.js
  4. resetData: function (endpoint) {
  5. mstats.dataStore.clear(mstats.getRelativeEndpointUrl(endpoint));
  6. }

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.

* 1. JavaScript
  2. // contained in mstats.statistics.js
  3. refreshData: function () {
  4. this.\_getStatisticsData();
  5. },
  6. requeryData: function () {
  7. this.options.invalidateData(this.options.dataUrl);
  8. this.refreshData();
  9. },
  10. 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

* 1. 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.
  2. 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

* 1. For detailed information on jQuery UI widgets, see Chapter 3, "jQuery UI Widgets."
  2. For detailed information on the global Ajax error handler, see Chapter 6, "Application Notifications."
  3. For information on data validation, see Chapter 11, "Server-Side Implementation."
  4. HTML 5 Web Storage:  
     <http://dev.w3.org/html5/webstorage/>
  5. jQuery:  
     <http://jquery.com/>
  6. jQuery ajax() method:  
     <http://api.jquery.com/jQuery.ajax/>
  7. jQuery data() method:  
     <http://api.jquery.com/data/>
  8. jQuery ajaxError() method:  
     <http://api.jquery.com/ajaxError/>

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