







# Проектирование

Урок 6



## Что будет на уроке сегодня

-  API: возвращение к основам
-  CRUD и интеграции
-  Проектирование API
-  Инструменты для проектирования API
-  Автоматическая документация API
-  Генерация документации Swagger



# API: возвращение к основам

API – это ключевой элемент любого веб-приложения. Это то, что позволяет нам создавать интерактивные, динамические и полнофункциональные приложения, которые мы видим и используем каждый день.

API могут быть различными: веб-API, операционные системы имеют свои API, библиотеки программирования также используют API.





## CRUD и интеграции

**1. Создание (Create).**

**2. Чтение (Read).**

**3. Обновление (Update).**

**4. Удаление (Delete).**



# Проектирование API

Если API плохо спроектирован, это может вызвать ряд проблем:

**1. Непредсказуемое поведение.**

**3. Проблемы с безопасностью.**

**2. Сложности в поддержке.**

**4. Плохой пользовательский опыт.**



## Обработка ошибок

1

**4xx ошибки клиента**

2

**5xx ошибки сервера**



## Исключения в Spring

```
1 @GetMapping("/users/{id}")
2 public User getUser(@PathVariable Long id) {
3     return userRepository.findById(id)
4         .orElseThrow(() -> new ResourceNotFoundException("User not found with id " + id));
5 }
```

```
1 @ControllerAdvice
2 public class GlobalExceptionHandler {
3
4     @ExceptionHandler(ResourceNotFoundException.class)
5     public ResponseEntity<?> handleResourceNotFoundException(ResourceNotFoundException
6         exception) {
7         return ResponseEntity
8             .status(HttpStatus.NOT_FOUND)
9             .body(exception.getMessage());
10 }
```



# Практика

## Шаг 1: Проектирование API сервиса для книжного магазина

Давайте начнем с проектирования эндпоинтов:

1. GET /books.
2. GET /books/{id}.
3. POST /books.
4. PUT /books/{id}.
5. DELETE /books/{id}.





## Практика

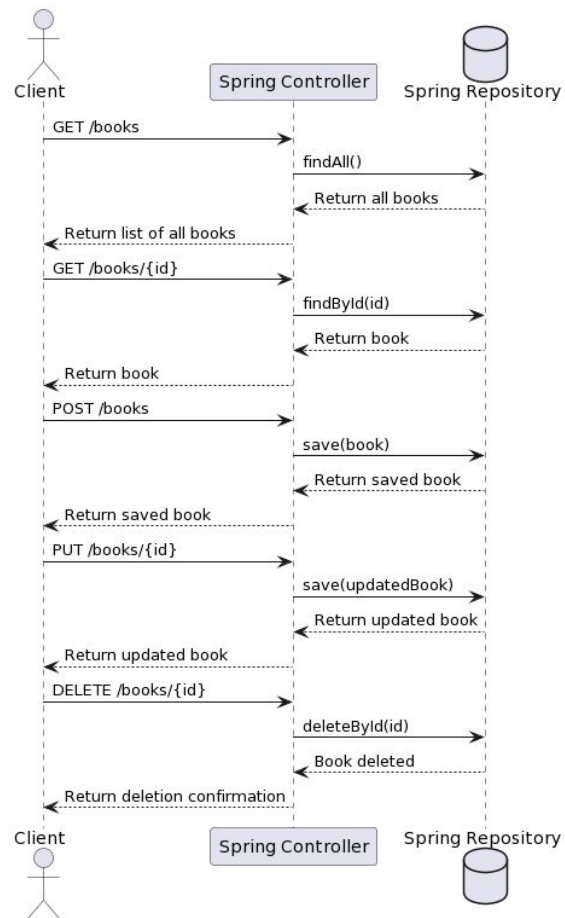
Теперь, давайте определим структуру данных для нашей книги.

Каждая книга будет иметь следующие атрибуты:

- id: Уникальный идентификатор книги.
- title: Название книги.
- author: Автор книги.
- isbn: Уникальный номер ISBN книги.
- publicationYear: Год публикации книги.



# Практика





# Практика

## Шаг 2: Создание проекта и моделей

1. Перейдите на Spring Initializr.
2. Выберите “Maven Project”, “Java” и версию Spring Boot.
3. Введите информацию о проекте в соответствующие поля.
4. Добавьте необходимые зависимости. Для нашего проекта нам понадобятся:  
Spring Web, Spring Data JPA, Spring HATEOAS и H2 Database.
5. Нажмите “Generate” для создания проекта.



## Практика

```
9      @Id
10     @GeneratedValue(strategy = GenerationType.AUTO)
11     private Long id;
12
13     private String title;
14
15     private String author;
16
17     private String isbn;
18
19     private int publicationYear;
20
21     // getters and setters
22 }
```



## Практика

### Шаг 3: Реализация слоя репозитория с использованием Spring Data JPA

```
1 import org.springframework.data.jpa.repository.JpaRepository;  
2  
3 public interface BookRepository extends JpaRepository<Book, Long> {  
4 }
```



# Практика

## Шаг 4: Разработка слоя сервисов

1. Слой обеспечивает разделение ответственности.
2. Слой сервисов делает наш код более переиспользуемым.
3. Слой сервисов облегчает тестирование.



## Практика

### Шаг 5: Реализация слоя сервисов с использованием Spring

```
1 import org.springframework.beans.factory.annotation.Autowired;
2 import org.springframework.stereotype.Service;
3 import java.util.List;
4 import java.util.Optional;
5
6 @Service
7 public class BookService {
8
9     private final BookRepository bookRepository;
10
```



## Практика

### Шаг 5: Реализация слоя сервисов с использованием Spring

```
11     @Autowired
12     public BookService(BookRepository bookRepository) {
13         this.bookRepository = bookRepository;
14     }
15
16     public List<Book> findAll() {
17         return bookRepository.findAll();
18     }
19
20     public Optional<Book> findById(Long id) {
```





## Практика

### Шаг 5: Реализация слоя сервисов с использованием Spring

```
21         return bookRepository.findById(id);
22     }
23
24     public Book save(Book book) {
25         return bookRepository.save(book);
26     }
27
28     public void deleteById(Long id) {
29         bookRepository.deleteById(id);
30     }
31 }
```



## Практика

### Шаг 6: Реализация контроллера с использованием Spring

```
1 import org.springframework.beans.factory.annotation.Autowired;
2 import org.springframework.http.ResponseEntity;
3 import org.springframework.web.bind.annotation.*;
4
5 import java.util.List;
6 import java.util.Optional;
7
8 @RestController
9 @RequestMapping("/books")
10 public class BookController {
11
12     private final BookService bookService;
13 }
```



## Практика

### Шаг 6: Реализация контроллера с использованием Spring

```
14     @Autowired
15     public BookController(BookService bookService) {
16         this.bookService = bookService;
17     }
18
19     @GetMapping
20     public List<Book> findAll() {
21         return bookService.findAll();
22     }
23
24     @GetMapping("/{id}")
25     public ResponseEntity<Book> findById(@PathVariable Long id) {
26         Optional<Book> book = bookService.findById(id);
```



## Практика

### Шаг 6: Реализация контроллера с использованием Spring

```
26         Optional<Book> book = bookService.findById(id);
27         return book.map(ResponseEntity::ok)
28             .orElseGet(() → ResponseEntity.notFound().build());
29     }
30
31     @PostMapping
32     public Book save(@RequestBody Book book) {
33         return bookService.save(book);
34     }
35
```



## Практика

### Шаг 6: Реализация контроллера с использованием Spring

```
36     @PutMapping("/{id}")
37     public Book update(@RequestBody Book book, @PathVariable Long id) {
38         book.setId(id);
39         return bookService.save(book);
40     }
41
42     @DeleteMapping("/{id}")
43     public void deleteById(@PathVariable Long id) {
44         bookService.deleteById(id);
45     }
46 }
```



## Запуск и тестирование сервиса

**1. GET /books.**

**2. POST /books.**

**3. GET /books/{id}.**

**4. PUT /books/{id}.**

**5. DELETE /books/{id}.**



## Возможные проблемы

**1. Недостаток структуры и организации.**

**2. Неэффективность и повторение кода.**

**3. Проблемы с производительностью.**

**4. Трудности в интеграции.**

**5. Увеличение времени и стоимости разработки.**



# Инструменты для проектирования API

1

**Swagger (OpenAPI)**

3

**Apiary**

5

**Microsoft Azure API Management**

2

**Postman**

4

**AWS CloudFormation**

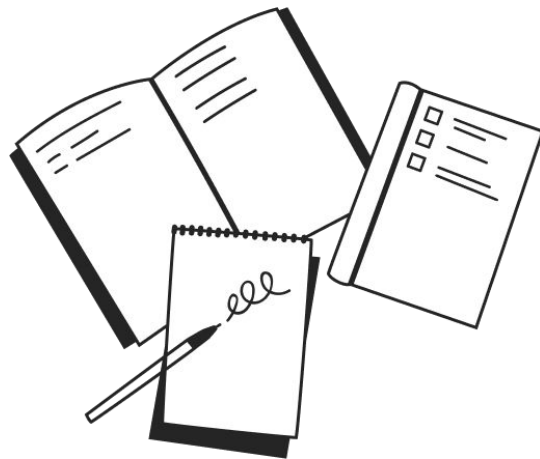




# Автоматическая документация API

Существуют два основных способа использования Swagger для автоматической документации:

1. Аннотации Swagger в коде.
2. Swagger Codegen.





## Генерация документации Swagger

```
1 <!-- для Maven -->
2 <dependency>
3     <groupId>io.springfox</groupId>
4     <artifactId>springfox-boot-starter</artifactId>
5     <version>3.0.0</version>
6 </dependency>
7 // для Gradle
8 implementation 'io.springfox:springfox-boot-starter:3.0.0'
```



## Генерация документации Swagger

```
15 @EnableSwagger2
16 @Configuration
17 public class SwaggerConfig {
18     @Bean
19     public Docket api() {
20         return new Docket(DocumentationType.SWAGGER_2)
21             .select()
22             .apis(RequestHandlerSelectors.basePackage("com.example"))
23             .build();
24     }
25 }
```



**Спасибо за внимание**

