OffBeatEscape

Code Documentation

University of Windsor

Objective

The purpose of this document is to highlight the various coding practices and conventions used in the development of the product OffBeatEscape.

Recipient - Dr. Aznam Yacoub

Author–Sidhartha Verma

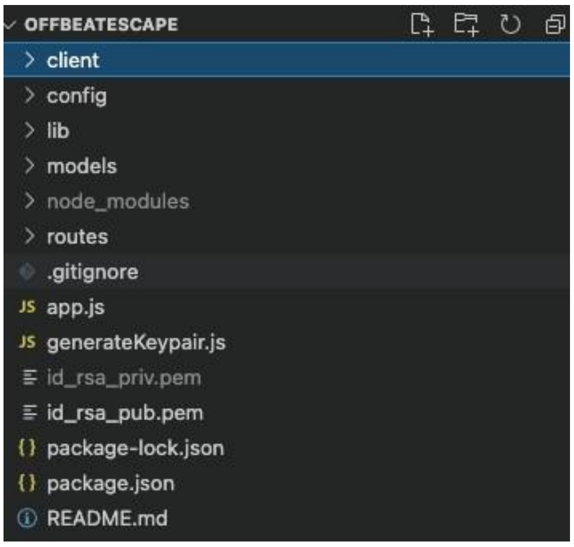
Written On – 13-04-2021

Index

Project Structure

The project follows a simple structure. It mainly consists of a front-end and a backend. The front-end part of the application is inside the folder called "client". For reference a screenshot of the project structure is attached.

Project Structure- Offbeat Escape Screenshot :



```
OFFBEATESCAPE
> client
> config
> lib
> models
> node_modules
> routes
.gitignore
app.js
generateKeypair.js
id_rsa_priv.pem
id_rsa_pub.pem
package-lock.json
package.json
README.md
```

The front-end is written in Angular and it follows all the latest coding conventions as per Angular's style guide. Every page of the application is segregated into components for better understanding. A central module called the "app module" is responsible for instantiating all components. It is also responsible for invoking all necessary external libraries used in the application. Config files are separated from main application and therefore lie parallel to "src"folder where main business logic and components exist. All dependencies of the project including external libraries are in "package.json" file.

The back-end is written in Node.js using the standard coding conventions. The back-end is structured by layering the components. App.js is the entry point of the application, routes folder has the application routes for individual pages. Config folder has the environment variables and connection details . The Lib folder has the utilities used in the application and the encryption logic for credentials .Node modules comprises of the packages used for the application .Model folder contains the database model details and generateKeypair.js has *public and private keypair which is used to save the current directory.*

Coding Conventions

Front-end

Front-end developed in Angular uses all its latest coding conventions and styles as specified in Angular coding style guide. Following are some of the conventions below:

1) **File Structure Conventions**: Represent a particular feature. Companions such as services are represented with the same file name. For ex: dashboard.component.ts and dashboard.component.scss

2) **Single Responsibility:** All components have a single responsibility. All services, CSS files etc. serve the same component. This is done for better readability and maintainability.

3) **File Names with dots and dashes:** Conventional type name have been used including .service, .component, .pipe etc.

4) **BootStrapping:** Bootstrapping and platforming logic is placed inside the file “main.ts”. Error handling is done inside the file incase bootstrapping of the application fails.

5) **Component Selectors:** Component selectors are using dashed case. For ex <app-header>.

6) **Pipe names:** Pipes have consistent names and named after the feature they represent. Pipe class names follow UpperCamel case while pipe names follow lowerCamel case.

7) **Folder by feature structure:** All folder represent a particular feature: For ex: dashboard folder represents dashboard page and login folder represents login page.

8) **Styles and templates inside their component:** Every component should have their own styling and template.

9) **Delegate complex component logic to services:** API calls and complex logic are written under service files. This is done to keep the component file clean.

Back-end

Back-end developed in Node follows the conventions specified in Node.js official documentation

1. Structuring the project:

i. Solution by components

Division of stack into self contained components that is easy to reason about

ii. Layering the components

Components composed of ‘layers’ – a dedicated object for the web, logic, and data access code

iii. Wrap Common utilities

Wrapping utilities by own code and expose them as npm packages

iv. Separate the app and server

Separating app.js for API declarations and server for networking concerns

v. Use environment aware, secure and hierarchical config

A perfect and flawless configuration setup should ensure (a) keys can be read from file AND from environment variable (b) secrets are kept outside committed code (c) config is hierarchical for easier findability.

2. Error Handling

i. Use Async-Await or promises for async error handling

Usage of a reputable promise library or async-await instead which enables a much more compact and familiar code syntax like try-catch.

ii. Distinguish Operational Vs programmer errors

Operational errors (e.g. API received an invalid input) refer to known cases where the error impact is fully understood and can be handled thoughtfully. On the other hand, programmer error (e.g. trying to read an undefined variable) refers to unknown code failures that dictate to gracefully restart the application.

iii. Exit gracefully

Smooth restart of application when unknown error occurs

3. Code Style Practices

i. Use ESLint

ESLint can automatically fix code styles, other tools like prettier and beautify are more powerful in formatting the fix and work in conjunction with ESLint.

ii. Node.js specific plugins

add Node.js specific plugins like eslint-plugin-node, eslint-plugin-mocha and eslint-plugin-node-security.

iii. Start a Codeblock’s Curly Braces on the Same Line

The opening curly braces of a code block should be on the same line as the opening statement

iv. Separating the statements pro

ESLint to gain awareness about separation concerns. Prettier or Standardjs can automatically resolve these issues.

v. Create different routers for different api’s.

Rather than implementing all the routes in a single index.html file, use routers to organize the code. For example, all the requests to user collection are maintained in user router. Similarly, all the api's for the post collection are in posts router.

4. Security practices

i. Implement strong authentication.

Used a secure library (BCrypt) to store hashedpasswords.

ii. Avoiding errors that reveal too much

Don't give exact details in error messages which can reveal user information or information such as paths, or other libraries in use.

Also, wrap the routes with catch clause so the Node.js don't crash when error is encountered. This will stop any one from crashing the app.

iii. Extract secrets from Conifg and encrypt

Never store plain-text secrets in configuration files or source code. Instead, make use of secret-management systems like Vault products, Kubernetes/Docker Secrets, or using environment variables.

iv. Prevent brute force attacks

A simple and powerful technique is to limit authorization using IP address

5. Production Practices

i. Use Monitoring

Monitor to ensure a healthy state – CPU, server RAM, Node process RAM , the number of errors in the last minute, number of process restarts, average response time

ii. Use smart logging

This is used to increase transparency

iii. Lock dependencies

Use npm config files, .npmrc, that tell each environment to save the exact (not the latest) version of each package in order to maintain identical code across all environments