APPRENTICE TECHNICAL LOG DOC

TODO APP

Victor Hugo Vazquez Rojas

OVERVIEW – Welcome to my Apprentice Technical Log for my first breakable toy project during Encora’s Spark Program, the Todo App. This document is focused on breaking down the design, thought process, technical decisions, and a few detours I took along the way.

This was a project where we got to put our programming skills into action in order to create a full stack application. This project was also a great opportunity, at least for me, to learn new frameworks and technologies that I had never used before the project like Spring and some testing libraries like Jest and JUnit.

While I did get to make some decisions regarding structure and approaches, we had a strict set of requirements and technologies to stick to, which made it feel even more real-life, like dealing with client expectations or team agreements in a professional environment.

CONTEXT – Our hypothetical client needed a to-do list application to help users track daily tasks, prioritize important work, and understand their performance over time. At a first glance it seems easy, but when you start getting deeper into not only functional requirements but also non-functional requirements, it is more difficult, because it affects the way in which you have to structure your code and write it, and add some other things that you didn’t consider like documentation and testing.

Here's the high-level breakdown of what was expected:

* Core Functionalities like creating tasks, editing them, marking them as done/undone, filtering, sorting, and pagination.
* Nice-to-Have Features like background color indicators for due dates and strikethrough font for completed items.
* Technical Architecture had to be well structured and, in the case of the backend, had to be able to allow an easy transition to a real database instead of Java collections.
* And also, testing and documentation were part of the deal.

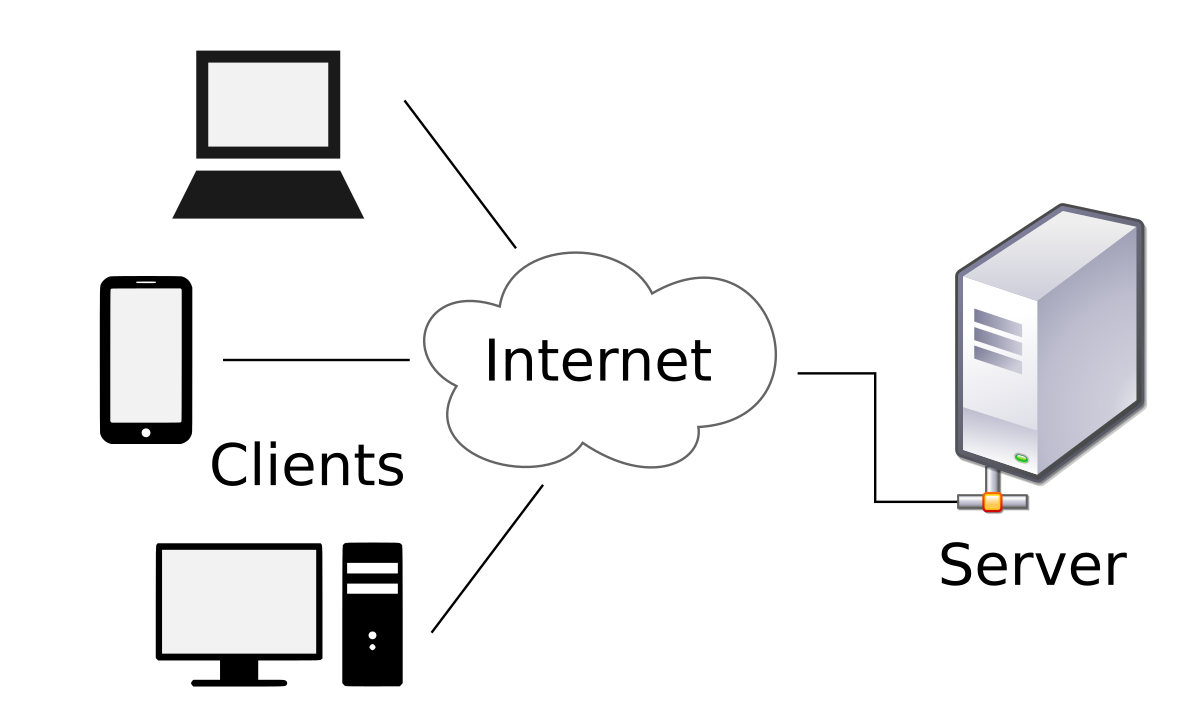
And, besides the functional requirements, this project was also about other important stuff happening behind the scenes that were also important like:

* Applying software engineering best practices like DRY, SOLID, and clean code.
* Using Git properly for version control.
* Communicating decisions through documentation and demos.
* Learning how to break down a project into manageable pieces, and iterate on it week by week.

Generally speaking, this was an exercise in building something not only functional but also maintainable, and aligned with real-world expectations, challenging you as a developer during the process.

SOLUTION (YOUR APPROACH) - **Big Picture**

In general, this was the architectural model that I used during the development of the application, which is the classic Client-Server model:

****

## BACKEND

### Tech Stack

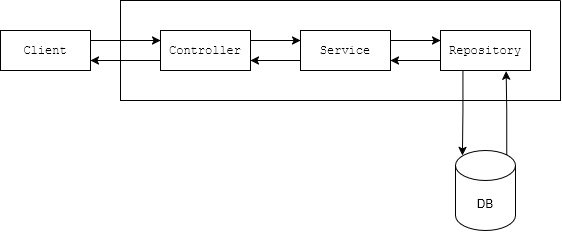
* Language: Java
* Framework: Spring Boot
* Build Tool: Maven
* Testing: JUnit
* Storage: In-memory (Java Collections)

### Backend Architecture

For the backend I applied a 3 layer architecture with a clear separation of concerns (Controller, Service and Repository):

* **Controller Layer** — handles routing and incoming requests.
* **Service Layer** — business logic lives here (validations, processing).
* **Repository Layer** — this layer manages the interaction of the application with the database. For this application it uses an in-memory storage with Java Collections, but it allows us an easy transition to a persistent database.

Additionally, I chose a package-by-feature structure, meaning all classes related to a specific domain (e.g. todos) live in their own package. This improves modularity and also keeps together all the things related, making the development process easier.

****

(The database in the diagram is just a representation as we are using Java Collections, however the assignment specified that it was important to consider an easy integration to a real database and it represents that future database and how the repository layer will interact with it)

### Backend Feature Implementations

The data structure that I have used in order to store the information is a List from the Java Collection, it allowed me to use the Java stream() feature and it made all the querying stuff easier.

#### Create To-Do

* **Endpoint:** POST /todos
* **Description:** Creates a new todo item with the specified text, priority, and optional due date. The todo is automatically marked as not done and assigned a creation date of the current timestamp.
* **Implementation:** The endpoint accepts a JSON payload with todo details, validates the fields, and stores it in the repository. The system automatically sets the creation date and initializes the todo as undone. Returns the created todo with a generated ID.

#### Edit To-Do

* **Endpoint:** PUT /todos/{id}
* **Description:** Updates an existing todo item by its ID. Allows partial updates of text, priority, and due date fields. Only provided fields will be updated.
* **Implementation:** The endpoint looks up the todo by ID, updates only the provided fields while maintaining existing values for unspecified fields. Returns the updated todo or throws an exception if the ID doesn't exist.

#### Filter, Search & Sorting

* **Endpoint:** GET /todos
* **Query Params:** page (default: 0), size (default: 10), sortBy (in format field\_direction-field\_direction), status (filter), text (filter), priority (filter)
* **Description:** Retrieves a paginated list of todos with support for filtering, searching, and sorting. Results can be filtered by status, priority, and text content, and sorted by priority and due date.
* **Implementation:** The endpoint processes multiple query parameters to filter and sort todos. Sorting supports priority and due date fields, handling null values appropriately. Results are paginated and include both the filtered todos and total item count.

#### Mark as Done

* **Endpoint:** POST /todos/{id}/done
* **Description:** Marks a specific todo item as completed. When marked as done, the system automatically sets the done date to the current timestamp.
* **Implementation:** The endpoint updates the todo's done status to true and automatically sets the doneDate to the current timestamp. If the todo is already marked as done, it returns the unchanged todo. Returns the updated todo or throws an exception if the ID is not found.

#### Mark as Undone

* **Endpoint:** PUT /todos/{id}/undone
* **Description:** Marks a completed todo item as incomplete, removing its done status and done date.
* **Implementation:** The endpoint updates the todo's done status to false. If the todo is already marked as undone, it returns the unchanged todo. Returns the updated todo or throws an exception if the ID is not found.

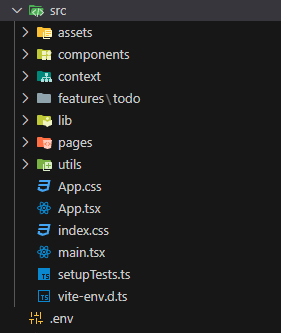
## FRONTEND

* Language: TypeScript
* Framework: React
* Styling: TailwindCSS + ShadCN
* Testing: Jest
* State Management: React Context
* Linting: ESLint + Prettier
* Requests: Axios

### Frontend Architecture Decisions

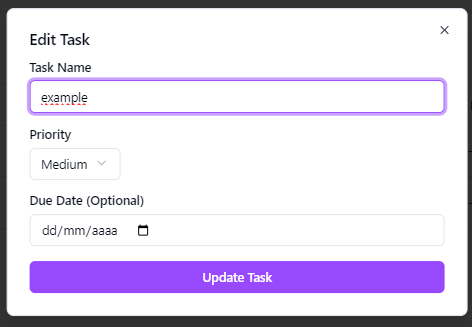
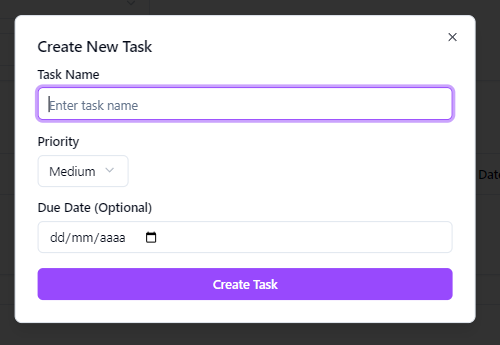
For the Frontend I went with this structure:

* **Features** — organize the services, components, tests and context for the different features in order to keep all that logic together.
* **Components** — global reusable UI components.
* **Pages** — here we have the app views and those uses the content in features.
* **Lib** — manage the adapter patterns applied to 3rd party libraries (in this case Axios).
* **Context** — global state shared between all views or components of our applications.
* **Utils** — general-purpose functions.

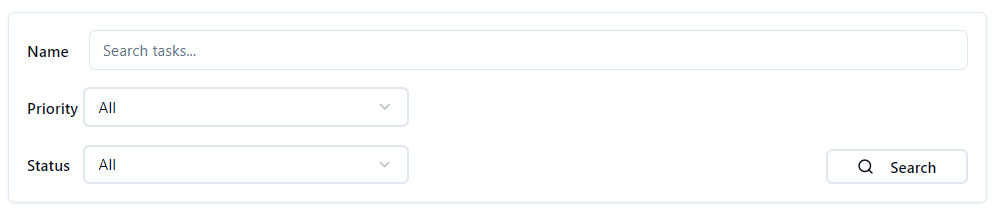
****

### Frontend Feature Implementations

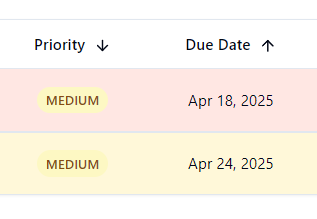
#### Create / Edit To-Do Modal

****This is a reusable modal component for creating and editing todo tasks. The modal implementation leverages Radix UI's Dialog component to create a flexible form interface that handles both task creation and editing, all of this is achieved by the use of the “mode” prop. It also integrates toast notifications in order to show to the user the result of its actions.

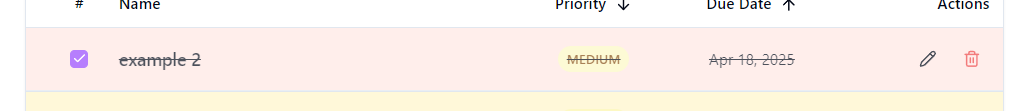
#### Search and Filtering

****Provides filtering capabilities for the todo list. You can search by the name of the tasks and filter using the Priority and Status with the provided options. This section of the application is also connected to the state of the todos (TodoContext) in order to filter the queries for the backend.

#### Sorting Columns

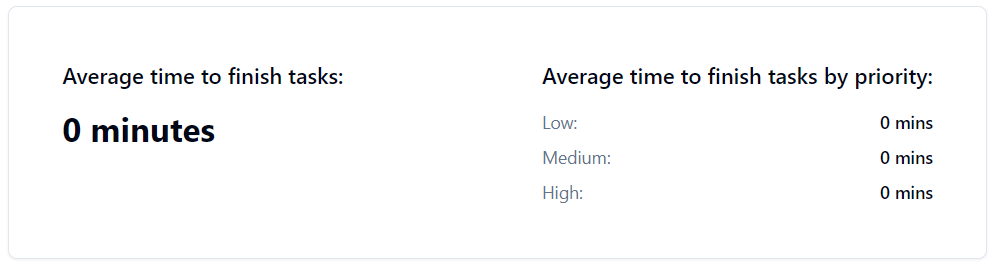
****This feature implements sortable columns for Priority and Due Date. It uses three states (none/ ascending/descending) with arrows to indicate these states visually. This section builds a sort string (e.g., "priority\_asc-duedate\_dsc") that's passed to the API.

#### Mark Done/Undone

**  
**

#### This feature was implemented inside the TodoItem through a checkbox component that allows us to mark a task as done or undone. It uses the TodoContext in order to change the Todo state, and the context state also uses the todoServices in order to send this change to the backend.

#### Metrics Dashboard

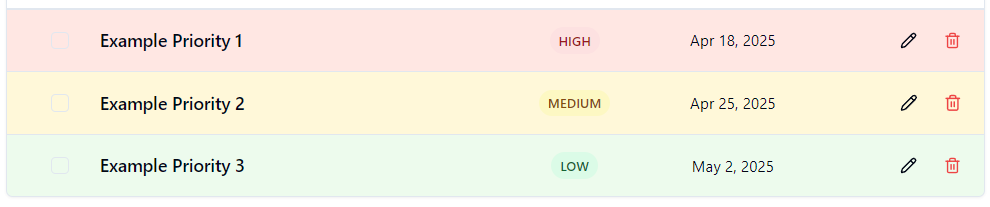
****This section of the app returns the performance metrics based on the creation date and the done date. It also has a section that shows us this performance by priority. The component receives data through the TodoContext and automatically updates when tasks are completed.

#### Pagination Controls



This component was created using the Pagination component from ShadCN. The pagination state is managed through the TodoContext and updates the API calls with the correct page parameters. It has a previous/next navigation and page number indicator.

#### Priority-based Colors

****

This Priority-based colors feature allows us to differentiate easily between less important tasks and more important tasks. It is implemented using Tailwind CSS classes and changing them depending on the priority of the class field, in the case of the badge, and depending on the DueDate, for the background color used.

ALTERNATIVE SOLUTIONS –

For state management, I considered using Redux since it was allowed and powerful, especially for large applications. However, for this to-do app, the state wasn't complex enough to justify the overhead, so I went with React Context, which kept things simpler and easier to manage.

On the backend, I thought about using packaging by layer, but I preferred packaging by feature because it helps keep related files together and makes the project easier to navigate. The architecture itself didn’t change, just the way it’s organized.

There were also simpler architectures I could’ve used, like a flat structure, but one of the project’s requirements was to make it easy to transition to a database in the future. Also, maintainability and best practices were part of the learning objectives, so I decided to go with a more structured layered design even if it seemed like more effort for a small app.