

A photograph of a man with a beard and short hair, wearing a light blue hoodie, sitting at a desk and working on a silver laptop. He is looking down at the screen with a focused expression. The background is a plain, light-colored wall.

Spring Boot – wprowadzenie

- 
1. Wprowadzenie do Spring Boot
 2. Tworzenie beanów i wstrzykiwanie zależności
 3. Konfiguracja projektu
 4. Auto-konfiguracja i Spring Actuator
 5. Spring Data JPA
 6. Tworzenie REST API
 7. Programowanie aspektowe, Spring Security i inne

Dlaczego warto uczyć się Springa?

- największa popularność wśród frameworków używanych w projektach opartych o Javę.
- usunięcie złożoności, bałaganu i standaryzacja kodu czyli ułatwienie programowania.
- framework jest aktywnie rozwijany, posiada obszerną dokumentację oraz dużą społeczność użytkowników.
- stabilność, mimo rozwoju, core frameworka nie zmienia się od lat.

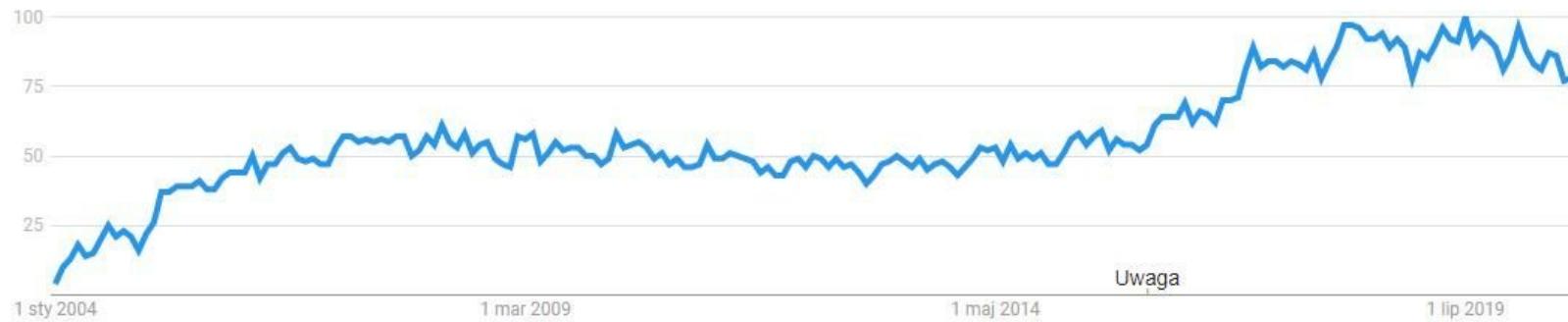
Popularność

Spring Framework
Oprogramowanie

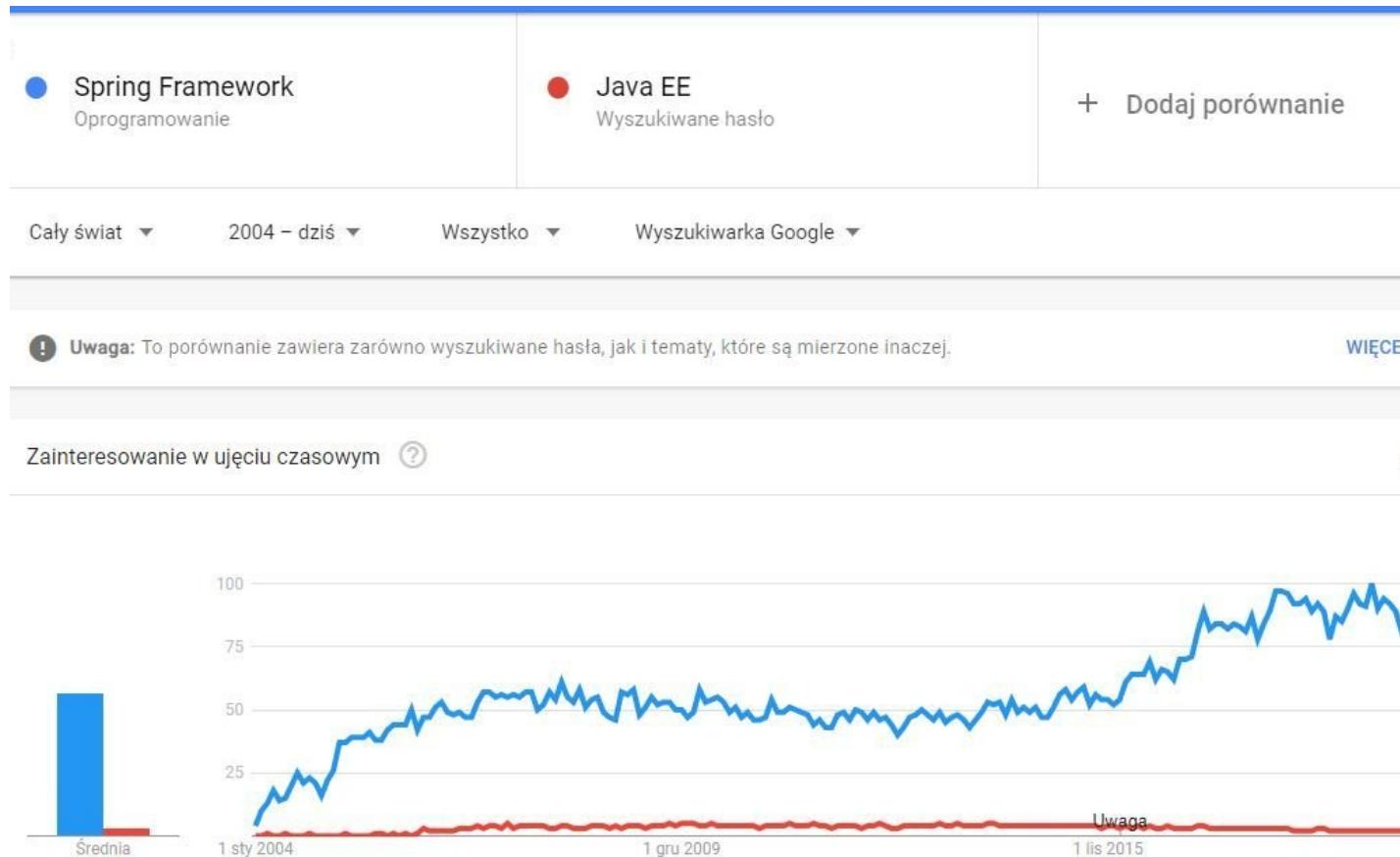
+ Porównaj

Cały świat ▾ 2004 – dziś ▾ Wszystko ▾ Wyszukiwarka Google ▾

Zainteresowanie w ujęciu czasowym ?



Spring vs Java EE



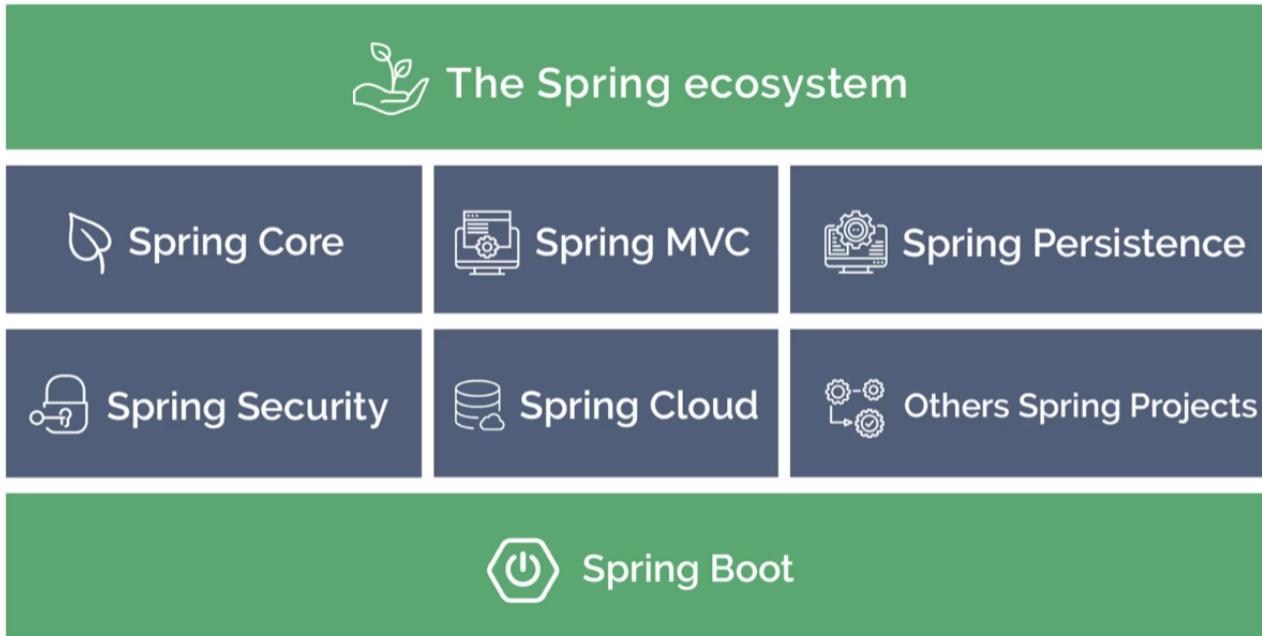
Zapytania na stackoverflow

- Spring Questions – 187k
- Hibernate Questions – 87k
- Maven Questions – 79k
- Tomcat Questions – 41k
- Servlets Questions – 32k
- Java EE Questions – 29k
- Grails Questions – 29k

Spring Boot

- Spring Boot jest rozszerzeniem frameworka Spring, które jest dostarczane z domyślną konfiguracją.
- Przed Spring Boot tworzenie aplikacji Spring wymagało dużej ilości konfiguracji, aby rozpocząć projekt.
- Domyślna konfiguracja może zostać nadpisana i dostosowana do naszych potrzeb.

Czym jest Spring?



Spring Boot

Podstawowe funkcje Spring Boot:

- **Startery** - mogą być postrzegane jako moduły, które zawierają określoną funkcjonalność (JPA, testowanie, sieć, poczta, logowanie itp.).
- **Automatyczna konfiguracja** — domyślna konfiguracja, która jest ładowana w określonych warunkach.
- **Zarządzanie zależnościami** – pomaga rozwiązać problem niezgodnych wersji zależności.

Pierwszy projekt

- **Java 11** (zainstalowane + ścieżki dostępu)
- Narzędzie do budowy wersji: **Maven**
- Narzędzie do repozytorium kodu: **Git**
- **Spring Initializer** (strona lub IntelliJ Idea)
 - Uruchomienie projektu:
 - * IntelliJ Idea
 - * Linia komend: mvn clean install

pom.xml

```
<groupId>pl.fis</groupId>
<artifactId>spring1b</artifactId>
<version>0.0.1-SNAPSHOT</version>

<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.5.3</version>
    <relativePath/> 
</parent>
```

pom.xml

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

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

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

Struktura projektu

- src/main/java
- src/main/resources
- src/test/java
- src/test/resources

Zależności (dependencje)

Klasa A jest zależna od klasy B, gdy wchodzi z nią w interakcje w jakikolwiek sposób.

Wstrzyknięcie to po prostu proces wstrzykiwania obiektu typu B do obiektu typu A.

Zależności (dependencje)

```
public class A {  
    private B bDependency;  
  
    public A() {  
        bDependency = new B();  
    }  
}
```

```
public class A {  
    private B bDependency;  
  
    public A(B bDependency) {  
        this.bDependency = bDependency;  
    }  
}
```

Spring IOC Container

Spring IOC Container tworzy obiekty, łączy je ze sobą, konfiguruje i zarządza ich pełnym cyklem życia.

Spring Component Scanning

Najprostszą adnotacją której możemy użyć jest `@Component`.

Zasadniczo, podczas procesu ładowania kontekstu, Spring skanuje katalog projektu w poszukiwaniu klas z adnotacjami `@Component` i tworzy ich instancje jako beany.

```
@Component
public class ProjectRepositoryImpl implements IProjectRepository {
    // ...
}
```

Spring Component Scanning

Jest to realizowane dzięki istniejącej w konfiguracji projektu adnotacji *ComponentScan*.

Jeśli podglądnimy definicję adnotacji `@SpringBootApplication` (w głównej klasie projektu), to zobaczymy, że zawiera ona adnotację `@ComponentScan`.

Spring Component Scanning

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(
    excludeFilters = {@Filter(
        type = FilterType.CUSTOM,
        classes = {TypeExcludeFilter.class}
    ), @Filter(
        type = FilterType.CUSTOM,
        classes = {AutoConfigurationExcludeFilter.class}
    )}
)
public @interface SpringBootApplication {
```

Spring Component Scanning

```
@Configuration  
@ComponentScan(basePackages= {"com.baeldung.ls.persistence"})  
public class PersistenceConfig {  
    //  
}
```

Inne adnotacje typu stereotype

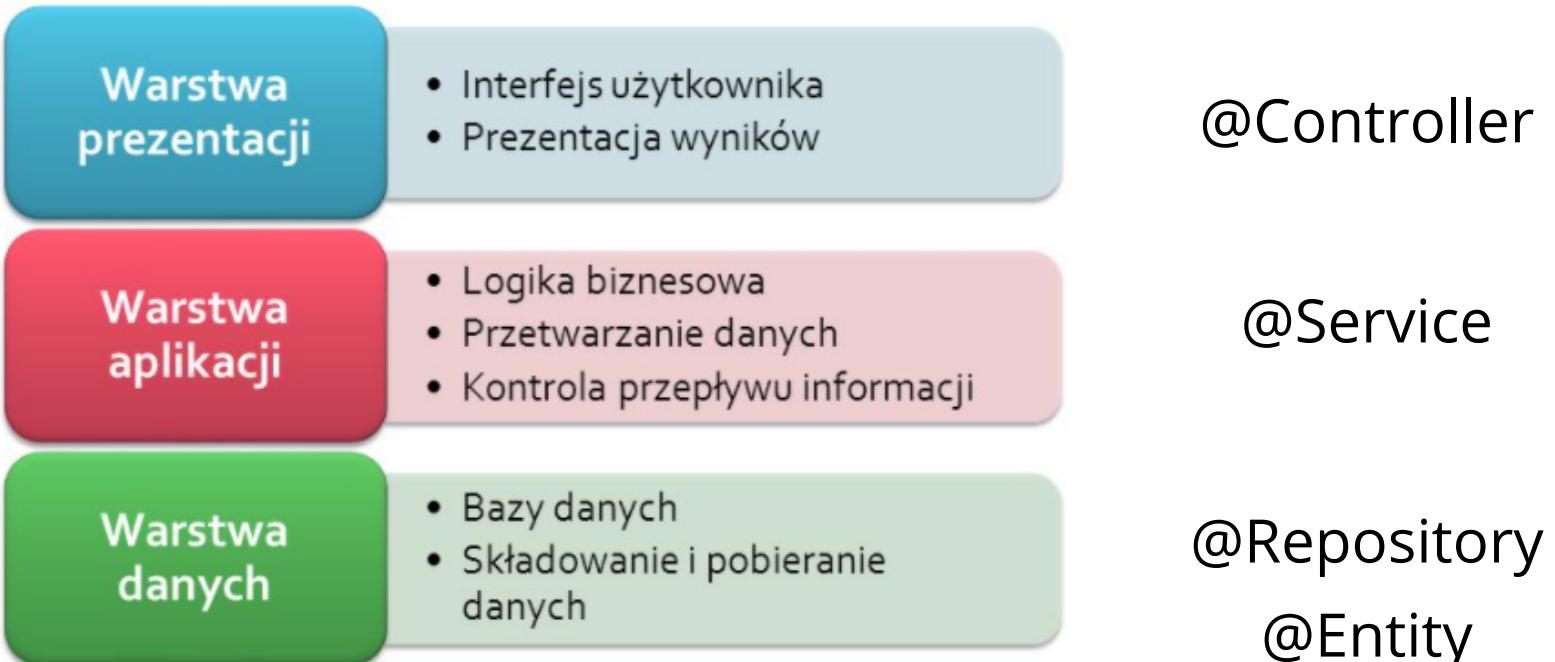
@Repository

@Service

@Entity

@Controller

Warstwy aplikacji webowej



Lazily Initialized Beans

Domyślnie kontener tworzy i konfiguruje wszystkie ziarna natychmiast w trakcie uruchomienia systemu.

Ogólnie takie zachowanie jest pożąданie, ponieważ błędy dotyczące konfiguracji komponentów bean są wykrywane natychmiast po uruchomieniu aplikacji.

Ale jeśli nie chcemy, aby tak się stało, zawsze możemy temu zapobiec, dodając anotację @Lazy do naszych definicji beanów.

Cykl życia beana

Zasadniczo cykl życia beana składa się z 3 faz:

- faza inicjalizacji
- faza użycia
- faza zniszczenia

Faza inicjalizacji

Składa się z 2 etapów:

- załadowanie definicji beanów
- tworzenie obiektów beanów

```
@PostConstruct  
public void post() {  
    //  
}
```

Faza zniszczenia

Gdy dany bean jest niepotrzebny np. w przypadku beanów sesyjnych, które żyją tylko tyle ile sesja użytkownika lub zakończenia aplikacji to następuje procedura ich niszczenia.

Następnie beany są dostępne dla garbage collectora, który może je fizycznie usunąć z maszyny wirtualnej Java.

```
@PreDestroy  
public void preDestroy() {  
    log.info("@PreDestroy annotated method is called.");  
}
```

Wstrzykiwanie zależności

- poprzez konstruktor
- poprzez setter
- poprzez pole

Poprzej konstruktor

```
@Service
public class ProjectServiceImpl implements IProjectService {

    private IProjectRepository projectRepository;

    public ProjectServiceImpl(IProjectRepository projectRepository) {
        this.projectRepository = projectRepository;
    }

    // ...
}
```

Poprzej setter

```
public class ProjectServiceImplSetterInjection implements IProjectService {  
  
    private IProjectRepository projectRepository;  
  
    @Autowired  
    public void setProjectRepository(IProjectRepository projectRepository) {  
        this.projectRepository = projectRepository;  
    }  
  
    // ...  
}
```

Poprzez pole

```
@Service
public class ProjectServiceImplAutowiring implements IProjectService {

    @Autowired
    private IProjectRepository projectRepository;

    // ...
}
```

Dobre praktyki

- użyj wstrzykiwania poprzez konstruktor dla zależności, które są niezbędne.
- użyj wstrzykiwania poprzez pole lub setter dla opcjonalnych zależności.

Wstrzykiwanie poprzez konstruktor ma istotną przewagę nad innymi - prowadzi do bardziej testowalnego kodu.

Powodem jest po prostu to, że przy wstrzyknięciu poprzez konstruktor wszystkie niezbędne obiekty są zainicjalizowane.

@Qualifier

```
@Service
public class ProjectServiceImpl1 implements ProjectService{
```

```
@Service
public class ProjectServiceImpl2 implements ProjectService{
```

```
@Autowired
ProjectService projectService;
```

@Qualifier

```
 @Service  
 @Qualifier("ps1")  
 public class ProjectServiceImpl implements ProjectService{  
  
 @Service  
 @Qualifier("ps2")  
 public class ProjectServiceImpl2 implements ProjectService{  
  
 @Autowired  
 @Qualifier("ps1")  
 ProjectService projectService;}
```

@Primary

```
 @Service  
 @Primary  
 public class ProjectServiceImpl1 implements ProjectService{  
  
 @Service  
 public class ProjectServiceImpl2 implements ProjectService{  
  
 @Autowired  
 ProjectService projectService;
```

Beans Scope

- Singleton
- Prototype
- Request
- Session
- Application
- Websocket

Beans Scope

```
@Service  
@Primary  
@Scope("singleton")  
public class ProjectServiceImpl implements ProjectService{
```

```
@Service  
@Scope("prototype")  
public class ProjectServiceImpl2 implements ProjectService{
```

Application Context

Reprezentuje kontener IoC. Odpowiada za tworzenie instancji, konfigurowanie beanów i wstrzykiwanie zależności.

Dostęp do AC - sposób 1

```
@Service
public class ProjectServiceImpl2 implements ProjectService, ApplicationContextAware {

    @Override
    public List<Project> findAllProjects() { return null; }

    @Override
    public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {
        }
}
```

Dostęp do AC - sposób 2

```
@Service  
public class ProjectServiceImpl2 implements ProjectService {  
  
    @Autowired  
    private ApplicationContext applicationContext;  
  
    @Override  
    public List<Project> findAllProjects() { return null; }  
  
}
```

Pobieranie beanów

z AC

```
ProjectService projectService = applicationContext.getBean("projectServiceImpl", ProjectService.class);
```

Properties

application.properties

propertyName=PropertyValue

Kod Java:

@Value("\${propertyName}")

Properties - przykład

application.properties

project.prefix=PRO

project.suffix=123

Kod Java:

```
@Value("${project.prefix}")
```

```
private String prefix;
```

```
@Value("${project.suffix}")
```

```
private Integer suffix;
```

Profile

```
@Profile("dev")
@Repository
public class ProjectRepositoryDevImpl implements IProjectRepository {

    Map<Integer, Project> projectMap = new HashMap<>();

    @Override
    public Optional<Project> findById(Long id) {
        return Optional.ofNullable(projectMap.get(id));
    }

}
```

Profile

```
    @Profile("prod")
    @Repository
    public class ProjectRepositoryImpl implements IProjectRepository {

        @Override
        public Optional<Project> findById(Long id) {
            //getting project from db
            return Optional.empty();
        }

    }
```

Aktywowanie profilu

application.properties

spring.profiles.active=dev

Logowanie aktywności aplikacji

```
private static final Logger LOG = LoggerFactory.getLogger(ProjectRepositoryImpl.class);  
  
LOG.info("Finding Project By Id {}", id);
```

Logowanie aktywności aplikacji

Poziomy logowania:

- ERROR
- WARN
- INFO
- DEBUG
- TRACE

Poziom logowania domyślnie ustawiony jest na INFO.

Oczywiście możemy dostroić ten poziom.

Jeśli chcemy zmienić ogólny poziom logowania aplikacji:

`logging.level.root=WARN` (zmiana w application.properties)

Logowanie aktywności aplikacji

Zmiana poziomu logowania dla konkretnej grupy klas (pakietu).

logging.level.org.springframework=INFO

logging.level.pl.fis.spring1.service=DEBUG

Logowanie aktywności aplikacji

Dodatkowe ustawienia logowania:

```
//logowanie do pliku  
logging.file.name=app.log
```

```
//format daty używany przy logowaniu  
logging.pattern.dateformat=dd-MM-yyyy HH:mm:ss
```

Testowanie aplikacji

Foldery dla testów:

- src/test/java
- src/test/resources

Najprostszy test

```
@SpringBootTest
class SpringApplicationTests {

    @Test
    void contextLoads() {
    }

}
```

Test integracyjny

```
@Test  
public void whenSavingProject_thenOK() {  
    Project savedProject = projectService.save(new Project(name: "name", LocalDate.now()));  
    assertThat(savedProject, is(notNullValue()));  
}
```

Test jednostkowy

W przypadku testów jednostkowych framework Spring nie jest uruchamiany, a co za tym idzie np. zależności nie są wstrzykiwane automatycznie.

Narzędzie do mockowania - Mockito!

Test jednostkowy

```
public class ProjectServiceUnitTest {

    @Mock
    ProjectRepository projectRepository;

    @InjectMocks
    ProjectServiceImpl projectService;

    @BeforeEach
    public void initMocks() {
        MockitoAnnotations.openMocks( testClass: this);
    }

    @Test
    public void whenSavingProject_thenOK() {
        Project project = new Project( name: "name", LocalDate.now());
        when(projectRepository.save(project)).thenReturn(project);

        Project savedProject = projectService.save(project);
        assertNotNull(savedProject);
    }
}
```

Autokonfiguracja

Spring Boot opiera się na adnotacji
@Conditional i wielu jej odmianach:

- @ConditionalOnClass
- @ConditionalOnMissingClass
- @ConditionalOnBean
- @ConditionalOnMissingBean

W zależności od zależności znajdujących się w pliku pom.xml
niektóre konfiguracje są inicjalizowane, a inne nie.

logging.level.org.springframework.boot.autoconfigure=DEBUG

Autokonfiguracja

Raport logowania automatycznej konfiguracji będzie zawierał informacje o:

- **Pozytywne dopasowania** - automatyczne konfiguracje, które są włączone, gdyż ich warunek został spełniony.
- **Negatywne dopasowania** - klasy autokonfiguracji z warunkami ocenianymi jako fałszywe, które pozostają wyłączone.
- **Wykluczenia** - klasy, które wykluczamy.
- **Klasy bezwarunkowe** - konfiguracje bez warunków.

Spring Actuator

- Narzędzia monitorujące.
- Udostępniane poprzez różne endpoint'y głównie HTTP.
- Pomagają w monitorowaniu i zarządzaniu aplikacją

Spring Actuator

Konieczne dodanie nowej dependencji w pliku pom.xml

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

Domyślnie wiele endpointów jest wyłączona.

Aby to zmienić należy dodawać application.properties
następujący wpis:

```
management.endpoints.web.exposure.include=*
```

/health endpoint

np. {"status": "UP"}

Inne statusy to: DOWN, OUT_OF_SERVICE i UNKNOWN.

`http://localhost:8080/actuator/health`

/info endpoint

Ten punkt końcowy pokazuje informacje o naszej aplikacji. Domyślnie nie będzie zawierał żadnych danych, trzeba je najpierw skonfigurować np. w application.properties.

```
info.spring1.name=Example Spring app  
info.spring1.description=A long description...
```

```
{"spring1": {"name": "Example Spring app",  
            "description": "A long description..."}}
```

<http://localhost:8080/actuator/health>

Inne endpointy

Oprócz tego istnieje wiele innych, ciekawych endpointów na temat konfiguracji i życia aplikacji.

Aby zobaczyć listę dostępnych endpointów wystarczy przejść pod adres:

`http://localhost:8080/actuator/`

Dynamiczna zmiana poziomu logowania

Za pomocą ścieżki

<http://localhost:8080/monitoring/loggers/{{logger}}>

możliwa jest zmiana poziomu logowania dla aplikacji w trakcie działania aplikacji (bez konieczności zmian w application.properties i restartu aplikacji).

{{logger}} należy zastąpić słowem ROOT lub nazwą pakietu dla której chcemy ustawić poziom logowania np.

<http://localhost:8080/monitoring/loggers/ROOT>

To jest metoda typu POST więc należy użyć np. narzędzia Postman.

Dynamiczna zmiana poziomu logowania

Treść payloadu do zmiany poziomu logowania jest następująca:

```
{ "configuredLevel": level}
```

np. { "configuredLevel": "DEBUG" }