Rafał Misiak
Lead Java Developer at Vectaury SAS

Podstawy Java **Enterprise Edition**

Materiały do zajęć

https://github.com/infoshareacademy/**jjdd5-materialy-jee**

Wi-fi relief...

1. Jeśli siedzisz bezpośrednio przy oknie i masz je po swojej prawej stronie, zaczynasz. W przeciwnym przypadku czekasz.
2. Zrealizuj zadanie ze slajdu nr 16.
3. Jak tylko skończysz, przekaż realizację zadania osobie siedzącej bezpośrednio po Twojej lewej stronie.
4. Jeśli otrzymałeś zadanie, przejdź do kroku nr 2.
5. Jeśli zakończyłeś zadanie, przejdź do aktywnego udziału w zajęciach ☺

Serwerowa platforma programistyczna języka Java
Java **Enterprise Edition**

Charakterystyka

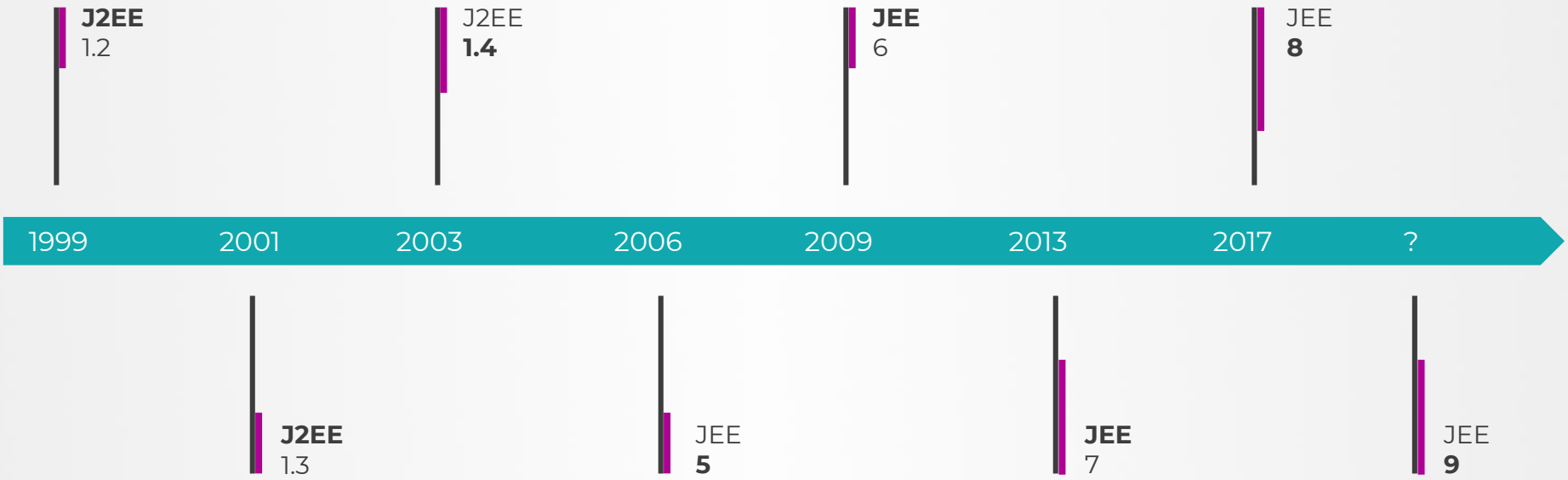
// serwerowa platforma programistyczna

// definiuje standard oparty na wielowarstwowej architekturze komponentowej

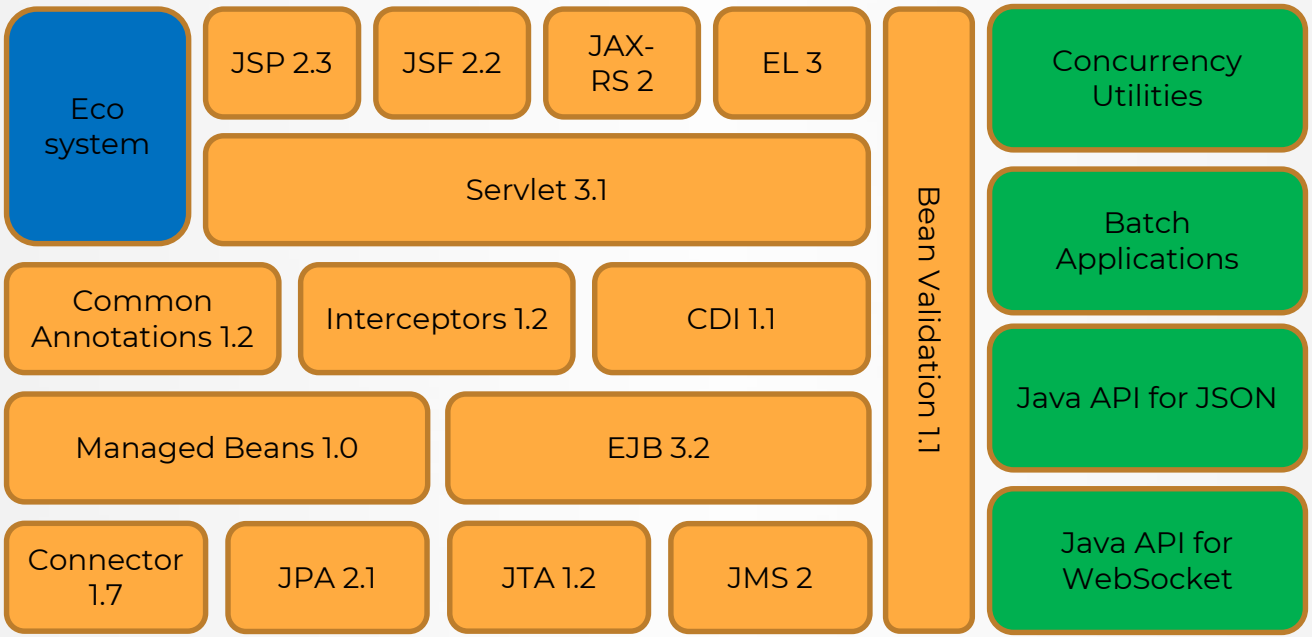
// określa zbiór interfejsów jakich implementację musi dostarczać zgodny serwer aplikacyjny

// specyfikacja zestawu API dla Javy ma na celu usprawnić wytwarzanie komercyjnego oprogramowania

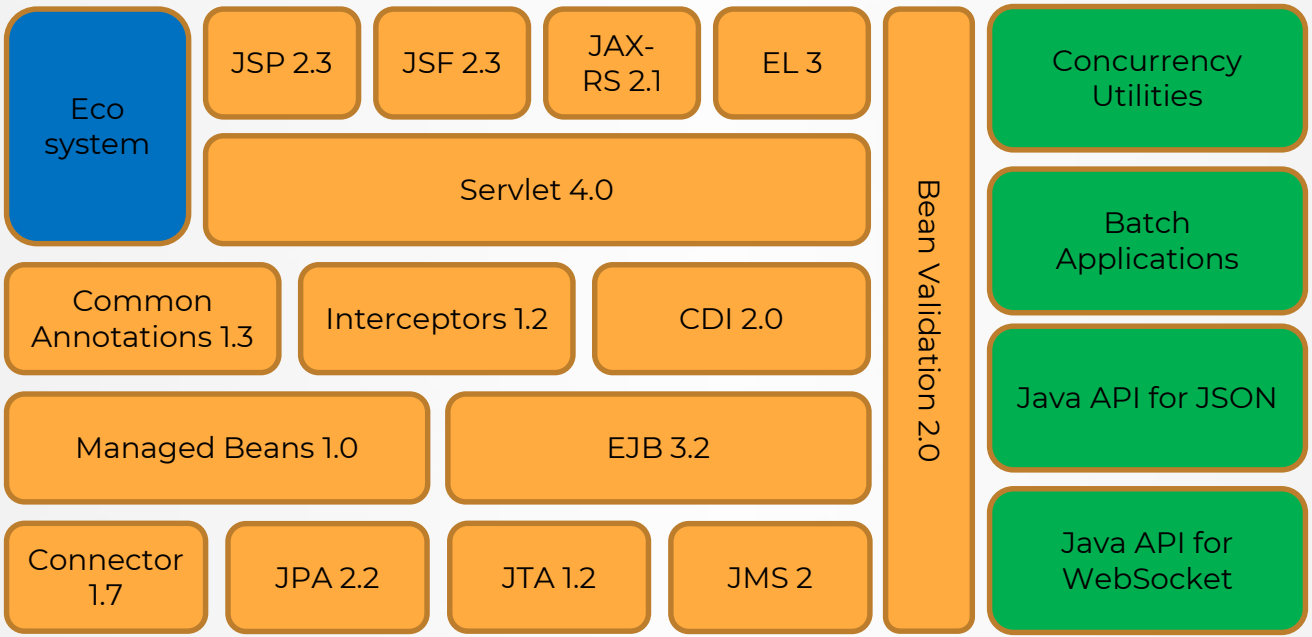
Ewolucja


J2EE 1.2
J2EE 1.3
J2EE 1.4
JEE 5
JEE 6
JEE 7
JEE 8
JEE 9
1999
2001
2003
2006
2009
2013
2017
?

JEE7 - Komponenty



JEE8 - Komponenty



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, tworzenia GUI, itp.

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

Pierwsze uruchomienie Serwery **Java Enterprise Edition**

Wybór serwera

ORACLE
WEBLOGIC

JBoss™

WildFly

jetty://

IBM
WebSphere

Nasz wybór: Wild**Fly**

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

Zadanie: Zacznij od Javy

```
$ java -version
$ echo $JAVA_HOME
```

Pusto?

```
$ cd
$ nano .bash_profile
```

Dopisz na końcu pliku:

export JAVA_HOME=/usr/lib/jvm/default-java

Zapisz, opuść plik, wykonaj:

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

Zadanie: Pobierz serwer

```
Jako własny użytkownik (nie root!)
$ cd

$ wget
http://download.jboss.org/wildfly/14.0.1.Final/wildfly-
14.0.1.Final.zip

$ tar -zxvf wildfly-14.0.`.Final.tar.gz

$ ln -s wildfly-14.0.1.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**

Nadajemy własną **nazwę** i **hasło**.

Zezwalamy na dostęp do zdalnego API.

I gotowe!

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.

Ewolucja aplikacji
Dążymy do **Java Enterprise Edition**

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 moduły, 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>3.2.2</version>
    </plugin>
    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-jar-plugin</artifactId>
        <version>3.1.0</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**

// Ustaw wersję javy na 1.8

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

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

Zadanie: Nowy projekt Maven

// 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.

Zadanie: Klasa User

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

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

// Zapewnij **gettery** i **settery** dla wskazanych pól

Zadanie: Klasa DAO

// Utwórz pakiet **com.isa.usersengine.dao** i **com.isa.usersengine.repository**

// W pakiecie **com.isa.usersengine.dao** stwórz interfejs **UsersRepositoryDao** z metodami:
addUser, **getUserById**, **getUserByLogin**, **getUsersList** – jak powinny wyglądać sygnatury metod oraz jaki powinien być typ zwracany przez te metody?

// Dostarczone repozytorium **UsersRepository** umieść w pakiecie **com.isa.usersengine.repository**

// W pakiecie **com.isa.usersengine.dao** utwórz klasę **UsersRepositoryDaoBean** implementującą interfejs **UsersRepositoryDao**. Zaimplementuj wymagane metody wykorzystując klasę **com.isa.usersengine.repository.UsersRepository**

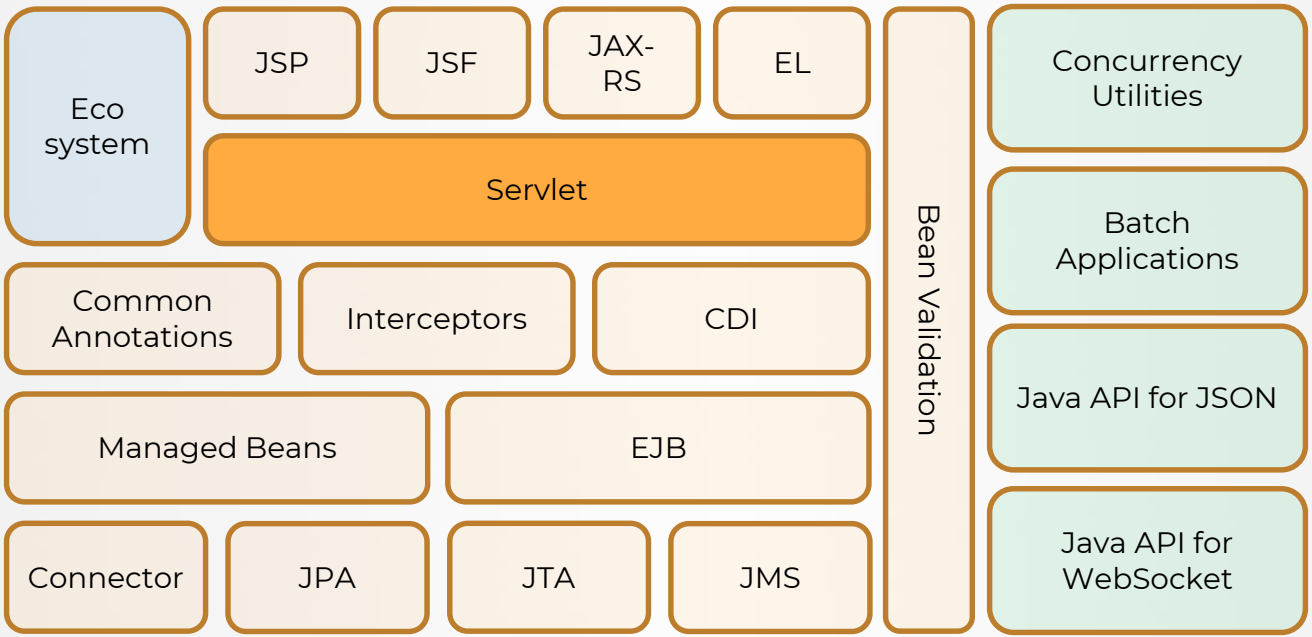
Zadanie: Klasa Main

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

// Dlaczego używamy klasy **UsersRepository**?

Komunikacja request-response
Java Enterprise Edition **Servlets**

Komponenty


Eco system
JSP
JSF
JAX-RS
EL
Servlet
Common Annotations
Interceptors
CDI
Managed Beans
EJB
Connector
JPA
JTA
JMS
Bean Validation
Concurrency Utilities
Batch Applications
Java API for JSON
Java API for WebSocket

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

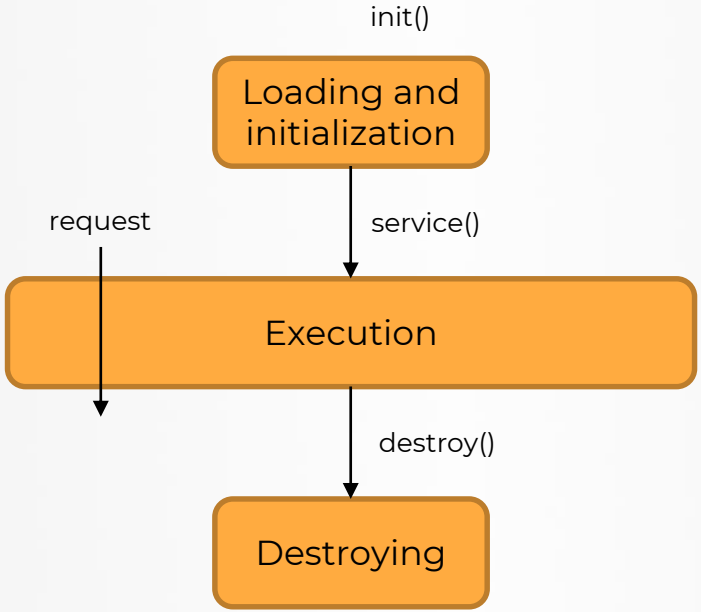
Zarządzaniem Servletami zajmuje się część serwera aplikacji zwanego **kontenerem webowym**.

Z oparciu 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>8.0</version>
    </dependency>
<dependencies>
```

Cykl życia


init()
Loading and initialization
request
service()
Execution
destroy()
Destroying

Metody komunikacji HTTP

GET – odczyt rekordu

POST – tworzenie rekordu

PUT – edycja 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**.

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

Za pomocą servletu 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, którym będziemy pisać kod wynikowy.

Zadanie: Servlet Hello

// Stwórz pakiet **com.isa.usersengine.servlets** – 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-context]

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_11_0.xsd" version="11.0">
    <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 requescie, zwróć status **BAD_REQUEST** – wykorzystaj do tego klasę ze statycznymi kodami **HttpServletResponse**

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 requescie, zwróć status **BAD_REQUEST** – wykorzystaj do tego klasę ze statycznymi kodami **HttpServletResponse**

// Do rozwiązania wykorzystaj klasy **DAO**, **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 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.

Request: Parameter vs Attribute

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

// Aby zapisać nowe wartości parametrów w **request** i przekazać je dalej w aplikacji webowej, np. do innego servletu lub widoku korzystamy z atrybutów.

```
String price = req.getParameter("price");

req.setAttribute("doubledPrice", Integer.parseInt(price) * 2);

int doubledPrice = (int) req.getAttribute("doubledPrice");
```

Session: Parameter vs Attribute

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

// Aby zapisać nowe wartości parametrów w **sesji** i przekazać je dalej w aplikacji webowej, np. do innego servletu lub widoku korzystamy z atrybutów.

```
String price = req.getParameter("price");

req.getSession().setAttribute("doubledPrice", Integer.parseInt(price) * 2);

int doubledPrice = (int) req.getSession().getAttribute("doubledPrice");
```

Zadanie: AddUserServlet

// Utwórz nowy Servlet **UserServlet**, 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

Śledzenie deploymentu

Na potrzeby debugowania, śledzenia zmian, pozyskania informacji o zdeployowanej aplikacji, testów wydajności, itp... istnieje mechanizm informujący o zarejestrowanych w kontenerze aplikacji servletach oraz przechowujący statystyki użycia poszczególnych servletów.

Informacje o zarejestrowanych servletach (w tym o statystykach):

Deployments -> **[:artefakt]** -> **View** -> **Management model** -> **subsystem** -> **undertow** -> **servlet** -> **[:servlet]**

Zbieranie statystyk

Aby statystyki były zbierane, należy je aktywować:

Configuration -> **Subsystems** -> **Web** – **Undertow** -> **Global Settings** -> **View** -> **Edit** -> **Statistics enabled=true**

Pamiętaj o restarcie serwera!

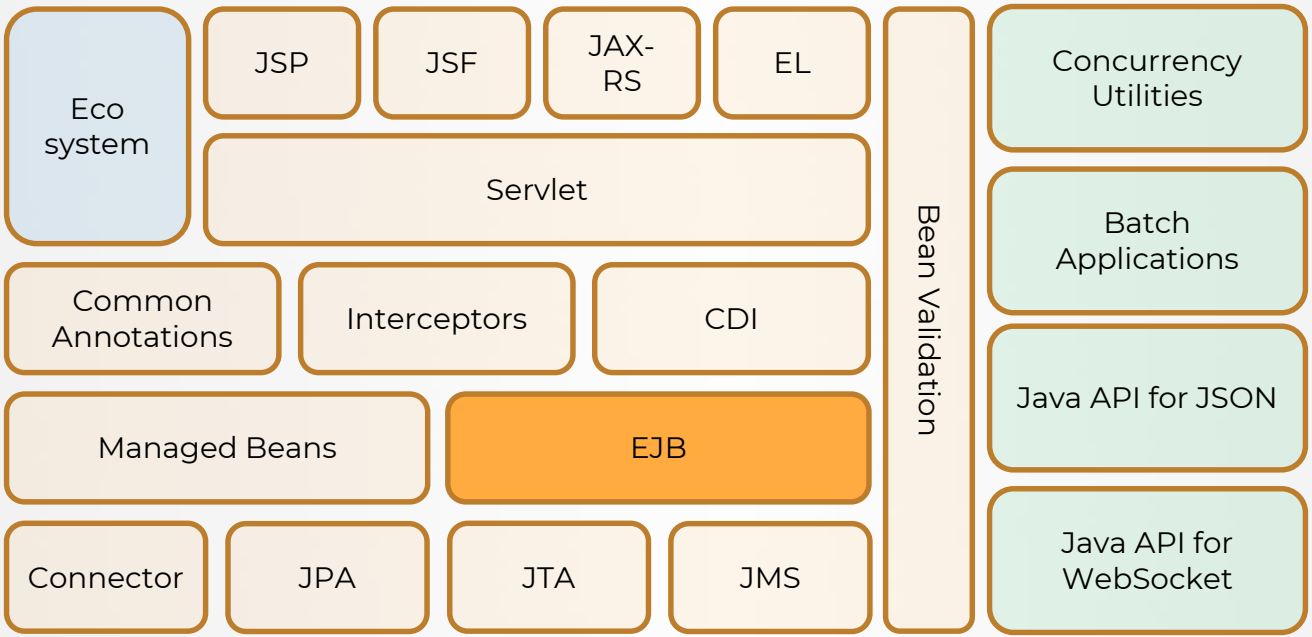
Zadanie: Statystyki

// Aktywuj statystyki servletów

// Przetestuj ich działanie

Enterprise Java Bean / Context and Dependency Injection.
EJB / **CDI**

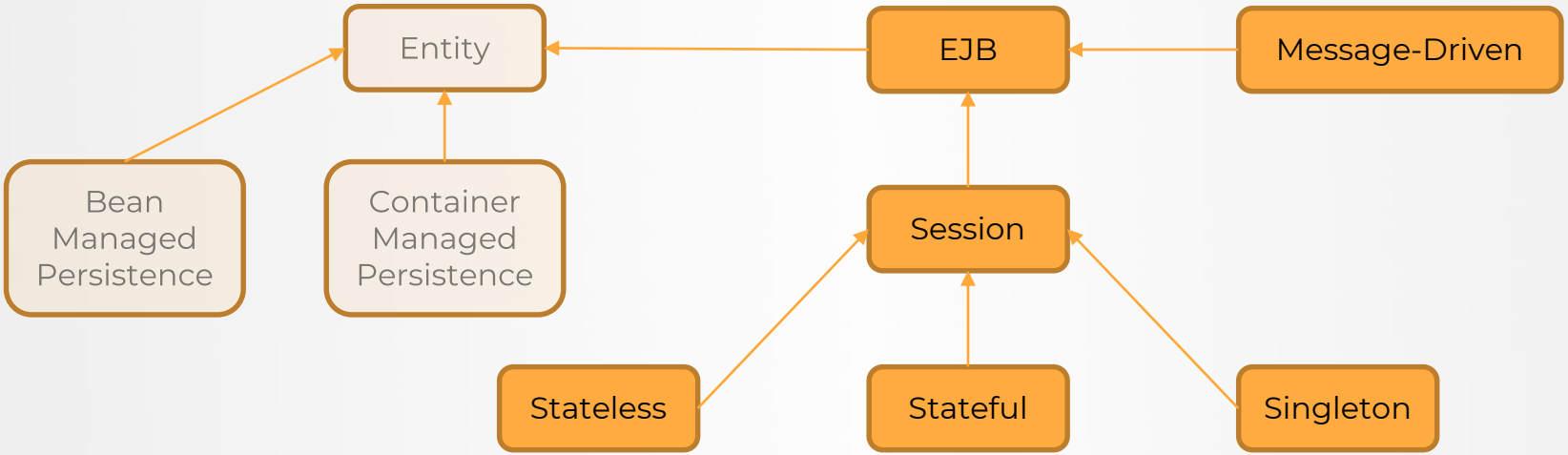
Komponenty


Eco system
JSP
JSF
JAX-RS
EL
Servlet
Common Annotations
Interceptors
CDI
Managed Beans
EJB
Connector
JPA
JTA
JMS
Bean Validation
Concurrency Utilities
Batch Applications
Java API for JSON
Java API for WebSocket

EJB: Charakterystyka

- Jedna z najpopularniejszych części JEE
- Opisuje logikę biznesową aplikacji
- Zarządzana przez kontener EJB
- Udostępnia usługi:
  - transakcyjność
  - trwałość
  - rozproszenie
  - odseparowanie warstwy prezentacji od logiki biznesowej
  - skalowalność

EJB: Budowa


Entity
EJB
Message-Driven
Bean Managed Persistence
Container Managed Persistence
Session
Stateless
Stateful
Singleton

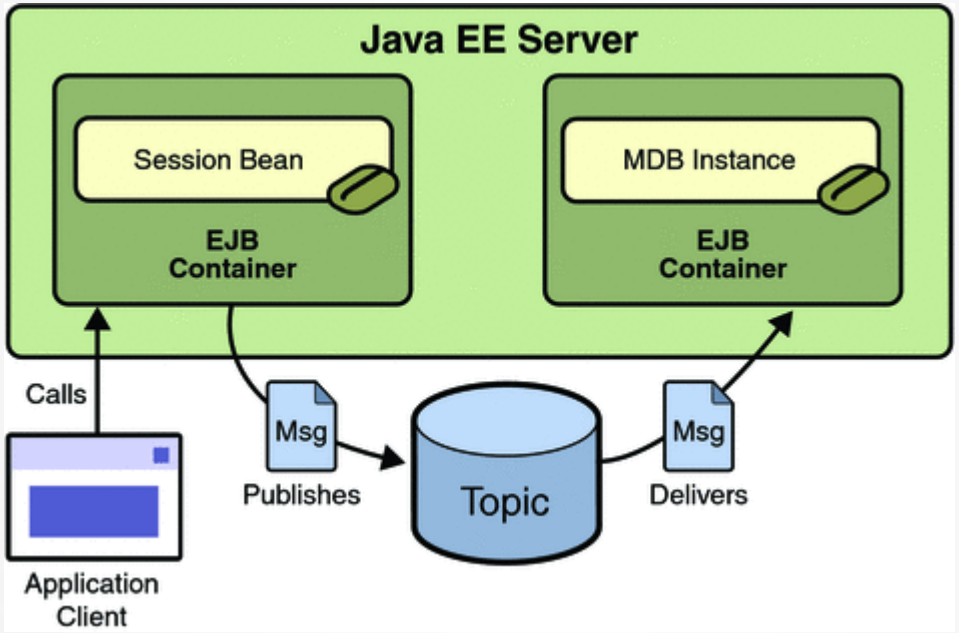
Message-Driven: Charakterystyka

Komponent sterowany wiadomościami, często opierający się na mechanizmie JMS (część specyfikacji JEE).

Komponent ten, nie jest wywoływany bezpośrednio przez klienta. Reaguje na wiadomości umieszczone, np. w kolejce.

Taka obsługa pozwala na podejście całkowicie asynchroniczne.

MDB: Schemat działania



Session Bean: Charakterystyka

Sesyjny komponent EJB realizuje konkretne zadania klienta. Klient zleca komponentowi wykonanie jakiegoś zadania poprzez wywołanie metody dostępnej w interfejsie tego komponentu.

Sesyjny komponent może obsługiwać tylko jednego klienta na raz.

Stan sesyjnego komponentu nie wykracza poza sesję, jego stan nie jest utrwalany, np. w bazie danych.

Session Bean: Stanowość

Stateful (stanowy) – pamięta stan dla konkretnej sesji z klientem, „stan konwersacji” z klientem, obejmujący wiele wywołań metod.

Stateless (bezstanowy) – nie pamięta konwersacji z klientem, nie zachowuje swojego stanu nawet na czas trwania jednej sesji, jego stan jest zachowany na czas wywołania jednej metody, serwer nie daje gwarancji utrzymania tego stanu na dalszym etapie komunikacji

Singleton – istnieje tylko jeden stan (jedna instancja) w skali całej aplikacji, bez względu na to ilu klientów zostanie do niego podłączonych

Przykład: Stanowość

```
import javax.ejb.Stateless;
import java.util.List;

@Stateless
public class UsersRepositoryDaoBean implements UsersRepositoryDao {

    …

}
```

Session Bean: Zasięg

Adnotacja **@Local** oznacza, że metody będzie można wywoływać maksymalnie z innego modułu jednak mieszczącego się w tej samej paczce EAR

Adnotacja **@Remote** oznacza, że metody będzie można wywoływać nie tylko tak jak w **@Local** ale również z całkowicie niezależnego modułu mieszczącego się zarówno na tym samym serwerze jak również na zdalnej maszynie.

Przykład: Zasięg

```
import javax.ejb.Local;

@Local
public interface UsersRepositoryDao {

    …

}
```

Zadanie: EJB

// Przekonwertuj interfejs **UsersRepositoryDao** oraz klasę **UsersRepositoryDaoBean** na **bezstanowe EJB** o zakresie **lokalnym**.

EJB: Wstrzykiwanie zależności

// Użycie zdefiniowanych EJB wykorzystuje adnotacje **@EJB/@Inject**

// **Dependency Injection** jako wzorzec architektury oprogramowania.

// Charakteryzuje się architekturą **pluginów** zamiast jawnego tworzenia bezpośrednich zależności między klasami.

// Polega na przekazywaniu między obiektami gotowych, ustanowionych obiektów danych klas (beanów).

Dependency injection (?)

Bez użycia kontenera aplikacji JEE możemy DI zrealizować za pomocą konstruktora.

```
public class User {
    private final static Permissions permissions;

    public User() {
        this.permissions = new Permissions();
    }
}
```

```
public class User {
    private final static Permissions permissions;

    public User(Permissions permissions) {
        this.permissions = permissions;
    }
}
```

EJB: Dependency injection

Istnieje możliwość przekazania zarządzania zależnościami naszemu kontenerowi aplikacji JEE. Takie działanie nazywane jest **Inversion of Control** i jest jednym ze wzorców projektowych.

Takie podejście zapewnia nam **loose coupling**, które ma na celu jak najmniejsze powiązanie obiektów między sobą.

Kontener aplikacji JEE dostarcza nam mechanizm nazywany **wstrzykiwaniem zależności**.

Kontener posiadający funkcjonalność wstrzykiwania nazywany jest kontenerem **DI** lub **IoC**. **EJB** jest kontenerem **DI**.

Przykład: Dependency injection

Używając kontenera aplikacji JEE natomiast, możemy użyć adnotacji **@EJB**.

```
public class User {

    @EJB
    PermissionsInterface permissions;

    public User() {
    }
}
```

Zadanie: EJB

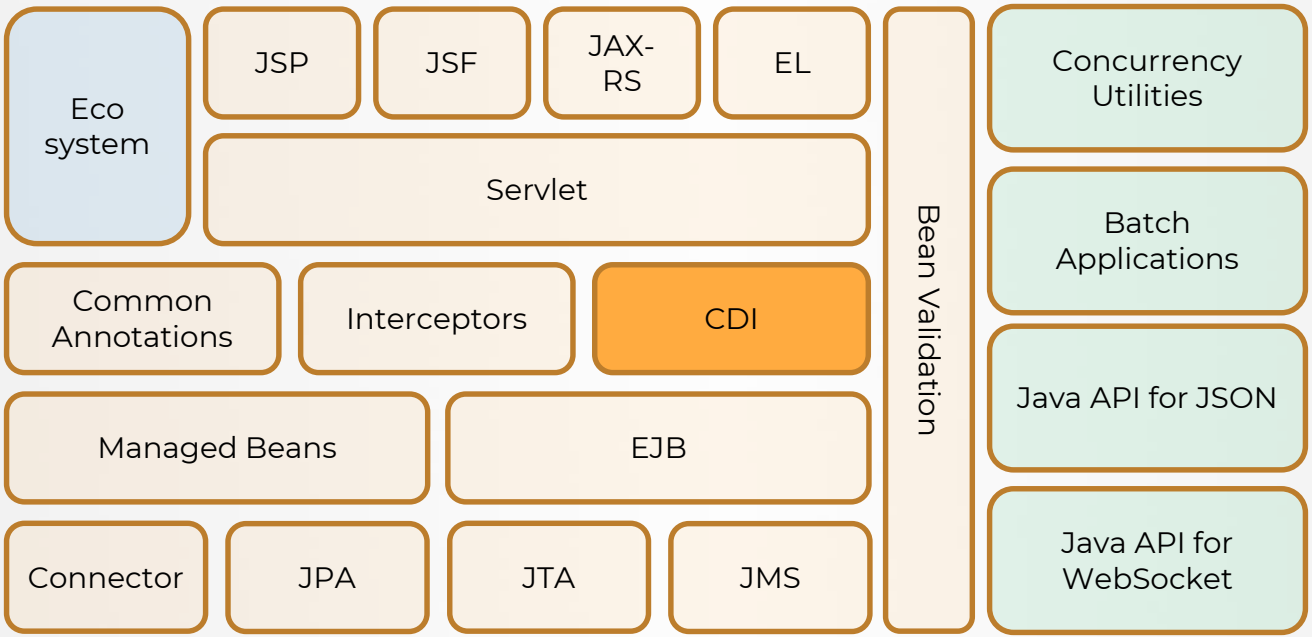
// Przekształć servlet **FindUserByIdServlet** na postać korzystającą z **EJB DI**.

CDI: Context and Dependency Inj.

Zestaw usług pozwalający na zachowanie **loose coupling** między warstwami aplikacji.

Pozwala na wstrzyknięcie większości obiektów występujących w ramach aplikacji. Nie muszą być one **EJB**.

Komponenty


Eco system
JSP
JSF
JAX-RS
EL
Servlet
Common Annotations
Interceptors
CDI
Managed Beans
EJB
Connector
JPA
JTA
JMS
Bean Validation
Concurrency Utilities
Batch Applications
Java API for JSON
Java API for WebSocket

CDI: Charakterystyka

Oznaczenia beanów **CDI**:

@ApplicationScoped – stan współdzielony przez użytkowników w kontekście całej aplikacji

@SessionScoped – stan na czas interakcji użytkownika z aplikacją webową w ramach wielu requestów

@RequestScoped – stan na czas interakcji użytkownika z aplikacją webową w ramach jednego requestu

Zadanie: CDI

// Stwórz pakiet **com.isa.usersengine.cdi**
// Stwórz w nim trzy interfejsy: **RandomUserCDIApplicationDao**, **RandomUserCDIRequestDao**, **RandomUserCDISessionDao**
// Każdy z interfejsów powinien definiować jedną metodę: **User getRandomUser();**
// Stwórz trzy implementacje tych interfejsów w postaci CDI beanów o zakresie request, session, application (analogicznie względem nazw). Metoda powinna zwracać losowego użytkownika z repozytorium.
// Utwórz servlet **RandomUserServlet** i wyświetl w nim wynik metody pochodzącej z każdego z trzech powyższych beanów.
// Przeprowadź eksperyment wyświetlając i odświeżając stronę w przeglądarce (karta standardowa oraz karta incognito)

CDI: Dodatkowe funkcje

Oznaczenia beanów CDI:

@Interceptor – uczestniczy w procesie działania aplikacji dostarczając dodatkową logikę, odseparowaną od logiki biznesowej aplikacji

@Decorator – możliwość dodawania nowej logiki dla obiektów dekorowanych klas

@Stereotype – wiązanie adnotacji w celach re-używalności definiowanych zachowań/cech

EJB > CDI

Przewagą EJB nad CDI będzie fakt, że kontener przejmie kontrolę nad transakcjami, bezpieczeństwem, współbieżnością, pulami obiektów.

@EJB vs @Inject

@EJB pozwala na wstrzykiwanie tylko i wyłącznie obiektów zarządzanych przez kontener EJB.

@Inject obsługiwana jest przez kontener CDI i pozwala na wstrzykiwanie zarówno obiektów zarządzanych przez kontener EJB jak i pozostałych beanów.

Nawiązując do rady Adama Bienia:
You can use both annotations to inject EJBs. Start with @Inject and if you encounter any problems, switch to @EJB.

@EJB vs @Inject: Definicje

```java
@Target({ElementType.METHOD, ElementType.CONSTRUCTOR, ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Inject {
}
```

```java
@Target({ElementType.TYPE, ElementType.METHOD, ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
public @interface EJB {
    String name() default "";

    String description() default "";

    String beanName() default "";

    Class beanInterface() default Object.class;

    String mappedName() default "";

    String lookup() default "";
}
```

Zadanie: CDI

// Stwórz nowy typ **enum Gender {MAN, WOMAN}** w pakiecie **com.isa.usersengine.domain**
// Dodaj atrybut płeć (**gender**) do klasy **User**. Zapewnij obsługę tego atrybutu. Uzupełnij repozytorium użytkowników o wartość tego atrybutu.
// Utwórz nowy CDI Bean o nazwie **MaxPulseBean** o zakresie requestu, a w nim dwie metody liczące maksymalny statystyczny puls dla kobiet i dla mężczyzn.
// Rozszerz funkcjonalność servletu **FindUserByIdServlet** o wyświetlenie odpowiedniej wartości.

MEN = 202 - (0.55 * age);
WOMEN = 216 - (1.09 * age);

Zadanie: JEE Elementary

// Zaimplementuj pozostałe metody klasy **UsersRepositoryDaoBean** jeśli nadal nie istnieją.

Thanks!

Contact

Rafał Misiak
Slack: #rafalmisiak
rafalmisiak@gmail.com