

## **Лабораторная работа №1. Введение в Spring Boot**

Spring - это фреймворк для разработки приложений на языке Java, который обеспечивает комплексный набор инструментов и библиотек для упрощения создания веб-приложений. Spring предоставляет модульную архитектуру, которая позволяет разработчикам выбирать только те компоненты, которые им необходимы, и интегрировать их в свои проекты.

Основные возможности Spring:

Внедрение зависимостей (Dependency Injection) - механизм, позволяющий управлять зависимостями между компонентами приложения, облегчая тестирование и повторное использование кода.

Аспектно-ориентированное программирование (Aspect-Oriented Programming) - позволяет выносить пересекающиеся аспекты функциональности приложения в отдельные модули, что способствует уменьшению дублирования кода и повышению его модульности.

Модульная архитектура - Spring предоставляет множество модулей, которые можно использовать по мере необходимости, включая модули для веб-разработки, доступа к данным, безопасности и другие.

Облегченная разработка веб-приложений - с помощью модуля Spring MVC (Model-View-Controller) разработчики могут создавать веб-приложения, используя конфигурацию через аннотации или XML.

Поддержка транзакций - Spring предоставляет механизм управления транзакциями, позволяя разработчикам управлять транзакционным поведением своих приложений.

Java Spring широко используется в индустрии разработки программного обеспечения и является одним из самых популярных фреймворков для создания Java-приложений.

**Цель работы:** Получить практические навыки работы с Spring Framework.

**Пример выполнения работы.**

Для начала нам необходимо создать проект, который будет поддерживать Spring.

Шаг 1: Перейдите на веб-сайт Spring Initializr по адресу <https://start.spring.io/>.

Шаг 2: На главной странице Spring Initializr укажите следующие параметры для вашего проекта:

**Project:** Выберите "Maven Project" или "Gradle Project", в зависимости от предпочтений сборки проекта. (В данной работе используется Maven)

**Language:** Выберите язык программирования, на котором хотите разрабатывать приложение (в данной работе Java).

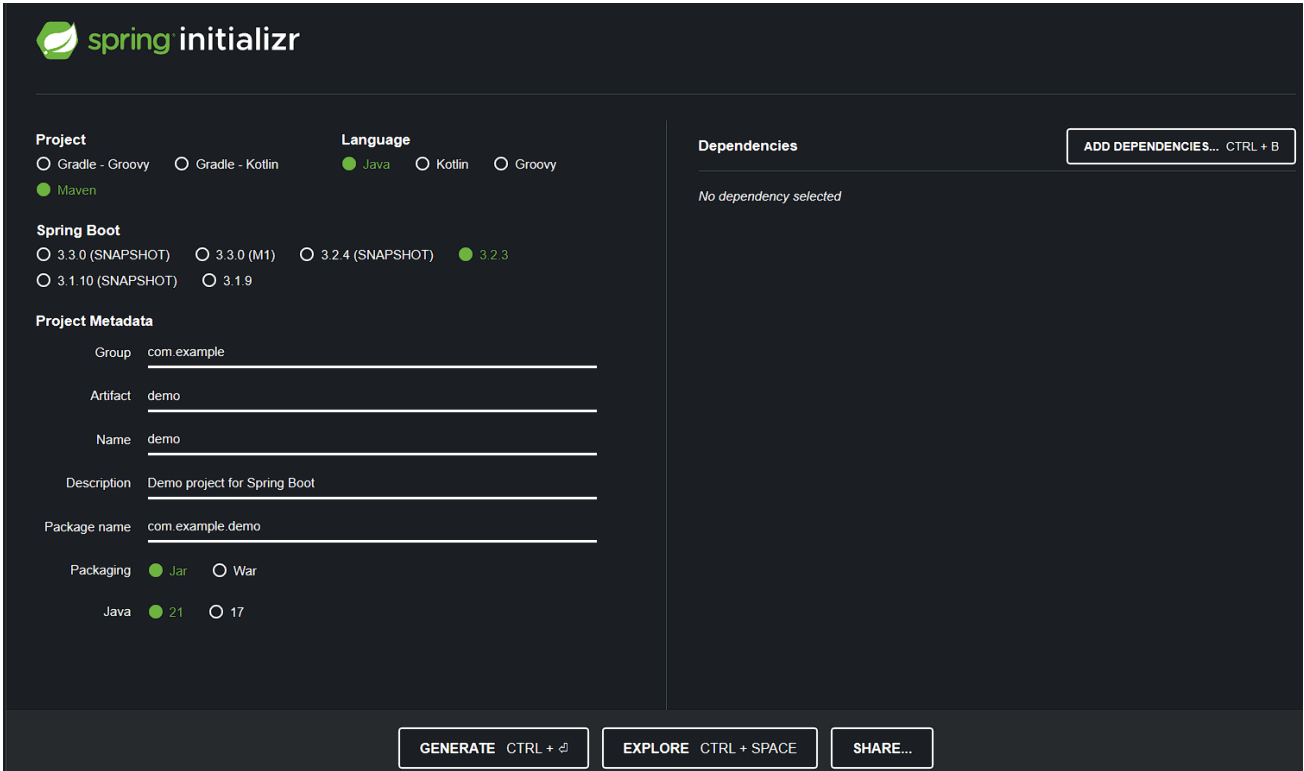
**Spring Boot:** Выберите версию Spring Boot, которую хотите использовать. Рекомендуется выбрать последнюю стабильную версию.

**Шаг 3:** Нажмите кнопку "Generate" для создания проекта. Spring Initializr сгенерирует архив проекта с заданными параметрами.

**Шаг 4:** Распакуйте сгенерированный архив проекта на вашем компьютере.

**Шаг 5:** Импортируйте проект в вашу любимую среду разработки (например, IntelliJ IDEA, Eclipse) как проект Maven или Gradle в зависимости от выбранного типа проекта.

Теперь у вас есть базовый проект на Spring, который готов к разработке.



The screenshot displays the Spring Initializr web interface with a dark theme. The interface is organized into several sections:

- Project:** Includes radio buttons for "Gradle - Groovy", "Gradle - Kotlin", "Java" (selected), "Kotlin", and "Groovy". Below this, "Maven" is selected.
- Spring Boot:** Includes radio buttons for versions "3.3.0 (SNAPSHOT)", "3.3.0 (M1)", "3.2.4 (SNAPSHOT)", "3.2.3" (selected), "3.1.10 (SNAPSHOT)", and "3.1.9".
- Project Metadata:** A form with input fields for "Group" (com.example), "Artifact" (demo), "Name" (demo), "Description" (Demo project for Spring Boot), and "Package name" (com.example.demo).
- Packaging:** Includes radio buttons for "Jar" (selected) and "War".
- Java:** Includes radio buttons for "21" (selected) and "17".
- Dependencies:** A section on the right with the text "No dependency selected" and a button "ADD DEPENDENCIES... CTRL + B".
- Footer:** Three buttons: "GENERATE CTRL + G", "EXPLORE CTRL + SPACE", and "SHARE..."

Рис. 1 – Пример конфигурации проекта

Разберем структуру проекта.

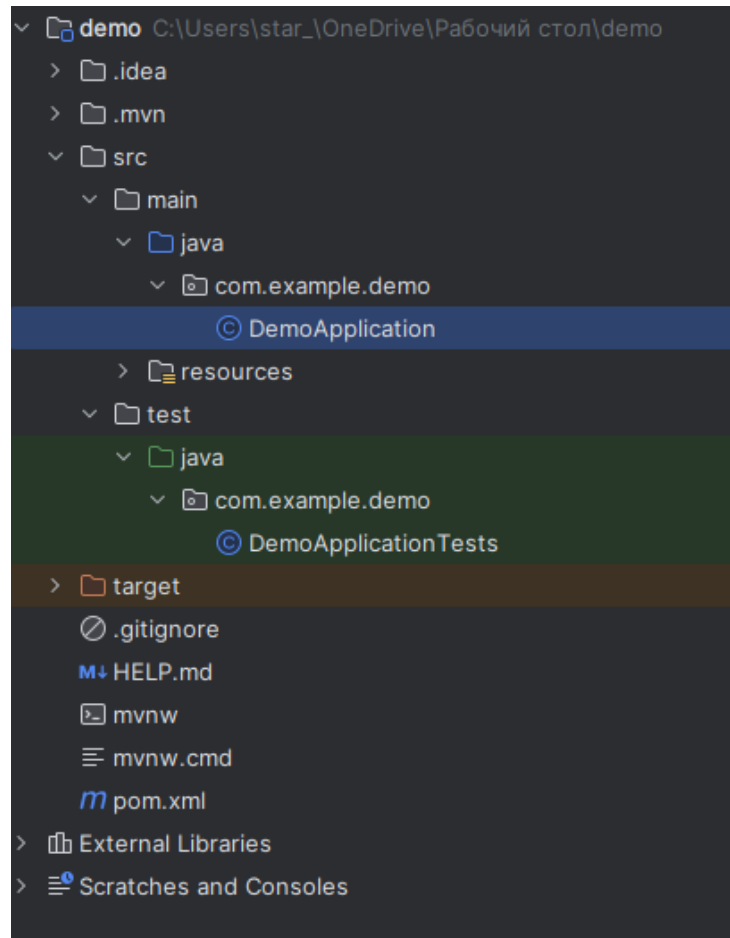


Рис. 2 – структура проекта

src: Это каталог, содержащий все исходные файлы вашего проекта.

main: Этот каталог содержит основной код вашего приложения.

java: Здесь находятся Java-классы вашего приложения.

com.example.demo: Пакет вашего приложения. Название пакета может быть изменено в зависимости от указанного идентификатора группы и артефакта в файлах POM.

resources: Этот каталог содержит ресурсы вашего приложения.

application.properties (или application.yml): Файлы конфигурации Spring Boot, где вы можете настраивать свойства приложения.

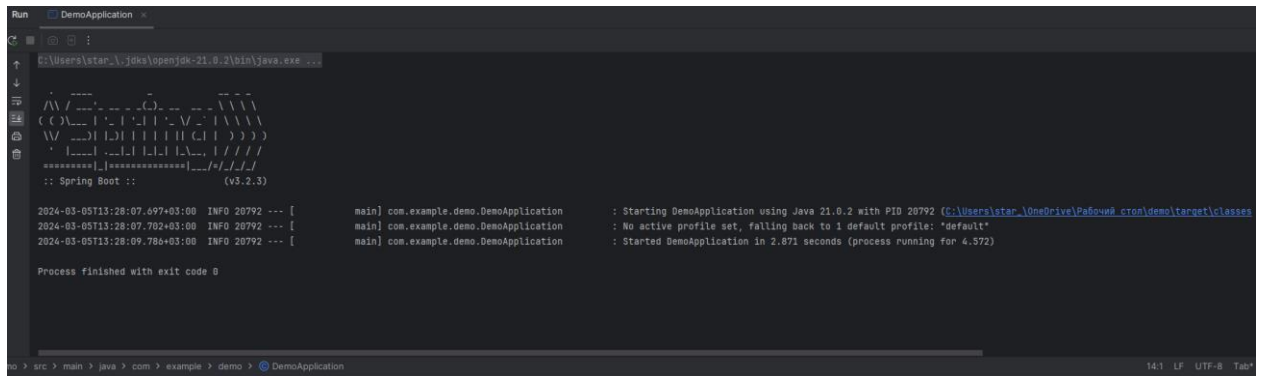
test: Этот каталог содержит тесты вашего приложения.

java: Здесь находятся Java-классы ваших тестов.

com.example.Demo: Пакет ваших тестов.

pom.xml: Это основной файл проекта Maven. Он содержит информацию о зависимостях, плагинах и другие конфигурационные данные проекта.

При запуске данного приложения появляется следующее сообщение:



```
Run DemoApplication
C:\Users\star\Idea\openjdk-21.0.2\bin\java.exe
  ____  _
 / ___|| | | |
| |___| |_| |
 \___ \|  _/
      |_|

:: Spring Boot :: (v3.2.3)

2024-03-05T13:28:07.697+03:00 INFO 20792 --- [main] com.example.demo.DemoApplication : Starting DemoApplication using Java 21.0.2 with PID 20792 (C:\Users\star\Idea\openjdk-21.0.2\bin\java.exe)
2024-03-05T13:28:07.702+03:00 INFO 20792 --- [main] com.example.demo.DemoApplication : No active profile set, falling back to 1 default profile: "default"
2024-03-05T13:28:09.786+03:00 INFO 20792 --- [main] com.example.demo.DemoApplication : Started DemoApplication in 2.871 seconds (process running for 4.572)

Process finished with exit code 0
```

Рис. 3 – программа успешно отработала

В данной работе мы разработаем простое Web-приложение, которое будет выполнять несколько разнообразных запросов.

Класс DemoApplication останется без изменений.

```
package com.example.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class DemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }

}
```

@SpringBootApplication: Это аннотация, которая объединяет несколько других аннотаций в одну. В частности, она включает в себя @Configuration, @EnableAutoConfiguration и @ComponentScan. Эта аннотация говорит Spring Boot автоматически настраивать приложение и искать компоненты в пакетах, начиная с пакета, в котором находится класс с этой аннотацией.

public class DemoApplication: Это основной класс приложения. В нем содержится метод main, который является точкой входа в приложение. Он вызывает статический метод run класса SpringApplication, который запускает Spring приложение.

SpringApplication.run(DemoApplication.class, args): Этот метод запускает Spring приложение. Первым аргументом он принимает класс, который содержит конфигурацию приложения (в данном случае это DemoApplication.class). Вторым аргументом передаются аргументы командной строки (которые передаются в метод main).

Далее нам необходимо дополнить файл `pom.xml`. Как уже было сказано данный файл `pom.xml` в проектах, использующих инструмент управления зависимостями Maven, является файлом конфигурации проекта. Он выполняет следующие функции:

**Управление зависимостями:** В `pom.xml` определяются зависимости проекта. Это позволяет Maven автоматически загружать все необходимые библиотеки (JAR-файлы), которые нужны для компиляции и выполнения проекта, и таким образом упрощает управление внешними зависимостями.

**Конфигурация проекта:** В `pom.xml` вы можете определить различные настройки проекта, такие как версии плагинов Maven, настройки компиляции, фазы жизненного цикла проекта и т. д.

**Сборка проекта:** Maven использует файл `pom.xml` для определения процесса сборки проекта, включая то, какие цели (goals) выполнять, в какой последовательности и с какими параметрами.

**Распространение проекта:** Файл `pom.xml` также может содержать информацию о том, как упаковать проект (например, в виде JAR или WAR-файла), а также метаданные, такие как имя проекта, описание, версия и т. д., которые могут использоваться при распространении проекта.

Сейчас нам необходимо добавить в него новую зависимость:

Для этого в раздел `<dependencies>` нужно вписать дополнительный блок:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Разберем его составляющие:

`<dependency>`: Это элемент, который определяет зависимость проекта от внешней библиотеки.

`<groupId>`: Определяет группу, к которой принадлежит библиотека. В данном случае это `org.springframework.boot`, что означает, что эта библиотека является частью проекта Spring Boot.

`<artifactId>`: Определяет идентификатор артефакта (библиотеки). В данном случае это `spring-boot-starter-web`, что указывает на стартовый набор для разработки веб-приложений с использованием Spring Boot.

Добавление этой зависимости в файл `pom.xml` означает, что проект будет использовать Spring Boot Starter для веб-приложений, что включает в себя все необходимые библиотеки и конфигурации для создания веб-приложения на базе Spring Boot. Полный текст файла `pom.xml` в приложении 1

Теперь мы разработаем web-приложения для хранения заметок. Для этого создадим новый класс с названием `MessageController`.

Сначала добавим в класс необходимые зависимости:

```
package com.example.demo;
import org.springframework.web.bind.annotation.*;
import java.util.ArrayList;
import java.util.List;
```

Каждая из этих зависимостей имеет свою функциональность и необходима для определенных задач в приложении на базе Spring:

```
import org.springframework.web.bind.annotation.*;
```

Эта зависимость предоставляет аннотации, которые используются для создания контроллеров и определения точек входа для веб-запросов. Например, аннотация `@RestController` используется для создания класса контроллера веб-сервиса, а `@RequestMapping` - для определения URL-адреса, по которому будет обрабатываться запрос. Эти аннотации облегчают разработку веб-приложений в Spring Framework, так как они позволяют определить обработчики запросов в декларативном стиле.

```
import java.util.ArrayList;
```

Эта зависимость предоставляет доступ к классу `ArrayList`, который является реализацией списка в Java. `ArrayList` предоставляет динамический массив, который может увеличиваться и уменьшаться по мере необходимости, что делает его удобным для хранения и манипулирования данными в приложениях.

```
import java.util.List;
```

Эта зависимость также предоставляет доступ к классу `List`, который является интерфейсом, определяющим структуру списка. `List` представляет собой упорядоченную коллекцию элементов, которая позволяет хранить дубликаты и обеспечивает доступ к элементам по индексу. В Java существует несколько реализаций интерфейса `List`, например, `ArrayList`, `LinkedList`, `Vector` и др., и выбор конкретной реализации зависит от требований и особенностей приложения.

Далее добавим аннотацию к классу MessageController

```
@RestController
public class MessageController {

    private List<String> userMessages = new ArrayList<>();
```

@RestController:

Это аннотация Spring, которая объявляет класс как контроллер RESTful. Контроллеры RESTful обрабатывают HTTP-запросы и возвращают данные в формате JSON или XML. Они являются основным строительным блоком для создания веб-сервисов и API в приложениях на базе Spring.

```
public class MessageController { ... }:
```

Это определение класса MessageController, который является контроллером веб-приложения. Он содержит методы для обработки HTTP-запросов и обработки бизнес-логики. В данном случае, класс MessageController будет обрабатывать запросы, связанные с сообщениями.

```
private List<String> userMessages = new ArrayList<>();
```

Это приватное поле класса MessageController, которое представляет список (List) строк (String), содержащих сообщения пользователя. В данном случае, userMessages инициализируется новым экземпляром ArrayList, что обеспечивает хранение сообщений пользователя в динамическом массиве. Каждый раз, когда будет создаваться новый экземпляр MessageController, он будет иметь свой собственный список сообщений пользователя.

Добавим обработчики запросов:

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

@GetMapping("/messages")
public List<String> getAllMessages() {
    return userMessages;
}

@PostMapping("/messages")
public String publishMessage(@RequestBody String message) {
    userMessages.add(message);
    return "Message published successfully!";
}

@PutMapping("/messages/{index}")
public String updateMessage(@PathVariable int index, @RequestBody String message) {
    if (index >= 0 && index < userMessages.size()) {
        userMessages.set(index, message);
    }
}
```

```

        return "Message updated successfully!";
    }
    return "Message not found at index " + index;
}

@DeleteMapping("/messages/{index}")
public String deleteMessage(@PathVariable int index) {
    if (index >= 0 && index < userMessages.size()) {
        userMessages.remove(index);
        return "Message deleted successfully!";
    }
    return "Message not found at index " + index;
}
}

```

`@GetMapping("/")` - это аннотация, которая обозначает метод контроллера, который обрабатывает HTTP GET запросы по корневому пути приложения.

`helloWorld()`:

Обработчик GET-запроса по корневому URL (/). Он возвращает строку "Hello, World!". Этот метод будет вызван при обращении к корневому URL вашего приложения.

`getAllMessages()`:

Обработчик GET-запроса по URL /messages. Он возвращает список всех сообщений пользователя (userMessages). Этот метод будет вызван при обращении к URL /messages и вернет список сообщений в формате JSON.

`publishMessage(@RequestBody String message)`:

Обработчик POST-запроса по URL /messages. Он принимает тело запроса в виде строки (message) и добавляет его в список сообщений пользователя (userMessages). Затем возвращает строку "Message published successfully!".

`updateMessage(@PathVariable int index, @RequestBody String message)`:

Обработчик PUT-запроса по URL /messages/{index}, где {index} - это переменная часть URL. Он принимает индекс сообщения и новое сообщение, обновляет соответствующее сообщение в списке userMessages и возвращает сообщение об успешном обновлении.

`deleteMessage(@PathVariable int index)`:

Обработчик DELETE-запроса по URL /messages/{index}, где {index} — это переменная часть URL. Он принимает индекс сообщения, удаляет соответствующее сообщение из списка userMessages и возвращает сообщение об успешном удалении.



Эти методы обрабатывают различные типы HTTP-запросов (GET, POST, PUT, DELETE) и выполняют операции CRUD (Create, Read, Update, Delete) над сообщениями пользователя.

Полный код класса MessageController.java в приложении 2.

Теперь опишем API для взаимодействия с приложением. Мы будем отправлять запросы на сервер через консоль, делать это можно как в самой IDE, так и в powershell.

Получение приветственного сообщения:

```
Invoke-RestMethod -Uri "http://localhost:8080/messages"
```

Получение всех сообщений:

```
Invoke-RestMethod -Uri "http://localhost:8080/messages"
```

Публикация сообщения:

```
$message = "Your message here"
```

```
Invoke-RestMethod -Uri "http://localhost:8080/messages" -Method Post -  
Body $message -ContentType "application/json"
```

Обновление сообщения по индексу:

```
$index = 0 # Замените на желаемый индекс
```

```
$message = "Updated message"
```

```
Invoke-RestMethod -Uri "http://localhost:8080/messages/$index" -Method  
Put -Body $message -ContentType "application/json"
```

Удаление сообщения по индексу:

```
$index = 0 # Замените на желаемый индекс
```

```
Invoke-RestMethod -Uri "http://localhost:8080/messages/$index" -Method  
Delete
```

Запустим получившееся приложение:

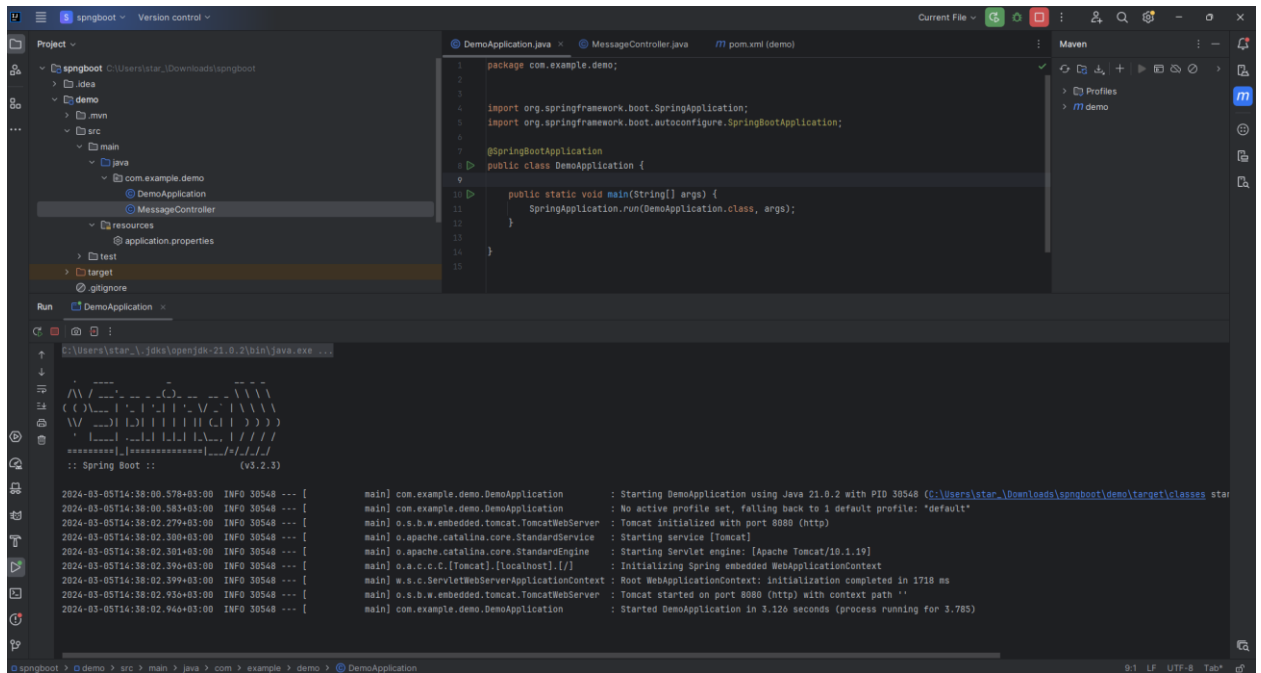


Рис. 4 – сообщение об успешном запуске.

Далее открываем браузер и вводим в адресной строке [localhost:8080](http://localhost:8080).

Получаем сообщение:

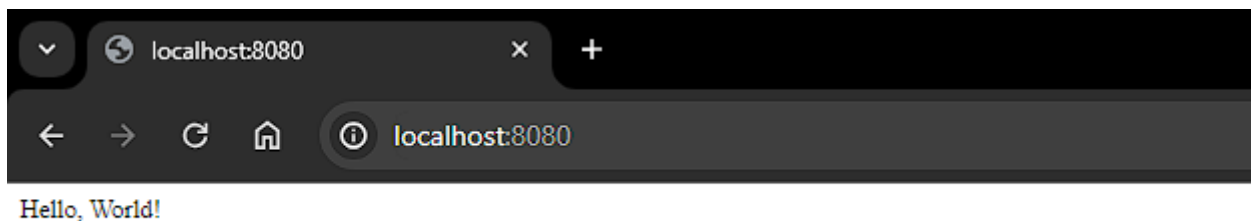


Рис. 5 – сообщение с приветствием.

Далее открываем <http://localhost:8080/messages>

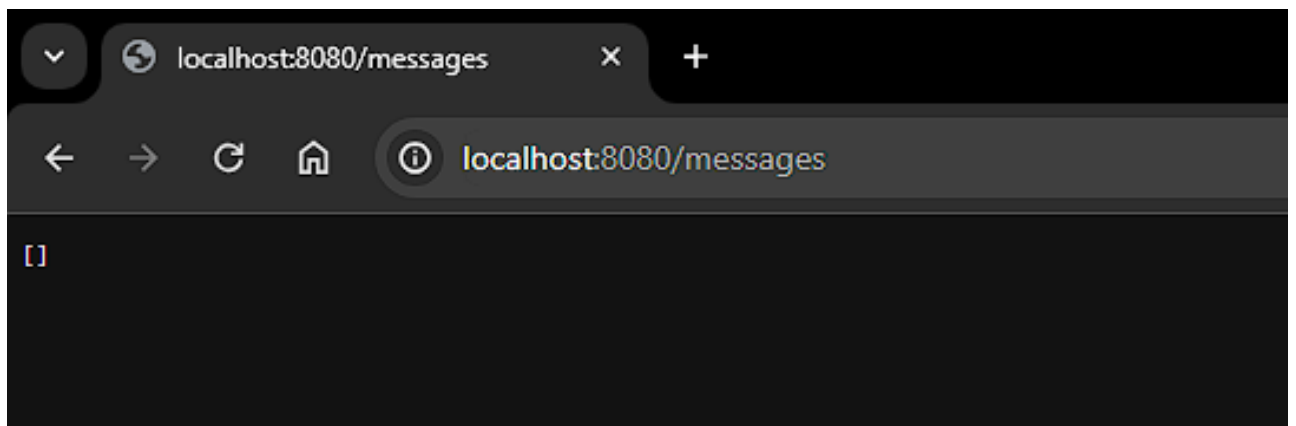
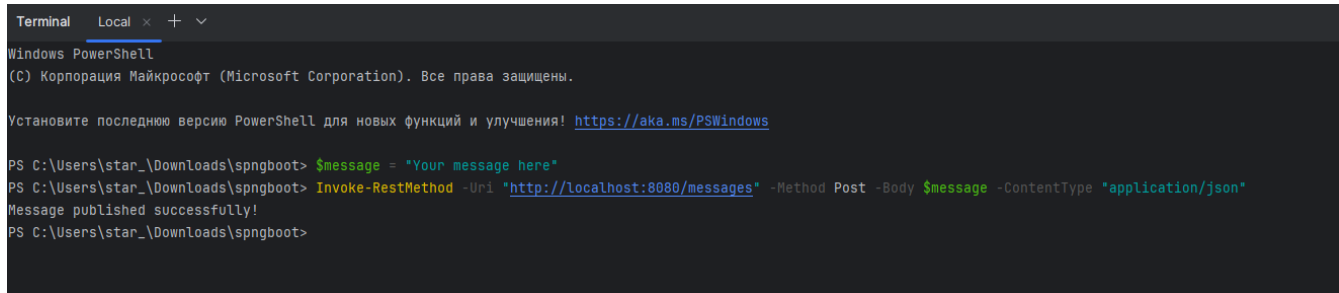


Рис. 6 – Пустой список с сообщениями.

После этого откроем терминал powershell в IDE и вводим следующие запросы:

```
$message = "Your message here"
```

```
Invoke-RestMethod -Uri "http://localhost:8080/messages" -Method Post -  
Body $message -ContentType "application/json"
```



```
Terminal Local x + v  
Windows PowerShell  
(C) Корпорация Майкрософт (Microsoft Corporation). Все права защищены.  
  
Установите последнюю версию PowerShell для новых функций и улучшения! https://aka.ms/PSWindows  
  
PS C:\Users\star_\Downloads\spngboot> $message = "Your message here"  
PS C:\Users\star_\Downloads\spngboot> Invoke-RestMethod -Uri "http://localhost:8080/messages" -Method Post -Body $message -ContentType "application/json"  
Message published successfully!  
PS C:\Users\star_\Downloads\spngboot>
```

Рис. 7 – Сообщение об успешном публикации.

Обновляем в браузере страницу [localhost:8080/messages](http://localhost:8080/messages) и проверяем:

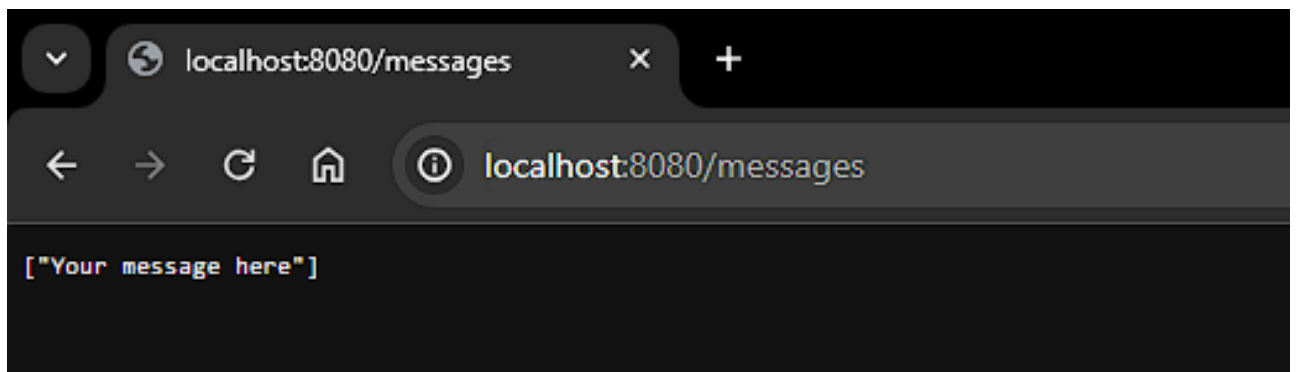


Рис. 8 – Сообщение об успешно опубликовано.

Проверяем запрос на обновление:

```
$index = 0
```

```
$message = "Updated message"
```

```
Invoke-RestMethod -Uri "http://localhost:8080/messages/$index" -Method  
Put -Body $message -ContentType "application/json"
```

Проверяем изменения:

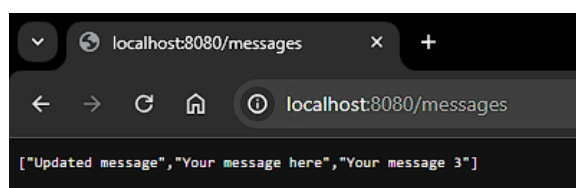


Рис. 9 – Сообщение с индексом 0 успешно изменилось.

**Задание:**

1. Изучить теоретический материал;
2. Написать приложение следуя примеру;
3. Выполнить дополнительное задание;
4. Сделать отчет.

**Дополнительное задание.**

Напишите два дополнительных метода.

**Варианты заданий:**

1. Реализовать метод `clearAllMessages()`, который будет очищать список всех сообщений.
2. Добавить проверку на пустое сообщение при публикации нового сообщения. Если сообщение пустое, вернуть соответствующее сообщение об ошибке.
3. Создать метод `getMessageByIndex(int index)`, который будет возвращать сообщение по указанному индексу.
4. Добавить возможность ограничения количества сообщений, которые могут храниться в списке. Например, если список превысит определенный предел, старые сообщения должны удаляться.
5. Реализовать возможность получения сообщений, опубликованных после определенной даты и времени.
6. Создать метод `countMessages()`, который будет возвращать общее количество сообщений в списке.

## Приложение 1

pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.2.3</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <groupId>com.example</groupId>
  <artifactId>demo</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>demo</name>
  <description>Demo project for Spring Boot</description>
  <properties>
    <java.version>21</java.version>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>
</project>
```

## Приложение 2

### MessageController.java

```
import org.springframework.web.bind.annotation.*;
import java.util.ArrayList;
import java.util.List;

@RestController
public class MessageController {

    private List<String> userMessages = new ArrayList<>();

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

    @GetMapping("/messages")
    public List<String> getAllMessages() {
        return userMessages;
    }

    @PostMapping("/messages")
    public String publishMessage(@RequestBody String message) {
        userMessages.add(message);
        return "Message published successfully!";
    }

    @PutMapping("/messages/{index}")
    public String updateMessage(@PathVariable int index, @RequestBody String
message) {
        if (index >= 0 && index < userMessages.size()) {
            userMessages.set(index, message);
            return "Message updated successfully!";
        }
        return "Message not found at index " + index;
    }

    @DeleteMapping("/messages/{index}")
    public String deleteMessage(@PathVariable int index) {
        if (index >= 0 && index < userMessages.size()) {
            userMessages.remove(index);
            return "Message deleted successfully!";
        }
        return "Message not found at index " + index;
    }
}
```