Development Project

I thought this would be a good opportunity to demonstrate not only my understanding of React using several modern technologies, but to also demonstrate my experience with modern practices, design, and architecture. Such as the use of a monorepo architecture, which I will explain in the next section. I really am taking this opportunity to explain what I know and demonstrate it in the code.

In this exercise all the services (which are not many because it is an exercise) are deployed to AWS Serverless Lambda functions. Which is, currently, the latest and greatest trend for scaling and deploying services through a cloud provider.

Pre-requisites

If you do NOT have Yarn installed, you can install it with the following command:

- `$ npm i -g yarn`
  - Yarn, itself, is an NPM package.

Quick Start

- `$ yarn bootstart`
  - Do NOT confuse this with the command `bootstrap` in the `package.json` file. The command `bootstart` will call `clean`, `boostrap`, and then `start` (hence the name `bootstart`). This will perform all the tasks required to clean, setup and run the application in one command.

Installation

- `$ yarn bootstrap`
  - This will install all the required dependencies for the monorepo.

It is worth noting that if you need to ***REINSTALL*** all the dependencies there is script for this:

- `$ yarn refresh`
  - This will perform, first, a `yarn clean` and then, once completed, a `yarn bootstrap`. This will rebuild all the dependencies across the monorepo.

Starting the App (front-end)

- `$ yarn start`
  - This will call the `start` script for the `wmi` application project in the monorepo.

If you want a `development` or `production` build of the web app:

- DEVELOPMENT BUILD: `$ yarn dev`
- PRODUCTION BUILD: `$ yarn prod`

Cleaning

- `$ yarn clean`
  - Removes all packages, build folders, and residual dependencies.

Source Code

The entry point for the web application is as follows, from the base directory of the monorepo:

- `src/wmi/src/index.jsx`

The entry point for the service API is as follows, from the base directory of the monorepo:

- `services/cars-api/serverless.yml`

Architecture

The architecture demonstrated here is definitely overkill for this exercise, however, it is a good chance to give a demonstration of what I know. Which is the entire point, right?

Monorepo

The problem a monorepo resolves is related to large scale projects, testing, and management of code, according to Lerna, they explain that: ***"Splitting up large codebases into separate independently versioned packages is extremely useful for code sharing. However, making changes across many repositories is messy and difficult to track, and testing across repositories becomes complicated very quickly."***

Additionally, if the viability of this solution seems unreasonable, consider that ***"projects like Babel, React, Angular, Ember, Meteor, Jest, and many others develop all of their packages within a single repository."*** Some of the biggest companies in technology utilize this monorepo structure for their large scale projects.

In short, a "monorepo" is a single repository that contains multiple projects and modules code that can be treated as separate and independant modules within the repo.

Cons:

- A little more complex to understand at first.
- Does not conform to the legacy architectures.

Pros:

- Less space required for the packages because only ONE copy of a depenency ever exists across the entire monorepo. Even if other projects use the same package.
- Only ONE code base to maintain.
- Can contain multiple projects (i.e mobile, web, and/or desktop apps, etc).
- All modules exist within the same code base and are easily accessible.
- Cross-project(s) testing can be done consistantly across the entire repo.
- Fewer code repositories and less complex coding interactions.
- Simpler build lifecycles.
- Easier deployment processes that may use shared pipeline resources accross projects.

Technologies

Yarn

This is a package manager alternative to `NPM`. Why would someone want an alternative to the classic `NPM`? There are several reasons. Most of all is flexibility of the projects capabilities as well as the reliablility to download, maintain, and update packages faster, more securely, and (if necessary) in a specific way. Yarn advertises itself as faster than NPM and, from experience, it is..., significantly. Especially, when you are dealing with a large number of packages as projects grow. Additionally, you get the flexibility of "workspaces", which is a "monorepo"-like concept. Except, that tools like Lerna actually use Yarn Workspaces in order to make their solution work. There are other benefits to using yarn that you get that NPM just does not have the functionality for. I like to think of NPM as a Honda Civic and Yarn as the Cadillac Escalade. Yes, NPM will get you from point "A" to point "B", but it will take a lot more work to have the same features that a Cadillac has, by default. The difference is that with the Cadillac

you get there more easily and with less issues and all the comforts and flexibility you could desire.

Yarn Workspaces

NOTE: This a direct quote from their website.

"Workspaces are a new way to set up your package architecture that's available by default starting from Yarn 1.0. It allows you to setup multiple packages in such a way that you only need to run yarn install once to install all of them in a single pass."

Webpack

In short, WebpackJS is an open-source JavaScript module bundler. It can be used for both the front-end and back-end of systems that utilize JavaScript as their language of choice. I wanted to emphasize that Webpack is NOT a solely front-end technology. It is a bundler technologie that has many applications as both a front-end and back-end bundler. Therefore, is you had a NodeJS service, I would highly recommend using it to bundle the back-end modules as well.

The benefit to bundling everything is that it is a central point of compatibility, transpilation, performance, and analysis. With Webpack on the back-end you can utilize a Babel transpiler, target a specific version of NodeJS, and generate a bundle analysis that will tell you how the bundle is performing. These same things are true for the front-end, except that you could target a browser version instead of a NodeJS version.

Babel

NOTE: This a direct quote from their website.

"Babel is a toolchain that is mainly used to convert ECMAScript 2015+ code into a backwards compatible version of JavaScript in current and older browsers or environments. Here are the main things Babel can do for you:"

- "Transform syntax."
- "Polyfill features that are missing in your target environment (through @babel/polyfill)."
- "Source code transformations (codemods)."

Personally, because it targets JavaScript, you can use this for NodeJS versions of JavaScript as well. It works great for both front-end JavaScript and back-end NodeJS services.

Serverless

The technology is useful, not the biggest fan of the people. In this example project I am using Serverless on an AWS NodeJS platform to serve as the serverless back-end API service for the MongoDB database.

There is more info here.

AWS Serverless Configuration

Using AWS with a configured profile from AWS in the Serverless framework.

- Serverless Framework Credentials:
  - `$ sls config credentials --provider aws --key <aws_access_key_id> --secret <aws_secret_access_key>`

MongoDB

NOTE: This a direct quote from their website.

"MongoDB is a general purpose, document-based, distributed database built for modern application developers and for the cloud era."

Due to the large amount of adoption and fast-paced growth and support of MongoDB and other BSON (JSON) based NoSQL databases, I highly recommend and encourage this databases use. It is much easier and quicker to work with when dealing with large amounts of data, from my experiences. Large RDMS's tend to become extremely complex and the relationships between tables usually ends up spanning other databases or there ends up being partial primary keys made up of 15 different columns to identify one piece of data in one table leading to relational nightmares if data gets entered in the database incorrectly. MongoDB encourages larger single documents over numerous relationships to other tables. In MongoDB a collection is simular to a single topic for data and makes scaling a little easier in my experiences.

Lerna Monorepo's

NOTE: This a direct quote from their website `README.md` page.

"Lerna is a tool that optimizes the workflow around managing multi-package repositories with git and npm."

"Lerna can also reduce the time and space requirements for numerous copies of packages in development and build environments - normally a downside of dividing a project into many separate NPM packages. See the hoist documentation for details."