Managing State
HTML5 Applications

Written by Kevin Griffin

YOUR APP. NEW. AGAIN.

Overview

Back in 2008, I wrote my first major web application. It was a management portal for a company who ran a network of campground memberships around the country. Our goal with the application was to provide an interface and experience that simulated the look, feel, and responsiveness of using a desktop application. This wasn't a particularly easy task with our budget and level of knowledge. Our server technology was ASP.NET WebForms and it wasn't particularly well suited for our task. jQuery adoption had grown large over the last year or two and served as one of the only trustworthy libraries to use for heavy client side management.

Our troubles came from how to best control state in the application. ASP.NET WebForms back then wanted to control the majority of a page lifecycle. If you clicked on an element, it would postback to the server. Change a dropdown box? Postback to the server. Press a button? Postback to the server. The power and flexibility of ASP.NET WebForms came from managing the entire state. To make our application work well, we had to disable parts of ASP.NET that wanted to take control even though that meant giving up some of the features ASP.NET provided.

On the client-side, we essentially had to roll our own framework. jQuery provided a reliable conduit between elements on our page and our custom JavaScript. We wrote a lot of redundant code for retrieving and setting the value of elements on our page. Event management was a chore. We lacked a mechanism for maintaining the state of the page and the context of the elements on it. One hack we used on a regular basis was to inject special classes into our elements to define meaning. . For example, if you looked a table row for a user listing, it would likely have a class name userid-XYZ. During runtime, we used jQuery to extract the classes starting with "userid-" and pulling the values off of it.

Web applications have really matured since this application was built. We are in a unique position of having immediate access to amazing libraries and frameworks that have simplified how we can build robust, scalable web applications. There is no any holy grail though. You still have to design responsibly with the plethora of tools available to you. In this article, we will discuss what you should be concerned about when managing state in HTML5 applications and the tools that are available to you.

Considerations of Stateless Web Applications

The primary rule for building a web application is that there is no such thing as state on the server. Every request made by the client should provide enough information for the server to properly serve up a response.

You might think, "How do applications built with ASP.NET get around this?
They manage state." Absolutely!

There are a variety of frameworks built to handle these concepts for you. They are all built on top of the principle that the server only knows how to handle one request at a time, and they add facades around several of the concepts we will discuss.

For example, imagine you are making a GET request to http://myDomain.com/api/users
This provides the server with two very important pieces of information regarding your intent. First, you have told the server you are wishing to GET a particular resource. HTTP verbs provide the server with context of what you'd like to accomplish. GET requests return data. POST requests create new data. PUT requests update or replace data. DELETE requests remove data.

Next, you are providing the server with a URL. This is considered an endpoint. The server uses the URL in combination with the HTTP verb to determine what action should be taken. It is also possible your request includes some headers. These headers can include information such as who you are (via a cookie), what data format you'd like to receive the response in, and if your browser accepts compressed content. The absence or presence of headers can cause the server to respond differently.

What if you don't pass credentials? The server may respond with a 403 (Forbidden) status code. The server can respond with different HTTP status codes to provide clues into what happened. In some cases, data will be provided, but sometimes you may only receive the status code.

Since the server cannot reliably know anything about the previous requests we have made, the majority of the pressure for maintaining state is on the heels of the client.

Client-Side Management

Importance of Patterns

As client-side HTML5 and JavaScript applications have grown in size, our ability to manage them has become more of an issue. Single Page Applications are especially concerning, because the client needs to store as much state as possible.

In the classic days of web development, all the communication between the markup on the page and the code in JavaScript was the responsibility of the developer. jQuery smoothed most of the bumps, but it was still the responsibility of the developer to perform common user interface manipulation. What if you needed to set or get the value of a text box? It was your job to write that code. What if you needed to populate an empty table with fifty rows? It was your job to write that code. jQuery simply acted as a mechanism to quickly and easily write that code all while being cross-browser.

Over the past several years, we have seen many common desktop patterns move to the browser such as MVC, MVVM, and MVP. As an example, Knockout.JS can be easily considered an implementation of MVVM. Given a page and accompanying JavaScript, the developer could bind elements of the page to a central model defined in code.

If the page contained a text box and you changed the value of it, the value would automatically sync to a variable within code. Additionally, if you changed the value of the variable, it would automatically sync to the text box on the screen. Dozens of lines of code could be saved by using the framework.

Even the act of updating a table or repeating element to the contents of an array is second nature for these type of frameworks. In old jQuery-based systems, each row and cell would need to be created and appended to each other. With Knockout.JS, you only need to update the array and Knockout will take care of the rest.

The implementation of patterns allows the developer to concentrate more of the development process by implementing repetitive tasks. Each pattern has a defined process for how to manage state, how to listen for events, and how to communicate back to the server.

Address Bar

The address bar acts as your first point of contact between the browser and server, and that role holds a lot of responsibility. The address bar tells both the server and client what the initial view of the application should be at render time.

Let's examine the components of an address:

http://myDomain.com/some/server/path#/some/client/path

The pound sign (#) acts as the delimiter for the path. The left side of the URL is the server path, and the right side is the client path. Server path refers to what the server is responsible for responding to. In an ASP.NET environment, the URL should direct to an appropriate Controller and Action or a corresponding ASPX file. The page returned by the server will include references to JavaScript files that will define the client aspect of the application, and process the client path appropriately.

The client path is used to inform the browser about the current state of the page being loaded. Think about it this way: what happens when your users press "Refresh" or "Back", accidentally closes the browser window, or the browser crashes. What should happen when the user recovers from any of these scenarios? Ideally, the page should be able to return itself as close to the original state of the page before the event happened.

What does this have to do with the address bar? The URL in the address bar is tracked by the browser, and in most cases, it is the only piece of information available to the browser for state reconstruction. In a single page application (SPA), the client path tells the browser what the current view shouldbe or provides the client with additional parameters and state data. For example, if you are a Gmail user, watch the address bar as you open up emails or compose drafts. The address bar will track which messages you are looking at and which open compose email views you have open. On a regular interval, Gmail will save drafts and update the address with a message draft ID. On a page refresh, Gmail can construct the view with almost a 100% level of accuracy.

Frameworks such as Angular make implementing these routing scenarios easy. What happens when you have long living views such as those associated with data-entry? It might not make sense to post data to the server on a regular interval. There are many options available for storing data across requests to the server.

Client-Side Storage Options

There are numerous methods for storing data locally in the users' browser.
The two mechanisms that are worth talking about because they are widely available across all evergreen and legacy browsers.

Cookies

The browser cookie is the longest living mechanism for maintaining state information for website (Netscape implemented in 1994). They provide a simple key/value store for browsers to save data. The contents of a cookie are transmitted with every request to a domain. Cookies have theoretical limits depending on the browser you are using. In most case, your cookies are limited to about 20 cookies per domain and a size limit of 4096 bytes per cookie. After a little bit of math, you will see that cookies are not particularly well suited for large amounts of data. Rightfully so, because you would not want to send a massive amount of information to the server on every request.

Session and Local Storage

The next two mechanisms can be talked about in unison because their APIs are identical. Session storage is a feature in browsers that allow websites to store data for a users' browser session.

As with cookies, we are storing data within a simple key/value store. Data access is extremely quick, and alleviates the requirement for having to make frequent server calls with updates. In session storage, the contents of the storage go away once the browser window instance (or tab) is closed. More easily put, the storage lasts for the duration of the browser window session. If the session goes away, so does the storage. There is also no cross-storage between browser window instances.

Local storage follows the same syntax as session storage, but it is designed to persist past a browser instance closing. Imagine the above scenario when a browser crashes. If your user is working within a long living page instance, you can periodically save the data to local storage and retrieve it after the crash. Additionally, you can report local storage changes across multiple browser instance. A user with a dozen open tabs can keep them all in sync by subscribing to browser events.

Storage limits still depend on the browser, but generally you can store up to 5 megabytes per domain. Developers that need to support back to Internet Explorer 8 can rest assured that the Session and Local Storage APIs are available.

Other Options

Developers that are living in the bleeding edge might want to also look at WebSQL and IndexedDB. Both of these options provide a more SQL-like client side database.

Any issues will relate to these features not actively working in common browsers (such as Internet Explorer or Safari). It is recommended that developers tread lightly when exploring the use of these features in their web applications.

Server-Side Management

As alluded to at the beginning of this article, the server needs to be relativity dumb. Any request coming into the server should provide the necessary information to accomplish the requested task. This can bring up several questions on how to secure and manage the server side aspect of your application.

Authentication/Validation

Authentication in a web application can be quite a daunting subject, and several articles and courses have been written about it. Building on top of the statement above, security in a web application simply falls back to the client having to prove it is allowed to access a resource.

Depending on your framework, this functionality might already be taken care for you. ASP.NET, for example, has a full identity framework built into in. Once you authenticate a user for the first time, the user is provided a session token or key. Subsequent requests are subjected to providing the session token.

In cases where your framework does not provide the functionality for authentication, you can implement a similar scheme by transmitting authorization tokens in cookies or within request headers. ASP.NET implements security in this fashion. Session identifiers are transmitted within a cookie, and the framework takes an action of authenticating a user based off of the session identifier provided. This wheel has been reinvented several times, so it is generally not good advice to try to roll your own authentication platform.

Mapping State to a Session

Assuming a request is authenticated successfully, what devices are available on the server to maintain information that needs to be associated with a particular user session? For example, imagine your application provides a shopping cart experience. The contents of the shopping cart typically would be maintained by the server.

Frameworks such as ASP.NET provide features that will automatically manage session state on your behalf. For the shopping cart example, developers can maintain a session object which holds the cart contents. As items are added, updated, or removed, the code pulls the current shopping cart from the session, updates it, and restores it back in the session.

In frameworks that don't provide this functionality, you can implement a similar technique by keying a session off of the user's session token. Whenever a request from a particular session comes in, and that session is properly authenticated, the request session token can be used to load the session state into memory.

There are different rules of thought about where you should store state on a server.

In Process session storage means that all data corresponding to a session is stored in memory. Access to the session is extremely fast, but it's also volatile. If your server crashes, the session state will fade away.

Also, in a load balanced scenario, In Process cannot work unless your load balancer implements session affinity (meaning the load balancer will direct requests from a single source to one machine in the balanced set).

SQL Databases provide a reliable, persistent data store for session information. In scaled scenarios, node can go up and down continuously while having no effect on a user session.

A major drawback of this approach that it can be costly in terms of performance since you would need to constantly serialize and deserialize data between the server and the database.

In-Memory Databases such as Redis provide the same reliable, persistent data store found with SQL databases. However, data storage is done in memory and using key/value stores. Data access is lightning fast comparably.

In-memory databases are often used for state management in web applications. Compared to in process session storage, in-memory databases work more efficiently in scaled environments. The database acts as a common backplane for all web servers to communicate with.

ASP.NET Session Storage Best Practices

ASP.NET has had a mechanism for session state since its inception. However, in the ever growing world of high performance web applications and cloud environments, some developers have difficulty tuning their applications to take best use of ASP.NET session state.

Here are a couple great tips:

Change your sessions state provider
By default, all ASP.NET applications are configured to use in-process session state. As discussed above, this can run into a multitude of problems when your application is deployed into a scaled, load balanced scenario. It is not uncommon for requests to a load balancer to bounce between various machines in the load balanced set.

In these cases, ASP.NET only stores session state information on the machine which originally set the state value. Requests directed to other machines cannot obtain access to this session state. A recommendation would be to use a different session state provider. ASP.NET currently provides state providers for both SQL Server and Redis.

While Redis is the preferred backplane for session state, developers will also find SQL Server an acceptable alternative to in-process.

Limit what is stored in session state

As a worst case scenario, everything you put into session state will be retrieved and deserialized on every request to the server.

Therefore, it is a good plan to ensure you are not putting too much into session state.

As a common rule of thumb, do not store anything in session state that does not need to persist across requests.

For example, you do not need to persist a user profile within session state, because that information could be more easily obtained via your identity provider.

Keep the data simple

When using session state, it is important to remember that data will need to be serialized and deserialized between your application and the data store.

Complex objects make this process difficult, time consuming, and in some cases, impossible. Stay close to basic data-types.

Do not forget to Abandon

When a user explicitly logs out of your application, make sure to call the Session.Abandon() method. This method call will remove all session information from your data stores, and ensure you are maintaining a clean house.

Do not worry too much though if you forget, because ASP.NET will do this for you automatically after a session timeout. However, a timeout could possibly take up to 15 minutes or more to occur.

Conclusion

Managing the state of your web application is a continuous problem. Applications are becoming more complicated and feature rich.

Clients and servers need to work in harmony to insure a performant, reliable experience for the user. We are lucky to live within a mature ecosystem of frameworks and libraries that aim to solve many of the common problems with building large scale web applications.

As a developer, you can do yourself a favor by following many of the best practices outlined in this article. Your applications will grow and scale gracefully while leaving you sane and free of stress!

.