COURSEWARE

Data Transfer Objects

Contents

Overview

A **Data Transfer Object (DTO)** is used to set the structure of the data provided by an API.

The importance of DTOs

There are two main benefits to using DTOs in a Spring API:

1. They allow us to combine data from multiple tables together into a single object.
   This allows us to send quite complex data over a single request, rather then having to chain a bunch of requests together.

2. They prevent the entities from being exposed directly to the user.
   Consider this example:

```
@Entity
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String username;

    private String password;

    // Getters and Setters
}
```

```
@RestController
public class UserController {

    private UserService service;

    // constructor

    @PostMapping("/register")
    public User registerUser(@RequestBody User newUser) {
        this.service.registerUser(newUser);
    }
}
```

In this example the `registerUser` method accepts a new `User` object, passes it onto the service to be persisted and then sends the persisted `User` in the response.
Take another look at the `User` entity, is there any data in that class that you wouldn't be comfortable with being public knowledge?

Obviously, we don't want to be returning the `password` variable as that's private information (even if it's hashed), so how can we return the relevant information *without* revealing the password?

The answer is to create a `UserDTO`:

```
public class UserDTO {

    private Long id;

    private String username;

    public UserDTO(User user) {
        this.id = user.getId();
        this.username = user.getUsername();
    }

    // getters and setters
}
```

We can then return one of these in our controller instead of the `User` object

```
@RestController
public class UserController {

    private UserService service;

    // constructor

    @PostMapping("/register")
    public UserDTO registerUser(@RequestBody User newUser) {
        User registeredUser = this.service.registerUser(newUser);
        return new UserDTO(registeredUser);
    }
}
```

So now we're only returning the public data and the private data never leaves Spring!

Tutorial

Let's add DTOs to our previous example!

At the moment we have our `Person` entity:

```
@Entity
public class Person {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    private int age;

    public Person() {
        super();
    }

    public Person(String name, int age) {
        super();
        this.name = name;
        this.age = age;
    }

    // getters and setters
}
```

So let's make a `PersonDTO`:

```
public class PersonDTO {

    private Long id;

    private String name;

    private int age;

    public PersonDTO() {
        super();
    }

    // getters and setters
}
```

In this case our DTO is pretty much the same as the entity, and that's ok! It's still good practice to have separate classes for entities and Data Transfer Objects.

Now that we've got that set up we need a simple way to convert from the entity to the DTO and to do that we'll be using an external library - **`ModelMapper`**.

The first thing we'll do is add its dependency to the `pom.xml`:

```
    <dependency>
        <groupId>org.modelmapper</groupId>
        <artifactId>modelmapper</artifactId>
        <version>2.3.8</version>
    </dependency>
```

Since we only need one instance of the `ModelMapper` across the entire Spring application, we'll create a singleton-scoped bean for it inside our `AppConfig` class:

```
@Configuration
public class AppConfig {

    @Bean
    public ModelMapper mapper()
        return new ModelMapper();
    }

}
```

Now we can add the `ModelMapper` bean to the `PersonService` as a dependency:

```
@Service
public class PersonService {

    private PersonRepo repo;

    private ModelMapper mapper;

    public PersonService(PersonRepo repo, ModelMapper mapper) {
        super();
        this.repo = repo;
        this.mapper = mapper;
    }

    // CRUD methods
}
```

Next, create a method in `PersonService` that uses the `MethodMapper` to convert a `Person` to a `PersonDTO`:

```
    private PersonDTO mapToDTO(Person person) {
        return this.mapper.map(person, PersonDTO.class);
    }
```

The `map` method of `ModelMapper` accepts an `Object` 'source' (whatever object you want to convert - `person` in this case) and the `Class` of whatever type you wish to convert it into (`PersonDTO). The`ModelMapper`will then iterate through the fields of the 'source' (`person`) and extract the data using the **getters** which will then be inserted into the`PersonDTO` via its **setters**.

So what we've just created is roughly equivalent to:

```
    private PersonDTO mapToDTO(Person person) {
        PersonDTO dto = new PersonDTO();
        dto.setAge(person.getAge());
        dto.setId(person.getId());
        dto.setName(person.getName());
        return dto;
    }
```

As we can see, the `ModelMapper` allows us to cut out *a lot* of boilerplate code! When taking advantage of this functionality it is important to remember that the `ModelMapper` will automatically transfer data *only* if the fields have the *exact same names* so always remember to check that your property names match in your entities and DTOs!

From here it's a simple matter of updating the `PersonService` and `PersonController` to work with it:

```java
@Service
public class PersonService {

    private PersonRepo repo;

    private ModelMapper mapper;

    public PersonService(PersonRepo repo, ModelMapper mapper) {
        super();
        this.repo = repo;
        this.mapper = mapper;
    }

    private PersonDTO mapToDTO(Person person) {
        return this.mapper.map(person, PersonDTO.class);
    }

    public PersonDTO addPerson(Person person) {
        Person saved =  this.repo.save(person);
        return this.mapToDTO(saved);
    }

    public List<PersonDTO> getAllPeople() {
        return
this.repo.findAll().stream().map(this::mapToDTO).collect(Collectors.toList());
    }

    public PersonDTO updatePerson(Long id, Person newPerson) {
        Optional<Person> existingOptional = this.repo.findById(id);
        Person existing = existingOptional.get();

        existing.setAge(newPerson.getAge());
        existing.setName(newPerson.getName());

        Person updated =  this.repo.save(existing);
        return this.mapToDTO(updated);
    }

    public boolean removePerson(Long id) {
        this.repo.deleteById(id);
        boolean exists = this.repo.existsById(id);
        return !exists;
    }

}
```

```java
@RestController
public class PersonController {

    private PersonService service;

    public PersonController(PersonService service) {
        super();
        this.service = service;
    }

    @PostMapping("/create")
    public PersonDTO addPerson(@RequestBody Person person) {
        return this.service.addPerson(person);
    }

    @GetMapping("/getAll")
    public List<PersonDTO> getAllPeople() {
        return this.service.getAllPeople();
    }

    @PutMapping("/update")
    public PersonDTO updatePerson(@PathParam("id") Long id, @RequestBody Person
person) {
        return this.service.updatePerson(id, person);
    }

    @DeleteMapping("/delete/{id}")
    public boolean removePerson(@PathVariable Long id) {
        return this.service.removePerson(id);
    }

    @GetMapping("/test")
    public String test() {
        return "Hello, World!";
    }

}
```

Exercises

Implement an `AccountDTO` in your account project and update the service and controller to work with it.