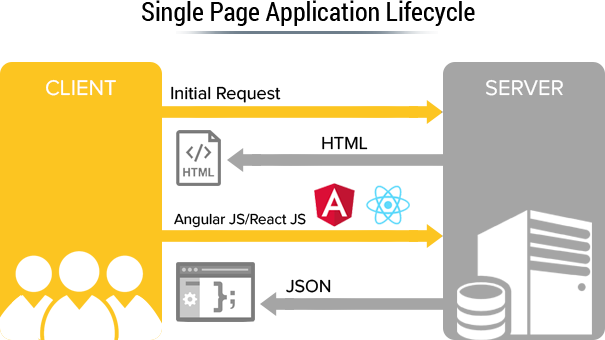
# Lecture 05 – Hosting High Geared Single Page Applications



You may be wondering why in a DevOps course we would have a lesson on the intricacies of Single Page Applications. With 4+ million weekly downloads of React and Progressive Web Apps (PWA) on the horizon, SPAs if not PWAs will be in your development future, that is for certain. It’s considered a necessity for a DevOps Engineer to understand how these web applications which are continuously striving for JavaScript nirvana, work. We are going to explore the build processes involved in them along with how to properly host them as not all hosting solutions are created equally.

It’s very rare these days to find a newly developed application on the web making roundtrips to the server. Page refreshes are typically avoided at all costs to produce a buttery smooth user experience. To meet the demand, developers make requests for data asynchronously in order to provide the illusion of content smoothly populating their intended UI elements.

Angular, Vue, React and some other JavaScript frameworks/libraries are heavily relied upon in order to accommodate the asynchronous data fetching technique responsible for primo UX. It really doesn’t matter what framework your working with, as long it’s a client-side application, it’s going to require static hosting. With static hosting comes many possibilities for deployment which brings various techniques to ensure scalability. We’ll look at what’s out there, along with the infrastructure, architecture and deployment processes that support them.

When performing an application build for a SPA, you’re going to be blessed with a bunch of static assets when the process completes. You’ll need to know what to do with them, how to work with them in the context of a CI/CD process and ensure optimal performance once they’re deployed in the wild. With respect to a SPA’s build process, they can get kind of complicated. Sometimes, more complicated than they should be. There are JS build tools designed to facilitate this client-side craziness and we’ll also talk about a couple of them. But first, what does it mean truly to be a Single Page Application? I won’t be making a case for one JS framework versus another, you can find enough developers arguing the ins and outs of this on Hacker News or Reddit.

## What is a SPA?

A boring technical definition for a **SPA is a website that re-renders content in response to an event**, like clicking a link or scrolling a trackpad. The application initially loads a single HTML document. As the user interacts with the page, content will dynamically update. This is heavily driven and made possible with the web browsers core JavaScript API. With JavaScript and the languages ability to perform asynchronous web requests in response to user interactions, content is dynamically rewritten.

SPA’s have become an integral part of the modern webs landscape. So much so that you have Google, Facebook and even Uber investigating big money into open source projects like Angular and React. With the single load required to fetch that initial HTML document which is intended to be dynamically populated through JS events, it brings some problems that need addressing. One of them being the ability to operate performantly for Search Engine Optimization (SEO). Some solutions have been designed like having the initial document rendered server-side before it’s delivered to the client. This is known as Server-Side Rendering (SSR).

Imagine shopping on a site like Amazon and not having to deal with page refreshes anymore. It would be a much-improved user experience that would most likely have a really positive impact on the time customers spend on the site. With SSR, this is can be made possible for more complex sites. Airbnb, the Toronto Star, Nike and the colorful and bold website of New York’s 2019 Pride celebration all implement SSR for their SPAs.

Aside from potential SEO drawbacks, routing can be a rather complex issue to solve in a feature rich SPA. This is where the frameworks from tech giant heavyweights come in to play. Remember, SPAs technically only have one page, so complex routing involving multiple pathnames or nested routes can get messy quick.

Now that SPA drawbacks have been identified, let’s get into why SPA’s are most likely to wind up on a DevOps Engineers plate. They are super easy to deploy and continuously version up over time. Compared to traditional monolith web apps dealing with a bunch of source code, we’re only really working with one index.html file at the end of the day. Along with the assets needed to style and populate content. These assets are typically referred to as bundles.

One thing to keep in mind is that bundles can grow quickly, especially if you have a developer who’s overzealous with third-party dependencies. It can not only slow the applications Time to Load (TTL) but also become chaotic to maintain quickly if oversight is not implemented. A DevOps engineer is usually responsible for ensuring that source maps are included in the client-side SPA for proper monitoring as error tracking can be quite cumbersome when working with minified assets.

A source map is a file that maps the transformed source to the original source, one that is legible and debug-able, opposed to the minified and compressed version served to the client. It enables a browser to reconstruct the original source and present the reconstructed original in the browser’s developer tools.

**With SPAs, after the initial page load no more HTML is sent over the network.** Instead **the application is hydrated with requests for data from a backend API**. This means that only data typically in the form of JSON payloads is sent through the wire. This drastically reduces the initial TTL as we don’t have to wait for content that the user doesn’t necessarily even care about yet.

# Hosting SPAs

As we’ve uncovered Single Page Applications are a bit of a different animal when compared to a traditional client-server model for a web application. This means that our options for hosting can also be quite different and uniquely designed for CI/CD processes. We also have some potential cost advantages of leaving a smaller fiscal footprint when it comes to infrastructure costs dependent on the applications business requirements.

Hosting can come in the form of Azure Storage, an AWS S3 Bucket, Docker containers, Pods with Kubernetes or VM’s. Some of these options are available for your traditional round-trip server web apps, so we are only going to focus on easy and cost-efficient deployments for SPAs.

It's best to ask the following questions to help decide which service to use when hosting a Single Page Application:

1. How much effort is it to manually vs automatically deploy an app?
2. How much configuration is needed?
3. How is the application expected to scale?
4. How does it currently perform in a local environment?
5. What kind of computational resources are required to support the app?

With these questions answered, we’re going to be able to make more qualified decisions when choosing a hosting service. Let’s imagine that we have a intermediately complex SPA, like a Reddit.

To keep things cheap, since we know we’re dealing with a somewhat complex Single Page App, let’s see what the tech giants offer.

## Firebase Hosting with Google

Google has included in it’s suite of Firebase products a hosting service that’s supercharged for SPA deployments. It supports custom domains, and you can easily extend your SPA with cloud functions, storage, Firestore and use one of Google’s many options for authentication.

Another advantage with Firebase is that it’s a certifiably production-grade solution for developers to leverage for hosting. They also provide an Software Development Kit for CLI deploys which makes scripting deployments into a pipeline quite straight forward and extremely customizable. With the quick deploys also comes Google’s **Content Delivery Network (CDN) for geographically optimized application serving**. Bonus feature using this service – Google automatically provisions SSL certificates for all your domains so content is securely served.

## Azure App Service

Azure’s App Service is very DevOps focused. Your SPA can come in many flavors too, the service is quite agnostic when it comes to facilitating different tech stacks. .NET, .NET Core, Java, PHP, Ruby, Node or Python. It’s designed with the intent of being integrated with other cloud services offered by Microsoft’s Azure platform to provide a fully manageable suite for builds, deploys and horizontal scaling.

Although you won’t have auto-provisioned SSL certs like you would with Google, they do make it really easy to manage certificates on their platform. Everything is typically Windows fashion here, designed for a next, next, finish, user experience. They also offer NoSQL or SQL databases for application integration. Monitoring is somewhat baked into the platform, they offer detailed performance and application health insights for accelerated troubleshooting.

## Amazon Web Services

AWS offers a unique solution for static hosting. One of their storage products known as Simple Storage Solution or S3 for short, can be configured specifically for static hosting at the click of a button. They also help with routing a bit out of the box, HTTP status codes can be easily managed with the redirect options available in an S3 bucket.

S3 can also be coupled with their CDN service, CloudFront. Using CloudFront, “distributions” can be created and delivered to over 100 edge nodes around the world. Aside from CDN’s AWS also offers cloud functions and services to help really make a single page app shine when integrated with a number of other products that they offer. If you purchase a domain name with AWS, you’ll get free SSL certificates like you would with Google. Should you choose not to use them as a domain name registrar, you can use a service like Let’s Encrypt.

# GitHub Pages

GitHub has a feature called pages which has the ability to convert a GitHub repository into a living, breathing website at the click of a checkbox in the repositories settings. The domain name offered OOB is somewhat decent too. The format looks something like this: <https://username.github.io/repo>. Not too bad for free hosting. With GitHub comes all the things one would need when integrating a build pipeline (also for free) like web hooks, authentication tokens along with quick and easy application deployments.

It’s so easy and so free that we’ll be using it to host, deploy and integrate to our CI/CD pipeline for this course. The only caveat with this service is that if your using React routing heavily, it’s known to not play too nicely. Take that into consideration when possibly choosing them as a hosting provider for your future SPAs.

## Netlify

Offering a very stress-free and easy-to-use interface that literally just takes one click for a deployment process to occurr, Netlify is another great contender when choosing a hosting provider that’s catering for SPAs. There’d be no issue hosting a full-fledged React application here with complex routing either. They also have a great A/B test suite to allow simple A/B testing for your product.

Also offering an SDK for quick command line deploys, it’s easily configurable for an integration into any CI/CD build process.

## Ziet

A revolutionary serverless deployment service for all kinds of applications Ziet’s Now is a major contender in 2019 for SPA deploys. With the service comes an extremely robust CLI and smooth integration with GitHub

## Roast

Considered to be a CDN static web host, it’s an ideal choice for deploying SPAs that are expected to have far reaching geographical traffic. Its platform is built with speed and simplicity at the core of its service. They also offer provisions for not only SSL certs but SSR too! Again, another great choice for static hosting when it comes to dealing with SPAs. They too, offer a very feature-rich CLI which makes for CI/CD integrations a cinch to script out.

With some options laid out for optimized static hosting providers, we now know of the services that are at our disposal but before we continue moving forward with a forensic analysis of a SPA’s build process, we’ll talk about some of the fundamentals involved in building a single page application, learn some of the terminology and discover why there needs to be a build process.

## Single Page Application Build Process

Without having to go into the history on the vast changes which have occurred over the years in the realm of front-end development, let’s look at some of the current steps in a modern build process individually and examine some of the problems they solve. You may be familiar with some of the terms listed below as we’ve briefly touched upon them. Now it’s time to unpack them a little bit further.

## Steps in a FE build:

* **Transpiling**
* **Bundling**
* **Minifying**
* **Packaging**

## Transpiling

One of the original problems that really hindered front-end development, especially with JavaScript (JS) (never mind CSS for now) was cross browser compatibility. Earlier we talked about the dependence on a browsers core JS API. With each browser comes their own JavaScript API. For some reason Edge (formerly IE), Firefox and Chrome have never been able to cohesively establish a common way of implementing the same JS API. They all have traditionally looked at supporting language features in different ways at different times.

With incompatible support across browser platforms, it made for developing JavaScript and CSS a little troublesome. Enter the Transpiler. **Transpiler’s were brought in to solve the issues of cross-platform compatibilities.** With the frequent changes and improvements continually introduced to the JS specifications, browsers have had a hard time keeping up and ensuring parity across platforms.

A transpiler can take the most modern syntax available in JavaScript and turn the code into it’s strictly vanilla JS alternative. A lot of the changes in JS are just syntactical sugar. Transpilers take the sugar, process it and produce source code that is cross-platform compatible.

CSS traditionally has faced the same issue as browsers again, still, just cannot agree on a single way of supporting stylesheets. To solve some of the headaches in CSS, SCSS or SASS was created. SCSS offers many features that a front-end dev could only dream of when working with CSS but for the context of a DevOps Engineer, it’s only important to be aware of these issues and how they may affect the day-to-day operations. Not so much the detailed intricacies so we’ll be leaving them out for now.

An important thing to know when working with front-end developers, they are heavily dependent on monkey patch tools to produce source code that is cross-compatible amongst all platforms. Again, just to get the concept engrained, it’s important to know that **transpilers take new and improved syntax and intelligently produce source code that is cross-browser compliant.**

Bundling

With the rapid innovations being made in front-end development, we know that browsers have struggled to provide a unified environment for code to run in. With that came a plethora of tools, libraries and frameworks. Have you ever looked at the source code of a web page and noticed an overwhelming number of script tags in the documents head? Bundling provides the ability for front-end developers to take traditional spaghetti code and gain control through modularizing their work.

Ultimately this resulted in a major innovation with NPM and its ability to manage dependencies in a centralized repository and make them accessible via command line installations. This allows for a more controlled environment for developers to work in and made the process of choosing which third-party dependency that would best solve the problem a lot easier.

Now, instead of importing a plethora of script tags in a documents head, a developer can use new syntax to “import” or “require” packages in their code. **Bundling is the process of taking all these import statements, finding the correlating JS package and adding them in the applications source codes appropriate scope** for usage. It produces one big JS file. It’s the front-end developers jobs to ensure that the JS file does not grow in size and implement techniques like asset splitting or lazy loading to ensure that the bundles remain sane in size.

## Minifying And Obfuscating

As previously mentioned, with bundling an application, if a FE dev is not careful, the bundle can balloon out of control. This results in a slower TTL. With the large number of packages and libraries included in your average modern SPA, final deliverables can be a size that’s unacceptable. This is where a minification process comes in to play as every bit and byte counts.

Minifying reduces the final file size by removing things like white-space, comments and line breaks. Things can be taken a step further in the build process with Obfuscation. Obfuscation will change variable and method names, obscuring code so it's less human readable when served to the client. It can help with security as it drastically increases the effort required to understand the code.

## Packaging

If we were to include all the tools previously mentioned into a build sequence, we’d be left with our bundle. The build tool that’s being used (we’ll look at some soon) will place the produced bundle and place it where we would like specified. That specification is typically managed in the build tools config file and is something generally like ./dist/app.js If SCSS and/or HTML is included in the build process, that must be specified for output locations as well. If this sounds a bit confusing to you, the next section is designed to further elaborate on build tools and what the process looks like when it’s all put together.

## Common Tools

We won’t need to get too deep with examples as it’s out of scope for this course, but it would be of benefit to know some of the common terminology used when speaking with a front-end developer to help produce an optimally performant pipeline.

Below is a short summary of some of the FE lingo you’ll hear when communicating with a front-end developer in 2019 and beyond. As mentioned, you won't need to know how to use the tools, that’s the front-end dev’s job.

Package Manager: npm, bower

Transpiling: babel

Bundling: webpack, gulp, Browserify, Rollup.js, Parcel

Minifying: UglifyJS, minify

Besides the tools mentioned above, some other nice-to-haves in your build process, would be linting of the code. **Linters are responsible for enforcing a conventional syntax to ensure consistency amongst development work exists**. No one wants to work in a code base where things are random acts of chaos. Linting ensures things stay consistent and can be a part of the build process. If a developer makes a PR and it happens to break a defined convention, you can trigger a build fail and send an automated notification to a developer through Slack or any other messaging platform that allows web hooks to exist.