

# JUnit

Paweł Staniszewski |  logintęgra



**Wyższa Szkoła Bankowa  
w Gdańsku**

## Plan spotkania

1. Dlaczego warto pisać testy automatyczne.
2. Rodzaje testów — jednostkowe, integracyjne, funkcjonalne.
3. Testy jednostkowe — jakie są cechy dobrych testów jednostkowych.
4. JUnit — narzędzie do testowania programów pisanych w Javie.
5. Konfiguracja [JUnit5](#) plus trochę o [Maven](#) i [Gradle](#).
6. Testy jednostkowe w Spring Boot.

Oraz, przy okazji, wspomnimy o:

- [Mockito](#);
- TDD — Test Driven Development.

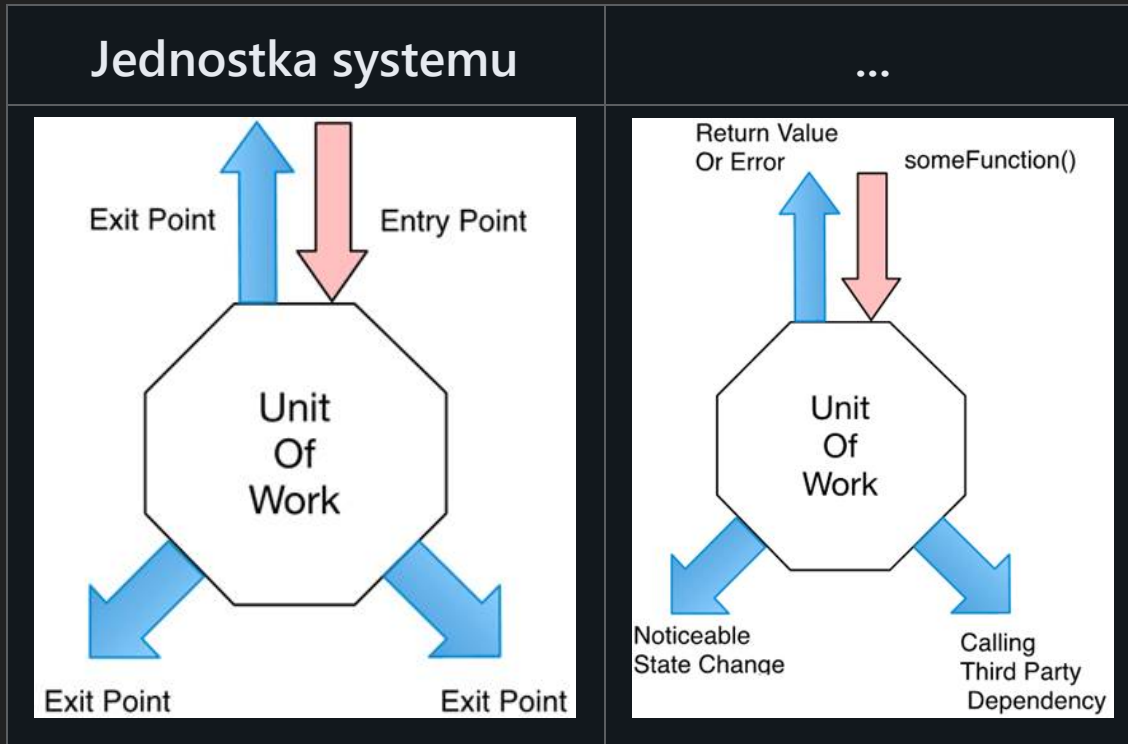
## Test jednostkowy — definicja

A unit test is an automated piece of code that invokes a unit of work in the system and then checks a single assumption about the behavior of that unit of work.

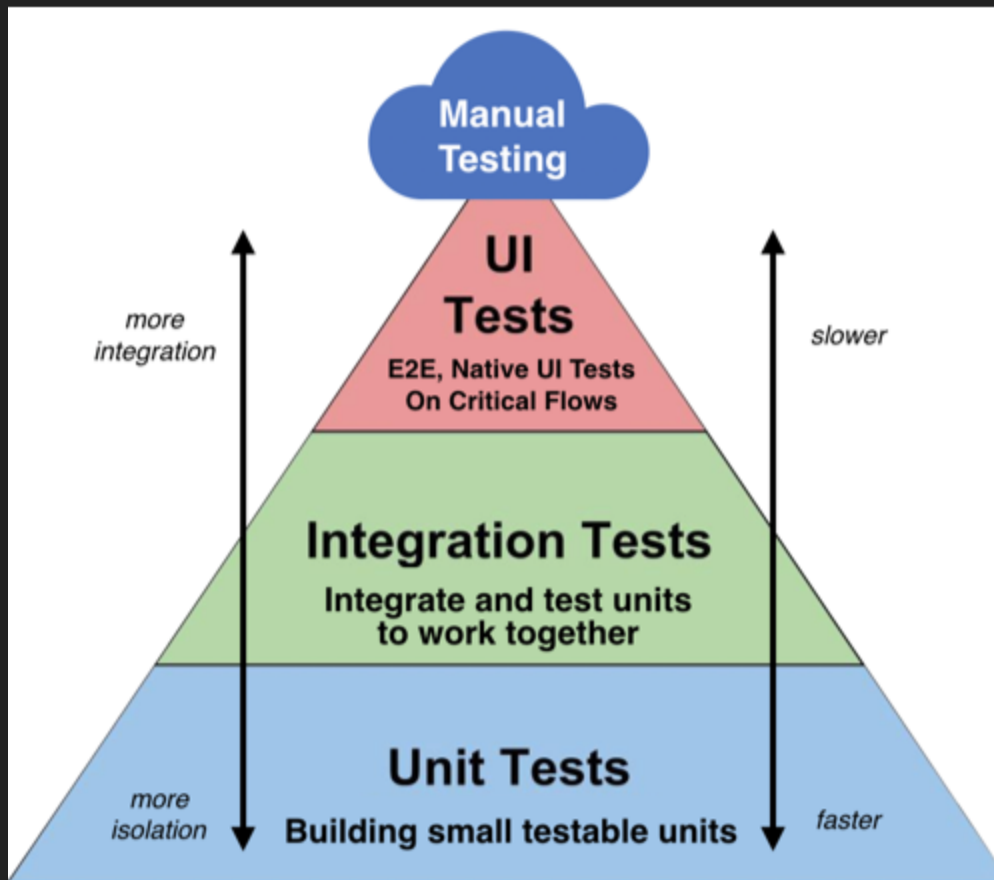
<https://www.artofunittesting.com/definition-of-a-unit-test>

**Test jednostkowy** to fragment kodu, który automatycznie wywołuje pojedynczy element (jednostkę) systemu w celu sprawdzenia jednego konkretnego założenia dotyczącego tego elementu.

A unit of work is a single logical functional use case in the system that can be invoked by some public interface (in most cases).



Źródło: [livebook.manning.com](https://livebook.manning.com)



Źródło: [medium.com](https://medium.com)

Cechy dobrego testu jednostkowego za <https://www.artofunittesting.com/definition-of-a-unit-test>:

- Able to be fully automated
- Has full control over all the pieces running (Use mocks or stubs to achieve this isolation when needed)
- Can be run in any order if part of many other tests
- Runs in memory (no DB or File access, for example)
- Consistently returns the same result (You always run the same test, so no random numbers, for example. save those for integration or range tests)
- Runs fast
- Tests a single logical concept in the system
- Readable
- Maintainable
- Trustworthy (when you see its result, you don't need to debug the code just to be sure)

Żeby zacząć używać *JUnit* w naszym projekcie, musimy dodać go jako *zależność*. W świecie *Javy* najczęściej używamy w tym celu dwóch popularnych narzędzi do budowania oprogramowania:

- **Maven** <-- tego będziemy używać na tych zajęciach,
- oraz **Gradle**.

*Maven* to narzędzie konsolowe — nie posiada interfejsu graficznego. Jego instalacja jest zależna od systemu:

- **IntelliJ** — ma domyślnie zainstalowaną wtyczkę dla Mavena -- nie musimy instalować dodatkowych rzeczy
- **Linux** — `maven` powinien być dostępny w repozytorium

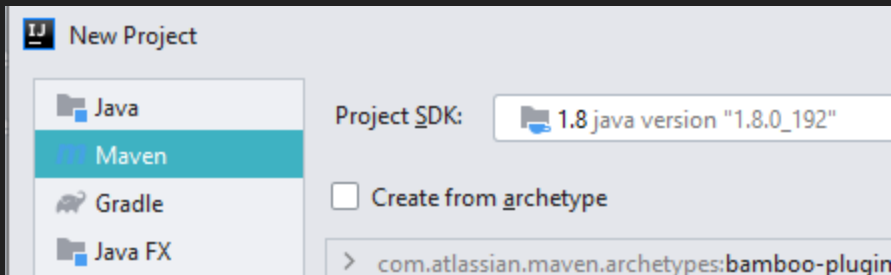
```
sudo apt-get install maven
```

- **Windows** — tu sprawa jest odrobinę trudniejsza, bo musimy ręcznie ustawić zmienne środowiskowe; można spróbować przejść instrukcję na [baeldung.com](https://baeldung.com)
  - Uwaga: Jak pisałem wyżej, IntelliJ — także na Windows — ma wbudowanego Mavena.



## Nowy projekt:

- IntelliJ:



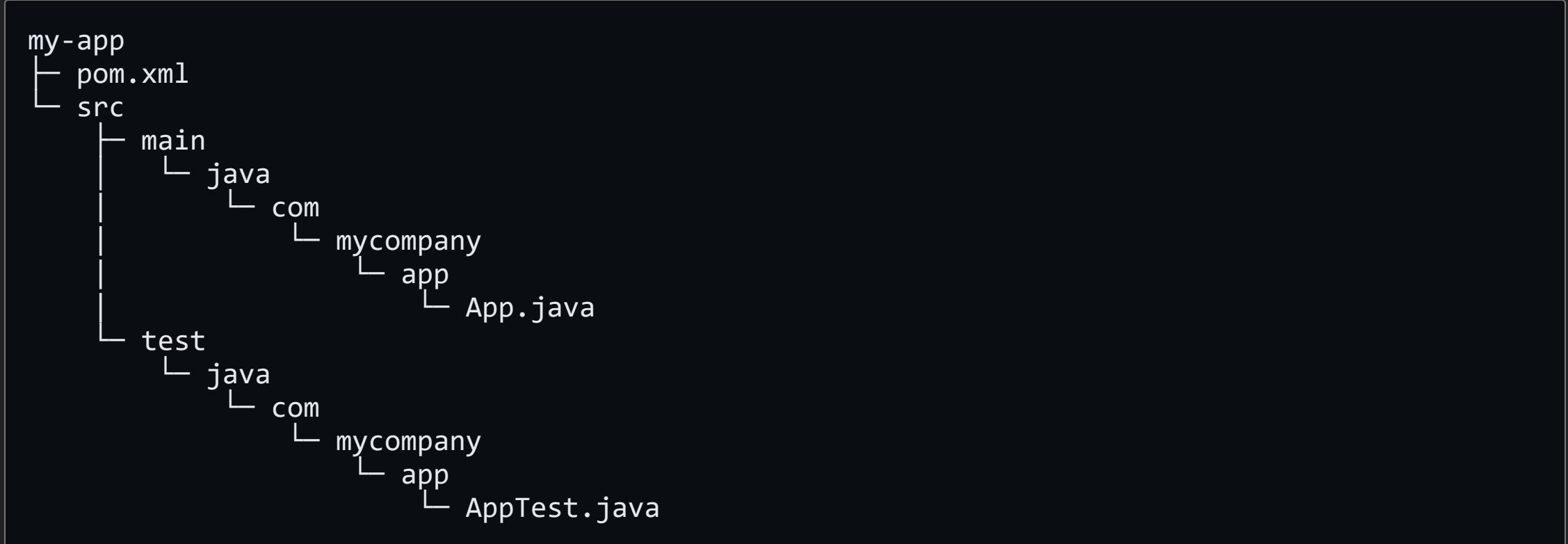
- z konsoli, za pomocą `mvn archetype:generate` — [dokumentacja](#).

Możemy też dodać *Maven* do istniejącego projektu

- IntelliJ --> [dokumentacja](#),
- Dostaniemy nową strukturę folderów (patrze kolejny slajd).

## Maven — struktura folderów

Maven wymaga specyficznej struktury folderów i plików — na pewno się z nią już nieraz spotkaliście:



Najważniejszym plikiem jest jednak `pom.xml` — to on zawiera wszystkie informacje potrzebne do zbudowania, przetestowania, uruchomienia projektu wraz z licencjami, autorami itd. — jest tego **dużo**.

W swojej podstawowej wersji (takiej, jaką utworzy nam IntelliJ) nie jest taki straszny:

```
<?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 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.logintegra</groupId>
  <artifactId>maven-sample-project</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
    <maven.compiler.source>8</maven.compiler.source>
    <maven.compiler.target>8</maven.compiler.target>
  </properties>

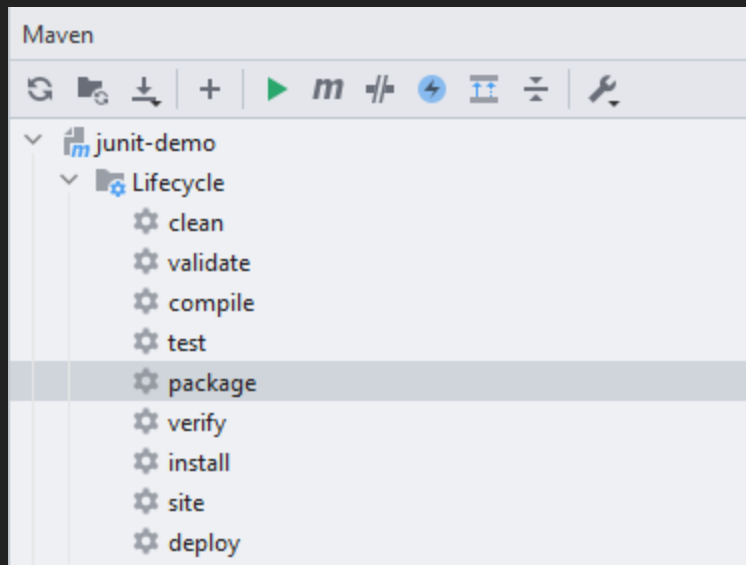
</project>
```

Elementami, które musimy ustawić na początku, są dane naszego projektu:

```
<groupId>com.logintegra</groupId>  
<artifactId>maven-sample-project</artifactId>  
<version>1.0-SNAPSHOT</version>
```

- `groupId` -- możemy to traktować jak `package` w Javie;
- `artifactId` -- nazwa naszego projektu;
- `version` -- wersja naszej aplikacji, najczęściej dwu- lub trzy-cyfrowa, np. `1.0`, `1.0.2`.

--> [Dokumentacja](#)



```
java -cp target/junit-demo-1.0-SNAPSHOT.jar com.logintegra.Main
```

## .gitignore

Plik `.gitignore` określa pliki **celowo** wykluczone z kontroli wersji. Takimi plikami są np.:

- logi aplikacji,
- klasy i pliki tworzone podczas budowania projektu,
- dokumenty zawierające poufne dane (np. hasła).

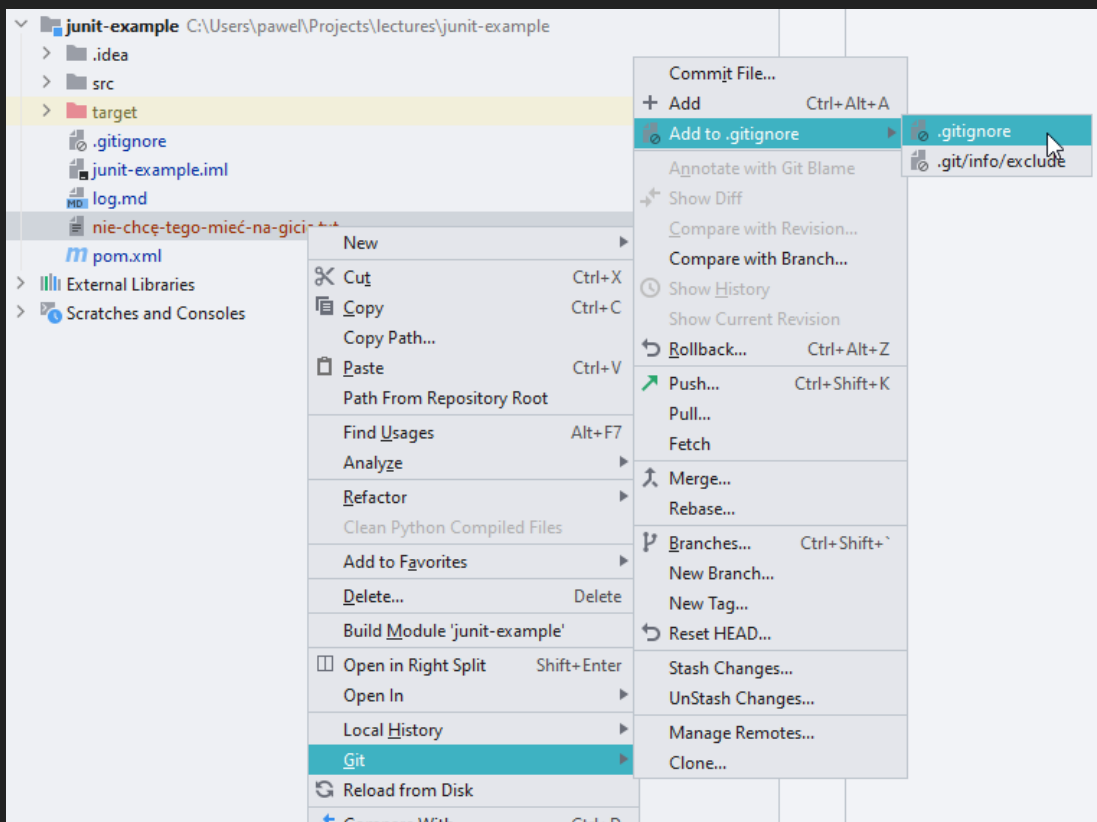
## gitignore.io — serwis generujący pliki `.gitignore`

Pod [tym adresem](#) znajdziecie wygodny serwis generujący typowe `.gitignore` dla popularnych języków, bibliotek i narzędzi, np.:

- IntelliJ: <https://www.toptal.com/developers/gitignore/api/intellij>
- Maven: <https://www.toptal.com/developers/gitignore/api/maven>
- Java: <https://www.toptal.com/developers/gitignore/api/java>

Dodatkowo IntelliJ pozwala nam dodać wybrane pliki do `.gitignore` :

prawoklik na pliku --> Git --> Add do .gitignore --> .gitignore





Wiemy już jak obsłużyć Maven — teraz możemy skonfigurować JUnit 5 w naszym projekcie. W tym celu dodajemy do `pom.xml` dwie zależności:

```
<dependencies>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.7.0</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.7.0</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Napiszmy w końcu nasz pierwszy test. Mamy taką prostą metodę zwracającą pierwszą literę podanego słowa:

```
package com.logintegra;

public class StringUtils {

    /** Zwraca pierwszą literę podanego ciągu znaków. ...*/
    public String getFirstLetter(String s) {

        if (s == null) {
            return "";
        }

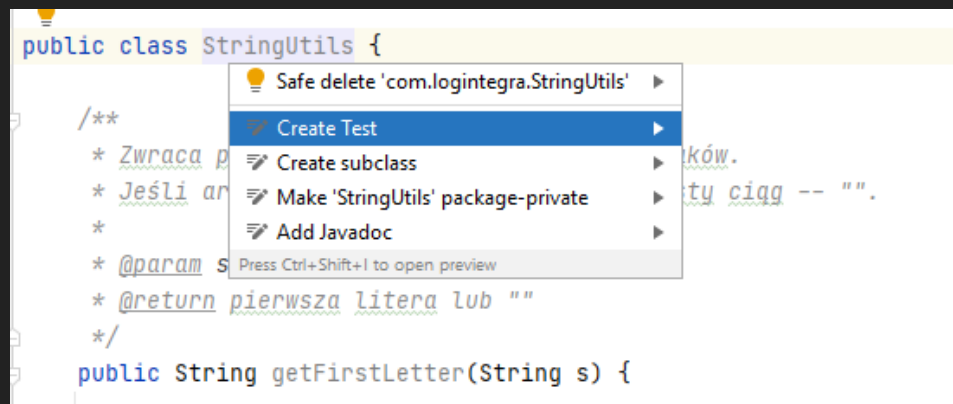
        return s.substring(0, 1);
    }
}
```

Ponieważ używamy Mavena i struktura plików jest ściśle określona, test do tej metody powinien się znaleźć w odpowiednim folderze:

```
junit-sample-project
├── pom.xml
└── src
    ├── main
    │   ├── java
    │   │   ├── com
    │   │   │   ├── logintegra
    │   │   │   │   └── StringUtils.java
    │   └── resources
    └── test
        ├── java
        │   ├── com
        │   │   ├── logintegra
        │   │   │   └── StringUtilsTest.java // To będzie nasz test dla StringUtils.java
        └── resources
```

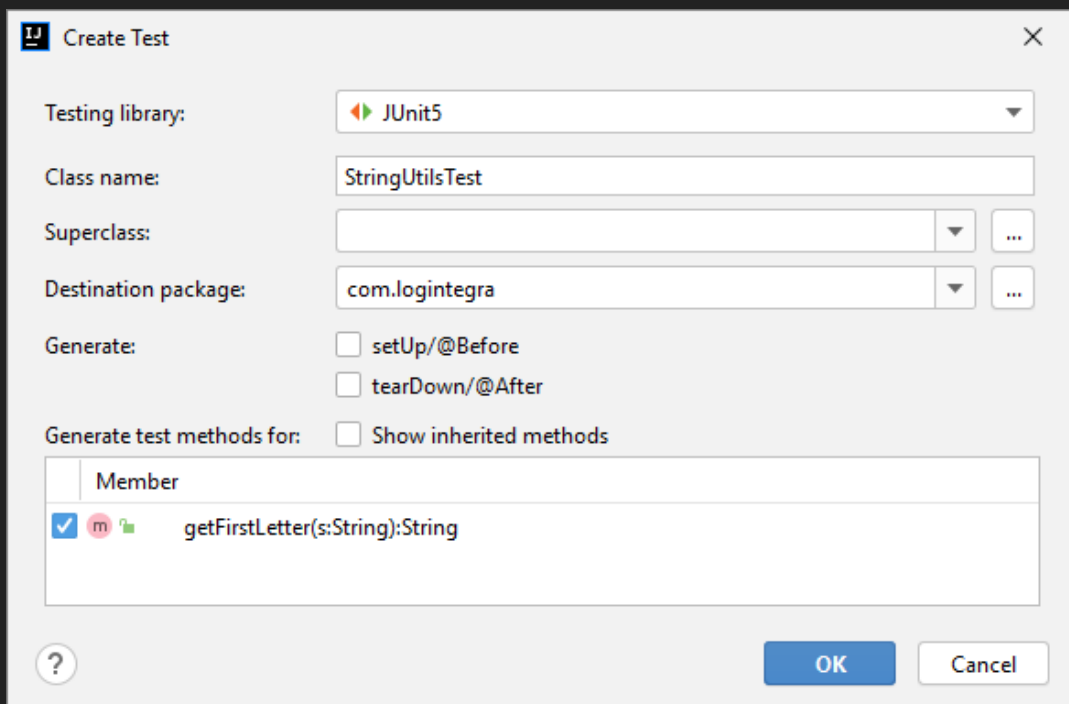
Na szczęście IntelliJ może nam w tym pomóc:

1. Otwieramy klasę, dla której chcemy utworzyć test.
2. Naciskamy **Alt + Enter** \*.
3. W wyświetlonym okienku wybieramy *Create Test*:



\* **Alt + Enter** pokazuje akcje dostępne w danym kontekście — warto go zapamiętać.


Po wybraniu opcji *Create Test* powinno ukazać nam się nowe okienko — większość informacji będzie już wypełniona przez IntelliJ; my musimy tylko zaznaczyć metody, dla których chcemy mieć test:



Uwaga: O `setUp/@Before` i `tearDown/@After` powiemy sobie później.

Jeśli wszystko pójdzie zgodnie z planem, to po naciśnięciu *OK*:

- w odpowiednim folderze utworzona zostanie klasa z sufiksem *Test* — będą już tam importy oraz adnotacje potrzebne do wykonania testu:



The screenshot shows an IDE with a project structure on the left and a code editor on the right. The project structure on the left shows a folder named 'junit-sample-project' with subfolders '.idea', 'out', 'src', and 'test'. The 'src' folder contains 'main' and 'test' subfolders. The 'main' folder contains 'java' and 'com.logintegra' subfolders. The 'test' folder contains 'java' and 'com.logintegra' subfolders. The 'com.logintegra' folder in the 'test' subfolder contains a file named 'StringUtilsTest'. The code editor on the right shows the content of 'StringUtilsTest.java'.

```
1 package com.logintegra;
2
3 import org.junit.jupiter.api.Test;
4
5 import static org.junit.jupiter.api.Assertions.*;
6
7 class StringUtilsTest {
8
9     @Test
10    void getFirstLetter() {
11    }
12 }
```

- dostaniemy pytanie o to, czy dodać nowy plik do *Git* — chcemy, testy są przecież bardzo ważne.

## Nazewnictwo

Niestety, o ile w przypadku `Java` mamy [ustalone konwencje](#) — używamy *camelCase*, nazwy metod zaczynamy od małej, a klas od wielkiej litery itp. — o tyle w przypadku testów tego [nie ma](#).

Najpopularniejszą konwencją (polecaną np. przez [Microsoft](#)) jest złożenie nazwy z

- nazwy testowanej metody,
- scenariusza, który sprawdzamy,
- oczekiwanego rezultatu.

Na przykład dla naszej testujemy metody `getFirstLetter` byłoby to coś w stylu:

```
void getFirstLetter_simpleValue_returnsString() { ... }
```

Osobiście bardziej podoba mi się podejście opisane np. w [tym artykule](#). Testy znacznie częściej są *czytane* niż *pisane* — mają z nimi do czynienia także testerzy, projektanci i nawet klienci — dobrze więc, gdy są zrozumiałe nawet dla osoby, która nie programuje.

- skupiamy się na tym, by nazwy były czytelne także dla nie-programistów;
- nie trzymamy się jednej ścisłej konwencji — ważniejsza jest czytelność;
- oddzielamy słowa podkreślnikami — tak się łatwiej czyta;
- nie uwzględniamy nazwy metody w nazwie testu — chyba, że jest to metoda typowa *techniczna*, niezwiązana z logiką biznesową (jak np. nasza `getFirstLetter`); w przeciwnym razie w metodzie staramy się opisać działanie funkcjonalności.



Przykłady przyzwoitych\* nazw testów:

- `delivery_with_a_past_date_is_invalid ;`
- `carrier_can_move_time_window ;`
- `balance_is_reduced_after_the_money_is_withdrawn`
- `getFirstLetter_returns_first_letter` — tu użycie nazwy metody wydaje się sensowne;
- `getFirstLetter_return_empty_string_when_given_null ;`

*\* Przyzwoitych według mnie. Jak mówiłem jest wiele podejść do nazywania testów — być może w innym zespole takie nazwy będą nieakceptowalne. Najważniejsze jest i tak, jak zawsze, zachowywanie spójności i dbanie o to, by nazwy miały sens.*

Żeby metoda została wykonana podczas przeprowadzania testów, musi być oznaczona jedną z poniższych adnotacji ([dokumentacja](#)):

- `@Test` --> najważniejsza i najczęściej stosowana adnotacja,
- `@ParameterizedTest`,
- `@RepeatedTest`,
- `@TestFactory`,
- `@TestTemplate`.

Samo *sprawdzanie działania* odbywa się za pomocą **asercji** (*assertions*). Najważniejsze z nich to (o każdej jeszcze opowiemy szczegółowo, z przykładami):

- `assertEquals(arg1, arg2)` — czy `arg1` jest równy `arg2`,
- `assertNotNull(arg)` — czy `arg` jest różny od `null`,
- `assertTrue(arg)` — czy `arg` jest *prawdą*, np. `assertTrue(1 + 1 == 2);`,
- `assertThrows(Exception.class, methodUnderTest());`,
- `assertTimeout(ofMillis(10), () -> { methodUnderTest() });` — czy metoda `methodUnderTest()` wykonuje się w określonym czasie,
- `assertAll(name, () -> ...)` — wykonuje wszystkie podane asercje, nawet jeśli któraś się wyłoży

Poniżej mamy minimalny kod, który sprawdzi działanie metody `getFirstLetter`:

```
import org.junit.jupiter.api.Test;

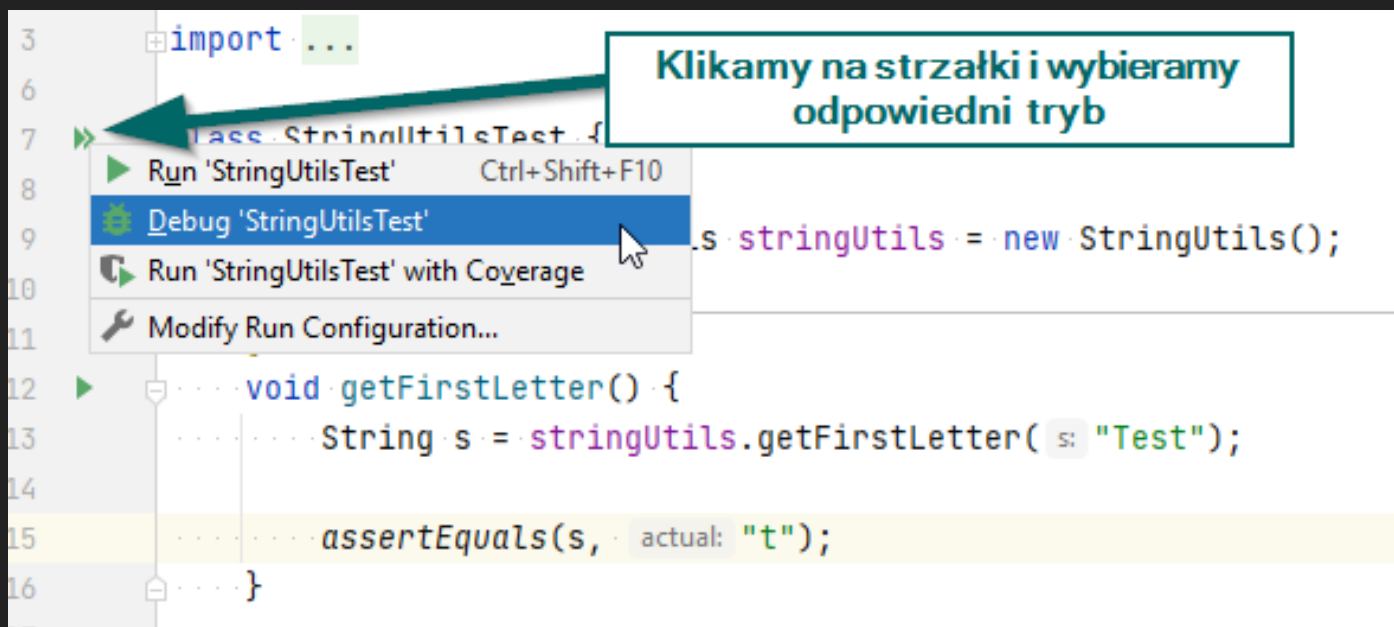
import static org.junit.jupiter.api.Assertions.*;

class StringUtilsTest {

    private final StringUtils stringUtils = new StringUtils();

    @Test
    void getFirstLetter_returns_first_letter() {
        String s = stringUtils.getFirstLetter("Test");
        assertEquals(s, "T");
    }
}
```

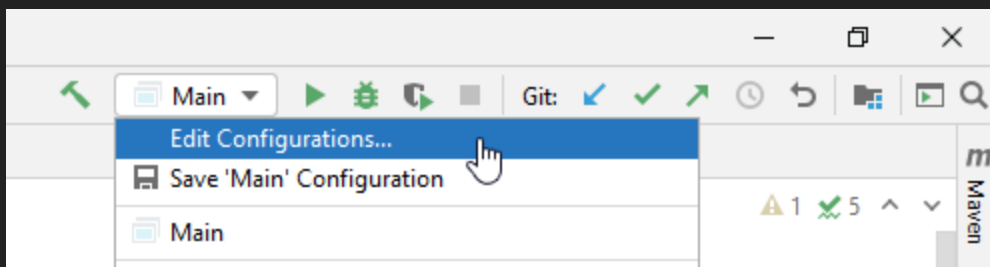
W IntelliJ możemy uruchomić testy całej klasy lub dowolnej metody z poziomu pliku:



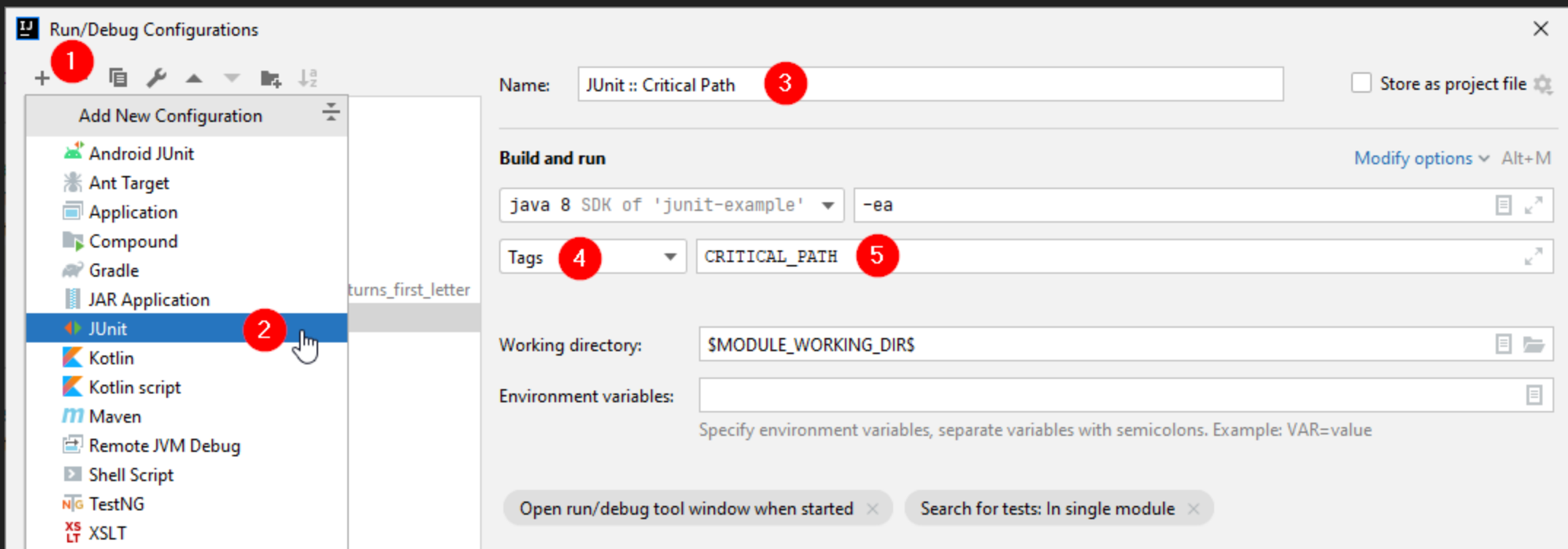
Testy możemy uruchomić w trybie `Run` lub `Debug` — w którym będziemy mogli używać debuggera i w testach, i w samym kodzie, co jest jedną z wielu zalet testów jednostkowych.

Możemy też przygotować konfigurację uruchamiającą wszystkie testy w paczce (*package*), lokalizacji lub określone wybranymi tagami — ta opcja bywa bardzo przydatna.

Żeby utworzyć taką konfigurację, musimy rozwinąć listę znajdującą się obok przycisków uruchamiających projekt (*Run*, *Debug* itp.) i wybrać opcję *Edit Configurations...*:



Następnie wybieramy plusik (1) i *JUnit* z wyświetlonej listy (2). Podajemy czytelną dla nas nazwę konfiguracji (3) oraz wybieramy opcję *Tags*. Na koniec określamy, które tagi powinny zostać uruchomione w tej konfiguracji (5):



Tak przygotowana konfiguracja uruchomi tylko testy oznaczone adnotacją `@Tag("CRITICAL_PATH")`:

```
class StringUtilsTest {  
  
    private final StringUtils stringUtils = new StringUtils();  
  
    @Test  
    void getFirstLetter_returns_first_letter() {...}  
  
    @Test  
    @Tag("CRITICAL_PATH")  
    void getFirstLetter_returns_empty_string_when_given_empty_string() {...}  
}
```

W powyższym przykładzie tylko drugi test zostanie uruchomiony.

Uwaga: Dość łatwo możemy przygotować własną adnotację `@CriticalPath`, żeby nie pisać ciągle `@Tag("CRITICAL_PATH")` — [dokumentacja](#).



## Ćwiczenie nr 1

1. Dodaj do klasy `StringUtils` metodę `getLastLetter(String s)`, która zwróci ostatnią literę podanego w argumencie słowa.
2. Dodaj do klasy `StringUtilsTest` testy sprawdzające:
  - czy metoda zadziała dla *poprawnych* argumentów, np. dla słowa "Java" zwróci "a",
  - czy metoda zadziała dla argumentów `null` oraz `""`.
3. Uruchom wszystkie testy znajdujące się w `StringUtilsTest`.

**@DisplayName** — pozwala napisać dowolną (włączając polskie znaki i emoji) nazwę do wyświetlenia w IDE i raportach

```
@Test
@DisplayName("Metoda `getFirstLetter` zwraca pierwszą literę słowa 🔥")
void getFirstLetter_returns_first_letter() {
    String s = stringUtils.getFirstLetter("Test");

    assertEquals(s, "T");
}
```

✓ Test Results	55 ms
✓ StringUtilsTest	55 ms
✓ Metoda `getFirstLetter` zwraca pierwszą literę słowa 🔥	55 ms

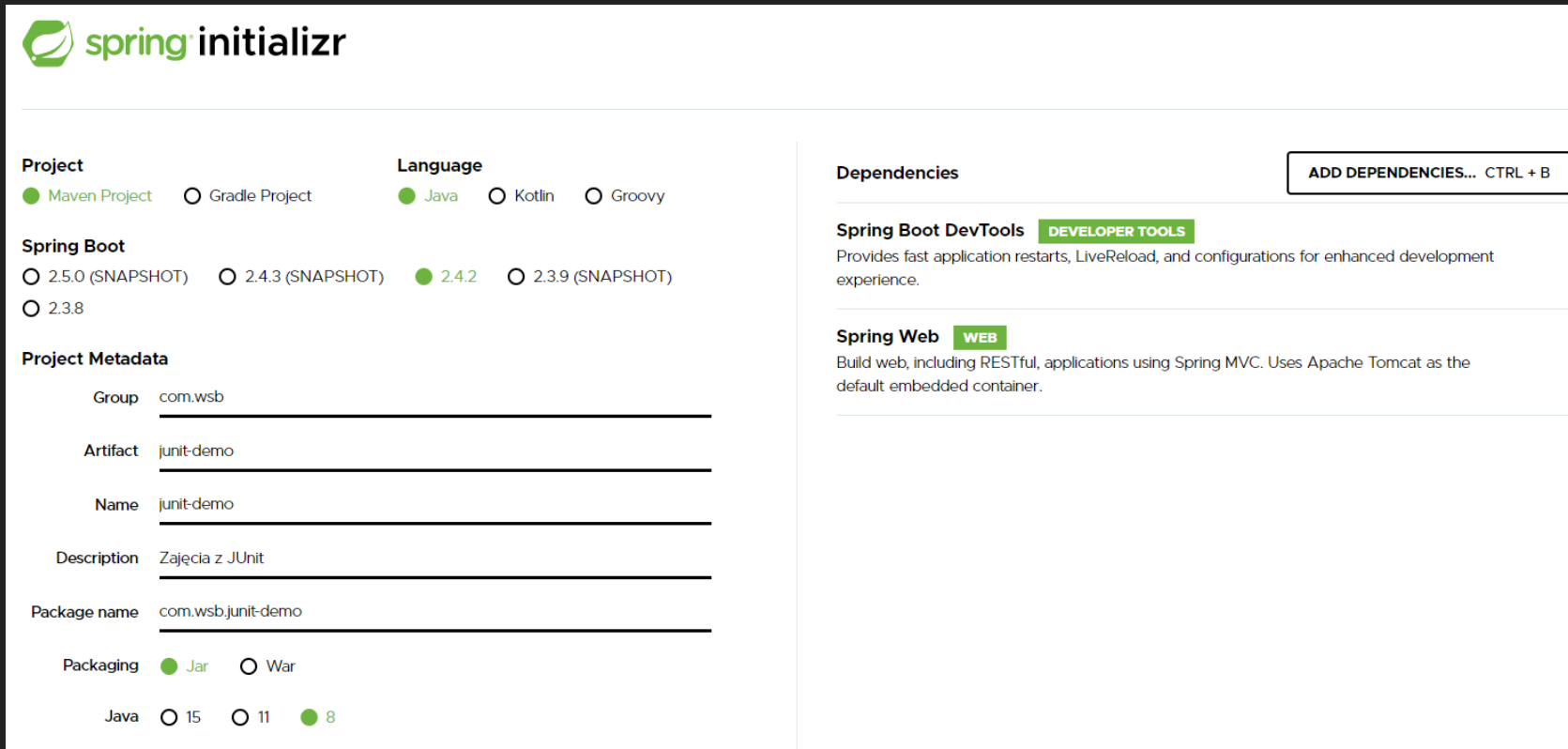
`@DisplayNameGeneration` — pozwala wyświetlić nazwę testu zamieniając podkreślniki na spację, co jeszcze ułatwi jej czytanie:

```
@DisplayNameGeneration(DisplayNameGenerator.ReplaceUnderscores.class)
class StringUtilsTest {

    @Test
    void getFirstLetter_returns_empty_string_when_given_empty_string() {
        ...
    }
}
```

⌵	❗ Test Results	64 ms
⌵	❗ StringUtilsTest	64 ms
	❗ getFirstLetter returns empty string when given empty string	54 ms
	✅ Metoda `getFirstLetter` zwraca pierwszą literę słowa 🔥	10 ms

**Spring Initializr** pozwala utworzyć szablon projektu — dodanie zależności *Spring Web* pozwoli nam wykonywać testy w JUnit.



The image shows the Spring Initializr web form. It is divided into several sections: Project, Language, Spring Boot, Project Metadata, Dependencies, and a button to add more dependencies.

**Project**

☒ Maven Project ☐ Gradle Project

**Language**

☒ Java ☐ Kotlin ☐ Groovy

**Spring Boot**

☐ 2.5.0 (SNAPSHOT) ☐ 2.4.3 (SNAPSHOT) ☒ 2.4.2 ☐ 2.3.9 (SNAPSHOT)

☐ 2.3.8

**Project Metadata**

Group: com.wsb

Artifact: junit-demo

Name: junit-demo

Description: Zajęcia z JUnit

Package name: com.wsb.junit-demo

Packaging: ☒ Jar ☐ War

Java: ☐ 15 ☐ 11 ☒ 8

**Dependencies**

**Spring Boot DevTools** **DEVELOPER TOOLS**

Provides fast application restarts, LiveReload, and configurations for enhanced development experience.

**Spring Web** **WEB**

Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

**ADD DEPENDENCIES... CTRL + B**

## Prosta akcja REST:

```
public class Hello {  
  
    private String name;  
    private String content;  
  
    public Hello(String name) {  
        this.name = name;  
        this.content = "Hello, " + name;  
    }  
}
```

```
@RestController  
public class HelloController {  
  
    @GetMapping("/hello")  
    public Hello hello(@RequestParam(value = "name", defaultValue = "WSB") String name) {  
        return new Hello(name);  
    }  
}
```

Test jednostkowy sprawdzający poprawność odpowiedzi:

```
@WebMvcTest(HelloController.class)
class HelloControllerTest {

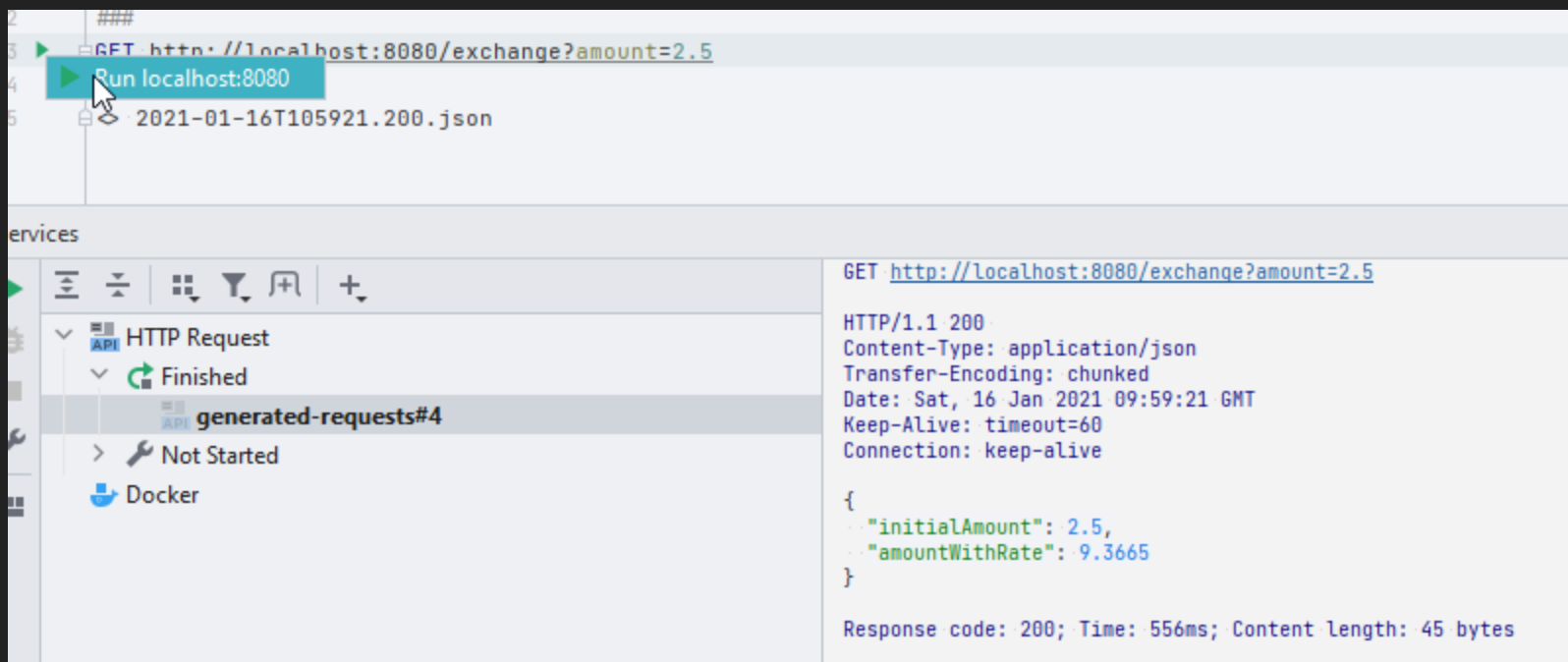
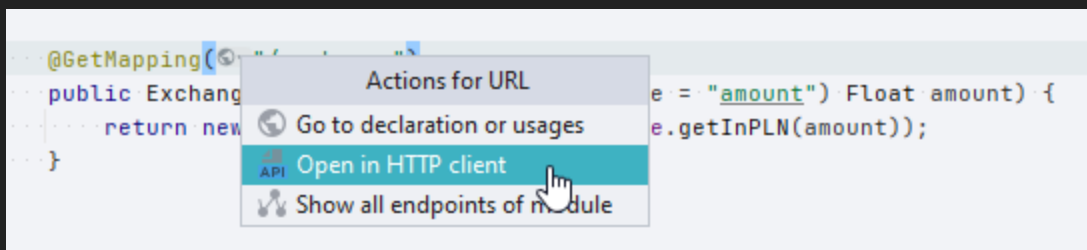
    @Autowired
    private MockMvc mockMvc;

    @Test
    void hello_responds_with_correct_answer_for_params() throws Exception {
        String name = "JUnit";

        this.mockMvc.perform(get("/hello")
            .param("name", name))
            .andExpect(status().isOk())
            .andExpect(jsonPath("$.content")
                .value("Hello, " + name));
    }
}
```

Repozytorium z projektem: <https://github.com/pawel-stan/junit-spring-boot-demo>

## Testujemy REST w IntelliJ





## Artykuły i materiały on-line

- <https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices> — pisane z myślą o `C#`, ale mają sens także dla `Java` ;
- <https://devstyle.pl/2020/06/25/mega-pigula-wiedzy-o-testach-jednostkowych/> — słabe żarty, ale jest to niezłe i zwięzłe omówienie testowania; po polsku;
- <https://enterprisecraftsmanship.com/posts/you-naming-tests-wrong/> — o nazewnictwie;
- <https://www.infoq.com/articles/JUnit-5-Early-Test-Drive/> — zwięzłe wprowadzenie do JUnit 5;

- <https://junit.org/junit5/docs/current/user-guide/> — oficjalna instrukcja JUnit 5 jest porządna,
- <https://semaphoreci.com/community/tutorials/stubbing-and-mocking-with-mockito-2-and-junit> — ładny, obszerny artykuł o Mockito, Junit 5 i dobrych praktykach pisania testów.
- <https://youtu.be/i5Qu3qYOfsM> — o pisaniu testów w Spocku (popularna alternatywa dla Mockito) od twórców IntelliJ.

## Książki za pieniądze

- Roy Osherove — *The Art of Unit Testing*
- Kent Beck — *Test-driven Development: By Example*