SOFTWARE REQUIREMENT SPECIFICATION

Agile Team Sprint Tracker with GitHub Integration

`Team Members:

- **Saim Ali Khan**
- **Arsalan Nawaz**
- **Khuzaima Asif**

Subject:

Software Construction And Development

Submit to

Dr. Nabeel Awan / Miss Zainab

Dated : 27 April,2025

Purpose:

The Agile Team Sprint Tracker is built to help software development teams plan, organize, and track their sprints. It connects directly to GitHub, allowing users to sync issues and pull requests automatically. This document explains the system's requirements, both functional and non-functional, and defines how the system should behave.

Links to GitHub Repository:

https://github.com/arslan12941/Agile-Team-Sprint-Tracker.git

Design Patterns Used:

1. MVC (Model-View-Controller) Pattern

The application clearly implements the MVC architecture pattern, separating data (Model), user interface (View), and application logic (Controller).

How MVC is Implemented:

Models (Entity classes):

- User.java - Represents user data
- Sprint.java - Represents sprint data
- Task.java - Represents task data

Controllers:

- AuthController.java - Handles authentication requests
- SprintController.java - Manages sprint operations
- TaskController.java - Manages task operations

Views:

- The frontend side that consumes the REST API (React components implied by the package.json)

Implementation Details:

- Controllers use @RestController annotations to handle HTTP requests
- Models are defined as @Entity classes for database persistence
- Controllers inject and delegate to services for business logic

2. Repository Pattern

The Repository pattern is used to abstract the data access logic.

How Repository Pattern is Implemented:

Repository Interfaces:

- UserRepository.java
- SprintRepository.java
- TaskRepository.java

Each repository extends JpaRepository, providing data access methods without exposing the underlying database operations.

3. DTO (Data Transfer Object) Pattern

DTOs are used to transfer data between processes, reducing the number of method calls and decoupling the domain model from the presentation layer.

How DTO Pattern is Implemented:

DTO Classes:

- AuthResponseDto.java - For authentication responses
- SprintCreateDto.java - For sprint creation requests
- SprintResponseDto.java - For sprint responses
- TaskCreateDto.java - For task creation requests
- TaskResponseDto.java - For task responses

4. Service Layer Pattern

The Service Layer pattern is used to encapsulate business logic and coordinate application activities.

How Service Layer Pattern is Implemented:

Service Classes:

- UserService.java - Handles user-related operations
- SprintService.java - Handles sprint-related operations
- TaskService.java - Handles task-related operations

Each service is responsible for its domain's business logic, interacting with repositories for data access.

5. Factory Method Pattern

A variation of the Factory Method pattern is used in the OAuth2 authentication flow.

How Factory Method Pattern is Implemented:

The `CustomOAuth2UserService` class serves as a factory that creates and configures OAuth2User objects:

java

```
@Override
public OAuth2User loadUser(OAuth2UserRequest userRequest) {
    OAuth2User oAuth2User = super.loadUser(userRequest);
    Map<String, Object> attributes = oAuth2User.getAttributes();



    return new DefaultOAuth2User(
            Set.of(() -> "ROLE_" + user.getRole().name()),
            attributes,
            "login"
    );
}
```

Class Diagram of Design Patterns

Class Diagram of Design Patterns

Diagram

Detailed Explanation of Pattern Implementations

1. MVC Pattern Implementation

The MVC pattern in your Spring Boot application clearly separates:

- **Models (Entity classes)**: Define data structure and relationships
- **Controllers**: Handle HTTP requests and responses
- **Services**: Contain business logic (acting as part of the Controller layer in Spring's interpretation of MVC)

The benefit of this approach is clean separation of concerns, where:

- Controllers focus on HTTP request handling
- Services focus on business logic
- Repositories handle data access
- Entities represent the data model

Example controller method from SprintController.java:

```java
@PreAuthorize("hasAuthority('ADMIN')")
@PostMapping
public ResponseEntity<Sprint> createSprint(@RequestBody SprintCreateDto dto) {
  return ResponseEntity.ok(sprintService.createSprint(dto));
}
```

2. Repository Pattern Implementation

Your application implements the Repository pattern through Spring Data JPA repositories:

```java
public interface SprintRepository extends JpaRepository<Sprint, Long> {
}
```

By extending JpaRepository, you gain access to standard CRUD operations plus the ability to define custom queries. This pattern:

- Abstracts data access logic
- Removes duplicate query code
- Makes testing easier through mock repositories
- Allows switching the underlying data storage without changing application code

3. DTO Pattern Implementation

Your application uses DTOs to:

- Transfer data between the client and server
- Decouple the internal data model from API representations
- Control what data is exposed to clients

For example, SprintCreateDto contains only the fields needed to create a Sprint:

```java
@Data
```

```
public class SprintCreateDto {
    private String name;
    private String goal;
    private LocalDate startDate;
    private LocalDate endDate;
}
```

4. Service Layer Pattern Implementation

Services encapsulate business logic and act as an interface between controllers and repositories:

```
java
@Service
public class SprintService {
    private final SprintRepository sprintRepository;



    public Sprint createSprint(SprintCreateDto dto) {
        Sprint sprint = new Sprint();
        sprint.setName(dto.getName());
        sprint.setGoal(dto.getGoal());
        sprint.setStartDate(dto.getStartDate());
        sprint.setEndDate(dto.getEndDate());
        sprint.setStatus(SprintStatus.PLANNED);
        return sprintRepository.save(sprint);
    }

    // Other methods...
}
```

This pattern:

- Centralizes business logic
- Makes controllers thinner
- Improves testability
- Enables transaction management at the appropriate level

5. Factory Method Pattern Implementation

The Factory Method pattern is implemented in CustomOAuth2UserService, which creates customized OAuth2User objects:

java

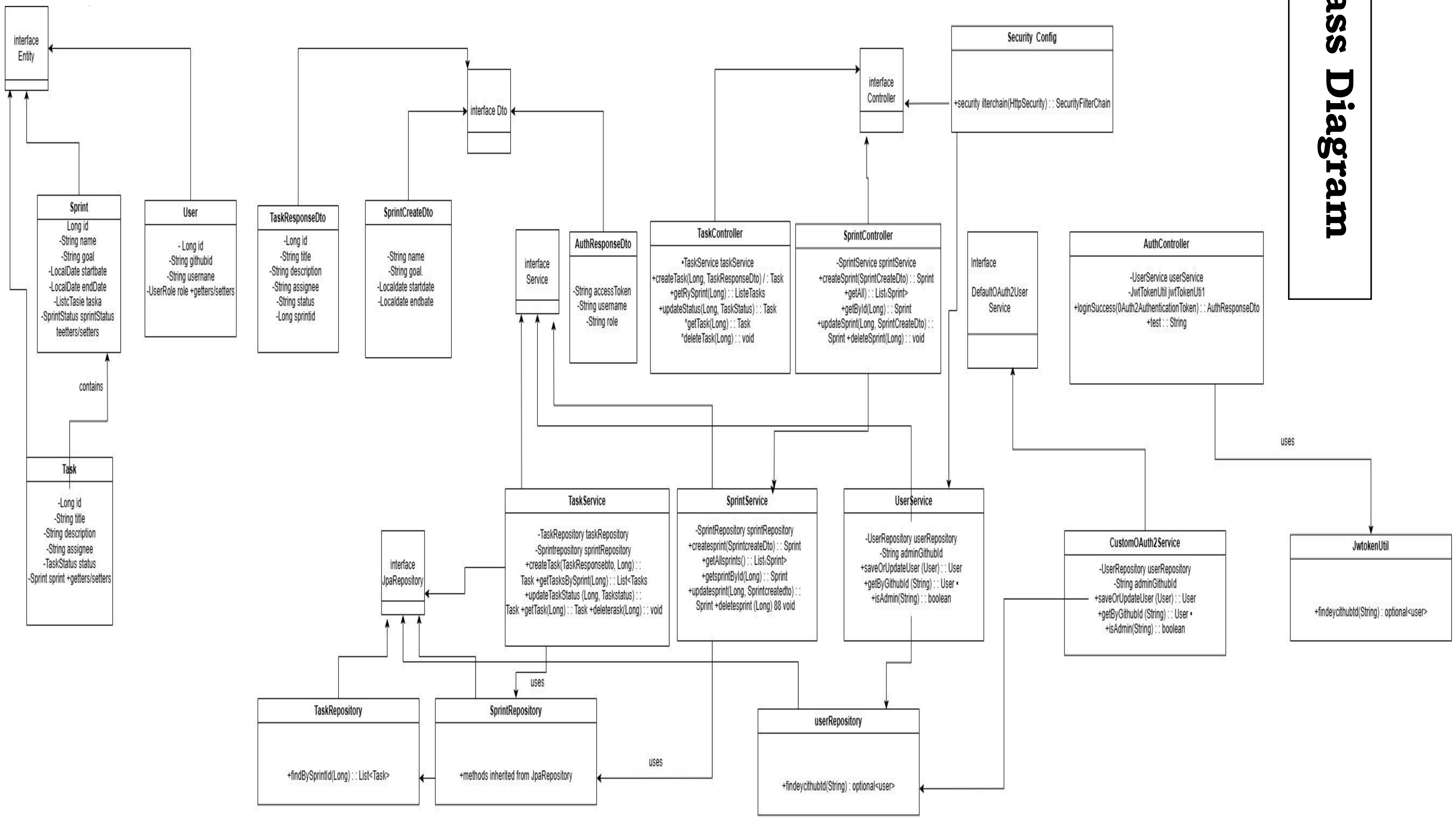
```
@Override
public OAuth2User loadUser(OAuth2UserRequest userRequest) {
  // Process authentication and create user...

  // Factory method creating a new OAuth2User
  return new DefaultOAuth2User(
      Set.of(() -> "ROLE_" + user.getRole().name()),
      attributes,
      "login"
  );
}
```

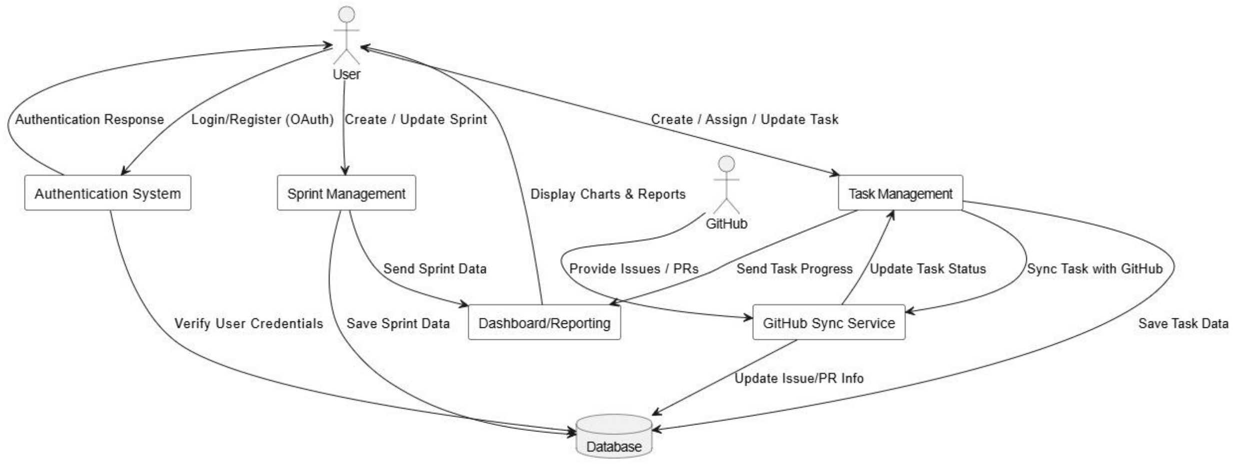
This pattern:

- Creates objects without exposing creation logic
- Allows customization of created objects
- Integrates with Spring Security's authentication flow

Class Diagram


interface Entity
Sprint
Long id
-String name
-String goal
-LocalDate startbate
-LocalDate endDate
-ListcTasie taska
-SprintStatus sprintStatus
teetters/setters
User
- Long id
-String githubid
-String usernane
-UserRole role +getters/setters
TaskResponseDto
-Long id
-String title
-String description
-String assignee
-String status
-Long sprintid
SprintCreateDto
-String name
-String goal.
-Localdate startdate
-Localdate endbate
interface Dto
AuthResponseDto
-String accessToken
-String username
-String role
interface Service
TaskController
•TaskService taskService
+createTask(Long, TaskResponseDto) / : Task
+getRySprint(Long) : : ListeTasks
+updateStatus(Long, TaskStatus) : : Task
*getTask(Long) : : Task
*deleteTask(Long) : : void
SprintController
-SprintService sprintService
+createSprint(SprintCreateDto) : : Sprint
+getAll() : : List‹Sprint›
+getById(Long) : : Sprint
+updateSprint(Long, SprintCreateDto) : :
Sprint +deleteSprint(Long) : : void
interface Controller
Security Config
+security ilterchain(HttpSecurity) : : SecurityFilterChain
Interface
DefaultOAuth2User
Service
AuthController
-UserService userService
-JwtTokenUtil jwtTokenUti1
+loginSuccess(0Auth2AuthenticationToken) : : AuthResponseDto
+test : : String
contains
Task
-Long id
-String title
-String description
-String assignee
-TaskStatus status
-Sprint sprint +getters/setters
uses
interface JpaRepository
TaskService
-TaskRepository taskRepository
-Sprintrepository sprintRepository
+createTask(TaskResponsebto, Long) : :
Task +getTasksBySprint(Long) : : List<Tasks
+updateTaskStatus (Long, Taskstatus) : :
Task +getTask(Long) : : Task +deleterask(Long) : : void
SprintService
-SprintRepository sprintRepository
+createsprint(Sprintcreatedto) : : Sprint
+getAllsprints() : : List‹Sprint›
+getsprintById(Long) : : Sprint
+updatesprint(Long, Sprintcreatedto) : :
Sprint +deletesprint (Long) 88 void
UserService
-UserRepository userRepository
-String adminGithubld
+saveOrUpdateUser (User) : : User
+getByGithubld (String) : : User •
+isAdmin(String) : : boolean
CustomOAuth2Service
-UserRepository userRepository
-String adminGithubld
+saveOrUpdateUser (User) : : User
+getByGithubld (String) : : User •
+isAdmin(String) : : boolean
JwtokenUtil
+findeycithubtd(String) : optional<user>
uses
TaskRepository
+findBySprintId(Long) : : List<Task>
SprintRepository
+methods inherited from JpaRepository
uses
userRepository
+findeycithubtd(String) : optional<user>

Data Flow Diagram


User
Authentication Response
Login/Register (OAuth)
Create / Update Sprint
Create / Assign / Update Task
Authentication System
Sprint Management
Display Charts & Reports
GitHub
Task Management
Send Sprint Data
Provide Issues / PRs
Send Task Progress
Update Task Status
Sync Task with GitHub
Verify User Credentials
Save Sprint Data
Dashboard/Reporting
GitHub Sync Service
Save Task Data
Update Issue/PR Info
Database