DESING DOCUMENT

honeybadger-inc

- Joel Rauma, jr425042
- Shamsur Raza Chowdhury, 50359798
- Patrik Salmensaari, 150987125

Contents

Idea for software:

We discussed about the types of APIs we are familiar with and which ones we would like to use. Each of us conducted some research on APIs and then decided it would be a good idea to focus on those connected to Tampere. We reviewed the Tampere APIs together and found that some related to traffic were particularly interesting. Each of us did further research on how to combine these APIs, and during this time, we used AI to assist us. We consulted an AI for suggestions on which APIs to use and how to visualize the data fetched from them. We discussed the AI-generated ideas and how we could implement them. Ultimately, we decided to gather data from Tampere's traffic cameras, incidents, and roadworks, and display this information on a map. We got the idea to display this data from the AI.

Prototype:

We decided to use Figma Prototype because it's easy to make and then we get a good idea of how the transitions could work. In addition, it is easy to make changes to the Figma or to sketch different ideas. In the prototype, the idea is that the bottom has a map on which the data will be displayed. Both datas are shown on the same map and the map always shows an icon next to the event (camera or incident). If you click on that point, more detailed information will be shown to the user. For example, the time, what kind of situation it is, a possible picture, etc.

A sidebar is added to the page, from which the user can limit the events he wants to see, for example, if the user wants to see only road works, it can be selected from the page.

The user can also see a list view, where all events are listed on a separate page, on the page you can see all the detailed information and you can browse them.

For later in this project we decided to remove list view from our application since we noticed that it wouldn't add anything new or make the information more readable for user.

Link to Figma prototype: https://www.figma.com/design/g94JTduJzX31Xx6gCJk2id/SW-Design-course-Prototype?node-id=0-1&t=e7FsKeaehl3CtIJ0-1

APIs used:

- Tampere Traffic Incidents and Roadworks API
    - Link: https://data.tampere.fi/data/en_GB/dataset/tampereen-kaupungin-liikennetiedoterajapinta
    - Documentation: https://wiki.itsfactory.fi/images/b/b1/Infotripla_D2Light_City_documentation_v1_1.pdf
- Tampere traffic camera API
    - Link: https://wiki.itsfactory.fi/index.php/Tampere_traffic_camera_API
    - Documentation: https://traffic-cameras.tampere.fi/#/
    - Request URL: `https://traffic-cameras.tampere.fi/api/v1/cameras`

- Possible third API
    - We have designed the software so it would be quite easy to add more API:s to it. If we want to add more API we just need to make fetch functions to those and classes were we manage API data. We also need to add components to view so data will be displayed to user.

LLMs Used:

- ChatGPT 3.5 and 4.0
- Perplexity.ai
- Claude AI
- GitHub Copilot

LLMs Usage:

- In writing the code, we have used GitHub Copilot in the IDE.
- To debug the application and implementing major changes, at time we have used Claude 3.5 Sonnet. Usually by feeding the entire code base into it and then asking questions like, “How to make this controller read a query string?” and as such.
- We have used ChatGPT to write some of the code in the TrafficIncident component, although mainly it was after all done by hand. The AI recommended using static classes withing the main class file, which I found to not work, and be overly confusing.
- In TrafficCamera components, Perplexity.ai was used to ask "what could I have done better or more efficiently"
- ChatGPT was used in fixing bugs from the code.
- In the design part, ChatGPT assisted in getting started with the documentation with queries like “How should I start the documentation for the following code snippet?”, but the text was naturally overly verbose and did not focus on the concrete things, so it was rewritten. Despite of this, it did speed up the development and documentation by giving a starting point.
- Chat GPT was used to fix grammar issues after the first version of documentation was written.

High level description of the design:

The MVC design pattern is a widely used architectural pattern that separates an application into three main components: **Model-View-Controller (MVC)**. Here’s how this pattern can be applied to traffic camera and traffic incident application.

- **Model**: Manages the data and business logic. it will handle API calls to fetch traffic data, such as camera presets, locations, and traffic incidents.
    - **Implementation:**
        - Classes like `TrafficCamera`, `Coordinates`, `Geometry`, `Location and SituationRecord` can be part of the Model. These classes represent the data structures and handle the business logic related to traffic cameras and incidents.

        - The Model will also include methods to fetch data from APIs, process the data, and update the internal state accordingly.

- **View**: The View represents the user interface and is responsible for displaying the data provided by the Model. In your application, it will display the map and traffic information. View will be done using web and Sprinbootframework. It will receive data from the Controller and update the display accordingly.
- **Controller**: The Controller acts as an intermediary between the Model and the View. It processes user inputs, updates the Model, and instructs the View to display the updated data.
    - **Implementation:**
        - The Controller will handle user interactions such as clicking on a camera or requesting traffic incident data.
        - It will call the appropriate methods on the Model to fetch or update data and then notify the View to render the updated information.
        - There will be two classes, TrafficCameraController and `TrafficIncidentController`

Benefits of using MVC and why we decided to use it:

- **Separation of Concerns:** Each component has a clear responsibility, promoting better organization and easier maintenance. The Model handles data-related logic, the View manages user interface elements, and the Controller orchestrates interactions between them
- **Scalability:** This modular approach simplifies development and enhances scalability as new features can be added with minimal impact on existing code. This is important thing if we want to add new API later.
- **Reusability:** Components can be reused across different parts of the application, reducing code duplication.
- **Testability:** The separation of concerns makes it easier to test each component independently.
- **Why:** We decided to choose MVC because with it, making code is clearer, even if the number of classes grows large, it is easy to add new APIs afterwards and the code is easy to read. By structuring application using the MVC design pattern, we ensure a maintainable architecture that is scalable and easy to extend.

Summary

By structuring the application using MVC, each component has a clear responsibility, promoting better organization and easier maintenance. The Model handles all data-related logic, while the View manages user interface elements, and the Controller orchestrates interactions between them. This modular approach not only simplifies development but also enhances scalability as new features can be added with minimal impact on existing code.

User Interface

The frontend architecture combines server-side rendering with interactive client-side features, utilizing three main technologies:

1. **Thymeleaf Integration**
    - Serves as the primary template engine
    - Enables server-side rendering of dynamic content
    - Handles data integration between controllers and views
2. **Leaflet.js Map System**
    - Powers the interactive mapping functionality
    - Manages geographic data visualization
    - Handles marker placement and interaction
    - Provides popup functionality for information display
3. **HTML/CSS Structure**
    - Provides consistent styling across views
    - Manages layout organization

View Organization

The application's views are organized into two main categories:

Camera Views

1. **Map View**
    - Displays an interactive map showing all camera locations
    - Shows preview images when cameras are selected
    - Provides navigation to detailed views
2. **Details View**
    - Shows comprehensive camera information
    - Displays full-size camera images
    - Presents technical specifications

Incident Views

1. **Map View**
    - Shows active traffic incidents
    - Visualizes incident severity through color coding
    - Displays incident areas and affected routes

2. **Details View**
    - Presents complete incident information
    - Shows temporal data (start/end times)
    - Lists related traffic advisories

Controller Integration

The frontend-controller integration follows a clear pattern:

1. **Traffic Camera Controller**
    - Manages camera-related view routing
    - Handles data preparation for camera displays
    - Controls camera detail access
2. **Traffic Incident Controller**
    - Coordinates incident view presentation
    - Processes incident data for display
    - Handles incident detail routing

Dependencies

Maven

Maven is a build automation and dependency management tool for Java projects. It simplifies project setup and management by defining project structure, dependencies, and build processes in a single configuration fil, the pom.xml in our git repository. Maven automatically downloads libraries, plugins, and required resources, ensuring consistency across developers and their development environments, and makes it easy to get the application running from source file by automatically setting up the dependencies, instead of the developer or user needing to mind or manually importing and downloading libraries. We have used Maven as our default build system in this project.

Spring Boot

The application leverages Spring Boot 3.3.4 as its foundational framework, chosen for several key advantages:

1. **Application Configuration**
    - Autoconfiguration handles basic setup
    - Properties-based configuration enables easy environment management
    - Built-in application server (embedded Tomcat)
    - Streamlined dependency management through starter packages

2. **Dependency Injection**
    - Constructor-based injection for services
    - Lifecycle management of application beans
    - Clear separation of concerns through Spring's IoC container
3. **Web Layer**
    - MVC pattern implementation
    - RESTful endpoint management
    - Thymeleaf template integration
    - Static resource handling

Why Spring Boot for this project?

- Rapid development capabilities since our members were familiar with web technologies and wanted to explore Spring Boot as it is
- Extensive community support
- Comprehensive documentation

Spring WebFlux Implementation

The application utilizes Spring WebFlux for some of its reactive programming capabilities:

1. **Reactive Web Clients**
    - Non-blocking HTTP calls to external APIs
    - Better handling of long-lasting connections

Why WebFlux for this project?

- Handles external API calls efficiently and integrates well with Spring Boot making it a no brainer choice.

Lombok, Getters and Setters in code

We decided to use public library named Lombok to create Getters and Setters automatically.

- Read more: https://projectlombok.org/
- Lombok in overall
    - The Lombok library is a powerful tool in Java development that automates the generation of boilerplate code, including getters, setters, constructors, and other methods, making code more efficient, maintainable and readable.
    - Lombok ensures that getter and setter methods follow a consistent naming convention and structure, which is in line with the JavaBeans standard.
- Why we choose Lombok:
    - It is easy to use and make code more readable.

- We can focus on designing when we don't need to waste time writing Getters and Setters

API data handling

Idea of how we designed to do Response classes for both API.

1. **TrafficCameraResponse**
   - Purpose is to represent the overall response from Tampere Camera API.
   - **Attributes**
     - `List<CameraData> results`: A list of camera data objects.
     - `Meta meta`: Metadata about the API response.
   - **Methods**
     - Getters and setters for each attribute using Lombok.

2. **TrafficIncidentResponse**
   - Purpose is to represent the overall response from Tampere Traffic Incident API.
   - **Attributes**
     - `String modelBaseVersion`: The version of the model used to generate the response.
     - `SituationPublicationLight situationPublicationLight`: A lightweight representation of the situation publication.
   - **Methods**
     - Getters and setters for each attribute using Lombok.

APIs:

Traffic camera API

Overview

Design for integrating the Traffic Cameras API of Tampere. The API provides data about traffic cameras, including their locations and presets. The goal is to create a structured approach to fetch and manage this data within our application. The idea is to show the information on the map at the point where the camera is in the environment. By clicking on the icon on the map, it will show the user more detailed information and the image / images that the API provides of that location.

Request URL: `https://traffic-cameras.tampere.fi/api/v1/cameras`

Data Structure

The API response will be parsed into several classes to facilitate easy access and manipulation of the data. Below are the key classes that will be created:

TrafficCamera

1. **CameraPreset**
   - Purpose is to represents a preset configuration for a traffic camera.
   - **Attributes**
     - **`String presetId`**: A unique identifier for the preset.
     - **`String presentationName`**: The name used for presenting the preset.
     - **`String imageUrl`**: The URL of the image associated with the preset.
     - **`String directionDescription`**: A description of the direction related to the preset.
     - **`ZonedDateTime latestPictureTimestamp`**: The timestamp of the latest picture taken with this preset.
   - **Methods**
     - Getters and setters for each attribute using Lombok.
2. **Coordinates**
   - Purpose is to Represents geographical coordinates.
   - **Attributes**
     - **`Double lon`**: The longitude value.
     - **`Double lat`**: The latitude value.
   - **Methods**
     - Getters and setters for each attribute using Lombok.
3. **Geometry**
   - Purpose is to Represents geometric details.
   - **Attributes**
     - **`String type`**: The type of geometry (e.g., "Point").
     - **`Coordinates coordinates`**: An object containing the geographical coordinates, latitude and longitude information.
   - **Methods**
     - Getters and setters for each attribute using Lombok.
4. **Location**
   - Purpose is to Represents the location details of a traffic camera or other geographical entities.
   - **Attributes**
     - **`Geometry geometry`**: An object containing the geometric details of the location.
   - **Methods**
     - Getters and setters for each attribute using Lombok.
5. **Meta**
   - Purpose is to Represents metadata associated with API responses.
   - **Attributes**
     - **`Object requestFilters`**: An object containing the filters applied to the request.
     - **`ZonedDateTime responseTs`**: The timestamp when the response was generated.
   - **Methods**
     - Getters and setters for each attribute using Lombok.
6. **TrafficCamera**

- Purpose is to represents a traffic camera and its associated details.
- **Attributes**
  - **`String cameraId`**: A unique identifier for the traffic camera.
  - **`String cameraName`**: The name of the traffic camera.
  - **`String location`**: The geographical location of the traffic camera.
  - **`List<CameraPreset> cameraPresets`**: A list of presets associated with the traffic camera.
- **Methods**
  - Getters and setters for each attribute using Lombok.

Traffic incident API

Overview

The traffic incident API fetches data about the different traffic incidents and roadworks ongoing in the City of Tampere. This data is used within our application to visualize the locations of the incidents, and possible roadblock and/or delays caused by them.

Request URL: https://traffic-incidents.tampere.fi/api/v1

The returned JSON is parsed into a nested structure of Java objects, especially since the JSON is heavily nested

TrafficIncident

1.**SituationReport**

This serves as the general root class of the API response, having the following attributes

Attributes:

- String modelBaseVersion: A version number of the report
- SituationPublicationLight situationPublicationLight: A next-level root class for containing the report collection itself and some meta information

2. **SituationPublicationLight**

This class serves as the “sub-root” class of the traffic incident API response.

Attributes:

- String lang: Language used in the reports as a language code, usually fi
- String publicationDate: The latest time when the API response has been updated
- PublicationCreator publicationCreator: An object which contains information about the publisher of the report
- List situationRecord: A List of different traffic incidents, discussed down below

3. **PublicationCreator**

An info object. The SituationPublicationLight object has one as an attribute, containing the following information about the creator of the report:

Attributes:

- String country: The country code of the creator of the report
- String nationalIdentifier: This has the name of the entity publishing the report. One is “Infotripla”

4. **SituationRecord**

This is the class containing a single incident. It has several attributes.

Attributes:

- String id: identification number
- String version: the version of the incident record used
- String creationTime: when the incident was created
- String versionTime:
- String startTime: When the incident was deemed starting. This can be before the creationTime.
- String endTime: When the incident was considered to be over.
- Type type: a container class containing the Type of the incident
- DetailedType detailedType: A container class containing the detailed type of the incident
- String detailedTypeText: More specific description of the incident, in the language of the report.
- String severity: The severity of the incident
- boolean safetyRelatedMessage: Boolean describing whether this is a safety related message.
- String sourceName: Name of the source where the data about the incident came from
- String generalPublicComment: A short description of the incident to the general public, containing guidance for avoiding the incident zone.
- String situationId: another identification number

5**. Type**

A wrapper class wrapping just one String, since the JSON is defined that way.

Attributes:

• String value: Value of the type

6**. DetailedType**

Another wrapper class containing the type of the incident in more detail, but still just one String.

Attributes:

• String value: Value of the type

7. **Location**

A location object, which contains a verbal description of the location, as well as a coordinate object.

Attributes:

• String locationDescriptor: A verbal description of the location of the incident
• CoordinatesForDisplay coordinatesForDisplay: An object containing the latitude and longitude of the incident.

8. CoordinatesForDisplay

A wrapper class for containing coordinates in latitude and longitude format.

Attributes:

• double latitude: Latitude of the incident
• double longitude: Longitude of the incident

UML diagram

UML diagram for the core classes has been published in github.

Self-evaluation

We have been able to implement the basic idea very well, which we created in connection with the prototype. The only big difference compared to the prototype is that we have decided to completely omit the list view, as we found that it does not provide the user with new information or data more clearly.

In the demo version, the APIs are in different views, but this is a conscious choice for the demo version, as it allows us to examine the functions and data presentation formats of individual APIs more easily. In the final version, the data will be displayed on the same map.

The current version lacks a sidebar section, but it is planned to be implemented later. The sidebar will be implemented as planned, with various parameters that allow the displayed data to be filtered.

We have decided to add the Lombok library to the work according to the previous plan, which simplifies coding. This has been explained before.

Unlike the original plan, we also did not implement the program using Java FX but switched to the SpringBootframework, as implementing the current UI sections with Java FX would have been very challenging. The current library gave us better opportunities to implement the desired idea with less effort.

We have been able to implement the data retrieval and processing of the APIs according to the original plans, as the data was reasonably clear in our opinion, making it possible to design it qualitatively.

We are not entirely sure if we will need to change some of the planned implementations in a later phase if the coding proves too challenging, but at this stage, we believe we can do everything as planned.

We have not implemented a more detailed design of tests, but we will conduct tests for the most essential classes.

What next?

Later we will implement unit test for java classes

fix bugs from the code

do sidebar

> There will be possible to choose different parameters to show data, example type of roadwork, only cameras, only incidents, only somekind of incidents etc.

fix how data is shown to user like, ? will be replaced with ö å or ä

Time of the data will be replaced, 2024-10-08T21:00:00Z To 10.8.2024 21:00.

Maybe some other things but we can’t know now, those will be documented later.