CarDash Backend Guide

Node.js

CarDash primarily uses Node.js for both its frontend and backend code. Large parts of the code base are usable by both frontend and backend, which reduces development time and reduces the need for specialized teams that can only work on one small piece of the codebase (e.g. iOS team that only works with Swift, Android team that only works in Java, backend team that only works in Python, etc). Node.js also has an easy-to-use package manager called NPM that allows us to easily import common packages so we don’t have to write code that is non-specific to CarDash.

Node.js gained popularity due to its asynchronous nature. Traditional server architectures were typically prone to performance degradation because of I/O operations. Node.js is non-blocking by default, which means that I/O does not prevent the server from handling more requests. For example, during a database query, Node.js will perform other tasks until the event has completed, in which case it will resume execution of the code that follows the database query.

We also use Express, which is an application framework for Node that just about everyone who uses Node for web applications uses.

There is nothing unique about the CarDash code base that requires it to be written in JavaScript, but we believe that the advantages listed above make it a compelling choice for our technology stack.

Installing Node.js

Please download the latest version from this [link](https://nodejs.org/en/) for your specific operating system. We also recommend installing [n](https://github.com/tj/n), a command-line tool that enables you to easily switch Node versions as needed.

If you are not familiar with Node, there are many resources for you to learn the basics. [NodeSchool](https://nodeschool.io/) is a good place to start.

React

Why React?

React is a frontend framework that allows you to write JSX files that look similar to HTML but with the added flexibility of JavaScript. It was initially developed by Facebook and released in 2013. React is now the most popular JavaScript frontend framework. The framework lets you write web pages that can be rendered and executed in both the frontend and backend, however CarDash does not currently use server-side rendering.

A Quick Overview

React allows you to write and compose elements easily:

```js

const HelloComponent = (props) => {

return <div>Hello {props.name}!</div>

}

const Page = () => {

return <HelloComponent name=”World” />

}

```

We recommend reading the [docs](https://facebook.github.io/react/) to get started, then following [this](https://scotch.io/tutorials/learning-react-getting-started-and-concepts) tutorial and [this](https://www.codecademy.com/learn/react-101) tutorial. It is not important that you have much experience with React, but it would be helpful to know the basics of [unidirectional data flow](https://www.youtube.com/watch?v=i__969noyAM), the component lifecycle, and other basics of React.

GraphQL

GraphQL vs. REST

[GraphQL](http://graphql.org/) is another technology created by Facebook that improves upon an older technology. It introduces a type system that allows developers to create and compose complex queries without needing specifically defined endpoints as with REST. We recommend using [Graphiql](https://github.com/graphql/graphiql), a desktop app that lets you test GraphQL APIs locally.

GraphQL provides many advantages, the main one being that both the frontend and backend don’t have to explicitly create an endpoint per page. The frontend dictates what data is returned from the backend. This allows for increased flexibility, but is also a performance improvement as it prevents unnecessary data from being returned.

Apollo Framework

The [Apollo framework](http://dev.apollodata.com/react/) is a client-side framework that allows React components to easily fetch data from GraphQL APIs. The framework wraps a React component with a query that is automatically executed when the page is loaded. This framework greatly reduces boilerplate and lets you quickly populate a component with data simply by specifying the appropriate GraphQL query.

There is a great tutorial on Apollo [here](https://www.howtographql.com/react-apollo/0-introduction/).

Builds and Deployments

Lerna: Why a monorepo?

CarDash uses [Lerna](https://github.com/lerna/lerna), a JavaScript tool that allows for the development of multiple JavaScript packages in a single repository. Lerna lets CarDash developers work together while avoiding dependency issues by allowing packages to directly reference other packages in the same repository. Read more here about the benefits of a [monorepo](https://danluu.com/monorepo/) here.

Running packages

All CarDash packages share a package.json file which has a script defined as “dev”. This allows any developer to simply `cd` into a directory and type `npm run dev`. These scripts lets developers easily start developing on packages they may not have used before.

Creating packages

We try to keep the packages as small as possible and only create new ones when logically necessary. When creating new packages, you can use existing packages as templates as appropriate. For example, when creating a new Lambda package, look at existing packages such as cardash-api as a template on how to organize your code.

ES6 / Advanced JavaScript Features

Advanced JavaScript features

CarDash uses many JavaScript features that are not available in all Node versions. Some of these features include async / await, import, arrow functions, and more. Many of these features were borrowed from other languages. If you’re familiar with another C-style language, transitioning to JavaScript should not be too difficult.

We recommend reading about advanced JavaScript (ES6) features [here](http://es6-features.org/).

Async / Await

Many JavaScript programmers used callbacks. In fact, one of the main advantages of Node.js (the event loop) relies on callbacks. However, callbacks often leads to “spaghetti code” that contains lots of indirection. To solve this, async / await was introduced to allow for asyncnchronous programming that can be written in an asynchronous manner. Read more [here](https://ponyfoo.com/articles/understanding-javascript-async-await) about async / await.

Babel

As said above, CarDash takes advantage of many new advanced JavaScript features. Since many of these features are not enabled out of the box, we need to use a code transpiler called [Babel](https://babeljs.io/). Babel allows you to use features that are not yet available in current Node versions. It also allows us to use JSX files necessary for React. We have already set up much of the Babel tooling, but it is a good idea to become familiar with Babel. The most important thing to remember is since code must be transpiled, you have to remember that changes in one package must be re-transpiled before they can be reflected in other packages.

Webpack

Webpack is another part of the CarDash toolkit. Its purpose is to modularize and package the JavaScript code into bundles. Webpack can be difficult to setup, luckily we have already finished most of the heavy lifting. Still, we recommend reading the [Webpack docs](https://webpack.github.io/) to see how it works.

ESLint

Purpose of Linting

Linting prevents common code errors, improves performance, and probably most importantly, makes code style as uniform as possible to allow developers to program without worrying about style. We use ESLint which can easily be setup with Sublime Text, Atom, or your preferred text editor. Linting is required before submitting PRs.

Knex

Knex / Objection is a JavaScript tool that lets us wrap the database layer in an ORM. It comes with a CLI tool that lets us do migrations so that our database can be kept in sync. You can install the command line tool [here](http://knexjs.org/).

Typically, when you make changes to the GraphQL API, you will also edit the underlying schema and database table. To keep data consistent among local, staging, and production versions, you must create a migration file. This migration file can then be executed by any developer to keep their version in sync.

Authentication

Authentication is necessary to protect endpoints from public access. The GraphQL API allows endpoints to be protected on a per-resource basis.

JWT Tokens

To use the GraphQL API, you must use [JWT](https://jwt.io/) tokens. JWT tokens are Base 64 encoded tokens that contain user and other information that can be used to authenticate a client. You must pass this JWT token to the authorization header as a bearer token. JWT tokens only work for specific environments. For example, staging JWT tokens only work for staging and not for production. To obtain a JWT token, you must first log-in to the appropriate admin environment and fetch the JWT token from local storage.

JWT tokens protect against attacks involving cookies and sessions. They are superior also because they can be easily passed between services. However, care must still be taken to keep JWT tokens secret. Frontend pages must be protected from XSS, otherwise tokens may be hijacked and used maliciously.

AWS Lambda

Serverless Architecture

Traditional monolithic architectures require code to be deployed and bundled together. With a serverless architecture, you deploy functions instead of entire applications. With this approach, you can easily scale your infrastructure. For example, if your user service requires more resources than your billing service, then using [Lambda functions](https://aws.amazon.com/lambda/) may be beneficial as they can be autoscaled. The alternate would be to deploy a single monolithic application that can’t scale services independently. For simplicity, CarDash uses hybrid architecture wit both a monolithic application and serverless components.

The GraphQL API which handles most of the business logic is deployed as a monolithic application. Serverless components, for example an API that allows us to render PDFs from HTML, is deployed as separate Lambda functions. You can read more about [about serverless architecture](https://martinfowler.com/articles/serverless.html) here.

Deploying Lambda functions

We use the [Serverless framework](https://serverless.com/) to simplify deployment. The command line is simple and easy to use. One downside with serverless functions is that you must deploy them independently. This same disadvantage can also be an advantage, as you could selectively only re-deploy certain parts of your application stack.

Workflow

Git rebase

We assume you are familiar with git. We use the git rebase workflow which requires you to rebase all commits to the branch you’re developing off of, so that it can be easily reviewed and merge without any extraneous merge commits. You can read about rebasing [here](https://git-scm.com/docs/git-rebase) and [here](https://www.atlassian.com/git/tutorials/merging-vs-rebasing).

Automated workflow

We recommend some familiarity with JIRA or a similar bug tracker. We tag commit messages that allow us to automate our deployment pipeline. For example, submitting a branch with CARDASH-101 or a commit message that has CARDASH-101 in its title automatically moves a JIRA ticket from “Ready” to “In-Progress”. You can read more about [JIRA](https://confluence.atlassian.com/get-started-with-jira-software/i-want-to-learn-more-about-jira-software-844502159.html) here.