

Spring Boot

v3.0

Spring Boot

Spring Boot

Spring Boot upraszcza tworzenie aplikacji z wykorzystaniem Springa i dostępnych w jego ramach projektów.

Wymaga bardzo niewielkiej lub zerowej konfiguracji, czyli eliminuje powtarzalne i nudne czynności.

Pozwala także w prosty sposób uruchomić utworzone aplikacje, używając polecenia **java -jar** i zarządzać zależnościami w projekcie.

Strona projektu: <https://projects.spring.io/spring-boot/>

Spring Boot

Tworząc aplikacje z wykorzystaniem **Springa** oraz jego komponentów, zauważamy, że w większości przypadków, z aplikacji na aplikację kopiujemy odpowiednie pliki konfiguracyjne, a kod w nich zawarty większości pozostaje taki sam.

Twórcy **Springa** również to zauważyli i zgodnie z oczekiwaniami developerów powstał projekt **Spring Boot**, którego założeniem jest ograniczenie konfiguracji do minimum.

Spring Boot

Spring Boot wykorzystuje mechanizm skanowania bibliotek projektu i na podstawie wykrycia określonych klas konfiguruje domyślne komponenty.

Oczywiście jeżeli dodamy własną konfigurację nadpisze ona tą określoną przez **Spring Boot**.

Jest to przykład podejścia **konwencja nad konfigurację**.

https://pl.wikipedia.org/wiki/Convention_Over_Configuration

Wymagania

Domyślnie **Spring Boot** wymaga Java 7 oraz Spring w wersji 4.3.10 lub wyższej.

Możliwe jest uruchomienie biblioteki w Javie 6 po przeprowadzeniu dodatkowej konfiguracji.

Polecana przez twórców Spring Boot wersją jest oczywiście Java 8.

Spring Boot zapewnia wsparcie dla Apache Maven w wersji 3.2 lub wyższej i Gradle 2.9 lub wyższej oraz wersji 3.

Instalacja i uruchomienie

Spring Boot może być używany jak zwykła biblioteka Javy.

Wystarczy dodać odpowiednie pliki **spring-boot-*.jar** do projektu.

Spring Boot nie wymaga żadnych dedykowanych narzędzi programistycznych.

Może być używany w dowolnym **IDE** czy też w edytorze tekstu, a programy napisane przy użyciu **Spring Boot** niczym się nie wyróżniają spośród innych programów napisanych w Javie.

Choć można używać **Spring Boot**, kopiując wprost odpowiednie biblioteki jar, wygodniejszym i jednocześnie zalecanym sposobem jest użycie **Apache Maven** lub **Gradle**.

Tworzenie projektu - Maven

Powszechną metodą tworzenia projektu jest bezpośrednio użycie **Mavena**.

Typowy plik **pom.xml** projektu dziedziczy po starterze **spring-boot-starter-parent**, dzięki czemu wpisując pozostałe zależności projektu, można pominąć sekcję **<version>**.

Wersje dodawanych bibliotek zależnych będą dobierane przez projekt **spring-boot-starter-parent** w taki sposób, aby były ze sobą kompatybilne.

Przykłady przedstawione w niniejszym dziale są oparte o wersję **1.5.9**.

Tworzenie projektu - Maven

Tworzymy projekt **Maven** za pomocą **IDE**.

Plik **pom.xml** uzupełniamy o następujące wpisy:

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.5.9.RELEASE</version>
</parent>
```

Dziedziczenie po **spring-boot-starter-parent** powoduje, że projekt jest konfigurowany przez pewne parametry domyślne, m.in.:

- domyślna wersja kompilatora Java 1.6
- kodowanie znaków UTF-8
- mechanizm zarządzania zależnościami pozwalający na pomijanie sekcji <version> w dodawanych zależnościach.

Tworzenie projektu - Maven

Domyślną wersję kompilatora można zmienić poprzez dodanie do pom.xml właściwości **<java.version>**

```
<properties>  
    <java.version>1.8</java.version>  
</properties>
```

Tworzenie projektu - Maven

Dodajemy również opcjonalny plugin **spring-boot-maven-plugin**, dzięki któremu będzie można wygenerować wykonywalny plik **.jar** z gotowym programem.

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

Tworzenie projektu - Maven - zależności

Startery w rozumieniu **Spring Boot** są to wygodne zależności, które można dołączyć do projektu.

Pojedynczy starter może zawierać zestaw kilku zależności, dzięki czemu programista może szybko dodać pewną funkcjonalność.

Dodajemy do projektu zależność odpowiedzialną za obsługę **Spring MVC**:

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

Startery

Przykłady popularnych starterów:

- **spring-boot-starter-data-jpa** – obsługa JPA
- **spring-boot-starter-freemarker** - system szablonów
- **spring-boot-starter-thymeleaf** – system szablonów
- **spring-boot-starter-web** – Spring MVC
- **spring-boot-starter-validation** – wsparcie dla walidacji
- **spring-boot-starter-test** – wsparcie dla testowania
- **spring-boot-devtools** – monitoruje aplikację i po zmianach automatycznie ją przeładowuje

Używanie Spring Boot

Spring Boot nie wymaga specjalnego układu kodu, zalecane jest jednak trzymanie się pewnych ogólnie przyjętych praktyk.

Po pierwsze klasy, które nie zawierają deklaracji pakietu (package), są traktowane, jakby znajdowały się w „**pakiecie domyślnym**”.

Powinniśmy unikać takich sytuacji. Może to powodować problemy podczas stosowania adnotacji Spring Boot takich jak **@ComponentScan** czy **@SpringBootApplication**.

Przyjęło się, żeby nazwy pakietów zaczynały się od odwróconej nazwy domeny, a następnie nazwa aplikacji, w naszym wypadku **pl.coderslab.projectname**.

Używanie Spring Boot

Zaleca się, aby główny plik aplikacji powinien umieszczać w pakiecie głównym projektu, ponad pozostałymi klasami.

Główna klasa ma często adnotację **@EnableAutoconfiguration**, która niejawnie ustawia bazową ścieżkę wyszukiwania dla poszczególnych elementów aplikacji (np. klas encji **@Entity**, gdy aplikacja korzysta z JPA).

Mechanizm autokonfiguracji próbuje automatycznie skonfigurować aplikację na podstawie bibliotek dołączonych do projektu.

Struktura projektu

Struktura projektu może wyglądać następująco:

```
pl
+- coderslab
  +- projectname
    +- Application.java
    |
    +- domain
    |   +- User.java
    |   +- UserAddress.java
    |
    +- service
    |   +- UserService.java
    |
    +- web
    |   +- UserController.java
```


Klasa Startowa

Uzupełniamy projekt o klasę startową dla naszej aplikacji:

```
package pl.coderslab;

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

@SpringBootApplication
public class Application {
    public static void main(String[] args) throws Exception {
        SpringApplication.run(Application.class, args);
    }
}
```

Adnotacja @SpringBootApplication

Adnotacja **@SpringBootApplication** grupuje kilka innych poznanych przez nas adnotacji m.in. :

- **@ComponentScan**,
- **@Configuration**,
- **@EnableAutoConfiguration** - informuje **Spring Boot** by dodawał ziarna konfiguracyjne na podstawie elementów naszej aplikacji.

Przykładowo wykrywa, że mamy zależności do **spring-webmvc** i na tej podstawie konfiguruje np. **DispatcherServlet** - oszczędzając nam pracy.

Testowa akcja

Dodajemy przykładowy kontroler oraz akcje:

```
package pl.coderslab.web;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class HomeController {

    @RequestMapping("/")
    @ResponseBody
    public String home() {
        return "hello world !!!";
    }
}
```

Instalacja i uruchomienie

Tak przygotowaną aplikację możemy już uruchomić za pomocą **Mavena** przy pomocy konsolowej komendy:

```
mvn spring-boot:run
```

Możemy również przy pomocy **IDE** wybierając z menu kontekstowego **Run As** następnie **Spring Boot App** lub **Java Application**.

Wywołując polecenie **Mavena** :

mvn package - wygenerujemy w katalogu **target** naszej aplikacji plik wykonywalny **jar**.

Plik ten możemy uruchomić za pomocą polecenia: **java -jar nazwa_pliku.jar**

np.: **java -jar boot-maven-0.0.1-SNAPSHOT.jar**

Wynik naszej pracy możemy zobaczyć w przeglądarce po wejściu na adres:

<http://localhost:8080/>

Tworzenie projektu - Spring Initializr

Tworząc projekt z wykorzystaniem **Spring Boot** możemy również użyć serwisu internetowego **Spring Initializr**, który umożliwia określenie elementów, z jakich ma się składać aplikacja.

Na stronie projektu należy wpisać metadane tworzonego projektu jak np nazwa, grupa, artefakt oraz wybrać składniki (zależności i startery), które mają być dołączone do projektu, a wygenerowany projekt można pobrać w postaci archiwum zip.

Tak pobrany plik po rozpakowaniu, możemy następnie otworzyć przy pomocy dowolnego IDE.

Spring Initializr dostępny jest pod adresem: <http://start.spring.io/>

Spring Boot

Alternatywnym sposobem na utworzenie takiego projektu jest wykorzystanie wsparcia, jakie udostępnia nasze IDE – na przykładzie **Intellij**.

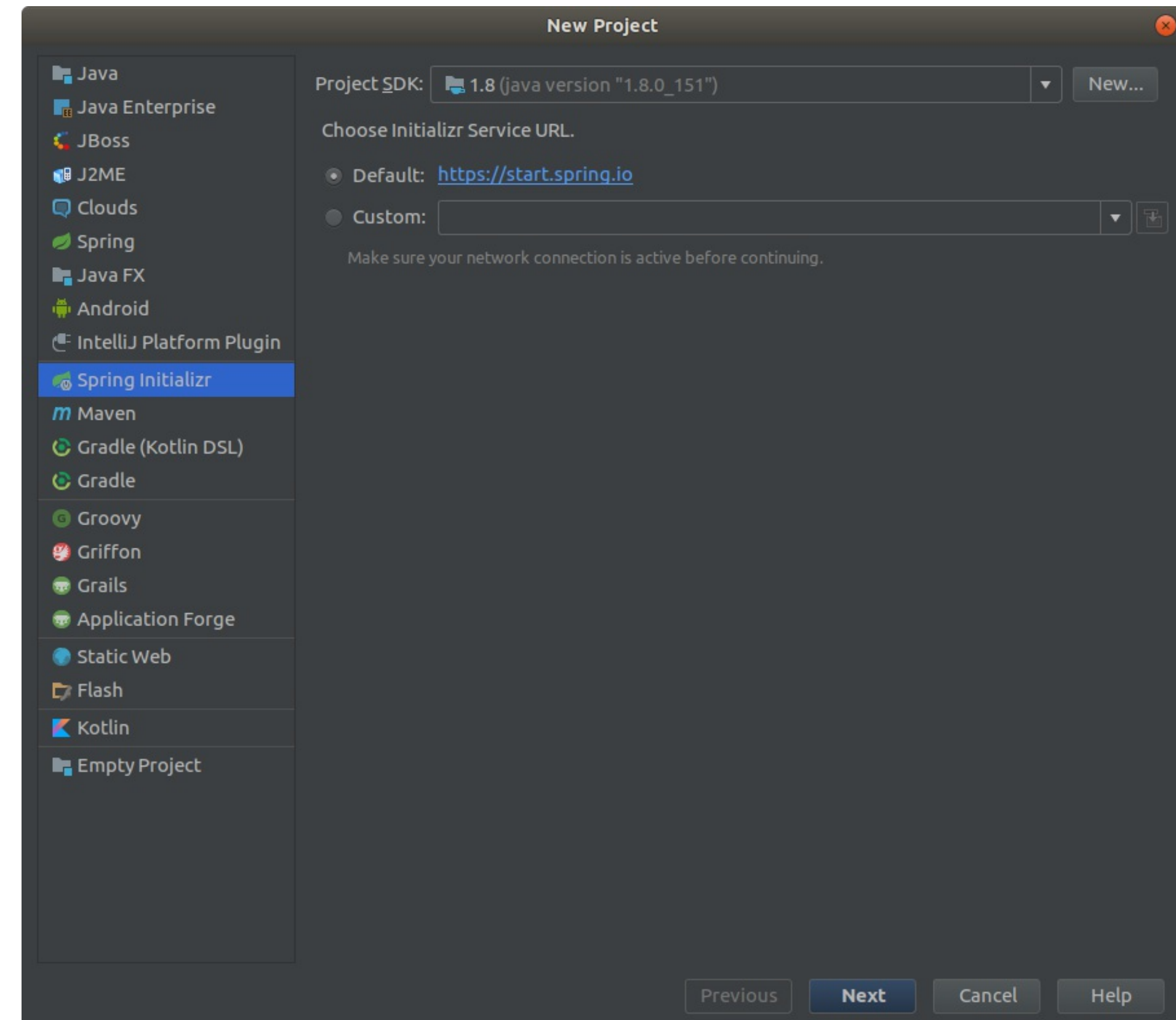
Z menu górnego wybieramy:

File → New → Project...

a następnie zaznaczamy opcję:

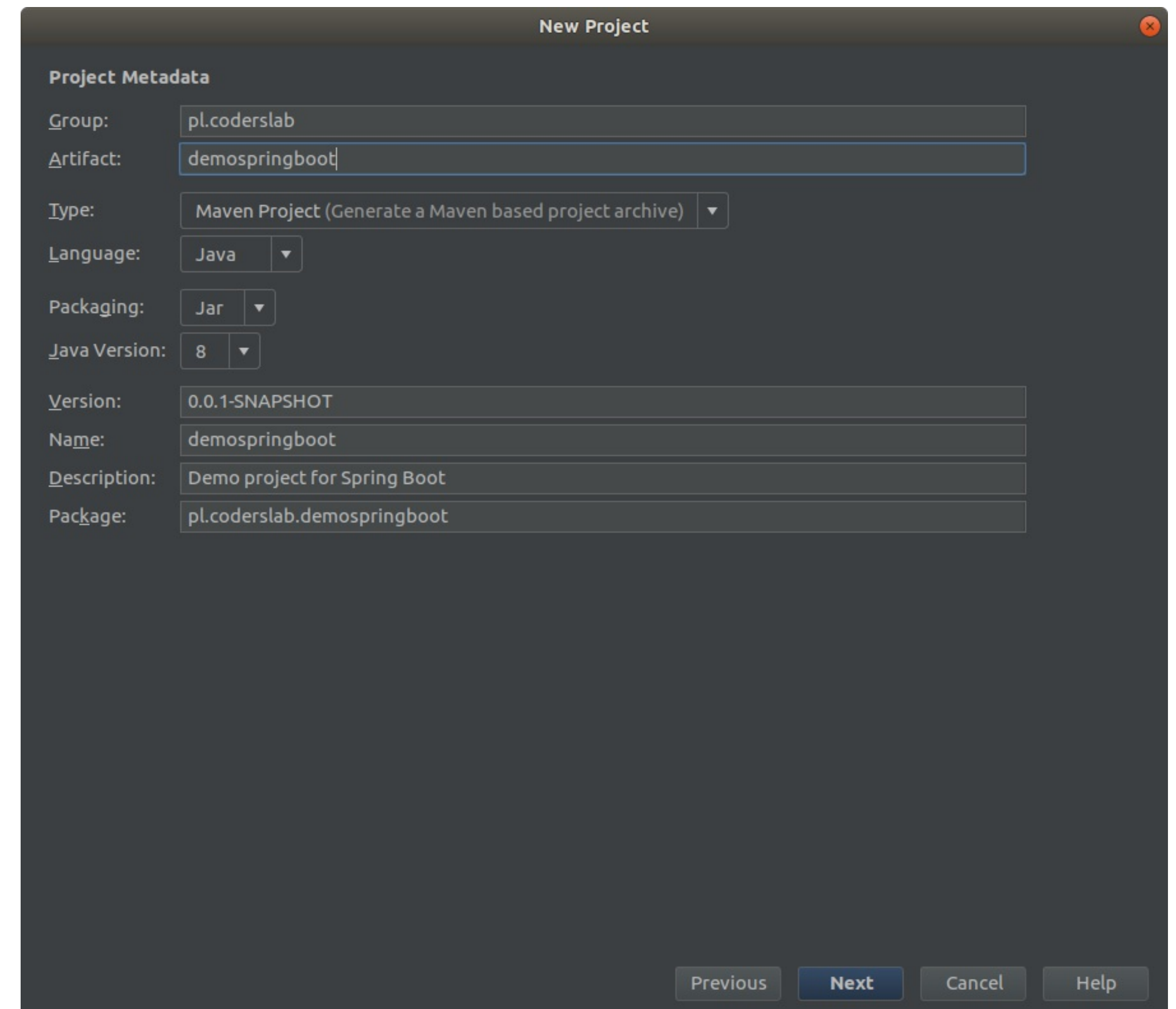
Spring Initializr.

Klikamy **Next**.



Spring Boot

Wypełniamy podstawowe dane – analogicznie jak w przypadku projektu **Maven**.



The screenshot shows the 'New Project' dialog box with the following fields and values:

Project Metadata	
Group:	pl.coderslab
Artifact:	demospringboot
Type:	Maven Project (Generate a Maven based project archive) ▼
Language:	Java ▼
Packaging:	Jar ▼
Java Version:	8 ▼
Version:	0.0.1-SNAPSHOT
Name:	demospringboot
Description:	Demo project for Spring Boot
Package:	pl.coderslab.demospringboot

At the bottom right, there are four buttons: 'Previous', 'Next' (highlighted in blue), 'Cancel', and 'Help'.

Spring Boot

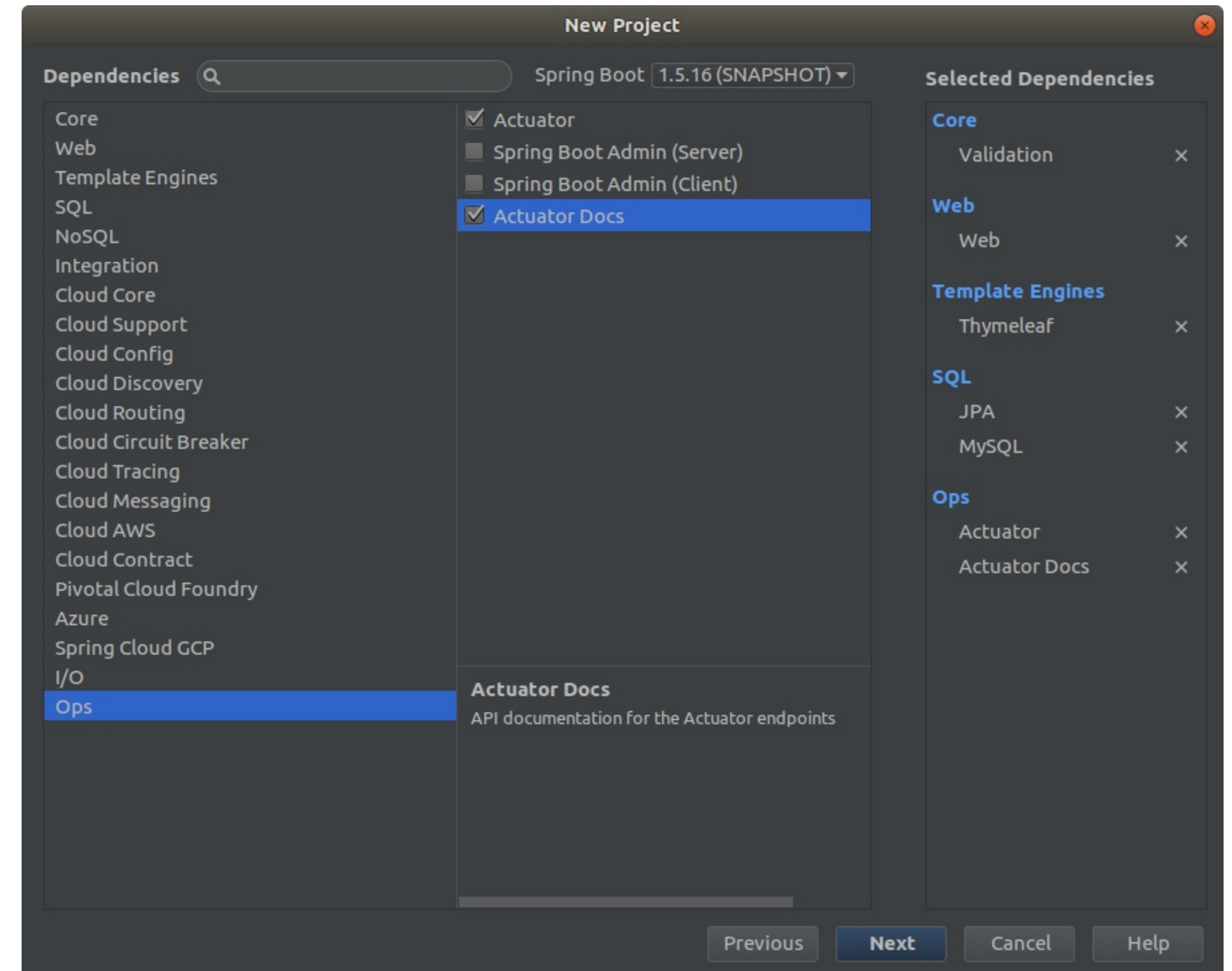
W następnym kroku określamy, jakie startery chcemy umieścić w naszej aplikacji.

W menu po lewej są moduły, które zawierają po kilka opcji. Lista wszystkich wybranych zależności znajduje się po prawej stronie.

Na niebiesko są wypisane moduły, a pod nimi startery, które trzeba zaznaczyć przy tworzeniu nowej aplikacji.

Wybierz je analogicznie w swoim projekcie.

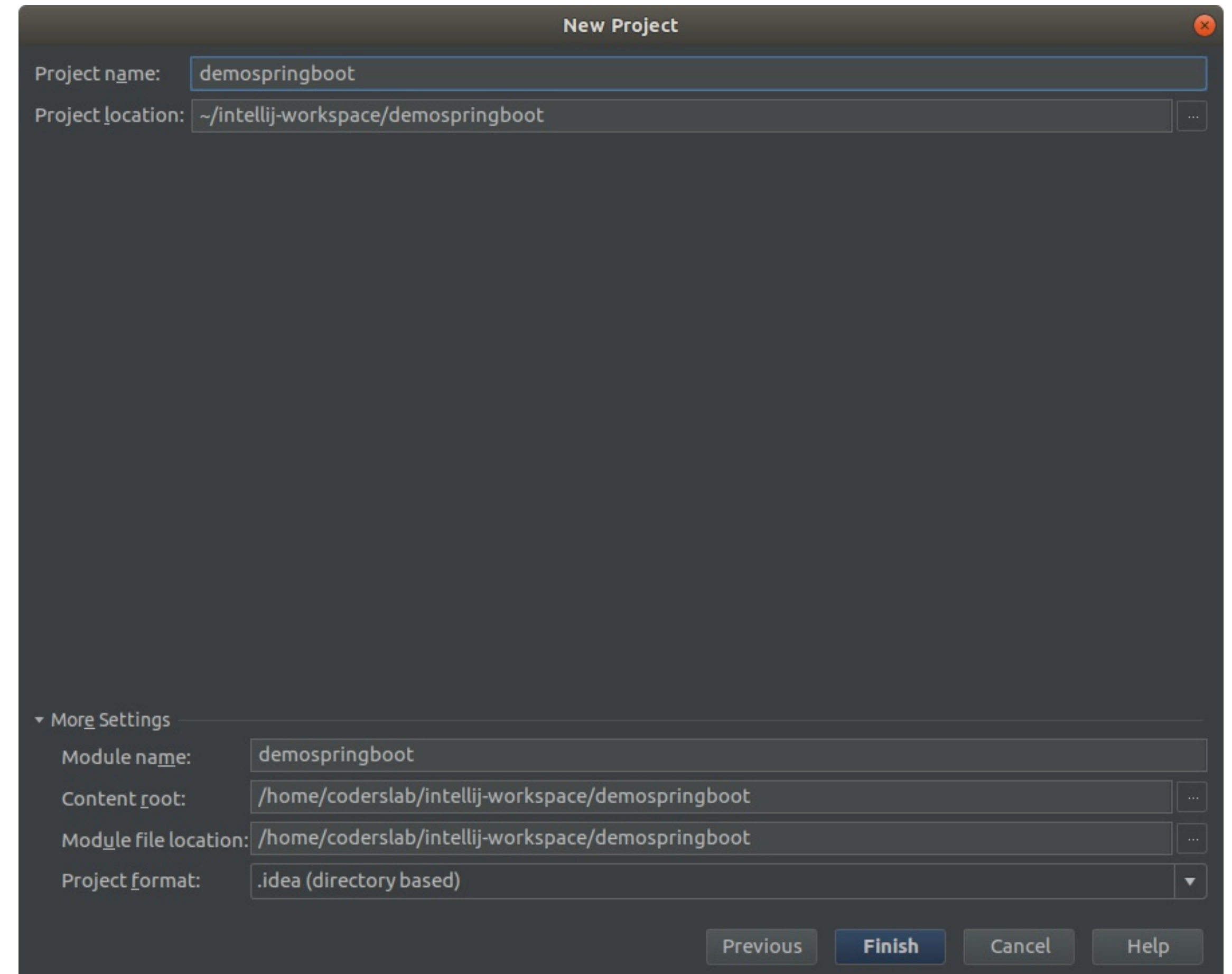
Wybierz także wersję **Spring Boota**, która zaczyna się od cyfry 1.



Spring Boot

Pojawi się okno końcowe, w którym klikamy opcję **Finish**.

Tworzy się nowy projekt już uzupełniony w potrzebne zależności i skonfigurowany odpowiednio do tworzenia aplikacji w Springu.



Spring Boot

Jako przykład skonfigurujemy aplikację zawierającą:

- **Web** - czyli wsparcie dla Spring MVC
- **Thymeleaf** - system szablonów
- **Validation** - wsparcie dla walidacji
- **JPA** - czyli wsparcie dla JPA
- **MySQL** - czyli wsparcie dla bazy danych
- **Actuator** - czyli narzędzie do monitorowania stanu aplikacji

Struktura wygenerowanej aplikacji

Struktura katalogów:

- **src/main/java** - tutaj umieścimy nasze pakiety oraz klasy Java, zawiera automatycznie utworzoną klasę startową o nazwie **Application**),
- **src/test/java** tutaj umieścimy testy naszej aplikacji,
- **src/main/resources** to miejsce na zasoby; podkatalog static ma zawierać pliki przetwarzane po stronie klienta (obrazki, JavaScript), a podkatalog templates szablony przetwarzane po stronie serwera,

Spring Boot - uruchomienie

Ponieważ zdefiniowaliśmy, że nasza aplikacja ma używać **MySQL**, przed uruchomieniem aplikacji **Spring Boot** wymaga od nas zdefiniowania danych dostępowych do bazy.

Dane konfiguracyjne aplikacji musimy umieścić w pliku **application.properties**.

Na konsoli otrzymamy komunikat o błędzie:

```
*****  
APPLICATION FAILED TO START  
*****
```

Description:

```
Cannot determine embedded database driver class for  
database type NONE
```

Ustawienia aplikacji

application.properties - jest to plik, który w naszej wygenerowanej aplikacji znajduje się w lokalizacji **src/main/resources** w pliku tym umieszczamy wpisy w postaci:

klucz = wartość

Przykładowe ustawienia danych dostępowych do bazy danych:

```
spring.jpa.hibernate.ddl-auto=create  
spring.datasource.url=jdbc:mysql://localhost:3306/db_example  
spring.datasource.username=springuser  
spring.datasource.password=ThePassword
```

Ustawienia aplikacji

Listę najbardziej popularnych ustawień znajdziemy pod adresem: <https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html>

Podpowiedzi udzieli nam również **IDE** tak jak w przypadku plików **javvy** przy użyciu kombinacji klawiszy **Ctrl+ spacja**.

Spring Boot - JSP

W przypadku gdy jako warstwy widoku chcemy używać plików **jsp** dodajemy do pliku **pom.xml** następujące zależności:

```
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>jstl</artifactId>
</dependency>
<dependency>
  <groupId>org.apache.tomcat.embed</groupId>
  <artifactId>tomcat-embed-jasper</artifactId>
</dependency>
```

Jeżeli tworząc projekt za pomocą **IDE** lub **Initializera**, wybraliśmy opcję **Thymeleaf** należy odpowiedzialną za niego zależność usunąć z pliku.

Spring Boot - JSP

W pliku **application.properties** dodajemy wpisy odpowiedzialne za konfigurację **ViewResolver**:

```
spring.mvc.view.prefix=/WEB-INF/views/  
spring.mvc.view.suffix=.jsp
```

Jeżeli nie posiadamy w naszej aplikacji pliku **application.properties** należy go utworzyć w lokalizacji: **/src/main/resources**.

Dla powyższej konfiguracji pliki **jsp** umieszczamy w **/src/main/webapp/WEB-INF/views**.

Dlaczego dopiero teraz ?

Wiele osób może sobie zadać pytanie, skoro mogę w łatwy sposób wygenerować podstawowy szablon aplikacji, w którym od samego początku można zacząć pisać kod realizujący logikę biznesową - dlaczego poznajemy go tak późno.

Bez znajomości ręcznej konfiguracji nie bylibyśmy w stanie stwierdzić, gdzie może leżeć błąd w przypadku jego wystąpienia oraz jakie komponenty mogą być odpowiedzialne za zmiany, które chcemy wprowadzić.

Spring Actuator

Spring Actuator

Actuator daje nam możliwość uzyskania w czasie rzeczywistym przydatnych informacji na temat działania naszej aplikacji.

Aby skorzystać z jego możliwości, dodajemy do pliku **pom.xml** następujący starter:

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

Możemy go również wskazać podczas tworzenia projektu **Spring Boot** przy użyciu **Initializer** lub możliwości naszego **IDE**.

Spring Actuator

Actuator udostępnia adresy (**endpointy**) wyświetlające określone informacje.

Endpointy możemy podzielić na publiczne i prywatne (dostępne tylko po zalogowaniu).

Co jest całkowicie zrozumiałe ze względu na bezpieczeństwo naszej aplikacji.

Endpointy publiczne:

- **/health** - zwraca proste informacje na temat statusu naszej aplikacji
- **/info** - wyświetla dowolne informacje o aplikacji.

Spring Actuator

Istnieje możliwość upublicznienia endpointów prywatnych - w tym celu musimy w pliku **application.properties** dodać następujący wpis:

```
management.security.enabled=false
```

W kolejnym module dowiemy się, w jaki sposób zabezpieczyć dostęp do naszej aplikacji przy pomocy **Spring Security** - pamiętajmy wtedy o zmianie tego ustawienia.

Nie należy udostępniać publicznie aplikacji z dostępem do wszystkich endpointów Actuatora.

Za pomocą ustawień w **application.properties** możemy również określić prefiks dla wszystkich endpointów, służy do tego wpis:

```
management.context-path=/appinfo
```

Endpoint /info

Adres ten zwraca dowolne informacje o naszej aplikacji, które przy pomocy odpowiednich wpisów dodajemy do pliku **application.properties**

Przykładowe wpisy:

```
info.app.description=Created with love  
info.app.java.source=@java.version@  
info.app.version = @version@  
info.app.name=Coderslab.pl Example Actuator App
```

Endpoint /info

Adres ten zwraca dowolne informacje o naszej aplikacji, które przy pomocy odpowiednich wpisów dodajemy do pliku **application.properties**

Przykładowe wpisy:

```
info.app.description=Created with love  
info.app.java.source=@java.version@  
info.app.version = @version@  
info.app.name=Coderslab.pl Example Actuator App
```

@java.version@ - jest to wartość pobierana z właściwości określonych w pliku **pom.xml** - tag **<java.version>** zawarty w tagu: **<properties>**.

Endpoint /info

Adres ten zwraca dowolne informacje o naszej aplikacji, które przy pomocy odpowiednich wpisów dodajemy do pliku **application.properties**

Przykładowe wpisy:

```
info.app.description=Created with love  
info.app.java.source=@java.version@  
info.app.version = @version@  
info.app.name=Coderslab.pl Example Actuator App
```

@java.version@ - jest to wartość pobierana z właściwości określonych w pliku **pom.xml** - tag **<java.version>** zawarty w tagu: **<properties>**.

@version@ - jest to wartość pobierana z właściwości określonych w pliku **pom.xml** - tag **<version>**.

Endpoint /info

W efekcie wywołania adresu: <http://localhost:8080/info>
otrzymamy:

```
{
  app: {
    version: "0.0.1-SNAPSHOT",
    description: "Created with love",
    java: {
      source: "1.8.0_151"
    },
    name: "Coderslab.pl Example Actuator App"
  }
}
```

Endpoint /health

Endpoint **/health** zwraca informacje na temat statusu naszej aplikacji. W efekcie wywołania adresu: <http://localhost:8080/info> otrzymamy informacje zbliżone do poniższych:

```
{
  status: "UP",
  diskSpace: {
    status: "UP",
    total: 237607321600,
    free: 7443353600,
    threshold: 10485760
  },
  db: {
    status: "UP",
    database: "MySQL",
    hello: 1
  }
}
```

Spring Actuator Docs

Łatwym sposobem na poznanie wszystkich endpointów, jakie są dostępne, jest dodanie do projektu dodatkowego startera, który udostępni nam dokumentację pod domyślnym adresem:

<http://localhost:8080/docs/>

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-actuator-docs</artifactId>
</dependency>
```

Możemy go również wskazać podczas tworzenia projektu **Spring Boot** przy użyciu **Initializer** lub możliwości naszego **IDE**.

Przydatne endpointy

Pozostałe endpointy przydatne do analizy stanu naszej aplikacji to:

- **/mappings** - udostępnia informacje o wszystkich adresach url, jakie udostępnia nasza aplikacja
- **/trace** - udostępnia informacje o ostatnich adresach url, jakie zostały wywołane wraz z nagłówkami - domyślnie dostępnych jest 100 ostatnich wywołań
- **/metrics** - udostępnia informacje o ostatnich adresach url, jakie zostały wywołane wraz z nagłówkami - domyślnie dostępnych jest 100 ostatnich wywołań
- **/beans** - wyświetla wszystkie beany naszej aplikacji

Listę wszystkich dostępnych endpointów znajdziemy pod adresem

<https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-endpoints.html#production-ready-endpoints>

Thymeleaf

Thymeleaf

Thymeleaf - jest to alternatywa do JSP, w której pliki szablonu są tworzone przy pomocy zwykłych plików html.

Dzięki temu że szablony są zwykłymi plikami html - możemy je bez problemu otwierać za pomocą przeglądarki internetowej.

Informacje do wyświetlenia są umieszczane w dokumencie html za pomocą dodatkowych atrybutów a nie zawartości tagów.

Mają automatyczną konfigurację dla aplikacji opartych o **Spring Boot**

Dostępne jest również wsparcie dla **Spring Security**.

Pierwszy szablon

Widoki umieszczamy w katalogu **/src/main/resources/templates**.

Przykładowy widok:

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Getting Started: Serving Web Content</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
</head>
<body>
<p th:text="'Hello Thymeleaf'" />
</body>
</html>
```

Pierwszy szablon

Widoki umieszczamy w katalogu `/src/main/resources/templates`.

Przykładowy widok:

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Getting Started: Serving Web Content</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
</head>
<body>
<p th:text="'Hello Thymeleaf'" />
</body>
</html>
```

Jest to definicja przestrzeni nazw, nasze atrybuty będą zaczynać się od **th** .

Pierwszy szablon

Widoki umieszczamy w katalogu `/src/main/resources/templates`.

Przykładowy widok:

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Getting Started: Serving Web Content</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
</head>
<body>
<p th:text="'Hello Thymeleaf'" />
</body>
</html>
```

Tag tekstowy - dzięki niemu w ramach akapitu `<p>` zostanie umieszczona zawartość **Hello Thymeleaf**.

Dokument html

Domyślna konfiguracja sprawia że pliki html których używamy jako widoków naszej aplikacji muszą być składniowo poprawne, tzn. że wszystkie tagi muszą być poprawnie domknięte.

W przypadku umieszczenia w aplikacji kodu:

```
<link href="../../vendor/bootstrap/css/bootstrap.min.css" rel="stylesheet">
```

Otrzymamy wyjątek:

```
org.thymeleaf.exceptions.TemplateInputException: Exception parsing document:  
    template="layout", line 38 - column 3 (index:4)] with root cause  
org.xml.sax.SAXParseException:  
    The element type "link" must be terminated by the matching end-tag "</link>".
```

W przypadku gdy korzystamy z gotowych szablonów html, wymaga to od nas poprawy wszystkich niedomkniętych tagów.

Dokument html

Możemy zmienić ustawienia przetwarzania szablonów, tak by dopuszczalne były nie domknięte tagi, w tym celu dodajemy do pliku **pom.xml**:

```
<dependency>
  <groupId>net.sourceforge.nekohtml</groupId>
  <artifactId>nekohtml</artifactId>
</dependency>
```

a następnie uzupełniamy w pliku **application.properties** poniższy wpis:

```
spring.thymeleaf.mode=LEGACYHTML5
```

Przekazywanie danych

Przekazywanie danych z kontrolera niczym nie różni się od poznanego przez nas wcześniej sposobu z wykorzystaniem obiektu klasy **Model** oraz widoku **jsp**.

```
@GetMapping("/hello")  
public String hello(Model model) {  
    model.addAttribute("message", "Hello Coderslab");  
    return "hello";  
}
```

Przekazywanie danych

Przekazywanie danych z kontrolera niczym nie różni się od poznanego przez nas wcześniej sposobu z wykorzystaniem obiektu klasy **Model** oraz widoku **jsp**.

```
@GetMapping("/hello")  
public String hello(Model model) {  
    model.addAttribute("message", "Hello Coderslab");  
    return "hello";  
}
```

Pod kluczem **message** ustawiamy wartość **Hello Coderslab**.

Odbieranie danych

Dane tekstowe przekazane przez **Model** do widoku **html** możemy pobrać i wyświetlić przy pomocy atrybutu **th:text** w dowolnym tagu dokumentu html, np:

```
<h1 th:text="${message}" ></h1>  
<p th:text="${message}" />  
<span th:text="${message}" ></span>
```

Wygenerowany w przeglądarce kod będzie wyglądał następująco:

```
<h1>Hello Coderslab</h1>  
<p>Hello Coderslab</p>  
<span>Hello Coderslab</span>
```

th:text

Napisy możemy łączyć z wartościami pobranymi z modelu za pomocą znaku **+** :

```
<p th:text="'Witaj ' + ${person.firstName}">Tekst Powitania</p>
```

Zwróć uwagę na pojedyncze apostrofy, w których umieszczony jest napis **Witaj**

alternatywnie możemy użyć znaku **|** tzw pipe, w ramach którego umieszczamy elementy przeznaczone do połączenia, np:

```
<p th:text="|Witaj z pipe  ${person.firstName}|">Tekst Powitania</p>
```


th:text

Za pomocą tagu **th:text** możemy również umieszczać w naszym dokumencie informacje pochodzące z plików tłumaczeniowych.

Do oznaczenia, że napis ma zostać pobrany z odpowiedniego pliku tłumaczeniowego, służy znak **#**

```
<p th:text="#{app.name}">Nazwa aplikacji</p>
```

Klucze do tłumaczeń umieszczamy w pliku o nazwie **messages_pl.properties** znajdującym się w katalogu:

/src/main/resources.

Wpisy te są postaci **klucz = wartość**, np.:

```
app.name=Aplikacja Demo
```

.

th:text

Dla naszej aplikacji musimy jeszcze nadać odpowiednie ustawienia lokalizacyjne, za pomocą pliku **application.properties** możemy nadać niezmiennie ustawienie, dodając wpisy:

```
spring.mvc.locale=pl_PL  
spring.mvc.locale-resolver=fixed
```

Inną dostępną do ustawienia wartością z poziomu pliku jest opcja:

```
spring.mvc.locale-resolver=accept-header
```

ustawienia lokalizacyjne zostaną pobrane z nagłówka http.

Parametry internacjonalizacji

Do naszych wpisów tłumaczeniowych możemy również wstawiać zmienne, zobrazujemy to za pomocą przykładu. W tym celu posłużymy się prostą klasą **Person**:

```
package pl.coderslab.model;
public class Person {

    private String firstName;
    public Person(String firstName) {
        this.firstName = firstName;
    }
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
}
```

Parametry internacjonalizacji

Przekazujemy z kontrolera nowy obiekt klasy **Person**

```
@GetMapping("/hello")
public String hello(Model model) {
    model.addAttribute(new Person("Jan"));
    return "hello";
}
```

Ustawiając w ten sposób atrybut, będziemy mieli do dyspozycji obiekt klasy **Person** o nazwie **person**, nazwa ta jest generowana automatycznie - poniżej link do dokumentacji opisujący zasady jej powstawania:

<https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/core/Conventions.html#getVariableName-java.lang.Object->

Parametry internacjonalizacji

W pliku tłumaczenia dodajemy wpis:

```
hello.message= Witam {0}
```

W pliku szablonu umieszczamy wartość za pomocą następującej konstrukcji:

```
<p th:text="#{hello.message(${person.firstName})}">Tekst Powitania</p>
```

Jako wynik otrzymamy następujący html:

```
<p>Witam Jan</p>
```

th:href

Do tworzenia odnośników używamy atrybutu **th:href** jako wartość dajemy wyrażenie w postaci: **@{wyr}**

określając adres url, do którego ma prowadzić nasz odnośnik, np:

```
<a th:href="@{/panel}">Panel</a>
```

lub przekazując dodatkowe parametry w postaci:

```
<a th:href="@{/panel(userName=${person.firstName})}">Panel</a>
```

Jako wynik otrzymamy url postaci:

```
http://localhost:8080/panel?userName=Jan
```

th:href

Jeżeli chcemy dodać więcej niż jeden parametr oddzielamy je znakiem przecinka, np:

```
<a th:href="@{/panel(userName=${person.firstName},lastName='Kowalski')}">Panel</a>
```

otrzymamy url:

```
http://localhost:8080/panel?userName=Jan&lastName=Kowalski
```

Parametry ścieżki ustawiamy w następujący sposób:

```
<a th:href="@{/panel/{firstName}/user(firstName=${person.firstName})}">Panel</a>
```

otrzymamy url:

```
http://localhost:8080/panel/Jan/user
```


th:href

Możemy jednocześnie skorzystać z wartości umieszczonej w pliku tłumaczeń:

```
<a th:href="@{/panel(userName=${person.firstName})}"  
    th:text="#{url.panel.text}">Panel</a>
```

Dla klucza w pliku tłumaczeń:

```
url.panel.text = Link do panelu
```

Wartości atrybutów

Za pomocą odpowiednich znaczników **Thymeleaf** mamy możliwość ustawienia atrybutów występujących w html, np:

- th:class
- th:width
- th:alt
- th:src

Przykład umieszczenia obrazka:

```

```

Zasoby statyczne takie jak obrazki, pliki css lub skrypty javascript umieszczamy w katalogu **/src/main/resources/static**.

th:if

Odpowiednik instrukcji warunkowej:

```
<p th:if="${person}" th:text="'Informacje o Janie.'"></p>  
<p th:if="${person1}" th:text="'Informacje o Janie.'"><p>
```

Pierwszy z paragrafów się wyświetli ponieważ istnieje atrybut o nazwie **person**.

Warunki mogą również przyjmować bardziej rozbudowaną postać:

```
<p th:if="${person.firstName.equals('Jan')}" th:text="'Informacje o Janie.'"></p>
```

th:unless

Tag ten stanowi odwrotność dla **th:if**, np:

```
<p th:unless="${person.firstName.equals('Janek')}"  
  th:text="'Jeżeli nie jest Jan'"></p>
```

W wyrażeniach możemy wykorzystywać także specjalne obiekty np: **#string** - służący do operacji na napisach, np:

```
<h1 th:unless="${#strings.substring(person.firstName,0,1).equals('A')}"  
  th:text="'Pierwsza litera imienia to nie A'"></h1>
```

Element wyświetli się tylko w przypadku gdy pierwsza litera imienia nie jest równa **A**.

Obiekty pomocnicze

Do dyspozycji **Thymeleaf** daje nam obiekty pomocnicze, np.:

- **#strings** - do operacji na obiektach typu **String**
- **#numbers** - metody do formatowania obiektów numerycznych
- **#dates** - do operacji na obiektach typu **java.util.Date**

Listę wszystkich obiektów wraz z opisem ich przeznaczenia znajdziemy tutaj:

<http://www.thymeleaf.org/doc/tutorials/2.1/usingthymeleaf.html#expression-utility-objects>

Przykłady ich wykorzystania znajdziemy tutaj:

<http://www.thymeleaf.org/doc/tutorials/2.1/usingthymeleaf.html#appendix-b-expression-utility-objects>

th:switch

Odpowiednikiem instrukcji **switch .. case** jest atrybut **th:switch** definiujemy go w następujący sposób:

```
<div th:switch="${person.firstName}">
  <span th:case="Jan">Cześć Jan</span>
  <span th:case="Janek" th:text="|Cześć ${person.firstName}|"></span>
  <span th:case="*">Witamy na stronie.</span>
</div>
```

th:switch

Odpowiednikiem instrukcji **switch .. case** jest atrybut **th:switch** definiujemy go w następujący sposób:

```
<div th:switch="${person.firstName}">
  <span th:case="Jan">Cześć Jan</span>
  <span th:case="Janek" th:text="|Cześć ${person.firstName}|"></span>
  <span th:case="*">Witamy na stronie.</span>
</div>
```

Sprawdzamy wartość atrybutu modelu.

th:switch

Odpowiednikiem instrukcji **switch .. case** jest atrybut **th:switch** definiujemy go w następujący sposób:

```
<div th:switch="${person.firstName}">
  <span th:case="Jan">Cześć Jan</span>
  <span th:case="Janek" th:text="|Cześć ${person.firstName}|"></span>
  <span th:case="*">Witamy na stronie.</span>
</div>
```

Sprawdzamy wartość atrybutu modelu.

Wykorzystujemy znak pipe | do połączenia statycznej zawartości i dynamicznej pobranej z obiektu.

th:switch

Odpowiednikiem instrukcji **switch .. case** jest atrybut **th:switch** definiujemy go w następujący sposób:

```
<div th:switch="${person.firstName}">
  <span th:case="Jan">Cześć Jan</span>
  <span th:case="Janek" th:text="|Cześć ${person.firstName}|"></span>
  <span th:case="*">Witamy na stronie.</span>
</div>
```

Sprawdzamy wartość atrybutu modelu.

Wykorzystujemy znak pipe | do połączenia statycznej zawartości i dynamicznej pobranej z obiektu.

Odpowiednik **default** z instrukcji **Javy**.

th:object

Korzystając z wyrażenia `*{...}` oraz atrybutu **th:object**, możemy pobrać odpowiedni atrybut wcześniej wskazanego obiektu.

Zamiast:

```
<p th:text="${person.firstName}">Person firstName</p>
```

możemy określić obiekt, którym w ramach danego tagu będziemy się posługiwać, np:

```
<div th:object="${person}">  
  <p th:text="*{firstName}">Person firstName</p>  
</div>
```

th:object

Korzystając z wyrażenia `*{...}` oraz atrybutu **th:object**, możemy pobrać odpowiedni atrybut wcześniej wskazanego obiektu.

Zamiast:

```
<p th:text="${person.firstName}">Person firstName</p>
```

możemy określić obiekt, którym w ramach danego tagu będziemy się posługiwać, np:

```
<div th:object="${person}">  
  <p th:text="*{firstName}">Person firstName</p>  
</div>
```

Określamy obiekt, którym wewnątrz elementu **<div>** będziemy się posługiwali.

th:object

Korzystając z wyrażenia `{...}` oraz atrybutu **th:object**, możemy pobrać odpowiedni atrybut wcześniej wskazanego obiektu.

Zamiast:

```
<p th:text="${person.firstName}">Person firstName</p>
```

możemy określić obiekt, którym w ramach danego tagu będziemy się posługiwać, np:

```
<div th:object="${person}">  
  <p th:text="*{firstName}">Person firstName</p>  
</div>
```

Określamy obiekt, którym wewnątrz elementu **<div>** będziemy się posługiwali.

Wskazujemy właściwość do wyświetlenia, nie poprzedzając jej nazwą obiektu. [Zwróć uwagę na znak gwiazdki *](#)

th:each

W celu iteracji po elementach tablic, kolekcji, map wykorzystujemy atrybut **th:each**:

Do modelu przekazujemy listę obiektów typu **Person**:

```
@GetMapping("/helloEach")  
public String helloEach(Model model) {  
    List<Person> people = new ArrayList<>();  
    people.add(new Person("Arek"));  
    people.add(new Person("Darek"));  
    model.addAttribute("people", people);  
    return "helloList";  
}
```

th:each

Przykład dla elementów **ul** oraz **li**:

```
<ul>
  <li th:each="person: ${people}" th:text="${person.firstName}"></li>
</ul>
```

Przykład dla tabeli html:

```
<table>
  <tr th:each="person: ${people}">
    <td th:text="${person.firstName}"></td>
  </tr>
</table>
```


th:each

Do dyspozycji mamy dodatkową zmienną przechowującą status pętli, aby uzyskać do niej dostęp definiujemy pętlę w następujący sposób:

```
<tr th:each="person, iterStat: ${people}" th:class="${iterStat.odd}? 'odd' ">
    <td th:text="${person.firstName}"></td>
    <td th:text="${iterStat.count}"></td>
    <td th:text="${iterStat.index}"></td>
</tr>
```

iterStat - to nazwa zmiennej do której mamy dostęp wewnątrz naszej pętli.

th:each

Do dyspozycji mamy dodatkową zmienną przechowującą status pętli, aby uzyskać do niej dostęp definiujemy pętlę w następujący sposób:

```
<tr th:each="person, iterStat: ${people}" th:class="${iterStat.odd}? 'odd' ">
    <td th:text="${person.firstName}"></td>
    <td th:text="${iterStat.count}"></td>
    <td th:text="${iterStat.index}"></td>
</tr>
```

iterStat - to nazwa zmiennej do której mamy dostęp wewnątrz naszej pętli.

Wykorzystujemy atrybut o nazwie **odd**, który ma wartość **true** jeżeli obrót pętli jest nieparzysty, przy pomocy kolejnego atrybutu **th:class** oraz operatora trój-argumentowego ustawiamy klasę **odd**.

th:each

Do dyspozycji mamy dodatkową zmienną przechowującą status pętli, aby uzyskać do niej dostęp definiujemy pętlę w następujący sposób:

```
<tr th:each="person, iterStat: ${people}" th:class="${iterStat.odd}? 'odd' ">
    <td th:text="${person.firstName}"></td>
    <td th:text="${iterStat.count}"></td>
    <td th:text="${iterStat.index}"></td>
</tr>
```

iterStat - to nazwa zmiennej do której mamy dostęp wewnątrz naszej pętli.

Wykorzystujemy atrybut o nazwie **odd**, który ma wartość **true** jeżeli obrót pętli jest nieparzysty, przy pomocy kolejnego atrybutu **th:class** oraz operatora trój-argumentowego ustawiamy klasę **odd**.

Właściwości które mamy do dyspozycji to **count** - liczba iteracji, **index** - numer iterowanego elementu z tablicy/kolekcji - liczony od 0. (pozostałe to **even**, **last**, **first**, **size**).

Fragmenty

Zarówno w większości systemów szablonowych, jak i w **Thymeleaf** mamy możliwość definiowania elementów wielokrotnego użytku, które załączamy w innych plikach.

Fragment definiujemy przy użyciu atrybutu **th:fragment** :

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
  <body>
    <div th:fragment="pagefooter">
      Projekt i realizacja: CodersLab Team.
    </div>
  </body>
</html>
```

Fragmenty

Do wstawiania fragmentów w innych plikach służą atrybuty:

th:include - wkleja zawartość załączanego tagu do środka danego tagu

th:replace - załącza element zastępując tag.

Aby załączyć do pliku fragment znajdujący się pliku o nazwie **footer.html** w katalogu **fragments** wstawiamy następujący kod:

```
<footer th:include="fragments/footer :: pagefooter"></footer>
```

w wyniku otrzymamy:

```
<footer>  
    Projekt i realizacja: CodersLab Team.  
</footer>
```

Fragmenty

Używając w tym celu kodu:

```
<footer th:replace="fragments/footer :: pagefooter"></footer>
```

w wyniku otrzymamy:

```
<div>  
    Projekt i realizacja: CodersLab Team.  
</div>
```

Formularze

Dla obsługi formularzy wykorzystujemy dodatkowe tagi, tak jak w poniższym przykładzie:

```
<form th:action="@{/person/add}" th:object="${person}" method="post">  
    <p>First name: <input type="text" th:field="*{firstName}" /></p>  
    <p><input type="submit" value="Submit" /></p>  
</form>
```


Formularze

Dla obsługi formularzy wykorzystujemy dodatkowe tagi, tak jak w poniższym przykładzie:

```
<form th:action="@{/person/add}" th:object="${person}" method="post">  
    <p>First name: <input type="text" th:field="*{firstName}" /></p>  
    <p><input type="submit" value="Submit" /></p>  
</form>
```

Za pomocą atrybutu **th:action** definiujemy akcję formularza.

Formularze

Dla obsługi formularzy wykorzystujemy dodatkowe tagi, tak jak w poniższym przykładzie:

```
<form th:action="@{/person/add}" th:object="${person}" method="post">
    <p>First name: <input type="text" th:field="*{firstName}" /></p>
    <p><input type="submit" value="Submit" /></p>
</form>
```

Za pomocą atrybutu **th:object** definiujemy obiekt modelu do którego dane będą bindowane.

Formularze

Dla obsługi formularzy wykorzystujemy dodatkowe tagi, tak jak w poniższym przykładzie:

```
<form th:action="@{/person/add}" th:object="${person}" method="post">  
    <p>First name: <input type="text" th:field="*{firstName}" /></p>  
    <p><input type="submit" value="Submit" /></p>  
</form>
```

Za pomocą atrybutu **th:field** określamy pola jakie zawiera bindowany obiekt.

Layout

Thymeleaf umożliwia tworzenie szkieletów elementów wspólnych dla wszystkich podstron naszego serwisu.

Przykładowy layout

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org"
      xmlns:layout="http://www.ultraq.net.nz/web/thymeleaf/layout">
  <body>
    <div layout:fragment="content">
      Tutaj pojawi się zawartość widoku korzystającego z tego layoutu.
    </div>
  </body>
</html>
```

Layout

Thymeleaf umożliwia tworzenie szkieletów elementów wspólnych dla wszystkich podstron naszego serwisu.

Przykładowy layout

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org"
      xmlns:layout="http://www.ultraq.net.nz/web/thymeleaf/layout">
  <body>
    <div layout:fragment="content">
      Tutaj pojawi się zawartość widoku korzystającego z tego layoutu.
    </div>
  </body>
</html>
```

Określamy przestrzeń nazw dla atrybutów **layout**.

Layout

Thymeleaf umożliwia tworzenie szkieletów elementów wspólnych dla wszystkich podstron naszego serwisu.

Przykładowy layout

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org"
      xmlns:layout="http://www.ultraq.net.nz/web/thymeleaf/layout">
  <body>
    <div layout:fragment="content">
      Tutaj pojawi się zawartość widoku korzystającego z tego layoutu.
    </div>
  </body>
</html>
```

Określamy miejsce, w które ma zostać wstawiona zawartość konkretnego widoku.

Layout

Modyfikujemy widok tak by korzystał z layoutu:

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org"
      xmlns:layout="http://www.ultraq.net.nz/web/thymeleaf/layout"
      layout:decorator="layout">
<body>
<section layout:fragment="content">
    //Zawartość widoku
</section>
</body>
</html>
```

Layout

Modyfikujemy widok tak by korzystał z layoutu:

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org"
      xmlns:layout="http://www.ultraq.net.nz/web/thymeleaf/layout"
      layout:decorator="layout">
<body>
<section layout:fragment="content">
    //Zawartość widoku
</section>
</body>
</html>
```

Określamy nazwę **layoutu**, z jakiego korzysta nasz widok.