


Podstawy Java Enterprise Edition



Cześć

Łukasz Chrzanowski

 /lukasz-chrzanowski-dev

 /lukasz4coders

lukasz@chrzanowski.co

Materiały do zajęć

<https://github.com/infoshareacademy/jjddr1-materialy-jee>

Materiały

Pliki:

add-user.html – plik HTMLowy z gotową strukturą formularza, do wykorzystania w czasie zadania

UserDb.java – inicjalna baza danych użytkowników, przechowywana w pamięci

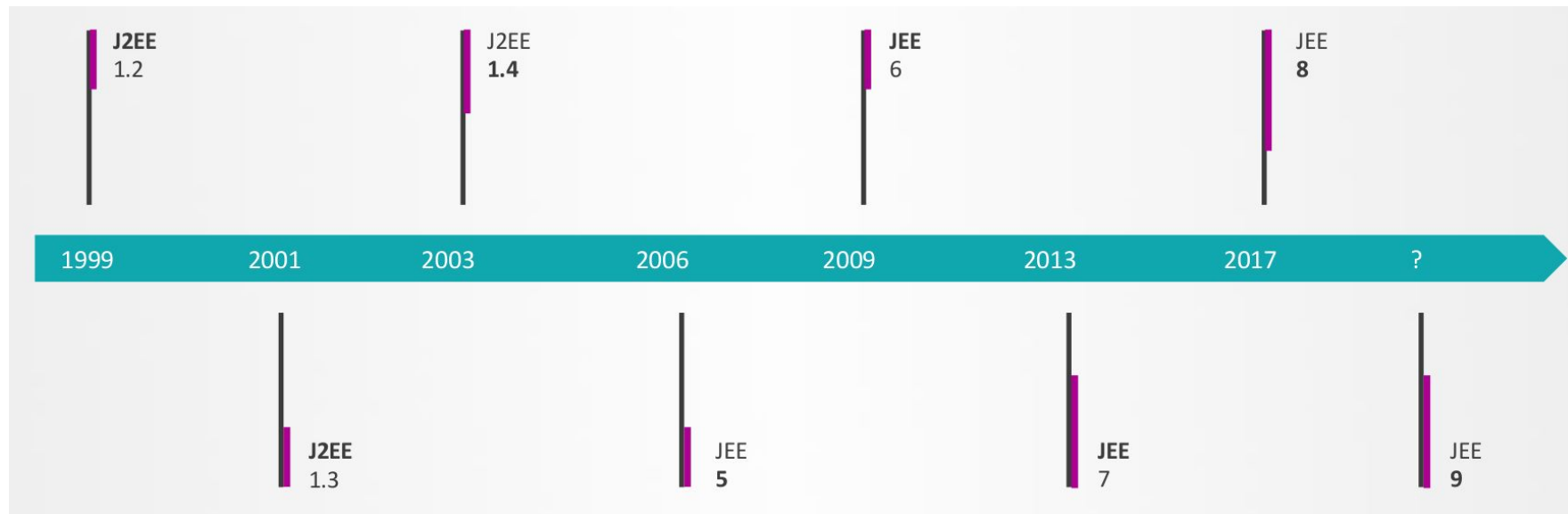
Java **Enterprise Edition**

Serwerowa platforma programistyczna języka Java

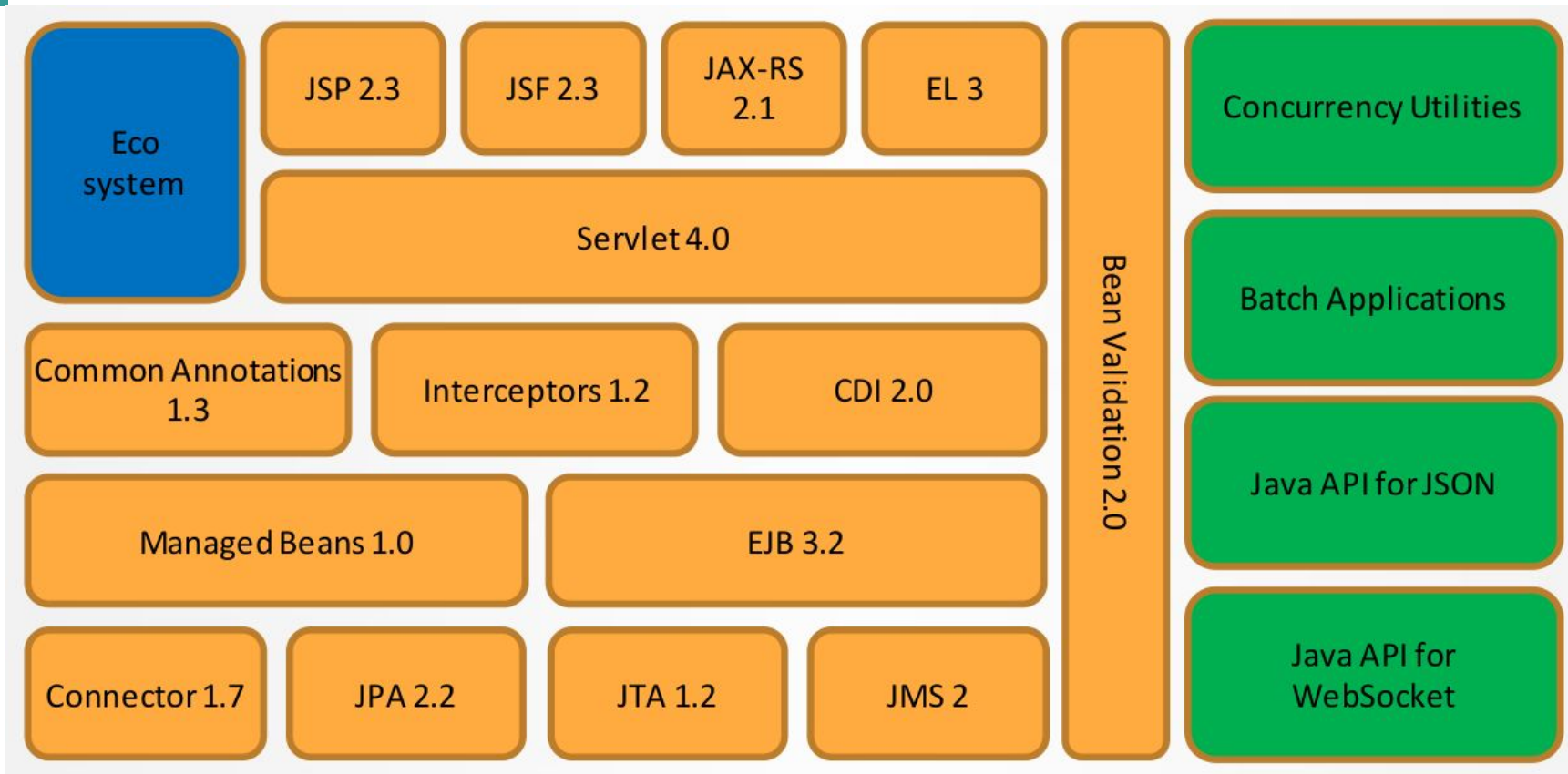
Charakterystyka

- serwerowa platforma programistyczna
- definiuje standard oparty na wielowarstwowej architekturze komponentowej
- określa zbiór interfejsów jakich implementację musi dostarczać zgodny serwer aplikacyjny
- specyfikacja zestawu API dla Javy ma na celu usprawnić wytwarzanie komercyjnego oprogramowania

Ewolucja



JEE8 - Komponenty



Java Enterprise Edition

Od lutego 2018 zmiana nazwy na Jakarta EE

JEE7 vs JEE8: Servlet 4.0

- Wsparcie dla HTTP/2 – jedno połączenie, HTTPS, jedno żądanie o zasoby, żądania binarne a nie tekstowe, priorytetyzacja

Demo: <http://www.http2demo.io/>

- PushBuilder – serwer jest w stanie wysyłać informacje do klienta

JEE7 vs JEE8: Bean Validation 2.0

Nowe adnotacje walidujące:

@Email, @NotEmpty, @NotBlank, @Positive, @Negative,
@PositiveOrZero, @NegativeOrZero, @PastOrPresent and
@FutureOrPresent

JEE7 vs JEE8

Pozostałe zmiany do doczytania:

<https://dzone.com/articles/the-top-5-new-features-in-java-ee-8>

Platforma Java

Java to zarówno język programowania jak i platforma w jednym.

Java jest wysoko poziomowym, zorientowanym obiektowo językiem programowania.

Platforma Java jest środowiskiem uruchomieniowym dla aplikacji napisanych w języku Java.

JSE a JEE

JSE dostarcza podstawowe funkcjonalności, definiuje wszystko od podstawowych typów, obiektów po rozbudowane klasy, które są używane komunikacji sieciowej, tworzenia zabezpieczeń, dostępu do baz danych, parsowania XML/JSON, tworzenia GUI, itp.

JEE jest rozszerzeniem dla JSE. Dostarcza **API** oraz środowisko uruchomieniowe dla aplikacji budowanych na wielką skalę, wielowarstwowych, skalowalnych, *niezawodnych*.

API

Application Programming Interface

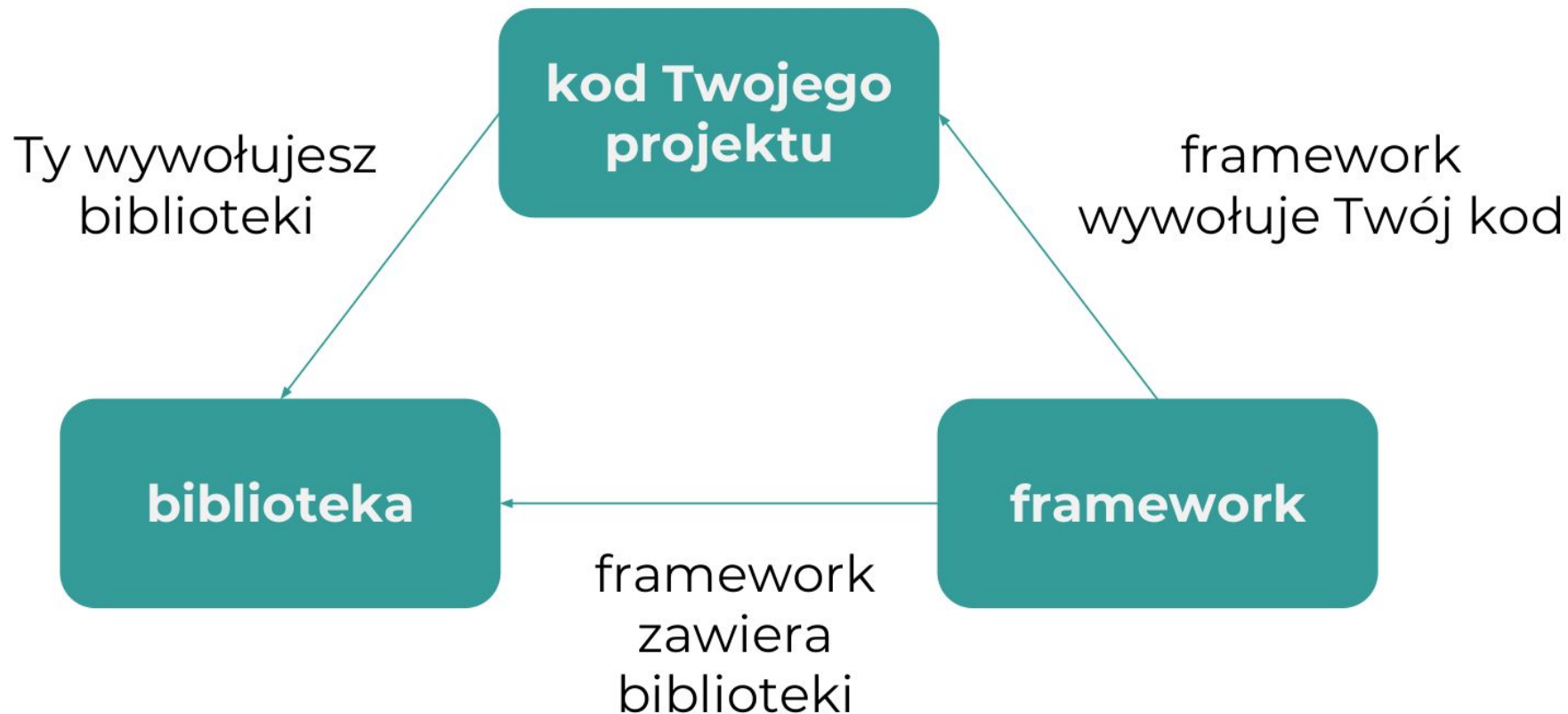
zestaw reguł i specyfikacji sposobu komunikacji programów
między sobą

https://en.wikipedia.org/wiki/Application_programming_interface

Framework vs biblioteka

to zależy...

Framework vs biblioteka



Warstwy aplikacji (uogólnione)

View Layer

Service Layer

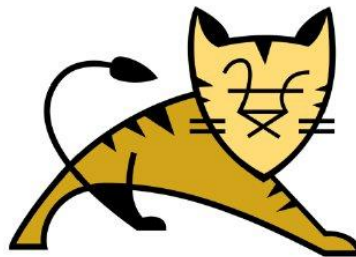
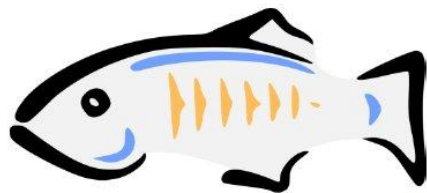
Data Access (Repository) Layer

Persistence Layer

Pierwsze uruchomienie

Serwery Java Enterprise Edition

Wybór serwera



Nasz wybór: WildFly

Serwer na którym aplikacja jest **wdrożona** (**deploy**) zapewnia pełne zarządzanie cyklem życia aplikacji, jej skalowalnością, **dostarcza implementację** JEE API.





Zadanie: Narzędzia pomocnicze JAVA

```
$ java -version
```

```
$ echo $JAVA_HOME
```

Pusto?

```
$ cd
```

```
$ nano .bash_profile
```

```
$ export JAVA_HOME=/usr/lib/jvm/defaultjava
```

```
$ source ~/.bash_profile
```

Zadanie: Pobierz serwer



Jako własny użytkownik (nie root!)

```
$ cd
```

```
$ wget https://download.jboss.org/wildfly/20.0.1.Final/wildfly-20.0.1.Final.zip
```

```
$ tar -zxvf wildfly-20.0.1.Final.tar.gz
```

```
$ ln -s wildfly-20.0.1.Final wildfly
```

Zadanie: Skonfiguruj ścieżki



```
$ cd
```

```
$ nano /home/user/.bash_profile
```

Dopisz na końcu pliku:

```
export JBOSS_HOME=/home/user/wildfly
```

```
export WILDFLY_HOME=$JBOSS_HOME
```

Zapisz, opuść plik, wykonaj:

```
$ source ~/.bash_profile
```




Zadanie: Zapewnij autokonfigurację

```
$ nano /home/user/.bashrc
```

Dopisz na końcu pliku:

```
. /home/user/.bash_profile
```

Zapisz, opuść plik.

Zalecana wyjątkowa ostrożność przy edycji .bashrc !

Zadanie: Dodaj użytkownika



```
$ cd $WILDFLY_HOME
```

```
$ ./bin/add-user.sh
```

Dokonujemy wyboru **(a) Management User**

Na wszystkie pytania yes/no odpowiadamy **yes**

Na pytanie dt grup odpowiadamy enterem bez podawania wartości.

Nadajemy własną **nazwę i hasło**.

Zezwalamy na dostęp do zdalnego API.

Zadanie: Uruchom serwer



```
$ cd $WILDFLY_HOME
```

```
./bin/standalone.sh
```

Odwiedź adresy:

127.0.0.1:8080

127.0.0.1:9990

Powinny odpowiedzieć odpowiednio: domyślną stroną startową Wildfly oraz konsolką administracyjną z monitem o zalogowanie się.

Zaloguj się.

Aplikacja **Java Enterprise Edition**

Maven support

Plugin Maven – maven-[jar|war]-plugin

Pluginy umożliwiają kompilację oraz zbudowanie docelowego **artefaktu** typu **JAR** lub **WAR**.

To, do jakiego pliku ostatecznie nasza aplikacja zostanie zapakowana, określa konfiguracja w **pom.xml**:

```
<packaging>jar</packaging>
```

lub

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

Artefakty: jar, war, ear

- **jar (Java Archive)** – zawiera biblioteki, dodatkowe zasoby, pliki konfiguracyjne, backendową logikę aplikacji
- **.war (Web Application Archive)** – zawiera warstwę webową aplikacji, może ona zostać zdeployowana w kontenerze servletowym/jsp. Zawiera najczęściej kod jsp, html, javascript jak również dodatkowe kontrolery zarządzające tą warstwą aplikacji napisane już w języku Java.
- **.ear (Enterprise Application Archive)** – zawiera jeden lub więcej modułów, używany do deploymentu bardziej złożonych aplikacji w postaci jednej paczki, która zawiera wszystkie swoje składowe

maven-[jar|war]-plugin

```
<plugins>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-war-plugin</artifactId>
    <version>3.2.3</version>
  </plugin>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-jar-plugin</artifactId>
    <version>3.2.0</version>
  </plugin>
</plugins>
```

Zadanie: Nowy projekt Maven



- Utwórz projekt Maven z wykorzystaniem archetypu: **maven-archetype-webapp** o nazwie **users-engine**
- Utwórz katalog **java** w drzewie projektu **main**
- Ustaw katalog **java** jako **Sources Root**
- Stwórz pakiet **com.isa.usersengine**
- Wykorzystaj plugin **maven-jar-plugin** oraz packaging jar
- Stwórz klasę **Main** z metodą **main** wyświetlającą do konsoli "Hello World!".
Zbuduj, uruchom projekt w konsoli.

```
<groupId>com.isa</groupId>  
<artifactId>users-engine</artifactId>  
<version>1.0-SNAPSHOT</version>
```




Zadanie: Struktura i wersja projektu

- Utwórz katalog java w drzewie projektu main
- Ustaw katalog java jako Sources Root
- Ustaw wersję javy

```
<properties>
```

```
...
```

```
    <maven.compiler.source>8</maven.compiler.source>
```

```
    <maven.compiler.target>8</maven.compiler.target>
```

```
...
```

```
</properties>
```

Zadanie: Uporządkuj pom.xml



Skasuj FIXME i węzeł `<url>`

Skasuj zależność junit 4

Skasuj całą sekcję `<pluginManagement>`

Zadanie: Klasa Main



Stwórz pakiet **com.isa.usersengine**

Stwórz klasę **Main** z metodą **main** wyświetlającą do konsoli "Hello World!".

Zbuduj, uruchom projekt w konsoli. Klasa powinna znajdować się w pakiecie **com.isa.usersengine**

Upewnij się że maven goal **package** działa poprawnie zwracając BUILD SUCCESS

Zadanie: Deployment



Wykonaj deploy aplikacji

Sprawdź czy działa domyślny widok JSP dostarczony wraz z archetypem Mavena

<http://127.0.0.1:8080/users-engine/>

Struktura katalogów

W aplikacji JEE możemy zaobserwować kilka kluczowych katalogów:

- main/java - właściwy kod aplikacji
- main/test - kod testów aplikacji
- main/**webapp** - publiczny kod udostępniany po HTTP
- main/**webapp/WEB-INF** – prywatny katalog na potrzeby konfiguracji warstwy prezentacji

Zadanie: Klasa User

Utwórz pakiet **com.isa.usersengine.domain**

Stwórz w nim klasę User z polami:
id, name, login, password, age.

Zapewnij **getter**y i **setter**y dla wskazanych pól

Zadanie: Klasa DAO

- Utwórz pakiet **com.isa.usersengine.storage** i **com.isa.usersengine.repository**
- W pakiecie **com.isa.usersengine.repository** stwórz interfejs **UserRepository** z metodami: **save**, **findById**, **findByLogin**, **findAll** – jak powinny wyglądać sygnatury metod oraz jaki powinien być typ zwracany przez te metody?
- Dostarczone repozytorium **UserDb** umieść w pakiecie **com.isa.usersengine.storage** – sprawdź główny katalog repozytorium, gdzie znajduje się plik **UserDb.java**
- W pakiecie **com.isa.usersengine.repository** utwórz klasę **UserRepositoryBean** implementującą interfejs **UserRepository**. Zaimplementuj wymagane metody wykorzystując klasę **com.isa.usersengine.storage.UserDb**

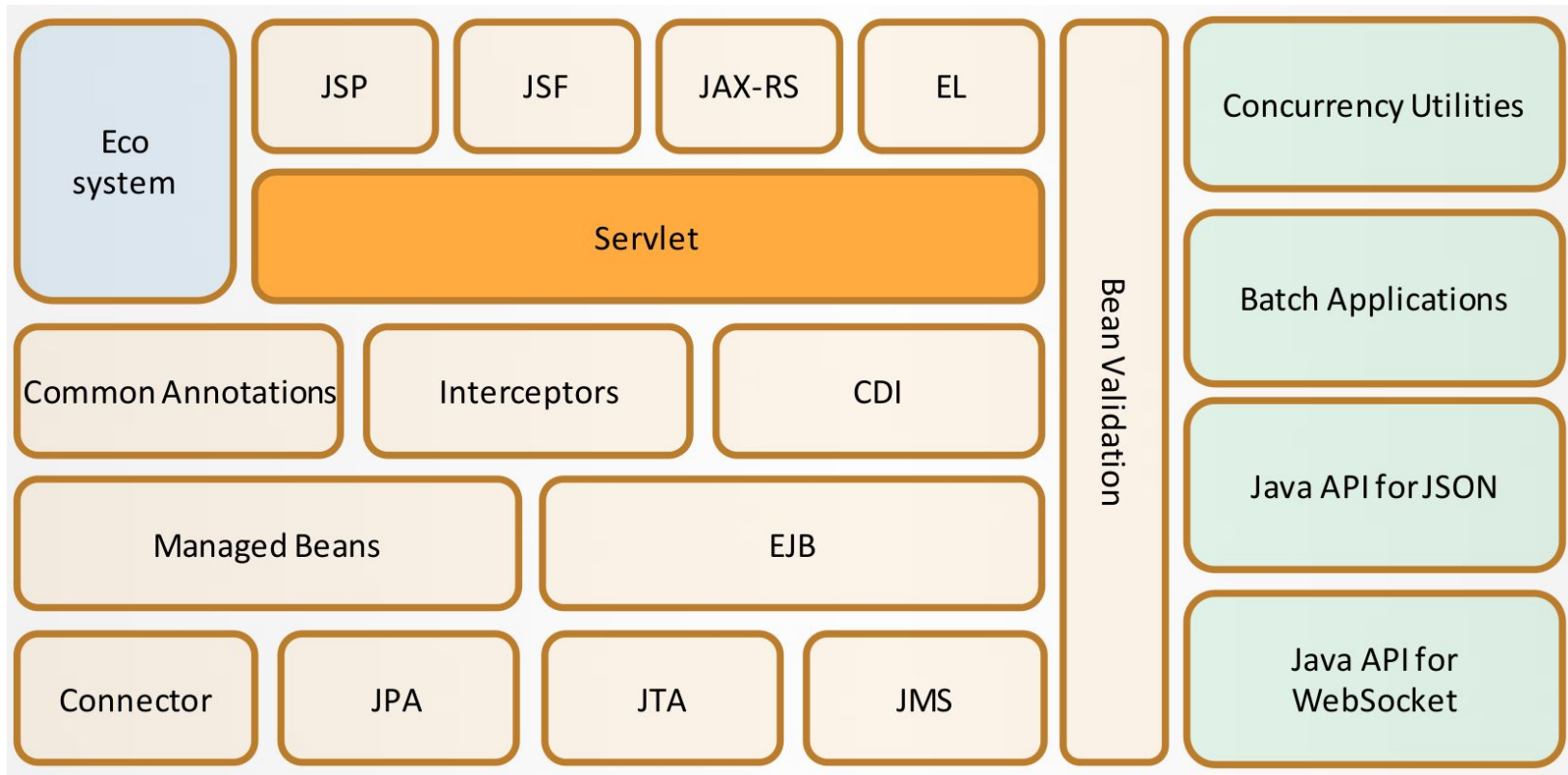
Zadanie: Klasa Main

Wykorzystaj klasę **Main** do wyświetlenia imion wszystkich użytkowników repozytorium.
Użyj **DAO**.

Java Enterprise Edition: Servlets

Komunikacja request-response

Komponenty



Specyfikacja

Specyfikuje klasy odpowiedzialne za obsługę requestów **HTTP**

Servlet API to dwa kluczowe pakiety:

javax.servlet – zawiera klasy i interfejsy stanowiące kontrakt pomiędzy klasą servletu, a środowiskiem uruchomieniowym

javax.servlet.http – zawierający klasy i interfejsy stanowiące kontrakt między klasą servletu, a środowiskiem uruchomieniowym gdzie komunikacja odbywa się w protokole HTTP

Kontener webowy

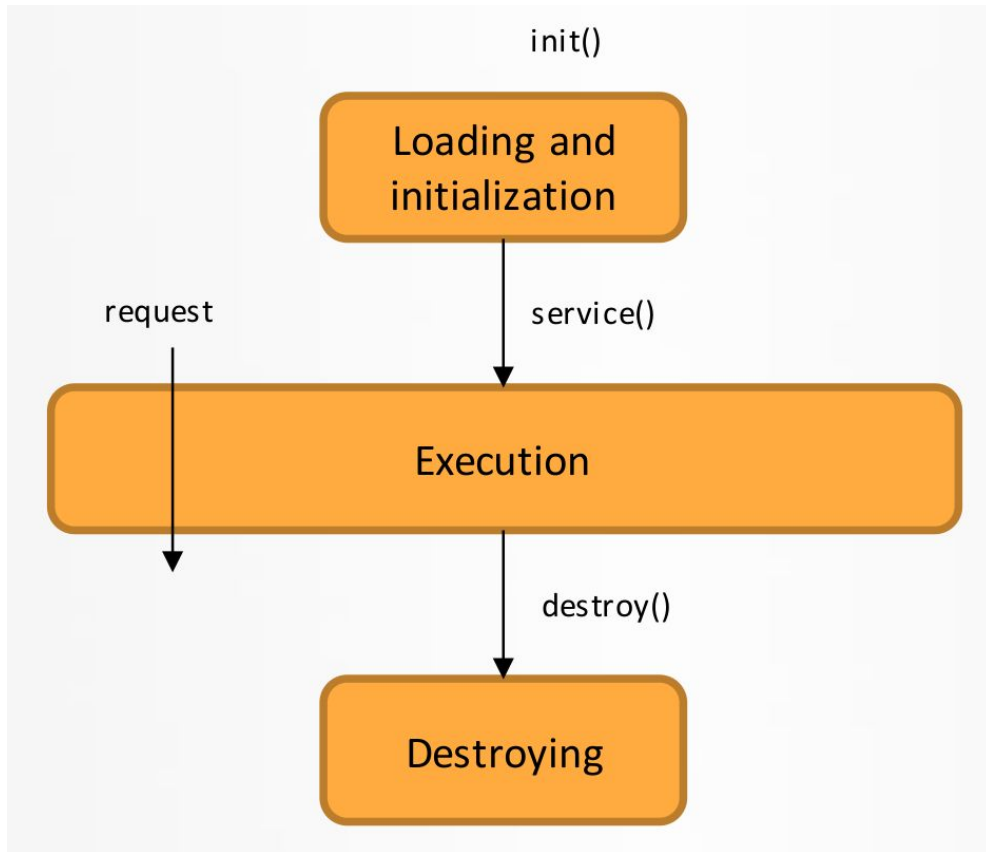
Zarządzaniem Servletami zajmuje się część serwera aplikacji zwanego **kontenerem webowym**.

Z oparciem o nasz wybór, w **Wildfly** kontenerem webowym jest **Undertow**, którego konfigurację możemy znaleźć na liście subsystemów serwera.

Zależność JEE API

```
<dependencies>
  <dependency>
    <groupId>javax</groupId>
    <artifactId>javaee-api</artifactId>
    <version>8.1</version>
  </dependency>
</dependencies>
```

Cykl życia



Metody komunikacji HTTP

- GET** – odczyt rekordu
- POST** – tworzenie rekordu
- PUT** – edycja całego rekordu
- DELETE** – kasowanie rekordu

Servlet: GET

```
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

@WebServlet("/first-servlet")
public class FirstServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
        // provide your code here
    }
}
```


HttpServletRequest

Servlet jest wrażliwy na zadane parametry requestu.

`http://[host]:[port]/[servlet-context]?name=John&age=32`

Zmienna lokalna **HttpServletRequest#req** zawiera sporo pomocnych nam informacji. Między innymi możemy ją wykorzystać do pobrania danych **requestu**.

```
req.getParameter("name")
```

Pobierze nam wartość parametru **name** z adresu w przeglądarce.

Obsługa parametrów

- Wszystkie parametry wysłane na przykład przez adres URL w przeglądarce znajdują się w obiekcie **requestu** i dostępne są przez metodę **getParameter(String var1)**
- Metoda **getParameter(String var1)** zwraca obiekt typu **String**. Należy dokonać rzutowania/parsowania do oczekiwanego typu na własną rękę.

UWAGA! Parametry **requestu** są typu **read-only**. Nie ma możliwości zmiany ich wartości.

HttpServletResponse – Obsługa odpowiedzi

Za pomocą servletu również możemy generować odpowiedzi.

Zmienna lokalna **HttpServletResponse#resp** pozwala na generowanie odpowiedzi. Za pomocą:

```
resp.setContentType("text/html;charset=UTF-8");  
PrintWriter writer = resp.getWriter();
```

```
writer.println("<!DOCTYPE html>");
```

Możemy ustawić kodowanie strony jak również pobrać writera, który będziemy pisać kod wynikowy.

Kody odpowiedzi HTTP

Każda odpowiedź wiąże się z ustawionym kodem odpowiedzi.

Dostępne kody:

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

Zadanie: Servlet Hello



- Stwórz pakiet **com.isa.usersengine.servlet** – tutaj umieszczaj wszystkie kolejne servlety
- Stwórz pierwszy servlet o nazwie **HelloServlet** w kontekście **hello-servlet**
- Spraw aby wyświetlał on **Hello World from my first Servlet!**
- Wykorzystaj **plugin maven-war-plugin** oraz **packaging war**
- Zbuduj projekt, utwórz paczkę **war** dla projektu
- Wykonaj deploy aplikacji na serwerze
- Uruchom w przeglądarce

App Root Path

Domyślnym adresem naszej aplikacji jest:

`http://[host]:[port]/${project.artifactId}/[servlet-context]`

Istnieje możliwość nadania własnej ścieżki do aplikacji. Zamiast zmiennej **`${project.artifactId}`** możemy użyć dowolnego ciągu znaków.

Nazwą tą zarządzamy w pliku **`pom.xml`**:

```
<build>
  ...
  <finalName>${project.artifactId}</finalName>
  ...
</build>
```

Context Root Path

Kolejną opcją jest możliwość ustawienia domyślnego **context root** jako naszej aplikacji, czyli zamiast odwołania:

`http://[host]:[port]/${project.artifactId}/[servlet-contet]`

Odwołamy się:

`http://[host]:[port]/[servlet-context]`

W tym celu definiujemy plik **jboss-web.xml** i umieszczamy go w katalogu **webapp/WEB-INF**

jboss-web.xml

Plik **jboss-web.xml** tworzymy w katalogu **WEB-INF**:

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss-web xmlns="http://www.jboss.com/xml/ns/javaee"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee
http://www.jboss.org/j2ee/schema/jboss-web_14_1.xsd" version="8.0">

    <context-root>/</context-root>

</jboss-web>
```


Zadanie: Context Root /



Skonfiguruj aplikację tak aby uruchamiana była z

domyślnego kontekstu /

bez konieczności dodawania nazwy aplikacji w ścieżce

Zadanie: WelcomeUserServlet



Przygotuj servlet **WelcomeUserServlet** w kontekście **welcome-user**, który wyświetli napis **Hello :name!** gdzie **:name** to wartość parametru z requestu.

Opakuj to zdanie w prostego HTML'a:

```
<!DOCTYPE html><html><body>...</body></html>
```

Jeśli parametr **name** nie został podany w requestcie, zwróć status

BAD_REQUEST

– wykorzystaj do tego klasę ze statycznymi kodami **HttpServletResponse**

Zadanie: Service



Utwórz nowy pakiet **com.isa.usersengine.service**

W nowym pakiecie utwórz klasę **UserService**

Zapewnij implementację dwóch metod w klasie serwisowej (wykorzystaj DAO):

```
public void save(User user) {
```

```
}
```

```
public User findById(Long id) {
```

```
}
```

Zadanie: FindUserByIdServlet



Utwórz nowy servlet **FindUserByIdServlet** w kontekście **find-user-by-id**

Wykonaj wyszukiwanie użytkownika po zadanym w request parametrze **id**.

Jeśli parametr **id** nie został podany w request, zwróć status **BAD_REQUEST** – wykorzystaj do tego klasę ze statycznymi kodami **HttpServletResponse**

Do rozwiązania wykorzystaj klasy **service, domain, repository**

Servlet: POST

```
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

@WebServlet("/add-user")
public class AddUserServlet extends HttpServlet {

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
        // provide your code here
    }
}
```

Interpretacja POST

- Interpretacja komunikacji typu POST jest analogiczna do typu GET.
- Wszystkie parametry wysłane formularzem znajdują się w obiekcie **requestu** dostępne przed metodę **getParameter(String var1)**
- Metoda **getParameter(String var1)** zwraca obiekt typu **String**.
Należy dokonać rzutowania/parsowania do oczekiwanego typu na własną rękę.

UWAGA! Parametry **requestu** są typu **read-only**. Nie ma możliwości zmiany ich wartości.

Zadanie: **UserServlet**



Utwórz nowy Servlet **AddUserServlet** w kontekście webowym **/user**, który będzie obsługiwał metodę komunikacji **POST**

Użyj dostarczonego pliku **add-user.html** do dodawania użytkownika

Wykonaj dodawanie nowego użytkownika wg danych podanych w formularzu do repozytorium użytkowników w pamięci oraz wyświetl komunikat, że operacja się powiodła.



Dzięki