Tampereen yliopisto

DESIGN DOCUMENT – RECIPE FINDER

COMP.SE.110 Software Design

Group members
Iivari Karrila K4377292
Jade Karrila 50304882
Emil Selroos 152112341
Pan Zhengyang 151782154

Introduction and high-level description

The designed solution is a Java desktop application with a Web frontend. The app is to function as a recipe finder which provides price details of groceries from the Statistics Finland API and gives recipe suggestions from Spoonacular API based on the ingredients chosen from the first one. Users can filter recipes further by dietary preferences, cuisines and nutritional information. Key functionalities include displaying grocery price trends, recipe filtering, and user profile management for saving preferences and favorite recipes. Our chosen architecture pattern is Model-View-Controller, making it easy to split the web-based part from backend, distribute tasks, and create both vertical and horizontal workstreams internally. Scalability, modularity and flexibility are additional benefits, which in this small-scale application are only partially noticeable.

The frontend is built with vite.js due to previous familiarity and being simple to combine with Java, providing a lightweight framework for UI design. Our web interface allows selecting ingredients and displays the price data graph based on data from Statistics Finland API, enabling users to filter food item categories. The frontend also facilitates recipe browsing and saving selected ones within the user's profile. It does not do anything else besides managing requests from the backend and formatting the View parts of the app.

The backend is built with Java Spring Boot, as it takes care of most of our initialization and is simple to set up. It also offers a variety of libraries that offer additional help with integration. The backend requests data from Statistics Finland API and retrieves recipes based on user-defined ingredients and filters, which are relayed to the frontend. It also manages business logic, data transformation, validation, and recipe filtering based on cuisine, diet and nutritional information. User profile data is stored in CSV format, which includes preferences and saved recipes. The backend is separated into services, controllers, models and repositories to create modular structure, enhancing maintainability and scalability.

The main functionality the Statistics Finland API provides is up-to-date price data of selected groceries and ingredients. After data is retrieved, it is then delivered by the backend to the Spoonacular API, which recommends recipes based on the chosen ingredients and other user-selected parameters.

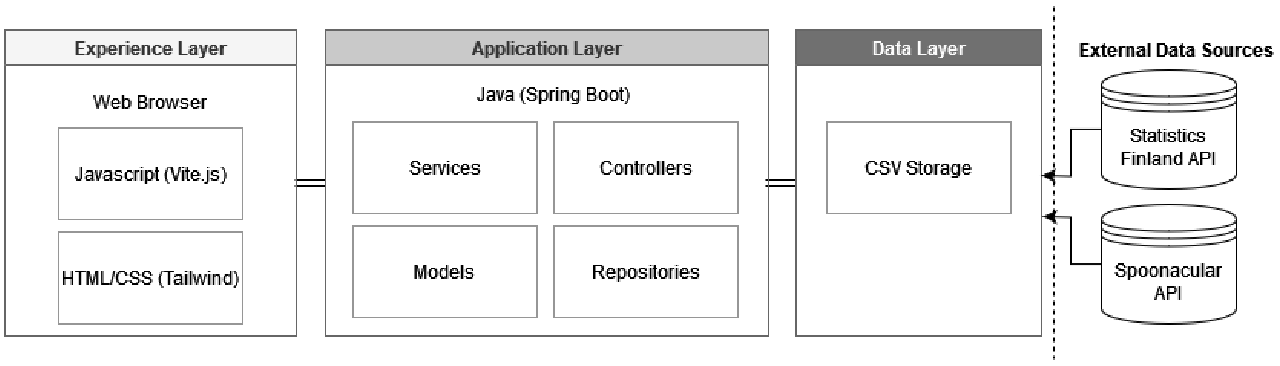


Figure 1. High-level system architecture diagram

Experience layer in Figure 1 corresponds to "View" in MVC, with Controllers taking care front-to-back integration and "Models" constituting for almost everything else.

Internal boundaries and interfaces

Our application relies on three different sources of data: (1) Spoonacular REST API, (2) Statistics Finland REST API, and (3) local CSV files that serve as a lightweight database. These data sources are controlled by different repositories which have methods to interact with the data. Controllers, which receive HTTP requests from frontend users, use repositories through services which handle processing of the data if needed. Repositories convert the data from external API or CSV files into models our application can more easily interact with.

We didn't want to include a proper SQL or NoSQL database in this project to keep it as simple as possible and to focus on the main topic of the course. It also makes it easier to test the application by fellow students.

The basic flow of our Recipe Finder application works as follows:

1. Controller receives HTTP request from Client/User.
2. Controller asks relevant Service to get the requested data.
3. Service layer gets the data from Repository and processes data if needed.
4. Controller responses to the Client with the data as HTTP response.
5. Frontend handles updating the user interface accordingly.

Example flow for getUserById method:

1. Client sends HTTP GET request to /api/profile/{id}.
2. UserController's method getUserById handles the request.
3. UserService's method getUserById is called by the controller.
4. UserRepository's method findById is called by the service.
5. In repository, relevant data is pulled from the CSV file.
6. Repository returns the user data to the service.
7. The service returns the user data to the controller.
8. The controller returns the user data in a JSON format and as HTTP response to the frontend.
9. Frontend updates the view based on the received data.

For the graph (food price dashboard), we use filtered data from Statistic Finland API, reconstruct the data elements as our custom internal objects with simplified structure in the backend. Then we send the Price Data objects to the frontend using our endpoints. The price dashboard graph in the frontend page utilizes **asynchronous loading design** to make sure other parts of the homepage are rendered properly.

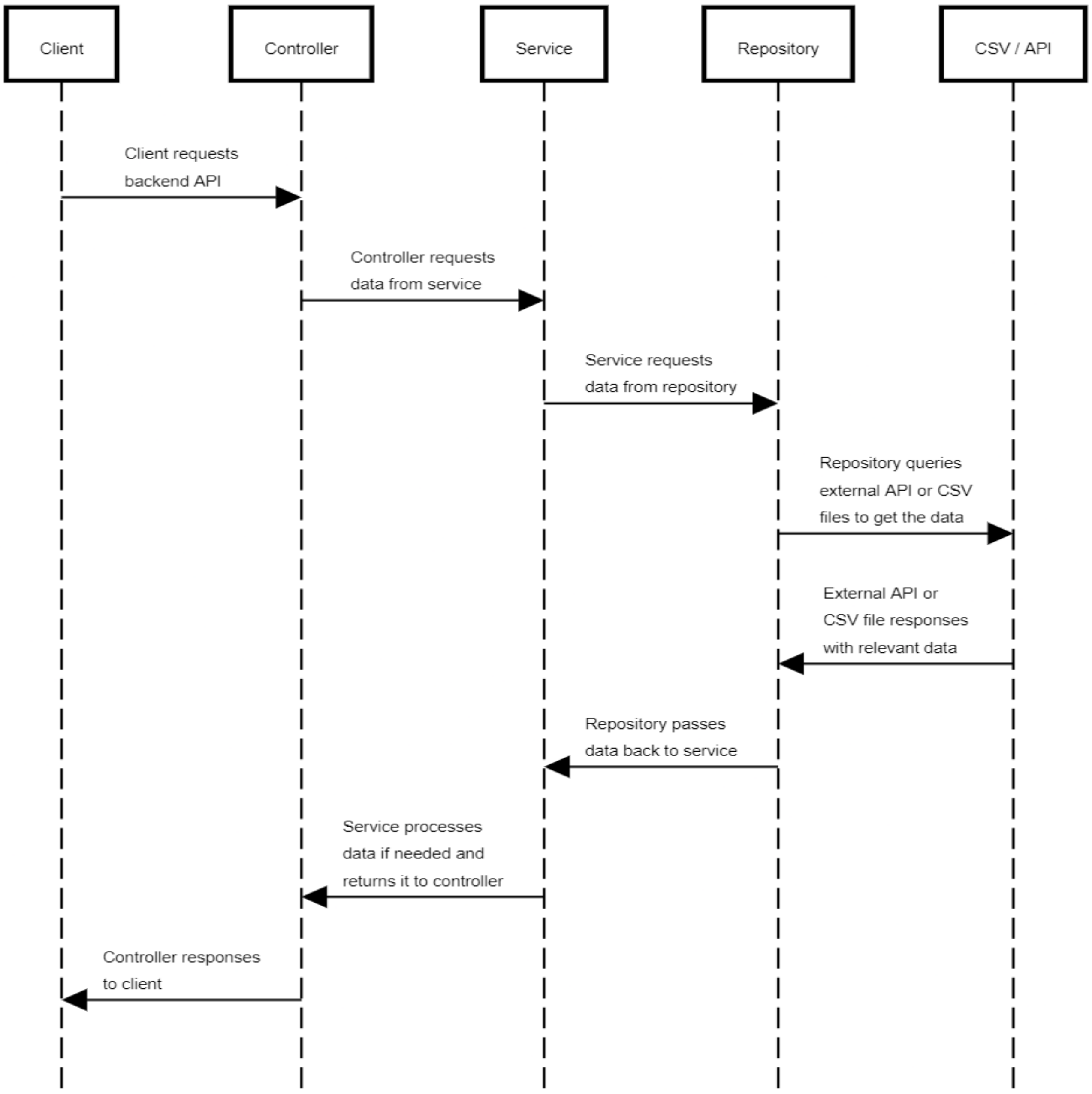

Client
Controller
Service
Repository
CSV / API
Client requests backend API
Controller requests data from service
Service requests data from repository
Repository queries external API or CSV files to get the data
External API or CSV file responses with relevant data
Repository passes data back to service
Service processes data if needed and returns it to controller
Controller responses to client

Figure 2. Data flow through different parts of application

Components and responsibilities

In this chapter we'll discuss each component of the application and what they do on a general level. In addition, we'll show the internal structure on a detailed level, as well as demonstrate our thinking and decision-making on the use of lower-level design patterns.

Backend:

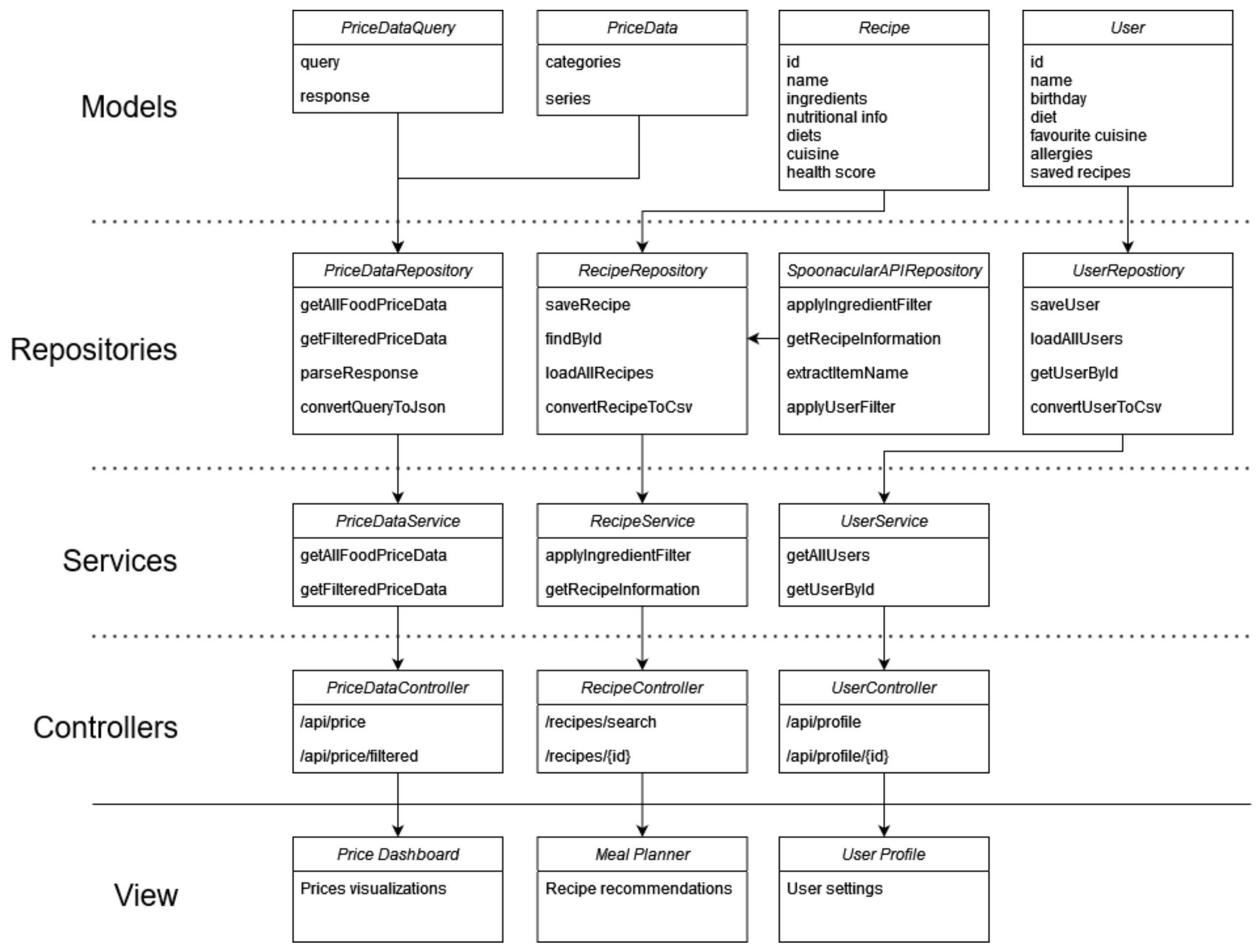
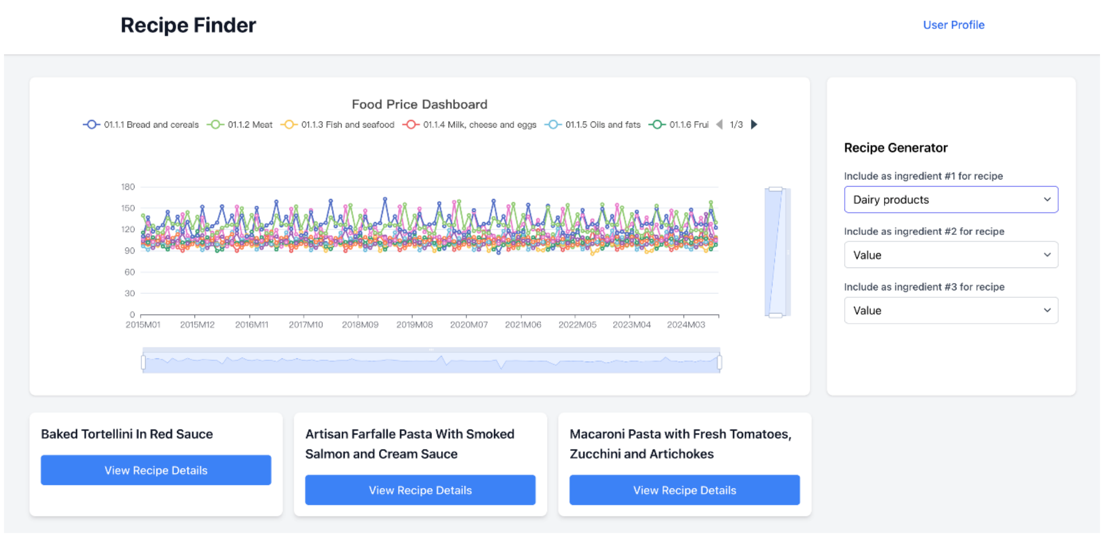


Figure 3. All classes, layers and methods visualized

Our Java backend implements the Model and Controller parts of the MVC pattern. As such, there are four key items developed using Java: controllers, services, repositories, and models. The full scope with all planned and implemented methods for each class can be seen above on Figure 2. Vertically there are three functional groups to consider: price data, recipes and user data.

Controllers

The three controllers manage the flow of data between the frontend and backend. There are one for the price data from Statistics Finland API, recipe data from Spoonacular API and one for managing user related data. They focus on managing incoming requests from the frontend, directing them to the appropriate services. The purpose of the controllers is to handle user interactions and route them to the necessary functions.



Include as ingredient #1 for recipe: Dairy products

Include as ingredient #2 for recipe: Value

Include as ingredient #3 for recipe: Value

Baked Tortellini In Red Sauce

Artisan Farfalle Pasta With Smoked Salmon and Cream Sauce

Macaroni Pasta with Fresh Tomatoes, Zucchini and Artichokes

Figure 4. The Price Data dashboard view on our front page

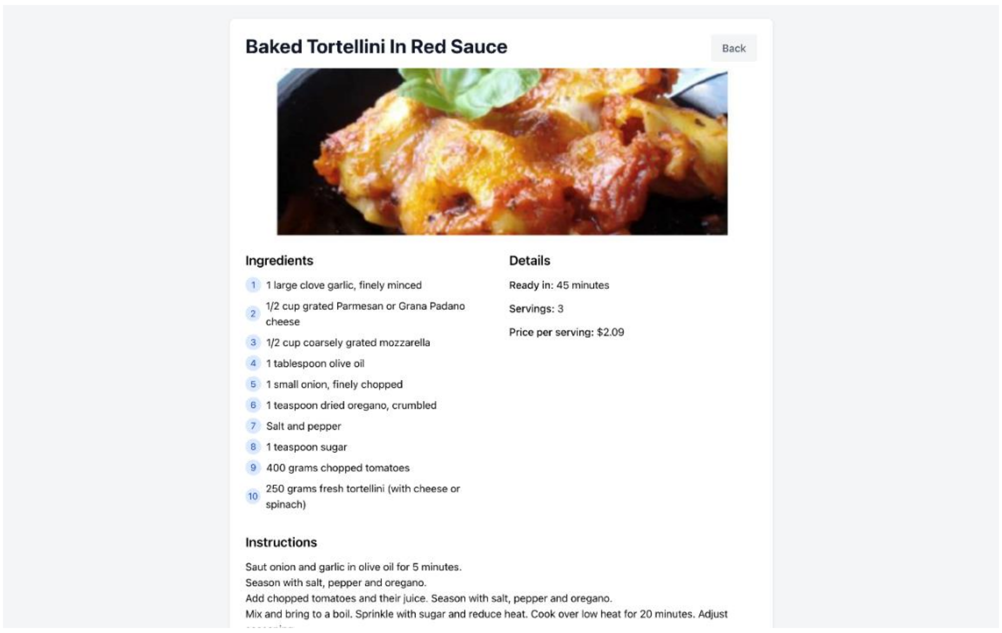
PriceDataController.java has one method to gather all recent price data on a high level, and another to apply specific filters. Example use cases of both below.

GET /api/price

GET /api/price/filtered?startMonth=2023M01&endMonth=2023M06&commodities=0111,0112

Description: This endpoint retrieves food price data filtered by optional parameters: startMonth, endMonth, and a list of commodities. The controller passes these parameters to the PriceDataService for custom query building.

These filters can be combined and expanded as needed, and the backend will take care of forming the JSON queries that it then relays to respective APIs. The same principle applies to both requests made towards Statistics Finland API and Spoonacular.



Baked Tortellini In Red Sauce

Ingredients

1. 1 large clove garlic, finely minced
2. 1/2 cup grated Parmesan or Grana Padano cheese
3. 1/2 cup coarsely grated mozzarella
4. 1 tablespoon olive oil
5. 1 small onion, finely chopped
6. 1 teaspoon dried oregano, crumbled
7. Salt and pepper
8. 1 teaspoon sugar
9. 400 grams chopped tomatoes
10. 250 grams fresh tortellini (with cheese or spinach)

Details

Ready in: 45 minutes

Servings: 3

Price per serving: $2.09

Instructions

Saut onion and garlic in olive oil for 5 minutes.
Season with salt, pepper and oregano.
Add chopped tomatoes and their juice. Season with salt, pepper and oregano.
Mix and bring to a boil. Sprinkle with sugar and reduce heat. Cook over low heat for 20 minutes. Adjust

Figure 5. Recipe details shown on the web interface

RecipeController.java can be used in a similar way:

GET /recipes/search?ingredients=chicken,garlic,tomato

Filters coming to the recipe search from the user settings (such as allergies) and the meal plan filters coming from the selections made in the frontend have been divided into two endpoints working in a similar way because these filters are not necessary applied every search.

GET /recipes/search/user?ingredients=chicken,garlic,tomato

GET /recipes/search/user/mealplan?ingredients=chicken,garlic,tomato

In addition there is a specific call for search details of individual recipes:

/recipes/{id}

Description: This endpoint retrieves detailed information about a specific recipe identified by its unique id. The recipe information includes details like ingredients, description, nutrition value, etc. The controller calls getRecipeInformation(id) in the RecipeService to retrieve the data.

GET /recipes/active-filters

Description: This endpoint returns the information about what filters are active without sending a new API call. This was implemented for testing purposes and for making it easier to see the logic in the frontend without overwhelming the API.

UserController.java manages user profiles, handling requests to retrieve all user profiles or a specific user profile by ID. It also manages possible changes to active user (which in our case is hard coded as user with ID = 1). The controller interacts with the UserService, which contains the business logic to retrieve user data from local CSV files acting as a lightweight database in this project.

GET /api/profile

Description: Allows displaying full set of users in the system for admin purposes, whereas

GET /api/profile/{id}

Description: The endpoint retrieves a specific user profile based on the user's unique ID, which will always be 1 in our MVP implementation. The id is passed as a path variable, and the controller uses the UserService to fetch the corresponding user. If the user is found, their profile is returned. Otherwise, a 404 Not Found response is returned.

PUT /api/profile/{id}

Description: This endpoint is used in the reverse order for changing user details, such as preferences, dietary constraints, and personal details. It only affects one user at a time, changing some or all of the parameters in the local database.

Services

There are three services in total, one mapping to each respective controller and only holding the available method initialization. They focus on encapsulating the business logic and coordinating data processing. This allows logic to be reusable and independent of the way data is stored or retrieved.

The role of the services in our design is to act as a layer of abstraction between the repository and the controller. Each service offers a range of methods for the endpoints to use, with or without parameters. The general principle is to separate functional topics (User, Recipe, Price data) vertically along the whole backend, ensuring that we can update them separately both horizontally and vertically with minimal conflicts.

PriceDataService.java acts as the intermediary between PriceDataController and PriceDataRepository. It encapsulates the food price data business logic by abstracting lower-level detail of data retrieval and filtering and handles the filtering logic for food price data, allowing users to specify optional parameters. The service delegates data retrieval and paring to the repository layer ensuring a clear separation between data access and business logic.

Methods: getAllFodPriceData() fetches all available food price data without filters and provides a unified interface to access comprehensive price data, ensuring controllers do not need to handle API specific. getFilteredPriceData() fetches price data filtered by specific months and commodities and encapsulates complex query-building logic and ensures controllers can retrieve customized datasets without directly interacting with repository details.

RecipeService.java handles the business logic related to recipes between RecipeController and SpoonacularAPIRepository. It provides methods to filter recipes based on ingredients, user preferences and meal plan settings, maintains and updates session-specific active filters using an internal ConcurrentHashMap, delegates API-related tasks, such as URL construction and integrates with the UserService to retrieve user preferences and apply them when filtering recipes. Ingredients used by the methods are provided by the price data API.

Methods: getMealplanREcipes() retrieves recipes filtered by ingredients, user preferences and meal plan settings. getUserRecipes() fetches recipes filtered by user preferences and ingredients. getIngredientRecipes() returns recipes solely based on the provided ingredients. getRecipeInformation() retrieves detailed information for a specific recipe using ID. updateActiveFilters() updates the internal record of active filters for the current session. getActiveFilters() returns the currently active filters as a JSON object for easier frontend integration.

UserService.java acts as the bridge between UserController and UserRepository. It provides methods to fetch and manage user profiles stored in a lightweight CSV file-based database, ensuring seamless interaction between the frontend and backend. It handles retrieval of user profiles, basic error handling and abstraction of repository logic.

Methods: getAllUsers() fetches all user profiles stored in the local CSV file. getUserById() retrieves a specific user profile by their unique ID.

The services use façade patterns to simplify the interaction with underlying repository methods, offering a single access point for retrieving price data. They also ensure flexibility and testability with dependency injection and separation of concerns. Concurrency management is also used to manage active filters safely across multiple threads in the RecipeService.

Repositories and design patterns

Repositories handle the interactions with data storage and provide a layer dedicated to data access. This ensures that data management is organized and can be easily modified. Repositories use a combination of patterns depending on what best suits each one. Just for instance the Price Data repository uses many patterns by itself:

For example, the PriceDataRepository.java has a method buildQueryWithOptionalMonths, which uses the **Builder Pattern** by building PriceDataQuery objects with nested queryItem and Selection objects. This helps in setting up the logic for complex query parameters based on the startMonth, endMonth and commodities. Thinking behind selecting this pattern was to make the codebase cleaner and to reduce errors as the setup is greatly simplified.

This pattern is also used in the RecipeFilter objects in order to achieve dynamic filter combinations in the combineFilters method of SpoonacularAPIReposity. The pattern is used to assemble and manage the set of filters for recipe searches based on user preferences, meal plans and selected ingredients. RecipeFilter object encapsulates all active filters in a cohesive manner ensuring reusability and separation of concerns. Builder pattern is also use in the logic how the API query URL is handled as parameters are appended by mimicking step-by-step construction.

PriceDataRepository also has createQueryItemWithValues method that follows the **Factory Method Pattern** to create and initialize QueryItem instances. We decided to use it here to centralize the setup of QueryItem objects and to reduce duplication, because it allows simplifying future adjustments to the item creation process.

parseResponse acts like a Template Method for parsing JSON responses from the Statistics Finland API. It provides a consistent parsing flow that could be reused or overridden if needed to handle variations in the response structure. This approach provides a structured way to interpret responses and populate PriceData objects while handling exceptions in a standardized way.

SpoonacularAPIRepository.java and PriceDataRepository classes act as Adapters between the application and external APIs. They translate internal method calls into specific HTTP requests and responses. This is our implementation of the **Adapter Pattern,** allowing us to easily swap APIs if needed, hence leading to enhanced modularity and flexibility. We chose the adapter pattern deliberately to adapt other parts of the design.

Last major pattern in the case of our Recipe Finder repositories are the **Singletons** (Java Spring beans uses them by default). They are not only limited to PriceData, but every class definition using @Service and @Repository annotations e.g. has Singletons running on the background.

Models

The models represent core data structures used throughout the application. The purpose of models is to format and configure the data relationships, making it easy to manage and use consistently across the whole application. The number of models was reduced in order to increase simplicity in the structure.

PriceData.java stores data from the Statistics Finland consumer price index API. Each price data object is associated with a list including relevant categories of food products, and another list for time series to draw filterable graphs.

PriceDataQuery.java was created to construct the queries themselves and to have a clear distinction between the data objects returned and query objects being sent to the API. The model has list of query items and the response as key fields.

Recipe.java represents a recipe, including the ingredients used in the recipe and nutritional information. The key fields are id, name, list of ingredients, diets, cuisines, health score and Boolean attributes fetched from the API is the recipe dairy free, gluten free, sustainable, cheap, popular and/or healthy. These are used for filtering different recipes. It is linked to Ingredient, Nutritional info and the User model as the user can save favorite recipes in a list.

RecipeFilter.java represents the meal plan filters that the user can set up in the interface as additional filters on top of the used ingredients and filters coming from the user profile. The model has the included and excluded ingredients and also tracks if the calories, protein or carbs filters are on. Key fields also include, cuisine, diet, isDairyFree, isGlutenFree and the min and max calories, protein and carbs.

User.java stores the user profile information. The key fields are id, name, birthday, diet, favorite cuisine, allergies and the list of saved recipes. It is connected to the Recipe model through the saved recipes.

Decision-making and Design reasoning

We had weekly design/sprint meetings each Saturday to discuss the tools, architecture and distribute implementation tasks for each respective week. In these meetings we decided to go with the MVC pattern, with the tools previously introduced, and focus on scalability, modularity and prioritize development of Models and Controllers first, focusing on the View later.

Technology Choices

Backend: Spring Boot

- Provides robust framework for building RESTful APIs with built-in dependency injection
- Excellent integration capabilities for external APIs (Statistics Finland and Spoonacular)
- Strong support for MVC architecture pattern
- Team members' prior Java experience makes it a practical choice

Frontend: Vite.js (React)

- Modern build tool offering faster development of new features
- Simplified state management mechanisms
- Team's existing web development experience makes it a suitable choice

Data Storage: CSV Format

- The format is simple, lightweight and easy to work
- Easy integration to Java backend through common Java libraries
- Data volume of the application is estimated to be low
- Minimal setup and hosting requirements, enabling rapid testing if needed

Scalability considerations

1. **Asynchronous Loading**
   a. Implemented for price dashboard graphs and recipe information
   b. Ensures responsive UI even with large datasets
   c. Prevents blocking of other components during data loading
2. **Modular Component Design**
   a. Separate controllers for price data, recipes, and user management
   b. Makes it easier to add new functionality
3. **Query Parameter Handling**
   a. Flexible filtering system for both price data and recipes
   b. Custom query building for optimal API interactions
   c. Reduces unnecessary data transfer

Frontend design patterns

The frontend of the Recipe Finder application follows a modern, modular design using React.js with Vite for fast builds and efficient state management. React’s component-based architecture ensures reusability, while hooks and the Context API handle local and global state seamlessly. Tailwind CSS simplifies styling with its utility-first approach, keeping the codebase lightweight and consistent. Features like asynchronous data loading, responsive design, and client-side routing improve user experience and performance.

Backend design patterns

As already mentioned, the backend has been organized based on general MVC structure, allowing separation into distinct layers. This decision was made uniformly and has served us well. Individual methods in the model and controllers use e.g., Builder, Factory, Adapter, and Singleton patterns as discussed. We didn't want to limit ourselves to only a selected few, so they are used where they offer adaptability, maintainability, or some other clear benefit. Each chosen use of a design pattern serves a very specific purpose, addressing some local challenges, such as the builder for price data simplifying complex queries, or adapter pattern allowing modular interaction with APIs.

Testing

The application testing plan focuses on ensuring that key components of the program are working as they are supposed to within the time limitations of the course.

CI/CD pipeline and unit testing

We wrote in total 14 tests for individual methods in the backend to test core functionalities, for example CSV data handling and recipe filtering. These unit tests verify that a method produces the correct output, being otherwise limited in their scope. Implementing them helps catching errors early in the development process, reducing the required effort for debugging.

- **Tool**: Junit, GitHub Actions

Additional tests

These are designed tests that we didn't have time to implement due to prioritization, but would have in a real-world project environment.

- API pinning to ensure the APIs are up and running
- Integration tests to check how the components work together
- Tools: Spring Boot testing support

Self-evaluation

Design Review:

Our initial design was reasonably high-level and based on our limited knowledge of software architecture patterns at that time. Considering what we know now, some elements we could have simplified (such as ingredient model) and others expanded upon already in the beginning. It would have also been beneficial to have a list of all possible patterns somewhere, and together map those to our methods/classes/components to discuss the approach more methodically.

However, all things considered, we did manage to follow the initial plan almost step-by-step throughout the implementation process, and it was immensely helpful in structuring work, distributing tasks, and keeping track of overall status. The benefits of thinking about architecture patterns have been very well demonstrated by this exercise, and so far, we are quite happy with the results.

Furthermore, the initial design would have been even more useful with some additional sequence diagrams showcasing in detail what the backend is supposed to do in each step, as well as having a full set of endpoints already decided from the moment the implementation began. The way we progressed was not as structured as it could have been.

Changes to Original Design:

Additions:

- New Controller, Service and Repository for user management was implemented to support user creation. This was in the initial plan, but not part of the documentation.
- PriceData model was split into two: PriceData.java and PriceDataQuery.java to manage query parameters and construct PriceData objects separately. This decision was based on the lecture about builders and factories, and was done to test them out, but also simplify the general querying process.
- CSV format for data storage was defined. Previously we thought about using some type of SQL storage, but CSV was deemed sufficient for this assignment. It also reduces the amount of effort required to manage more complex databases, although we are aware that CSV is not the most modern option.

Future expansion:

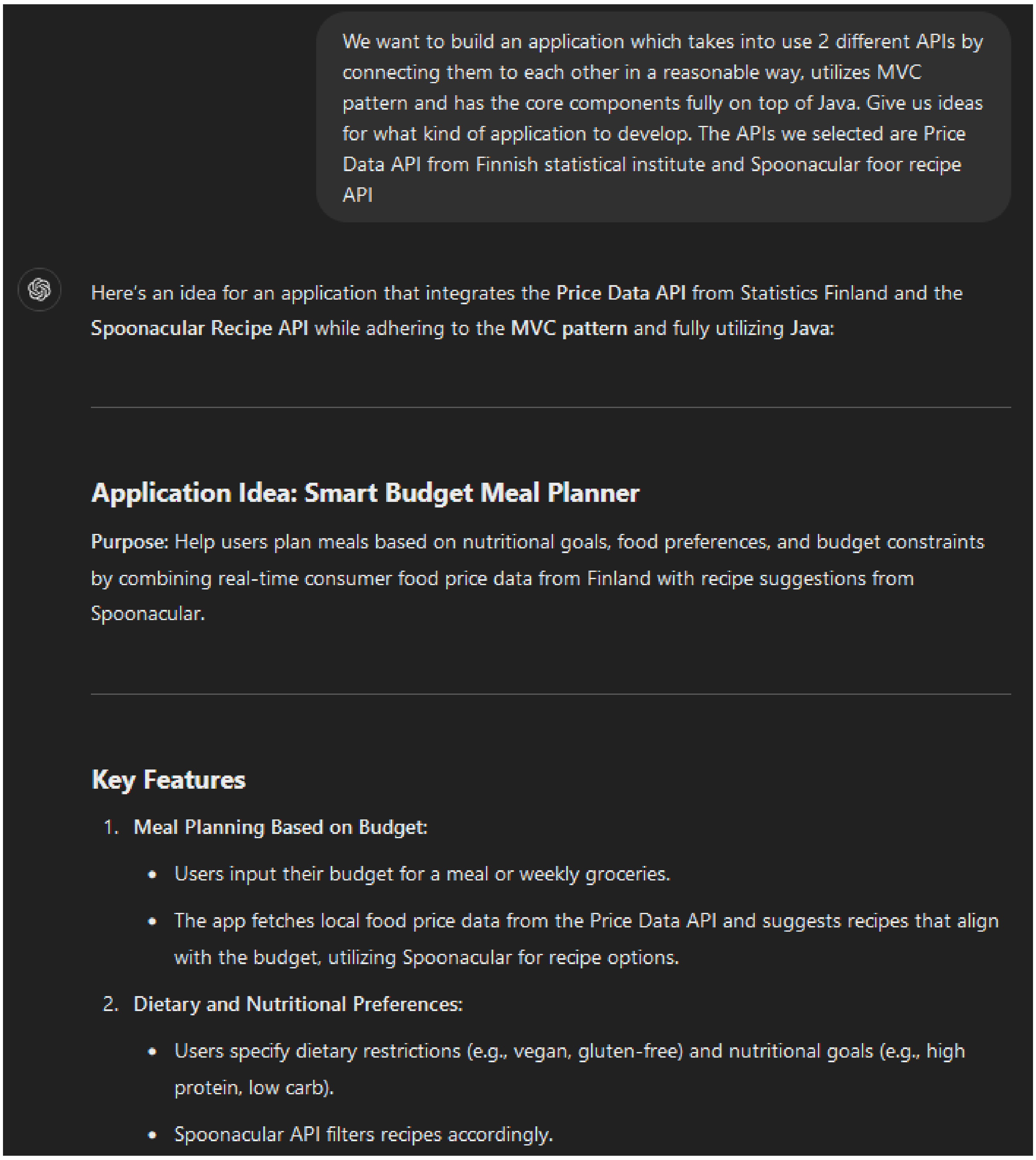
We have implemented most of the code required for the backend (Models and Controllers). Adding and expanding the now existing controllers, services and repositories should be rapid, but also simple. A few new endpoints we have already included for UserController to patch user data, methods to save recipes, and a method to PriceDataService to apply additional filters. Otherwise, we have a stable foundation for any continuous implementation built on top of MVC pattern. For example, due to the design decision to split the API repository and our local repository for recipes, we could easily switch to using any other recipe API if the old one would e.g. become depreciated. This is just an example of how the modularity of our design ensures scalability and flexibility.

The use of AI

What tools did you use, and how?

We used ChatGPT to prompt the idea for our group assignment, as well as the Copilot in VS Code to help with debugging, documentation, and general improvement of codebase. In the early research phase we also used ChatGPT to prompt how MVC pattern manifests itself in a typical Java codebase, and got information on how structure the folders, which components should be created and how to best maintain growing complexity throughout the course of the project.

Our prompt from the beginning of the project is shown below



We want to build an application which takes into use 2 different APIs by connecting them to each other in a reasonable way, utilizes MVC pattern and has the core components fully on top of Java. Give us ideas for what kind of application to develop. The APIs we selected are Price Data API from Finnish statistical institute and Spoonacular foor recipe API

Here's an idea for an application that integrates the **Price Data API** from Statistics Finland and the **Spoonacular Recipe API** while adhering to the **MVC pattern** and fully utilizing **Java**:

Application Idea: Smart Budget Meal Planner

Purpose: Help users plan meals based on nutritional goals, food preferences, and budget constraints by combining real-time consumer food price data from Finland with recipe suggestions from Spoonacular.

Key Features

1. **Meal Planning Based on Budget:**
   - Users input their budget for a meal or weekly groceries.
   - The app fetches local food price data from the Price Data API and suggests recipes that align with the budget, utilizing Spoonacular for recipe options.
2. **Dietary and Nutritional Preferences:**
   - Users specify dietary restrictions (e.g., vegan, gluten-free) and nutritional goals (e.g., high protein, low carb).
   - Spoonacular API filters recipes accordingly.

Figure 6. Prompt for coming up with the idea at the beginning of the course.

Small adjustments were made to consider the fact that we don’t have access to prices for the recipes themselves. Otherwise, the implementation idea came directly from ChatGPT and was implemented as suggested.

Approach was to first prompt it to suggest a list of possible APIs, from which it came up with movie APIs, stock picking APIs, and lastly the food recipe API. We decided to go with that, combine it with the Statistics Finland API and wrote the previous prompt based on that. In our opinion the process was better than direct prompting, which for all resulted in roughly the same exact suggestions, none of which were that good or original. We had slightly more control over the topics by first picking the tools, only then what to do with them.

During the project's implementation phase, we all used GitHub Copilot in VS Code to assist ourselves in creating the code. It is very hard and almost impossible to document these prompts as working with Copilot in terms of software development is very interactive and fast-paced. Copilot was especially helpful in debugging issues faced during the development. Copilot’s predictive editor was also very helpful tool and was in active use by our team. The use AI and especially Copilot was helpful and reduced time used to repetitive tasks and debugging of faced issues.

There are also risks related to the over-reliance use of AI which is always good to keep in mind. On the other hand AI tools are used daily by software developers working in the industry and should not be underestimated in universities; instead, their use should be encouraged as knowing how to use them is essential in working life now and in the future.

How we have met the requirements

Description on the fulfillment of the functional requirements of the course assignment:

1) The application must retrieve data from two separate third-party APIs

These are the Statistics Finland API and Spoonacular recipe API, which have been combined by sending the food categories from the former directly as part of query parameters to the latter.

2) The data from the services must be combined in a meaningful way and shown to the user

We achieved this by enabling the user to track recent food product prices by category and find matching recipes that use some of those categories, as well as consider personal preferences and dietary restrictions. The user can easily find ingredients, recipe instructions, and other details to come up with new ideas.

The price dashboard includes graphs as the main visualization, in addition to which there is the recipe catalogue and other parts of the user interface. Parameters such as timeframe, scale, and visible categories can be changed by the user.

3) The user can save preferences for producing visualizations

The user can save recipes to a csv file for later reference, as well his or her personal preferences and details. Preferences will be used in retrieving relevant recipes for the meal plan.

4) The design must be such that further data sources (e.g. another third-party API), or additional data from existing sources could be easily added

The MVC pattern is developed in a way that enables easily switching the APIs to new ones without major reforms to the code base. Furthermore, the backend is separate from the frontend, allowing separate development streams, testing pipelines, and hosting. Additional data can be added by creating new models and repositories, without affecting the existing ones.

Practical requirements:

• The use of design patterns and other principles has been described above

• Readability of code – The difficult-to-understand sections contain exhaustive comments, function naming scheme has been designed to be consistent across the codebase, and additional documentation can be found from this paper

• Use of version control – 2 GitHub repositories with feature branches and clear tagging

• Unit testing of crucial (core) components for Java has been implemented

• The use of AI (tools) in design and implementation of the project – Idea, documentation and debugging has been done using AI tools, mostly ChatGPT and Copilot