Sprint 2: REST APIs and Frontend Development

Lecture 4 Notes

School of Computing Communication and Media Studies

Masoud Hamad

CS2113 – Software Development Project
Academic Year 2025

Contents

1 Sprint Retrospective

The **Retrospective** is a Scrum event where teams discuss process issues and develop solutions. It occurs after the Sprint Review, concluding the current Sprint.

1.1 Retrospective Principles

- All team members discuss process challenges openly
- Every voice must be heard
- Solutions should be concrete actions implementable during the upcoming Sprint
- Previously identified issues shouldn't resurface in the next Retrospective

1.2 Mad, Sad, Glad Technique

Team members categorize their Sprint experiences:

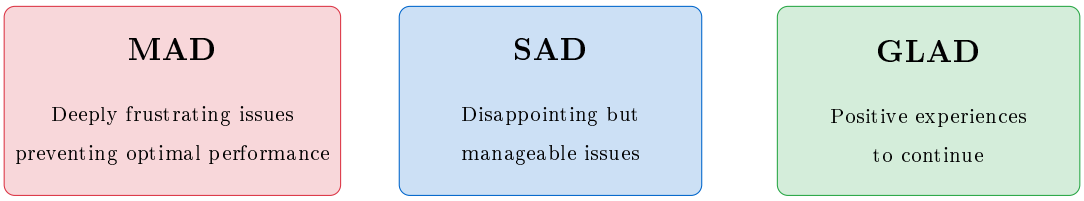

Figure 1: Mad, Sad, Glad Retrospective Categories

Examples:

- **Mad:** "Spring Boot application won't start on my computer"
- **Sad:** "Daily Scrum meetings take too long"
- **Glad:** "Team communication was transparent"

1.3 Retrospective Process

1. Team members independently write cards for each category (no discussion yet)
2. Review all cards category by category
3. Identify highest-priority issues from Mad and Sad categories
4. Develop at least one concrete action per issue
5. Document actions on the retrospective board

2 REST APIs

2.1 Traditional vs REST Approach

Traditional Web Application:

1. User opens page in browser
2. Browser requests resource from server
3. Controller processes request
4. Controller retrieves data from database
5. HTML page is generated
6. HTML response sent to browser

REST API Approach:

- Instead of HTML, serialize Java objects to **JSON format**
- Easier consumption by different clients (web, mobile, etc.)
- Clear separation between frontend and backend

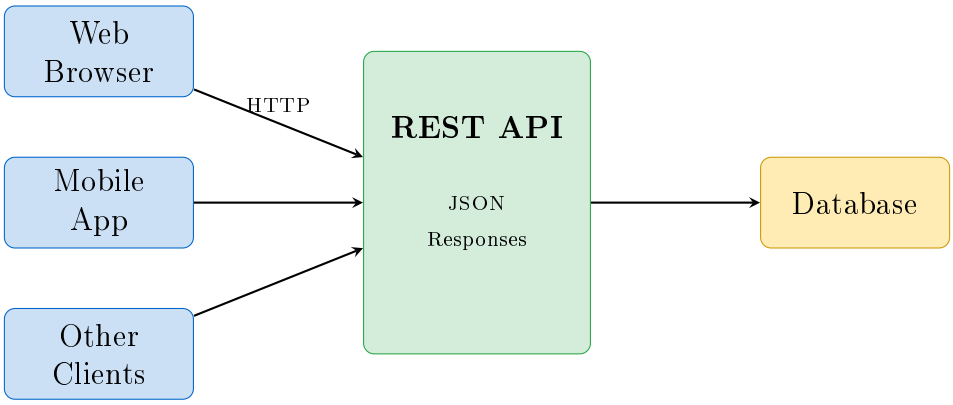

Figure 2: REST API Architecture

2.2 RESTful API Characteristics

- **Stateless:** Each request contains all information needed
- **Client-Server Separation:** Frontend and backend are independent
- **Resource-Based Paths:** URLs represent resources
- **HTTP Method Semantics:** Methods indicate actions

3 HTTP Methods

3.1 RESTful Endpoints Convention

Method	Action	Description
GET	Read	Retrieve data without modification
POST	Create	Create new database entries
PUT	Update	Update existing entries
DELETE	Remove	Delete database entries

Table 1: HTTP Methods and Their Actions

Method	Path	Description
GET	/api/quizzes	List all quizzes
GET	/api/quizzes/{id}	Get quiz by ID
POST	/api/quizzes	Create new quiz
PUT	/api/quizzes/{id}	Update quiz
DELETE	/api/quizzes/{id}	Delete quiz
GET	/api/quizzes/{id}/questions	Get quiz questions

Table 2: RESTful Endpoint Examples

Key Insight

Resource collections use **plural nouns** (e.g., "quizzes" not "quiz"). Path variables like `{id}` identify specific resources.

4 REST Controller Implementation

4.1 Basic REST Controller

```
@RestController
@RequestMapping("/api")
@CrossOrigin(origins = "*")
public class QuizRestController {

    @Autowired
    private QuizRepository quizRepository;

    @GetMapping("/quizzes")
    public List<Quiz> getAllQuizzes() {
        return quizRepository.findAll();
    }

    @GetMapping("/quizzes/{id}")
    public Quiz getQuizById(@PathVariable Long id) {
        return quizRepository.findById(id)
            .orElseThrow(() -> new ResponseStatusException(
                HttpStatus.NOT_FOUND,
                "Quiz with id " + id + " does not exist"
```

```
                ));
        }
}
```

4.2 POST Endpoint with Validation

```
@PostMapping("/quizzes")
public Quiz createQuiz(
        @Valid @RequestBody CreateQuizDto quiz,
        BindingResult bindingResult) {

    if (bindingResult.hasErrors()) {
        throw new ResponseStatusException(
            HttpStatus.BAD_REQUEST,
            bindingResult.getAllErrors()
                .get(0).getDefaultMessage()
        );
    }

    Quiz newQuiz = new Quiz(
        quiz.getName(),
        quiz.getDescription()
    );
    return quizRepository.save(newQuiz);
}
```

5 DTO Pattern (Data Transfer Objects)

DTO (Data Transfer Object) is a pattern using dedicated classes to control request/response body format. DTOs prevent undesired attribute updates and hide sensitive information.

5.1 Request DTO Example

```
public class CreateQuizDto {
    @NotBlank(message = "Name is required")
    private String name;

    @Size(max = 500, message = "Description too long")
    private String description;

    private String courseCode;

```

```
    // Constructors, getters, setters
}
```

5.2 Response DTO Example

```
public class QuizResultDto {
    private Long questionId;
    private String questionText;
    private String questionDifficulty;
    private int totalAnswers;
    private int correctAnswers;
    private int wrongAnswers;

    // Constructors, getters, setters
}
```

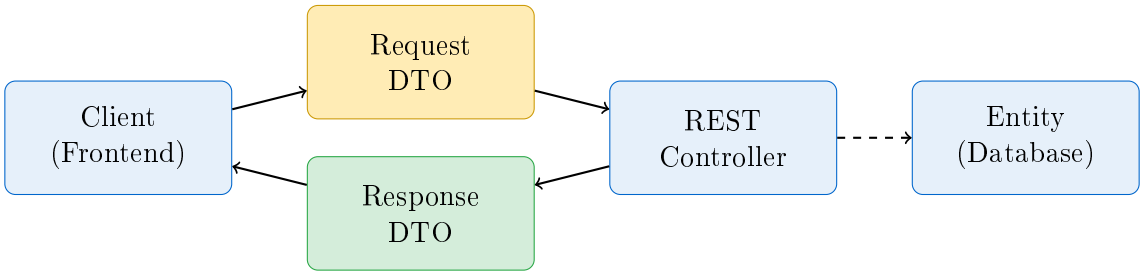

Figure 3: DTO Pattern in REST APIs

6 HTTP Status Codes

6.1 Status Code Ranges

- **200–299:** Successful responses
- **400–499:** Client errors
- **500–599:** Server errors

6.2 Common Status Codes

6.3 ResponseEntity for Custom Responses

```
@PostMapping("/quizzes")
public ResponseEntity<?> createQuiz(
        @Valid @RequestBody CreateQuizDto quiz,
        BindingResult bindingResult) {
```

Code	Meaning
200 OK	Request successful
201 Created	Resource created successfully
400 Bad Request	Invalid request (e.g., validation error)
401 Unauthorized	User not authenticated
403 Forbidden	User lacks authorization
404 Not Found	Resource not found
500 Internal Server Error	Server-side error

Table 3: Common HTTP Status Codes

```
5
6       if (bindingResult.hasErrors()) {
7           return ResponseEntity
8                   .status(HttpStatus.BAD_REQUEST)
9                   .body(bindingResult.getAllErrors());
10      }
11
12      Quiz newQuiz = new Quiz(quiz.getName());
13      quizRepository.save(newQuiz);
14
15      return ResponseEntity
16          .status(HttpStatus.CREATED)
17          .body(newQuiz);
18  }
```

7 Swagger API Documentation

Swagger automatically generates API documentation from controller classes using the OpenAPI standard format.

7.1 Adding Spring Doc Dependency

```
1  <dependency>
2      <groupId>org.springdoc</groupId>
3      <artifactId>springdoc-openapi-starter-webmvc-ui</
           artifactId>
4      <version>2.3.0</version>
5  </dependency>
```

7.2 Access URLs

- JSON format: `http://localhost:8080/v3/api-docs`

- User-friendly UI: `http://localhost:8080/swagger-ui/index.html`

7.3 Documentation Annotations

```java
@Tag(name = "Quizzes",
     description = "Operations for quiz management")
@RestController
@RequestMapping("/api/quizzes")
public class QuizRestController {

    @Operation(
        summary = "Get quiz by ID",
        description = "Returns the quiz with the provided ID
            "
    )
    @ApiResponses(value = {
        @ApiResponse(responseCode = "200",
            description = "Quiz retrieved successfully"),
        @ApiResponse(responseCode = "404",
            description = "Quiz not found")
    })
    @GetMapping("/{id}")
    public Quiz getQuizById(@PathVariable Long id) {
        // ...
    }
}
```

8 Frontend-Backend Communication

8.1 Fetch API - GET Request

```javascript
fetch("http://localhost:8080/api/quizzes")
    .then(response => response.json())
    .then(quizzes => {
        console.log(quizzes);
    });
```

8.2 Fetch API - POST Request

```javascript
fetch("http://localhost:8080/api/quizzes", {
    method: "POST",
    headers: {
        "Accept": "application/json",
        "Content-Type": "application/json"
    },
```

```
    body: JSON.stringify({
        name: "JavaScript Quiz",
        description: "Test your JS knowledge"
    })
})
.then(response => response.json())
.then(newQuiz => {
    console.log(newQuiz);
});
```

8.3 Service Function Pattern

```
const BACKEND_URL = "http://localhost:8080";

export function getAllQuizzes() {
    return fetch(`${BACKEND_URL}/api/quizzes`)
        .then(response => response.json());
}

export function createQuiz(quiz) {
    return fetch(`${BACKEND_URL}/api/quizzes`, {
        method: "POST",
        headers: {
            "Accept": "application/json",
            "Content-Type": "application/json"
        },
        body: JSON.stringify(quiz)
    })
    .then(response => {
        if (!response.ok) {
            throw new Error("Failed to create quiz");
        }
        return response.json();
    });
}
```

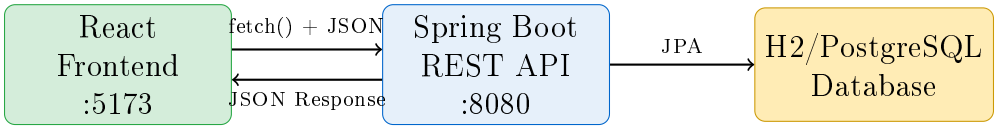

Figure 4: Full Stack Application Communication

9 Data Model Documentation

Document your data model in README.md using Mermaid syntax:

````
## Data Model

```mermaid
erDiagram
    QUIZ ||--o{ QUESTION : contains
    QUESTION ||--o{ ANSWER_OPTION : has
    QUIZ }o--|| CATEGORY : belongs_to

    QUIZ {
        Long id
        String name
        String description
        String courseCode
        Boolean published
        LocalDateTime createdAt
    }

    QUESTION {
        Long id
        String text
        String difficulty
    }
```
````

10 Exercises

Sprint 2 Deadline

All work must be pushed to GitHub repository before the Sprint deadline.

10.1 Exercise 1: Retrospective

Conduct a Mad-Sad-Glad retrospective using Flinga. Document the board link in README.md.

10.2 Exercise 2: Select New Scrum Master

Choose a different team member as Scrum Master for Sprint 2.

10.3 Exercise 3: Close Sprint 1 Issues

Close completed Sprint 1 issues and create Sprint 2 milestone.

10.4 Exercise 4: Create Labels

Create "frontend" and "backend" labels for categorizing issues.

10.5 Exercise 5: User Story Issues

Create issues for Sprint 2 user stories with appropriate labels and milestone.

10.6 Exercises 6–9: Task Planning

Break down each Sprint 2 user story into technical tasks.

10.7 Exercise 10: Data Model Documentation

Create entity relationship diagram using Mermaid syntax in README.md.

10.8 Exercises 11–17: REST Endpoint Implementation

Implement REST endpoints for:

- Get quiz by ID
- Get quiz questions
- Create answer submission
- Get quiz results
- Get all categories
- Get category by ID
- Get category quizzes

10.9 Exercise 18: Swagger Documentation

- Add Spring Doc dependency
- Add @Operation annotations to all endpoints
- Add @ApiResponses for success/error responses
- Group endpoints using @Tag
- Add Swagger link to README.md

10.10 Exercises 19–24: Frontend Development

Plan and implement frontend user stories for student dashboard.

10.11 Exercise 25: Deploy Frontend

Deploy frontend application to production (e.g., Render).

10.12 Exercise 26: Developer Guide

Update README.md with frontend startup instructions and technology stack.

10.13 Exercise 27: Create Release

Create GitHub release "Sprint 2" with feature description.

10.14 Exercise 28: Sprint Review Preparation

Prepare working demo of both teacher and student dashboards.
