DESING DOCUMENT

honeybadger-inc

- Joel Rauma
- Shamsur Raza Chowdhury
- Patrik Salmensaari

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

As the project has progressed, we have been compelled to make some minor changes to the prototype out of necessity. Additionally, some changes have been made because we felt they would improve the app.

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
- Weather camera API
    - Added after second TA meeting, since traffic camera API was down!
    - Link: https://www.digitraffic.fi/tieliikenne/
    - Kelikamerat API used
    - Documentation: https://tie.digitraffic.fi/swagger/openapi.json
    - More info: https://www.digitraffic.fi/
    - Request URL: https://tie.digitraffic.fi/api/weathercam/v1/stations/{id}
- Possible fourth 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.
- Some of the javadocs has been created with a help of Chat GPT.
- Chat GPT has been used to create model classes in WeatherCamera, meaning Meta, Feature and Station
- Chat GPT has been used to assist in creating some of the test cases for the Incident API classes

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.

Other design patterns:

Dependency Injection Pattern

Spring implements dependency injection by automatically providing objects with their required dependencies. In our application, this is primarily achieved through constructor-based injection, where dependencies are declared in class constructors. This promotes loose coupling and makes the code more maintainable and testable.

Singleton Pattern

Spring manages singletons through its component annotations. The application uses several stereotypes:

- Controllers are annotated with @Controller for handling web requests
- Services are marked with @Service for business logic

- Configuration classes use @Configuration for application-wide settings These components are maintained as singletons by the Spring container.

Observer Pattern

The application implements the observer pattern through Spring's reactive programming support. This is evident in the service layer where WebClient and Mono types are used for asynchronous operations and event handling in API communications.

Template Method Pattern

This pattern is implemented through Spring Boot's auto-configuration mechanism. The application extends core Spring Boot functionality while allowing for customization of specific behaviors through configuration classes and properties.

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 three main categories. We decided to place each API in its own view, as we noticed that this would be much clearer for the user. If all the data were displayed on the same map, there would be too many icons, making the app difficult to use. We also decided to add a home page, as it provides users with access to every view. These are the main differences in view component between the prototype and the final version.

Home view

Welcome page for application, short information about app. Includes link to every view.

Camera Views

Camera views are same for weather camera and traffic camera.

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
3. **Weather Camera Controller**
    - Handles GET requests for weather camera map and details
    - Prepares data for displaying weather cameras
    - Controls access to specific weather camera details.

4. **Home Controller**
    - Handles requests to the root URL and directs them to the appropriate view.
    - Returns the view name for the index page

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

Http components

- **Apache HttpClient 5**
    - This library is an advanced HTTP client that supports the latest HTTP protocol standards, including HTTP/1.1, HTTP/2, and WebSockets.
    - It provides a rich API and powerful extensions, making it suitable for building applications that require robust HTTP protocol processing.
    - Key features include:
        - Support for asynchronous requests.
        - Improved connection management and pooling.
        - Enhanced SSL/TLS configuration and security.
        - Compatibility with the latest HTTP standards and protocols.
- **Apache HttpCore 5**
    - This library provides the core functionality for HTTP protocol handling, including low-level HTTP transport and message parsing.
    - It is a dependency required by Apache HttpClient 5 and is used internally for the underlying HTTP protocol operations

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.
3. **WeatherCameraResponse**
   - Purpose is to represent the overall response from Weather camera API.
   - **Attributes**
     - `Meta meta`: Metadata about the API response.
     - `List<WeatherCameraFeatures> results`: A list of 'WeatherCameraFeature' objects, each representing a weather camera's data.
   - **Methods**
     - Getters and setters for each attribute using Lombok.

APIs:

IMPORTANT

In the middle of the project, the TrafficCamera API service encountered problems. Maintenance was performed on the API, which affected the operation of our group. We decided that we will leave the TrafficCameraAPI in the app, but we will not focus on its development in the future with as much importance as the rest of the program, because we could not know how the API will change and when it will be operational again. The API may work or may have undergone

changes, but we didn't want to remove a potentially working part of the API that we had designed to be part of it. We decided to add third API WeatherCameraAPI to the app so we will have at least two working API if TrafficCameraAPI won't work. Few days before the deadline TrafficCameraAPI is still partly under maintenance, but some of the cameras are working.

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

Weather camera API

Overview

We hadn't originally planned to use this API service, but we had to add it, because the TrafficCamera API was down. That's why the API in question is not in the prototype or previous plans. The API in question works very similarly to the TrafficCameraAPI, so adding the API did not change the basic idea of the program.

The Weather Camera API is designed to provide data about weather cameras, including their locations, presets, and associated images. The goal is to integrate this API into our application to display the camera locations on a map and provide detailed information and images when a camera icon is clicked. We decided to add this API while Tampere traffic camera was temporary unavailable.

Data Structure

The API response will be parsed into several classes to facilitate easy access and manipulation of the data.

WeatherCameraPreset

- **Purpose**: Represents a weather camera feature.
- **Attributes**:
  - `String type`: The type of the feature.
  - `String id`: A unique identifier for the weather camera.
  - `Geometry geometry`: An object containing the geographical coordinates.
  - `Properties properties`: An object containing detailed properties of the weather camera.
- **Methods**:
  - Getters and setters for each attribute using Lombok.

Geometry

- **Purpose**: Represents geometric details.
- **Attributes**:
  - `String type`: The type of geometry (e.g., "Point").
  - `double[] coordinates`: An array containing the longitude, latitude, and optionally altitude.
- **Methods**:
  - Getters and setters for each attribute using Lombok.

Properties

- **Purpose**: Represents detailed properties of the weather camera.
- **Attributes**:
  - `String id`: The ID of the weather camera.
  - `String name`: The name of the weather camera.
  - `String cameraType`: The type of the camera.
  - `int nearestWeatherStationId`: The ID of the nearest weather station.
  - `String collectionStatus`: The status of data collection.
  - `String state`: The state of the camera.

    - `String dataUpdatedTime`: The timestamp when the data was last updated.
    - `int collectionInterval`: The interval at which data is collected.
    - `Names names`: An object containing names in different languages.
    - `RoadAddress roadAddress`: An object containing road address details.
    - `String liviId`: The LIVI ID of the camera.
    - `String country`: The country where the camera is located.
    - `String startTime`: The start time of the camera's operation.
    - `String repairMaintenanceTime`: The time for repair and maintenance.
    - `String annualMaintenanceTime`: The time for annual maintenance.
    - `String purpose`: The purpose of the camera.
    - `String municipality`: The municipality where the camera is located.
    - `int municipalityCode`: The code of the municipality.
    - `String province`: The province where the camera is located.
    - `int provinceCode`: The code of the province.
    - `List<Preset> presets`: A list of presets associated with the camera.
- **Methods**:
    - Getters and setters for each attribute using Lombok.

Names

- **Purpose**: Represents names in different languages.
- **Attributes**:
    - `String fi`: The name in Finnish.
    - `String sv`: The name in Swedish.
    - `String en`: The name in English.
- **Methods**:
    - Getters and setters for each attribute using Lombok.

RoadAddress

- **Purpose**: Represents road address details.
- **Attributes**:
    - `int roadNumber`: The number of the road.
    - `int roadSection`: The section of the road.
    - `int distanceFromRoadSectionStart`: The distance from the start of the road section.
    - `String carriageway`: The carriageway of the road.
    - `String side`: The side of the road.
    - `String contractArea`: The contract area.
    - `int contractAreaCode`: The code of the contract area.
- **Methods**:
    - Getters and setters for each attribute using Lombok.

Preset

- **Purpose**: Represents a preset configuration for a weather camera.
- **Attributes**:
    - `String id`: A unique identifier for the preset.
    - `String presentationName`: The name used for presenting the preset.
    - `boolean inCollection`: Whether the preset is in collection.

    - `String resolution`: The resolution of the preset.
    - `String directionCode`: The direction code of the preset.
    - `String imageUrl`: The URL of the image associated with the preset.
    - `String direction`: The direction of the preset.
- **Methods**:
    - Getters and setters for each attribute using Lombok.

Meta

- **Purpose**: Represents metadata associated with API responses.
- **Attributes**:
    - `String timestamp`: The timestamp when the response was generated.
    - Other meta fields as needed.
- **Methods**:
    - Getters and setters for each attribute using Lombok.

WeatherCameraResponse

- **Purpose**: Represents the overall response from the weather camera API.
- **Attributes**:
    - `Meta meta`: Metadata about the API response.
    - `List<WeatherCameraFeature> results`: A list of weather camera feature objects.
- **Methods**:
    - Getters and setters for each attribute using Lombok.

UML diagram

UML diagram for the core classes has been published in github. UML diagram won't include HTML or CSS code.

Data Flow in App

This section describes how the MVC structure supports the data flow and the roles of components in the Tampere Traffic App. The application follows the MVC architectural model (Model-View-Controller), where each component (controller, service, model, and view) is responsible for its clear task.

Data Flow and Components

1. **Controller**
   The controller acts as the traffic manager, receiving and processing user requests, calling services, and providing data to views, which generate the HTML response for the user.
    a. Example: When the user navigates to the /camera/map URL, the TrafficCameraController receives the request and calls the

TrafficCameraService to retrieve data (such as camera information and metadata).
   b. The controller passes the data to the view by adding them as attributes (e.g., trafficCameras and meta) for use with the Thymeleaf template.
2. **Service**
   The service class is responsible for the application's business logic and data processing. In this application, TrafficCameraService handles calling the external API, processing the data, and temporarily storing it.
   a. WebClient client: TrafficCameraService uses the WebClient client to make asynchronous API requests in the fetchAndStoreTrafficCameras() method, which fetches JSON-formatted camera data and metadata.
   b. Cache usage: The data is stored in internal variables (storedTrafficCameras and storedMeta) to make it quickly available for future requests, reducing the number of API calls and improving performance.
3. **Model**
   Model classes contain the data structure used by the application and form the internal data structure for the data returned by the API.
   a. Data description and processing: The TrafficCamera class and its associated helper classes, such as CameraPreset, Coordinates, Geometry, Location, and Meta, define camera information, coordinates, images, and other essential details.
   b. The model ensures that the data can be easily transferred to the controller and view in a logically structured manner. The model classes also ensure that the raw data from the API is easily accessible within the application's internal logic.
4. **View**
   The view uses the Thymeleaf templating library to generate HTML pages, ensuring that data presentation to the user is clear and visually appealing.
   a. Map view: The map view (cameraMapView.html) uses the Leaflet map library to visualize camera information. The camera data passed by the controller appears as markers on the map. Popup windows provide more detailed information, such as the camera's location and images.
   b. Camera details: The detailed view (cameraDetailsView.html) displays the details of a single camera, including the camera name, location, and the timestamp of the latest image.

Comprehensive Data Flow Example

1. **Request**: The user sends a request to the URL /camera/map, which is received by the TrafficCameraController.
2. **Service call**: The controller calls the getTrafficCameras() and getMeta() methods of the TrafficCameraService class to retrieve camera information and metadata.
3. **Data processing**: The service makes an API call using the WebClient client, processes the received JSON response, and stores the data in the storedTrafficCameras and storedMeta variables.
4. **Response to view**: The controller adds the camera data (trafficCameras) and metadata (meta) to the Thymeleaf view, which generates an HTML page displaying the map view with camera objects as markers.
5. **View rendering**: The Leaflet map library visualizes the camera data for the user on the map. The user can click on individual markers to see more information about the

camera in a popup window or navigate to a detailed view displaying the camera's detailed information.

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. We also changed the design so that every camera has its own view. Since we created a home view, we decided that every time a user refreshes the site or changes the view, the latest update is pulled using the API.

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.

In the second TA meeting we noticed that Tampere Traffic Camera API was temporarily unavailable. We decided that we can't take risk about the API won't work and did some reseach about similar API. We found that digitraffic.fi has a quite similar API and we added that API to our software. This made a little extra work for all of us and that is also the reason why the app might look a little different than the prototype.

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. There was some problems with tight schedule while adding WeatherCamera API and since TrafficCamera wasn't working.

We have implemented a basic unit test for data handling classes, but haven't done those for other classes. We wanted to focus on getting app working like we want and thought that unit test are not so importat.

We hadn't planned to do a Home view for application in prototype, but we changed our plan since we added third API to app. The Home view makes app easier to use and gives easy access to every view. We also removed the view which shows every datapoint.

We created a sidebar and some filters for Incident and Weather views, but didn't do this for Camera view since we were not sure if it works. Incident view has multiple filters, since Weather view has only one; this is due to tight schedule with the weather camera API. We also added a sidebar on top of map instead of right side of the map since, since we noticed that adding a sidebar was harder than adding it to the top of the map. Our original plan was to create two filters for both APIs, but we were only able to implement one for both of them. This was due to the difficulty in identifying filters that would provide significant value to the user.

We are satisfied with our final result, although some minor changes could still be made. The code could undergo more extensive testing. The user's last visited page could be stored in memory and opened if the user so desires. Potentially, the filters utilized by the user could be saved and applied in subsequent sessions. However, these are minor additional improvements that do not significantly add value for the end user.