HTTP и CRUD приложения

- Добавление страницы
- Показ страницы
- Удаление страницы
- Список страниц
- Обновление страницы

В этом уроке мы разберем основные элементы Spring Boot для работы с HTTP на примере CRUD приложения для создания страниц сайта. Под такими страницами обычно понимают что-то подобное:

```
https://ru.hexlet.io/pages/about
```

Нам понадобится модель страницы. Для начала создадим класс `Page` с тремя полями:

- *slug* — слаг (идентификатор, который используется для построения ссылок)
- *name* — имя страницы
- *body* — содержимое страницы

```
// src/main/java/io/hexlet/spring/model/Page.java
package io.hexlet.spring.model;

import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

@NoArgsConstructor
@Setter
@Getter
public class Page {
    // Используется как идентификатор в ссылках /pages/about
    // Здесь about — это слаг
    private String slug;
    private String name;
    private String body;
}
```

Структура директорий может быть любой, потому что Spring не накладывает ограничений на нее. Мы будем придерживаться общепринятых стандартов и хранить наши файлы так, как принято в реальных проектах. Например, модели часто кладут в директорию *model*:

```
tree src
src
└── main
    └── java
        └── io
            └── hexlet
                └── spring
                    └── Application.java
                    model
                    └── Page.java
```

Следующий шаг — реализуем пять CRUD-маршрутов:

Метод	Маршрут	Описание
GET	/pages	Список страниц
GET	/pages/{id}	Страница
POST	/pages	Создание новой страницы
PUT	/pages/{id}	Обновление страницы
DELETE	/pages/{id}	Удаление страницы

Для реализации этих маршрутов нужна возможность хранить добавленные данные. В будущих уроках мы научимся делать это через базу данных, а сейчас просто будем сохранять данные в поле класса.

Взаимодействие с этим приложением с использованием консольной утилиты `http` для выполнения запросов выглядит так:

```
# Это создание страницы — здесь возвращается информация о добавленной странице
http post localhost:8080/pages slug=someslug name=somename body=somebody

{
    "body": "somebody",
    "name": "somename",
```

```
    "slug": "someslug"
}

# Это просмотр созданной страницы  – здесь возвращается информация о странице
http localhost:8080/pages/someslug

{
    "body": "somebody",
    "name": "somename",
    "slug": "someslug"
}

# Это добавление еще одной страницы
http post localhost:8080/pages slug=someslug2 name=somename2 body=somebody2

{
    "body": "somebody2",
    "name": "somename2",
    "slug": "someslug2"
}

// # Это вывод списка добавленных страниц – здесь возвращается информация обо всех страниц
http localhost:8080/pages                                                    <<<

[
    {
        "body": "somebody",
        "name": "somename",
        "slug": "someslug"
    },
    {
        "body": "somebody2",
        "name": "somename2",
        "slug": "someslug2"
    }
]

# Это вывод списка страниц с указанием лимита вывода – в примере возвращается один элемент
http "localhost:8080/pages?limit=1"

[
    {
```

```
        "body": "somebody",
        "name": "somename",
        "slug": "someslug"
    }
]

# Это удаление страницы — ничего не возвращается
http delete localhost:8080/pages/someslug

# Это просмотр списка страниц — выводится всего одна страница, потому что вторая удалена
http localhost:8080/pages                                                    <<<

[
    {
        "body": "somebody2",
        "name": "somename2",
        "slug": "someslug2"
    }
]
```

Наше приложение возвращает данные в JSON-формате, причем в зависимости от ситуации:

- Приложение возвращает объект, если мы работаем с одиночным ресурсом — например, созданием
- Приложение возвращает массив, если мы работаем с коллекцией — например, списком

Подробнее на эту тему мы поговорим в уроке про REST API.

Рассмотрим пример кода такого приложения:

```
package io.hexlet.spring;

import java.util.ArrayList;
import java.util.List;
import java.util.Optional;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
```

```java
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.RequestParam;
import org.springframework.web.bind.annotation.RestController;

import io.hexlet.spring.model.Page;

@SpringBootApplication
@RestController
public class Application {
    // Хранилище добавленных страниц, то есть обычный список
    private List<Page> pages = new ArrayList<Page>();

    // Запуск приложения
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

    @GetMapping("/pages") // Список страниц
    public List<Page> index(@RequestParam(defaultValue = "10") Integer limit) {
        return pages.stream().limit(limit).toList();
    }

    @PostMapping("/pages") // Создание страницы
    public Page create(@RequestBody Page page) {
        pages.add(page);
        return page;
    }

    @GetMapping("/pages/{id}") // Вывод страницы
    public Optional<Page> show(@PathVariable String id) {
        var page = pages.stream()
            .filter(p -> p.getSlug().equals(id))
            .findFirst();
        return page;
    }

    @PutMapping("/pages/{id}") // Обновление страницы
    public Page update(@PathVariable String id, @RequestBody Page data) {
        var maybePage = pages.stream()
```

```
            .filter(p -> p.getSlug().equals(id))
            .findFirst();
        if (maybePage.isPresent()) {
            var page = maybePage.get();
            page.setSlug(data.getSlug());
            page.setName(data.getName());
            page.setBody(data.getBody());
        }
        return data;
    }

    @DeleteMapping("/pages/{id}") // Удаление страницы
    public void destroy(@PathVariable String id) {
        pages.removeIf(p -> p.getSlug().equals(id));
    }
}
```

Сначала разберем общие концепции, а затем поговорим о каждом обработчике в отдельности.

Над классом приложения добавлена аннотация `@RestController`. Эта аннотация указывает на классы, которые содержат обработчики маршрутов. Пока мы помещаем обработчики в том же классе, в котором стартует приложение — так проще. В будущем мы будем размещать обработчики в контроллерах.

Каждый обработчик помечен аннотацией описания маршрута. Эти аннотации отвечают за то, какой маршрут обрабатывает обработчик и какой метод HTTP при этом используется. Это не одна аннотация, а набор аннотаций под каждый HTTP-метод — `@GetMapping()`, `@DeleteMapping` и так далее.

Каждый обработчик возвращает либо коллекцию объектов, либо объект, либо ничего не возвращает. А как Java-сущности превращаются в JSON? Это происходит автоматически с помощью библиотеки Jackson. Она входит в пакет *spring-boot-starter-json*, который подключается через *spring-boot-starter-web*.

Автоматическая конвертация — это очень удобная штука, которая убирает шаблонный код.

При работе со Spring Boot имена обработчиков не принципиальны. Намного важнее, какой маршрут они обрабатывают. Здесь мы используем подход к именованию, принятый в большинстве веб-фреймворках.

Добавление страницы

Обработчик добавления страницы принимает на вход данные страницы, добавляет их в коллекцию страниц и возвращает эти же данные наружу:

```
@PostMapping("/pages")
public Page create(@RequestBody Page page) {
    pages.add(page);
    return page;
}
```

Обсудим, как данные из HTTP-тела попадают в код. Это происходит с помощью аннотации `@RequestBody`, которой помечается переменная с типом `Page`. Имя переменной не важно. Spring Boot автоматически создает объект этого типа и заполняет его данными, которые были в теле. Для работы этого механизма имена полей в `Page` должны совпадать с именами в теле запроса. При этом данных может быть отправлено меньше, чем содержится в `Page`.

Показ страницы

Обработчик вывода страницы принимает на вход идентификатор, в качестве которого выступает слаг страницы. Затем обработчик ищет эту страницу в списке страниц и возвращает наружу. Страница может не найтись, поэтому возвращается `Optional`:

```
@GetMapping("/pages/{id}")
public Optional<Page> show(@PathVariable String id) {
    var page = pages.stream()
        .filter(p -> p.getSlug().equals(id))
        .findFirst();
    return page;
}
```

Здесь мы видим работу с параметрами пути в Spring Boot. В маршруте они указываются через фигурные скобки `{id}`, а в параметрах метода — извлекаются с помощью аннотации `@PathVariable`. Имя переменной должно совпадать с именем переменной в шаблоне, потому что происходит сопоставление.

Удаление страницы

Удаление страницы просто удаляет объект из списка страниц. Сам метод ничего не возвращает:

```
@DeleteMapping("/pages/{id}")
public void destroy(@PathVariable String id) {
    pages.removeIf(p -> p.getSlug().equals(id));
}
```

Список страниц

Вывод списка возвращает список страниц с возможностью указать лимит по выводу:

```
@GetMapping("/pages") // Список страниц
public List<Page> index(@RequestParam(defaultValue = "10") Integer limit) {
    return pages.stream().limit(limit).toList();
}
```

Лимит определяется параметром запроса `localhost:8080/pages?limit=1`. Параметры запроса попадают в код через параметры обработчика, помеченные аннотацией `@RequestParam`. У параметра можно задать значение по умолчанию в аннотации с помощью `defaultValue`. Имя переменной сопоставляется с именем параметра запроса.

Обновление страницы

Обновление устроено сложнее всего. Этот обработчик принимает на вход идентификатор обновляемой страницы и новые данные. Затем выполняется поиск необходимой страницы в общем списке. Если страница найдена, то выполняется ее обновление:

```
@PutMapping("/pages/{id}") // Обновление страницы
public Page update(@PathVariable String id, @RequestBody Page data) {
    var maybePage = pages.stream()
        .filter(p -> p.getSlug().equals(id))
        .findFirst();
    if (maybePage.isPresent()) {
        var page = maybePage.get();
        page.setSlug(data.getSlug());
```

```
            page.setName(data.getName());
            page.setBody(data.getBody());
        }
        return data;
    }
```

Здесь используются уже знакомые нам концепции, но они появляются вместе. Параметры в обработчиках могут идти в любом порядке, с любыми именами и допустимыми аннотациями. Технически вызов таких методов работает через механизм Reflection, который позволяет определить сигнатуру метода и правильно вызывать его в рантайме.

Далее →