Software-Projekt WiSe 23/24

Aufgabenblatt 1

Abgabe bis: 26.11.2023 23:59 Uhr MEZ

Autoren: Karsten Hölscher, Marcel Steinbeck

Auf diesem Aufgabenblatt werden wir uns mit einer sehr kleinen Webanwendung[1] beschäftigen, die unter Verwendung der Technologien *Jakarta EE*[2] (im Backend) und *Jakarta Faces*[3] (im Frontend) erstellt wurde.

Bei Jakarta EE (***E**nterprise **E**dition*) handelt es sich nicht um eine Bibliothek oder ein Framework im klassischen Sinne, sondern um eine Reihe von Spezifikationen zur Realisierung sog. *Enterprise-Anwendungen*, also Softwarekomponenten, die unternehmensbezogene Prozesse unterstützen sollen. Technisch gesehen handelt es sich bei den Spezifikationen im Wesentlichen um Java-Interfaces und Annotationen, die von *Jakarta EE Containern* realisiert (implementiert) werden. Jakarta Faces ist eine Spezifikation für *user interfaces* (UI) von Java Webanwendungen und Teil von Jakarta EE.

In aller Regel sind Jakarta EE Container Serveranwendungen. Entsprechend wird Jakarta EE primär in Webanwendungen (auch *Webapplikationen* oder kürzer *Webapps* genannt) eingesetzt. Es gibt jedoch auch Implementierungen, die nur bestimmte Teile der Jakarta EE Spezifikationen realisieren (z. B. die Kommunikation mit Datenbanken) und als eigenständige Bibliotheken in Java SE (Java Standard Edition) integriert werden können. Ein Beispiel hierfür ist *Hibernate*[4] – was wir im Laufe der folgenden Aufgabenblätter noch kennenlernen werden.

Voraussetzungen

Zunächst solltet ihr euch Gedanken über die Dokumentenverwaltung (dazu zählt auch der Quellcode) in eurem Projekt machen. Die Art der Versionskontrolle ist bereits vorgegeben, da ihr von eurer Tutor:in in ein Gitlab-Projekt eingeladen wurdet, in dem ihr die Aufgaben und auch das eigentliche Softwareprojekt bearbeitet. Clont das besagte Projekt in ein Verzeichnis eurer Wahl (ab jetzt *Arbeitskopie* genannt).

Da ihr möglicherweise nicht nur die Implementierung sondern auch andere relevante Dokumente in dem Gitlab-Projekt verwalten möchtet, empfiehlt es sich, im Gitlab-Repository ein Unterverzeichnis für die Implementierung (z. B. mit dem Namen `Implementierung`) anzulegen und darunter alles zu speichern, was mit der Programmierung im Softwareprojekt zu tun hat.

Wir gehen ab jetzt der Einfachheit davon aus, dass in eurem Gitlab-Repository auf oberster Ebene ein Verzeichnis `Implementierung` existiert, in dem ihr an den Aufgabenblättern arbeitet. Wenn eure Verzeichnis-Organisation anders aussieht, müsst ihr entsprechend umdenken.

---

[1] `https://www.dwds.de/wb/Webanwendung`
[2] `https://jakarta.ee/specifications/platform/10/`
[3] `https://jakarta.ee/specifications/faces/4.0/`
[4] `https://hibernate.org/`

Eigentlich sind die Aufgabenblätter so gestaltet und gedacht, dass ihr beginnend mit dem Basisprojekt im Laufe der Aufgabenblätter immer weitere Ausbaustufen erzeugt, die am Ende zur fertigen Software weiterentwickelt werden können. Wenn ihr das nicht möchtet (und also nach dem letzten Aufgabenblatt lieber mit einer neuen Software beginnen möchtet) solltet ihr für die Aufgabenblätter entsprechend ein dediziertes Verzeichnis bzw. Unterverzeichnis einrichten.

Ihr findet bei Stud.IP das Archiv *eksadat.zip*. Darin befindet sich ein sehr kleines Projekt für eine Webanwendung namens `Eksadat`[5], das als Basis für dieses und die folgenden Aufgabenblätter dient und ebenso zur Erstellung der finalen Applikation im Rahmen des Softwareprojektes verwendet werden kann.

Entpackt das besagte Archiv in das Verzeichnis `Implementierung` eurer Arbeitskopie. Dort befindet sich dann also das Unterverzeichnis `eksadat`. In diesem Verzeichnis gibt es einige Dateien und Unterverzeichnisse, die wir später im Detail diskutieren. Wenn ihr bereits einen eigenen Namen für euer Softwareprojekt gewählt haben (und das Basisprojekt zum fertigen Endprodukt ausbauen wollen), dann solltet ihr das Verzeichnis bereits jetzt entsprechend umbenennen.

Bevor wir uns aber mit dem Aufbau und Inhalt der Webanwendung beschäftigen, möchten wir es *bauen*, auf einen Jakarta EE Container *deployen* und dort ausführen. Da die Kund:innen bereits einen *WildFly*-Server[6] (das ist ein konkreter Jakarta EE Container) betreiben und eure fertige Anwendung später darauf laufen soll, verwenden wir während der Entwicklung zum Ausführen unserer Webanwendung einen lokalen *WildFly*-Server (ab jetzt der Einfachheit halber *WildFly* genannt). Eine systemweite Installation ist dabei glücklicherweise nicht nötig.

Ladet euch das entsprechende Zip-Archiv von `https://github.com/wildfly/wildfly/releases/download/29.0.1.Final/wildfly-29.0.1.Final.zip` herunter.

Da für alle Aufgabenblätter und die finale Webanwendung derselbe *WildFly* verwendet werden kann und sollte, empfiehlt es sich, das heruntergeladene Zip-Archiv in euer Verzeichnis `Implementierung` zu entpacken. Dort gibt es dann also neben dem `eksadat`-Verzeichnis ein weiteres Verzeichnis `wildfly-29.0.1.Final`.

Die Entwicklung der Webanwendung kann mit einer Vielzahl an Werkzeugen erfolgen (streng genommen geht das sogar mit *BlueJ*) – wir empfehlen allerdings dringend die professionelle IDE IntelliJ IDEA Ultimate[7] (ab jetzt einfach *IntelliJ* genannt) und gehen im Folgenden davon aus, dass ihr diese IDE in einer Version größer oder gleich `2023.2` verwendet. Der Einsatz anderer Werkzeuge ist übrigens explizit erlaubt – ihr könnt dann allerdings keine Unterstützung durch die Tutor:innen erwarten und müsst euch um die Konfiguration etc. selbst kümmern.

Die *ULTIMATE*-Edition von IntelliJ IDEA ist mit fast 600 Euro für das erste Jahr recht teuer, aber die Entwicklung einer Webanwendung ist in der kostenlosen *Community*-Edition nur rudimentär unterstützt und bereitet wenig Freude. Glücklicherweise könnt ihr euch bei Jetbrains (dem Hersteller von IntelliJ IDEA) mit eurer `@uni-bremen`-Email-Adresse einen Account erstellen, der zur kostenlosen Nutzung von IntelliJ IDEA Ultimate (und anderer IDEs desselben Herstellers) im Rahmen des Studiums berechtigt.[a]

[a] `https://account.jetbrains.com/login`

[5] Der Name ergibt sich aus dem dänischen Wort *Eksamensdato*, das eine mögliche Übersetzung des Begriffs *Prüfungstermine* ist.

[6] `https://www.wildfly.org/`

[7] `https://www.jetbrains.com/idea/`

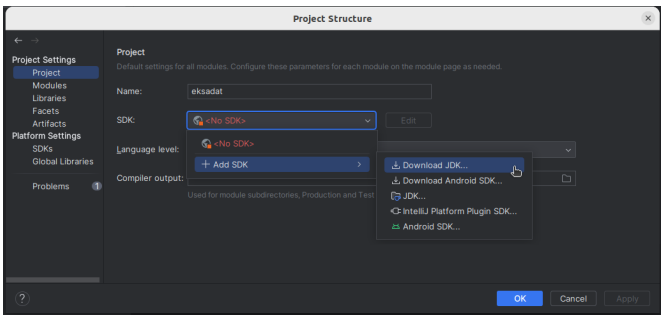
Eksadat ist eine Webanwendung, die mit dem *build tool* `Maven`[8] (später mehr dazu) erstellt wurde. Eine solche Anwendung kann in den meisten IDEs direkt geöffnet und verwendet werden. Wählt also in IntelliJ *open* (um ein Projekt zu öffnen), navigiert zu eurem `Eksadat`-Verzeichnis und wählt es zum Öffnen aus. Jetzt dauert es – je nach System und Auslastung länger oder kürzer – und dann ist das Projekt in IntelliJ geöffnet und indexiert.

IntelliJ sollte von sich aus erkennen, dass es sich um eine Webanwendung handelt und alle notwendigen Projektkonfigurationen automatisch vornehmen.

⚠ IntelliJ wird euch vorschlagen, diverse Plugins (wie *Jakarta EE: Batch Applications* o. ä.) zu installieren – falls die in eurer IntelliJ-Installation nicht ohnehin schon vorhanden sind. Diese Pugins werden **nicht** zum Bauen und Ausführen der Webanwendung benötigt. Ihr solltet lediglich das Plugin `Jakarta EE: Server Faces (JSF)` installieren, da es die Arbeit mit dem Frontend erleichtert. In einigen Fällen gab es mit der Installation der anderen Plugins diverse Probleme, so dass wir davon abraten, die Plugins an dieser Stelle zu installieren. Bei Bedarf können sie später noch installiert werden.

Es kann an dieser Stelle passieren, dass IntelliJ sich beschwert, dass es keine Projektdateien im Projekt findet. Das ist jetzt erstmal unwichtig, denn wir bauen das Projekt zunächst, indem wir im Menü `Build` → `Rebuild Project` auswählen.

ⓘ Je nach Konfiguration eurer Plattform und IntelliJ kann es jetzt vorkommen, dass ihr ein Pop-Up-Fenster seht, indem euch gemeldet wird, dass kein SDK (***S**oftware **D**evelopment **K**it*) für das Modul `eksadat` konfiguriert ist. Das bedeutet, dass IntelliJ auf eurem System kein passendes JDK für die Java-Version 17 gefunden hat. Klickt dann entweder auf den ebenfalls im Fenster angezeigten Link `Configure...` oder wählt im Menü `File` → `Project Structure`. Im sich öffnenden Fenster (siehe Abbildung) wählt ihr links `Project` und klickt dann das Auswahlfeld `SDK` an. Dann wählt ihr den Punkt `Add SDK`. Solltet ihr bereits ein passendes JDK für die Version 17 auf eurem System haben, dann wählt ihr den Punkt `JDK` und teilt IntelliJ im folgenden Dialog den Speicherort des JDKs mit. Solltet ihr kein passendes JDK haben, dann wählt ihr wie in der Abbildung den Punkt `Download JDK`.



Wählt im sich öffnenden Fenster die Version `20` und ein `Oracle Open JDK 20...`. Wir möchten zwar eigentlich Java 17 verwenden, aber höhere JDK-Versionen können in Java Quellcode eines niedrigeren Sprachlevels übersetzen und entsprechende Binaries erzeugen. Den vorgeschlagenen Speicherort für das herunterzuladende JDK ändert ihr nur, wenn ihr sehr genau wisst, was das bedeutet und was ihr tut. Wenn das neue JDK heruntergeladen wurde, wählt ihr erneut `Build` → `Rebuild Project`.

[8] `https://maven.apache.org/`

Wenn das Projekt erfolgreich gebaut werden konnte, soll eure Webanwendung direkt über IntelliJ auf euren *WildFly* deployt werden. Dazu müsst ihr eine entsprechende Run-Konfiguration erstellen.

Die folgenden Abbildungen können je nach IntelliJ-Version und Installation etwas anders aussehen. Das grundsätzliche Vorgehen sollte aber identisch sein.

Wählt im Menü `Run` → `Debug`. Im sich öffnenden Pop-Up-Fenster wählt ihr dann den Punkt `Edit Configurations...`. Im folgenden Fenster fügt ihr durch Anklicken des Plus-Buttons eine neue Konfiguration auf Basis eines Templates hinzu (vgl. Abbildung 1):

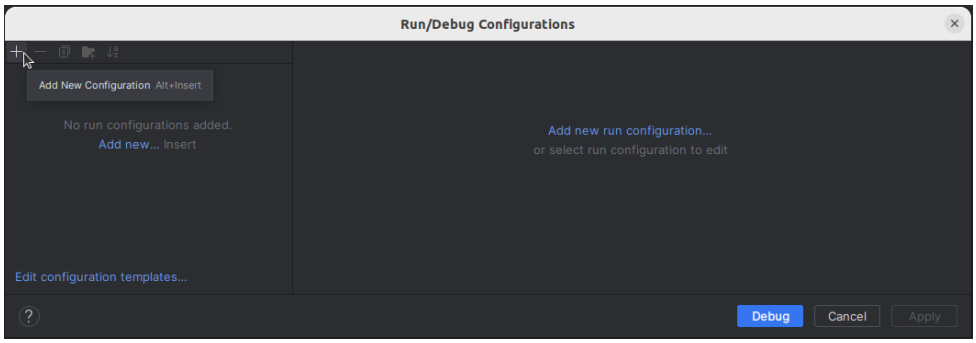


Abbildung 1: Eine Konfiguration auf Basis eines Templates erstellen.

In dem folgenden Dialog müsst ihr nun das Template `JBoss/WildFly Server` → `Local` auswählen. (vgl. Abbildung 2):

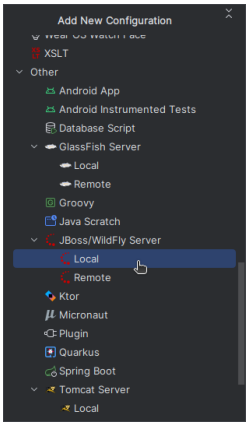


Abbildung 2: Konfiguration auf Basis des Templates *JBoss/WildFly Server* → *Local*.

WildFly ist die Community-Version von JBoss (vormals *JBoss AS*).

Nehmt in dem sich nun öffnenden Dialog (vgl. Abbildung 3) zunächst folgende Konfigurationen vor: Tragt unter *configure...* das Hauptverzeichnis eurer WildFly-Installation ein (1). Ihr könnt außerdem den Browser angeben, der beim Start eurer Webapplikation geöffnet werden soll (2).

Wenn ihr anschließend auf *Fix* klickt (3), erscheint der in Abbildung 4 dargestellte Dialog (es findet ein automatischer Wechsel auf den *Deployment*-Tab statt).

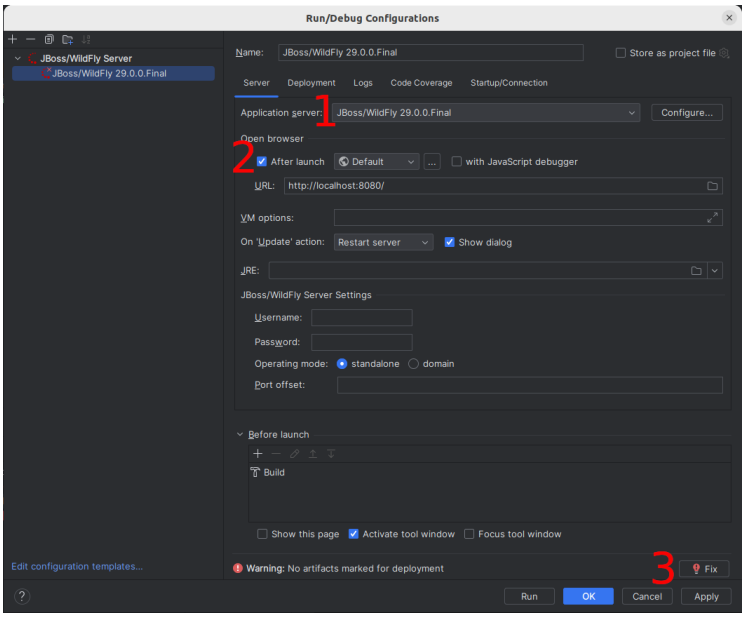


Abbildung 3: Run-Konfiguration einrichten.

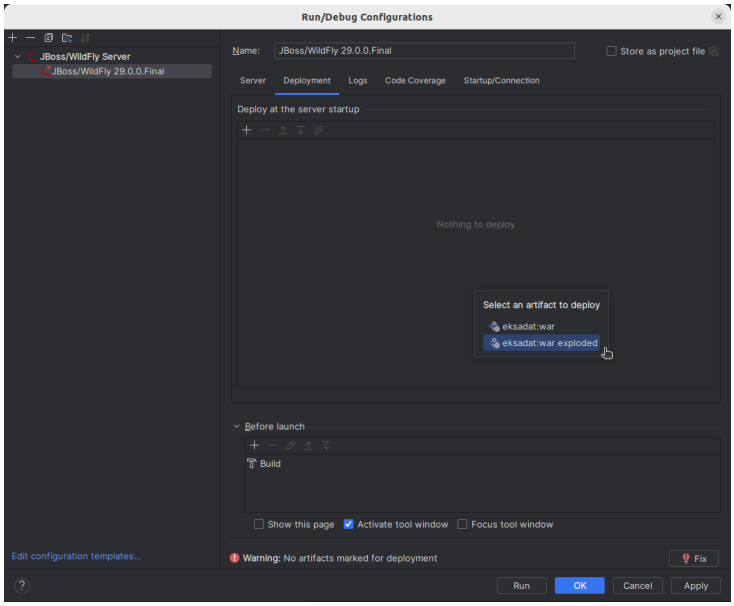


Abbildung 4: Hot-Deployment auswählen.

Hier müsst ihr angeben, welches Artefakt deployed werden soll. *Exploded* klingt zwar etwas merkwürdig, ist hier jedoch die richtige Wahl. Mit *exploded* ist gemeint, dass nicht das eigentliche Web-Archiv deployed wird, sondern eine Referenz auf euer `target`-Verzeichnis (dort wird durch Maven die kompilierte Fassung eurer Webanwendung abgelegt). Der Vorteil von *exploded* ist, dass IntelliJ somit einzelne Teile eurer Anwendung neu übersetzen und in `target` ablegen kann. Anschließend kann IntelliJ ein sogenanntes *Hot-Deployment* durchführen, d. h. nur die geänderten Teile eurer Software werden neu geladen. Dieses Feature ist während der Entwicklung extrem

hilfreich, da ein komplettes Erstellen des Web-Archivs und dessen Deployment deutlich mehr Zeit kosten!

Nachdem ihr das zu deployende Artefakt (exploded) hinzugefügt habt, wechselt zurück auf den *Server*-Tab. Der Inhalt dieses Tabs sollte nun wie in Abbildung 5 dargestellt aussehen:

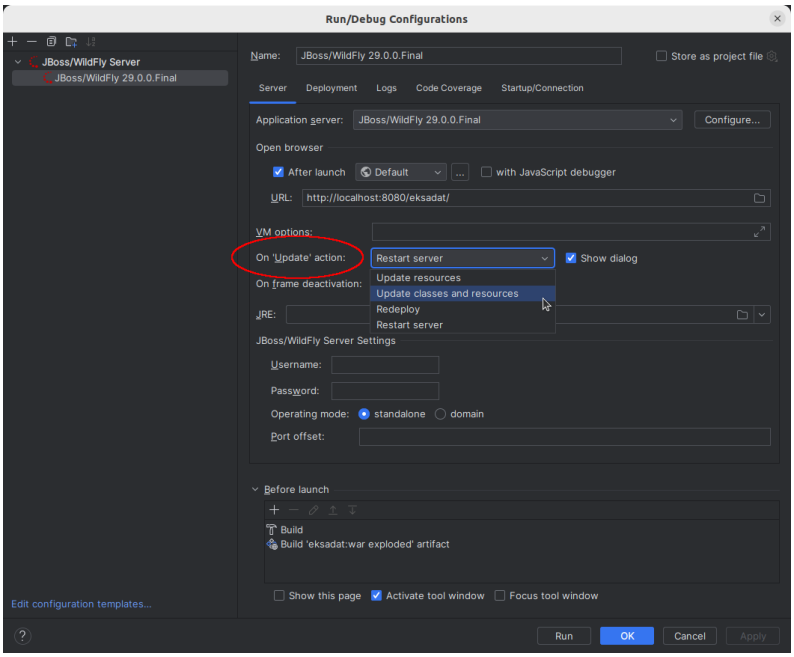


Abbildung 5: Hot Deployment aktivieren.

Um Hot-Deploy standardmäßig zu aktivieren, setzt `On 'Update' action` auf `Update classes and resources` (rot markierter Kreis). Ihr könnt zusätzlich auch `On frame deactivation` (direkt darunter) auf `Update classes and resources` setzen. Dadurch wird ein Hot-Deployment immer dann durchgeführt, sobald IntelliJ den Mausfokus verliert (z. B. wenn ihr von IntelliJ auf euren Browser wechselt). Wir belassen es allerdings üblicherweise bei `Do nothing`. Schließt die Konfiguration dann mit einem Klick auf den `OK`-Button ab.

Um WildFly zu starten und eure Anwendung zu deployen, klickt ihr auf den Debug-Button (der kleine grüne Käfer neben eurer Run-Konfiguration). Es empfiehlt sich, den WildFly immer im Debug-Modus zu starten. Nur in diesem Modus kann IntelliJ ein Hot-Deployment durchführen. Außerdem könnt ihr eure Anwendung nur in diesem Modus debuggen (wie der Name des Modus vermuten lässt). Startet ihr eure Anwendung im Run-Modus und stellt später fest, dass ihr doch einen Breakpoint setzen und nutzen wollt, müsst ihr den Server erst stoppen und über den Debug-Modus neu starten. Achtet außerdem darauf, dass ihr den Server nicht bereits in einem anderen Kontext (z. B. einer Shell) gestartet habt.

Wenn ihr alles korrekt umgesetzt habt, sollte der Server wie eben beschrieben gestartet werden können. Nach dem Deployment (das kann einige Sekunden dauern) sollte sich der von euch konfigurierte Browser mit einem neuen Tab und der Adresse `http://localhost:8080/eksadat/` öffnen und eine nicht sehr ansehnliche Webseite anzeigen. Neben einer Willkommensnachricht seht ihr dort einen Text, der eine Anzahl von Klicks (initial 0) anzeigt und ein Button mit dem Text `click`. Wenn ihr den Button anklickt, erhöht sich die angezeigte Klick-Anzahl auf 1. Weitere Klicks bewirken allerdings keine Änderung. Ladet ihr die Seite allerdings neu oder in einem anderen Browser, so steht die Klick-Anzahl wieder auf 0.

Das Projekt kennenlernen (und verstehen?!?)

Wir wollen uns jetzt zunächst mit den Bestandteilen des Projektes auseinandersetzen und eine grobe Vorstellung davon entwickeln, wie eine Webanwendung konzeptionell funktioniert.

Maven

Im Hauptverzeichnis von *eksadat* findet ihr die Datei `pom.xml` (von Studis früherer Jahrgänge liebevoll *Pommes* genannt). Die Datei `pom.xml` stellt die Konfigurationsdatei eines Maven-Projektes dar. Der Name ergibt sich aus *Project Object Model* (POM) und beschreibt die Struktur, Abhängigkeiten, Build-Konfigurationen, Ressourcen und andere Informationen über das Projekt. Es dient als Blaupause für das Projekt und ermöglicht es Maven, das Projekt zu verwalten und zu bauen.

Das POM spielt eine zentrale Rolle in Maven, da es die Grundlage für die Build-Automatisierung und die Verwaltung von Abhängigkeiten bildet. Durch die Verwendung einer einheitlichen POM-Struktur kann Maven Projekte konsistent und effizient erstellen und Abhängigkeiten verwalten.

Durch die Trennung von Konfigurationsdetails vom Quellcode ermöglicht es Maven auch die Wiederverwendung von Build-Skripten über mehrere Projekte hinweg und erleichtert die Zusammenarbeit und Integration in Build-Servern oder IDEs.

Die `pom.xml` in *eksadat* beginnt mit den folgenden Zeilen:

```
1 <project xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2          xmlns="http://maven.apache.org/POM/4.0.0"
3          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
4                              http://maven.apache.org/xsd/maven-4.0.0.xsd">
5    <modelVersion>4.0.0</modelVersion>
```

Wie die Datei-Endung vermuten lässt, handelt es sich hierbei um eine XML-Datei (*e**X**tensible **M**arkup **L**anguage*).[9] Das ist eine Textdatei, die strukturierte Daten in einem hierarchischen Format speichert. XML wurde entwickelt, um Informationen auf eine Weise darzustellen, die von Maschinen leicht verarbeitet und von Menschen leicht gelesen und verstanden werden kann. Es ist eine Metasprache, die verwendet wird, um benutzerdefinierte Auszeichnungssprachen zu erstellen.

Die grundlegende Syntax einer XML-Datei besteht aus sogenannten *Tags*, die von spitzen Klammern (< und >) umschlossen sind und eine Baumstruktur ergeben, in der verschachtelte Elemente Hierarchien bilden. Ein öffnendes Tag (z. B. `<modelVersion>` in Zeile 5) zeigt den Start eines *Zweiges* an, der durch ein schließendes Tag (hat den gleichen Namen wie das öffnende Tag, allerdings mit einem vorangestellten Slash (/)) beendet wird (z. B. `</modelVersion>` ebenfalls in Zeile 5).

So wie ein Baum häufig eine Wurzel hat (vgl. *PI 2*), hat auch ein XML-Dokument ein Wurzelelement bzw. ein Wurzel-Tag. Im Fall einer Maven-Projektkonfigurationsdatei lautet das Wurzelelement `project`. Ein Tag kann optional Attribute enthalten, über die es ergänzend konfiguriert werden kann. Die Attribute werden dann in der Form *attributname* = *attributwert* zwischen dem Tag-Namen und der schließenden spitzen Klammer notiert. Mehrere Attribute werden dabei durch Whitespaces getrennt.

In den Zeilen 1 bis 4 sehen wir, dass das Wurzel-Tag drei Attribute hat, nämlich `xmlns:xsi`, `xmlns` und `xsi:schemaLocation`.

[9] `https://www.xml.com/pub/a/98/10/guide0.html`

`xmlns:xsi` ist ein Attribut, das die XML-Namespace-Deklaration für das ***X**ML **S**chema **I**nstance* (XSI) angibt. Es wird verwendet, um das XML Schema für das Dokument anzugeben, das die Struktur und Validierungsregeln für die Elemente und Attribute der XML-Datei definiert.

Das `xmlns:xsi`-Attribut wird normalerweise im Wurzelelement einer XML-Datei deklariert und enthält den Namespace-URI (***U**niform **R**esource **I**dentifier*) für XSI. Der in Zeile 1 verwendete Wert `http://www.w3.org/2001/XMLSchema-instance` wird häufig verwendet.

Das `xmlns:xsi`-Attribut ermöglicht die Verwendung von XSI-spezifischen Attributen in der XML-Datei, die für die Validierung und Datentyp-Definition nützlich sind. Eines der häufig verwendeten XSI-spezifischen Attribute ist `xsi:schemaLocation`, mit dem der Ort des XML-Schemas für das Dokument angegeben werden kann.

Durch die Verwendung von XML-Schemas kann die Struktur und Validität einer XML-Datei überprüft werden, um sicherzustellen, dass sie den erwarteten Anforderungen entspricht.[10] Darüberhinas können fortgeschrittene IDEs basierend auf dem Schema erweiterte Funktionalität wie z. B. Autovervollständigung anbieten.

Das Attribut `xmlns` (***XML N**amespace*) ist ein spezielles Attribut in XML-Dateien, das verwendet wird, um den Standard-Namespace für die Elemente in der XML-Datei anzugeben. Es wird normalerweise im Wurzelelement der XML-Struktur deklariert und ermöglicht die Verwendung von benutzerdefinierten oder standardisierten Elementen und Attributen.

Der Zweck von XML-Namespaces besteht darin, Elementnamen in XML-Dokumenten zu unterscheiden, die aus verschiedenen Quellen oder Anwendungen stammen, und so Konflikte zwischen identisch benannten Elementen zu vermeiden. Indem ein eindeutiger Namespace mit dem `xmlns`-Attribut definiert wird, wird sichergestellt, dass Elemente mit denselben Namen, aber unterschiedlichen Namespaces, unterschiedliche Bedeutungen haben können.

In der hier betrachteten XML-Datei hat das Wurzelelement den Namespace mit dem `xmlns`-Attribut auf `http://maven.apache.org/POM/4.0.0` festgelegt. Dadurch gelten alle Elemente innerhalb des Wurzelelements (also innerhalb des *project*-Zweiges) automatisch für diesen Namespace – es sei denn, es wird ein neuer Namespace innerhalb eines bestimmten Elements definiert.[11]

In den Zeilen 7 bis 11 werden grundlegende Informationen über das Projekt festgelegt. Diese Angaben sind wichtig, um das Projekt zu identifizieren und zu dokumentieren.

```
7     <groupId>de.unibremen.cs.swp.eksadat</groupId>
8     <artifactId>eksadat</artifactId>
9     <name>Eksadat</name>
10    <version>0.1a</version>
11    <packaging>war</packaging>
```

Die `groupId` sollte ein möglichst eindeutiger Bezeichner sein, der einen Namensraum für all eure Projekte darstellt. Es hat sich die Konvention etabliert, die Domain der entsprechenden Organisation in umgekehrter Reihenfolge anzugeben. Da ihr (im Rahmen von SWP) keiner formalen Organisation untersteht, verwendet wir hier einfach die Fake-Domain `de.unibremen.cs.swp`. Maven prüft nicht, ob es die angegebene Domain bzw. `groupId` auch tatsächlich gibt, daher kann hier grundsätzlich etwas beliebiges angegeben werden. Die `artifactId` dient zur eindeuti-

[10] Dies ist besonders nützlich, wenn Daten zwischen verschiedenen Systemen ausgetauscht werden, da XML-Schemas die einheitliche Interpretation der Daten sicherstellen können.

[11] Die Verwendung von XML-Namespaces ist besonders wichtig, wenn verschiedene XML-Dokumente in einem Kontext zusammengeführt werden oder wenn XML in Kombination mit anderen Technologien wie XSLT oder XML-Schema verwendet wird.

gen Identifizierung des erstellten Artefakts (z. B. JAR-Datei, WAR-Datei, Bibliothek), das aus dem Projekt erzeugt wird. Das Tag `name` legt den Namen eures Projektes und das Tag `version` die Version fest. Als Version des Projektes wählen wir hier 0, weil das Projekt weit entfernt von einer ersten auslieferungsfähigen Version ist, 1 für das erste Aufgabenblatt und a für die Tatsache, dass es sich um die Basisfassung für die Bearbeitung eines **A**ufgabenblattes handelt.

Webanwendungen, die in einen Jakarta EE Container (ab jetzt der Einfachheit halber *Container* genannt) deployed werden, müssen als `WAR` (***W**eb **Ar**chive*) verpackt werden. Dazu muss der Eintrag `packaging` auf `war` gesetzt werden (Zeile 11). Damit das ordnungsgemäß funktioniert, muss auch noch die Version des Maven WAR-Plugins korrekt gesetzt werden (siehe unten).

In der `pom.xml` können beliebige *Properties* (Eigenschaften) definiert werden. Jede Property muss in dem `properties`-Zweig des XML-Dokumentes spezifiziert werden:

```
13      <properties>
14          <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
15          <maven.compiler.source>17</maven.compiler.source>
16          <maven.compiler.target>17</maven.compiler.target>
17
18          <jakarta.version>10.0.0</jakarta.version>
19
20          <maven-war-plugin.version>3.3.2</maven-war-plugin.version>
21      </properties>
```

Wir legen hier fest, dass die Quelldateien im Encoding UTF-8 kodiert sind (ein sinnvoller Standard für alle Plattformen) und dass die Java-Quellcode-Version und die Version der erzeugten Binaries Java 17 entspricht.

Zusätzlich können wir im Properties-Zweig Variablen definieren und mit Werten belegen. Diese Variablen können dann im weiteren Dokument verwendet werden. Hier deklarieren und initialisieren wir Variablen für die zu verwendende Jakarta EE Version und die zu verwendende Version des Maven WAR Plugins (Zeilen 18 und 20).

Eine besondere Stärke von Maven ist die Verwaltung von Abhängigkeiten (und deren transitiven Abhängigkeiten). Dazu dient der *Dependencies*-Zweig:

```
23      <dependencies>
24          <dependency>
25              <groupId>jakarta.platform</groupId>
26              <artifactId>jakarta.jakartaee-api</artifactId>
27              <version>${jakarta.version}</version>
28              <scope>provided</scope>
29          </dependency>
30      </dependencies>
```

Maven übernimmt dann alle notwendigen Schritte, um die entsprechenden Abhängigkeiten einzubinden. Jeder `dependency`-Eintrag (also jede Abhängigkeit) ist wie folgt strukturiert:

groupId (benötigt) Der Namensraum der Organisation, die die konkrete Abhängigkeit (siehe `artifactId`) verwaltet.

artifactId (benötigt) Der Name der Abhängigkeit (innerhalb des Namensraums `groupId`).

version (optional) Die Version der Abhängigkeit. Wird keine Version angegeben, wird die aktuellste Version genommen. Auch wenn dieser Wert optional ist, ist es empfehlenswert,

immer eine konkrete Version anzugeben. Andernfalls kann es euch passieren, dass euer Projekt von einem auf den anderen Tag nicht mehr kompiliert werden kann, da sich die API einer eurer Abhängigkeiten geändert hat.

scope (optional) Gibt an, in welchem Teil des Maven-Lebenszyklus eine Abhängigkeit bereitgestellt werden soll. Eine Auflistung aller *scopes* findet ihr unter: `https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html#Dependency_Scope`.

Wir definieren in den Zeilen 24 bis 29 eine Abhängigkeit zu Jakarta EE in der Version, die wir durch die oben deklarierte Variable festgelegt haben (ein Variablenzugriff erfolgt durch `$` gefolgt vom Variablennamen in geschweiften Klammern).

Der Scope `provided` gibt an, dass die entsprechende Abhängigkeit zwar zum Zeitpunkt der Kompilation eingebunden wird, jedoch nicht Teil des finalen Archivs ist. Stattdessen wird erwartet, dass die durch die Abhängigkeit spezifizierten Komponenten (Klassen, Interfaces usw.) von der Laufzeitumgebung des Zielsystems (dem Container) bereitgestellt werden. Es wäre zwar technisch möglich, den Scope der oben gelisteten Abhängigkeit auf `compile` zu stellen, jedoch würde das keinen Unterschied machen. Wie der Name der Abhängigkeit vermuten lässt (`jakartaee-api`), handelt es sich hierbei nur um ein API, d. h. die Implementierung der Schnittstellen ist ohnehin nicht enthalten.

Im *build*-Zweig können Einstellungen für den Build-Prozess (das „Bauen“) wie z. B. Compiler-Optionen, Zielpfade, zu verwendende Plugins usw. festgelegt werden:

```
23      <build>
24          <finalName>eksadat</finalName>
25          <plugins>
26              <plugin>
27                  <groupId>org.apache.maven.plugins</groupId>
28                  <artifactId>maven-war-plugin</artifactId>
29                  <version>${maven-war-plugin.version}</version>
30              </plugin>
31          </plugins>
32      </build>
33  </project>
```

Normalerweise verwendet Maven automatisch eine Namenskonvention für das erstellte Artefakt, die auf den Projekt-Koordinaten (`groupId`, `artifactId` und `version`) basiert. Das Tag `finalName` ermöglicht es, diese Namenskonvention zu überschreiben und dem erzeugten Artefakt einen benutzerdefinierten Namen zu geben. Diesen benutzerdefinierten Namen legen wir in Zeile 24 fest.

Es folgt die Konfiguration von Plugins, die während der Bau-Phase zum Einsatz kommen (Zeilen 25 bis 31). Aktuell wird hier nur, wie oben bereits erwähnt, die konkrete Version des Maven WAR Plugins konfiguriert.

In Zeile 33 wird schließlich der `project`-Zweig durch das schließende Tag beendet.

Ganz allgemein arbeitet Maven nach dem Prinzip „convention over configuration“, d. h. es werden Vereinbarungen festgelegt, die dann nicht mehr konfiguriert werden müssen. Eine solche Vereinbarung ist z. B. die Verzeichnisstruktur eines Maven-Projekts:

- `pom.xml` Projektkonfiguration

- `src` Verzeichnis für Quelldateien
  - `main` Verzeichnis für Quellen des Produktivsystems
    - `java` Verzeichnis für Java-Quelldateien
    - `resources` Verzeichnis für Ressourcen
    - `webapp` Verzeichnis für Quelldateien (und Verzeichnisse) einer Webanwendung
  - `test` Verzeichnis für Quelldateien der Tests
    - `java` Verzeichnis für Java-Quelldateien der Tests
    - `resources` Verzeichnis für Ressourcen der Tests

Die Verzeichnisse für die Tests und für Ressourcen sind optional und können auch weggelassen werden.

HTTP und HTML

HTML steht für ***H**yper**t**ext **M**arkup **L**anguage* und ist eine Auszeichnungssprache, die zur Strukturierung und Gestaltung von Inhalten im World Wide Web verwendet wird. Es bildet die Grundlage für die Erstellung von Webseiten und ermöglicht es, Texte, Bilder, Links und andere Medien auf einer Webseite zu organisieren und zu formatieren. Webbrowser (wie z. B. Firefox, Chrome, Safari, ...) können HTML-Dokumente anfragen und sie parsen, um Webseiten entsprechend der Struktur und Formatierungen im zugehörigen HTML-Dokument anzuzeigen.

Die Kommunikation zwischen einem Webbrowser (kurz: Browser) und einer Webanwendung erfolgt in der Regel über HTTPS (***H**yper**t**ext **T**ransfer **P**rotocol **S**ecure*) oder während der Entwicklung über HTTP (*Hypertext Transfer Protocol*). Konzeptionell läuft das – grob – folgendermaßen ab:

1. **Der Browser sendet eine Anfrage (*Request*):** Wenn eine URL (***U**niform **R**esource **L**ocator*) in die Adressleiste des Browsers eingegeben oder auf einen Link geklickt wird, wird eine HTTP-Anfrage an den entsprechenden Server gesendet. Diese Anfrage enthält darüber, welche Ressource (z. B. eine HTML-Seite, ein Bild, ein Skript, ...) benötigt wird und welche Methode (z. B. GET, POST, PUT, DELETE) verwendet werden soll.

2. **Der Webserver empfängt die Anfrage:** Der Webserver ist ein Programm oder ein spezieller Server, der auf Anfragen von Browsern oder anderen Clients horcht. Er empfängt die HTTP-Anfrage von einem Browser und beginnt, sie zu verarbeiten.

3. **Die Webanwendung verarbeitet die Anfrage:** Die Webanwendung ist der Code oder die Software, die auf dem Webserver läuft und die Anfragen verarbeitet. Sie interpretiert die Anfrage, ruft die entsprechenden Daten z. B. aus einer Datenbank ab (falls erforderlich), verarbeitet die Geschäftslogik und generiert die gewünschte Antwort.

4. **Der Webserver sendet die Antwort (*Response*):** Nachdem die Webanwendung die Anfrage verarbeitet hat, generiert sie eine HTTP-Antwort (Response) mit den entsprechenden Daten und Informationen. Die Antwort enthält normalerweise den Statuscode, der den Erfolg oder das Scheitern der Anfrage angibt, sowie den Inhalt der gewünschten Ressource (z. B. HTML, JSON[12], XML usw.).

[12] `https://www.json.org/json-en.html`

5. **Der Browser empfängt die Antwort:** Der Browser empfängt die HTTP-Antwort vom Server und beginnt, die erhaltenen Daten zu interpretieren und darzustellen. Wenn die Antwort eine HTML-Seite enthält, rendert der Browser sie und zeigt sie auf dem Bildschirm an. Wenn es sich um andere Ressourcen handelt, wie z. B. Bilder oder Skripte, werden diese durch der Browser entsprechend verarbeitet.

Interaktive Elemente: Die HTML-Seite kann interaktive Elemente wie Formulare, Schaltflächen oder Links enthalten. Wenn eine Nutzer:in mit diesen Elementen interagiert, sendet der Browser erneut Anfragen an den Server, um weitere Daten abzurufen oder Aktionen auszuführen. Der Prozess wiederholt sich, und die Webanwendung reagiert entsprechend auf die neuen Anfragen.

In unserem Fall ist der Webserver die laufende WildFly-Instanz. Ein WildFly-Server ist also zunächst mal in der Lage, einen HTTP-Request entgegenzunehmen und zu verarbeiten.

Jakarta Faces

Jakarta Faces (früher Jakarta Server Faces, davor Java Server Faces (*JSF*)) ist ein Framework-Standard zur Entwicklung von grafischen Benutzeroberflächen für Webanwendungen. Basierend auf Servlets gehört Jakarta Faces zu den Webtechnologien der Jakarta EE. Mit Hilfe von Jakarta Faces (ab jetzt der Einfachheit halber kurz *Faces* genannt) können auf einfache Art und Weise Komponenten für Benutzerschnittstellen in Webseiten eingebunden und die Navigation definiert werden.

Web Deployment Descriptor

Unsere Webanwendung wird über einen *Web Deployment Descriptor*[13] konfiguriert. Dabei handelt es sich wieder um eine XML-Datei namens `web.xml`, die ihr im Verzeichnis `src/main/webapp/WEB-INF` findet. Sie hat folgenden Inhalt:

```
1  <web-app
2      xmlns="https://jakarta.ee/xml/ns/jakartaee"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
5                          https://jakarta.ee/xml/ns/jakartaee/web-app_6_0.xsd"
6      version="6.0">
7
8      <servlet>
9          <servlet-name>FacesServlet</servlet-name>
10         <servlet-class>jakarta.faces.webapp.FacesServlet</servlet-class>
11         <load-on-startup>1</load-on-startup>
12     </servlet>
13
14     <servlet-mapping>
15         <servlet-name>FacesServlet</servlet-name>
16         <url-pattern>*.xhtml</url-pattern>
17     </servlet-mapping>
18
19     <welcome-file-list>
20         <welcome-file>hello.xhtml</welcome-file>
```

[13] https://jakarta.ee/specifications/faces/4.0/jakarta-faces-4.0#web-application-deployment-descriptor

```
21      </welcome-file-list>
22
23      <context-param>
24          <param-name>jakarta.faces.FACELETS_SKIP_COMMENTS</param-name>
25          <param-value>true</param-value>
26      </context-param>
27
28  </web-app>
```

Das Wurzel-Tag dieser Datei ist offensichtlich `web-app`. In den Zeilen 2 bis 7 werden wieder in bekannter Weise Namespaces und verwendete Schemata konfiguriert.

In den Zeilen 8 bis 12 werden die *Servlets* deklariert, die in der Webanwendung verwendet werden. Servlets sind Java-Klassen, die HTTP-Anfragen verarbeiten und HTTP-Antworten generieren können. Wir verwenden hier das von Jakarta EE ausgelieferte `FacesServlet`. Das `load-on-startup`-Element ist ein optionales Element und akzeptiert einen numerischen Wert (normalerweise eine positive Ganzzahl). Der Wert gibt die Priorität oder Reihenfolge an, in der das `FacesServlet` beim Starten der Webanwendung initialisiert werden soll. Ein Wert von 0 oder größer bedeutet, dass das FacesServlet beim Starten der Anwendung initialisiert wird. Ein negatives oder nicht angegebenes `load-on-startup`-Element bedeutet, dass das `FacesServlet` erst geladen und initialisiert wird, wenn es zum ersten Mal angefordert wird (*on-demand*).

Wir deklarien das `FacesServlet` mit `load-on-startup`-Wert 1. Dadurch wird festgelegt, dass das `FacesServlet` beim Starten der Webanwendung direkt initialisiert wird. Die Faces Framework-Ressourcen werden vorab geladen, bevor die erste Faces-Seite angefordert wird. Dies kann die Leistung verbessern, da das Framework bereits vorbereitet ist, um Anfragen zu verarbeiten, wenn sie eintreffen.

In den Zeilen 14 bis 17 wird das Mapping von Anfragen konfiguriert, d. h. es wird festgelegt, welchen URL-Patterns (Muster) jedes Servlet zugeordnet ist, sodass sie auf bestimmte HTTP-Anfragen reagieren können. Hier wird festgelegt, dass eine Anfrage, die auf `.xhtml` endet vom `FacesServlet` unserer Webanwendung verarbeitet wird.

In den Zeilen 19 bis 21 werden die Willkommensseiten konfiguriert. Die `welcome-file-list` gibt an, welche Dateien als Startseite oder Willkommensseite der Webanwendung verwendet werden sollen, wenn der Browser die Basis-URL der Anwendung öffnet (z. B. `http://localhost:8080/eksadat/`). Der Webserver wird versuchen, die Dateien in der angegebenen Reihenfolge zu finden und die erste Datei anzeigen, die vorhanden ist. In unserem Fall gibt es nur eine Willkommensseite, nämlich `hello.xhtml`.

Über den Zweig `context-param` können wir diverse Aspekte unserer Faces-Webanwendung konfigurieren. Wir stellen hier in den Zeilen 24 bis 25 erstmal nur sicher, dass auskommentierte Bereiche in unseren zu erstellenden Webseiten auch wirklich auskommentiert sind und nicht von Faces interpretiert werden. Das erspart enorm viel Verwirrung in der späteren Entwicklung.

Bean-Archive-Descriptor

Im Verzeichnis `src/main/webapp/WEB-INF` findet ihr den *Bean Archive Descriptor* in Form einer Datei namens `beans.xml`. In diesem Descriptor kann der CDI-Mechanismus (***C**ontexts and **D**ependency **I**njection*[14]) sehr spezifisch konfiguriert werden. Die Datei enthält folgendes:

[14] Java-spezifische Umsetzung von *Dependency Injection*:
`https://en.wikipedia.org/wiki/Dependency_injection`

```
1 <beans xmlns="https://jakarta.ee/xml/ns/jakartaee"
2       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3       xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
4                           https://jakarta.ee/xml/ns/jakartaee/beans_4_0.xsd"
5       version="4.0" bean-discovery-mode="annotated"
6 >
7 <!-- CDI configuration here. -->
8
9 </beans>
```

Die beiden Attribute in Zeile 5 sind Standard-Werte und könnten auch weggelassen werden (IntelliJ schlägt das auch vor). Tatsächlich ist die gesamte Datei optional, aber da wir sie später benötigen, haben wir sie schon mal angelegt. Version 4 ist die aktuelle Version von Jakarta CDI.[15] Mit `bean-discovery-mode="annotated"` wird konfiguriert, dass der Container beim Start der Webanwendung nach Klassen mit annotierten CDI-Scopes (dazu gleich mehr) und EJBs suchen soll und nur diese als durch CDI *injizierbar* (dazu später mehr) vermerken darf.

Über diese Datei können wir z. B. konkrete Implementierungen von Interfaces austauschen oder *Interceptoren* usw. konfigurieren.

Konfiguration der Webanwendung

Die Datei `faces-config.xml`[16] im Verzeichnis `src/main/webapp/WEB-INF` erlaubt spezifische Einstellungen für die Webanwendung. Hier können z. B. Navigationsregeln hinterlegt werden oder eigene *Validatoren* konfiguriert werden, mit denen Eingaben von Benutzer:innen validiert werden. Die Datei ist optional, ist aber im aktuellen Projekt schon mal leer angelegt, falls sie später verwendet werden soll:

```
1 <faces-config xmlns="https://jakarta.ee/xml/ns/jakartaee"
2               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3               xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
4                       https://jakarta.ee/xml/ns/jakartaee/web-facesconfig_4_0.xsd"
5               version="4.0"
6 >
7 <!-- Faces configuration here. -->
8
9 </faces-config>
```

Das bisher einzige Facelet

Im Verzeichnis `src/main/webapp` findet ihr das bisher einzige *Facelet*:[17] `hello.xhtml`. Ein Facelet ist eine spezielle Art von Vorlage (Template). Es wird verwendet, um die Benutzeroberfläche einer Faces-Anwendung zu definieren und zu gestalten. Es handelt sich dabei um eine Erweiterung von HTML.

[15] `https://jakarta.ee/specifications/dependency-injection/2.0/`
[16] `https://jakarta.ee/specifications/faces/4.0/jakarta-faces-4.0#a6195`
[17] `https://jakarta.ee/specifications/faces/4.0/jakarta-faces-4.0#a5476`

```
1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2                        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3  <html lang="en"
4        xmlns:h="jakarta.faces.html"
5  >
```

In den Zeilen 1 bis 2 wird der Typ des Dokumentes festgelegt. Das Wurzel-Tag einer XHTML-Seite ist wie bei einer HTML-Seite das `html`-Tag. Die Sprache des Dokumentes setzen wir zunächst auf Englisch (Zeile 3).

Die Komponenten eines Facelets sind in *Namespaces* unterteilt. Um die Komponenten eines Namespaces in einem Facelet verfügbar zu machen, muss der entsprechende Namespace *importiert* (Zeile 4) und an einen *Namen* gebunden werden (`h` in Zeile 4). Anschließend können die Komponenten mit XML-Tags der Form `<Name:Komponente...` eingebunden werden (in Zeile 13 unten wird z. B. ein Ausgabetext mit `<h:outputText...` eingebunden).

```
7      <h:head>
8          <title>Eksadat</title>
9      </h:head>
```

Eine typische HTML-Seite hat einen Kopf-Bereich (*head*). Er enthält Metadaten, die Informationen über die Seite bereitstellen und das Verhalten der Seite steuern. Der Kopfbereich (Zeilen 7 bis 9) wird nicht im Browserfenster angezeigt, sondern beeinflusst die Darstellung und das Verhalten der Seite. Aktuell ist hier nur der Titel der Seite eingetragen (Zeile 8). Der Titel wird z. B. im Browsertab direkt angezeigt.

Führt eure Run-Konfiguration für die Webanwendung aus (über den Marienkäfer im Debug-Modus) und ladet die Willkommensseite. Der Browsertab und möglicherweise das gesamte Fenster sollte den konfigurierten Titel *Eksadat* anzeigen. Ändert den Titel in Zeile 8 jetzt auf einen beliebigen Wert. Klickt dann den Marienkäfer erneut an und wählt im sich öffnenden Dialog `Update ressources`. Damit werden geänderte Ressourcen-Dateien (und die Facelets sind in diesem Kontext Ressourcen) hot-deployed. Wenn ihr die Seite jetzt neu ladet, dann sollte im Browsertab der geänderte Titel angezeigt werden. Ohne hot-deploy hätte erst über Maven ein neues Web-Archive gebaut und auf den Server deployt werden müssen!

```
11     <h:body>
12         <h1>Welcome to Eksadat!</h1>
13         <h:outputText id="output" value="Number of clicks:
           ↪ #{helloController.noOfClicks}"/>
14         <h:form>
15             <h:commandButton value="click"
               ↪ action="#{helloController.increaseClicks}"/>
16         </h:form>
17     </h:body>
18
19 </html>
```

Die roten, gebogenen Pfeile am Zeilenanfang direkt unterhalb der Zeilen 13 und 15 zeigen an, dass der folgende Text zur vorigen Zeile gehört, aber nicht mehr in die Textbox passt. Daher haben diese Zeilen auch keine eigene Zeilennummer.

Der `body`-Bereich (Zeilen 11 bis 17) enthält alle Elemente, die auf der Webseite sichtbar sein sollen, wie Text, Bilder, Hyperlinks, Tabellen, Formulare und andere HTML-Elemente. Hier wird der eigentliche Inhalt der Webseite platziert, der den Nutzer:innen präsentiert wird.

Wir setzen dort zunächst mal einen Text als Überschrift (mit dem HTML-Tag `<h1>` in Zeile 12). Darunter wird eine Komponente für eine Textausgabe angezeigt (Zeile 13). Die Komponente bekommt die ID *output*. Eindeutige und lesbare IDs können später beim Testen nützlich sein. Alle Komponenten einer Webseite haben IDs – wenn wir keine vergeben, werden sie vom Container generiert. Dem Attribut `value` wird der Text zugewiesen, der angezeigt werden soll. Hier handelt es sich um eine Kombination aus der Zeichenkette *Number of clicks:* gefolgt von einem EL-Ausdruck (*Expression Language*[18]). Dynamische Teile, wie z. B. der Aufruf einer Java-Methode, werden durch EL-Ausdrücke realisiert (`#{...}`). Eine Besonderheit von EL ist, dass Attribute, die über eine Getter- und/oder eine Setter-Methode verfügen, direkt aufgerufen werden können (Attribute mit Getter- und/oder Setter-Methoden werden auch *Properties* genannt). In Zeile 13 wird durch den EL-Ausdruck die Methode *getNoOfClicks()* eines Objektes aufgerufen, das den Namen `helloController` trägt. Dieser Name entspricht dem Namen der Klasse, allerdings mit einem Kleinbuchstaben vorne. Die Erzeugung des Objektes übernimmt der Container und im Falle mehrerer Objekte des entsprechenden Typs ordnet er dem Facelet das jeweils korrekte Objekt zu.

IntelliJ ermöglicht euch die direkte Navigation zu der Methode. Klickt z. B. den Namen der Methode (`.noOfClicks`) bei gleichzeitig gehaltener CTRL-Taste (macOS: CMD-Taste) an und es öffnet sich die Klassendatei und der Cursor steht auf der entsprechenden Methode. Das funktioniert analog für die Klasse selbst, indem ihr mit gehaltener CTRL-Taste auf `helloController` klickt.

Zu der Klasse mit ihren Methoden kommen wir gleich. Wir machen erstmal mit dem Facelet weiter.

Da das Facelet eine Interaktion mit einer Nutzer:in ermöglichen soll (hier der Klick auf eine Schaltfläche), benötigen wir ein Formular (Zeilen 14 bis 16). Darin befindet sich eine Komponente für die besagte Schaltfläche (`h:commandButton` in Zeile 15). Das Attribut `value` spezifiziert den Text auf der Schaltfläche und das Attribut `action` die Aktion, die beim Klicken ausgeführt werden soll. In diesem Fall ist die Aktion durch einen EL-Ausdruck spezifiziert und bedeutet, dass die Methode `increaseClicks()` eines Objektes mit dem Namen `helloController` aufgerufen wird.

Controller Bean

Controller Beans (häufig auch *Backing Beans* genannt) sind Java-Klassen, die als Vermittler zwischen Backend (Jakarta EE) und Frontend (Faces) dienen. Sie stellen den Facelets Daten zur Verfügung und bieten Methoden zur Interaktion an. Derartige Controller-Klassen sind in der Regel sehr einfach aufgebaut und delegieren hauptsächlich zwischen den Interaktionen aus der Oberfläche und den Komponenten des Backends. Eine Controller-Klasse enthält daher typischerweise keine Logik. Die Geschäftslogik sollte in einer anderen Komponente implementiert werden, schon um das Prinzip „Entwurf nach Zuständigkeiten“ nicht zu verletzen.

[18] `https://jakarta.eaworld.io/specifications/expression-language/5.0/jakarta-expression-language-spec-5.0.html`

Eine Java Bean ist eine Java-Klasse, die bestimmte Konventionen befolgt, um einfachen Zugriff und Manipulation ihrer Eigenschaften (*Properties*) zu ermöglichen. Der Begriff *Java Bean* wurde von Sun Microsystems (jetzt Oracle Corporation) geprägt und bezieht sich auf eine spezielle Art von wiederverwendbaren und standardisierten Komponenten in Java-Anwendungen.
Eine Java Bean sollte u. a. folgende Eigenschaften erfüllen:

Standard-Default-Konstruktor: Die Klasse sollte einen öffentlichen, parameterlosen Standard-Konstruktor haben, damit sie einfach instanziiert werden kann.

Eigenschaften (Properties): Die Bean sollte private Instanzvariablen (Fields) haben, die als Eigenschaften (Properties) fungieren. Für jede Eigenschaft sollten Methoden bereitgestellt werden, um den Zugriff und die Aktualisierung der Eigenschaften zu ermöglichen.

Zugriffsmethoden: Die Methoden zum Zugriff auf die Eigenschaften sollten den Java-Beans-Konventionen entsprechen. Der Name des Getter- und Setter-Methodenpaars sollte mit *get* oder *is* für den Getter und *set* für den Setter beginnen, gefolgt vom Namen der Eigenschaft mit dem ersten Buchstaben großgeschrieben.

In unserem Projekt gibt es bisher nur die Controller Bean `HelloController` zum Facelet `hello.xhtml`.

```
1  package de.unibremen.cs.swp.eksadat.controller;
2
3  import jakarta.annotation.PostConstruct;
4  import jakarta.enterprise.context.RequestScoped;
5  import jakarta.inject.Named;
6
7  /**
8  * Controller for the hello facelet. Provides an attribute for the number of
9  * clicks and a method to increase said number.
10 *
11 * @author K. Hölscher
12 * @version 2023-07-20
13 */
14 @Named
15 @RequestScoped
16 public class HelloController {
```

Das Basispaket für alle Klassen innerhalb von *eksadat* haben wir ja bereits in der `pom.xml` festgelegt. Es empfiehlt sich, die jeweiligen Klassen des Projektes in Unterpakete nach Zuständigkeiten aufzuteilen. Dieser Idee folgend, sollten sich Controller-Klassen im Unterpaket `controller` befinden (Zeile 1).

Die Annotation `@Named` (Zeile 14) exponiert die Bean zur Verwendung in Facelets. Durch die Annotation wird diese Klasse zu einer *Managed Bean*, was bedeutet, dass wir uns um die Instanziierung und den Lebenszyklus nicht selbst kümmern müssen – das übernimmt der Container für uns. Den Zugriff auf eine solche Bean haben wir im Hello-Facelet bereits gesehen.

Wenn uns der Name der Klasse für den Zugriff im Facelet nicht gefällt, dann können wir den Namen durch Setzen eines Attributwertes der Annotation konfigurieren, z. B. `@Named(value="hello")`. Im Facelet könnten wir dann Objekte dieser Klasse über den EL-Ausdruck `#{hello.}` ansprechen.

Die Annotation `RequestScoped` (Zeile 15) legt die Lebensdauer der Bean auf einen Request fest. Das bedeutet, dass für jeden HTTP-Request, der auf das Facelet verweist, eine neue Bean erzeugt (ein neues Objekt der Klasse instanziiert) wird. Wir werden gleich noch andere Scopes kennenlernen.

Die Klasse deklariert nur ein einziges Attribut, nämlich für die Anzahl an Klicks:

```
18      /**
19       * The number of clicks.
20       */
21      private int noOfClicks;
```

Statt eines explizit implementierten Konstruktors (wir erinnern uns an PI 1: in diesem Fall erzeugt der Java-Compiler einen parameterlosen Konstruktor) gibt es eine Methode zur Initialisierung des Attributwertes:

```
23      /**
24       * Initializes the number of clicks with the value 0 after instantiation.
25       */
26      @PostConstruct
27      public void init() {
28          noOfClicks = 0;
29      }
```

Erwähnenswert ist hier die Annotation `@PostConstruct`. Sie garantiert, dass die damit annotierte Methode genau ein Mal im Lebenszyklus der Bean aufgerufen wird und zwar nachdem diese vollständig initialisiert wurde (inklusive aller Abhängigkeiten, die möglicherweise über CDI hergestellt werden müssen) und bevor sie das erste Mal verwendet wird. Eine solche Methode ist der Initialisierung im Konstruktor vorzuziehen, da die Container-gesteuerten Abhängigkeiten im Konstruktor noch nicht aufgelöst sind. Zudem kann es sein, dass eine Bean vom Container mehrfach instanziiert wird (z. B. wenn es eine Proxy-Klasse dafür gibt, was im Kontext von CDI üblich ist) und dann die Initialisierung mehrfach durchgeführt werden würde. Das wiederum kann unnötig teuer sein, wenn eine Initialisierung z. B. das Netzwerk oder eine Datenbank verwendet.

Es folgt eine konventionsgemäße Getter-Methode für die Anzahl der Klicks:

```
31      /**
32       * Returns the number of clicks.
33       *
34       * @return the number of clicks.
35       */
36      public int getNoOfClicks() {
37          return noOfClicks;
38      }
```

Zuletzt die Methode für das Erhöhen der Klick-Anzahl um den Wert Eins:

```
40      /**
41       * Increases the number of clicks by 1.
42       */
43      public void increaseClicks() {
44          ++noOfClicks;
45      }
```

Damit sind alle mitgelieferten Dateien des Projekts erläutert, bis auf die leere Datei `.gitkeep` im Verzeichnis `src/main/resources`. Diese Datei sorgt dafür, dass das Verzeichnis in einem Git-Repository angelegt wird, denn Git kann ja nur Dateien unter Versionskontrolle stellen. Der Name `.gitkeep` hat sich als Standard etabliert – letztlich könnt ihr so eine Datei nennen, wie ihr möchtet.

Ablauf

Wenn vom Browser eine Anfrage auf der URL `http://localhost:8080/eksadat` kommt, dann wird diese Anfrage zunächst vom WildFly entgegengenommen. Aufgrund des Aufbaus der URL „weiß“ der WildFly jetzt, dass es eine Anfrage für die Web-Applikation *eksadat* ist und leitet sie entsprechend an diese Web-Applikation weiter. In dessen Deployment-Descriptor steht, dass als Willkommensseite das Facelet `hello.xhtml` ausgeliefert werden soll. Ebenso steht dort, dass ein Facelet mit der Endung `.xhtml` vom Faces Servlet verarbeitet wird. Das Faces Servlet sorgt jetzt dafür, dass eine Antwort für die Anfrage durch Faces generiert wird. Es muss also aus dem Facelet eine HTML-Seite für den anfragenden Browser erzeugt werden. Dabei muss mindestens der EL-Ausdruck `#{helloController.noOfClicks}` ausgewertet werden.

Dazu wird beim CDI-Mechanismus eine entsprechende Bean angefragt. Da die zugehörige Bean-Klasse den Scope *RequestScope* hat, wird auf jeden Fall eine neue Instanz der Bean erzeugt, d. h. die Anzahl der Klicks wird auf 0 gesetzt. Diese Anzahl wird dann über den Aufruf der Methode `getNoOfClicks()` ermittelt, das Ergebnis in die HTML-Seite an der passenden Stelle eingesetzt und die HTML-Seite als Antwort auf die Anfrage an den Browser zurückgesendet. Nun rendert der Browser die Seite, so dass Nutzer:innen sie anschauen können.

Wenn eine Nutzer:in dann die Schaltfläche anklickt, wird dadurch ein POST-Request an den WildFly gesendet. Das führt dazu, dass über den EL-Ausdruck in der `action` der Schaltfläche die Methode `increaseClicks()` auf einer Bean vom Typ `HelloController` aufgerufen wird. Dazu wird allerdings zunächst eine derartige Bean benötigt. Wie oben erzeugt der CDI-Mechanismus eine neue Bean-Instanz (wegen Request Scope) und gibt sie zurück. Dann wird die Methode aufgerufen und der Wert der Anzahl an Klicks auf 1 erhöht. Die HTML-Seite wird dann wieder aufgebaut und die Getter-Methode für die Anzahl an Klicks liefert jetzt 1. Diese 1 wird entsprechend eingesetzt und die HTML-Seite wieder an den Browser zurückgesendet. Dieser rendert sie und die Nutzer:innen sehen jetzt eine 1 bei der Anzahl an Klicks.

Weitere Klicks erhöhen allerdings die Anzahl 1 nicht. Wir wissen jetzt auch, warum das so ist, denn jeder Klick ist ein neuer Request, der wie oben abläuft für jeweils eine neue Controller Bean. Wird die Seite ohne Klick neu geladen, steht die Anzahl auf 0.

Aufgabe 1 Aktuelle Version unter Versionskontrolle stellen

Spätestens jetzt ist es an der Zeit, den aktuellen Projektstand unter Versionskontrolle zu stellen, d. h. sie in das bereitgestellte Git-Repository zu pushen. Ihr könnt das direkt über IntelliJ erle-

digen oder mit einem externen Tool eurer Wahl. Wichtig ist, dass ihr nur die Quelldateien unter Versionskontrolle stellt. Es empfiehlt sich, eine Datei `.gitignore` im Implementierungsverzeichnis anzulegen mit folgendem Inhalt:

```
1 wildfly-29.0.1.Final
2 target
3 *.iml
4 .idea
```

Damit ignoriert Git das WildFly-Verzeichnis, das von Maven generierte target-Verzeichnis mit den erzeugten Dateien für das Web-Archiv und die IntelliJ-Projektdateien. Letztere können ja lokale Einstellungen einzelner Entwickler:innen enthalten, daher sollten sie im Normalfall nicht unter Versionskontrolle gestellt werden.

⚠ Wenn ihr andere als die benötigten Dateien unter Versionskontrolle stellt, wird eure Tutor:in euch abmahnen.

Aufgabe 2 Logging für Arme

Um zu sehen, was „hinter den Kulissen“ der Webanwendung passiert, können wir entsprechende Aktionen protokollieren (*loggen*). Ihr habt vielleicht schon bemerkt, dass sich der Bereich `Server` (üblicherweise unten rechts) in IntelliJ öffnet, wenn ihr den WildFly startet. Darin seht ihr die Protokoll-Nachrichten („das Log“) des WildFly-Servers (siehe Abbildung 6).

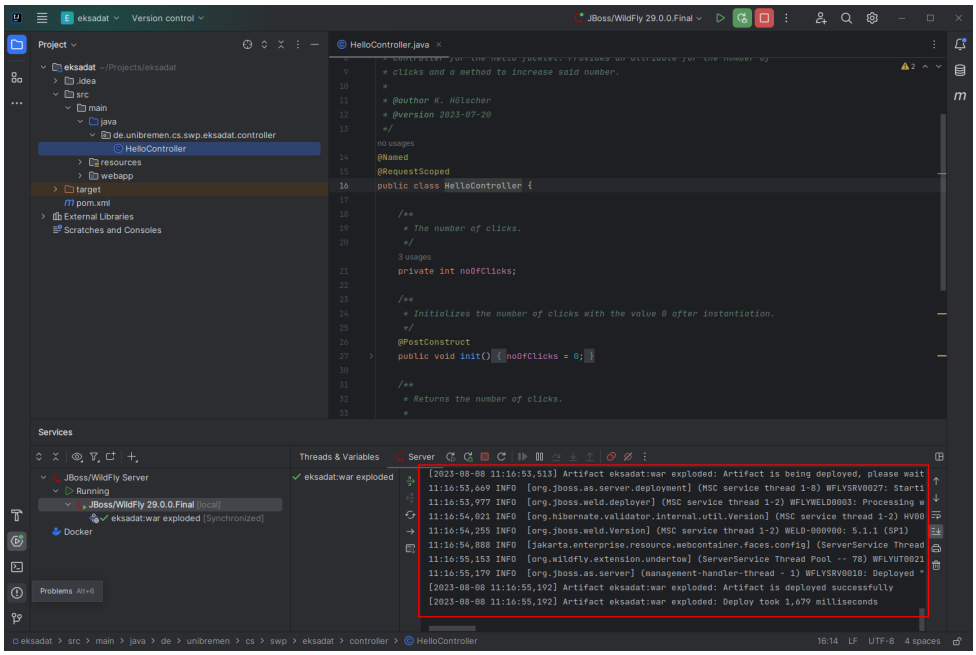


Abbildung 6: Protokoll-Nachrichten des WildFly in IntelliJ

Das liegt daran, dass der Protokoll-Mechanismus von WildFly derart konfiguriert ist, dass die Log-Nachrichten nicht nur in eine Datei sondern zeitgleich auch auf die Konsole geschrieben werden. IntelliJ zeigt uns hier den Inhalt der Ausgabekonsole an.

Wenn ihr jetzt in der Bean eine Ausgabe auf die Standardkonsole durchführt, dann wird diese vom WildFly entgegengenommen und in das Protokoll umgeleitet. Probiert das aus, indem ihr eine Ausgabe in die Methode *init()* einbaut und dann die Webseite ladet, klickt, erneut klickt usw. Ihr seht dann im Log des Servers, dass die Systemausgaben mit Uhrzeit und dem *Loglevel* (dazu später mehr) *INFO* versehen werden. Ändert eure Ausgabe, indem ihr auf den Standard-Fehlerkanal schreibt (`System.err`). Das Loglevel ändert sich dadurch auf *ERROR*.

Aufgabe 3 Klicks pro Webanwendung

Ändert den Scope der Bean `HelloController` zu `@SessionScoped`.[19] Der *Session Scope* definiert die Lebensdauer der entsprechenden Bean derart, dass sie so lange lebt wie eine Session und für die entsprechende Session zugreifbar ist. Eine typische Anwendung für diesen Scope ist ein Warenkorb in einem Webshop. Der Warenkorb soll ja über mehrere Anfragen derselben Session den konkreten Inhalt speichern. Hätte der Warenkorb den *Request Scope*, dann könntet ihr immer ein Teil hineinlegen, aber schon beim nächsten Anzeigen des Warenkorbes wäre er wieder leer. Dadurch würde der gesamte Einkauf natürlich billiger (weil ihr nie Produkte im Warenkorb habt), aber irgendwie auch sinnlos.

Zunächst stellen wir allerding fest, dass sich das Projekt jetzt nicht mehr übersetzen lässt. Das liegt daran, dass die Klasse `HelloController` jetzt den Scope *Session Scope* hat, aber nicht serialisierbar ist. Das lässt sich schnell beheben, indem ihr dafür sorgt, dass die Klasse das Interface `Serializable` implementiert.

Warum muss eine Bean, die eine längere Lebensdauer als *Request Scope* hat, serialisierbar sein? Das liegt daran, dass eine derartige Bean ja Zustände über Requests hinweg speichern soll. Wenn jetzt der Container unter Last gerät und möglicherweise nicht mehr genug Arbeitsspeicher hat, muss er Teile der Webanwendungen auf die Festplatte auslagern. Dazu wird dann einfach die Bean serialisiert und bei Bedarf wieder deserialisiert. Bedenkt dabei, dass ein einziger WildFly zahlreiche Webanwendungen hosten kann (auch wenn das in der Praxis eher unüblich ist).

Überprüft ihr jetzt das Verhalten der Webanwendung, so stellt ihr fest, dass die Klicks tatsächlich hochgezählt werden. Wenn ihr die Seite in einem anderen Tab mit demselben Browser ladet, wird euch auch dort die entsprechende Anzahl an Klicks angezeigt.

Das liegt daran, dass aufgrund des *Session Scope*s der Controller Bean zwischen eurem Browser und der Webanwendung eine Session etabliert wird. Jede weitere Anfrage desselben Browsers wird auf Server-Seite dadurch derselben Session zugeordnet. Es wird also jetzt nur eine Session Bean erzeugt und für jeden weiteren Request der Session wiederverwendet.

Ladet ihr die Seite jetzt in einem anderen Browser, so seht ihr dort die Klick-Anzahl 0. Ihr könnt jetzt in beiden Browsern die Klick-Anzahlen getrennt beeinflussen. Technisch existiert jetzt also pro Browser (genauer pro Session) eine eigene `HelloController`-Bean. Der CDI-Mechanismus in Kombination mit Faces sorgt dafür, dass dem jeweiligen Facelet die korrekte Bean zugeordnet wird.

Wir möchten jetzt aber alle Klicks zählen, die seit dem Start der Webanwendung durchgeführt werden. Dazu ändern wir den Scope auf `@ApplicationScoped`. Dieser Scope bedeutet, dass die Bean so lange lebt, wie die gesamte Webanwendung. Dadurch werden alle Klicks gezählt, egal

[19] Die entsprechend nötige import-Anweisung sollte IntelliJ korrekt erstellen.

in welchem Browser. Probiert das aus! Wenn der Server noch läuft von den vorherigen Überprüfungen, solltet ihr die Applikation neu deployen. Sonst bleiben eventuell etablierte Sessions hängen[20] und ihr seht keinen Effekt. Das ist ohnehin eine gute Grundregel: Wenn euch etwas seltsam vorkommt oder eine Änderung nicht den Erwartungen entspricht, dann deployt eure Webanwendung erneut oder startet sogar den WildFly neu.

Aufgabe 4 Logging einbauen und konfigurieren

Die Ausgabe auf der Konsole ist als Log-Mechanismus wenig professionell und unflexibel. Zudem werden unsere Log-Nachrichten dadurch im zentralen Log-File des WildFly-Servers gesammelt. Das ist spätestens dann nicht mehr sinnvoll, wenn der WildFly mehrere Webanwendungen hostet. Wir wollen daher jetzt „echtes“ Logging für unsere Webanwendung an einem eigenen Ort konfigurieren.

Das Logging dient dazu, später den Ablauf und die jeweiligen Systemzustände rekonstruieren zu können. Ein Logging-Framework erleichtert die Protokollierung, indem z. B. das Logging mit Zeitstempeln in Dateien (oder in eine Datenbank oder übers Netzwerk o. ä.) unterstützt wird. Ebenso kann in den gängigen Logging-Frameworks die Wichtigkeit der Protokollmeldung konfiguriert werden, da nicht alle Meldungen gleich wichtig sind. Bei der Untersuchung eines Fehlers kann dann z. B. nach Warnungen oder kritischen Fehlern gefiltert werden.

Wie so oft in der Java-Welt gibt es nicht nur ein Logging-Framework (Java bringt sogar einen eigenen Logging-Mechanismus mit – aber der ist wenig ausgereift[21]), sondern viele verschiedene, die sich mindestens in Details unterscheiden. Selbst wenn während der initialen Entwicklung einer Anwendung das Logging-Framework festgelegt wurde, kann es in der Wartungsphase passieren, dass das Logging-Framework gewechselt werden soll (z. B. weil es sicherheitsrelevante Fehler enthält[22] oder um Einheitlichkeit in der Entwicklungsabteilung zu haben oder auf Kund:innenwunsch o. ä.). Um den Wechsel zu erleichtern bzw. die Festlegung des konkreten Logging-Frameworks auf den Zeitpunkt der Auslieferung der Software zu verschieben, hat sich die ***S**imple **L**ogging **F**acade for **J**ava* (`slf4j`[23]) weitestgehend durchgesetzt. Während der Entwicklung wird gegen diese Fassade[24] programmiert und erst zum Auslieferungszeitpunkt entschieden, welches konkrete Logging-Framework verwendet wird. Das konkrete Logging-Framework kann dann auch ohne Änderungen am Quellcode ausgetauscht werden.

Wir legen uns jetzt hier willkürlich auf logback[25] als konkreten Logging-Mechanismus fest. Um Logback und die `sl4j`-Fassade nutzen zu können, tragen wir entsprechende Abhängigkeiten in unsere `pom.xml` ein – und zwar in den Zweig `dependencies`:

```
<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-classic</artifactId>
    <version>${logback.version}</version>
</dependency>
```

[20] Das ist ein eigentlich nützlicher Effekt von hot-deploy: Sessions bleiben typischerweise erhalten. Nach einem Redeployment sind Sessions nicht mehr gültig.

[21] Einige Gründe für diese Aussage findet ihr unter `https://devissuefixer.com/questions/why-not-use-javautillogging`

[22] `https://www.heise.de/news/Kritische-Zero-Day-Luecke-in-log4j-gefaehrdet-zahlreiche-Server-und-Apps-6291653.html`

[23] `https://www.slf4j.org/`

[24] Facade Pattern, siehe z. B. `http://www.blackwasp.co.uk/Facade.aspx`

[25] `https://logback.qos.ch/`

Die Version ist hier wie oben über eine Variable definiert – die müssen wir noch deklarieren und initialisieren. Wir möchten natürlich gerne die aktuellste Version nutzen – aber wie finden wir heraus, welche das ist? Dazu können wir z. B. im Maven Repository[26] den Suchbegriff *logback-classic* eingeben. Der erste Treffer sollte dann bereits die gesuchte Bibliothek beschreiben 7.

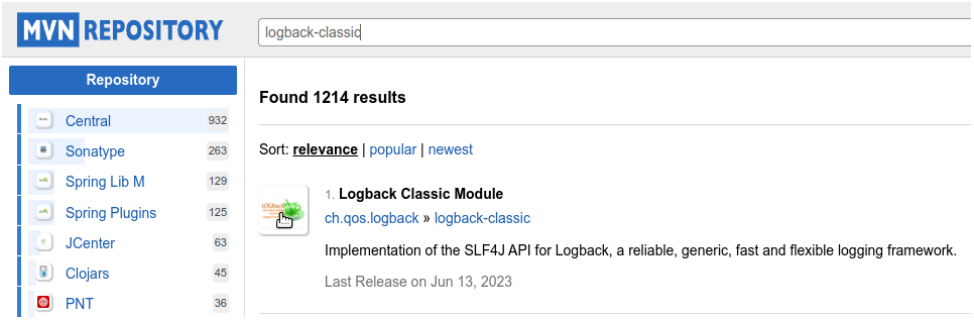


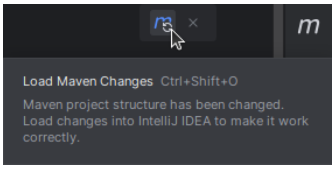
Abbildung 7: Suche nach logback im Maven Repository.

Den Treffer könnt ihr direkt anklicken und auf der folgenden Seite seht ihr dann alle im Maven Repository abgelegten Versionen. Wenn ihr dann auf die aktuellste Version klickt (zum Zeitpunkt der Erstellung des Aufgabenblattes die Version 1.4.8), dann öffnet sich eine weitere Seite auf der sich neben diversen Informationen zur Bibliothek auch der in die `pom.xml` einzutragende Text befindet (der auf Klick ins Clipboard kopiert wird). Das Feature benötigen wir jetzt aber nicht, da uns nur noch die konkrete Versionsnummer fehlt. Wir deklarieren also unsere noch fehlende Variable im `properties`-Zweig der `pom.xml`[27]:

```
<logback.version>1.4.8</logback.version>
```

Das genügt jetzt schon als Abhängigkeit, denn logback-classic bringt `slf4j` bereits mit.

Wenn ihr euer Projekt in einer IDE wie IntelliJ bearbeitet, dann gibt es ja immer zwei Projekte: einmal das Maven-Projekt, das IDE-unabhängig bearbeitet und konfiguriert werden kann und das Projekt innerhalb der verwendeten IDE. Änderungen an der `pom.xml` müssen daher dem IDE-Projekt mitgeteilt werden. IntelliJ bemerkt eine Änderung an der `pom.xml` und zeigt in der Standardeinstellung oben rechts im Editor-Fenster ein kleines Symbol an:



Wenn ihr das Symbol mit den beiden Pfeilen anklickt, werden die Änderungen an der `pom.xml` in das IntelliJ-Projekt übernommen. Besonders komfortabel ist, dass IntelliJ Änderungen an der `pom.xml` erkennt und automatisch einen Reimport durchführen kann. Ihr könnt dieses Feature für ein Projekt aktiveren, indem ihr unter `File` → `Settings` die Einstellungen aufruft (in macOS: `IntelliJ IDEA` → `Preferences`). Dort findet ihr den Punkt `Build, Execution, Deployment` und darunter den Unterpunkt `Build Tools`. Dort muss `Reload project after changes in the build scripts` angehakt sein und darunter `Any changes` ausgewählt sein (siehe Abbildung 8).

[26] `https://mvnrepository.com/repos/central`

[27] Wenn es zum Zeitpunkt der Bearbeitung des Aufgabenblattes eine neuere Version gibt, dürft ihr die selbstverständlich verwenden

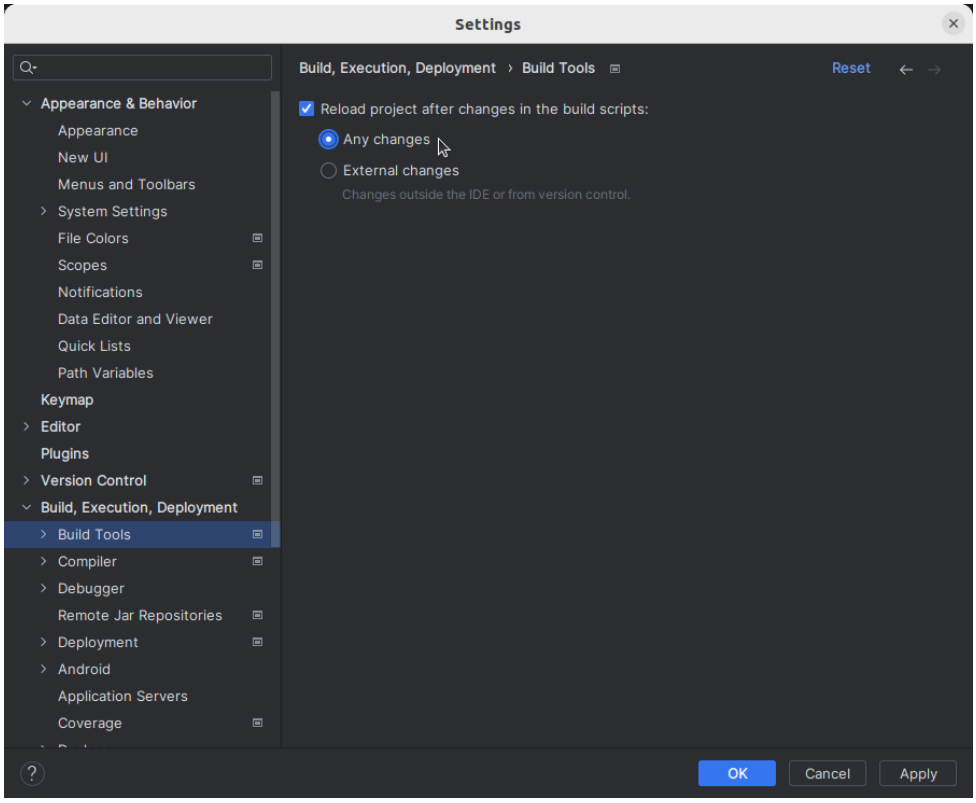


Abbildung 8: Einstellungen in IntelliJ für einen automatischen Reimport der Projekt-Konfiguration bei Änderungen in der `pom.xml`.

Wir möchten nun die Tatsache protokollieren, dass die Methode *init()* der Klasse `HelloController` aufgerufen wurde. Dazu passen wir die Klasse `HelloController` an. Wir fügen dort ein statisches Attribut für ein Logger-Objekt hinzu und initialisieren es direkt. Da sich alle Objekte einer Klasse das Logger-Objekt teilen, ist hier ein statisches Attribut sinnvoll.

```
private static final Logger logger =
 ↪ LoggerFactory.getLogger(HelloController.class);
```

Der Logger wird mit dem Klassenobjekt der umschließenden Klasse als Parameter erzeugt. Daraus ergibt sich der Name für den Logger. Das sollte immer so gehandhabt werden, damit klar ist, aus welcher Klasse eine Log-Nachricht kommt.

Der Klassenname `Logger` kommt sehr häufig vor. Es ist wichtig, dass ihr den Logger und auch die Factory aus dem Paket `org.slf4j` und also die entsprechenden import-Anweisungen
`import org.slf4j.Logger;`
`import org.slf4j.LoggerFactory;`
verwendet. Nur dann programmiert ihr gegen die `slf4j`-Fassade und könnt ggf. das konkrete Logging-Framework später problemlos austauschen. Hier sollte auf einen QuickFix (üblicherweise ALT-ENTER) verzichtet werden, da darüber häufig die falschen Klassen in den Namensraum importiert werden.

Das Logger-Objekt `logger` wird zum eigentlichen Protokollieren verwendet. Es wird nicht direkt instanziiert, sondern über eine *Factory*.[28] Das liegt daran, dass ja beim Entwickeln gar nicht klar ist, welches Framework konkret zum Einsatz kommt und wie dessen Logger-Objekt erzeugt wird.

Wir haben weiter oben erwähnt, dass Logging-Frameworks Nachrichten nach Wichtigkeit einstufen können und sollen. Welche Wichtigkeit hat nun die Tatsache, dass die *init()*-Methode aufgerufen wurde? Dazu hilft vielleicht der Vorschlag für Loglevels in Abbildung 9.[29]

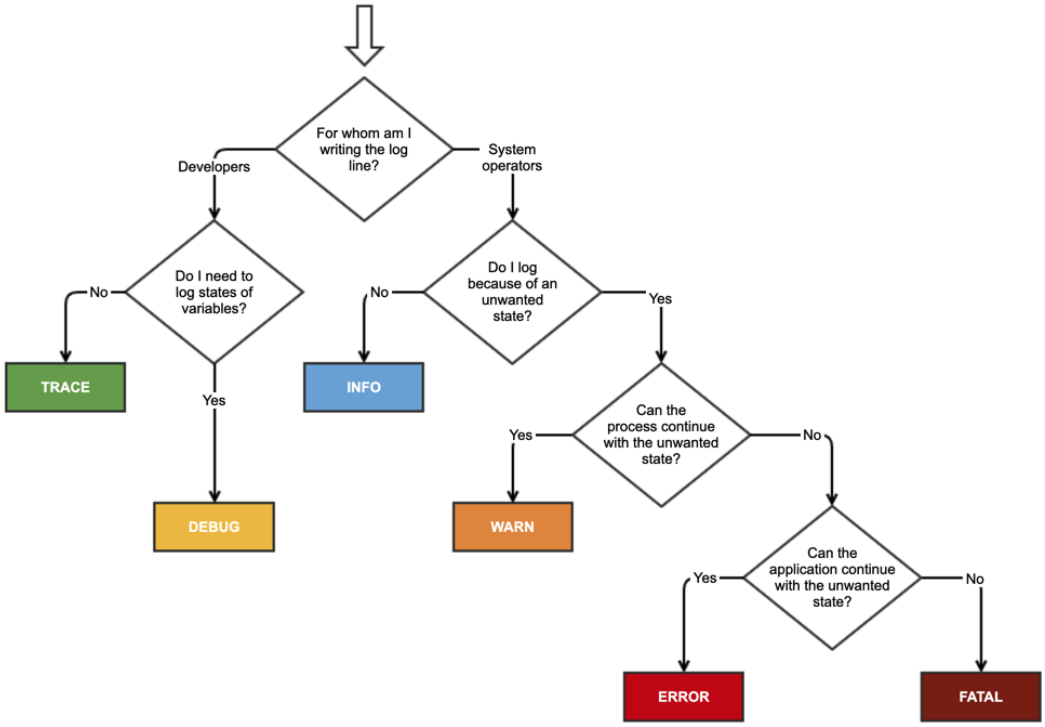


Abbildung 9: Auswahl des Loglevels

Aufgrund des Diagramms entscheiden wir uns für das Loglevel *trace*. Die in dem Diagramm aufgeführten Loglevels werden übrigens alle von `slf4j` (und logback) unterstützt (wobei das Level *FATAL* allerdings auf *ERROR* abgebildet wird).

Wir protokollieren also den Aufruf der Methode *init()*:

```
@PostConstruct
public void init() {
    logger.trace("Executing HelloController#init().");
    noOfClicks = 0;
}
```

Wir möchten jetzt auch noch den Aufruf der Methode *increaseClicks()* zusammen mit der aktuellen Zahl an Klicks protokollieren. Gemäß des Diagramms sollten wir dafür also das Loglevel *DEBUG* verwenden (da wir einen Variablenzustand protokollieren):

[28] Näheres dazu z. B. hier: `http://blackwasp.co.uk/FactoryMethod.aspx`
[29] `https://stackoverflow.com/questions/2031163/when-to-use-the-different-log-levels`

```
public void increaseClicks() {
    ++noOfClicks;
    logger.debug("Button has been clicked. Count is now {}.", noOfClicks);
}
```

Für jedes Loglevel stellt das Logger-Objekt eine entsprechende Methode bereit. Sie ist mehrfach überladen. Wir verwenden hier das Konzept des parametrisierten Loggings. Statt einen fertig zusammengebauten String (etwa über String-Konkatenation oder die Nutzung von `String.format` oder eines `StringBuilders`) übergeben wir einen Format-String mit Platzhalter in Form eines Paares von geschweiften Klammern. Wir können auch mehrere Platzhalter im Format-String definieren. Die Werte, die für die Platzhalter eingesetzt werden sollen, werden direkt durch Komma getrennt nach dem Format-String angegeben. Warum regeln wir das so?
Ganz einfach. Die Loglevels bilden eine Hierarchie:[30]

```
1  The six logging levels used by Log are (in order):
2
3  1. trace (the least serious)
4  2. debug
5  3. info
6  4. warn
7  5. error
8  6. fatal (the most serious)
```

Zur Laufzeit kann das kleinste interessierende Loglevel im Produktivsystem konfiguriert werden. Nehmen wir an, wir konfigurieren als kleinstes Loglevel das Level `warn`. Dann werden alle Protokollmeldungen mit einem kleineren Level (also `trace`, `debug` und `info`) nicht protokolliert. Das kann z. B. sinnvoll sein, um riesige Logdateien zu vermeiden oder auch die Performanz zu verbessern (da Logging in Dateien eine I/O-Operation und damit bekanntermaßen „Laufzeitteuer“ ist). Wenn nun aber die zu loggende Zeichenkette von uns bereits vor dem Aufruf der Logging-Methode zusammengebaut wird, dann ist das unter Umständen sinnlos, denn je nach konfiguriertem Mindestlevel wird sie eventuell gar nicht protokolliert. Wir haben dann aber Laufzeit und Speicher verschwendet, um eine Zeichenkette zusammenzubauen, die vom Log-Objekt verworfen wird.
In der oben verwendeten Variante wird die Zeichenkette erst dann zusammengebaut, wenn das Logger-Objekt sie tatsächlich auch protokolliert.
Wenn wir unser Projekt deployen, wird leider gar nichts geloggt. Das liegt daran, dass die Standard-Konfiguration des WildFly erwartet, dass eine von ihm gehostete Webanwendung den in WildFly eingebauten Log-Mechanimus nutzt. Das möchten wir nicht, daher konfigurieren wir unsere Webanwendung derart, dass das WildFly-Logging für sie deaktiviert wird. Dazu erstellen wir die Datei `jboss-deployment-structure.xml` im Verzeichnis `src/main/webapp/WEB-INF`. Sie hat folgenden Inhalt:

```
1  <jboss-deployment-structure>
2      <deployment>
3          <exclude-subsystems>
4              <subsystem name="logging" />
5          </exclude-subsystems>
6      </deployment>
7  </jboss-deployment-structure>
```

[30] `https://www.slf4j.org/api/org/apache/commons/logging/Log.html`

Durch diese Konfiguration wird das WildFly-*Subsystem* für das Logging für unsere Webanwendung exkludiert – wir müssen uns also selbst um das Logging kümmern.

Wir möchten in eine Datei mit einem von uns festgelegten Namen loggen und es soll für jeden Tag eine eigene Log-Datei geben. Es sollen die Log-Dateien der letzten 30 Tage aufbewahrt werden bis zu einer Größe von maximal 3 GByte. Falls diese Größe überschritten wird, werden die ältesten Log-Dateien gelöscht bis der Speicherverbrauch bei maximal 3 GByte liegt.

Um logback entsprechend zu konfigurieren, erzeugen wir die Datei `logback.xml` im Verzeichnis `src/main/resources`. Wir definieren darin unterhalb des Wurzel-Tags `configuration` einen entsprechenden *Appender*, der in eine Datei loggt:

```
1  <configuration>
2      <appender name="ROLLING_FILE"
       ↪ class="ch.qos.logback.core.rolling.RollingFileAppender">
3          <file>${JBOSS_HOME}/standalone/log/eksadat.log</file>
4          <rollingPolicy
           ↪ class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
5              <fileNamePattern>eksadat.%d{yyyy-MM-dd}.log</fileNamePattern>
6              <maxHistory>30</maxHistory>
7              <totalSizeCap>3GB</totalSizeCap>
8          </rollingPolicy>
9        <append>true</append>
10       <encoder>
11           <pattern>
12               %d{yyyy-MM-dd HH:mm:ss.SSS} %-5level %logger{36} - %msg%n
13           </pattern>
14       </encoder>
15   </appender>
```

In Zeile 2 vergeben wir einen eigenen Namen für unseren Appender. Die Basisklasse, die für den Appender verwendet werden soll entspricht der Klasse `FileAppender` im entsprechenden Paket.

In Zeile 3 legen wir den Speicherort unser Log-Datei fest. `${JBOSS_HOME}` ist dabei eine Umgebungsvariable, die vom WildFly entsprechend gesetzt wird. Darin steht der Pfad zum Speicherort der WildFly-Installation.

In den Zeilen 4 bis 8 legen wir die oben beschriebenen Aspekte fest: eine Log-Datei für jeden Tag (Zeilen 4 und 5) mit entsprechenden Dateinamen, die jüngsten 30 solcher Dateien sollen aufbewahrt werden (Zeile 6) bis zu einer maximalen Größe von 3 GByte (Zeile 7).

In Zeile 9 legen wir fest, dass Log-Nachrichten an eine vorhandene Log-Datei angehängt werden (das ist auch der Default-Wert).

In den Zeilen 10 bis 14 definieren wir unser eigenes Protokollmuster: Ein Logeintrag beginnt mit dem Datum und der Uhrzeit in Stunden, Minuten, Sekunden und Mikrosekunden. Es folgt das Loglevel linksbündig mit einer Breite von 5 Zeichen. Darauf folgt der Name des Loggers mit einer maximalen Breite von 36 Zeichen, dann ein in Leerzeichen eingeschlossenes Minuszeichen, dann die eigentliche Log-Nachricht und schließlich eine Zeilenschaltung.

Die Konfiguration kann sehr feingranular angepasst werden.[31]

Zusätzlich zum Speichern der Log-Nachrichten in die Dateien möchten wir während der Ent-

[31] `https://logback.qos.ch/manual/layouts.html#ClassicPatternLayout`

wicklung auch auf die Konsole loggen. Dadurch sehen wir ja unsere Protokoll-Nachrichten im Server-Tab in IntelliJ. Wir konfigurieren einen zweiten Appender dafür:

```
15      <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
16          <encoder>
17              <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} %-5level %logger{36} -
                ↪ %msg%n</pattern>
18          </encoder>
19      </appender>
```

Wir verwenden das gleiche Muster für die Log-Einträge wie oben.

Jetzt müssen wir nur noch einstellen, welcher Appender welches Loglevel protokolliert. Wir wählen einfach „alle“ für beide Appender:

```
21      <root level="all">
22          <appender-ref ref="STDOUT" />
23          <appender-ref ref="ROLLING_FILE" />
24      </root>
25  </configuration>
```

Wenn ihr das Projekt jetzt neu deployt, sollten entsprechende Log-Meldungen im Server-Tab in IntelliJ erscheinen – sowohl beim erstmaligen Laden der Seite als auch bei jedem Anklicken der Schaltfläche. Dort handelt es sich jetzt aber um Ausgaben auf die Konsole (durch unseren zweiten Appender), so dass sie von WildFly mit dem Level INFO geloggt werden. Unsere eigenen Log-Einträge beginnen tatsächlich erst mit der zweiten Uhrzeit-Angabe.

Zusätzlich wird allerdings in die Datei `eksadat.log` geloggt. Schaut sie euch an, indem ihr in das Installationsverzeichnis eures WildFlys navigiert und von dort aus in das Verzeichnis `standalone/log`.

Pusht die Änderungen in euer Gitlab-Repository und schließt damit das erste Aufgabenblatt erfolgreich ab.

Wir erwarten zum Bestehen des Aufgabenblattes, dass nur die wirklich für das Projekt nötigen Dateien in eurem Repository enthalten sind, dass das Projekt gebaut und deployt werden kann und dass die Aufgaben 1 bis 4 korrekt gelöst sind.

Spätestens nach der Erledigung dieses Aufgabenblattes sollten alle Gruppenmitglieder eine lokale Version des Projektes haben inkl. lokalem WildFly-Server, mit dem das Projekt ausgeführt und verwendet werden kann. Wenn das nicht frühzeitig sichergestellt ist, wird es schwierig bis unmöglich, produktiv am Projekt mitzuarbeiten.