JavaScript Model in a Nutshell

The purpose of this document is to introduce you to enough of the JavaScript language model as it is typically used in a web application like RWB. We will *not* cover the syntax and semantics of the language here. The course web site provides pointers to other JavaScript resources that do this, of which there are many online. Basic JavaScript is a very easy language to learn, by design. My goal here is to convey what JavaScript code is doing in a web application, why, and how.

JavaScript for Web Clients

The most common usage of JavaScript is to add functionality to web pages, allowing the combination of the web page and the JavaScript code to form a client application within the user's web browser. While this may seem constrained, it is actually a very powerful model since the web page content description language (HTML) is a very general tool for describing content and interface, JavaScript is a full-fledged programming language that can manipulate content and interface, and the browser itself embodies many people-years of effort for effectively rendering content and interface, and executing JavaScript fast. This combination means it is possible to write a complex application without too much pain and expect it to run decently on almost any platform. This is even the case for mobile applications, particularly using features of the HTML5 standard.[1]

Typically a complex application is split into a client application (which runs in the user's device) and a server application (which runs on servers in a data center). The server application is typically not written in JavaScript, but there are advocates for doing so. Most commonly (and the case for this class), the server application is responsible for providing the user with the client application and the initial web page it operates over, and then responding to further requests from the client application as the user interacts with it. The critical data stays within the server application, although the client application may cache it in numerous ways. The server application generally stores the data in a database that runs within the data center. The client application, server application, and the database are often referred to as the presentation, logic, and data tiers of the application.

[1] At the time of this writing many mobile applications are in fact written using much lower level mechanisms than JavaScript for the client side (specifically Java on Android, ObjectiveC on iOS). In part this is because mobile devices are very limited in terms of performance, memory, etc, compared to a laptop or desktop computer. It is controversial whether HTML and JavaScript are, can be, or will be sufficiently efficient. However, be that as it may, for a *database-backed* mobile application, such lower-level client implementation techniques don't really change the concepts behind client/server interaction very much.

The JavaScript language has nothing to do with the Java language. JavaScript is a high-level scripting language with dynamic typing, much more similar in syntax and semantics to other scripting languages like Perl, Python, Ruby, etc, than to Java or C#.[2]

Model/View/Controller in JavaScript in Web Clients

As you can tell from the term "presentation tier", JavaScript in the web client is typically used to implement the user interface of the application. To do so, it uses a common paradigm for graphical user interface systems, but in a very clever way. The paradigm, called Model/View/Controller (MVC), separates the concepts of the underlying data and computation of the application (the Model) from the current presentation of that data and computation to the user (the View), and the mechanisms by which the user can manipulate the data and computation (the Controller).

In a web client, the model is essentially the web page content itself. By this, we don't mean the page as you see it as a user, but rather the description of the page. This is often called the document. Generally, the document is written in a content description language like HTML or some variant of XML. The important thing to note here is that due to the design of the content description language, the document forms a tree.[3] For example, a simple web page document might contain a root node whose children include the title and body of the page. The body might have children that are section headers, paragraphs, images, and so on. A more complex document might also contain nodes that represent user interface components like zoom bars, menus, etc.

In a web client, the view corresponds to the rendering of the current document by the web browser. That is, the view is what the user actually sees, hears, smells, touches, etc. Because the document includes the user interface components, the rendered view causes these components to also be visible to the user. Rendering is a very hard problem, so instead of doing it ourselves, we let the browser developers do it for us. If they invent something clever, we benefit.

In a web client, the controller essentially corresponds to the JavaScript code. The browser interprets the JavaScript code based on user interactions with the rendered controls or other events. Fast interpretation of JavaScript is another hard problem that is the responsibility of the browser developers. When they make it faster, our application also benefits.

The JavaScript code can leverage multiple interfaces and libraries, but the most important is called the Document Object Model (or DOM). The DOM is simply the reprentation of

[2] JavaScript was the name chosen for marketing reasons in the late 1990s. The current committee-approved name is "ECMAScript", but that's such a horrible name that people still generally just call it JavaScript.

[3] Specially, the parse tree or abstract syntax tree, concepts you can learn more about in a compilers course.

the document as a tree and the library functions to search, edit, and otherwise manipulate that tree. You might be wondering where the JavaScript code is stored. It's stored within the document tree! Just as there are nodes for paragraphs, there are also nodes for chunks of JavaScript. As an application developer, we embed our JavaScript source code into the document, either directly, or by reference (via a URL pointing to the code we want to use).

Event-driven Programming Model and Blocking

JavaScript has a very simple programming model that is based in its roots as a tool for constructing user interfaces. In a typical user interface, we are interested in responding to events. A mouse movement, a tap on a touch screen, typing a key on a keyboard are all examples of user interface events. The basic idea of an event-driven programming model, as embodied in JavaScript, is that the programmer associates functions with events of different kinds. There are different terms for this, such as "registering a callback function" or "adding a listener". When an event of a particular kind occurs, the system calls the registered function, passing the particulars of the event as an argument. The registered function then "handles" the event, and returns, ideally as fast as possible. The function can change the state of the program, and it can also create new events and pass them to the system for later processing.

JavaScript allows the programmer to register for numerous kinds of events, including ones that are very specific to the web environment, for example to be run once the page has been loaded, or when a user interface action affects a particular paragraph or image on the page. For example, consider this HTML:

```
<button onclick="foo()">Click Here</button>
```

The HTML will result in the browser rendering a button control. The browser will also track the button control, and when the user clicks on it, the browser will invoke the JavaScript function `foo()`. The event in this instance is the click on the button.

The JavaScript code can also initiate new network interactions (for example, fetching more data from the server or sending some other request to it) which in turn generate new events when the network interaction finishes or has a problem. The typical model is that the JavaScript code asks the browser to do something on its behalf, and when it is complete, to call ("call back") another JavaScript function. As another example, consider this JavaScript, which implements `foo()`:

```
foo = function() {
      setTimeout(bar, 10000);
}

bar = function() {
      alert("10 seconds have passed since you clicked");
}
```

Here, `foo()` registers a callback function for a new event, the passage of 10 seconds of time. The callback function, `bar()` just puts up a message box indicating this has happened. The user clicks on the button and 10 seconds later gets the message box.

Note that this is very different from a typical system call or library call model you learned about in EECS 213 and other programming courses.

A key idea is that JavaScript code should never implicitly wait or "block". If waiting is required to handle an event, we have the browser wait for us. JavaScript that waits implicity, or even that executes for an overly long time in handling an event, is buggy. If you've ever a browser error complaining about an "unresponsive script" on a page, you have seen such a bug.

One reason for JavaScript's programming model is to hide concurrency from user interface programmers. The user, the browser, the logic tier, and the data tier are typically highly concurrent, but JavaScript gives the user interface programmer a completely sequential view – there are no threads, no locking, etc, just sequential event processing. This model works out well for the user interface, but it can make non-user interface code (e.g., networking) more of a challenge to write, and it can make composing code from different sources have unexpected issues.

Extending the Model Back to the Server and Database

In the Model/View/Controller terminology of the previous section, the model constitutes the state of the application, and, in a web application the model is essentially the document tree. For most interesting web applications, things are a bit more interesting in that the model (document tree) in the client application is a *cache* of the state of the server application. The "ground truth" of the state of the entire application (the server application and however many client applications are running) is stored in the database tier, the logic tier mediates between the database tier and clients, and the presentation tier of each client caches state relevant to it. In addition to allowing for the application to share a single ground truth among many users, this structure also allows users to interact with each other.

Putting it Together

We can now describe what happens when a user visits a URL for a web application. The browser requests the URL contents from the server. The server hands back an HTML document that has embedded in it JavaScript code. The HTML typically decorates various elements with callbacks to functions in the JavaScript code. The browser parses the HTML into a DOM tree, and then renders the page. It also executes any JavaScript code that is outside of a function or that is bound to a document loading event. This JavaScript then registers callbacks for other events of interest, for example network, localization, or timing events. The user interacts with the application's user interface via the browser, which creates events which the browser handles by calling the matching registered callbacks. The JavaScript code in these callbacks then initiates other computation or operations, for example fetching or supplying data over the network via

another URL, and modifies the DOM tree accordingly. Some of the network operations, which are handled by the logic tier, cause data to flow to/from the data tier, changing the ground truth state of the overall application. When the DOM tree is modified, the browser updates its rendering of the page, which the user then sees. This process just continues, event by event, until the user goes to a different URL.

Higher-level Frameworks

We can write a web application by hand at a low level. For example, RWB has the presentation tier written in JavaScript and HTML, the logic tier in Perl, and the data tier in SQL. This is valueable for understanding how this works, particularly the logic tier/data tier interaction via SQL, which is most relevant for a databases course. However, in many cases web applications follow common forms. There exist higher-level frameworks to allow fast development of such applications. The general idea is that these frameworks let you write the application using a single, simpler language and/or by filling out templates. The framework then generates the relevant JavaScript, HTML, SQL, Perl, etc, for the programmer.