PATTERNS OF

MODERN Web Apps

KEN CHAU

Patterns of Modern Web Applications with Javascript

Ken Chau

This book is for sale at http://leanpub.com/patterns-of-modern-web-applications

This version was published on 2015-07-10

Contents

Introduction

Who This Book is For

Web development technology has shifted from server to client. This allowed for highly interactive applications to be developed in the style of Single Page Applications (SPA). Traditional server-rendered pages are becoming more and more extinct as Web browsers and hardware are getting faster and faster. As developers develop these complex applications, new patterns are emerging that would solve some problems presented by the "traditional" SPA's.

I have written this book for developers like me. I'm assuming that you come from a background of Web development. This should probably not be the book you use to start learning Web development. I hope to share lessons learned from building large and complex Web applications. This book captures the common design and architecture pattern found in modern Web development today.

In this book, I am focusing on the front-end Web development stack written in Javascript. While developing complex applications, I look for frameworks and libraries from the Open Source community. The good news is that you can find just about anything in these package repositories. The bad news is that you can find just about anything in these package repositories. How do you judge good from bad; the signal from noise?

Wholeheartedly, I believe that none of the concepts contained in these packages are brand new. In fact, one of the most useful way to filter out the bad packages is to take the time to understand the underlying design patterns. Understanding the underlying principles of frameworks and libraries will help weed out the ones that are short-sighted and based on unsound principles. It is my hope that you will be equipped to discern the signal from the noise.

Overview

I have structured this book in three sections. The first section talks about the Javascript host environments. We will discuss the different hosts that Javascript can be run in modern Web stacks. In addition to this, we will discuss the topic of modularity. The reason why this comes first is because everything else that comes afterwards assume that reader has a firm grasp of the nuances of running Javascript on the server versus on the client. Also, almost all modern Javascript libraries and frameworks are consumed as packaged modules, so it is foundational to have a firm grasp of the modularity patterns before moving on to the other sections.

The second section dives into the different well-established and emerging design and architectural patterns of client and server side applications. We will take a whirlwind tour of many areas that make up a modern Web application. Everything from DOM rendering, to data flow, to testing, to integrations with server-side application frameworks.

The last section will highlight some of the most important architectural and design patterns of delivering a fast front-end application. We will go over the challenges and solutions of making your applications screaming fast. We will even take a detour from the request-reply pattern and discuss solutions with real-time communication technology.

Acknowledgements

TODO: Support me while I'm writing this book, and I'll put your name in the Acknowledgement section!

Help and Support

TODO: It is my hope to create a website dedicated to this book to assist readers in learning the concepts here better

Part 1. Javascript Environments and Modularity

Chapter One: Javascript Runtime Environments

Versions of Javascript

Developers need to throw out their preconceived notion about Javascript of being only appropriate for small DOM manipulation is outdated. This little language has somehow snuck into every single platform unnoticed. The idea of a Javascript engine running outside of the context of the DOM was not a new idea. Back in 1996, Microsoft had packaged this in the form of JScript. However, Microsoft platform developers mostly took their bets on VBScript on Windows for scripting needs.

Not many developers took Javascript seriously as a platform language until the amazing V8 engine came on the scene outside of the Google Chrome browser. Node.js was created as a package of the V8 engine, an event loop, and a set of asynchronous event I/O APIs. With this, Ryan Dahl from Joyent, set off a chain reaction that is manifesting in the vibrant Javascript community around the world.

Therefore, a developer will find Javascript at the heart of a DOM manipulating Web application, an evented Web service, high quality WebGL based 3D game engine (Unreal Engine[1]), and even controlling robots[2]. We will focus on how Javascript can be used on the client (browsers) and on the server (for writing Web services).

Start the Engines!

V8 inside Chrome

Found in the popular Google Chrome browser, the V8 engine is versatile and cross platform. It is open sourced, so that it allowed third party developers to even package this in other platform, such as Node.js and Phantom.js (the headless WebKit based browser). It is found in the Google Chrome browser, both on the desktop and on Android. It is versatile enough to run both on X86 and ARM processor architectures. As of the writing of this book, the V8 is an ECMAScript 5 compliant Javascript engine. This means that it has full support for Object properties among other features.

Certain aspects about the V8 Javascript engine that should be noted are:

1. Hidden anonymous classes
2. Dynamic machine code generation
3. Garbage collection

[1]https://www.unrealengine.com/html5/

[2]http://nodebots.io/

Hidden Anonymous Classes Given the dynamic nature of the Javascript language, types can be created on the fly. To ensure good performance in property lookup, V8 implements a dynamically generated hidden class everytime an object changes its property shape. That is to say, if a new property has been added to an object or if the type of the object had been changed, that would cause the dynamic compiler to kick in and create yet another instance of a hidden class. Bear this in mind when creating complex applications for the sake of saving memory and having good performance.

Dynamic Machine Code Generation It is important to understand that V8 is dynamically compiling Javascript source code into native machine code, rather than an intermediate form. This is unlikely other languages like C# or Java. The compilation is happening even as the code is running (just-in-time). Dynamically generated hidden classes are used in retrieving properties from hidden classes. Since the optimizer in V8 is able to inline the code to access properties from hidden classes, the JIT compiler is quite fast. This is another reason to avoid changing the property types once an object has been defined.

Garbage Collection Garbage collection in V8 causes the script execution to pause quickly. V8 tries to compensate for this pause by allowing partial clean up from the Garbage Collection. Similar to the JVM garbage collection, it also has the concept of generations. The heap is divided then by objects that are new and objects that survived a garbage collection.

Note: Bibliography - V8 development site[3]

The Mozilla Monkeys

Since the introduction of the venerable SpiderMonkey Javascript engine, Mozilla has released several other Javascript engines during the development of the Firefox browser. The current Javascript engine in the latest Firefox version is the IonMonkey. This particular monkey is the focus of this section. We assume this because Firefox and Chrome have taken the ever-green approach with their releases.

The main goals of this IonMonkey engine are to have a well organized, easily optimizable engine as well as having optimizations that generate fast code. Unlike the V8 engine, the IonMonkey does actually generate byte-code. While V8 relied on its clever use of hidden classes and dynamic machine code generation, IonMonkey added an intermediate form (both the machine independent MIR and machine dependent LIR byte-code) precisely so that a separate optimization can be implemented. Optimizations are applied at the machine independent phase with the MIR byte-code. It is worth noting that many database engines take this engine as their Javascript engine of choice, such as Riak, MongoDB, and CouchDB.

[3] https://developers.google.com/v8/

Chakra

Chakra has its origins in that 1996 JScript engine and has been modified from version to version upgrade of Internet Explorer. Since Internet Explorer had not been an evergreen browser in the past, the engine versions are closely tied to major versions of the browser. The classic jscript engine is included in IE6. The ActiveScript jscript engine is included in IE9. And finally the new embeddable Chakra engine are available if the host OS has IE11 installed. You will find more about embedding the new Chakra engine in the following section about server Javascript environments.

This engine has historic value, but the upcoming Windows 10 platform will bring this engine to the frontline more. Beyond strictly being a browser Javascript engine, this particular engine will be incorporated as a runtime target for Window's Universal App platform. One of the most interesting things about Chakra is that while embedding, the developer can choose to embed the legacy jscript9.dll or the engine used in the Microsoft Edge browser called chakra.dll.

Some quick points about Chakra's future is that it no longer deals with document modes. It does not support ActiveX objects, so the attack surface of this Javascript engine (and the Microsoft Edge browser) is reduced. It also means that legacy line of business apps that are written on ActiveX stack will need to be rewritten as Javascript applicatoins. The Microsoft Edge browser is continuously updated, so new features are expected to be added to the engine much like Chrome and V8. The browser, without the legacy cruft, is much faster. This is, in part, due to several advancements in the engine.

The Chakra switches between running interpreted code to running Just-in-Time (JIT) compiled machine code with optimizations. It usually begins running the interpreted code and moves onto using the optimizing JIT machine code when the compilation is completed. This process is handled by the engine automatically and the goal is to keep running more and more compiled machine code to make the engine fast. The JIT compilation process occurs in another thread traditionally in the legacy Chakra implementation. When Edge Chakra was developed, it started taking advantage of the mult-core nature of modern hardware. This allows the JIT compiler to run on multiple cores. One of the issues with the legacy Chakra engine is that Javascript might run into strange cases where the JIT'ed code no longer is accurate in a certain point in time. At that point, the interpreter has to kick back in and slow down execution. Edge Chakra solves this problem by including a non-optimizing Simple JIT compiler that can generate machine code without the expensive optimization step. This makes the engine run even faster when the runtime requires the engine to switch out of the optimized mode.

Roughly, the Microsoft Edge team has seen a 30% performance boost when using the different flavors of the Chakra engine to execute the Typescript compiler. This is a sizable codebase with sufficient complexity that really measures the improvements included in the Edge Chakra implementation. Some additional data point, with the currently version of the Octane benchmark running on my test laptop, the Google Chrome v43 vs Microsoft Edge was 22197 vs 24031, respectively. Given that Node.js currently packages an outdated version of V8 engine, it can only mean that the Chakra engine is poised to be a strong contender in terms of performance for desktop or server application in the near future.

Taken from the MSDN site about Javascript Runtime Hosting[4], the following .NET will import the Chakra engine:

```
/* for the legacy jscript9.dll */
[DllImport("jscript9.dll")]
public static extern JsErrorCode JsCreateRuntime(
    JsRuntimeAttributes attributes,
    JsRuntimeVersion version,
    JsThreadServiceCallback callback,
    out JsRuntimeSafeHandle runtime
);

[DllImport("jscript9.dll")]
public static extern JsErrorCode JsCreateContext(
    JsRuntimeSafeHandle runtime,
    IDebugApplication debugApplication,
    out JsContextRef newContext
);

[DllImport("jscript9.dll")]
public static extern JsErrorCode JsStartDebugging(
    IDebugApplication debugApplication,
);

/* for the chakra.dll like in Edge */
[DllImport("chakra.dll")]
public static extern JsErrorCode JsCreateRuntime(
    JsRuntimeAttributes attributes,
    JsThreadServiceCallback callback,
    out JsRuntimeSafeHandle runtime
);

[DllImport("chakra.dll")]
public static extern JsErrorCode JsCreateContext(
    JsRuntimeSafeHandle runtime,
    out JsContextRef newContext
);

[DllImport("chakra.dll")]
public static extern JsErrorCode JsStartDebugging();
```

[4]https://msdn.microsoft.com/en-us/library/dn903710(v=vs.94).aspx

To use the Javascript Runtime, you should look at the MSDN's documentation on the Javascript Runtime API[5] for further details on using the engine inside your .NET application.

Node.js

Node.js has depth beyond the scope of this book. It wouldn't take you very long to find scores of books around that has more comprehensive coverage of this Javascript environment. No wonder, its popularity comes from its ease of use and the amount of built-in features are unparalleled. It also raised an important point about Javascript being a capable server and desktop side language. It is deceptively easy to get started with Node.js and get lost in its many wonderful packages via the excellent Node Package Manager (npm). Many of the packages provide features that are simply unmatched in other communities.

Node.js is a Javascript environment that runs on a slightly older and stable version of the V8 engine. In addition to being able to run plain Javascript, Node.js provides standard library to allow developers take advantage of the underlying platform. For example, the "fs" file system package allows the Node.js program to read and write arbitrary files in the file system. This simply isn't available from the Javascript environment inside a Web browser.

Evented I/O

It would be incomplete in an introduction of the Node.js environment without mentioning that it was originally intended to be written for writing Evented I/O applications with minimal amount of code.

What does it mean to be Evented I/O? To explain this, we need to look at other standard implementations of Web servers. The traditional and easiest way to write a Web server is to put that a single socket connection on each thread. This way, it can handle tens of connections per second. Tens per second was fine in a small project, but in the modern world, services handle thousands or even tens of thousands of connections per second. To handle this, modern Web servers have switched to using libevent or IO Completion Ports on the Windows side. What that means is that these low level libraries have taken care of allowing only few threads in a thread pool to process the many requests. This model is the best way to write scalable services today.

Node.js is opinionated in that its engine, V8, expects services to be written against a single thread. The main defense it has against long-running tasks from these requests (such as disk access or network I/O) is its emphasis on asynchronous APIs. Almost everything in the standard Node.js library have a callback in their signatures. With this, Node.js continues to accept new requests and only respond when the long running tasks have completed.

Edge.js

Edge.js is a connection between the .NET framework and Node.js engine. There are two ways to use edge.js. Start by installing it as a npm package:

[5]https://msdn.microsoft.com/en-us/library/dn249536(v=vs.94).aspx

```
npm install edge
```

Then, you can start using C# code and libraries inside your Node.js applications.

```
var edge = require('edge');

var helloWorld = edge.func(function () {/*
    async (input) => {
        return ".NET Welcomes " + input.ToString();
    }
*/});

helloWorld('JavaScript', function (error, result) {
    if (error) throw error;
    console.log(result);
});
```

A few use cases immediately come to mind. There are Windows specific APIs that can only be accessed from a Node.js application this way. Perhaps utilizing the excellent Express.js microframework, a lightweight REST API can be created to convert any Word documents to PDFs. Traditionally, one would have to implement everything inside .NET technologies, including the Web service. Now, Node.js code can call C# code in process in a pattern much like the Java JNI or P/Invoke. It is worth noting that the C# lambda wrapped inside the JS comment contains the keyword: async. This means that the single-threaded JS application can call into C# function that executes in a separate thread.

The other interesting thing that can be done with Edge.js is that C# applications can call into Node.js packages:

```
using System;
using System.Threading.Tasks;
using EdgeJs;

class Program
{
    public static async void Start()
    {
        var func = Edge.Func(@"
            return function (data, callback) {
                callback(null, 'Node.js welcomes ' + data);
            }
        ");
        Console.WriteLine(await func(".NET"));
    }

    static void Main(string[] args)
    {
        Task.Run((Action)Start).Wait();
    }
}
```

It is no secret that the NPM package repository is filled with many packages that are simply not available in the public Nuget repository. One application of Edge.js running Node.js code inside a C# application may be to utilize the excellent CSS structural minifier written by Yandex developers. These packages traditionally had to be ported to C#, or use another bridging technology to run the Javascript. However, none of these other solutions actually embeds the Node.js process, so the NPM external modules cannot be loaded directly. The bundled Javascript would have to be created with a bundling tool, such as Browserify before using inside those other solutions. Edge.js takes care of all of these by using Node.js itself.

Chapter Two: Patterns of Modularity

The Need for Modularity

Most computer science problems ultimately are solved with a divide and conquer strategy. When an application becomes big enough, the developer begins searching for ways to divide the application into smaller and more main table pieces. Different programming languages have solved this in different ways. JavaScript was born out of a need to add interactivity to Web sites originally, and it was until quite recently that it was deemed fast enough to become a full-on general purpose language capable of creating complex client and server side applications

Because of the organic nature of this growth, the most deployed versions of the JavaScript language, ECMAScript 5, does not have any built in facility for this. The most fundamental way to break a problem into pieces in ES5 is through functions and objects. New JavaScript developers will simply put these into different script files and simply include these as SCRIPT tags. This is an acceptable practice until the code base grows to a point where certain amount of data encapsulation is needed.

Third party components come in different ways. Initially, they polluted the global namespace with the functions provided. Then, library developers began putting their library code inside named objects acting as namespaces. JavaScript does have the notion of prototypical inheritance. However, if used too liberally, even property lookups are costly. It is because there is no notion of a class being blueprints for instantiating objects.

JavaScript, while a simplistic languages, actually has a very powerful way to express scope that can replace these methods of encapsulation of data and functions. This is known as closure. When the idea of running JS code on the server was being developed, a standard emerged on modularizing application code. This is known as the CommonJS module format. With this, server applications can be composed from many interdependent modules. Developers no longer needed to worry about how the modules are loaded, they simply declare dependencies as objects and use those dependent object.

While the new ECMAScript standard actually has support for modules, the most deployed version still needs hints from the developer to understand modularity.

CommonJS

As mentioned, this format is popularized by Node.js. it has the distinct advantage of having a massive number of modules ready to be consumed on the public Node Package Manager (NPM) repository. npm is a package managernthat comes with Node.js's installation, so it is really easy to get started using this.

To consume packages from npm, the project root folder needs to have a package.json. As implied by the file extension, it simply is a declarative json file, and nothing more. To get scaffold a project, simply issue an `npm init` call.

```
npm install
```

It asks you a few questions about your project and creates the package.json. this file is the cornerstone of how the project expresses its dependencies. To add a dependent component is simple:

```
npm install some-package
```

It is not the goal of this book to help you master a command line interface. Go to npm[6] for more information.

The commonjs pattern relies on a very special function called require(). This function is used like this:

```
var foo = require("foo");
foo.doFooThings();
```

This simply instructs the Node.js environment to load in the module named "foo" into the score and assign it as a variable named "foo". Node.js also assumes that the dependent module of "foo" had been installed before. The installed dependent modules are stored locally in a directory called `node_modules`. Thus Node.js actually will crawl up the subdirectory recursively to import all the dependent modules with the help of this function.

Now this is enough information for a developer to begin consuming package published on the npm registry. To begin sharing packages with others, we have to look at another construct called `export`.

Below is an example of a very simple module:

```
var foo = require("foo");
function bar() {
    var f = foo.doFooThings();
    return f * 5;
}
exports = bar;
```

To export an object (a function in this case), assign the object into the exports property from the global scope. In this way, we have learned how to consume and produce modules in JavaScript. Using this simple but powerful pattern, complex solutions can be composed from small modules.

There is a distinct disadvantage of this modularity pattern, however. The main problem is that this cannot directly be used in a Web browser. This is mainly due to the fact that JavaScript files are downloaded over the network. Since networks are generally much less reliable than file systems, an asynchronous module loading mechanism is required.

[6]http://npmjs.org

Asynchronous Module Definition

Recognizing this issue, developers tweaked the CommonJS and have the pattern some asynchronous characteristics right from the definition of the modules themselves. This pattern places the actually module code inside a callback closure function forcing the code to be run only of an asynchronous script loading process has finished.

Using this pattern makes the HTML to be more concise reducing the number of script tags to be specified. Script loaders have been written to load dependent scripts. One of the most famous AMD script loaders is the require.js (do not confuse this with the require function from Commonjs pattern).

Here is an example of a simple module that consumes another module as a dependency:

```
define(['foo'], function(foo) {
    return function bar() {
        var f = foo.doFooThings();
        return f * 5;
    }
});
```