

# Code Quality Review Report

## URStreamSight

Raymond Knorr, Avery Cameron, Noah Rowbotham

Version: 1 2021-04-07

# Table of Contents

<b>Table of Contents</b>	<b>1</b>
<b>Code Formatting</b>	<b>2</b>
Alignment, Formatting & Code Blocks	2
Ensure Proper Naming Conventions	2
Code Should Fit Horizontally on a 14" Monitor	2
Remove Commented Out Code Blocks	2
Future Formatting Plans	2
<b>Architecture</b>	<b>2</b>
Separation of concerns	2
Proper Code Patterns and Technology	3
Design Patterns	3
<b>Coding Best Practices</b>	<b>3</b>
No Hard Coded Values	3
Group Similar Values in Enums	4
Comments	4
Avoid Multiple If/Else Blocks	4
Use Framework Features	4
<b>Non-Functional Requirements</b>	<b>4</b>
Maintainability	4
Reusability	5
Reliability	5
Extensibility	5
Security	5
Performance	6
Scalability	6
Usability	6
<b>Object-Oriented Analysis &amp; Design</b>	<b>6</b>
SOLID	6
<b>Appendix</b>	<b>7</b>

# Code Formatting

## Alignment, Formatting & Code Blocks

All of our code for our API uses ESLint for finding code smells, and Prettier for proper formatting. These two work hand in hand with built in support in Visual Studio Code. We can check whether our code passes linting using `npm run lint` or `npm run lint-and-fix` to lint and fix any automatically fixable errors. ESLint and Prettier check for a lot, such as tab-length, line-length, unused variables/imports, etc. Our Front-End application is made using React, and also has linting enabled using ESLint. It provides the same utility there. Our back-end SSIO module, made using Python, follows Python standards and was enforced using PyCharm's built-in linting.

## Ensure Proper Naming Conventions

Just like the previous point, ESLint checks all of our variable names and makes sure that they are accurate to the language we are using, in this case TypeScript. Classes start with capital letters, variables and functions start with a lowercase, and everything is in camelCase. File names in our API and front-end follow best practices for the frameworks we are using. This includes `.spec.ts` for testing modules, and other prefixes such as `*.service.ts` for NestJS specific practices. In SSIO, Python best practices are used, including snake\_casing which is again enforced using PyCharm.

## Code Should Fit Horizontally on a 14" Monitor

In our Front-End and API, we set the max-line length to be 80 characters. This easily fits the requirement of fitting on a 14" monitor. For SSIO, PyCharm also limits line length to 80 characters for easy viewing. **Code Snippet 1** in our appendix is a good example of line length and other formatting.

## Remove Commented Out Code Blocks

ESLint helps identify these in our API and Front-End. They were avoided in general, and any outstanding instances were removed in linting. SSIO was thoroughly inspected and purged of commented out code, important older code can be accessed through version-control.

## Future Formatting Plans

We plan on introducing pre-commit hooks to our repositories that will perform linting to make sure that any code entering source control has proper formatting. SSIO will get updated with more docstring for many of the features.

# Architecture

## Separation of concerns

Our application is split into four main modules, which are our API, Front-End, SSIO, and a collection of other AWS services. The API is responsible for data storage and acquisition, the

Front-End is responsible for data display, and SSIO is responsible for taking images and GPS data from the truck and putting it into AWS. The other AWS services have responsibilities such as running image classification, and putting classification results in the database. Our API file structure consists of various folders within the src tree that each have one controller. More about this in our section on Design Patterns. Our Front-End has its components in various sections of a `/components` subfolder of source. SSIO is further split into a series of modules including `gps`, `bin_management`, `bin_detection`, `camera`, `ipc`, and `upload`. Although there is room for improvement in the separation of responsibilities for the `camera.py` module. **Code Snippet 2** in our appendix shows the file structure for the controller/service/repository architecture used in the API.

## Proper Code Patterns and Technology

Our API is using NestJS, which is a modern and increasingly popular framework for building best-practice APIs. Some of its main claims to fame is its usage of Dependency Injection and its opinionated nature. Our Front-End is using React which is one of the most popular frameworks for creating scalable user-facing apps. Both of these repositories are using TypeScript, which includes the newest features of JavaScript as well as type checking and other benefits of compiled languages. SSIO is using a slightly older version of Python (3.7) to meet package dependencies. SSIO leverages multiple Python packages such as multiprocessing, PyTorch, serial, opencv, and many others.

## Design Patterns

The main design pattern that our code uses is the Controller/Service/Repository pattern in our API. This pattern essentially dictates that the logic is broken up into three types of modules, where the controller deals with input and output of the API via HTTPS, the service performs business logic, and the Repository represents the database in code. We used the Sequelize ORM to execute all of the SQL statements in our MySQL database. SSIO follows the python specific *pool of workers* pattern for leveraging multiprocessing. This lets us simultaneously operate upload and camera routines so that we don't miss bin tips while waiting for uploads to complete. SSIO also leverages a leader/follower pattern (typically known as master/slave). This pattern allows us to inherit and wrap one of our two instantiated Camera classes in a PrimaryCamera class. The PrimaryCamera is triggered by the detection model. Then it orders the follower Camera to take images based on logic rather than the detection model. In the future, we would like to implement a message broker and use the Observer pattern to more efficiently communicate between all of our modules.

## Coding Best Practices

### No Hard Coded Values

All of our code for our API and Front-End uses constants for main values and our server setup is handled with a `.env` file. Our linting can catch times when hard coded values are used but this is done locally to reduce the need to ignore rule comments for valid constant declarations that are caught at times. We have checked over our code with this tool to reduce the occurrence of 'magic numbers'. SSIO uses config files to specify most of the necessary

constants. There are two instances where a constant is used within a class rather than within the config. **Code Snippet 5** shows our constants file for our API, and **Code Snippet 6** shows our configuration for the API as well.

## Group Similar Values in Enums

Our code in the API contains good examples of enums for similar values. The environment variables, types of contaminants, role types, roles, service types and streams have all been grouped into enums to help create cleaner code. In SSIO, we group camera type, camera roles, camera topics, and bin states into enums. **Code Snippet 7** shows an example of one of the enums we used.

## Comments

Almost all of our code is free of comments explaining what the code does, and we tried our best to make high-level 'why comments' when needed. SSIO leverages 'why comments' when needed to clarify some of the more involved processes.

## Avoid Multiple If/Else Blocks

Most of our code did not require multiple if/else blocks, but when it did, and there were more than three cases, we used switch statements.

## Use Framework Features

The Front-End and API heavily use elements of their respective frameworks. There were several instances in our Front-End where we were using plain html tags in our React components instead of components from Material-UI. These were fixed in most places. The API has barely any code that is outside of framework features. In our API, we generated new endpoints using the NestJS CLI which routes imports and provides testing files.

# Non-Functional Requirements

## Maintainability

Since our code is going to be used and continually developed by Prairie Robotics, we tried our best to use best practices. All of the linting and formatting helps ensure that the code is readable, as well as using best practices for variable/file naming and comments. We have unit tests written for SSIO, and extensive integration tests written for our API. For the API and Front-End, almost all of our classes in use are inherited from library classes, and we don't use any static functions or singletons. Dependency injection in our API makes it very easy to use. Using NestJS for our API and NX for our mono-repo (containing the Front-End) give us hot-module reloading that assists with debuggability. We tried our best to follow SOLID principles, which also allows for simpler debugging as things tend to behave the way we expect them to. All of the configuration for our API is handled by `.env` files locally, and AWS Secrets when it is hosted. Prairie Robotics has met with us for regular meetings, code reviews, and code transfer. They have also been involved in feature decisions throughout the process and have a

great understanding of various technical choices we have made and why. This helps their understanding of our code and their ability to take over ownership of the code.

## Reusability

In our API, the use of NestJS and our Controller/Service/Repository architecture mean that for the most part, we are building custom functionality only when we need it. As much as we could, we attempted to abide by the Don't Repeat Yourself (DRY) principle. One example of code that we refactored into its own service is the validation of user input on the API, where essentially the same code was being run in every controller. We extracted this into its own `validationService` that is used in every controller. For our Front-End, using Material-UI for most of the basic components means that the majority of the components that we create are specific for their use case. When we needed a component in more than one place, we tried to abstract it into a component that is usable in both places. Additionally, the monorepo technology that we are using, Nx, allows for the creation of `library` modules that provide a home for any components or other bits of code that will be used in more than one sub-repo. **Code Snippet 9** shows our validation service. This code used to exist in every controller. Additionally, **Code Snippet 10** shows the library functionality of Nx.

## Reliability

Our API is hosted on AWS Lambda, so after around five minutes of inactivity, AWS will automatically stop the function and cleanup any resources. In terms of exceptions, NestJS offers an all-encompassing exceptions layer that returns a 500 Internal Server Error to the user if there is an uncaught exception. It also allows for throwing specific exceptions for other HTTPS status codes which we have implemented to catch various errors and provide more useful error messages. Our front-end is a statically hosted browser-based app, so when the user exits our page, the files are no longer running on their browser. SSIO is on the truck, and is only run while the truck is driving. SSIO requires good wireless data coverage which is provided by a cellular modem, but it has performed well in remote locations. When internet connection fails, SSIO's upload functionality idles until it returns so no data is lost. Our architecture is hosted on AWS which provides high reliability for our hosted services. We have tested our services which provides additional reliability and reduces the amount of uncaught bugs which may throw errors.

## Extensibility

In general, our code is modular and allows for easy replacement. The modularity also allows for easy creation of new resources, as they will almost always follow the same format as those previously created. The frontend components specifically can be easily replaced by different components without issue to other components.

## Security

Our Front-End authenticates the user with AWS Cognito, and grants them a JWT that allows access to our API. The signed JWT contains the user's username which is used to query their allowed roles from the database in the API. These roles are checked against specific roles per endpoint (check our RBAC document for more) to allow or block users from hitting our API.

We do not store the passwords of our users as they are handled by AWS Cognito. For all of the create or update endpoints on our API we are performing extensively tested input validation using the Joi library.

## Performance

Our API requests on the front-end and database accesses on the API use JavaScripts modern `async/await` to perform asynchronous tasks. Our API relies on real-time data, so caching our API requests for speed subtracts from the integrity of the displayed data. We were able to increase our performance by tuning the AWS Lambda function configuration it runs on to decrease response times by 75%. **Code Snippet 8** shows JS `async/await` syntax.

## Scalability

AWS Cognito allows for up to 50 000 monthly active users in its free tier, and our MySQL RDS database allows for automatic storage scaling with a simple config change. The database should not have any problems with data sizes far greater than 50 000. Our API lambda function currently would be around \$20 a month per one million requests. Some of the requests that are being sent to the API from the front-end would start returning massive results, so some refactoring or batch processing on the backend would be necessary to keep response times low. As well, adding indexes to the database and having multiple Lambda APIs with a potential load balancer function would allow for even more scalability.

## Usability

We believe that our Front-End is easy to use, and self explanatory. There are only a few interactable parts of the app, and four pages not including the sign-in page. In the future, adding accessibility features to the frontend would be a great addition. As well, implementing logging and various metrics to track the user experience in the front end could provide more valuable data to improve usability. Our API is built according to RESTful standards to the best of our ability, and we have autogenerated human-readable Swagger docs for all of the endpoints. SSIO is not meant to be interacted with by a person, and is installed on the onboard computers placed on the trucks.

## Object-Oriented Analysis & Design

### SOLID

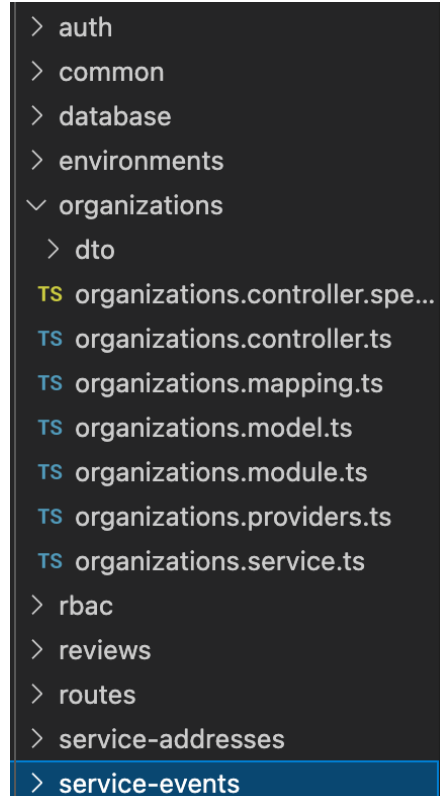
The use of the Controller/Service/Repository pattern on our API allows for easy object-oriented design. The main classes associated with an endpoint will be one of these three types. Additionally, NestJS makes great use of Dependency Injection, where it is simply part of the framework. Similarly, the 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. **Code Snippets 3&4** show dependency injection.

## Appendix

Code Snippet 1: Proper line spacing and TypeScript naming conventions. Lowercase for instances and uppercase for classes and types.

```
// Doesn't actually create an entity, just makes the shape the database will accept on a create
toEntity(organizationDto: OrganizationsDTO): Organization {
  return {
    id: organizationDto.id,
    name: organizationDto.name,
  } as Organization;
}
```

Code Snippet 2: Image of API folder structure clearly showing organizations with the controller, service and model files. This is the NestJS implementation of a controller/service/repository pattern



```
> auth
> common
> database
> environments
▼ organizations
  > dto
  TS organizations.controller.spe...
  TS organizations.controller.ts
  TS organizations.mapping.ts
  TS organizations.model.ts
  TS organizations.module.ts
  TS organizations.providers.ts
  TS organizations.service.ts
> rbac
> reviews
> routes
> service-addresses
> service-events
```



Code Snippet 3 & 4: Image of dependency injection in our NestJS API. OrganizationService is set up as Injectable, and the reference in the constructor of OrganizationsController automatically creates an instance for the controller.

```
export class OrganizationsController {
  constructor(
    private organizationsService: OrganizationsService,
    private validationService: ValidationService,
  ) {}
```

```
@Injectable()
export class OrganizationsService {
  constructor(
    @Inject(ORGANIZATIONS_REPOSITORY)
    private repository: typeof Organization,
    private mapper: OrganizationsMapping,
  ) {}
```

Code Snippet 5: A section of our constants.ts file, that includes various string constants for use in our API.

```
export const ROLES_KEY = 'roles';
export const APP_GUARD = 'APP_GUARD';
export const TEST = 'test';
export const DEMO = 'demo';
export const PROD = 'prod';
export const TRUE = 'true';
export const IS_TESTING = 'IS_TESTING';
export const SEED_GENERATED = 'SEED_GENERATED';
```

Code Snippet 6: NestJS implementation of a configService, using it to read in environment variables into the API.

```
const sequelize = new Sequelize({
  dialect: 'mysql',
  host: this.configService.get<string>('MYSQL_HOST'),
  port: this.configService.get<number>('MYSQL_PORT'),
  username: this.configService.get<string>('MYSQL_USER'),
  password: this.configService.get<string>('MYSQL_PASS'),
  database: this.configService.get<string>('MYSQL_DB_NAME'),
```

Code Snippet 7: Our enum used to designate the types of contaminants we are currently tracking.

```
export enum ContaminantEnum {
    BLACK_GARBAGE_BAG = 'Black garbage bag',
    BLACK_PLASTIC = 'Black plastic',
    CLAMSHELL_PACKAGING = 'Clamshell packaging',
    CLEAR_PLASTIC_BAG_OR_FILM = 'Clear plastic bag or film',
    GROCERY_BAG = 'Grocery bag',
    NON_BLACK_GARBAGE_BAG = 'Non-black garbage bag',
    PIZZA_BOX = 'Pizza box',
    POLYCOATED_CUP = 'Polycoated cup',
    POLYCOATED_FOOD_PACKAGING = 'Polycoated food packaging',
    SQUEEZABLE_TUBE = 'Squeezable tube',
    STAND_UP_POUCH = 'Stand-up pouch',
    STYROFOAM = 'Styrofoam',
}
```

Code Snippet 8: Image of our usage of async/await to retrieve a serviceAddress from our repository layer. Code snippet is from a service.

```
async findById(
    serviceAddressId: string,
    orgId: string,
): Promise<ServiceAddressDTO> {
    const serviceAddress = await this.repository.findOne({
        where: { id: serviceAddressId, orgId },
    });
    if (serviceAddress) {
        return this.mapper.toDTO(serviceAddress);
    } else {
        return null;
    }
}
```

Code Snippet 9: Our validation service, an example of a piece of code a lot of controllers were calling that was refactored into its own file.

```
You, 2 days ago | 2 authors (You and others)
@Injectable()
export class ValidationService {
  validate(body: any, joiSchema: Joi.Schema<any>): any {
    let validatedInputs;
    try {
      validatedInputs = Joi.attempt(body, joiSchema, {
        abortEarly: false,
      });
    } catch (error) {
      const errorReason = error.details
        .map((individualError) =>
          individualError.message.replace(/^[0-9a-z_\s]/gi, ''),
        )
        .join('. ');
      throw new BadRequestException(errorReason);
    }
    return validatedInputs;
  }

  pathParamsMatchBody(params: any, body: any): boolean {
    const paramNames = Object.keys(params);
    return paramNames.every((name) => params[name] === body[name]);
  }
}
```

Code Snippet 10: Our file structure in the monorepo. The apps folder holds the source code for the different applications, and the libs folder holds any reusable classes/functions/types.

```
> apps
> dist
▼ libs
  > components
  > data-access
```