Rationalization of Choices

URStreamSight

Raymond Knorr, Avery Cameron, Noah Rowbotham

Version: 3 2021-04-08

Table of Contents

Overall Rationale:

In our Capstone, a lot of consideration was done to ensure maintainability, testability, project transfer, ease of use and reusability. This has influenced and informed our design decisions described in this document. They are our guiding force that have allowed us to create clean code. Working for Prairie Robotics and the municipalities, we knew that at the end of the project, it would have to be maintained by Prairie Robotics and still used by the municipalities. When selecting the language, frameworks used, testing plans, CICD pipeline and AWS services, we had to consider the user and the maintainer. Creating a robust test plan and CICD pipeline benefits both Prairie Robotics and the municipalities by reducing errors that reach the users and providing trust in the code for Prairie Robotics.

Software Principles

SOLID

The use of the Controller/Service/Repository pattern on our API allows for easy object-oriented design. The main classes associated with an endpoint will be one of these three types. When designing our API, we followed the Single Responsibility Principle. The NestJS framework we selected helps to ensure the separation of code into classes that deal with only one part of the API. By separating route handling in the controllers, business logic in the service, and repository in the model, our code did not have multiple responsibilities in one file. By changing from a prototype in Express to using NestJS as described above, our code was able to adhere to this principle much better. Additionally, NestJS makes great use of Dependency Injection, where it is simply part of the framework. Similarly, the use of the React and NestJS frameworks means that any classes we create are implementations of library classes. Our code does not make use of that much inheritance, so the Liskov Substitutability Principle does not directly apply. Similarly for interface segregation. We do not have classes implementing very rigid interfaces. Although this type of object-oriented development is possible in TypeScript, it is not the norm with the frameworks we are using.

DRY - Don’t Repeat Yourself:

For our code, we have done our best to follow the DRY principle. We have done this to ensure there is a single point of truth and that we are working together and not repeating work others have already done. This creates cleaner code as code that is functionally the same is not written different ways by different authors. It also helps with future development by using modules that we have already created. This can be seen in our validation with joi, error throwing and various components in the frontend as well.

YAGNI - You Aren’t Gonna Need It:

We followed YAGNI to make sure that we were not getting distracted or over ambitious with the features we were implementing. Following YAGNI allowed us to better meet client needs and expectations and reduce wasted development time. If we came across a feature that could be implemented, we talked about it amongst ourselves and with

Prairie Robotics to check if it was truly needed. This check process reduced the complexity of our code and helped to focus development. We still developed to allow for additions in the future and noted features that could be added in our backlog.

KISS - Keep it Simple Stupid

Keeping it simple is a key part of our development. Keeping the backend simple and easy to work with helps with maintainability and the transfer of code to Prairie Robotics. We also worked to keep our frontend simple to allow for easy navigation by users. We implemented feedback from clients based on the demos and the feature backlog and improved navigation and ease of use.

Linting standards:

We are using a TypeScript linter to enforce style rules that are set up in our repository. We are using typescript-eslint for this as it is the widely accepted tool for TypeScript linting after TSLint was deprecated in favour of this to provide more uniform linting.

Linting provides a way to ensure our coding standards are enforced and that code will remain consistent across all commits by any team member (Gimeno, 2017). We are ensuring this by checking it as part of our CICD pipeline on each commit. The setup of our rules is based off of best practices at Shopify, eslint style guides and Prairie Robotics' own rules they were working with. We are also using Prettier for formatting changes as it is generally better suited for enhancing overall readability. Both of these tools work well with vscode but also with any text editor. We created commands that allow for linting checks and then fixes afterwards. The combination of linting and formatting provides consistent code and also can catch errors such as unimplemented functions or improper handling of callbacks.

CICD Pipeline:

Our CICD Pipeline uses GitHub Actions which runs on GitHub and can be set up to trigger automatically on commits, pull requests, comments and other events. GitHub Actions can run shell commands to execute custom scripts, run on different environments and run separate jobs. It also allows the use of Actions created by users to run custom checks. We are using GitHub Actions instead of alternatives like CircleCI or Jenkins because it allows our CICD to be in the same place as our code and it provides several free plugins for code testing, coverage and deployment.

We have several steps in our pipeline. First, on every push, we are running our shell script for linting files to ensure that the commit follows our style guides and ensures consistency. Next, we test our code using Jest and ensure that none of the tests fail. After the tests pass, code coverage is checked to ensure that it meets our standard. Our current standard is 80% coverage. The coverage criteria was based on our test plan and helps to ensure adequate testing and that new features are tested as wel. The API is then built and deployed using Serverless. The Serverless configuration takes our database credentials for an AWS RDS instance from AWS Secrets Manager and sets up a Lambda function and API

Gateway with access to the database. The front end is also built using `nx build ur-app` which uses Nx's built in builder. Then the distribution folder is uploaded to a public AWS S3 bucket with static web hosting.

Currently, this is deployed to our main server, this could be improved to be deployed to a test server. After brief end to end testing it would then be deployed to a staging server. Finally, after proper end to end testing, it would be deployed to a production server. This would help to prevent problems reaching production. Custom GitHub Actions in the future could improve linting, pull requests, or add Code Review tools to the pipeline.

Tech Stack:

Languages - TypeScript vs JavaScript:

When creating the frontend, consideration was put into the language chosen. We chose TypeScript over JavaScript for several reasons. We wanted to create clean, readable code that's easy to maintain and could be transferred to and used by Prairie Robotics. TypeScript provides typing for code and explicitly requires variable typing, expected parameters and return types. With JavaScript, it can require a lot of debugging checks and logging to determine type errors or what functions are expecting (Tomaszewski, 2020). TypeScript is easier to maintain by different developers and refactor as the types are explicitly mentioned and the purpose of the code is more clear. A large reason for its use, in addition to its benefits over JavaScript, is its use by Prairie Robotics. As it is already part of their tech stack, creating with TypeScript does not add another language to learn or manage for the team.

Cloud Services - AWS:

When selecting AWS services, we had to consider usability, responsiveness, and cost. For our front end we are using a public AWS S3 bucket with static web hosting enabled. This allows access to the site and for easy hosting and deploying from our CICD pipeline. With only deploying to S3, the URL is an Amazon URL and hard to navigate to. Another downside to S3 is that it uses HTTP. To fix the issue of the domain, we used the domain: *getstreamsight.com* which Prairie Robotics bought for hosting. To use the domain for S3 we had to use AWS Route 53 to route the traffic from S3 to the *getstreamsight.com* domain. Then we used Cloudfront for distribution and to ensure that HTTPS is enabled.

For the backend, because we used Severless, we were able to provide a role with permissions for creating an AWS Lambda endpoint with an API Gateway connection and connection to an Relational Database Service(RDS) instance we created. Using Serverless provided easy setup and configuration of our AWS services for our backend. For the RDS to have the correct configuration and permissions, we also had to configure our Virtual Private Cloud (VPC), Access Control Lists (ACL), Subnets and Security Groups. Our VPC is hosted in us-east2 with 3 subnets.

Based on Craig's comment, *"This project is really cool. Lambda/AWS - how do you folks feel about using Lambda as you are now handcuffed by AWS".* In some respects we are handcuffed by AWS, our API and frontend could be hosted outside of AWS with configuration in deployment to any other Cloud provider but this would add some difficulty.

Prairie Robotics is already using AWS for their systems which is a large reason for our use of AWS as well. Using AWS or a Cloud Provider in general adds reliability and development speed and easy hosting and configuration. With our work on Infrastructure as Code, the entire system can be quickly set up for new clients and requirements. The cost of hosting and setting up all the different portions on our own servers would be prohibitively expensive and AWS provides scaling which is great for the future. Overall, we recognize that we are handcuffed but we feel like a cloud provider is the right choice and that AWS is the right provider for our needs.

Monorepo - Nx:

What is Nx:

Nx is a monorepo that is used to combine multiple repositories into one. This reduces the need for imports, provides one source of truth and does not require libraries to be hosted as packages locally, on npm or other options (Vardhan, 2020).

Nx provides tools that allow for the easy creation of apps built with different frameworks such as React or Angular. It also helps with the generation of libraries for use between apps/repositories. Nx uses an 'apps' folder to separate what would generally be different repositories and a 'libs' folder to hold information to be shared across apps.

Alternatives:

A popular alternative to Nx is Learna. Learna tends to be used for open source projects which use a small subset of tools and dependencies (Barak, 2020). Prairie Robotics and our own team are developing multiple applications with different frameworks and tools. Nx is better suited to this type of work, with their increased CI efficiency for testing affected files and projects (Savkin, 2020). Using Learna would work but potentially cause longer CI times and result in slower feedback from the pipeline.

Another alternative is Yarn workspaces. Its second version is in active development and functionality is being added . It isn't as fully functioned as Nx and Learna and lacks affected file checking and some of the other features (Barak, 2020).

Why:

We signed a Mitacs contract in which we received funding for the ownership of the code we created. This allows us to be a part of the monorepo safely with no IP risk and to integrate with Prairie Robotics system easily and create code that is consistent with their style and rules. Nx was chosen for its app generation capabilities and the efficiency of Nx affected, build and serve (Nx, *Getting-Started: Getting started: Nx react documentation*).

Front End - React

What is React:

React is a frontend library that provides interactive UIs with encapsulated components that manage their own state. React is an open-source framework developed by Facebook. React uses components which promote code reuse and the DRY principle. You can create a small component for something like a button or checkbox and reuse that component in multiple larger components. We are using basic React components from

MaterialUI and creating reusable custom components when needed. React uses a combination of Typescript and HTML using the TSX file format. TSX allows code resembling HTML, or actual HTML, to be embedded in regular TypeScript. Other frameworks like Vue or Angular use TypeScript embedded in HTML files, which leads to confusing syntax and the use of multiple languages within one repository (Long, 2020).

Alternatives:

Angular is a frontend framework developed by Google. It tends to be more heavyweight than React and better suited to large enterprise applications. It is used in enterprise applications and is more work to set up compared to React. As well its use of TypeScript in HTML makes some frontend development more difficult (Long, 2020).

Vue is another frontend framework that is popular. It uses components, a virtual DOM and offers JSX and TSX similar to React. Vue tends to be great for small web page apps but it lacks some common components and plugins to be used by developers. As well, Vue has a smaller community when compared to React (Long, 2020).

Why:

In addition to its ease of use, Prairie Robotics also has been using React for its frontend. It has a relatively low learning curve and allows members new to React to quickly catch up. Our frontend follows Prairie Robotics Styling and layout. This helps the product flow for users of Prairie Robotics products. Also, several of our group members had previous experience using React.

API - NestJS:

What is NestJS:

NestJS is a framework that provides scalable, efficient and easy to read code for API routing (NestJS). NestJS is a highly opinionated framework and defines a lot of the folder structure, files and tools used (Rahman, 2019). This provides consistency across projects and teams and once learned it becomes easier to work with. Although NestJS is more involved in its setup, it provides commands to auto generate boilerplate code for new routes. NestJS structures things in a controller->service->guard->model->repository structure which adds layers and follows the single responsibility principle (NestJS). In the controller, NestJS uses decorators to define routes and the class has a decorator that prefixes all routes in that file.

Alternatives:

Express is a simple, light-weight library for routing API calls. It is an un-opinionated framework and leaves choices up to the user for folder structure, error handling and messages, and body parsing (Rahman, 2019). Express is easy and quick to implement but lacks set folder structure and tools that help it to scale well. Express can be augmented with different libraries and tools but this varies with each project, adds complexity. This reduces the ability for a team familiar with Express to understand a project that is using Express if they are unfamiliar with the tools implemented with it.

Routing-controllers is similar to NestJS in folder structure, but it does not contain the service layer and is less opinionated than NestJS overall (Rahman, 2019). This would require the creation of this layer which is not required to be specified or the combination of the two layers and loss of single responsibility. Routing-controllers also uses decorators for routes and query parameters similar to NestJS (Typestack). However, Routing-controllers requires the prefix, such as ‘/users’ to be present on every function instead of just the class itself, as in NestJS (Pleerock, 2017). Routing-controllers contain a JsonController option if all responses simply return Json and is better optimized for it (Typestack).

Why:

Overall, NestJS was chosen for this project because of its adherence to the DRY principle and creation of scalable code. The boilerplate generation speeds up development time and reduces errors and creates more maintainable code. Although routing-controllers is extremely similar, the lack of inclusion of the service layer, lack of file generation and its implementation of decorators for routing makes it a weaker option. Using NestJS provides Prairie Robotics with better code once it is transferred to them as it is easily readable and learnable by the team.

Database Schema and Normalization

Our database is in third normal form (3NF). This is standard practice but we are ensuring that no cell contains multiple entries and each record is unique, this achieves 1FN (Guru99). We are using single column primary keys to achieve 2NF to ensure that only one column is needed to identify a record (Guru99). To do this, we are using an id column filled with UUIDs with foreign keys to the organization and in some tables foreign keys to the service route and address. Next, we achieved 3NF by ensuring that no change in one column would require another column to change. By enforcing 3NF in our database, we have improved database querying while reducing the need for additional storage (Guru99). This also enforces consistency, there is a single point of truth for data changes which reduces errors.

The schema for the database was created with the support of Prairie Robotics. It contains all the tables for storing information as well as the join tables used by sequelize. The tables are generated from TypeScript files at the startup of the server if they do not currently exist. The schema can be viewed on our GitHub.

We are using sequelize, an object-relational mapping tool (ORM), which allows you to query and access the database more similar to objects than traditional queries. Using sequelize and the object like access it provides, allows for cleaner code with dot notation as well.

Testing

Testing is a balance between quality and time, resources and human resources. Because our time is limited in capstone but we want to ensure high quality work, we are using integration tests primarily. Using integration tests on the controllers allows for testing of the controllers and the service that handles the input. These functions are highly connected and this will still allow for high reachability. Testing them together also ensures that the controller is not infected by output from the service. Errors in the service will propagate properly to the controller and be revealed in the test output. This makes us confident in the choice to use integration tests when possible. Unit tests will are used in SSIO.

Along with our tests, we are checking test coverage. We are using a test coverage metric of 80% to ensure that most cases are covered but intend to raise this as much as possible to improve code quality. This is covered more in the CICD section above.

References

Barak, T. (2020, June 25). Selecting the right tool for your monorepo. Retrieved March 07, 2021, from https://blog.bitsrc.io/selecting-the-right-tool-for-your-monorepo-fafe409134b3

Gimeno, A. (2017, June 22). Why you should always use a linter. Retrieved March 06, 2021, from https://medium.com/dailyjs/why-you-should-always-use-a-linter-and-or-pretty-formatter-bb5471115a76

Guru99. (n.d.). What is NORMALIZATION? 1NF, 2NF, 3NF, BCNF database example. Retrieved March 07, 2021, from https://www.guru99.com/database-normalization.htm

Long, B. (2020, February). Vue vs React Vs Angular: Best javascript frameworks in 2020. Retrieved March 09, 2021, from https://www.tiny.cloud/blog/vue-react-angular-js-framework-comparison/

NestJS. (n.d.). Documentation: Nestjs - a PROGRESSIVE Node.js framework. Retrieved March 07, 2021, from https://docs.nestjs.com/

Nx. (n.d.). Cli: Affected:test: Nx react documentation. Retrieved March 07, 2021, from https://nx.dev/latest/react/cli/affected-test

Nx. (n.d.). Getting-Started: Getting started: Nx react documentation. Retrieved March 07, 2021, from https://nx.dev/latest/react/getting-started/getting-started

Pleerock. (2017, June 6). Pleerock/routing-controllers-express-demo. Retrieved March 07, 2021, from https://github.com/pleerock/routing-controllers-express-demo/tree/master/src

Rahman, S. (2019, November 10). Why I choose NestJS over other Node JS frameworks. Retrieved March 07, 2021, from https://medium.com/monstar-lab-bangladesh-engineering/why-i-choose-nestjs-over-other-node-js-frameworks-6cdbd083ae67

Savkin, V. (2020, December 29). Why you should switch from lerna to Nx. Retrieved March 07, 2021, from https://blog.nrwl.io/why-you-should-switch-from-lerna-to-nx-463bcaf6821

Typestack. (n.d.). Typestack/routing-controllers. Retrieved March 07, 2021, from https://github.com/typestack/routing-controllers

Tomaszewski, J. (2020, April 19). Why TypeScript is the best way to write front-end in 2019. Retrieved March 07, 2021, from https://medium.com/@jtomaszewski/why-typescript-is-the-best-way-to-write-front-end-in-2019-feb855f9b164

Vardhan, H. (2020, September 23). The pros and cons of monorepos, explained. Retrieved March 07, 2021, from https://medium.com/better-programming/the-pros-and-cons-monorepos-explained-f86c998392e1