Java EE – dzień 1

v3.1

Plan

1. Czym jest Maven?
2. Komunikacja http
3. Serwer
4. Projekt Maven
5. Servlety – wprowadzenie
6. Servlety

Czym jest Maven?

Maven

Maven – najogólniej mówiąc – służy do zarządzania projektem programistycznym na platformie Java.

Wspomaga programistę na każdym etapie tworzenia oprogramowania:

- utworzenie projektu,
- testowanie,
- tworzenie dokumentacji,
- zarządzanie zależnościami,
- kompilowanie,
- budowanie,
- deploy projektu w repozytorium.

Z funkcjonalności **Mavena** można korzystać poprzez konsolę systemu jak i wtyczki dostępne dla większości popularnych środowisk programistycznych (IDE), takich jak **IntelliJ IDEA**, **NetBeans**, czy **Eclipse**.

Instalacja

Instalacja **Apache Maven** jest bardzo prosta. Wystarczy pobrać aktualną wersję ze strony projektu:

https://maven.apache.org

i rozpakować na dysku lokalnym komputera.

Następnie trzeba zaktualizować zmienną **PATH**, aby wskazywała na katalog **bin** w katalogu głównym Mavena.

Weryfikacja instalacji

Aby zweryfikować, czy instalacja przebiegła poprawnie, należy wpisać w konsoli polecenie:

```
mvn -version
```

lub w wersji skróconej:

```
mvn -v
```

W odpowiedzi powinniśmy otrzymać informację na temat zainstalowanej wersji Mavena:

```
Apache Maven <wersja> ...
```

Jeśli Twoje oprogramowanie do kursu było instalowane z naszego skryptu, to masz już Mavena. Możesz dla pewności wpisać w konsoli powyższe polecenia weryfikujące.

Folder .m2

- Jeśli instalacja Mavena przebiegła poprawnie, w katalogu domowym użytkownika zostanie utworzony folder **`.m2`**
- Przechowywane są w nim pliki konfiguracyjne i repozytorium bibliotek (znajdują się tam zarówno biblioteki **`jar`** pobrane z internetu, jak i utworzone z naszych modułów).
- Z repozytorium lokalnego (znajdującego się w katalogu **`.m2`**) pobierane są wszystkie zależności potrzebne do budowania naszych projektów.
- Jeśli jakiejś biblioteki nie ma w repozytorium lokalnym, Maven szuka jej w repozytorium zewnętrznym, następnie pobiera i instaluje ją w repozytorium lokalnym.

Folder .m2

Po pewnym czasie pracy z Mavenem może się okazać, że repozytorium lokalne będzie zajmowało dużo przestrzeni dyskowej, można wtedy zmienić jego lokalizację.

Zmiany tej można dokonać poprzez modyfikację pliku konfiguracyjnego **settings.xml**, znajdującego się w folderze **conf**, w katalogu instalacyjnym Mavena.

```
<localrepository>/path/to/local/repo</localrepository>
```

Maven – podstawowe pojęcia

Plugin

Jest to wtyczka rozszerzająca możliwości Mavena, wykorzystywana do zrealizowania określonego celu.

Najczęściej wykorzystywane wtyczki:

- archetype,
- jar,
- war,
- javadoc.

Listę pluginów znajdziesz tutaj:

https://maven.apache.org/plugins/

Wywoływanie celów zdefiniowanych w pluginach schematycznie wygląda następująco:

```
mvn [plugin]:[goal]
```

- **plugin** to nazwa wtyczki
- **goal** to wywoływany cel.

Przykład dla pluginu **exec**:

```
mvn clean compile exec:java
```

http://www.mojohaus.org/exec-maven-plugin/

Podstawowe pojęcia

Artefakt

Jest to unikalna nazwa identyfikująca dany projekt w **Grupie** (najczęściej nazwa projektu).

Grupa

To określenie przestrzeni nazw w jakiej znajduje się **Artefakt**.

Grupa powinna mieć nazwę utworzoną zgodnie z zasadami **JavaBean** dotyczącej nazw pakietów (np. **pl.coderslab**).

Wersja

Wersja aplikacji – najpopularniejsza konwencja składa się z oznaczenia wersji:

Major-Minor-Incremental-Kwalifikator (np. 1.0.0-SNAPSHOT lub np. 2.2.4-Final).

Tworzenie projektu

Projekt można utworzyć wpisując w konsoli polecenie:

```
 mvn archetype:generate
-DgroupId=pl.coderslab
-DartifactId=my-app
-DarchetypeArtifactId=maven-archetype-quickstart
-DinteractiveMode=false
```

Polecenie to wykorzystuje plugin o nazwie **archetype**.

Tworzy on szkielet aplikacji na podstawie określonego archetypu, w tym wypadku:

maven-archetype-quickstart.

Uwaga: Polecenie musi być w jednej linii – poszczególne elementy komendy oddzielamy spacjami.

Tworzenie projektu

Znaczenie poszczególnych elementów polecenia:

```
mvn archetype:generate
-DgroupId=pl.coderslab
-DartifactId=my-app
-DarchetypeArtifactId=
     maven-archetype-quickstart
-DinteractiveMode=false
```

Więcej o tym pluginie: http://maven.apache.org/archetype/maven-archetype-plugin/

Tworzenie projektu

Znaczenie poszczególnych elementów polecenia:

```
mvn archetype:generate
-DgroupId=pl.coderslab
-DartifactId=my-app
-DarchetypeArtifactId=
     maven-archetype-quickstart
-DinteractiveMode=false
```

Wykorzystanie pluginu **archetype** z określeniem celu **generate**.

Więcej o tym pluginie: http://maven.apache.org/archetype/maven-archetype-plugin/

Tworzenie projektu

Znaczenie poszczególnych elementów polecenia:

```
mvn archetype:generate
-DgroupId=pl.coderslab
-DartifactId=my-app
-DarchetypeArtifactId=
     maven-archetype-quickstart
-DinteractiveMode=false
```

Wykorzystanie pluginu **archetype** z określeniem celu **generate**.

Grupa

Więcej o tym pluginie: http://maven.apache.org/archetype/maven-archetype-plugin/

Tworzenie projektu

Znaczenie poszczególnych elementów polecenia:

```
mvn archetype:generate
-DgroupId=pl.coderslab
-DartifactId=my-app
-DarchetypeArtifactId=
     maven-archetype-quickstart
-DinteractiveMode=false
```

Wykorzystanie pluginu **archetype** z określeniem celu **generate**.

Grupa

Artefakt

Więcej o tym pluginie: http://maven.apache.org/archetype/maven-archetype-plugin/

Tworzenie projektu

Znaczenie poszczególnych elementów polecenia:

```
mvn archetype:generate
-DgroupId=pl.coderslab
-DartifactId=my-app
-DarchetypeArtifactId=
    maven-archetype-quickstart
-DinteractiveMode=false
```

Wykorzystanie pluginu **archetype** z określeniem celu **generate**.

Grupa

Artefakt

Archetyp

Więcej o tym pluginie: http://maven.apache.org/archetype/maven-archetype-plugin/

Tworzenie projektu

Znaczenie poszczególnych elementów polecenia:

```
mvn archetype:generate
-DgroupId=pl.coderslab
-DartifactId=my-app
-DarchetypeArtifactId=
     maven-archetype-quickstart
-DinteractiveMode=false
```

Wykorzystanie pluginu **archetype** z określeniem celu **generate**.

Grupa

Artefakt

Archetyp

Oznacza, że polecenie będzie wykonane w trybie **batch** (zostaną ustawione opcje domyślne, użytkownik nie będzie pytany o nic w trakcie wykonywania polecenia).

Więcej o tym pluginie: http://maven.apache.org/archetype/maven-archetype-plugin/

Tworzenie projektu

W naszym wypadku powstanie projekt (artefakt) o następującej strukturze:

```
my-app
----| pom.xml
----| src
--------| main
------------| java
----------------| pl
--------------------| coderslab
------------------------| App.java
--------| test
------------| java
----------------| pl
--------------------| coderslab
------------------------| AppTest.java
```

pom.xml

Podczas zajęć skupimy się głównie na wykorzystaniu Mavena jako zarządcy zależności, warto jednak pamiętać że ma on większe możliwości.

"Sercem" projektu Mavenowego jest plik **pom.xml**. To w nim znajduje się konfiguracja projektu.

W pliku **pom.xml** definiowane są wszystkie zależności, sposób budowania, testowania i uruchamiania projektu, a także (m.in.) sposób generowania dokumentacji.

Na kolejnym slajdzie został umieszczony kod z pliku **pom.xml**, jaki utworzył się po wywołaniu w konsoli polecenia generującego projekt Maven.

```xml
<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/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>pl.coderslab</groupId>
  <artifactId>my-app</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>my-app</name>
  <url>http://maven.apache.org</url>
  <dependencies>
      <dependency>
          <groupId>junit</groupId>
          <artifactId>junit</artifactId>
          <version>3.8.1</version>
          <scope>test</scope>
      </dependency>
  </dependencies>
</project>
```

Budowanie projektu

Budowanie projektu, to wykonywanie określonych celów (targets).

Aby skompilować utworzony projekt, wystarczy z linii poleceń, z poziomu katalogu głównego aplikacji (tam gdzie znajduje się plik pom.xml), wykonać polecenie:

```
mvn compile
```

Jeśli projekt wygenerował się prawidłowo, po kilku chwilach na ekranie powinien pojawić się komunikat potwierdzający poprawne zbudowanie projektu.

```
[INFO] ------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------
[INFO] Total time: 0.923 s
[INFO] Finished at: 2018-08-25T19:30:53+02:00
[INFO] Final Memory: 13M/241M
[INFO] ------------------------------------
```

Budowanie projektu

Po wykonaniu kompilacji kodów źródłowych, w katalogu projektu pojawi się folder **target**, w którym zostaną umieszczone skompilowane pliki (`.class`).

Wywołanie poszczególnych faz można łączyć np.:

```
mvn clean compile
```

Komenda spowoduje wyczyszczenie projektu i kompilację źródeł.

Cykl życia projektu

Cykl życia jest to zestaw standardowych, precyzyjnie zdefiniowanych i wykonywanych w określonej kolejności faz.

Błąd na którymkolwiek etapie zatrzymuje wykonywanie kolejnych etapów.

Maven definiuje trzy cykle życia projektu:

- **default** – cykl budowy projektu,
- **clean** – cykl czyszczenia projektu,
- **site** – cykl tworzenia stron z dokumentacją projektu.

Opis faz w poszczególnych cyklach:

http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html#Lifecycle_Reference

Cykl życia projektu

Cykl default składa się z następujących faz:

- **validate** – sprawdza poprawność projektu,
- **compile** – kompiluje kod źródłowy,
- **test** – wykonuje testy jednostkowe,
- **package** – pakuje skompilowany kod w paczki dystrybucyjne (np. jar, war),
- **verify** – sprawdza poprawność paczki,
- **install** – umieszcza paczkę w lokalnym repozytorium, aby mogła być używana jako zależność przez inne moduły,
- **deploy** – umieszcza (publikuje) paczkę w zdalnym repozytorium.

Zależności

Zarządzanie zależnościami projektu bywa uciążliwe nawet w przypadku niezbyt rozbudowanych projektów.

Do tej pory załączaliśmy biblioteki korzystając z IntelliJ (np. sterownik do bazy danych), jednak Maven ze swoim systemem zarządzania zależnościami znacznie to ułatwia.

Aby dodać zależność do projektu, wystarczy w pliku **pom.xml** dodać element **<dependency>** i określić w nim dostawcę (**groupId**), nazwę modułu (**artifactId**), wersję (**version**) i zasięg (**scope**).

Dla przykładu, w utworzonym wcześniej testowym projekcie mamy wpisaną zależność do Junit, czyli do narzędzia do tworzenia testów jednostkowych:

```
<dependencies>
    <dependency>
        <groupid>junit</groupid>
        <artifactid>junit</artifactid>
        <version>3.8.1</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Zależności

Maven w trakcie budowania automatycznie pobierze z repozytorium bibliotekę w odpowiedniej wersji i dołączy ją do projektu.

W łatwy sposób można podnieść wersję używanej biblioteki przez modyfikację wartości w elemencie **<version>**.

Zamiast określać konkretną wersję biblioteki, możemy określić zakres przy użyciu nawiasów, np.:

```
<groupid>junit</groupid>
<artifactid>junit</artifactid>
<version>[3.8.1,]</version>
<scope>test</scope>
```

Zapis taki mówi, że projekt będzie korzystał z biblioteki **JUnit** w wersji **3.8.1** lub nowszej.

Repozytorium Mavena

W głównym repozytorium Mavena można znaleźć praktycznie wszystkie publicznie dostępne biblioteki Javy:

https://mvnrepository.com/

Jeśli czegoś tam nie znajdziemy, to możemy z dużym prawdopodobieństwem założyć, że coś czego szukamy nie istnieje.

Integracja z IDE

Korzystanie z Mavena za pomocą konsoli jest możliwe, ale bywa uciążliwe. Na szczęście większość środowisk programistycznych posiada wsparcie wbudowane lub dostępne za pomocą wtyczek.

IntelliJ jest narzędziem, które posiada wsparcie wbudowane w platformę.

Alternatywa dla Mavena

Alternatywą dla Mavena jest jego młodszy odpowiednik **Gradle**.

Mimo rosnącej popularności **Gradle** wśród developerów, podczas kursu wykorzystujemy **Mavena**, gdyż jest to utrwalony standard wykorzystywany przy większości istniejących projektów.

Więcej informacji na stronie projektu:

https://gradle.org/

Zadania

Komunikacja HTTP

Protokół HTTP

HTTP to protokół, który służy do przekazywania danych między komputerami.

- Jeden z nich jest serwerem, czyli komputerem, na którym wykonywane jest oprogramowanie back-end i generowane są strony.

- Drugi z komputerów jest klientem, czyli maszyną, która generuje żądania do serwera, odbiera je i wyświetla na ekranie (w przeglądarce) wynik działania aplikacji.

Następuje tutaj komunikacja podobna do rozmowy dwojga osób, z tym zastrzeżeniem, że scenariusz tej rozmowy i zasady są ściśle określone.

Request & response

Jak to działa?

1. **`klient`** nawiązuje połączenie z serwerem,
2. **`klient`** wysyła żądanie (**`HTTP Request`**), a serwer je odbiera,
3. **`serwer`** przetwarza żądanie i generuje odpowiedź (**`HTTP Response`**),
4. **`serwer`** wysyła odpowiedź do **`klienta`**,
5. połączenie jest zamykane,
6. **`klient`** wyświetla wynik na ekranie.

Metody HTTP

Protokół **HTTP** przenosi dane między komputerami używając różnych metod:

- **GET**
- **POST**
- **HEAD**
- **PUT**
- **DELETE**
- **OPTIONS**
- **TRACE**
- **CONNECT**
- **PATCH**

Najczęściej używanymi metodami HTTP w Javie są metody:

- **GET**
- **POST**

Są one powszechnie używane w aplikacjach internetowych, jako metody do przekazywania danych między przeglądarką a serwerem.

Serwer

Serwer aplikacji

Serwer aplikacji – oprogramowanie służące do uruchomienia aplikacji webowych.

Podstawową jego funkcjonalnością, z której będziemy korzystali, jest zapewnienie obsługi protokołu **HTTP**.

Istnieje wiele serwerów na których możemy uruchamiać nasze aplikacje, najpopularniejsze z nich to:

- **WildFly** (**JBoss AS**)
- **Glassfish**
- **WebSphere**
- **Apache TomEE**

Kontener servletów

Aplikacje uruchamiane są na serwerze zgodnym ze specyfikacją **Javy EE**.

Podczas zajęć będziemy korzystać z narzędzia **Tomcat** – nie wspiera on całkowicie wspomnianej specyfikacji i z tego względu określamy go mianem **kontenera servletów**.

Serwery aplikacji również zawierają w sobie kontener servletów.

Tomcat jest oprogramowaniem **open source**, co oznacza, że jest dostępny nieodpłatnie, zarówno do użytku domowego, jak i komercyjnego.

Tomcat

Uruchomienie aplikacji oznacza zainstalowanie jej (umieszczenie plików w określonym miejscu) na serwerze aplikacji.

Mówimy też że robimy deploy aplikacji na serwerze (ang. **deployment**).

Więcej informacji na temat **Tomcata** znajdziesz na jego oficjalnej stronie:

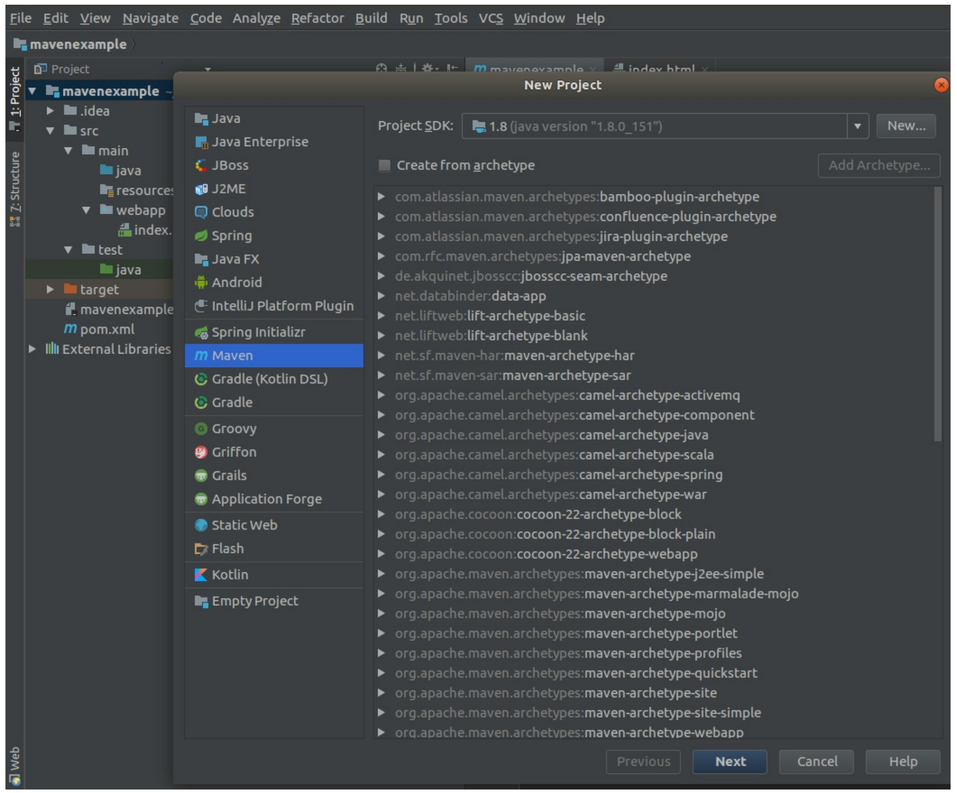
http://tomcat.apache.org/

Projekt Maven

Maven projekt

W celu utworzenia projektu z wykorzystaniem Mavena należy wybrać opcję:

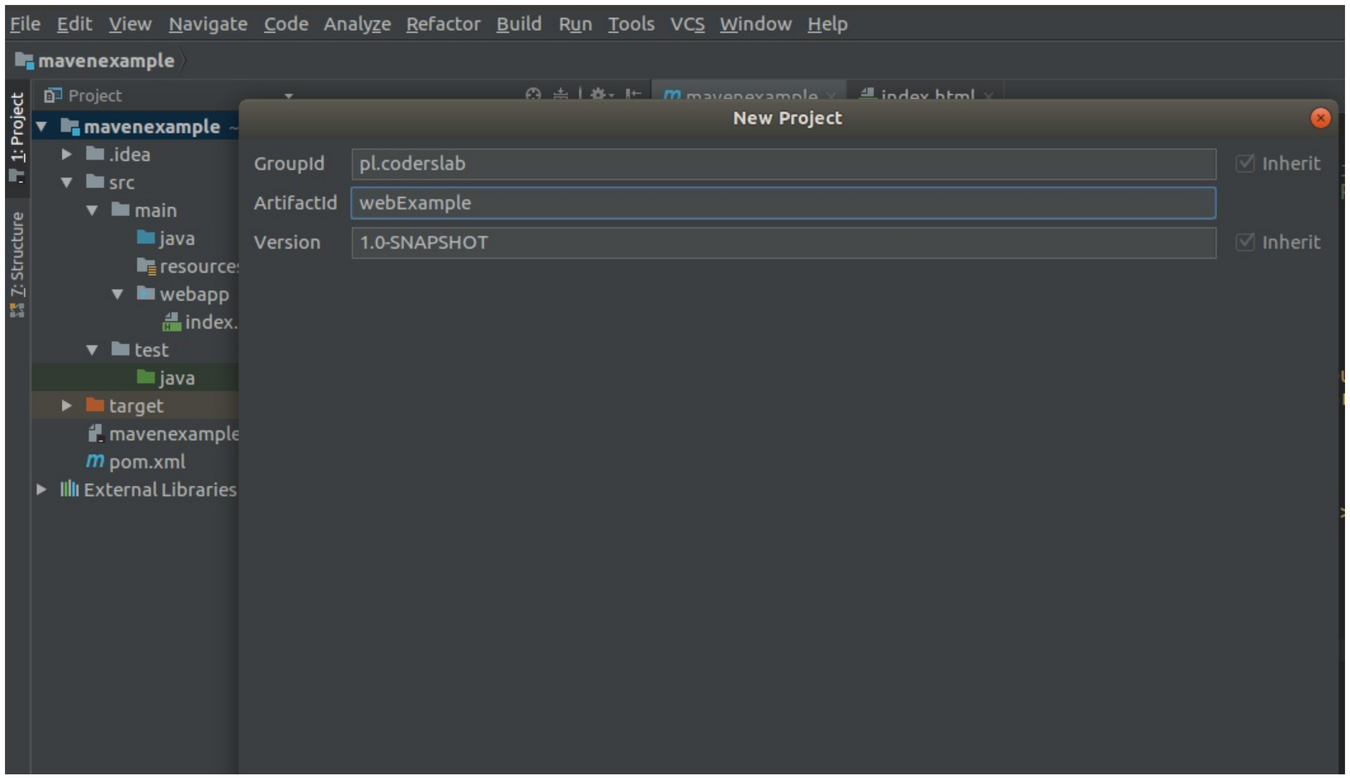
New –> Project –> Maven



Klikamy **Next**.

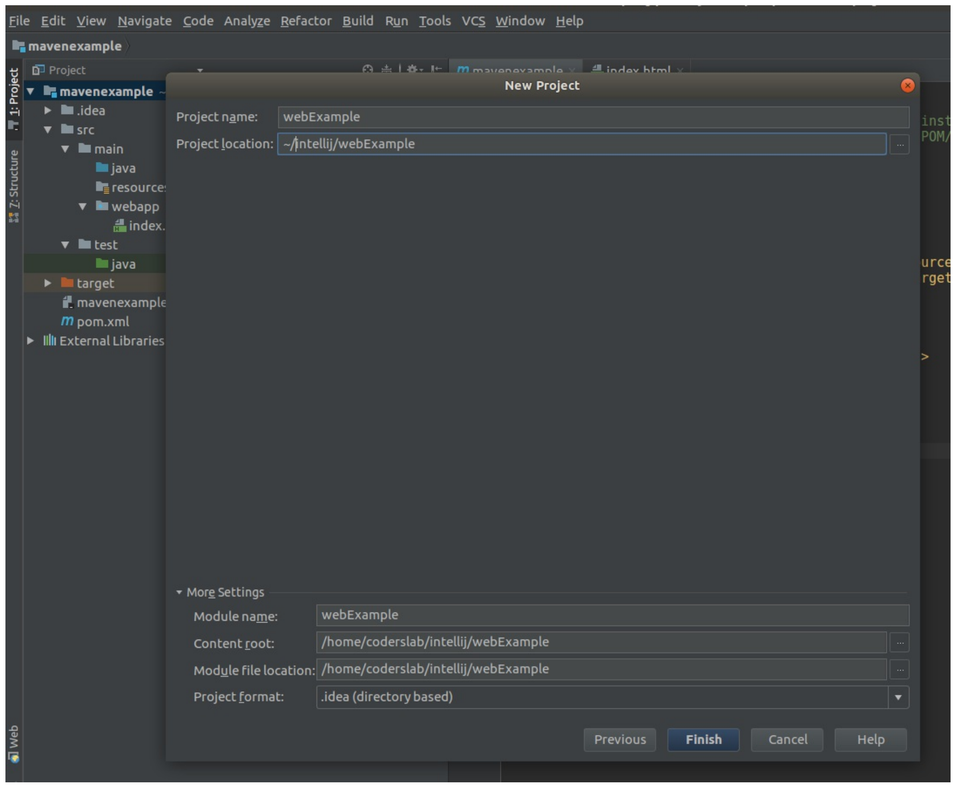
Maven projekt

W kolejnym oknie wpisujemy nazwę naszego projektu i zatwierdzamy klikając **Next**.



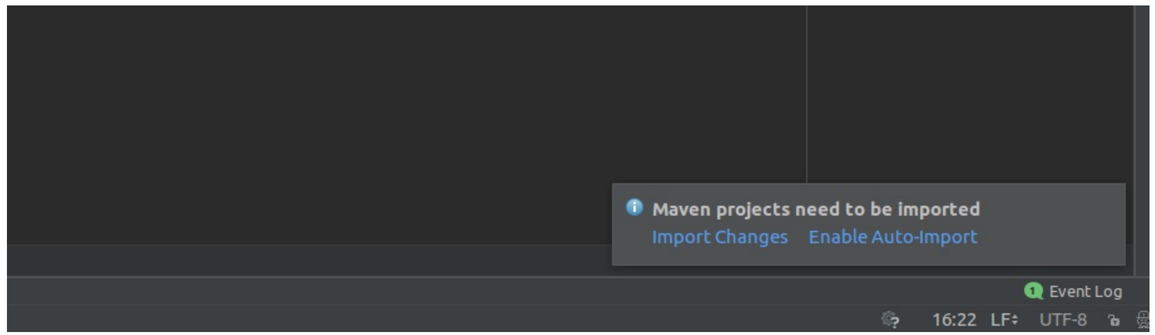
Maven projekt

Przechodzimy do następnego okna, w którym wybieramy lokalizację dla projektu i zatwierdzamy utworzenie projektu przyciskiem **Finish**.



Maven projekt

Po utworzeniu projektu, w prawym dolnym rogu pojawi się komunikat:



Zaznaczamy opcję **Enable Auto-Import**.

Musimy jeszcze dodać katalog, w którym będziemy umieszczać pliki **JSP**.

W strukturze projektu przechodzimy do folderu:

src –> main

i tam tworzymy katalog o nazwie: **webapp**.

Maven projekt

Kolejny etap, to dodanie ustawień kompilatora w pliku **pom.xml**:

```
<properties>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
</properties>
```

oraz sposobu tworzenia archiwum:

```
<packaging>war</packaging>
```

a następnie dodanie zależności dla servletów:

```
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>3.1.0</version>
    <scope>provided</scope>
</dependency>
```

Plik pom.xml

Na tym i kolejnym slajdzie zobaczysz jak powinien wyglądać cały plik **`pom.xml`** (oba slajdy tworzą wspólnie jeden plik):

```
<?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>pl.coderslab</groupId>
    <artifactId>webExample</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>war</packaging>
    <properties>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
    </properties>
```

Plik pom.xml cd.

```
<dependencies>
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>javax.servlet-api</artifactId>
            <version>3.1.0</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>
</project>
```

Pierwszy plik html

W folderze **webapp** umieścimy plik **index.html**.

Możemy skorzystać z gotowego, prostego szablonu, udostępnionego przez **IntelliJ**:

New –> HTML File

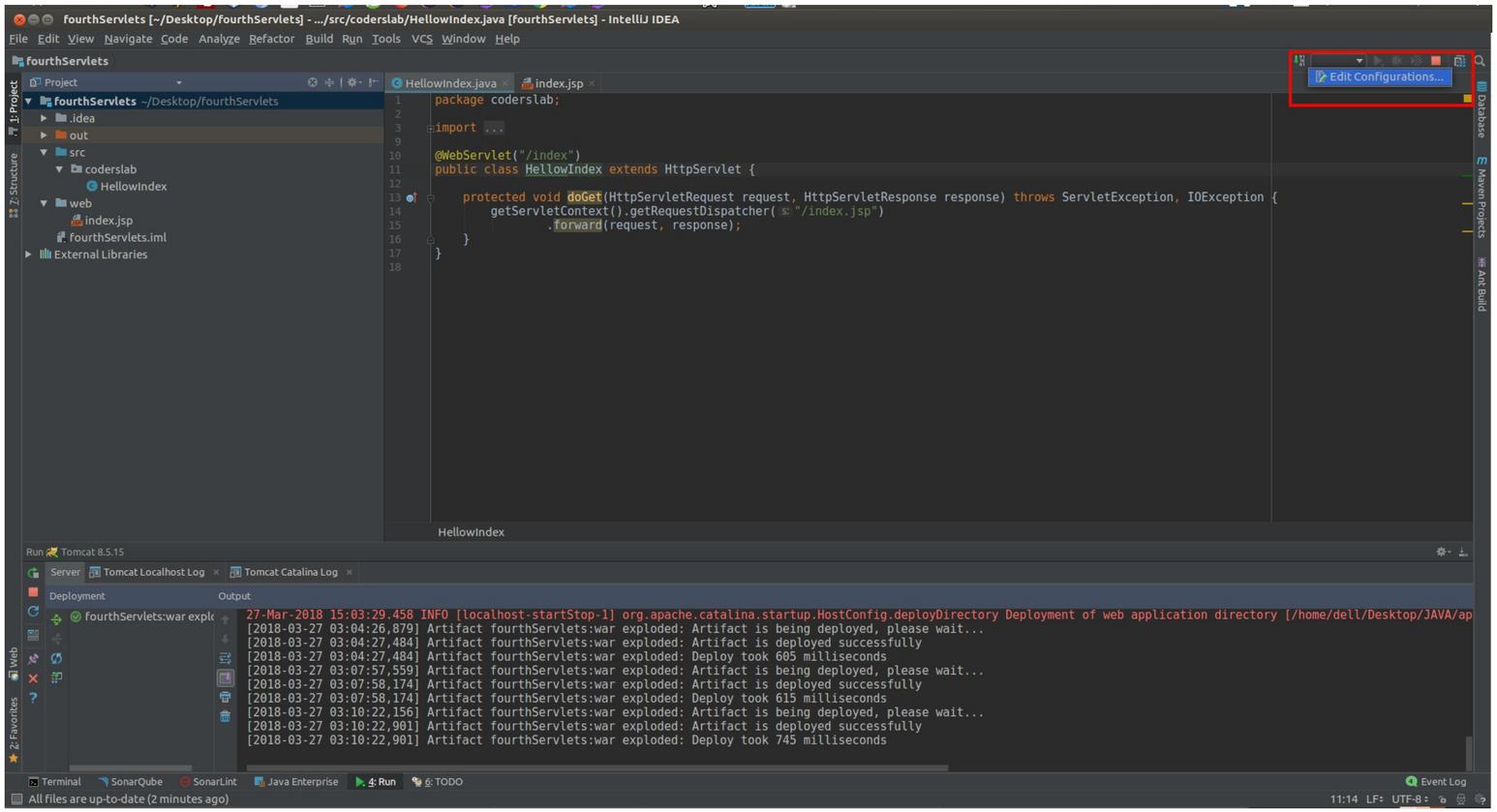
Jest to standardowy plik **html**, znany nam już z materiałów przygotowawczych.

Taki plik możemy otworzyć również bezpośrednio z poziomu przeglądarki.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>
Coders Lab wita w świecie Javy EE!
</title>
</head>
<body>
  <h1>Witaj Java EE!</h1>
  <p>To jest moja pierwsza aplikacja
        napisana w Javie EE.</p>
</body>
</html>
```

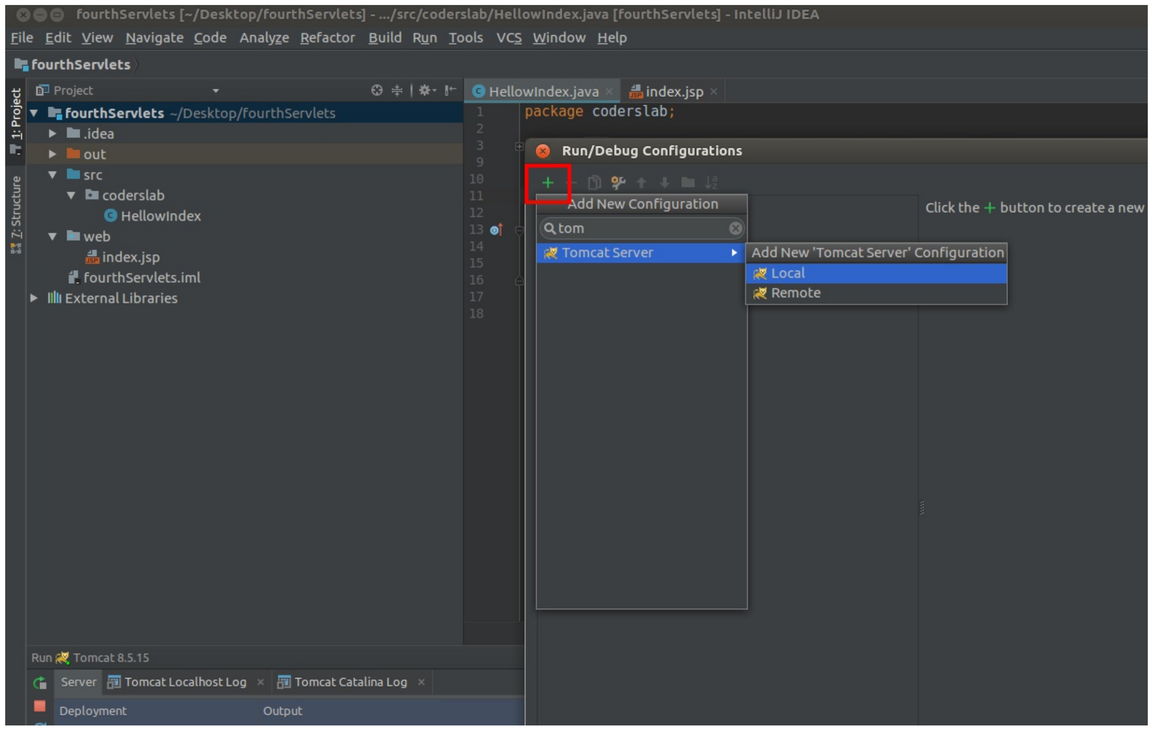
Uruchomienie projektu

Przy pierwszym uruchomieniu projektu musimy wybrać opcję **Edit Configuration**:



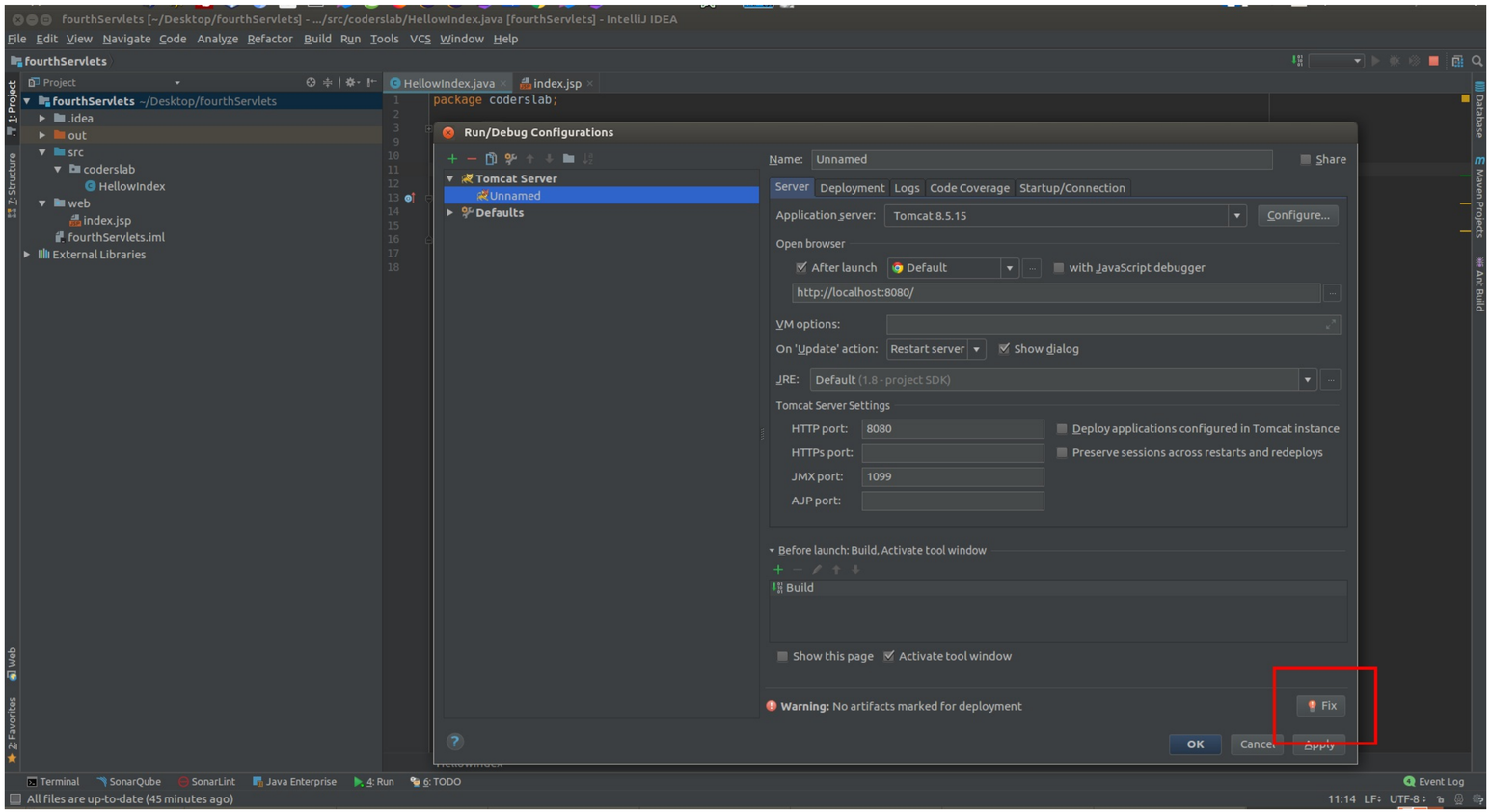
Uruchomienie projektu

Następnie wybieramy zdefiniowany uprzednio serwer, klikając w zielony plus:



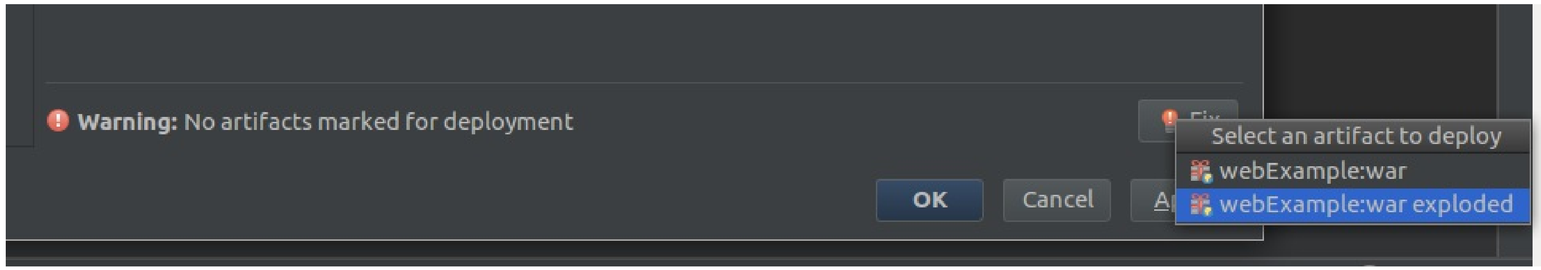
Uruchomienie projektu

Klikamy w przycisk **Fix**:



Uruchomienie projektu

Z listy, która się pojawi wybieramy opcję **war exploded**:



Oznacza to, że **IntelliJ** utworzy połączenie z **Tomcatem**, korzystając z rozpakowanego archiwum, co w praktyce powinno się przełożyć na przyśpieszenie procesu uruchamiania.

Uruchomienie projektu

Następnie w zakładce **Server** zaznaczamy w dwóch oknach:

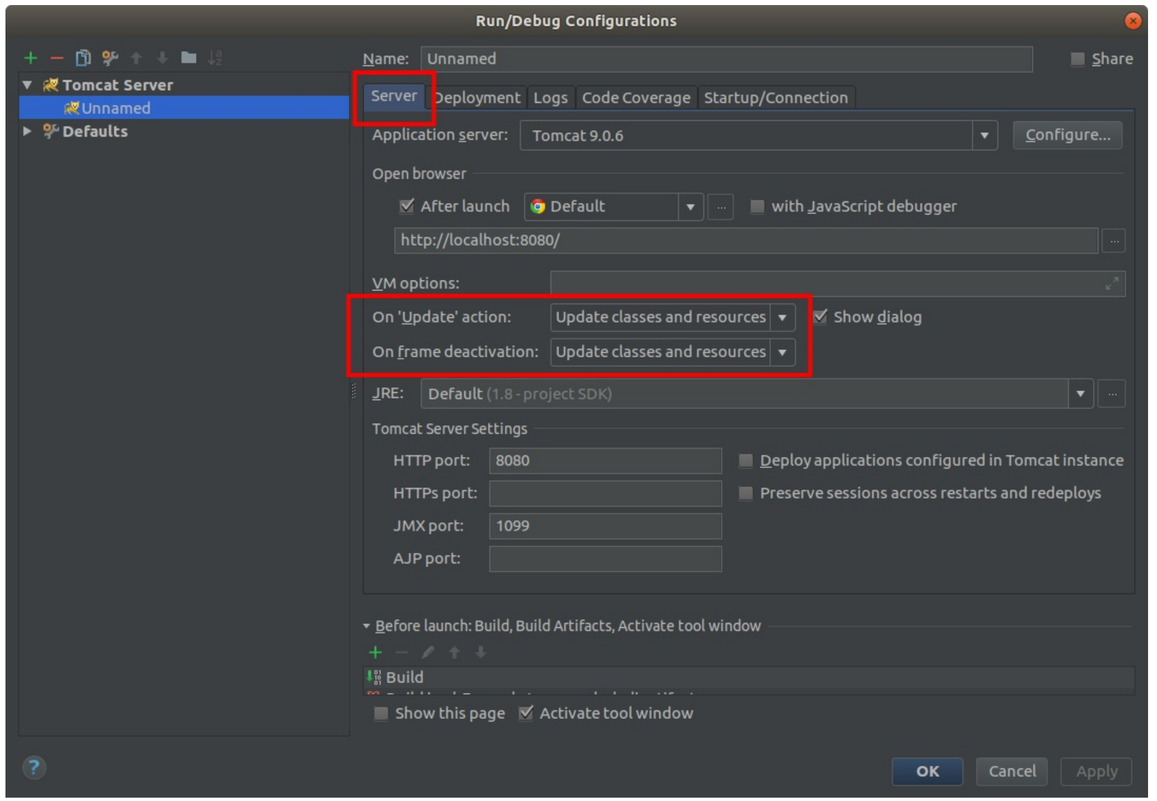
- **On 'Update' action**
- **On frame deactivation**

taką samą opcję:

Update classes and resources.

Dokładny opis tych opcji znajdziemy tutaj:

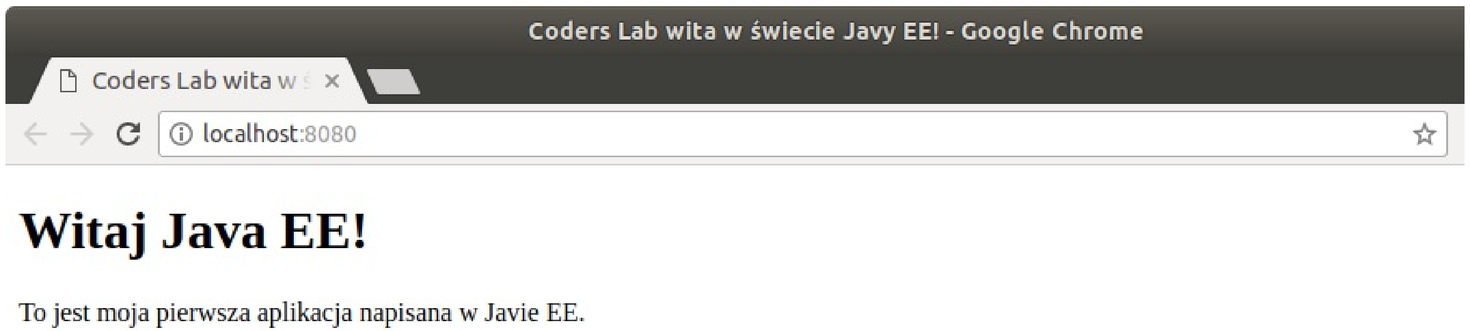
https://www.jetbrains.com/help/idea/updating-applications-on-application-servers.html



Uruchomienie projektu

Teraz możemy już uruchomić projekt, wybierając ikonę zielonej strzałki w prawym górnym rogu. Po poprawnym uruchomieniu serwera otworzy się przeglądarka z naszą stroną.

Przeglądarka Google Chrome:



Uruchomienie projektu

Jeśli podczas uruchomienia otrzymujesz poniższy błąd:

```
Error running 'Unnamed': Cannot run program "/home/dell/Desktop/JAVA/apache-tomcat-9.0.6/bin/catalina.sh" (in directory "/home/dell/Desktop/JAVA/apache-tomcat-9.0.6/bin"): error=13, Permission denied
```

- — przejdź w konsoli do katalogu **bin** rozpakowanego serwera **Tomcat**, a następnie wywołaj polecenie:

  ```
  chmod -R 777 catalina.sh
  ```

Servlety – wprowadzenie

Podstawowe pojęcia

Architektura sieci – jest to sposób przekazu danych pomiędzy urządzeniami.

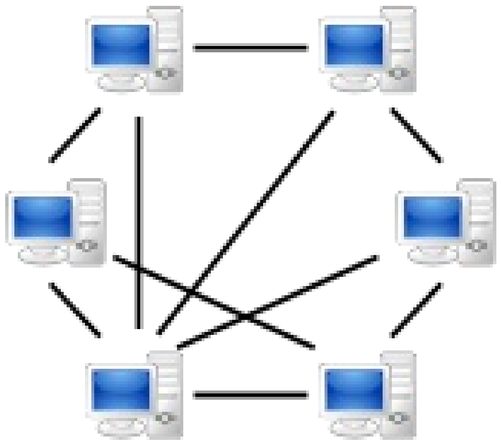
LAN – **Local Area Network** – sieć lokalna – sieć łącząca komputery na określonym obszarze takim jak blok, szkoła, laboratorium, czy też biuro.

WAN – **Wide Area Network** – sieć rozległa znajdująca się na obszarze wykraczającym poza miasto, kraj, kontynent.

Architektura P2P

Peer-To-Peer (P2P) – jest to architektura, w której każdy komputer podłączony do takiej sieci udostępnia część swoich zasobów.

Przykładem wykorzystania takiej architektury jest np. Torrent czy Napster.



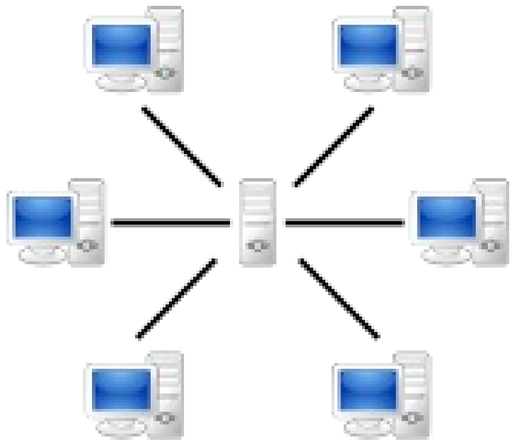
Źródło: https://pl.wikipedia.org/wiki/Peer-to-peer

Architektura klient-serwer

Architektura **klient-serwer**, która opiera się na rozdzieleniu zadań.

Serwer – zajmuje się udostępnianiem informacji.

Klient – wysyła do serwera żądanie
w określonym formacie i oczekuje na odpowiedź.



Źródło: https://pl.wikipedia.org/wiki/Peer-to-peer

Protokół HTTP

HTTP (ang. **Hypertext Transfer Protocol**) – protokół przesyłania dokumentów hipertekstowych to protokół sieci **WWW** (ang. **World Wide Web**).

Źródło: http://web-zlecenia.pl

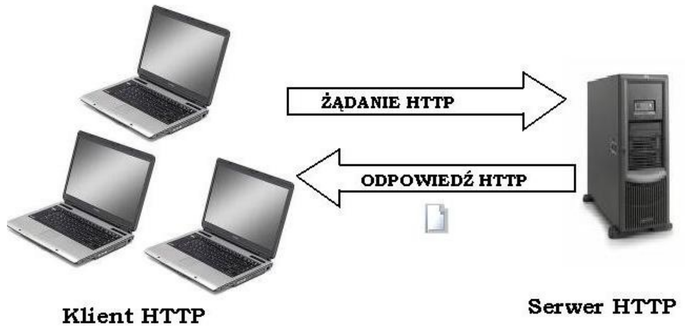
Za pomocą protokołu **HTTP** przesyła się żądania udostępnienia dokumentów WWW i informacje o kliknięciu odnośnika oraz informacje z formularzy. Zadaniem stron WWW jest publikowanie informacji, a umożliwia to właśnie protokół **HTTP**.

Źródło: https://pl.wikipedia.org/wiki/Hypertext_Transfer_Protocol

Protokół HTTP a Java EE

W początkowej fazie nauki zajmiemy się **servletami** – czyli podstawowymi elementami, dzięki którym będziemy mogli tworzyć aplikacje przetwarzające żądania **HTTP**.

W odpowiedzi będziemy wysyłać stronę **html**.



Źródło: https://tp.faculty.wmi.amu.edu.pl

Servlety

Servlety

`Servlet` – informacje podstawowe:

- jest to klasa Javy, którą uruchamia się po stronie serwera,
- zawiera ona metody obsługujące określone metody protokołu HTTP,
- przetwarza żądania i wysyła odpowiedź do użytkownika.

Zastosowanie:

- wyświetlanie stron i aplikacji internetowych.
- udostępnianie interfejsów programistycznych.

Szczegóły specyfikacji servletów:

http://download.oracle.com/otndocs/jcp/servlet-3_1-fr-eval-spec/index.html

Servlety

Servlety tworzymy rozszerzając klasę **`HttpServlet`** z pakietu **`javax.servlet.http`**.

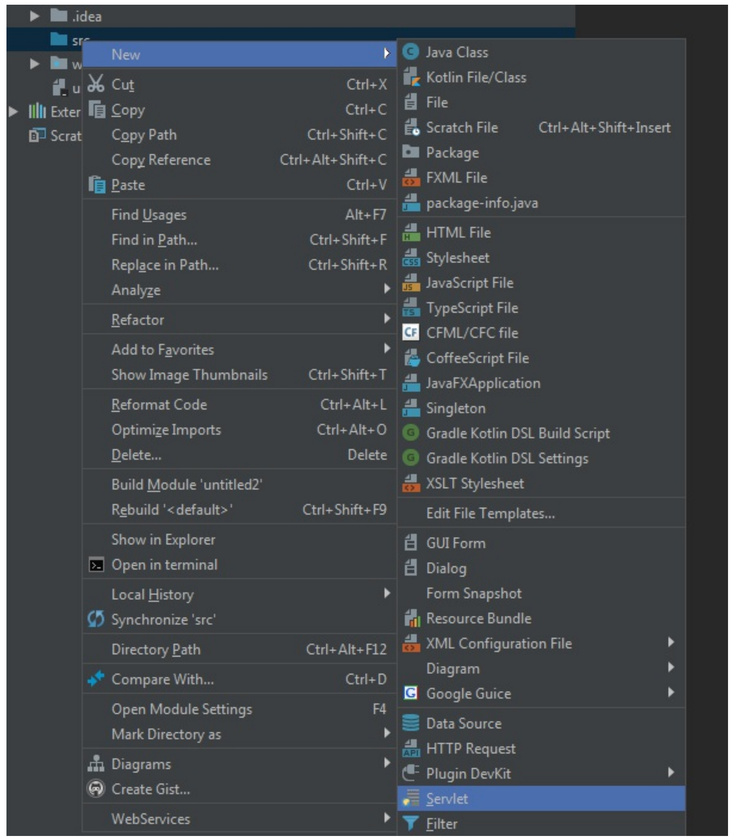
Możemy zatem utworzyć zwykłą klasę Javy, która dziedziczy po klasie **`javax.servlet.http.HttpServlet`**, a następnie dodać metody które chcemy obsługiwać, np. metodę **`doGet()`** – obsługującą żądanie **`GET`**.

Servlety mogą obsługiwać wszystkie dostępne metody protokołu HTTP:

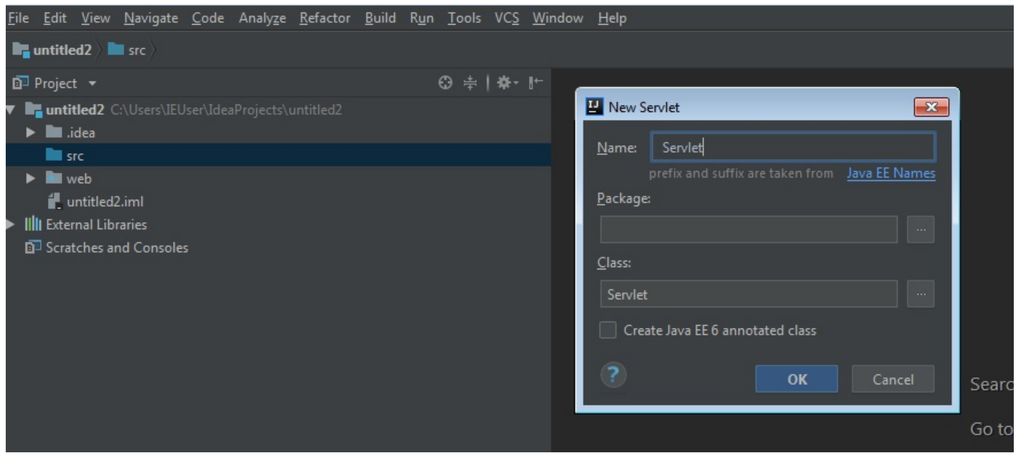
- **`doGet`** – **HTTP GET**
- **`doPost`** – **HTTP POST**
- **`doPut`** – **HTTP PUT**
- **`doDelete`** – **HTTP DELETE**
- **`doHead`** – **HTTP HEAD**
- **`doOptions`** – **HTTP OPTIONS**
- **`doTrace`** – **HTTP TRACE**

Servlety

Dużo wygodniejszą formą tworzenia servletów jest korzystanie z kreatora wbudowanego w IntelliJ. Wybieramy z menu: **New –> Servlet**



Następnie wpisujemy nazwę i zatwierdzamy utworzenie servletu:



Adnotacje

Adnotacja to konstrukcja, która pozwala na przekazywanie dodatkowych informacji na temat kodu.

Adnotacje rozpoczyna znak małpy (" **@**").

Następnie podawana jest nazwa klasy.

Poprzedza się nimi deklaracje klas, pól i metod.

Nad jednym elementem (np. klasa, pole) mamy możliwość dodania więcej niż jedną adnotację.

Przykłady adnotacji:

```
@Component
@Autowired
@GetParam
```

Adnotacje

Adnotacja to konstrukcja, która pozwala na przekazywanie dodatkowych informacji na temat kodu.

Adnotacje rozpoczyna znak małpy (" **@**").

Następnie podawana jest nazwa klasy.

Poprzedza się nimi deklaracje klas, pól i metod.

Nad jednym elementem (np. klasa, pole) mamy możliwość dodania więcej niż jedną adnotację.

Przykłady adnotacji:

```
@Component
@Autowired
@GetParam
```

- Te adnotacje poznamy dokładnie podczas omawiania frameworka **Spring**.

Adnotacja `@WebServlet`

Adnotacja `@WebServlet` pozwala na określenie parametrów servletu (np. nazwy servletu, adresu URL).

Spotkać się można z określeniem **`mapowania adresu na servlet`** – chodzi tutaj o przypisanie, pod jakim adresem URL dostępny jest określony **`servlet`**.
Użycie w tym celu adnotacji **`@WebServlet`** pozwala uniknąć edytowania pliku **`web.xml`**.

`Servlet` stworzony w IntelliJ dodaje automatycznie atrybut
`name` (który domyślnie jest taki, jak nazwa klasy) w adnotacji **`@WebServlet`**, np.:

```
@WebServlet(name = "MyFirstServlet")
```

Do mapowania servletu wystarczy też prostsza konstrukcja tej adnotacji:

```
@WebServlet("/addressURL")
```

Servlety

Przykład:

```
@WebServlet(name = "MyFirstServlet", urlPatterns={"/addressURL"})
public class MyFirstServlet extends HttpServlet {
  protected void doGet(HttpServletRequest request, HttpServletResponse response)
  throws ServletException, IOException {
    response.getWriter().append("<h1>Hello world.</h1>");
  }
}
```

Servlety

Przykład:

```
@WebServlet(name = "MyFirstServlet", urlPatterns={"/addressURL"})
public class MyFirstServlet extends HttpServlet {
  protected void doGet(HttpServletRequest request, HttpServletResponse response)
  throws ServletException, IOException {
    response.getWriter().append("<h1>Hello world.</h1>");
  }
}
```

Za pomocą adnotacji **@WebServlet** wskazujemy adres, pod jakim servlet będzie dostępny.
Można napisać też krótszą wersję:
@WebServlet("/addressURL")

Servlety

Przykład:

```
@WebServlet(name = "MyFirstServlet", urlPatterns={"/addressURL"})
public class MyFirstServlet extends HttpServlet {
  protected void doGet(HttpServletRequest request, HttpServletResponse response)
  throws ServletException, IOException {
    response.getWriter().append("<h1>Hello world.</h1>");
  }
}
```

Metoda **doGet** – zapewnia obsługę żądania typu **GET**.

Servlety

Przykład:

```
@WebServlet(name = "MyFirstServlet", urlPatterns={"/addressURL"})
public class MyFirstServlet extends HttpServlet {
  protected void doGet(HttpServletRequest request, HttpServletResponse response)
  throws ServletException, IOException {
    response.getWriter().append("<h1>Hello world.</h1>");
  }
}
```

W **servlecie** możemy korzystać z obiektu żądania – **request** oraz obiektu odpowiedzi – **response**.

Servlety

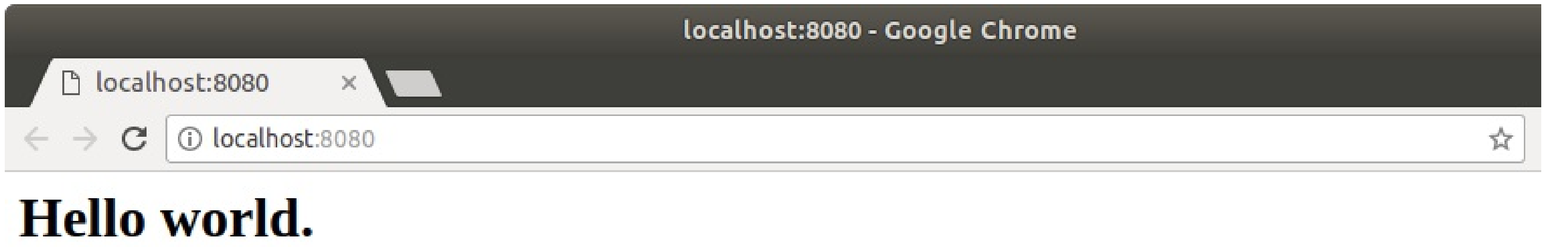
Przykład:

```
@WebServlet(name = "MyFirstServlet", urlPatterns={"/addressURL"})
public class MyFirstServlet extends HttpServlet {
  protected void doGet(HttpServletRequest request, HttpServletResponse response)
  throws ServletException, IOException {
    response.getWriter().append("<h1>Hello world.</h1>");
  }
}
```

Za pomocą pobranego **`writera`** – do obiektu odpowiedzi dodajemy zwykły tekst.

Servlety – wygenerowana strona

Po uruchomieniu servletu z poprzedniego slajdu otrzymamy w przeglądarce następujący wynik:



Metoda GET

GET

Dane przekazywane metodą **GET** są umieszczane w adresie URL.

Przykład:

```
www.test.pl/search ?name=ola &language=java
```

- znak zapytania (**?**) rozdziela adres od przesyłanych danych,
- po czym następuje seria par **klucz=wartość**, rozdzielanych znakiem ampersand (**&**).

Metoda GET

Parametry żądania przekazywane metodą **`GET`** charakteryzują się tym, że:

- można je cachować – oznacza to, że mogą być zapamiętane (razem z adresem),
- pozostają w historii przeglądarki,
- mogą być dodane do zakładek,
- mają ograniczenie długości (zależne od serwera),
- powinny być używane tylko do pobierania danych (z przyczyn bezpieczeństwa),
- nie powinno się ich używać do pracy z danymi wrażliwymi.

Przykład metody doGet

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
    System.out.println("CodersLab: żądanie GET");
}
```

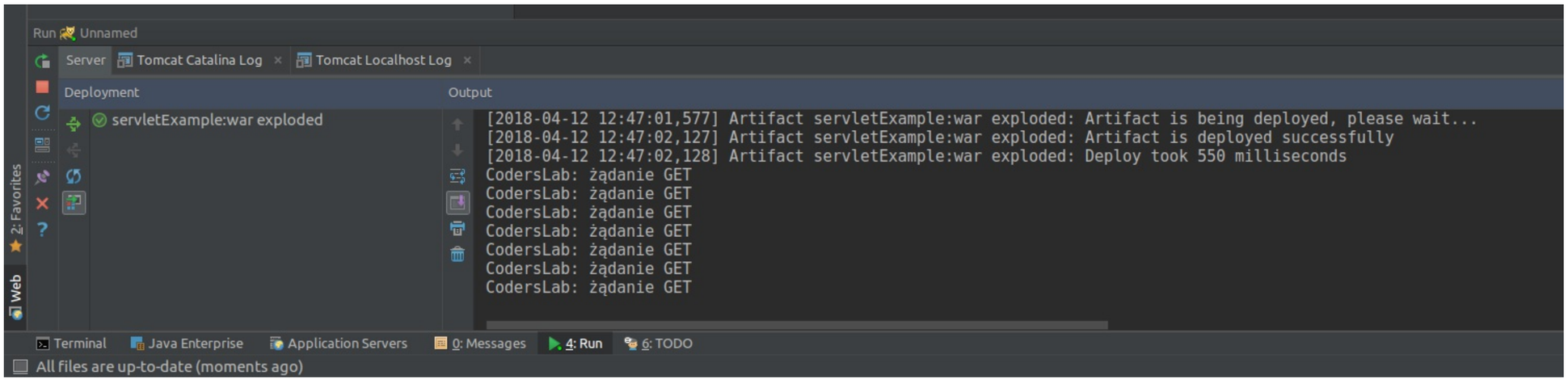
Przykład metody doGet

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
    System.out.println("CodersLab: żądanie GET");
}
```

- W metodzie **doGet** dodaliśmy – znane nam już doskonale – wyświetlenie informacji na standardowym wyjściu.
  Podobnie jak w przypadku wcześniejszych aplikacji – wynik będziemy mogli zobaczyć na konsoli.

Servlety

Każdorazowe odświeżenie strony spowoduje wypisanie na konsoli tekstu **`CodersLab: żądanie GET`**:

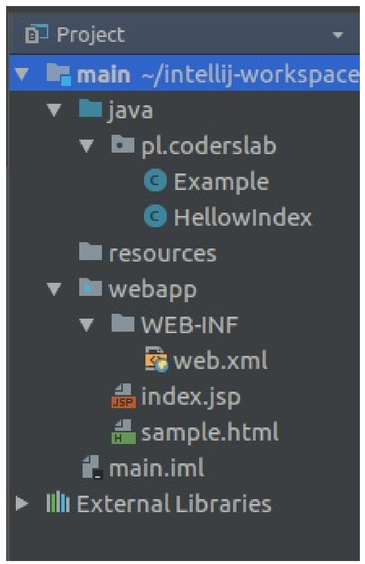


Plik web.xml

Domyślnym sposobem na określanie adresu URL, który będzie prowadził do danego servletu, są adnotacje.

Często jednak można się spotkać ze starszym rozwiązaniem, czyli opisem w pliku **`web.xml`**.

Plik ten umieszczamy w drzewie projektu, w lokalizacji: **`main -> webapp -> WEB-INF`**:



Plik web.xml

W pliku **web.xml** możemy zdefiniować listę nazw plików, które będą automatycznie wywoływane przez serwer przy uruchomieniu programu. Serwer rozpoczyna wywoływanie plików od listy podanej w web.xml (w kolejności, którą podamy). Jeśli nie zdefiniujemy tam żadnych plików, domyślnie serwer będzie szukał w naszej aplikacji plików w poniższej kolejności:

1. **`index.html`**
2. **`index.htm`**
3. **`index.jsp`**

Jeżeli nie znajdzie żadnego z tych plików, zwróci błąd z kodem 404.

Plik web.xml

Dla przykładu zdefiniujemy w pliku web.xml dwa pliki startowe:
home.html i **default.html**.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
         http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd" version="4.0">
  <welcome-file-list>
   <welcome-file>home.html</welcome-file>
   <welcome-file>default.html</welcome-file>
  </welcome-file-list>
</web-app>
```

Podajemy do serwera informację, że plikiem startowym, którego ma szukać w pierwszej kolejności, jest **home.html**, a jeśli go nie znajdzie, to **default.html**.

Mapowanie servletów

Mapowanie servletu na adres URL wykonujemy za pomocą wpisów **`servlet`** oraz **`servlet-mapping`**. Umieszczamy je wewnątrz tagu **`<web-app></web-app>`**.

Za pomocą definicji w pliku **`xml`** otrzymamy efekt analogiczny jak przy pomocy adnotacji.

```
<servlet>
  <servlet-name>TestServlet</servlet-name>
  <servlet-class>pl.coderslab.MyServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>TestServlet</servlet-name>
  <url-pattern>/AdresTestServlet</url-pattern>
</servlet-mapping>
```

Mapowanie servletów

Mapowanie servletu na adres URL wykonujemy za pomocą wpisów **`servlet`** oraz **`servlet-mapping`**. Umieszczamy je wewnątrz tagu **`<web-app></web-app>`**.

Za pomocą definicji w pliku **`xml`** otrzymamy efekt analogiczny jak przy pomocy adnotacji.

```
<servlet>
  <servlet-name>TestServlet</servlet-name>
  <servlet-class>pl.coderslab.MyServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>TestServlet</servlet-name>
  <url-pattern>/AdresTestServlet</url-pattern>
</servlet-mapping>
```

- Pełna pakietowa nazwa klasy.

Mapowanie servletów

Mapowanie servletu na adres URL wykonujemy za pomocą wpisów **`servlet`** oraz **`servlet-mapping`**. Umieszczamy je wewnątrz tagu **`<web-app></web-app>`**.

Za pomocą definicji w pliku **`xml`** otrzymamy efekt analogiczny jak przy pomocy adnotacji.

```
<servlet>
  <servlet-name>TestServlet</servlet-name>
  <servlet-class>pl.coderslab.MyServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>TestServlet</servlet-name>
  <url-pattern>/AdresTestServlet</url-pattern>
</servlet-mapping>
```

- Pełna pakietowa nazwa klasy.
- Za pomocą tej samej nazwy wiążemy **`servlet`** z określonym mapowaniem.

Mapowanie servletów

Mapowanie servletu na adres URL wykonujemy za pomocą wpisów **`servlet`** oraz **`servlet-mapping`**. Umieszczamy je wewnątrz tagu **`<web-app></web-app>`**.

Za pomocą definicji w pliku **`xml`** otrzymamy efekt analogiczny jak przy pomocy adnotacji.

```
<servlet>
  <servlet-name>TestServlet</servlet-name>
  <servlet-class>pl.coderslab.MyServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>TestServlet</servlet-name>
  <url-pattern>/AdresTestServlet</url-pattern>
</servlet-mapping>
```

- Pełna pakietowa nazwa klasy.
- Za pomocą tej samej nazwy wiążemy **`servlet`** z określonym mapowaniem.
- Adres URL, pod którym będzie dostępny **`servlet`**.

Request i response

W servlecie mamy dostęp do dwóch typów obiektów:

- `HttpServletRequest` – służy do przesyłania żądań. To są dane wysyłane na serwer od klienta – przeglądarki.

- `HttpServletResponse` – służy do przesyłania odpowiedzi. To są dane, które generujemy i wysyłamy do klienta – przeglądarki.

Obiekt HttpServletRequest

Przydatne metody obiektu **`HttpServletRequest`**:

- **`request.`** **`getParameter(String paramName)`** – pobiera wartość parametru o nazwie **`paramName`**.
- **`request.`** **`getParameterMap()`** – zwraca mapę wszystkich parametrów w postaci par klucz-wartość.
- **`request.`** **`getParameterValues(String paramName)`** – pobiera tablicę wartości parametru o nazwie **`paramName`**.
- **`request.`** **`getSession()`** – pobiera aktualną sesję, lub tworzy sesję – jeżeli wcześniej nie została stworzona.

Więcej informacji o sesjach pojawi się w kolejnej prezentacji.

Servlet – dodatkowe metody

Możemy zdefiniować dodatkowe metody, poza tymi, które obsługują żądania HTTP w ramach servletu, np.:

- **`init()`** – wywołana zostanie tylko raz w momencie tworzenia servletu.

```
public void init(){
    System.out.println("init");
}
```

Pamiętajmy, że **`servlet`** tworzy się w momencie pierwszego żądania.

- **`destroy()`** – wywołana w momencie usuwania aplikacji z serwera.

```
public void destroy() {
    System.out.println("destroy");
}
```

Wywołanie tej metody zaobserwujemy podczas zatrzymania serwera Tomcat.

Cykl życia servletu

`Cykl życia servletu` jest kontrolowany przez kontener – w naszym przypadku Tomcat. Wygląda on następująco:

- cykl ten rozpoczyna przekazanie żądania HTTP do serwera. Jeżeli instancja **`servletu`** nie istnieje, to jest ona tworzona,
- zostaje załadowana klasa **`servletu`**,
- kontener wywołuje metodę **`init()`** obiektu **`servletu`**,
- kontener wywołuje odpowiednią metodę obiektu **`servletu`**.

Przekierowania

Przy pomocy metody **`sendRedirect()`** obiektu odpowiedzi, możemy przekierować żądanie:

- z jednego servletu do innego servletu naszej aplikacji:

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    response.sendRedirect(request.getContextPath() + "/servlet2");
}
```

- lub do zupełnie innego adresu internetowego:

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
        response.sendRedirect("http://coderslab.pl");
}
```

Zadania