Advertise Here

# Important Considerations When Building Single Page Web Apps

Pavan Podila on Jan 21st 2013 with 51 Comments

## Tutorial Details

- 
- **Difficulty**: Intermediate
- **Completion Time**: 30 Minutes

View post on Tuts+ Beta**Tuts+ Beta** is an optimized, mobile-friendly and easy-to-read version of the Tuts+ network.

Single page web applications – or SPAs, as they are commonly referred to – are quickly becoming the de facto standard for web app development. The fact that a major part of the app runs inside a single web page makes it very interesting and appealing, and the accelerated growth of browser capabilities pushes us closer to the day, when all apps run entirely in the browser.

Technically, most web pages already are SPAs; it's the complexity of a page that differentiates a *web page* from a *web app*. In my opinion, a page becomes an app when you incorporate workflows, CRUD operations, and state management around certain tasks. You're working with a SPA when each of these tasks take place on the same page (using AJAX for client/server communication, of course).

Let's start with this common understanding, and dive into some of the more important things that should be considered when building SPAs.

---

There are numerous points to consider before building a new app; to make matters worse, the expansive web development landscape can be intimidating at the outset. I have been in those unsettling shoes, but fortunately, the past few years have brought consensus on the tools and techniques that make the application development experience as enjoyable and productive as possible.

Most apps consist of both client and server-side pieces; although this article focuses mostly on the client-side portion of an app, I'll provide a few server-side pointers toward the end of this article.

There is a colorful mix of technologies on the client-side, as well as several libraries and practices that enable a productive app development experience. This can be summarized, using the following word cloud.

I will expand on each of the points above in the following sections.

---

## Picking an Application Framework

There are an abundance of frameworks to choose from. Here's but a handful of the most popular:

- Backbone
- CanJS
- SpineJS
- BatmanJS

- EmberJS
- AngularJS
- Meteor

Choosing a framework is easily one of the most important choices you will make for your app. Certainly, you'll want to choose the best framework for your team and app. Each of the above frameworks incorporate the MVC design pattern (in some form or another). As such, it's quite common to refer to them as MVC frameworks. If we had to order these frameworks on a scale of complexity, learning curve and feature set, *from left to right*, it might look like:

Although dissimilar in their implementation and level of sophistication, all the aforementioned frameworks provide some common abstractions, such as:

> Just looking at the past five years, there has been an explosive growth in libraries, tools and practices.

- **Model**: a wrapper around a JSON data structure with support for property getters/setters and property change notification.
- **Collection**: a collection of models. Provides notifcations when a model is added, removed, or changed in the collection.
- **Events**: a standard pattern to subscribe to and publish notifications.
- **View**: A backing object for a DOM fragment with support for listening to DOM events relative to the DOM fragment. The View has access to the corresponding Model instance. In some frameworks, there is also a **Controller** that orchestrates changes between the View and Model.
- **Routing**: Navigation within an app via URLs. Relies on the browser history API.
- **Syncing**: Persisting model changes via Ajax calls.

More advanced frameworks, like CanJS, BatmanJS, EmberJS and AngularJS, expand on these basic features by providing support for automatic data–binding and client–side templates. The templates are data–bound and keep the view in sync with any changes to the model. If you decide to pick an advanced framework, you will certainly get a lot of out–of–the–box features, but it also expects you to build your app in a certain way.

Of all the previously listed frameworks, Meteor is the only full–stack framework. It provides tools not only for client–side development, but it also provides you with a server–side piece, via NodeJS, and end–to–end model synchronization, via MongoDB. This means that, when you save a model on the client, it automatically persists in MongoDB. This is a fantastic option, if you run a Node backend and use MongoDB for persistence.

Based on the complexity of your app, you should pick the framework that makes you the most productive. There certainly will be a learning curve, but that's a one–time toll you pay for express–lane development. Be sure to carve out some time to evaluate these frameworks, based on a representative use–case.

**Note**: If you want to learn more about these frameworks from their creators, listen to these videos from ThroneJS.

---

## Client-Side Templates

The most popular JavaScript–based templating systems are Underscore templates and Handlebars.

> Some of the advanced frameworks from the previous section offer built–in templating systems.

For example, EmberJS has built–in support for Handlebars. However, you do have to consider a templating engine if you decide to use a lean framework, such as Backbone. Underscore is an excellent starting point, if you have limited templating requirements. Otherwise, Handlebars works great for more advanced projects. It also offers many built–in features for more expressive templates.

If you find that you require a large number of client–side templates, you can save some computation time by pre–compiling the templates on the server. Pre–compilation gives you plain JavaScript functions that you invoke to improve the load time of the page. Handlebars supports pre–compilation, making it worth the time and effort to fully explore.

ExpressJS users can even use the same templating engine on the client as on the server, giving you the benefit of sharing your templates between both the client and server.

---

## Modular Development

> Using a preprocessor requires an extra step in your build process.

JavaScript code is traditionally added to the page, via the `<script />` element. You typically list libraries and other dependencies first, and then list the code that references those dependencies. This style works well, when you only need to include a few files; however, it will quickly become a nightmare to maintain, as you include additional scripts.

One solution to this problem is to treat each script file as a *Module*, and identify it by a name or relative file path. Using these semantics, and with the support of libraries, like RequireJS and Browserify, you can build your app using a module–based system.

The module thus becomes a way to identify the functionality within the app. You can organize these modules, using a certain folder structure that groups them based on a particular feature or functionality. Modules help in managing your application's scripts, and it also eliminates global dependencies that must be included with `<script />` elements before the application scripts. For libraries that are not AMD compatible, RequireJS offers a *shim* feature that exposes non–AMD scripts as modules.

> There are currently two types of module–based systems: AMD (Asynchronous Module Definition) and CommonJS.

In AMD, each module contains a single top-level `define()` statement that lists all required dependencies, and an export function that exposes the module's functionality. Here's an example:

```
define([
    // listing out the dependencies (relative paths)
    'features/module/BaseView',
    'utils/formatters'
], function(BaseView, formatters) { // Export function that takes

    // do something here

    // An explicit require
    var myModule = require('common/myModule');

    // Object exposing some functionality
    return { ... };
});
```

CommonJS module names are based on either a relative file path or a built-in module lookup process. There is no `define()` function in any module, and dependencies are explicitly stated by calls to `require()`. A module exposes its functionality, via the `module.exports` object, which each module automatically creates. Here's a CommonJS example:

```
var fs = require('fs'), // standard or built-in modules
    path = require('path'),
    formatters = require('./utils/formatters'); // relative file pa

// Export my code
module.exports = { ... };
```

The CommonJS module style is more prevalent in NodeJS applications, where it makes sense to skip the call to `define()` call – you are working with a file-system based module lookup. Interestingly, you can do the same in a browser, using Browserify.

---

## Package Management

Performance should be on your mind as you build and add features to your app.

Most apps have at least one dependency, be it a library or some other third party piece of code. You'll find that you need some way to manage those dependencies as their number increases, and you need to insulate yourself from any breaking changes that newer versions of those dependencies may introduce.

Package management identifies all the dependencies in your app with specific names and versions. It gives you greater control over your dependencies, and ensures that everyone on your team is using an identical version of the library. The packages that your app needs are usually listed in a single file that contains a library's version and name. Some of the common package managers for different tech stacks are:

- Linux: Aptitude
- .NET: Nuget
- PERL: CPAN
- Ruby: Gems
- PHP: Composer
- Node: NPM
- Java: Maven and Gradle

Although package management is more of a server-side ability, it's gaining popularity in client-side development circles. Twitter introduced Bower, a browser package manager similar to NPM for Node. Bower lists the client-side dependencies in `component.json`, and they are downloaded by running the `bower` CLI tool. For example, to install jQuery, from the Terminal, you would run:

```
bower install jquery
```

The ability to control a project's dependencies makes development more predictable, and provides a clear list of the libraries that an app requires. If you consider consolidating your libraries in the future, doing so will be easier with your package listing file.

## Unit and Integration Testing

It goes without saying that unit testing is a critical part of app development. It ensures that features continue to work as you refactor code, introduce libraries, and make sweeping changes to your app. Without unit tests, it will prove difficult to know when something fails, due to a minor code change. Coupled with end–to–end integration testing, it can be a powerful tool, when making architectural changes.

On the client–side, Jasmine, Mocha and Qunit are the most popular testing frameworks. Jasmine and Mocha support a more Behavior–Driven Development (BDD) style, where the tests read like English statements. QUnit, on the other hand, is a more traditional unit testing framework, offering an assertion–style API.

> Jasmine, Mocha or Qunit run tests on a single browser.

If you want to gather test results from multiple browsers, you can try a tool like Testacular that runs your tests in multiple browsers.

To take testing the whole nine yards, you'll likely want to have integration tests in your app, using Selenium and Cucumber/Capybara. Cucumber allows you to write tests (aka *features*) in an English–like syntax, called *Gherkin*, which can even be shared with the business folks. Each test statement in your Cucumber file is backed by executable code that you can write in Ruby, JavaScript or any of the other supported languages.

Executing a Cucumber feature file runs your executable code, which in turn tests the app and ensures that all business functionality has been properly implemented. Having an executable feature file is invaluable for a large project, but it might be overkill for smaller projects. It definitely requires a bit of effort to write and maintian these Cucumber scripts, so it really boils down to a team's decision.

---

## UI Considerations

Having a good working knowledge of CSS will help you achieve innovative designs in HTML.

The UI is my favorite portion of an app; it's one of the things that immediately differentiates your product from the competition. Although apps differ in their purpose and look and feel, there are a few common responsibilities that most apps have. UI design and architecture is a fairly intensive topic, but it's worth mentioning a few design points:

- **Form Handling**: use different input controls (numeric inputs, email, date picker, color picker, autocomplete), validations on form submit, highlight errors in form inputs, and propagating server-side errors on the client.
- **Formatting**: apply custom formats to numbers and other values.
- **Error Handling**: propagate different kinds of client and server errors. Craft the text for different nuances in errors, maintain an error dictionary and fill placeholders with runtime values.
- **Alerts and Notifications**: tell the user about important events and activities, and show system messages coming from the server.
- **Custom Controls**: capture unique interaction patterns in the app as controls that can be reused. Identify the inputs and outputs from the control without coupling with a specific part of the app.
- **Grid System**: build layouts using a grid system, like Compass Susy, 960gs, CSS Grid. The grid system will also help in creating responsive layout for different form factors.
- **UI Pattern Library**: get comfortable with common UI patterns. Use Quince for reference.
- **Layered Graphics**: understand the intricacies of CSS, the box models, floats, positioning, etc. Having a good working knowledge of CSS will help you achieve innovative designs in HTML.
- **Internationalization**: adapt a site to different locales. Detect the locale using the `Accept-Language` HTTP header or through a round-trip to gather more info from the client.

## CSS Preprocessors

CSS is a deceptively simple language that has simple constructs. Interestingly, it can also be very unwieldy to manage, especially if there are many of the same values used among the various selectors and properties. It’s not uncommon to reuse a set of colors throughout a CSS file, but doing so introduces repetition, and changing those repeated values increases the potential for human error.

banglalink
proudly towards
we are relentlessly working towards making a difference in your life everyday and this time with 3G.
3G

CSS preprocessors solve this problem and help to organize, refactor and share common code. Features, such as variables, functions, mixins and partials, make it easy to maintain CSS. For example, you could store the value of a common color within a variable, and then use that variable wherever you want to use its value.

> Using a preprocessor requires an extra step in your build process: you have to generate the final CSS.

There are, however, tools which auto–compile your files, and you can also find libraries that simplify stylesheet development. SASS and Stylus are two popular preprocessors that offer corresponding helper libraries. These libraries also make it easy to build grid–based systems and create a responsive page layout that adapts to different form factors (tablets and phones).

Although CSS preprocessors make it easy to build CSS with shared rules, you still have the responsibility of structuring it well, and isolating related rules into their own files. Some principles from SMACSS and OOCSS can serve as a great guide during this process.

> Scalable and Modular Architecture for CSS is included, as part of a Tuts+ Premium membership.

---

## Version Control

If you know a hip developer, then you're probably aware that Git is the reigning champion of all version control systems (VCS). I won't go into all the details of why Git is superior, but suffice it to say that branching and merging (two very common activities during development) are mostly hassle free.

A close parallel to Git, in terms of philosophy, is Mercurial (hg)—although it is not as popular as Git. The next best alternative is the long-standing Subversion. The choice of VCS is greatly dependent on your company standards, and, to some extent, your team. However, if you are part of a small task force, Git is easily the preferred option.

---

## Browser Considerations

> It goes without saying that unit testing is a critical part of app development.

There are a variety of browsers that we must support. Libraries, like jQuery and Zepto, already abstract the DOM manipulation API, but there are other differences in JavaScript and CSS, which require extra effort on our parts. The following guidelines can help you manage these differences:

- Use a tool, like Sauce Labs or BrowserStack to test the website on multiple browsers and operating systems.
- Use polyfills and shims, such as es5shim and Modernizr to detect if the browser supports a given feature before calling the API.

- Use CSS resets, such as Normalize, Blueprint, and Eric Myer's Reset to start with a clean slate look on all browsers.
- Use vendor prefixes (`-webkit-`, `-moz-`, `-ms-`) on CSS properties to support different rendering engines.
- Use browser compatibility charts, such as findmebyIP and canIuse.

Managing browser differences may involve a bit of trial and error; Google and StackOverflow can be your two best friends, when you find yourself in a browser-induced jam.

---

## Libraries

There are a few libraries that you might want to consider:

- **Visualizations**: Spark lines, Highcharts, D3, xCharts, and Raphaël.
- **Formatting**: numeraljs, accountingjs, and moment.
- **Controls**: Bootstrap, jQuery UI, and select2.
- If you decide to use **BackboneJS**, you can look at Backbone.Marionette, which provides several helper utilities to make your development go faster.
- **Helpers**: Underscore, Sugar, es5shim, Modernizr, and Html5 Boilerplate.

---

## Minification

Before deploying your application, it's a good idea to combine all of your scripts into a single file; the same can be said for your CSS. This step is generally referred to as minification, and it aims to reduce the number of HTTP requests and the size of your scripts.

You can minify JavaScript and CSS with: RequireJS optimizer, UglifyJS, and Jammit. They also combine your images and icons into a single sprite sheet for even more optimization.

**Editor's Note:** I recommend that you use Grunt or Yeoman (which uses Grunt) to easily build and deploy your applications.

## Tools of the Trade

> Twitter introduced Bower, a browser package manager similar to NPM for Node.

I would be remiss if I did not mention the tools for building SPAs. The following lists a few:

- **JsHint** to catch lint issues in your JavaScript files. This tool can catch syntactic issues, such as missing semicolons and enforcing a certain code style on the project.
- Instead of starting a project from scratch, consider a tool, such as **Yeoman** to quickly build the initial scaffolding for the project. It provides built-in support for CSS preprocessors (like SASS, Less and Stylus), compiling CoffeeScript files to JavaScript and watching for file changes. It also prepares your app for deployment by minifying and optimizing your assets. Like Yeoman, there are other tools to consider, such as MimosaJS and Middleman.
- If you're looking for a make-like tool for JavaScript, look no further than **Grunt**. It is an extensible build tool that can handle a variety of tasks. Yeoman uses Grunt to handle all of its tasks.
- **Nodemon** for auto-starting a Node program each time a file changes. A simliar tool is forever.
- **Code editors**, such as Sublime Text, Vim, and JetBrains WebStorm.
- **Command line** tools ZSH or BASH. Master the shell because it can be very, effective especially when working with tools like Yeoman, Grunt, Bower and NPM.
- **Homebrew** is a simple package manager for installing utilities.

## Performance Considerations

CSS preprocessors make it easy to build CSS with shared rules.

Rather than treating this as an after–thought, performance should be on your mind as you build and add features to your app. If you encounter a performance issue, you should first profile the app. The Webkit inspector offers a built–in profiler that can provide a comprehensive report for CPU, memory and rendering bottlenecks. The profiler helps you isolate the issue, which you can then fix and optimize. Refer to the Chrome Developer Tools for in–depth coverage of the Chrome web inspector.

Some common performance improvements include:

- Simplify CSS selectors to minimize recalculation and layout costs.
- Minimize DOM manipulations and remove unnecessary elements.
- Avoid data bindings when the number of DOM elements run into hundreds.
- Clean up event handlers in view instances that are no longer needed.
- Try to generate most of the HTML on the server–side. Once on the client, create the backing view with the existing DOM element.
- Have region–specific servers for faster turn around.
- Use CDNs for serving libraries and static assets.
- Analyze your web page with tools like YSlow and take actions outlined in the report.

The above is only a cursory list. Visit Html5Rocks for more comprehensive performance coverage.

---

## Auditing and Google Analytics

If you plan on tracking your app's usage or gathering audit trails around certain workflows, Google Analytics (GA) is probably your best solution. By including a simple GA script on each page with your tracking code, you can gather a variety of your app's metrics. You can also set up goals at the Google Analytics website. This fairly extensive topic is worth investigating, if tracking and auditing is an important concern.

---

## Keeping up With the Jones

The world of web development changes quickly. Just looking at the past five years, there has been an explosive growth in libraries, tools and practices. The best way to keep tabs on the web's evolution is to subscribe to blogs (like this one), newsletters and just being curious:

- Read How Browsers Work.
- Learn the platform – Web Platform, a project sponsored by the major vendors.
- Subscribe to Nettuts+!
- Subscribe to Html5 Weekly, JavaScript Weekly, and Web Design Weekly.
- Attend conferences, like JSConf, Html5DevConf, FluentConf, local user groups and conferences.
- Visit Html5 Rocks.
- Watch recorded videos from conferences, which are freely available online at YouTube, Blip.tv, Confreaks.
- Explore GitHub.

---

## Operations Management

The client-side, although looking like a large piece of the stack, is actually only half of the equation. The other half is the server, which may also be referred to as operations management. Although beyond the scope of this article, these operations can include:

- continuous integration, using build servers such as TeamCity, Jenkins, and Hudson.
- persistence, data redundancy, failover and disaster recovery.
- caching data in-memory and invalidating the cache at regular intervals.
- handling roles and permissions and validating user requests.
- scaling under heavy load.
- security, SSL certificates, and exploit-testing.

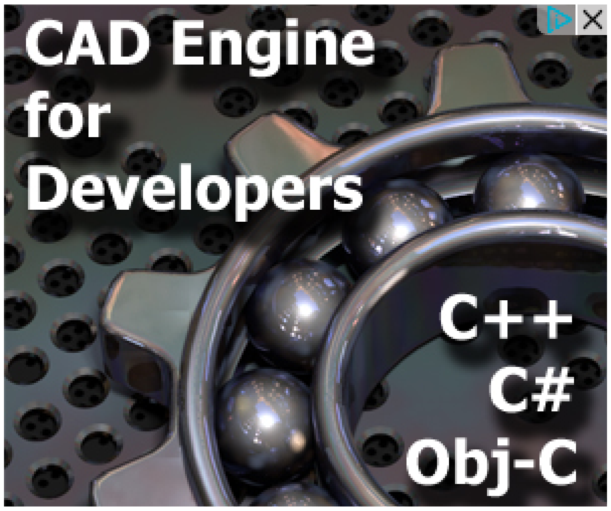
- password management.
- support, monitoring and reporting tools.
- deployment and staging.

---

## Summary

As you can see, developing an app and bringing it to production involves a variety of modern technologies. We focused primarily on client-side development, but don't forget the server-side portion of the app. Separately, they're useless, but together, you have the necessary layers for a working application.

With so much to learn, you wouldn't be alone if you feel overwhelmed. Just stick with it, and don't stop! You'll get there soon enough.



Tags: spa

### By Pavan Podila

I am a Financial Technologist building apps for Web and Mobile Platforms using Ruby, NodeJS and iOS. You can follow me on Twitter or read my Blog

**Note**: Want to add some source code? Type <pre><code> before it and </code></pre> after it. Find out more