FH Salzburg

Demo Infrastructure

Download Netbeans

Verwenden Sie die NetBeans Version 8.2 – Java EE! Diese können Sie hier herunterladen: https://netbeans.org/downloads/index.html

Dabei wählen Sie die Version wie in Abbildung 1 angegeben.

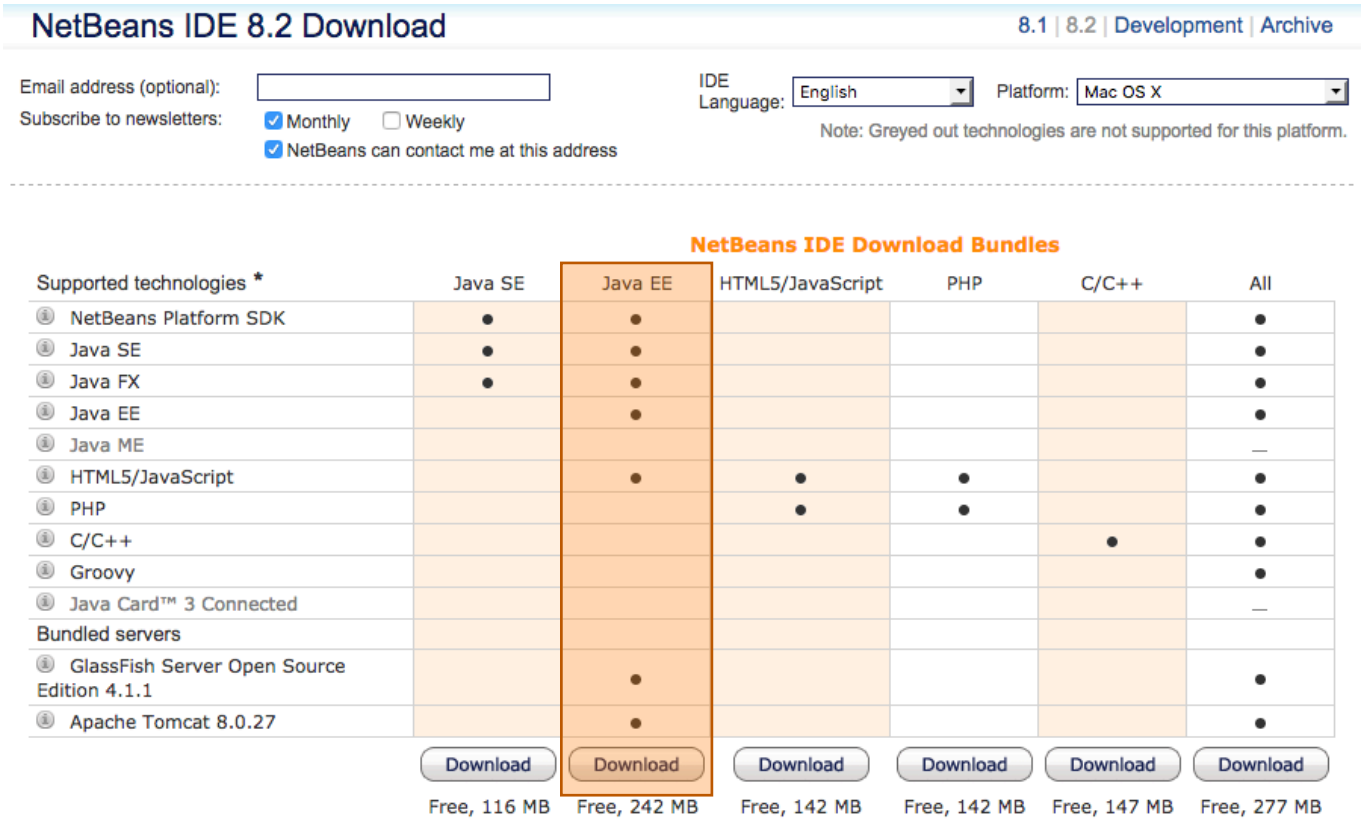

Abbildung 1: Netbeans Download

Hier können Sie jeweils die Version für Ihr System (Windows, MacOS X, Linux) herunterladen.

Installation

Die Applikation wird durch Ausführen der Setup-Datei gestartet. Dabei muss zunächst die Lizenzvereinbarung akzeptiert werden und es kann auch das Installationsverzeichnis angegeben werden.

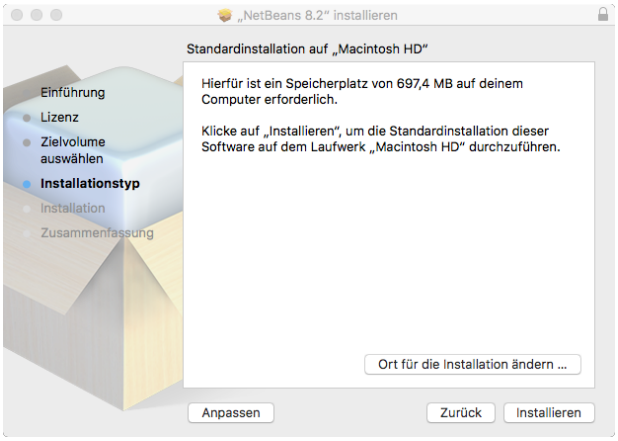

Abbildung 2: Installation Netbeans (1)

Wichtig: Die Java EE Version enthält die Glassfish-Version 4.1.1. Diese ist aber fehlerhaft!

Wir empfehlen daher bei der Installation von NetBeans auf die Installation der mitgelieferten Application Server zunächst zu verzichten, dies kann später nachgeholt werden.

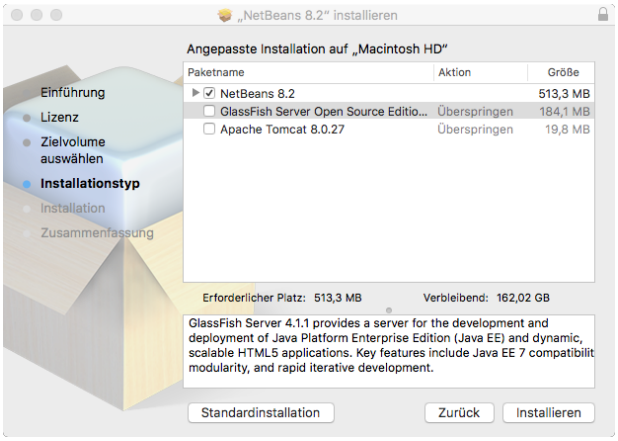

Abbildung 3: Installation Netbeans (2) - Anpassen

Sobald NetBeans installiert und geöffnet ist, präsentiert sich die Applikation in etwa wie in Abbildung 4 dargestellt. Wichtig ist an dieser Stelle ist Tab „Services" und die Eintrag „Servers" sowie „Databases"

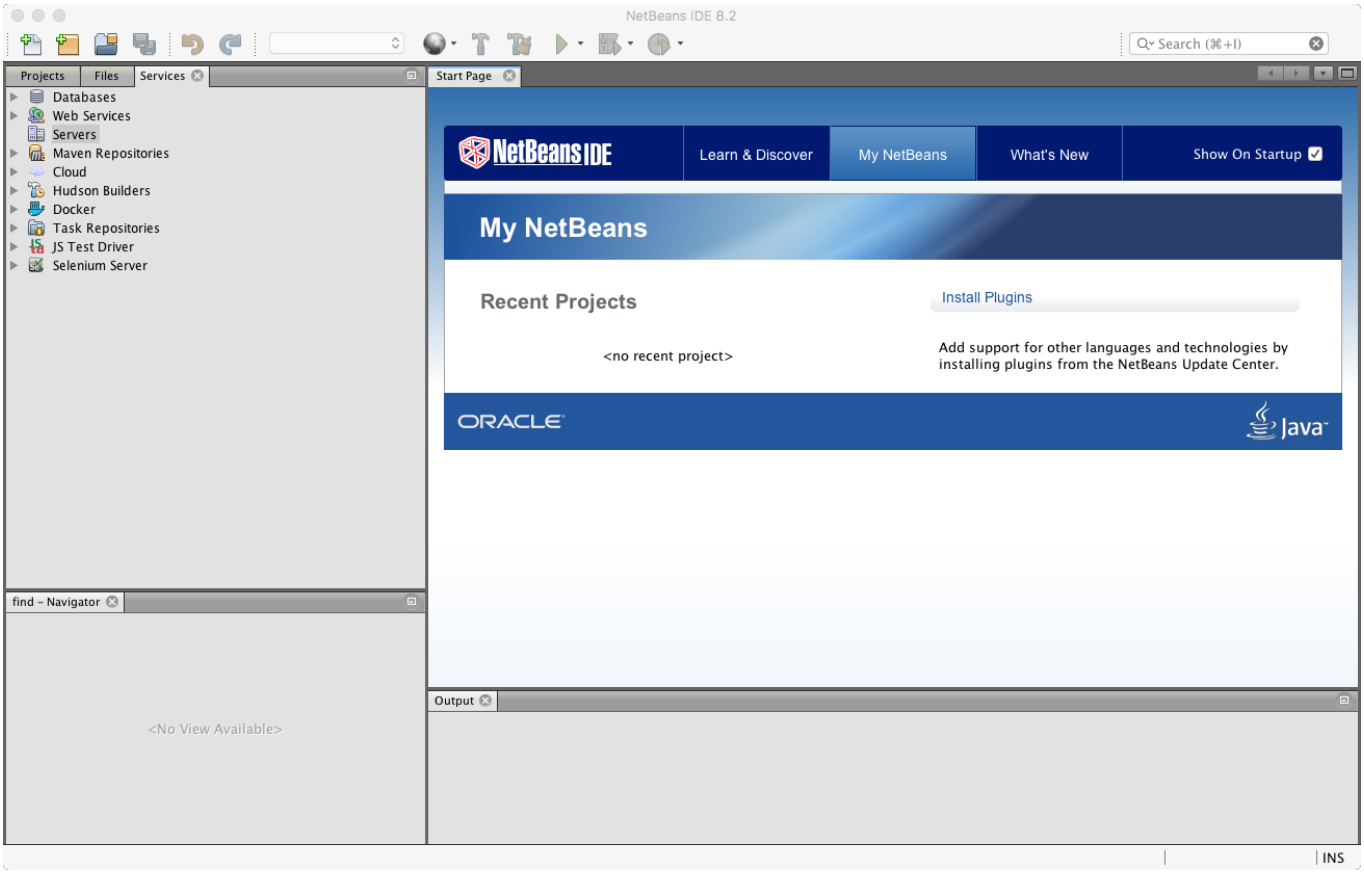

Abbildung 4: NetBeans Programmfenster

Installation und Konfiguration eines Servers

Achtung: Der ApplicationServer (Glassfish 4.0, Glassfish 5.0) benötigt JDK 8 u161. Neuere Java Versionen (aktuell ist JDK 11) können für den Server noch nicht verwendet werden. Ausnahme: Die Development-Version des Servers (Glassfish 5.0.1) funktioniert mit JDK 8 u181 bzw. auch JDK9. Diese kann hier http://download.oracle.com/glassfish/5.0.1/nightly bezogen werden.

Da wir während der Installation beide Application Server deaktiviert haben, müssen wir nun einen „brauchbaren“ Application Server hinzufügen und einbinden. Hierzu per Rechtsklick auf den Eintrag „Servers“ und den Menüpunkt „Add Server ...“ auswählen. Im Dialog „Add Server Instance“ kann nun der Typ (Tomcat, Glassfish, WildFly usw.) des Servers ausgewählt werden und dieser dann konfiguriert werden.

Automatische Server-Installation

NetBeans bietet uns an, einen brauchbaren Glassfish Server automatisch zu installieren. Wählen Sie zunächst Glassfish Server aus und wählen dann auf der folgenden Seite die Einstellungen „Remote Domain“ bzw. akzeptieren auch noch das License Agreement wie in Abbildung 5

ersichtlich. Der „Download Now...“ Button ist nun verfügbar und bietet uns eine Reihe von verfügbaren Glassfish-Server-Versionen an. Wählen Sie wie in Abbildung 5 zu sehen die Version „Glassfish 4.1“ aus. Diese Version ist für das Labor ausreichend!

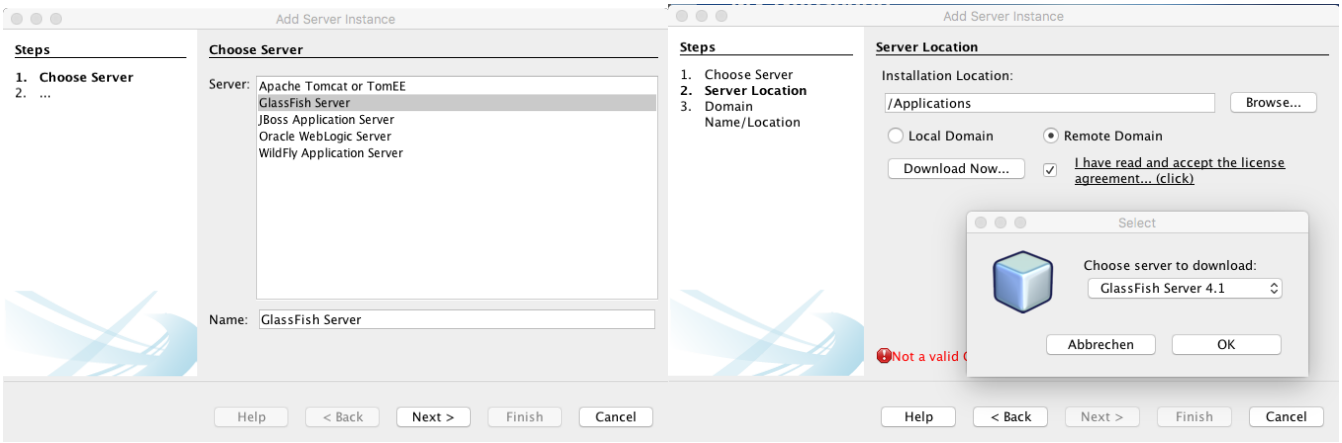

Abbildung 5: Server Instanz hinzufügen

Sobald der Server heruntergeladen und installiert wurde, kann der Server konfiguriert werden. Stellen Sie sicher, dass der Host („localhost“) erreichbar ist und dass im Feld „Domain“ ein Wert (Default: „domain1“) steht.

Alternative: Application Server manuell installieren

Im Dialog in Abbildung 5 kann der Application Server auch direkt (Installation Location) angegeben werden in dem per „Browse ...“ eine Glassfish Server Installation ausgewählt wird. Dazu muss der Server von https://javaee.github.io/glassfish/ geladen und entpackt werden.

Achtung: Bei der Auswahl des Servers ist das „Unterverzeichnis“ mit dem Namen „glassfish5/glassfish“ zu wählen!

Wird das Verzeichnis korrekt ausgewählt, kann der Server konfiguriert werden. Stellen Sie dabei sicher dass der Host („localhost“) erreichbar ist und dass im Feld „Domain“ ein Wert (Default: „domain1“) steht.

Starten des Servers, Anlegen einer DB

Sobald der Server konfiguriert ist, findet sich im Tab „Services“ ein Eintrag unter „Servers“ wie in Abbildung 6 dargestellt.

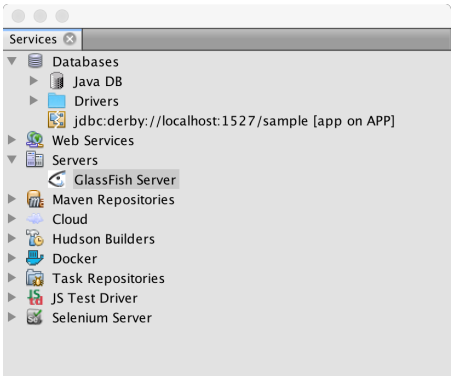

Abbildung 6: Servers - Glassfish Server

Dieser kann nun per Rechtsklick auf den Servereintrag gestartet werden. Dabei wird auch automatisch der Datenbank-Server (Apache Derby) mitgestartet. Letzterer kann auch unter der Rubrik „Databases – Java DB“ per Rechtsklick separat gestartet werden.

Hinweis: An dieser Stelle kann auch die Sample-DB neu erstellt/initialisiert werden.

Soll für die eigene Applikation NICHT die Sample-DB verwendet werden, kann eine eigene Datenbank erstellt werden. Diese eigene Datenbank muss dem Application Server jedoch erst „bekannt“ gemacht werden. Dazu muss nun der Application Server gestartet werden.

Sobald der Application Server aktiv ist kann dieser auch weiter konfiguriert werden. Dazu ist die Domain Admin Console unter http://localhost:4848 im Browser zu öffnen.

Datenbank-Verbindungen

Der Application Server verwaltet **JDBC Connection Pools**. Es wird zunächst grundsätzlich definiert mit welchen Informationen (Server, Port, DB-Name, User & Password) der Application Server eine Verbindung zu Datenbank aufbauen kann.

JDBC-Connection Pool eintragen[1]

In der Admin Console kann per „New ...“ ein neuer Connection Pool angelegt werden. Folgende Parameter müssen definiert werden:

- PoolName: Frei zu vergebender DB-Connection Pool Name
- Resource Type: **javax.sql.DataSource**
- Database Driver Vendor: **Derby**

Auf der folgenden Seite können die weiteren Verbindungsinformationen definiert werden. Achten Sie darauf, dass folgende „Properties“ (Groß-Klein beachten) vorhanden sind:

- **ServerName**: für uns „localhost“
- **PortNumber**: fix 1527

[1] Nur notwendig, wenn ihr NICHT die Sample-DB verwendet – die ist bereits konfiguriert

- **DatabaseName**: euer DB-Name
- **User**: Der Benutzer/Schema mit dem/auf das zugegriffen werden soll.
- **Password**: Passwort des Benutzers

Sobald diese Informationen gespeichert sind, kann der Connection Pool per „Ping“ getestet werden.

Sobald diese Verbindung klappt, ist der ApplicationServer für das Bereitstellen von DB-Connections verantwortlich. Es muss jedoch noch definiert werden, unter welchem Namen eine Applikation eine Verbindung zu einer Datenbank anfordern kann. Dies wird unter der Rubrik **JDBC Resources** definiert.

JDBC Resource eintragen

Eine JDBC Resource stellt das Mapping zwischen Applikationen und den Connection Pools sicher. Jede wird eindeutig durch einen JNDI-Namen identifiziert welcher von den Applikationen verwendet wird um Datenbankverbindungen anzufordern. Dazu muss in der Rubrik JDBC Resources ein neuer Eintrag hinzugefügt werden. Wesentlich hier ist der JNDI Bezeichner der die Resource „identifziert“. Hier die Namenskonvention berücksichtigen und einen Bezeichner wählen der mit dem Präfix „jdbc/“ beginnt. Aus der Liste der konfigurierten Connection Pools kann nun der zuvor erstellte Connection Pool ausgewählt werden. Die Datenbank ist somit konfiguriert und in Applikationen anhand des JNDI Namens nutzbar.

Im Falle einer Java EE Applikation wird dieser JNDI-Name in Configuration Files definiert (im EJB Modul → persistence.xml) wobei die DB-Verbindung im Element „`jta-data-source`“ mit dem JNDI Namen angefordert wird. Die Applikation selbst nutzt ihren eigenen Verbindungsnamen im Attbibut „`name`“ des Elements „`persistence-unit`“. Mit der JEE Annotation

```
@PersistenceContext(unitName="Demo-ejbPU")
private EntityManager em;
```

wird die DB-Verbindung angefordert. Die zugehörige persistence.xml ist nachfolgend aufgelistet.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
       xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
        xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
            http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
  <persistence-unit name="Demo-ejbPU" transaction-type="JTA">
    <jta-data-source>jdbc/sample</jta-data-source>
    <exclude-unlisted-classes>false</exclude-unlisted-classes>
    <properties/>
  </persistence-unit>
</persistence>
```

Hinweis: Die Datei “persistence.xml” wird automatisch von NetBeans erzeugt sobald die Entity-Classes erstellt werden (siehe Entity Classes erstellen). Die dafür benötigte Connection-Information (jdbc/sample, jdbc/__default, eure JDB-Resource) wird in die Datei persistence.xml

übertragen. Die Angabe des `unitName` kann auch entfallen, wenn wie im Beispiel nur ein Element „`persistence-unit`“ vorhanden ist. Die Anweisung ist dann vereinfacht

```
@PersistenceContext
private EntityManager em;
```

Laden der Datenbank

Sobald die Datenbankverbindung (JDBC Connection Pool) funktioniert kann die DB geladen werden. Das folgende DEMO-Script kann in NetBeans ausgeführt werden. Beim erstmaligen Aufruf werden noch 2 Fehler gemeldet da zunächst versucht wird bestehende Tabellen zu löschen. Ein zweiter Durchlauf meldet dann keine Fehler mehr.

```
drop table CUSTOMER;
drop table BANK;

create table BANK
(
   ID                   INTEGER               not null generated by default as identity,
   BIC                  VARCHAR(10)           not null,
   "NAME"               VARCHAR(100),
   constraint P_BANK_PK primary key (ID),
   constraint A_BANK_AK unique (BIC)
);

create unique index BANK_AK on BANK (
   BIC                  ASC
);

create table CUSTOMER
(
   ID                   INTEGER               not null generated by default as identity,
   BANK_ID              INTEGER               not null,
   CUSTOMER_NO          VARCHAR(20)           not null,
   "NAME"               VARCHAR(100),
   constraint P_CUSTOMER_PK primary key (ID),
   constraint P_CUSTOMER_AK unique (CUSTOMER_NO),
   constraint F_HAS_CUSTOMER foreign key (BANK_ID)
         references BANK (ID)
         on delete restrict on update restrict
);

create index HAS_CUSTOMER_FK on CUSTOMER (
   BANK_ID              ASC
);
```

Sobald das DB Schema geladen ist können per INSERT auch Demo-Daten eingetragen werden.

Erstellen der Demo Applikation

Wir erstellen eine Applikation in dem wir (nacheinander) die folgenden Bausteine realisieren:

- Applikation (Container)
- Persistenz
- Business-Logik
- Services (SOAP)
- Services (REST)

Enterprise Archive erzeugen

Per „New Project ...“ kann ein neues Projekt angelegt werden. Als „Stereotyp“ der zu erstellenden Applikation wählen wir die Kategorie „Java EE“ und „Enterprise Application“. Informationen zu Enterprise Applications finden sich im SWD-Skriptum.

Es muss ein Projekt-Name definiert werden bzw. kann auch der Ablageort des Projekt verändert werden. Auf der zweiten Seite des Wizards wird noch die Server-Runtime konfiguriert. Hier bitte sowohl ein **EJB-Modul** als auch ein **Web Application Modul** auswählen.

Wichtig: Das Enterprise Archive Projekt **enthält** die beiden Module, auch wenn sie in der Projekt-Übersicht gesondert dargestellt werden. Es wird auch immer das Enterprise Archive Projekt ausgeführt und niemals eins der beiden Module!!!

Entity Classes erstellen

Im EJB Modul werden die Entity-Classes erstellt. Entity-Klassen stellen eine Kopie zu den Tabellen in der Datenbank dar. Diese sind mit JPA-Annotationen versehen – weitere Informationen dazu – siehe SWD-Script.

Achtung: Beim Erzeugen der Entity-Klassen soll ein vernünftiger Package Name definiert werden – z.B. „at.fhs.bank.**model**“. Weiter empfiehlt es sich, auf der letzten Seite des Wizards die Option „*Use Column Names in Relationships*“ zu **deaktivieren**. Dies definiert ob die Methodennamen sich auf das Property `get/setBankId(Bank bankId)` oder auf die Klasse selbst `get/setBank(Bank bank)` beziehen. Letzteres ist intuitiver!

Zusätzlich zu den JPA-Annotationen (@Entity,@Id, @Column etc.) werden vom Wizard auch so genannte NamedQueries erzeugt. Damit werden vordefinierte Abfragen definiert, die in der Applikation einfach abgefragt werden können. Prüfen Sie in ihrer generierten Klasse Customer.java ob folgende NameQuery vorhanden ist:

```
@NamedQuery(name = "Customer.findByBank", query = "SELECT c FROM Customer c WHERE c.bankId = :bank")
```

Hinweis: Sollte diese Query nicht vorhanden sein, bitte eintragen (das Property „`c.bankId`“ kann auch nur „`c.bank`“ lauten – abhängig von der zuvor gewählten Option) – diese Query wird später benötigt!

Business Logik

Sobald die Entity-Klassen verfügbar sind können wir die Business-Logik implementieren, für unser (Mini)DB Beispiel wollen wir folgende Methoden implementieren:

- Anlegen der Bank
- Auflisten aller Banken
- Abfragen einer Bank anhand des BIC
- Anlegen eines Kunden zu einer Bank
- Abfragen aller Kunden einer Bank (BIC)

Das entsprechende UML Diagramm hierzu ist nachfolgend dargestellt.

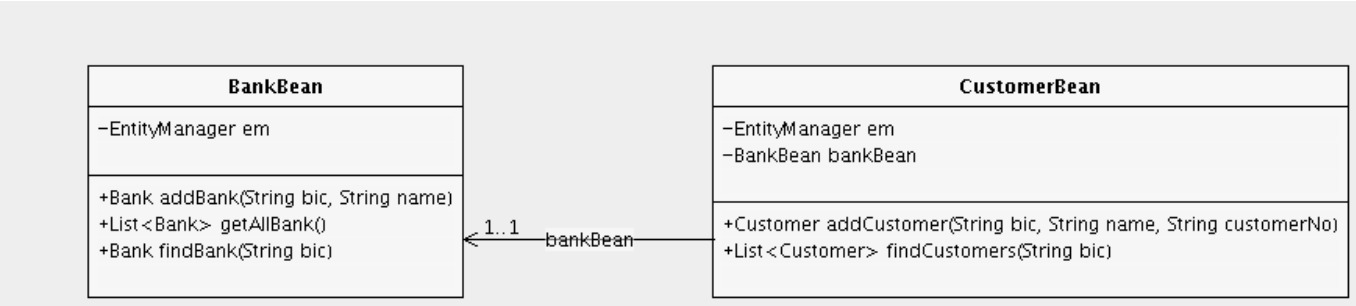

Abbildung 7: UML Diagramm Bank- und CustomerBean

Erstellen der BankBean

Anlegen eines Session-Beans wobei hier die Art des Session-Beans auf „stateless“ also zustandslos gesetzt wird. Weiterführende Informationen zu Stateless, Stateful etc. finden sich im SWD-Script.

```
@Stateless
@LocalBean
public class BankBean {
    // DB-Verbindung anfordern (siehe persistence.xml)
    @PersistenceContext
    private EntityManager em;
    /**
     * eine neue bank hinzufügen
     */
    public Bank addBank(String bic, String name) {
        // Instanz einer Bank erzeugen und initialisieren – die ID wird automatisch vergeben
        Bank b = new Bank();
        b.setBic(bic);
        b.setName(name);
        // Bank-Instanz Entity-Manager „übergeben“ – dieser sorgt für die Speicherung in der DB
        em.persist(b);
        return b;
    }
    /**
     * Alle Banken abfragen
     */
    public List<Bank> getAllBank() {
        return em.createNamedQuery("Bank.findAll", Bank.class)
                .getResultList();
    }
    /**
     * Eine Bank anhand des BIC finden, wenn kein Ergebis --> null
     */
    public Bank findBank(String bic) {
        try {
            return em.createNamedQuery("Bank.findByBic", Bank.class)
```

```
                    .setParameter("bic", bic)
                    .getSingleResult();
        } catch (NoResultException e) {
            return null;
        }
    }
}
```

Erstellen der CustomerBean

Analog zu `BankBean`, jedoch benötigen wir zum Anlegen eines Kunden auch dessen Bank. Daher wird zunächst die DB-Verbindung angefordert (analog zu `BankBean`) und auch eine `BankBean` – dazu wird vom Application Server per `@EJB` Annotation eine Instanz des `BankBeans` verlangt. Diese wird dann dazu genutzt, um anhand der BIC auch die richtige Bank zu finden.

```
@Stateless
@LocalBean
public class CustomerBean {
    @PersistenceContext
    private EntityManager em;

    @EJB
    private BankBean bankBean;
    /**
     * einen neuen Customer hinzufügen
     */
    public Customer addCustomer(String bic, String name, String customerNo) {
        // Die Logik aus dem BankBean nutzen um die angeforderte Bank zu erhalten
        // TODO: Proper Exception Handling – Bank might be NULL
        Bank bank = bankBean.findBank(bic);
        // Instanz des Customers erstellen und initialisieren
        Customer c = new Customer();
        // Customer gehört zu genau einer Bank
        c.setBankId(bank);
        c.setName(name);
        c.setCustomerNo(customerNo);
        // Customer dem EntityManager „übergeben“
        em.persist(c);
        return c;
    }
    public List<Customer> findCustomers(String bic) {
        Bank b = bankBean.findBank(bic);
        return em.createNamedQuery("Customer.findByBank", Customer.class)
                .setParameter("bank", b)
                .getResultList();
    }
}
```

Sobald diese Business-Logik erstellt ist können wir entsprechende Services publizieren, welche diese Business-Logik verwenden.

SOAP-Services erstellen

Anlegen eines „Web Services“ wobei lediglich ein Name für das Service (e.g. BankService) festzulegen ist. Da wir die Enterprise-Beans bereits erstellt haben müssen wir diese lediglich als Grundlage für das neue SOAP-Service verwenden.

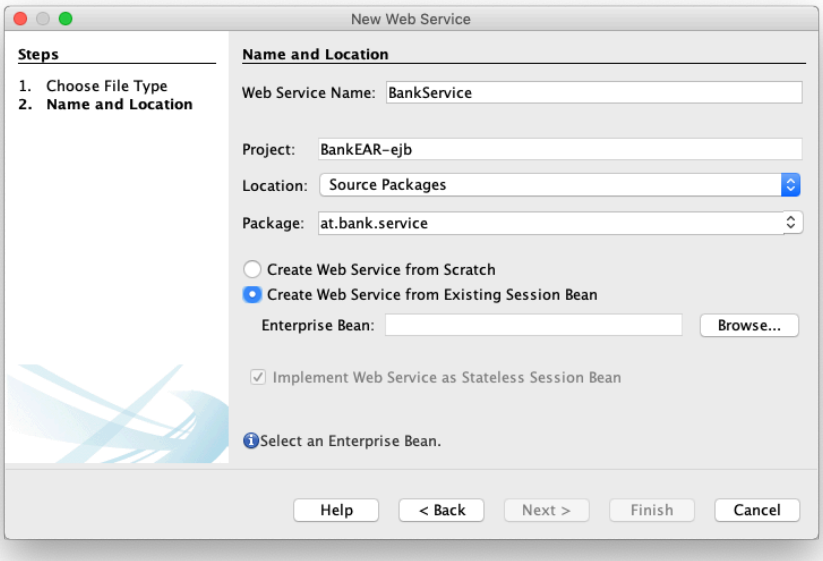

Abbildung 8: Web Service erstellen

Der Wizard „New Web Service ...“ bietet uns an, ein bestehendes SessionBean als Grundlage für ein Web-Service auszuwählen. Das generierte WebService bekommt eine @EJB Referenz auf das gewählte Session Bean und alle **public** Methoden des Session Beans werden per „Delegate“ – Pattern zugänglich gemacht.

```
// Expose the class as web service
@WebService(serviceName = "BankService")
@Stateless
public class BankService {

    // Reference to the wrapped Session Bean
    @EJB
    private BankBean ejbRef;

    @WebMethod(operationName = "addBank")
    public Bank addBank(@WebParam(name = "bic") String bic,
                        @WebParam(name = "name") String name) {
        return ejbRef.addBank(bic, name);
    }

    @WebMethod(operationName = "getAllBank")
    public List<Bank> getAllBank() {
        return ejbRef.getAllBank();
    }

    @WebMethod(operationName = "findBank")
    public Bank findBank(@WebParam(name = "bic") String bic) {
        return ejbRef.findBank(bic);
    }
}
```

Änderungen am Session Bean - z.B. neue Business-Logik-Methoden – werden nicht automatisch in das Webservice übertragen, dies muss manuell „nachgezogen“ werden oder ggf. kann das Service gelöscht und neu generiert werden.

REST-Services erstellen

Im WEB-Modul unseres Enterprise Archives können wir die RESTful Services erstellen. Zunächst erstellen wir das Service um per GET Requests auf die Bank-Daten zugreifen zu können.

```
@Named
@RequestScoped
@Path(value = "bank")
public class BankRest {

    @EJB
    private BankBean bean;

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<Bank> findAll() {
        return bean.getAllBank();
    }
    @GET
    @Produces(MediaType.APPLICATION_JSON)
    @Path(value="{bic}")
    public Bank findByBic(@PathParam(value = "bic") String bic) {
        return  bean.findBank(bic);
    }
}
```

Sobald die Klasse erstellt ist, findet sich in der Zeile mit dem Klassennamen eine kleine „Glühbirne“. NetBeans beschwert (on MouseOver) sich, dass REST noch nicht konfiguriert wurde und bietet uns (bei Mausklick) auch an dieses für uns zu erledigen. Dazu ist die Option **Configure REST using Java EE6 specification** zu wählen. Wir erstellen noch die Klasse zum Abfragen des CustomerBeans.

```
@Named
@RequestScoped
@Path(value = "customer")
public class CustomerRest {
    @EJB
    private CustomerBean bean;

    @GET
    @Path("list")
    @Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
    public List<Customer> getCustomer(@QueryParam(value = "bic") String bic) {
        return bean.findCustomers(bic);
    }

}
```

Testen der Applikation

Per Rechtsklick auf das Enterprise Archive in der Projektübersicht kann die Applikation auf den Application Server „deployed“ werden. Dabei wird ein (ZIP) komprimiertes File mit der Datei-Endung EAR erzeugt und an den Application Server übergeben.

Sobald das Deployment erfolgreich ist können wir die Applikation in der Admin-Console des Application Servers begutachten. Schließlich können wir noch die SOAP und REST Services testen.

SOAP-Services testen

- http://localhost:8080/CustomerService/CustomerService?Tester
- http://localhost:8080/CustomerService/BankService?Tester

REST Services testen

- http://localhost:8080/<war-modul>/webresources/bank
- http://localhost:8080/<war-modul>/webresources/bank/<bic>
- http://localhost:8080/<war-modul>/webresources/customer/list?bic=<bic>