Автоматическая конвертация сущностей в DTO и обратно

- Установка
- Использование
  - Сущность
  - DTO
  - Контроллер
  - Мапперы

Преобразование сущностей в DTO и обратно — это довольно утомительная операция с большим объемом однообразного кода. Например, такого:

```
// Импорты

@RestController
@RequestMapping("/api")
public class PostsController {
    // Здесь обработчики

    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;
    }
```

```
    private Post toEntity(PostUpdateDTO postDto, Post post) {
        post.setName(postDto.getName());
        post.setBody(postDto.getBody());
        return post;
    }
}
```

Кто-то в своих проектах выбирает такой подход, но есть и альтернатива. Существуют библиотеки, позволяющие автоматизировать конвертацию в обе стороны. Самая популярная из них — это MapStruct. В этом уроке мы подключим ее и научимся использовать.

Установка

Для начала установим MapStruct:

```
implementation("org.mapstruct:mapstruct:1.5.5.Final")
annotationProcessor("org.mapstruct:mapstruct-processor:1.5.5.Final")
```

Кроме обычной зависимости, MapStruct требует еще и установки обработчика аннотации. Такие обработчики выполняются во время компиляции и используются для генерации кода. Ниже мы увидим, зачем это делается и как работает.

Использование

MapStruct работает следующим образом. С его помощью создаются специальные мапперы под каждую сущность. Внутри них определяются правила конвертирования в DTO или из DTO в зависимости от потребностей. Дальше эти мапперы используются в нужных местах, сводя преобразования к одной строчке.

Разберем работу библиотеки на примере сущности `Post`, взятой из предыдущего урока. Чтобы было понятнее, мы начнем с конца. Сначала посмотрим, как использовать мапперы, а затем научимся их писать.

Сущность

Определение сущности выглядит так:

```java
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;
}
```

DTO

Для этой сущности реализуем три DTO для разных целей:

- Для создания сущности
- Для обновления сущности
- Для отображения сущности

Реализация выглядит так:

```
// 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;
}


// Обновление поста
package io.hexlet.spring.dto;

import lombok.Getter;
import lombok.Setter;

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

Контроллер

В конце мы напишем маппер, а пока посмотрим, как изменится код контроллера с их использованием. Все преобразование сведется к вызову `postMapper.map()`:

```
// Остальные импорты
import io.hexlet.spring.mapper.PostMapper;

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

    @Autowired
    private PostMapper postMapper;

    @PostMapping("/posts")
    @ResponseStatus(HttpStatus.CREATED)
    public PostDTO create(@RequestBody PostCreateDTO postData) {
        // Преобразование в сущность
        var post = postMapper.map(postData);
        repository.save(post);
        // Преобразование в DTO
        var postDTO = postMapper.map(post);
        return postDTO;
    }
```

```
    @PutMapping("/posts/{id}")
    @ResponseStatus(HttpStatus.OK)
    public PostDTO update(@RequestBody @Valid PostUpdateDTO postData, @PathVariable Long i
        var post = repository.findById(id)
                .orElseThrow(() -> new ResourceNotFoundException("Not Found"));
        postMapper.update(postData, post);
        repository.save(post);
        var postDTO = postMapper.map(post);
        return postDTO;
    }

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

Код контроллера стал предельно простым. Вместо ручного копирования данных здесь используется маппер `PostMapper`, который содержит:

- Метод `update()`
- Перегруженный метод `map()`, работающий сразу с тремя классами:
  - `PostCreateDTO`
  - `PostDTO`
  - `Post`

Мапперы

Перейдем к мапперам:

```
// src/main/java/io/hexlet/spring/mapper/PostMapper.java
package io.hexlet.spring.mapper;

import org.mapstruct.Mapper;
```

```java
import org.mapstruct.MappingConstants;
import org.mapstruct.MappingTarget;
import org.mapstruct.NullValuePropertyMappingStrategy;
import org.mapstruct.ReportingPolicy;

import io.hexlet.spring.dto.PostCreateDTO;
import io.hexlet.spring.dto.PostUpdateDTO;
import io.hexlet.spring.dto.PostDTO;
import io.hexlet.spring.model.Post;

@Mapper(
    nullValuePropertyMappingStrategy = NullValuePropertyMappingStrategy.IGNORE,
    componentModel = MappingConstants.ComponentModel.SPRING,
    unmappedTargetPolicy = ReportingPolicy.IGNORE
)
public abstract class PostMapper {
    public abstract Post map(PostCreateDTO dto);
    public abstract PostDTO map(Post model);
    public abstract void update(PostUpdateDTO dto, @MappingTarget Post model);
}
```

Маппер — это абстрактный класс с абстрактными методами для конвертации одних объектов в другие. Класс должен быть помечен аннотацией `@Mapper` с минимально указанной опцией `componentModel = MappingConstants.ComponentModel.SPRING`. Расположение класса, название класса и методов не фиксированы — программисты сами определяют, как все это организовать. MapStruct не ограничивает нас в DTO, мы можем преобразовывать объекты любых классов.

В документации MapStruct показаны примеры с интерфейсом, а не абстрактным классом. Технически эта библиотека работает и с интерфейсами, и абстрактными классами. Использовать последние удобнее, потому что в абстрактные классы можно сделать инъекцию зависимостей, если это необходимо.

Во время компиляции происходит генерация конкретных мапперов. Посмотреть исходник этих классов можно в директории *build/generated/sources/annotationProcessor/java/main/io/spring/mapper*. Это очень упрощает отладку. Код маппера `PostMapperImpl` созданного на базе абстрактного класса `PostMapper`:

```java
// PostMapperImpl.java
package io.hexlet.blog.mapper;
```

```java
import io.hexlet.blog.dto.PostCreateDTO;
import io.hexlet.blog.dto.PostDTO;
import io.hexlet.blog.model.Post;
import io.hexlet.blog.model.User;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

// Автоматически помечается как компонент, что дает возможность внедрять как зависимость
@Component
public class PostMapperImpl extends PostMapper {
   @Override
   public Post map(PostCreateDTO dto) {
      if (dto == null) {
         return null;
      } else {
         Post post = new Post();
         post.setSlug(dto.getSlug());
         post.setName(dto.getName());
         post.setBody(dto.getBody());
         return post;
      }
   }

   @Override
   public PostDTO map(Post model) {
      if (model == null) {
         return null;
      } else {
         PostDTO postDTO = new PostDTO();
         postDTO.setId(model.getId());
         postDTO.setSlug(model.getSlug());
         postDTO.setName(model.getName());
         postDTO.setBody(model.getBody());
         postDTO.setCreatedAt(model.getCreatedAt());
         return postDTO;
      }
   }

    @Override
    public void update(PostUpdateDTO dto, Post model) {
        if ( dto == null ) {
```

```
            return;
        }
        model.setName(dto.getName());
        model.setBody(dto.getBody());
    }
}
```

MapStruct самостоятельно написал тот код, который мы до этого писали руками. Но как он это сделал? MapStruct сравнивает методы обоих классов и автоматически распознает те, что совпадают. Кроме этого, MapStruct автоматически пытается преобразовать типы, если они не совпадают. В большинстве случаев это работает автоматически, но там где нет, всегда есть возможность дописать правила конвертации и преобразования типов. Для примера представим, что поле `name` переименовали в `title`. Если нам нужно сохранить внешнее API без изменений, то мы можем определить правила преобразования в маппере:

```
package io.hexlet.spring.mapper;

import org.mapstruct.Mapper;
import org.mapstruct.MappingConstants;
import org.mapstruct.NullValuePropertyMappingStrategy;
import org.mapstruct.ReportingPolicy;

import io.hexlet.spring.dto.PostCreateDTO;
import io.hexlet.spring.dto.PostDTO;
import io.hexlet.spring.model.Post;

@Mapper(
    nullValuePropertyMappingStrategy = NullValuePropertyMappingStrategy.IGNORE,
    componentModel = MappingConstants.ComponentModel.SPRING,
    unmappedTargetPolicy = ReportingPolicy.IGNORE
)
public abstract class PostMapper {
    @Mapping(target = "title", source = "name")
    public abstract Post map(PostCreateDTO dto);

    @Mapping(target = "title", source = "name")
    public abstract void update(PostUpdateDTO dto, @MappingTarget Post model);

    @Mapping(target = "name", source = "title")
    public abstract PostDTO map(Post model);
}
```

Аннотация `@Mapping` позволяет указать правила преобразования свойств. Самый частый случай — это когда имя свойства в исходном объекте не совпадает с целевым. В аннотации `source` указывает на объект, который передается как параметр, `target` — это объект, возвращаемый из метода.

Дополнительные материалы

1. Официальная документация
2. Mapstruct Spring Extensions

Далее →