Spring Data JPA - day 2

Datamatiker @ EK - 3rd semester

GitHub

Outline

- Recap
- DTO's with Java Records
- Value objects with `@Embeddable` and `@Embedded`
- Transactions with `@Transactional`
- Relationship - `@OneToOne` and `@ManyToMany`
- Cascade Types
- Fetch Types
- Repository testing with `@DataJpaTest`

Recap

You should be familiar with:

- JPA Entities and Annotations
- Spring Data JPA Repositories
- OneToMany and ManyToOne Relationships
- Bidirectional vs Unidirectional Relationships

Quiz

- What annotation is used to define a JPA entity?
- How do you define a primary key in a JPA entity?
- What is the difference between `@OneToMany` and `@ManyToOne` relationships?
- What is the purpose of Spring Data JPA repositories?
- When is `mappedBy` used in JPA relationships?

DTO's

> Data Transfer Objects (DTOs) are used to transfer data between different layers of an application.

Benefits of using DTO's:

- **Encapsulation**: They encapsulate the data and expose only what is necessary.
- **Decoupling**: They decouple the internal data model from the external representation.
- **Performance**: They can help optimize performance by transferring only the required data.
- **Security**: They can prevent overexposure of sensitive data.
- **Flexibility**: They allow for different representations of the same data for different use cases.

DTO's with Java Records

> Java Records provide a concise way to create **immutable** data carriers.

Controller accepts a DTO as a request body:

```
public record CreateTodoRequest(String title, String description) {}
```

And returns a DTO as a response:

```
public record TodoResponse(Long id, String title, String description) {}
```

Notice the different representation of the same data for request and response.

Example (Controller method using DTO's)

Controller method using DTO's:

```
@PostMapping
public ResponseEntity<TodoResponse> createTodo(
    @RequestBody CreateTodoRequest request
    ) {
    return ResponseEntity.ok(todoService.createTodo(request));
}
```

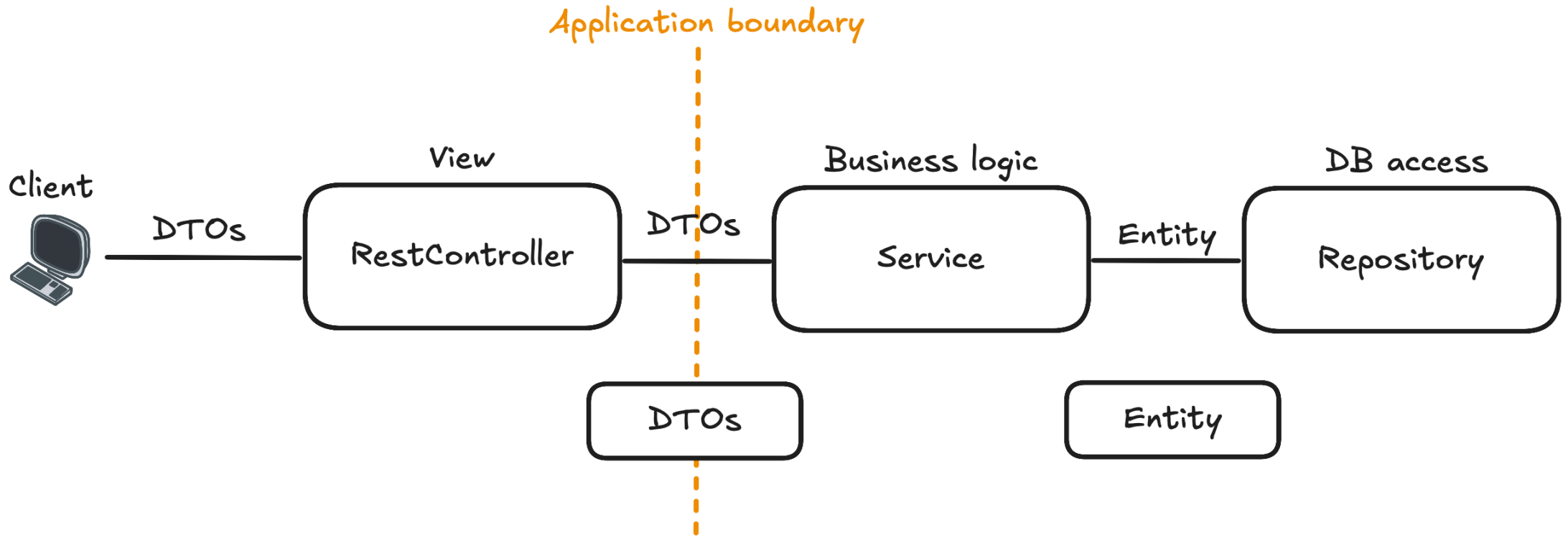
Example (Service method using DTO's)

The service class then needs to map between DTO's and entities:

```
public TodoResponse createTodo(CreateTodoRequest request) {
    Todo todo = new Todo();
    todo.setTitle(request.title());
    todo.setDescription(request.description());
    Todo savedTodo = todoRepository.save(todo);
    return new TodoResponse(
        savedTodo.getId(),
        savedTodo.getTitle(),
        savedTodo.getDescription()
    );
}
```

We can also create utility methods for mapping between entities and DTO's to keep the service methods clean.

Mental model of DTOs



Value Objects with `@Embeddable` and `@Embedded`

> Value objects represent a concept or attribute of an entity and do not have their own identity.

They are typically used to encapsulate related attributes.

Example: An `Address` value object that can be embedded in a `User` entity.

```
@Embeddable
public class Address {
    private String street;
    private String city;
    private String zipCode;
    // Getters and Setters
}
```

Embedding Value Objects

The `Address` value object can be embedded in the `User` entity:

```
@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    @Embedded
    private Address address;
    // Getters and Setters
}
```

The object is stored in the same table as the owning entity.

Transactions

> A transaction is a sequence of operations performed as a single logical unit of work.

Suppose we want to do several database operations that must all succeed or fail together. If we want to ensure that either all operations are completed successfully or none are applied, we can use transactions.

Transactions are a database concept that ensures data integrity and consistency.

Managing Transactions with `@Transactional`

In Spring, we can manage transactions using the `@Transactional` annotation.

```
@Service
public class UserService {
    @Transactional
    public void createUserAndProfile(User user, Profile profile) {
        userRepository.save(user);
        profileRepository.save(profile);
    }
}
```

If any operation within the method fails, all changes are rolled back.

Relationship - `@OneToOne`

> A `@OneToOne` relationship means that one entity is associated with exactly one instance of another entity.

```
@Entity
public class Profile {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String bio;
    @OneToOne
    private User user;
    // Getters and Setters
}
```

This creates a one-to-one relationship between `Profile` and `User`.

Bidirectional `@OneToOne`

In a bidirectional `@OneToOne` relationship, both entities reference each other:

```
@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    @OneToOne(mappedBy = "user")
    private Profile profile;
    // Getters and Setters
}
```

Remeber to use `mappedBy` to indicate the owning side of the relationship.

Relationship - `@ManyToMany`

> A `@ManyToMany` relationship means that multiple instances of one entity can be associated with multiple instances of another entity.

```
@Entity
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    @ManyToMany
    private List<Course> courses = new ArrayList<>();
    // Getters and Setters
}
```

This creates a many-to-many relationship between `Student` and `Course`.

Bidirectional `@ManyToMany`

In a bidirectional `@ManyToMany` relationship, both entities reference each other:

```
@Entity
public class Course {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String title;
    @ManyToMany(mappedBy = "courses")
    private List<Student> students = new ArrayList<>();
    // Getters and Setters
}
```

Again, use `mappedBy` to indicate the owning side of the relationship.

Relationship - `@ManyToMany` Join Table

By default, JPA creates a join table to manage the many-to-many relationship:

We can customize the join table using `@JoinTable`:

```
@ManyToMany
@JoinTable(
    name = "student_course",
    joinColumns = @JoinColumn(name = "student_id"),
    inverseJoinColumns = @JoinColumn(name = "course_id")
)
private List<Course> courses = new ArrayList<>();
```

This creates a join table named `student_course` with foreign keys to both `Student` and `Course`.

> Notice: That Student entity is responsible for persisting the relationship.

Cascading

> **Cascading** in JPA allows operations performed on a parent entity to be **automatically propagated** to its related child entities.

For example, when we create a `User` with an associated `Profile`, we want the `Profile` to be automatically persisted when we save the `User`, so we only need to call `userRepository.save(user)`.

Cascade Types

JPA provides several cascade types to specify which operations should be cascaded:

- `ALL` : Cascades all operations (persist, merge, remove, refresh, detach).
- `PERSIST` : When the parent is saved, the child is also saved.
- `MERGE` : When the parent is updated, the child is also updated.
- `REMOVE` : When the parent is deleted, the child is also deleted.

We don't often use the last two cascade types:

- `REFRESH` : Cascades the refresh operation.
- `DETACH` : Cascades the detach operation.

Example of CascadeType.PERSIST

```
@Entity
public class User {
    // ... other fields
    @OneToOne(cascade = CascadeType.PERSIST)
    private Profile profile;
    // Getters and Setters
}
```

To persist a `User` along with its `Profile`:

```
User user = new User();
Profile profile = new Profile();
user.setProfile(profile);
userRepository.save(user);
```

NO need to explicitly save the `Profile`. It will be automatically persisted.

Fetch Types

> Fetch types determine when related entities are loaded from the database.

This is important for performance optimization. We don't always want to load all related entities immediately, especially in large relationships.

JPA provides two fetch types:

- `EAGER` : Related entities are loaded immediately with the parent entity.
- `LAZY` : Related entities are loaded on-demand when accessed.

Default Fetch Types

Default is `EAGER` for:

- `ManyToOne` and `OneToOne`

Default is `LAZY` for:

- `OneToMany`, `ManyToMany`

You can override the default fetch type using the `fetch` attribute:

```
@OneToMany(fetch = FetchType.EAGER)
private List<Order> orders;
```

Repository Testing with `@DataJpaTest`

> `@DataJpaTest` is a specialized test annotation in Spring Boot for testing JPA repositories.

```
// Autoconfigures in-memory database
@DataJpaTest
class UserRepositoryTest {

    @Autowired
    private UserRepository userRepository;

    @Test
    void someTest() {
        // Test repository methods here
    }
}
```