

FlowForge

Workflow Management System

Introduction:

The **FlowForge** application is a **full stack application** that helps users manage their tasks in a simple and organized way. Designed for users who want to **create, update, and track** their tasks, it allows them to move tasks through different **stages**, such as **Not Started, In Progress, and Completed**.

With **FlowForge**, users can easily create new tasks, set priorities, **manage subtasks**, and **track their progress**, all through an intuitive interface. The application also supports **changing the state of tasks**, enabling users to update them as they work through each stage of their workflow.

Users Features:

- Create New Tasks
- Browse Available Tasks
- Add and Manage Subtasks
- Change Task States
- Set Task Priorities
- Search and Filter Tasks
- Remove Tasks
- Receive Notifications

All data is stored in a local **PostgreSQL database**. All dependencies are organized through **interfaces**, ensuring that the project is flexible, maintainable, and easily extendable. This structure allows for seamless integration of new features, while ensuring smooth interaction between components.

Which Design Patterns We Used?

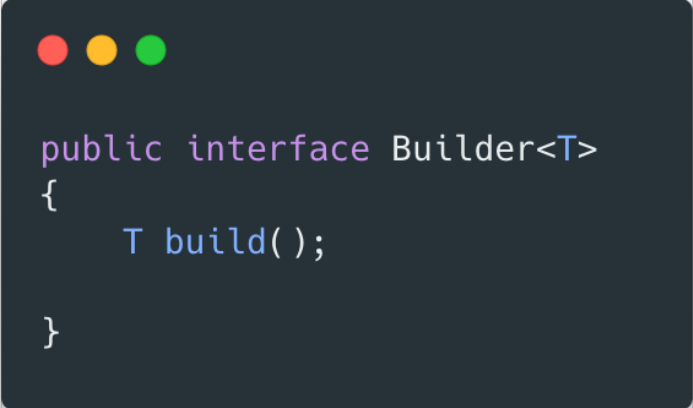
1)Builder

The **Builder Pattern** is used to simplify the creation of complex objects. Instead of using multiple constructors or setting properties manually, the builder allows us to **set** the properties **step by step** and then **build the object**.

Where it is used:

In this project, the Builder Pattern is used to **create Card and Task objects**

1)**Builder interface**: declares product construction steps that are common to all types of builders.



```
public interface Builder<T>
{
    T build();
}
```

2) **CardBuilder:Concrete class** for interface **Builder**

This builder helps create a Card object by setting its properties like title, description, status, and workspace.

```

public class CardBuilder implements Builder<Card> {

    private Card card;

    public CardBuilder() {
        this.card = new Card();
    }

    public static CardBuilder builder() {
        return new CardBuilder();
    }

    public CardBuilder title(String title) {
        card.setTitle(title);
        return this;
    }

    public CardBuilder description(String description)
{
    card.setDescription(description);
    return this;
}

    public CardBuilder status(CardStatus status) {
        card.setStatus(status);
        return this;
    }

    public CardBuilder workspace(Workspace workspace) {
        card.setWorkspace(workspace);
        return this;
    }

    @Override
    public Card build() {
        Card result = this.card;
        this.card = null;
        return result;
    }

}

```

3) **TaskBuilder**: same as **CardBuilder**, also implements the main interface for the builder and is used to **create a Task** object by setting properties like **title, completion status, card, and subtasks**.

```

public class TaskBuilder implements Builder<Task> {

    private Task task;

    public TaskBuilder() {
        task = new Task();
    }

    public static TaskBuilder builder() {
        return new TaskBuilder();
    }

    public TaskBuilder title(String title) {
        task.setTitle(title);
        return this;
    }

    public TaskBuilder completed(Boolean completed) {
        task.setCompleted(completed);
        return this;
    }

    public TaskBuilder lastUpdate(LocalDateTime lastUpdate)
{
    task.setLastUpdate(lastUpdate);
    return this;
}

    public TaskBuilder card(Card card) {
        task.setCard(card);
        return this;
    }
}

```

Problems Solved by Builder in the FlowForge Project

1. Simplified Object Creation

- To create a Task object without using the Builder Pattern, we might use a constructor with many parameters or set the properties one by one, like this:

```

Task task = new Task("Task 1", false, LocalDateTime.now(), card, parentTask, subTasks);
|

```

- This avoids the need for complex constructors and makes the code more readable.

2. Avoiding Constructor Overloading

- If we were to use constructors to set all combinations of properties for a **Task** object, we'd end up with multiple constructors, making the code difficult to maintain.

```

public Task(String title, Boolean completed, Card card) { ... }
public Task(String title, Boolean completed, Card card, Task parent) { ... }

```

- The Builder Pattern avoids this by providing a single builder with flexible **setter methods**, allowing developers to create tasks with only the necessary properties and in the correct order.

3. Flexibility for Future Changes

- When adding **new properties** to the **Task** or **Card** class, we would need to update constructors and other parts of the code, making the system harder to extend.

```

public TaskBuilder priority(String priority) {
    task.setPriority(priority);
    return this;
}

```

- If more fields are added to Task or Card, the Builder can be easily updated without breaking existing code, making the system more adaptable.

2) Facade:

The **Facade Pattern** in **FlowForge** simplifies complex interactions with **multiple services** by providing a single, **easy-to-use interface**. The **CardFacade** and **TaskFacade** classes offer simplified methods for creating, updating, and managing cards and tasks, **hiding the complexity** of services.

Where it is used:

1)**CardFacade**: Manages the creation, updating, and state transitions of **Card** objects, such as creating a default card, starting, completing, or reopening cards.

2)**TaskFacade**: Simplifies the process of creating and managing **Task** objects, including tasks linked to cards, toggling completion statuses, and handling parent-child relationships between tasks.

```

public class CardFacade {

    private final CardService cardService;
    private final WorkspaceService workspaceService;
    private final CardStateResolver cardStateResolver;

    public CardFacade(CardService cardService, WorkspaceService workspaceService, CardStateResolver
cardStateResolver) {
        this.cardService = cardService;
        this.workspaceService = workspaceService;
        this.cardStateResolver = cardStateResolver;
    }

    public CardDTO createDefaultCard(Long workspaceId) {
        Workspace workspace = workspaceService.getWorkspaceEntityById(workspaceId);

        Card card = CardBuilder.builder()
            .title("Untitled")
            .status(CardStatus.NOT_STARTED)
            .workspace(workspace)
            .build();

        return cardService.saveCard(card);
    }

    public CardDTO createCard(Long workspaceId, CardCreationDTO dto) {
        Workspace workspace = workspaceService.getWorkspaceEntityById(workspaceId);

        Card card = CardBuilder.builder()
            .title(dto.title())
            .description(dto.description())
            .status(CardStatus.NOT_STARTED)
            .workspace(workspace)
            .build();

        return cardService.saveCard(card);
    }
}

```

Problems Solved by Facade in the FlowForge Project

1. Simplifies Complex Interactions

- Without pattern: To create a new card, we would need to interact directly with multiple services, such as choosing a workspace, creating a card, and setting its state.

```

Workspace workspace = workspaceService.getWorkspaceEntityById(workspaceId);
Card card = new Card("Untitled", CardStatus.NOT_STARTED, workspace);
cardService.saveCard(card);

```

- The CardFacade provides a simplified interface:

```

cardFacade.createDefaultCard(workspaceId);

```

2. Improving Code Readability

The **Facade** hides the internal complexity and provides a simplified interface for the user. This makes the code more **readable** and **easier to understand** by reducing the number of interactions needed to do an operation.

3)State

The **State Pattern** is used in the **FlowForge** project to manage the state changes of **Card** objects. This pattern allows us to change the behavior of a **Card** based on its current state (Not Started, In Progress, Completed).

Where It's Used:

1)State interface

```
public interface CardState {  
  
    void start(Card card);  
    void complete(Card card);  
    void reopen(Card card);  
  
    CardStatus getStatus();  
  
}
```

The **CardState** interface defines the methods that all states must implement to manage the state transitions of a **Card**.

2)CardStateResolver

```
public class CardStateResolver {  
  
    private final Map<CardStatus, CardState> stateMap = new HashMap<>();  
  
    public CardStateResolver(List<CardState> states) {  
        for (CardState state : states) {  
            stateMap.put(state.getStatus(), state);  
        }  
    }  
  
    public CardState resolve(Card card) {  
        return stateMap.get(card.getStatus());  
    }  
  
}
```

Resolves and retrieves the correct state based on the **Card** current status. It maps a **CardStatus** to a corresponding **CardState**.

3)Concrete State Classes

```
public class InProgressState implements CardState {

    @Override
    public void start(Card card) {
        throw new IllegalStateException("Already started");
    }

    @Override
    public void complete(Card card) {
        card.setStatus(CardStatus.COMPLETED);
    }

    @Override
    public void reopen(Card card) {
        throw new IllegalStateException("Already in progress");
    }

    @Override
    public CardStatus getStatus() {
        return CardStatus.IN_PROGRESS;
    }

}
```

These classes, such as **NotStartedState**, **InProgressState**, and **CompletedState**, implement the state transition logic and define the behavior of each state.

Problems Solved by State in the FlowForge Project

1. Handling State Changes:

- Managing state transitions manually without encapsulating them in a state class would require checking the card status and writing conditional logic to handle transitions,

```
if (card.getStatus() == CardStatus.NOT_STARTED) {
    card.setStatus(CardStatus.IN_PROGRESS);
} else if (card.getStatus() == CardStatus.IN_PROGRESS) {
    card.setStatus(CardStatus.COMPLETED);
}
```

- The **State Pattern** allows state transitions to be handled by the specific state objects themselves, making the code cleaner and more maintainable

2. Avoiding Invalid State Transitions

- We have problems without pattern, like trying to complete a task that is not started

```
if (card.getStatus() == CardStatus.COMPLETED) {
    throw new IllegalStateException("Cannot reopen a completed task");
}
```

- Each state class handles invalid transitions internally and throws an exception when an illegal operation is attempted

3. Enhancing Flexibility for Future States:

- Without pattern adding new states would require modifying multiple parts of the code to handle the new state
- To add a new state you simply create a new state class without modifying the existing code. This makes the system more flexible and easier to extend in the future

```
public class OnHoldState implements CardState {
    @Override
    public void start(Card card) {
        card.setStatus(CardStatus.IN_PROGRESS);
    }

    @Override
    public void complete(Card card) {
        throw new IllegalStateException("Cannot complete a task on hold");
    }
}
```

The **State Pattern** in FlowForge makes Card state transitions easier by separating the logic into different classes. It ensures valid transitions, **improves readability**, and makes the system **easier to update with new states**. This keeps the system **organized** and **reduces errors**.

Applying SOLID Principles in the Architecture of the FlowForge Application

SOLID is a set of five design principles that help improve the maintainability, scalability, and readability of software. In the FlowForge project. Here's how each principle is applied:"

1)Single Responsibility Principle (SRP)

Where it's used:

In the FlowForge project, each class has one responsibility. For example, the **CardFacade** class is responsible for managing **card-related operations**, while the **TaskFacade** class handles **task-related operations**.

How it helps:

It ensures that each class has a single reason to change, making the system easier to understand and maintain

2)Open/Closed Principle (OCP)

Where it's used:

The State Pattern used for managing card states is an example of OCP in FlowForge. New states can be added without modifying the existing state classes.

How it helps:

It allows the system to be extended with new features without changing the existing code, reducing the risk of introducing bugs in already working parts of the system.

3)Liskov Substitution Principle (LSP)

Where it's used:

In FlowForge, subclasses like `CompletedState`, `InProgressState`, and `NotStartedState` all implement the same interface `CardState` and can be replaced without breaking the system.

How it helps:

It ensures that objects of a subclass can replace objects of the parent class without causing issues, making the system more reliable and adaptable.

4) Interface Segregation Principle (ISP)

Where it's used:

The CardState interface is divided into methods specific to the actions for each state, ensuring that each class only implements the methods it needs.

How it helps:

This prevents the forced implementation of methods that classes do not use, making the code more modular and understandable.

5) Dependency Inversion Principle (DIP)

Where it's used:

Classes like CardFacade and TaskFacade depend on abstractions (such as builders, resolvers, and interfaces) rather than directly on specific implementations in the builder and state packages.

How it helps:

By making high-level modules (like the facades) depend on interfaces, it allows for easier testing and makes it simpler to change or replace the implementation without affecting the rest of the system. This approach improves flexibility and maintainability.

By following **SOLID**, the FlowForge project is built in a way that supports future growth and easy maintenance, keeping the code clean, organized, and flexible.

How to use an application?

- 1) Login to your account or create your own
- 2) Create your workspace, card and task
- 3) Enjoy!

Technical Part

Backend:

- Java Core (Generics, Design Patterns)
- Spring Framework (Backend Development)
- JWT Authorization (Token-Based Authentication)

Frontend:

- React (Reactive Frontend Development)
- Typescript (Static Typing)

Database:

- PostgreSQL (Primary Database)

Other Technologies:

- Postman (REST API Testing)
- Lombok (Annotation Processing)
- Swagger-UI (API Documentation & Endpoint Visualization)
- FontAwesome (Icons & Fonts, External Library)
- UI-avatars (Avatar Generation, External API)
- Hibernate (ORM Framework)

Thank you for your attention!