Grobe Themenverteilung

Tag 1

- Servlets
- JavaServer Faces (JSF) letzter Durchgang
- Contexts and Dependency Injection (CDI)

Tag 2

- Bean Validation
- Java Database Connectivity (JDBC)
- Java Persistence API (JPA)

Tag 3

- Java API for RESTful Web Services (JAX-RS)
- Java API for XML Web Services (JAX-WS)
- Java Message Service (JMS)

Vorstellungsrunde

- Name?
- Seit wann in der Firma/in der Industrie beschäftigt?
- Erfahrung mit folgenden Technologien:
  - Java
  - IntelliJ IDEA
  - git
  - Maven
  - (Spring?)
  - HTML
  - SQL
  - ...sonstige?

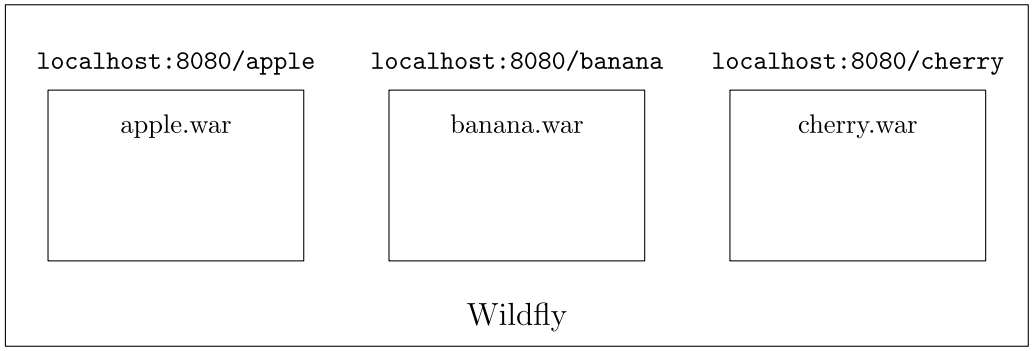
Java EE Grundlagen

Java EE (*Enterprise* Edition) ist eine Menge von Spezifikationen, die für *Unternehmens*-Software nützlich sind:

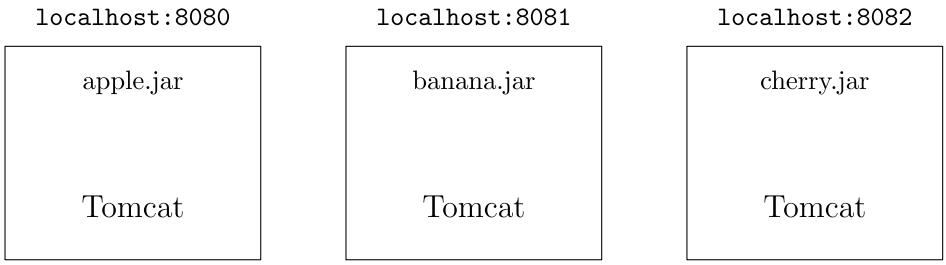
- Web-Technologien
- Persistenz
- Transaktionen
- Sicherheit
- Dependency-Injection
- u.v.m.

Diese Spezifikationen werden von Anwendungsservern wie Glassfish (Oracle), Wildfly (Red Hat) oder WebSphere (IBM) implementiert.

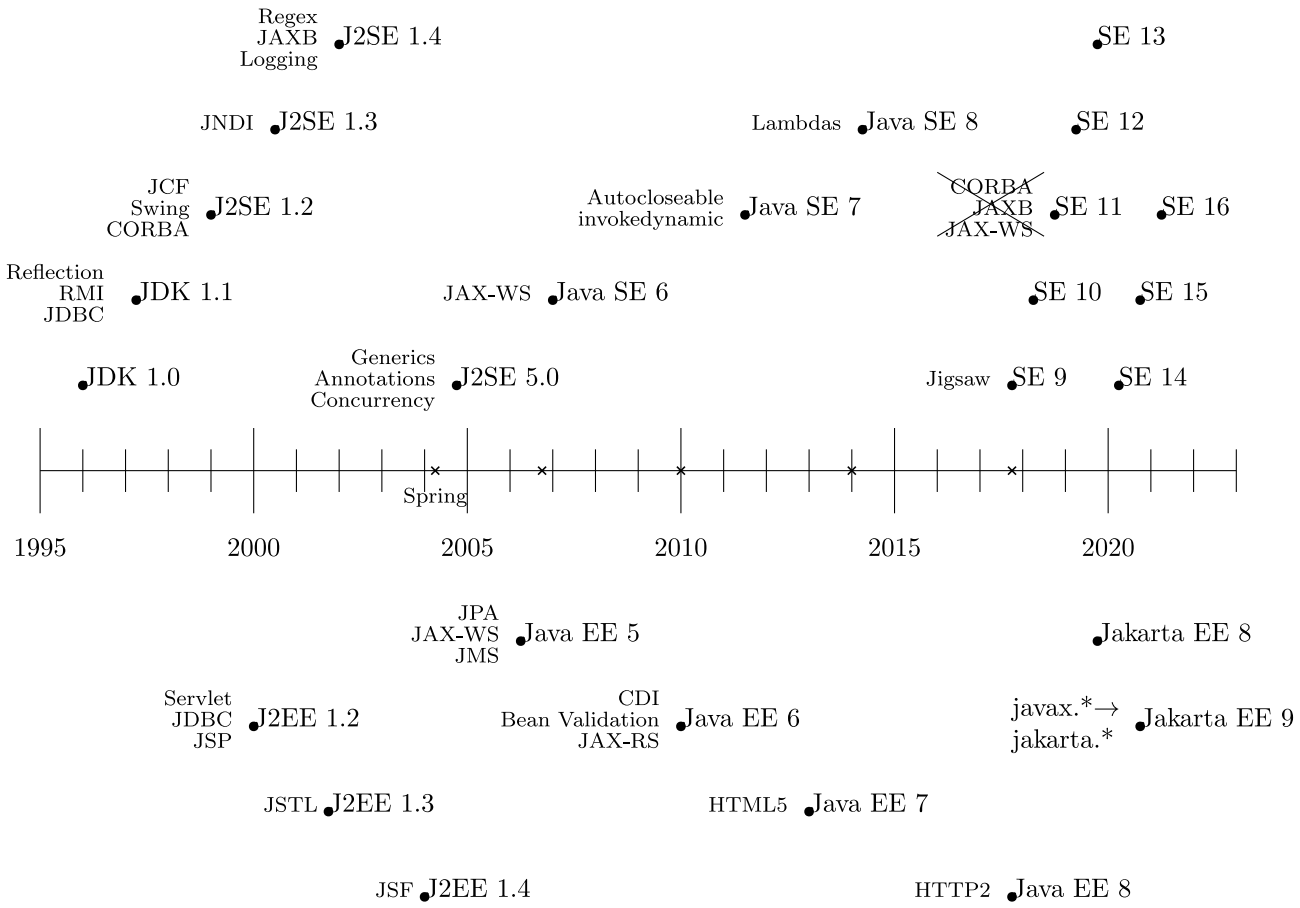
Eine Java EE Anwendung wird in Form einer komprimierten `.war`-Datei in den Anwendungsserver deployt, hier am Beispiel Wildfly:



Das ist auch für klassische Spring-Anwendungen üblich. Eine Spring-Boot-Anwendung wird dagegen meist als komprimierte `.jar`-Datei mit eingebettetem Webserver deployt, hier am Beispiel Tomcat:



Geschichte



Benötigte Software

Java Development Kit

Es existiert nur eine einzige Implementation des JDK, nämlich https://github.com/openjdk/jdk
Es gibt allerdings zahlreiche Distributionen mit unterschiedlichen Nutzungsbedingungen:

- https://adoptopenjdk.net
- https://www.azul.com/downloads/zulu
- https://aws.amazon.com/de/corretto
- https://developers.redhat.com/products/openjdk/download (erfordert Registrierung)
- https://jdk.java.net/15 (lediglich die aktuelle Java-Version wird mit Updates versorgt)
- https://www.oracle.com/java/technologies/javase-downloads.html (kostenpflichtig)

Der Java-Compiler übersetzt menschenlesbaren Quelltext in maschinenlesbaren Bytecode:

```
public class Hello {
    public static void main(String[] args) {
        System.out.println(args.length);
    }
}
```

```
javac Hello.java
```

Der Bytecode kann bei Interesse in ein menschenlesbares Format disassembliert werden:

```
javap -c Hello.class
```

```
public class Hello {
    public static void main(java.lang.String[]);
       0: getstatic     #2 // Field System.out:Ljava/io/PrintStream;
       3: aload_0
       4: arraylength
       5: invokevirtual #3 // Method java/io/PrintStream.println:(I)V
       8: return
}
```

Die Java Virtual Machine führt den Bytecode aus, beginnend in der `main`-Methode der Klasse:

```
java Hello darkness my old friend

4
```

Sowohl `javac` als auch `java` suchen `.class`-Dateien normalerweise nur im aktuellen Verzeichnis. Diesen *Classpath* kann man mit der Kommandozeilen-Option `-cp` konfigurieren:

```
java -cp .:dir:other/dir:lib1.jar:lib2.jar:dir/lib3.jar Hello
```

Achtung: Windows verwendet das Semikolon `;` statt dem Doppelpunkt `:` als Trennzeichen.

Java-ARchive mit der Dateiendung `.jar` sind im Wesentlichen umbenannte `.zip`-Dateien.

```
jar cvf new.jar *.class                 # Alle .class-Dateien archivieren
jar tvf old.jar                         # Enthaltene Dateien auflisten
jar cvfm my.jar manifest.txt *.class    # Eigenes Manifest verwenden
java -jar runnable.jar and some args    # Ausführbares Archiv starten
```

WildFly

Leicht aufzusetzender Anwendungsserver von Red Hat inkl. vorkonfigurierter H2-Datenbank

http://www.wildfly.org/downloads

Jakarta EE Full & Web Distribution herunterladen und entpacken. Ich benenne den entpackten Ordner gerne von `wildfly-22.y.z.Final` nach `wildfly` um, aber das ist Geschmackssache.

```
C:\Users\fred\wildfly\bin> standalone
```

Im Browser kann man sich unter `localhost:8080` vergewissern, dass WildFly läuft.

Falls man Kommandos eingeben will, während WildFly läuft, muss man ein zweites Terminal öffnen!

Um WildFly am Ende des Tages herunterzufahren, drückt man Strg+C.

Git (optional)

https://git-scm.com/downloads

```
C:\Users\fred\git> git clone https://github.com/frectures/jee-workshop
```

Für den unwahrscheinlichen Fall, dass git nicht sowieso schon auf dem Rechner installiert ist, lädt man sich einfach `https://github.com/frectures/jee-workshop/archive/master.zip` herunter und entpackt das ZIP.

Maven (optional)

Da IntelliJ IDEA bereits eine Maven-Integration bietet, ist eine separate Maven-Installation optional.

https://maven.apache.org/download.cgi

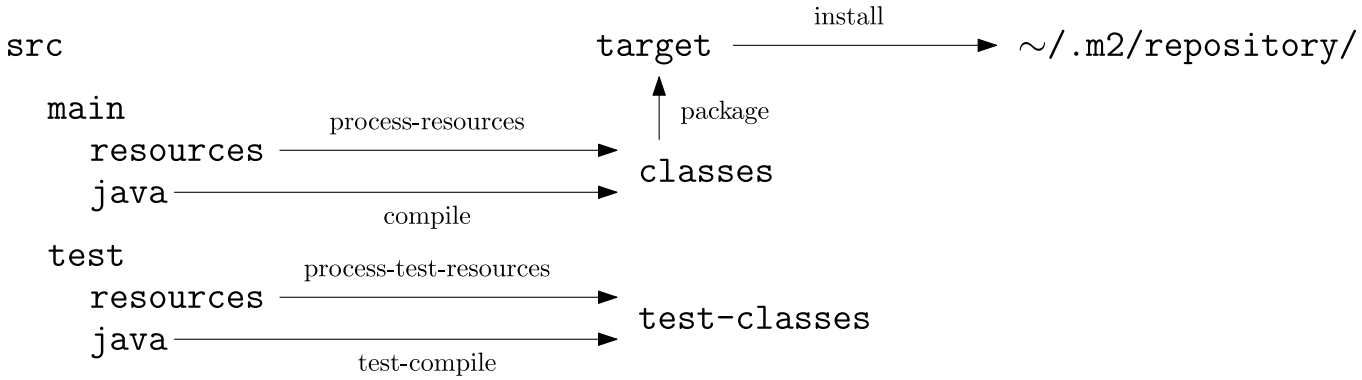
Binary zip archive herunterladen, entpacken und den `bin`-Ordner in die Umgebungsvariable `PATH` aufnehmen.

Maven löst 2 Probleme:

1. Dependency Management: benötigte Bibliotheken/Frameworks (und Maven-Plugins) werden automatisch heruntergeladen
2. Reproduzierbarer Build: unabhängig von der verwendeten/favorisierten Entwicklungsumgebung

Ein Maven-Projekt wird in der Datei `pom.xml` (*P*roject *O*bject *M*odel) konfiguriert.

Empfohlene Ordner-Struktur sowie Dateifluss zwischen den Ordnern innerhalb 6 wichtiger Phasen:



Mavens Standard-Lebenszyklus teilt sich in 23 Phasen auf, von denen 8 mit Plugins vorbelegt sind:

Phase	Plugin:Goal	Bedeutung
process-resources	resources:resources	kopiert Produktiv-Ressourcen (z.B. Bilder oder Textdateien)
compile	compiler:compile	kompiliert produktiven Quelltext `Example.java` zu JVM-Bytecode `Example.class`
process-test-resources	resources:testResources	kopiert Test-Ressourcen
test-compile	compiler:testCompile	kompiliert Test-Quelltext `ExampleTest.java` zu JVM-Bytecode `ExampleTest.class`
test	surefire:test	führt die Tests aus
package	jar:jar / war:war	komprimiert Produktiv-Klassen in ein `.jar` oder `.war`
install	install:install	kopiert das `.jar` in das lokale Maven-Repository
deploy	deploy:deploy	kopiert das `.jar` in ein entferntes Maven-Repository

Der Scope einer Dependency regelt ihre Verfügbarkeit zu 3 verschiedenen Zeitpunkten:

`<scope>`	compile	test	runtime	Beispiele
`compile`	✓	✓	✓	Apache Commons, Guava
`test`		✓		JUnit, Mockito, AssertJ
`provided`	✓	✓		Servlet API, Java EE API
`runtime`		✓	✓	JDBC Treiber, JPA Provider

Wenn man keinen expliziten Scope angibt, wird implizit `compile` verwendet.

IntelliJ IDEA

https://www.jetbrains.com/idea/download

Community herunterladen und je nach Betriebssystem ausführen bzw. entpacken.

```
File > New > Project From Existing Sources...
C:\Users\fred\git\jee-workshop\servlets
OK

(o) Import project from external model
Maven
Finish
```

Der Import kann ein paar Sekunden dauern, rechts unten sieht man den Fortschritt.

Falls keine Verzeichnisstruktur zu sehen ist: Projekt-Reiter (linker Rand) anklicken oder Alt+1 drücken

Das Maven-Plugin von Wildfly zum Deployen stößt man über den Maven-Reiter (rechter Rand) an:

- servlets
  - Plugins
    - wildfly
      - wildfly:deploy (Doppelklick)

Im Browser sollte man nun unter `localhost:8080/servlets` Datum und Uhrzeit sehen.

Zukünftige Deployments kann man über den grünen Rerun-Pfeil links unten veranlassen.

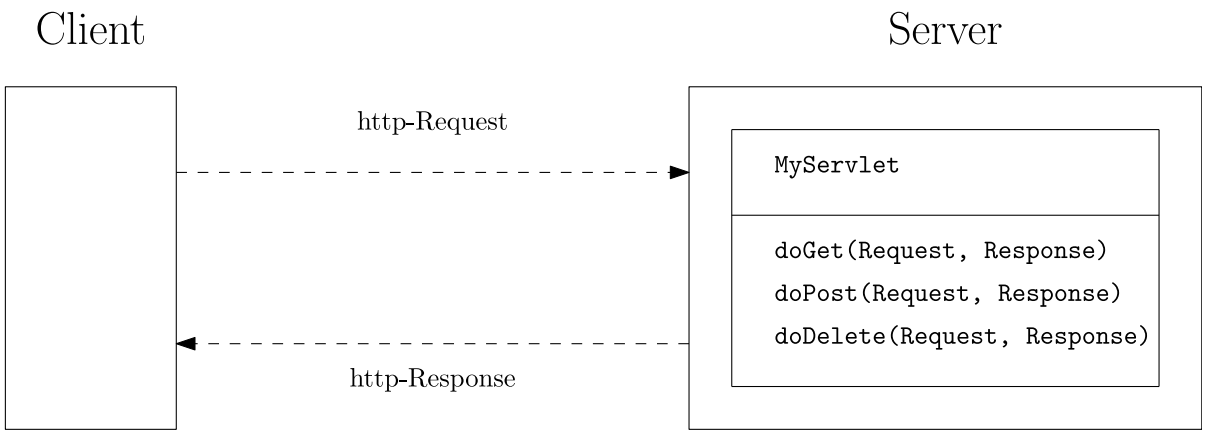
Aufgaben

1. Richte deinen Rechner gemäß obiger Anleitung ein, bis das `servlets`-Beispiel läuft.
2. Hilf deinen Sitznachbarn ggf. bei der Einrichtung ihrer Rechner.

Servlets

Ein Servlet ist eine objektorientierte Abstraktion des (zustandslosen) http-Protokolls:

1. Client sendet http-Request an Server
2. Server antwortet mit http-Response



Die manuelle Eingabe einer URL im Browser oder das Anklicken eines Hyperlinks bewirken einen GET-Request (URL beinhaltet evtl. Anfrageparameter), das Abschicken eines Formulars einen POST-Request.

```
@WebServlet("/")
public class HelloServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response)
            throws ServletException, IOException {
        try (PrintWriter out = response.getWriter()) {
            out.println("<html><body>");
            out.println("<h3>What is your favorite fruit?</h3>");
            out.println("<form method='post'>");
            out.println("<input type='text' name='fruit'>");
            out.println("<input type='submit' value='Yummy!'>");
            out.println("</form>");
            out.println("</body></html>");
        }
    }

    @Override
    protected void doPost(HttpServletRequest request,
                          HttpServletResponse response)
            throws ServletException, IOException {
        try (PrintWriter out = response.getWriter()) {
            out.println("<html><body>");
            String favoriteFruit = request.getParameter("fruit");
            out.println("I like " + favoriteFruit + ", too!");
            out.println("</body></html>");
        }
    }
}
```

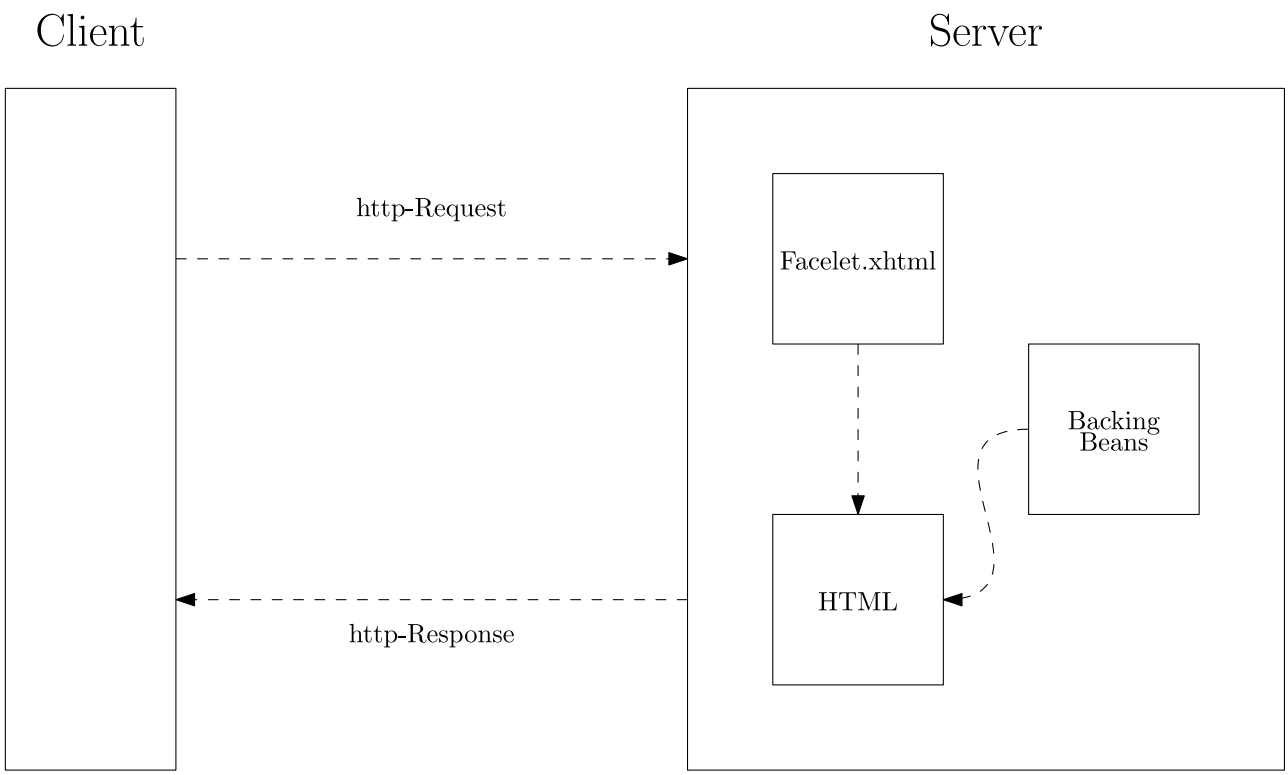
Aufgaben

1. Ändere obiges Servlet so ab, dass es nach Vorname und Nachname fragt und den Anwender entsprechend begrüßt.

2. Was passiert, wenn der Anwender HTML in ein Formularfeld eingibt, z.B. `<b>Max</b>`?
   Woran liegt das? Ist das schlimm? Wie könnte man dieses Problem beheben?

JavaServer Faces (JSF)

JavaServer Faces ist eine serverseitige Technologie für Web-Oberflächen. Der Server reagiert auf Anfragen mit einer dynamischen Konvertierung von ausdrucksstarken Facelet-Seiten in altbekanntes HTML:



Die dynamischen Daten werden dabei in *Properties* von *Backing Beans* abgelegt. Properties sind keine Java-Felder, sondern werden durch Methodenpaare `getProperty` und `setProperty` definiert.

Expression Language

Facelet-Zugriffe auf Backing Beans werden in der *Expression Language* formuliert:

Facelet `hello.xhtml`:

```
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://xmlns.jcp.org/jsf/html">

<h:body>
    <h3>What is your favorite fruit?</h3>
    <h:form>
        <h:inputText value="#{questionnaire.favoriteFruit}" />
        <!-- Expression Language: Zugriff mittels #{Bean.Property} -->

        <h:commandButton value="submit" action="welcome" />
        <!-- action="welcome" bewirkt Navigation zu welcome.xhtml -->
    </h:form>
</h:body>

</html>
```

Backing Bean `Questionnaire.java`:

```
import javax.enterprise.context.RequestScoped;
import javax.inject.Named;

@Named
@RequestScoped
public class Questionnaire {
    private String favoriteFruit;

    public String getFavoriteFruit() {
        return favoriteFruit;
    }

    public void setFavoriteFruit(String favoriteFruit) {
        this.favoriteFruit = favoriteFruit;
    }
}
```

Facelet `welcome.xhtml`:

```
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://xmlns.jcp.org/jsf/html">

<h:body>
    I like #{questionnaire.favoriteFruit}, too!
</h:body>

</html>
```

Die `.xhtml`-Dateien gehören in den Ordner `src/main/webapp`.

Konfiguration

Der Ordner `src/main/webapp/WEB-INF` beinhaltet diverse Konfigurationsdateien. Bisher relevant:

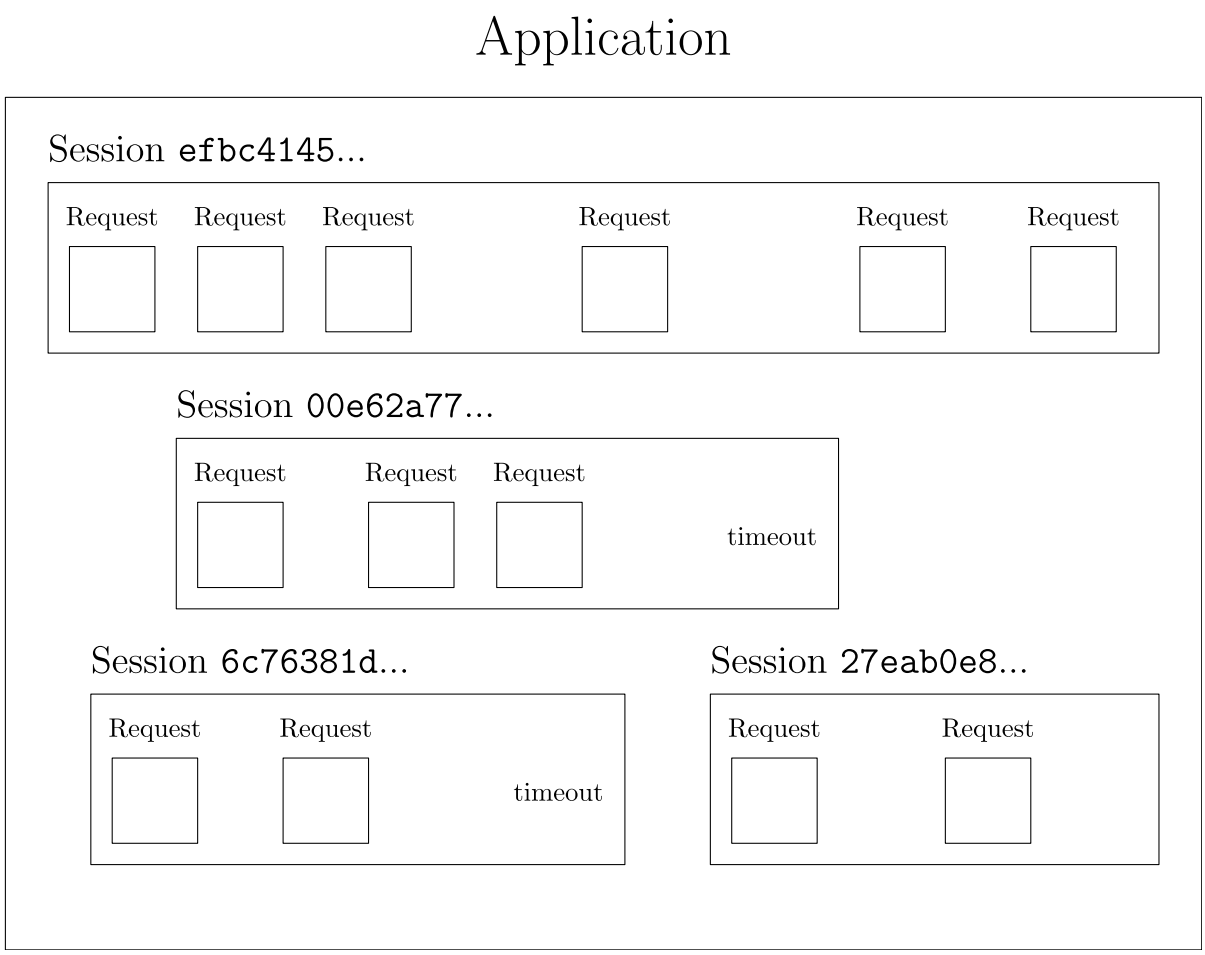
- `beans.xml` für Dependency Injection
- `faces-config.xml` für JavaServer Faces
- `web.xml` Startseite, Session-Timeout...

Lebensdauer

Mit den folgenden Annotationen (kleine Auswahl, weitere werden im Kapitel über Dependency Injection folgen) kann man die Lebensdauer einer Backing Bean regeln:

- javax.enterprise.context.**RequestScoped:** lebt innerhalb eines http-Requests
- javax.enterprise.context.**SessionScoped:** lebt innerhalb einer http-Session
- javax.enterprise.context.**ApplicationScoped:** lebt innerhalb der gesamten Anwendung ("global")

In einer Anwendung gibt es für jeden aktiven Client eine eigene Session. Eine Session besteht aus einer Abfolge von Requests mit derselben Session-ID. Nach längerer Inaktivität eines Clients läuft seine Session automatisch ab (Default 30 Minuten, konfigurierbar in `web.xml` unter `<session-config>`).



Aufgabe

`Questionnaire` ist mit `@RequestScoped` annotiert. Lege zwei weitere Backing Beans mit `@SessionScoped` bzw. `@ApplicationScoped` an. Verwende Properties aus diesen neuen Backing Beans in den Facelets. Wie wirken sich die Scopes bei gleichzeitiger Verwendung mehrerer Browser konkret aus?

```
@RequestScoped: Das Eingabefeld ist beim Laden der Seite stets leer.


@SessionScoped:


@ApplicationScoped:
```

Nützliche Tags

Die folgenden Tags werden für die nächste Aufgabe nützlich sein:

Tabellen

```
<h:dataTable value="#{backingBean.someCollection}" var="currentElement">
    <h:column>
        <f:facet name="header">ZIP code</f:facet>
        #{currentElement.zip}
    </h:column>
    <h:column>
        <f:facet name="header">Country</f:facet>
        #{currentElement.country}
    </h:column>
</h:dataTable>
```

Checkboxen

```
<h:form>
    <h:selectBooleanCheckbox value="#{backingBean.someBoolean}" />
</h:form>
```

Schleifen

```
<ul>
    <ui:repeat value="#{backingBean.someCollection}" var="currentElement">
        <li>#{currentElement}</li>
    </ui:repeat>
</ul>
```

Mittels `value` bestimmt man, über welche Sammlung iteriert wird. Den Namen des aktuellen Elements legt man über `var` fest. Über diesen kann man dann innerhalb der Schleife auf das aktuelle Element zugreifen.

Bedingte Generierung

Abhängig von einer Bedingung kann man steuern, ob für ein Element HTML generiert werden soll:

```
<ui:fragment rendered="#{backingBean.someCondition}">
    Dieser Text ist nur im generierten HTML enthalten,
    wenn die Bedingung wahr ist!
</ui:fragment>
```

Aufgabe

Baue eine Anwendung zum Auswählen von Pizza-Belägen, bestehend aus zwei Seiten. Auf der ersten Seite bekommt der Anwender eine Tabelle mit allen Pizza-Belägen angezeigt, aus denen er seine Favoriten ankreuzen kann. Auf der zweiten Seite werden ihm die ausgewählten Beläge in einer Liste angezeigt.

Wiederverwendung

Oft tauchen Inhalte in mehreren Views auf, z.B. eine Menüleiste am Anfang und ein Copyright-Hinweis am Ende. Anstatt solche Inhalte von Hand zu kopieren, verwendet man Includes und/oder Templates.

Includes

Mittels `ui:include` wird ein anderer View eingebunden:

```
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://xmlns.jcp.org/jsf/html"
    xmlns:ui="http://xmlns.jcp.org/jsf/facelets">

<h:body>
    <ui:include src="menu.xhtml" />

    My awesome view!

    <ui:include src="copyright.xhtml" />
</h:body>

</html>
```

Templates

Ein Template ist eine View-Vorlage mit variablen Bereichen, die in konkreten Views befüllt werden können.

`template.xhtml`:

```
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://xmlns.jcp.org/jsf/html"
    xmlns:ui="http://xmlns.jcp.org/jsf/facelets">

<h:body>
    <ui:include src="menu.xhtml" />

    <ui:insert name="content" />

    <ui:include src="copyright.xhtml" />
</h:body>

</html>
```

`awesome.xhtml`:

```
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://xmlns.jcp.org/jsf/html"
    xmlns:f="http://xmlns.jcp.org/jsf/core"
    xmlns:ui="http://xmlns.jcp.org/jsf/facelets">

<h:body>
    <ui:composition template="template.xhtml">
        <ui:define name="content">
            My awesome view!
        </ui:define>
    </ui:composition>
</h:body>

</html>
```

commandButton action

Das `action`-Attribut eines `<h:commandButton>` benennt den nächsten View:

```
<h:commandButton value="submit" action="welcome" />
```

Stattdessen kann man auch eine Methode angeben, die den Namen des nächsten Views liefert:

```
<h:commandButton value="submit" action="#{someBean.stringMethod}" />
```

Falls diese Methode `null` zurückliefert oder `void` als Ergebnistyp hat, bleibt der aktuelle View bestehen.

Aufgabe

Ergänze die Pizza-Beläge um einen Preis und zeige auf der zweiten Seite die Summe der Preise der ausgewählten Pizza-Beläge an.

PrimeFaces

PrimeFaces ist eine populäre, ausgereifte JSF-Komponentenbibliothek.

https://www.primefaces.org

Maven Dependency

```
<dependency>
    <groupId>org.primefaces</groupId>
    <artifactId>primefaces</artifactId>
    <version>8.0</version>
</dependency>
```

XML Namensraum

```
xmlns:p="http://primefaces.org/ui"
```

Beispielkomponente Analoge Uhr

```
<p:clock displayMode="analog" />

<style type="text/css">
    .ui-analog-clock {
        width: 500px;
    }
</style>
```

CSS auslagern

```
<h:outputStylesheet library="css" name="style.css" />
```

`src/main/webapp/resources/css/style.css`:

```
.ui-analog-clock {
    width: 500px;
}
```

Contexts and Dependency Injection (CDI)

Wenn ein Objekt ein anderes Objekt benötigt, dann bezeichnet man letzteres als eine *Dependency*:

```
class PersonService {
    private PersonRepository personRepository;   // eine Dependency
}
```

Die Variable `personRepository` speichert nur eine Referenz (bisher die null-Referenz). Es fehlt noch die Objekterzeugung und Bindung des Objekts an die Referenzvariable:

```
class PersonService {
    private PersonRepository personRepository;

    public PersonService() {
        personRepository = new PersonRepositoryImpl();
    }
}
```

Hier hat sich der Service sein benötigtes Repository selbst erzeugt. Wenn das Repository dagegen von außen hineingereicht wird, spricht man von *Dependency Injection*:

```
class PersonService {
    private PersonRepository personRepository;

    public PersonService(PersonRepository personRepository) {
        this.personRepository = personRepository;
    }
}
```

Anstelle von Konstruktor-Injektion kann man auch Setter-Injektion verwenden:

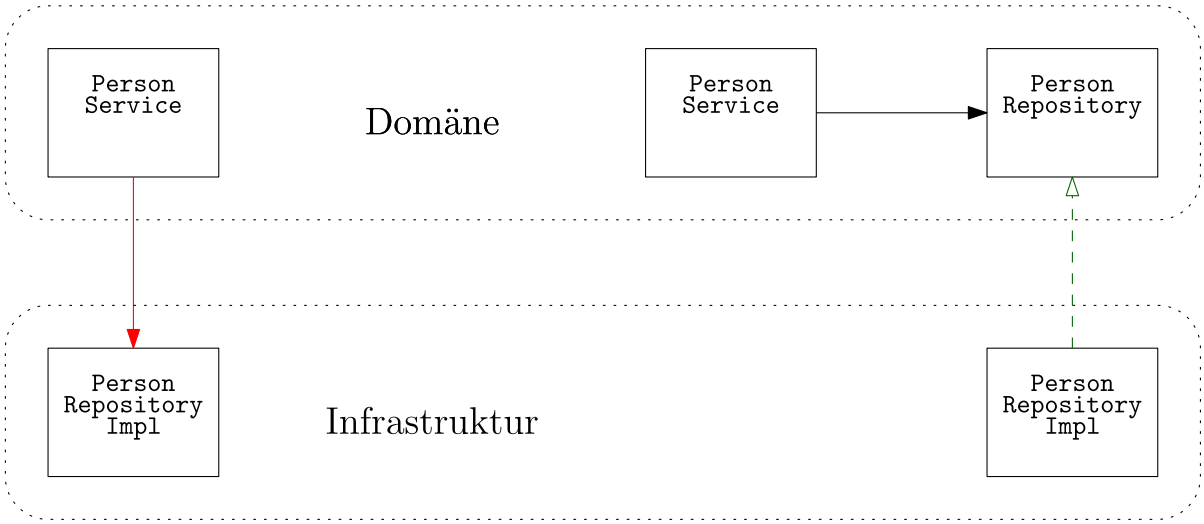
```
class PersonService {
    private PersonRepository personRepository;

    public PersonService() {
    }

    public void setPersonRepository(PersonRepository personRepository) {
        this.personRepository = personRepository;
    }
}
```

Dependency Injection führt zu einem flexibleren Design. Unit-Tests injizieren z.B. häufig sog. *Mock*-Objekte anstelle der Produktiv-Dependencies, um eine Klasse isoliert von ihren Dependencies zu testen.

Dependency Injection wird gerne mit *Dependency Inversion* kombiniert, um *Abhängigkeiten* zwischen Schichten *umzudrehen*:



(Variationen über dieses Thema: Clean Architecture, Hexagonale Architektur, Onion Architecture u.a.)

Automatische Erzeugung

Man kann die Erzeugung der Dependencies auch einem Framework überlassen. Dann muss man festlegen, welche potenziellen Dependencies zur Verfügung stehen, und wo diese Dependencies benötigt werden. Zu diesem Zweck stellt CDI geeignete Annotationen zur Verfügung:

```
@Named
@___Scoped
class PersonRepositoryImpl implements PersonRepository {
}

class PersonService {
    @Inject
    private PersonRepository personRepository;
}
```

`@___Scoped` ist ein Platzhalter für eine der folgenden Annotationen:

- javax.enterprise.context.**RequestScoped:** lebt innerhalb eines http-Requests
- javax.enterprise.context.**SessionScoped:** lebt innerhalb einer http-Session
- javax.enterprise.context.**ApplicationScoped:** lebt innerhalb der gesamten Anwendung ("global")
- javax.enterprise.context.**Dependent:** bei jeder Injektion wird ein neues Objekt erzeugt
- javax.faces.view.**ViewScoped:** lebt innerhalb eines Views
- javax.faces.flow.**FlowScoped:** lebt innerhalb eines benannten Flows (zusammengehörige Views)

Diese Annotationen legen die Lebensdauer einer entsprechend annotierten Bean-Instanz fest.

Lebenszyklus

Der Lebenszyklus einer Bean-Instanz läuft wie folgt ab:

1. Instanziierung und Injektion der eigenen Dependencies
    - entweder per Default-Konstruktor und anschließender Injektion in die Felder
    - oder per Injektion in die Parameter eines Nicht-Default-Konstruktors
2. Aufruf einer Methode, die mit `@PostConstruct` annotiert ist (sofern vorhanden)
3. ...ganz normale Verwendung...
4. Aufruf einer Methode, die mit `@PreDestroy` annotiert ist (sofern vorhanden)

FlowScoped

Die Annotation `@FlowScoped` hat einen Parameter für den Namen des Flows:

```
@Named
@FlowScoped("fred")
public class SomeBeanInFredFlow
```

Im Ordner `src/main/webapp` muss es einen Ordner mit demselben Namen geben, in diesem Fall also `src/main/webapp/fred`. Darin müssen sich mindestens zwei Dateien befinden:

1. `fred-flow.xml` für die Konfiguration des Flows
2. `fred.xhtml` als Einstiegs-View in den Flow

Weitere Views in dem Ordner können beliebig benannt werden.

Die Konfiguration `fred-flow.xml` ist wie folgt strukturiert, das Tag `<faces-config version="2.3" ...>` übernimmt man dabei am besten aus der `src/main/webapp/WEB-INF/faces-config.xml`:

```
<faces-config version="2.3" ...>
    <flow-definition id="fred">
        <flow-return id="returnFromFredFlow">
            <from-outcome>/index</from-outcome>
        </flow-return>
    </flow-definition>
</faces-config>
```

Der Einstieg in den Flow geschieht über den Namen des Flows, also `action="fred"`, *nicht* über den Namen des Einstiegs-Views, d.h. `action="fred/fred"` funktioniert nicht!

Der Ausstieg aus dem Flow geschieht über `action="returnFromFredFlow"` oder `action="/index"`.

Bean Validation

Bean Validation ist eine einheitliche Validierungs-API für Frontend, Datenbank und Webschnittstellen.

Vorgegebene Constraints

Abhängig vom Datentyp können Bean-Felder mit einem Satz vorgegebener Constraints annotiert werden:

Annotationen	Number	String	Collection	java.time
@NotNull	✓	✓	✓	✓
@Negative,OrZero	✓			
@Positive,OrZero	✓			
@Min(value)	✓			
@Max(value)	✓			
@DecimalMin(value, inclusive)	✓			
@DecimalMax(value, inclusive)	✓			
@Digits(integer, fraction)	✓	✓		
@Email		✓		
@Pattern(regexp)		✓		
@NotBlank		✓		
@NotEmpty		✓	✓	
@Size(min, max)		✓	✓	
@Past,OrPresent				✓
@Future,OrPresent				✓

(`String` ist eigentlich `CharSequence`, aber mit diesem Typ sind weniger Java-Programmierer vertraut.)

JSF

Validierungsmeldungen erscheinen normalerweise gesammelt unterhalb des Formulars.

Mit dem Tag `<h:messages>` kann man den Ort der jeweiligen Meldungen festlegen. Die Zuordnung geschieht dabei über die Attribute `id` und `for`:

```
<h:inputText id="someId" value="#{someBean.someProperty}" /><br/>
            <!-- ~~~~~~ -->
<h:messages for="someId" /><br/>
```