Преобразование DTO в сущность

Помимо преобразования в DTO, существует и обратная задача — преобразование DTO в Entity. Зачем это делать, если можно наполнять сущность напрямую?

Первая причина — это безопасность. Когда у нас есть API для создания или изменения сущности, обычно мы хотим дать возможность менять только часть свойств. Но если мы используем в `@RequestBody` нашу сущность напрямую, то у клиента API появляется возможность поменять любые свойства сущности:

```
@PutMapping("/users/{id}")
@ResponseStatus(HttpStatus.OK)
// Клиенты могут менять все свойства внутри пользователя
public UserDTO update(@RequestBody User user, @PathVariable Long id) {
    repository.save(user);
}
```

Мы не советуем использовать `userData` как сущность и сразу сохранять в базу — такой подход создает потенциальную опасность.

Вторая причина — схема данных. Со временем именование свойств может меняться и в базе данных, и в API — например, при внедрении новой версии. Разделение сущностей и DTO позволяет делать это независимо. DTO представляет внешний интерфейс для API. В свою очередь, сущности описывают внутреннюю модель данных.

Кроме того, существует еще несколько причин, которые мы разберем подробнее в других уроках:

- Дополнительные преобразования данных перед тем, как они попадут в сущность — например, нормализация электронной почты.
- Дополнительная валидация, которая может понадобиться в конкретном API. Хорошим примером служит подтверждение пароля. Подтверждение пароля не существует на уровне сущности, это вопрос проверки корректности входных данных.

Преобразование из сущности в DTO и наоборот обычно отличаются набором свойств. Например, в большинстве случаев идентификатор генерируется в базе данных — мы не хотим передавать его в API. При этом при возврате ответа в API мы хотим вернуть

идентификатор среди остальных свойств. Поэтому есть смысл создавать разные DTO для этих задач.

Разберем пример с созданием и выводом сущности `Post`:

```
package io.hexlet.spring.model;

import static jakarta.persistence.GenerationType.IDENTITY;

import java.time.LocalDate;

import org.springframework.data.annotation.CreatedDate;
import org.springframework.data.jpa.domain.support.AuditingEntityListener;

import jakarta.persistence.EntityListeners;
import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.Id;
import jakarta.persistence.Table;
import lombok.Getter;
import lombok.Setter;

@Entity
@Getter
@Setter
@Table(name = "posts")
@EntityListeners(AuditingEntityListener.class)
public class Post {
    @Id
    @GeneratedValue(strategy = IDENTITY)
    private Long id;

    @Column(unique = true)
    private String slug;

    private String name;

    @Column(columnDefinition = "TEXT")
    private String body;

    @CreatedDate
```

```
    private LocalDate createdAt;
}
```

Создадим два DTO для каждого действия. Создание потребует три поля — `slug`, `name` и `body`. В вывод добавятся поля `id` и `createdAt`:

```
// Создание поста
package io.hexlet.spring.dto;

import lombok.Getter;
import lombok.Setter;

@Setter
@Getter
public class PostCreateDTO {
    private String slug;
    private String name;
    private String body;
}
```

```
// Вывод поста
package io.hexlet.spring.dto;

import lombok.Getter;
import lombok.Setter;

@Setter
@Getter
public class PostDTO {
    private String id;
    private String slug;
    private String name;
    private String body;
    private LocalDate createdAt;
}
```

Реализуем создание и вывод. Вывод потребует преобразования только в DTO, а создание — оба преобразования (из сущности в DTO и наоборот):

```
package io.hexlet.spring.controller.api;
```

```java
import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;

import io.hexlet.spring.dto.PostCreateDTO;
import io.hexlet.spring.dto.PostDTO;
import io.hexlet.spring.model.Post;
import io.hexlet.spring.exception.ResourceNotFoundException;
import io.hexlet.spring.repository.PostRepository;

@RestController
@RequestMapping("/api")
public class PostsController {
    @Autowired
    private PostRepository repository;

    @GetMapping("/posts/{id}")
    @ResponseStatus(HttpStatus.OK)
    public PostDTO show(@PathVariable Long id) {
        var post = repository.findById(id)
                .orElseThrow(() -> new ResourceNotFoundException("Not Found: " + id));
        var postDTO = toDTO(post); // Только в DTO
        return postDTO;
    }

    @PostMapping("/posts")
    @ResponseStatus(HttpStatus.CREATED)
    public PostDTO create(@RequestBody PostCreateDTO postData) {
        var post = toEntity(postData); // Сначала в Entity
        repository.save(post);
        var postDTO = toDTO(post); // Потом в DTO
```

```
        return postDTO;
    }

    private PostDTO toDTO(Post post) {
        var dto = new PostDTO();
        dto.setId(post.getId());
        dto.setSlug(post.getSlug());
        dto.setName(post.getName());
        dto.setBody(post.getBody());
        dto.setCreatedAt(post.getCreatedAt());
        return dto;
    }

    private Post toEntity(PostCreateDTO postDto) {
        var post = new Post();
        post.setSlug(postDto.getSlug());
        post.setName(postDto.getName());
        post.setBody(postDto.getBody());
        return post;
    }
}
```

В методе `create()` мы поменяли тип входных данных на `PostCreateDTO`. Уже внутри эти данные копируются в только что созданный объект `post`. После сохранения в базу данных мы снова выполняем преобразование из `Post` в `PostDTO`, чтобы сформировать тело ответа. На этом моменте проявляется разница между тем, что приходит на вход, и тем, что должно быть на выходе. Например, идентификатор появляется только после того, как мы выполняем сохранение в базу данных. Поэтому мы получаем такую цепочку: *PostCreateDTO => Post => PostDTO*.

Далее →