



# Podstawy Java Enterprise Edition



# Cześć

Łukasz Chrzanowski

 /lukasz-chrzanowski-dev

 /lukasz4coders

[lukasz@chrzanowski.co](mailto:lukasz@chrzanowski.co)

# Materiały do zajęć

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

# Files Upload

Java EE + MultipartConfig Form

# Upload: założenia

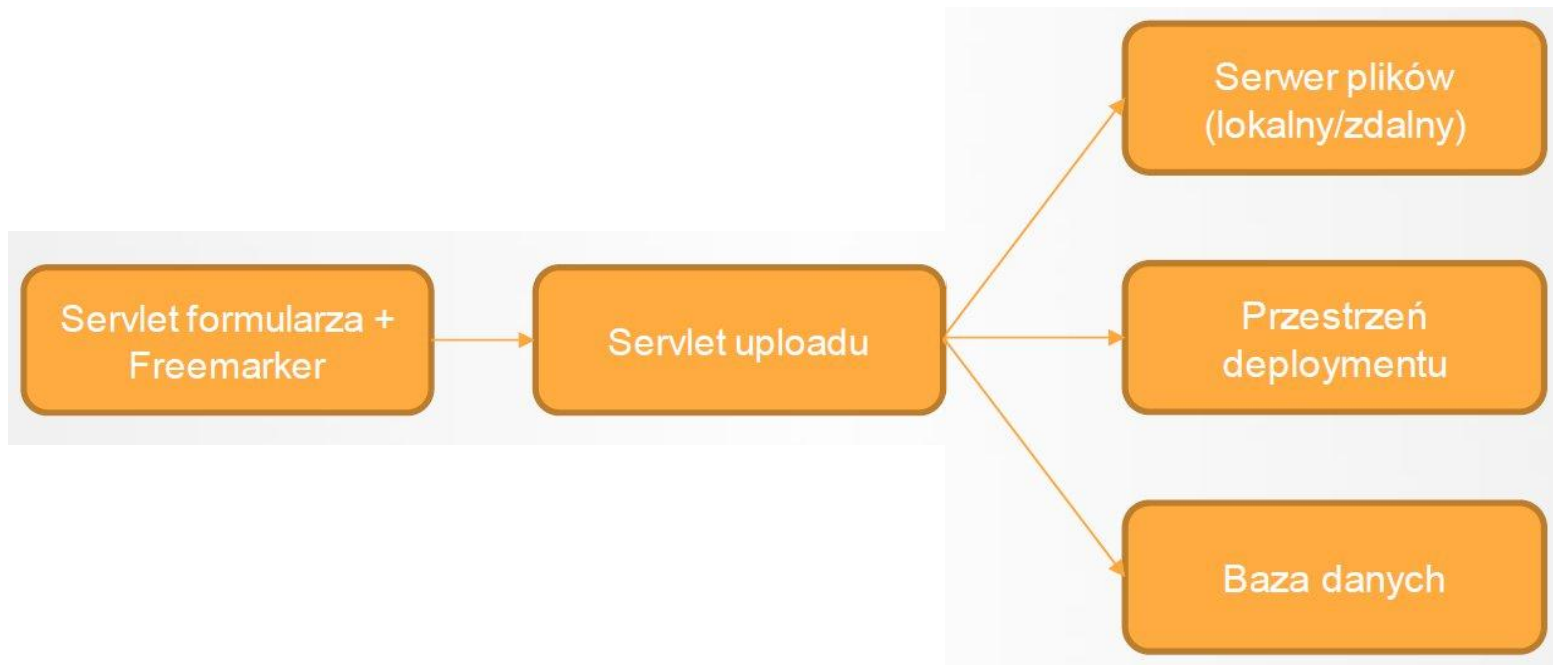
Oprócz zwykłych danych (tekstowych) istnieje możliwość wysłania (uploadu) plików na serwer.

Istnieje możliwość uploadu plików do przestrzeni zdeployowanej aplikacji.

Uwaga! W przypadku usunięcia deploymentu wszystkie dane (pliki) zostają skasowane.

# Upload: flow

Bardzo często architektura uploadu uwzględnia niezależny serwer obrazków / plików pod osobną wydzieloną domeną.



# Upload: dysk lokalny Istnieje

Istnieje również możliwość uploadu plików i zapisanie ich w ramach lokalnego systemu plików (na dysku) poza przestrzenią aplikacji.

Strona www nie jest w stanie jednak wyświetlać bezpośrednio obrazu (udostępniać bezpośrednio pliku) gdyż jest to przypadek naruszenia zasad bezpieczeństwa!

Można stworzyć servlet pośredniczący w serwowaniu plików.

# Upload: form

Aby aktywować możliwość przesyłania plików na serwer musimy rozpocząć pracę od przygotowania odpowiedniego formularza.

Tag **<form>** musi mieć zdefiniowany atrybut:

```
enctype="multipart/form-data" );
```

Oraz zawierać węzeł **<input >** typu:

```
type="file"
```



# Zadanie: Konfiguracja uploadu



Przekształć formularz dodawania (i edycji) użytkowników tak aby dodatkowo obsługiwał opcję dodawania plików. Pole opisuj jako „Image:” i nazwij je „**image**”

Utwórz katalog **/home/user/uploads** na dysku

Utwórz plik **settings.properties** w katalogu **resources** aplikacji oraz dodaj do niego klucz **Upload.Path.Images** z wartością wskazującą na katalog **uploads** - pamiętaj aby ścieżka była **absolutna**

# Zadanie: Pobieranie konfiguracji



Przygotuj nowy CDI Bean **FileUploadProcessorBean**

Stwórz w nim nową metodę, wraz z implementacją o sygnaturze **getUploadImageFilePath()**

Metoda ta powinna odczytywać plik properties oraz zwracać wartość wcześniej umieszczonego w niej klucza

# Upload: obsługa plików Servlet

Pliki wysłane przez formularz są tak zwanym elementem **Part** requestu

Plików nie pobieramy za pomocą znanej metody `getParameter(name)` tylko **`getPart(name)`**

```
Part filePart = req.getPart("image");
```

Dodatkowo servlet musi obsługiwać ten typ formularzy:

***@MultipartConfig***



## Zadanie: Upload Bean

We wcześniej przygotowanym beanie stwórz nową metodę o sygnaturze **uploadImageFile(Part filePart)** oraz zwracającą typ **File**.

Do klasy **User** dodaj nowe pole **imageURL** . Zapewnij gettery i settery.

Zintegruj użycie powyższej metody w servlecie **UserServlet**, ustaw wartość **imageURL** jako **"/images/" + file.getName()**, gdzie **file** to plik zwrócony przez wywołanie **uploadImageFile(Part filePart)**

# Zadanie: serwowanie plików

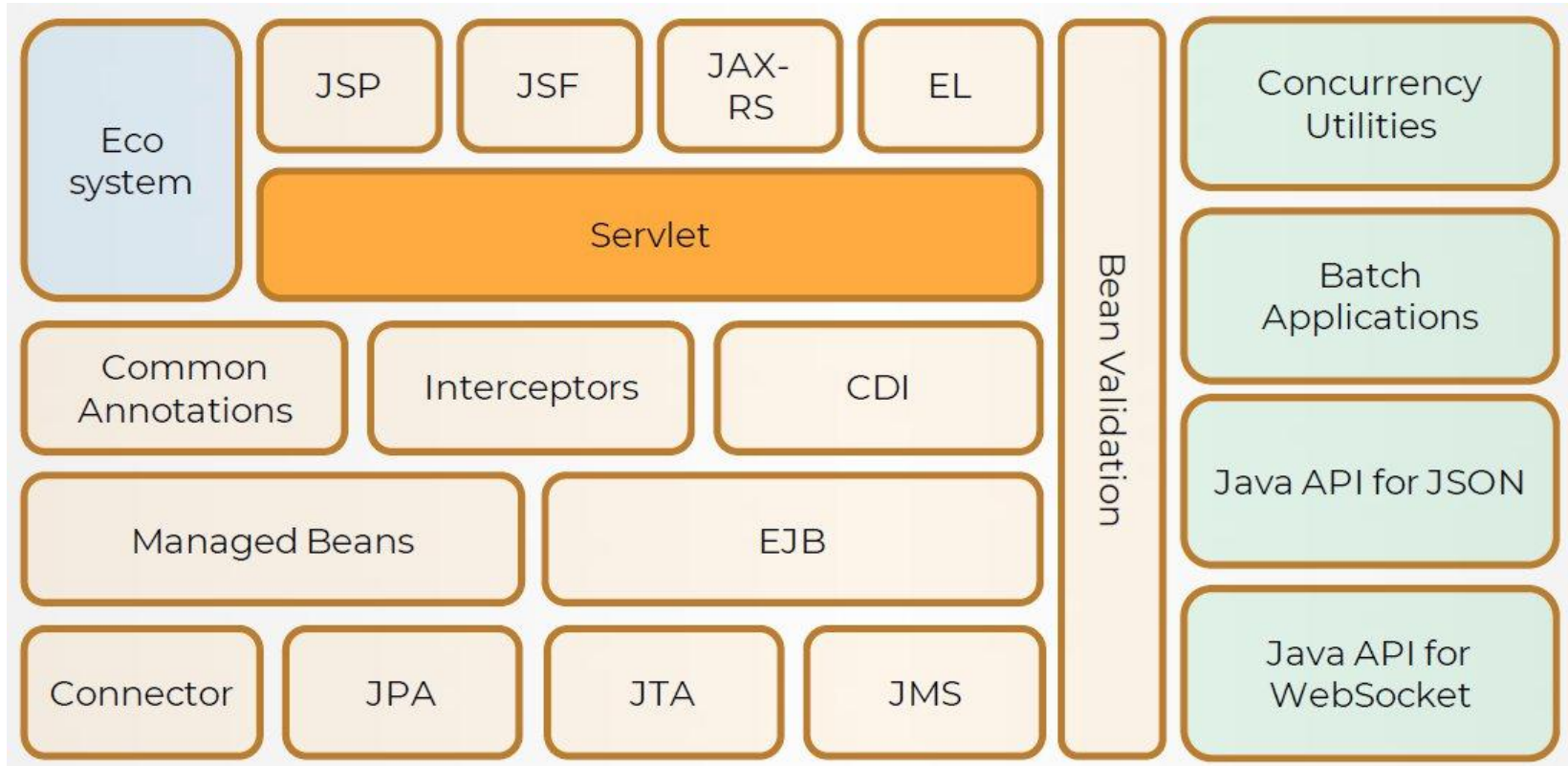


Stwórz nowy servlet **ImagesServlet**, który będzie serwował obrazki w kontekście **images**.

# Java EE **Filters**

Filtrowanie komunikacji

# Komponenty

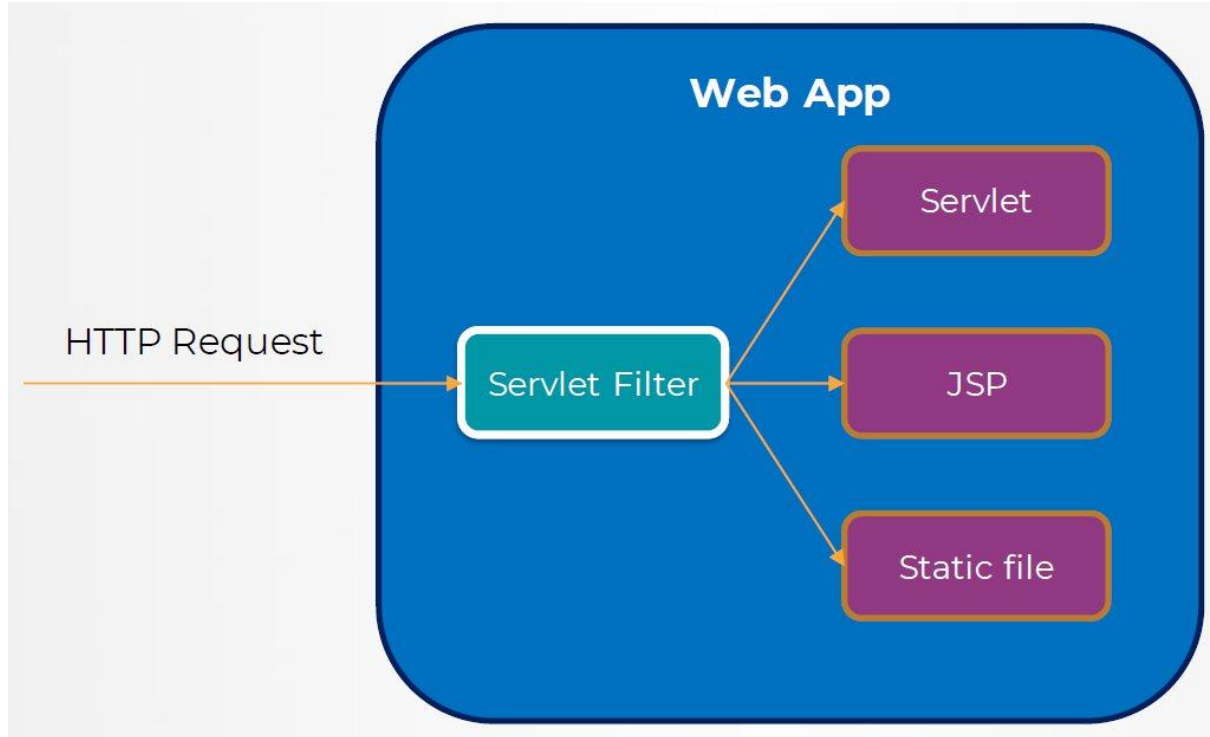


# Filtry: założenia

Filtry wykorzystywane w servletach oraz JSP są klasami używanymi do filtrowania i podejmowania akcji w komunikacji klient serwer.



# Filtery: flow



# Filtry: łańcuch wywołań



# Filtry: przeznaczenie

Przykładowe/sugerowane przeznaczenie filtrów:

- Autentykacja
- Autoryzacja
- Szyfrowanie
- Kompresja danych
- Weryfikacja i modyfikacja danych zanim trafią do servletu

# Filtry: składnia

```
@WebFilter(  
    filterName = "AuthenticationFilter",  
    urlPatterns = {"//*"},  
    initParams = {  
        @WebInitParam(name = "allowedUser", value = "root")  
    })  
public class AuthenticationFilter implements Filter {  
  
    ...  
  
}
```

# Filtry

Filtr jest klasą implementującą interfejs `javax.servlet.Filter`  
Posiada trzy metody:

- **init**(FilterConfig filterConfig)
- **doFilter**(ServletRequest request, ServletResponse response, FilterChain chain)
- **destroy**()

# Filtry: podstawowe metody

```
@Override
public void init(FilterConfig filterConfig) throws ServletException {

}

@Override
public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain
filterChain)

                                throws IOException, ServletException {

}

@Override
public void destroy() {

}
```

## Zadanie: Filtr WelcomeUser



- Napisz filtr **SalaryIncrementFilter**, który dla servletu **WelcomeUserServlet** będzie pobierał wartość parametru **salary** z **requestu**.
- Filtr powinien posiadać w konfiguracji parametr **minSalary** o wartości 100.
- Jeśli pobrana wartość będzie mniejsza niż **minSalary**, będzie ustawiał ją na wartość **minSalary** ustaloną w parametrze inicjowanym przez filtr.
- \*\*Nowa wartość powinna zostać zaprezentowana automatycznie w istniejącym widoku opartym na szablonie **welcome-user.ftlh**.

# Java EE

## Przekierowanie requestu



# Przekierowanie requestu

Istnieje możliwość przekierowania użytkownika na inny widok, jeśli spełnione są pewne warunki.

Istnieje możliwość przekierowania przeładowując (tworząc nowy request) jak również obsłużyć użytkownika w ramach tego samego requestu.

# Przekierowanie requestu

Nie dołącza ścieżki kontekstu (odnosi się do aplikacji / modułu, w którym zawarty jest serwlet)

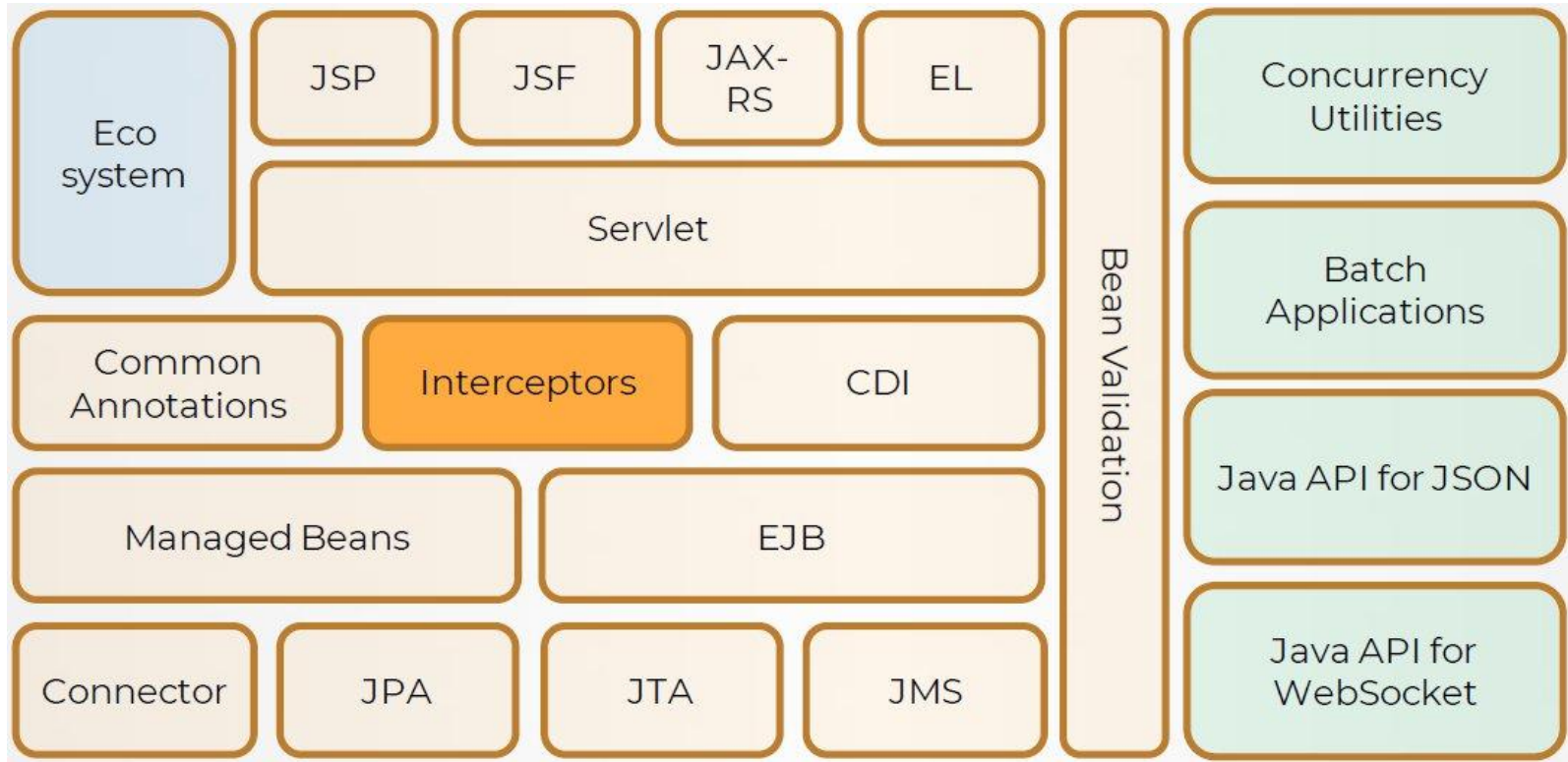
```
resp.sendRedirect("/redirec-to-view");
```

Doda ścieżkę kontekstu odpowiedniej aplikacji

```
RequestDispatcher requestDispatcher = req.getRequestDispatcher("/to-view");  
requestDispatcher.forward(req, resp);
```

# Java EE **Interceptor**

# Komponenty



# AOP: Intceptor

**Aspect Oriented Programming** – sposób tworzenia aplikacji polegający na jak najbardziej szczegółowym separowaniu elementów niezależnym względem siebie.

**Interceptory** w Javie realizują podejście AOP.

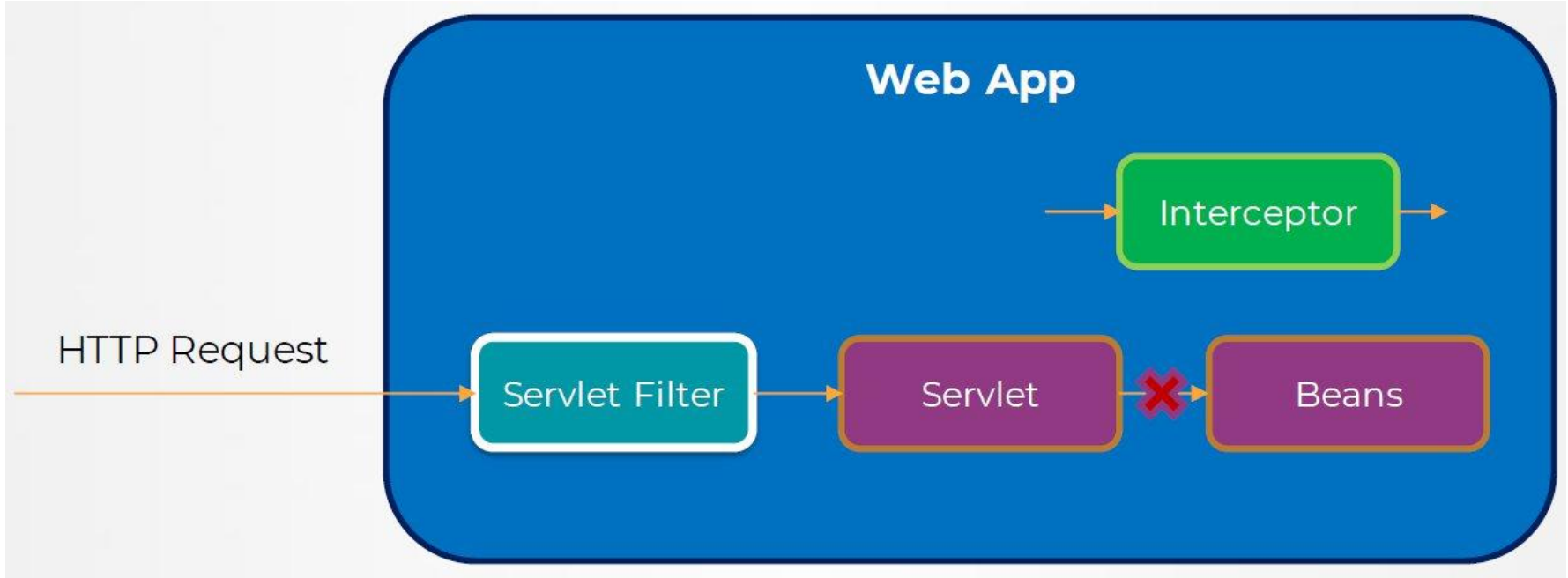
# Inteceptor

**Interceptor** to swoisty **obserwator**. Przejmuje sterowanie zadaniem jak tylko zostanie wywołany w przypadku wywołania obserwowanego bytu (np. metody).

# Inteceptor: przeznaczenie

**Interceptor** ma zastosowanie zbliżone do zastosowania filtrów z tą różnicą, że nie jest związany z **ServletContext**!

# Inteceptor: flow





# Intceptor: definicja

```
public class AddUserInterceptor {  
  
    Logger logger = Logger.getLogger(AddUserInterceptor.class.getName());  
  
    @AroundInvoke  
    public Object intercept(InvocationContext context) throws Exception {  
        logger.info("Add user has been invoked!");  
        return context.proceed();  
    }  
}
```

# Intceptor: użycie

```
@Override  
@Interceptors(AddUserInterceptor.class)  
public void addUser(User user) {  
    UsersRepository.getRepository().add(user);  
}
```



# Zadanie: Interceptor

- Napisz **interceptor** dla dodawania użytkownika, który będzie zgadywał płeć i ustawiał ją automatycznie.
- Załóż obsługę tylko polskich imion, gdzie imiona kończące się na literę „a” to imiona żeńskie, pozostałe to imiona męskie.

# @Schedule

Planowanie zadań cyklicznych

# Czym jest planowanie?

Istnieje możliwość wykonywania zadań o określonych porach dnia/tygodnia/miesiąca itp. jak również opisanie ich jako proces wykonywany cyklicznie.

# Jak definiujemy?

W pierwszej kolejności musi być to bean EJB (najczęściej Singleton).  
Uruchamiany wraz z aplikacją.

```
@Singleton  
@Startup  
public class SchedulerExample {
```

# Jak definiujemy?

Następnie musimy zaplanować zadania:

```
@Schedule (hour = "*", minute = "*", second = "*/5", info = "Every  
5 seconds timer")
```

# Skąd brać wartości?

Scheduler opiera się o definicje crontaba.

Istnieje masa generatorów online, jednym z wielu może być:

<https://www.freeformatter.com/cron-expression-generator-quartz.html>





# Zadanie: Scheduler

W pakiecie **scheduler** stwórz scheduler **UserCounterScheduler**, który będzie w co piątą sekundzie, rozpoczynając od sekundy drugiej, logował informację o liczbie użytkowników w bazie danych (pamięci).

# EJB beanName

Beany nazwane

# Beany nazwane

Beany EJB mogą implementować ten sam interfejs.  
Wstrzykiwanie musi wówczas odbywać się za pomocą wskazania nazwy beana.

# Definicje EJB nazwanych

@Local

```
public interface ShapeBean {
```

@Stateless(name = "square")

```
public class SquareBean implements ShapeBean {
```

@Stateless(name = "triangle")

```
public class TriangleBean implements ShapeBean {
```

# Wstrzyknięcie EJB nazwanych

```
@EJB(beanName = "square")  
ShapeBean shapeBean;
```



## Zadanie: Beany nazwane

Przygotuj EJB składające się z jednego interfejsu oraz dwóch implementacji tego interfejsu (ShapeBean, SquareBean, TriangleBean).

- EJB powinno być o zakresie lokalnym, bezstanowym.
- Pierwsza implementacja powinna obliczać pole powierzchni trójkąta, druga – pole kwadratu (metoda: calculateField).
- Wstrzyknij do nowego servletu każdy z beanów i oblicz pola dla przykładowych danych, dla dwóch figur.



# Dzięki