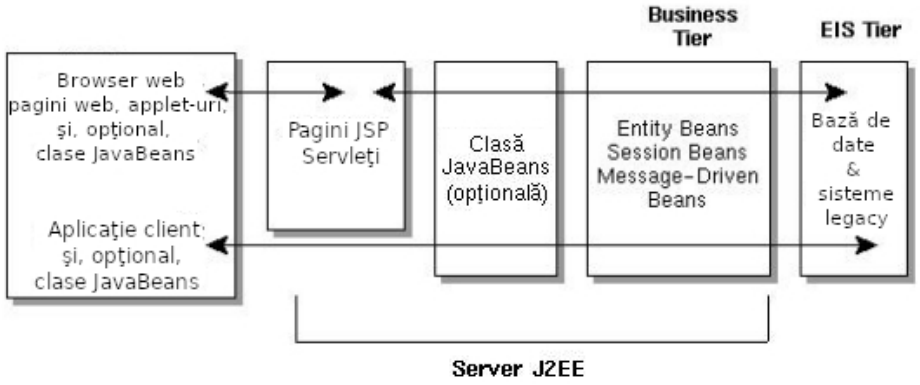
Laboratorul 2
Dezvoltarea componentelor de *business* într-o aplicaţie pentru întreprindere (JEE)

Introducere şi arhitectura JEE



În laboratorul 1 s-a pus accent pe o arhitectură simplificată JEE, ce conţine componente ce fac parte din ***web tier***, respectiv componente **client** (browser-ul web sau utilitar capabil de a trimite cereri HTTP - `curl`).

În acest laborator, veţi lucra la nivel de ***business tier***, punând astfel accent pe tipurile de componente *business* din platforma JEE:

- ***session beans*** - de 3 tipuri:
  - ***stateless session beans*** → nu menţin o legătură **client** ↔ ***bean***
  - ***stateful session beans*** → menţin legătura cu clientul apelant (fiecare client cu *stateful session bean*-ul lui)
  - ***singleton session beans*** → o singură instanţă disponibilă la nivel de server *enterprise*
- ***entity beans*** → încapsulează funcţionalitate, respectiv maparea datelor dintr-o bază de date sub formă de obiecte (învechite începând de la EJB 3.0, înlocuite de entităţile de tip ***JPA entity*** din Java Persistence API)
- ***message-driven beans*** → permit procesarea mesajelor în mod asincron

Aplicaţie JEE completă

În laboratorul 1, aţi creat un proiect JEE minimal, de tip **Web Application**, cu împachetare **WAR** (**W**eb **AR**chive). În acest laborator, deoarece urmează să utilizaţi componentele de *business tier*, veţi învăţa cum se creează un proiect JEE complet, de tip **Enterprise Application**, cu împachetare **EAR** (**E**nterprise **AR**chive) din IntelliJ IDEA Community şi cum se adaugă dependenţele necesare.

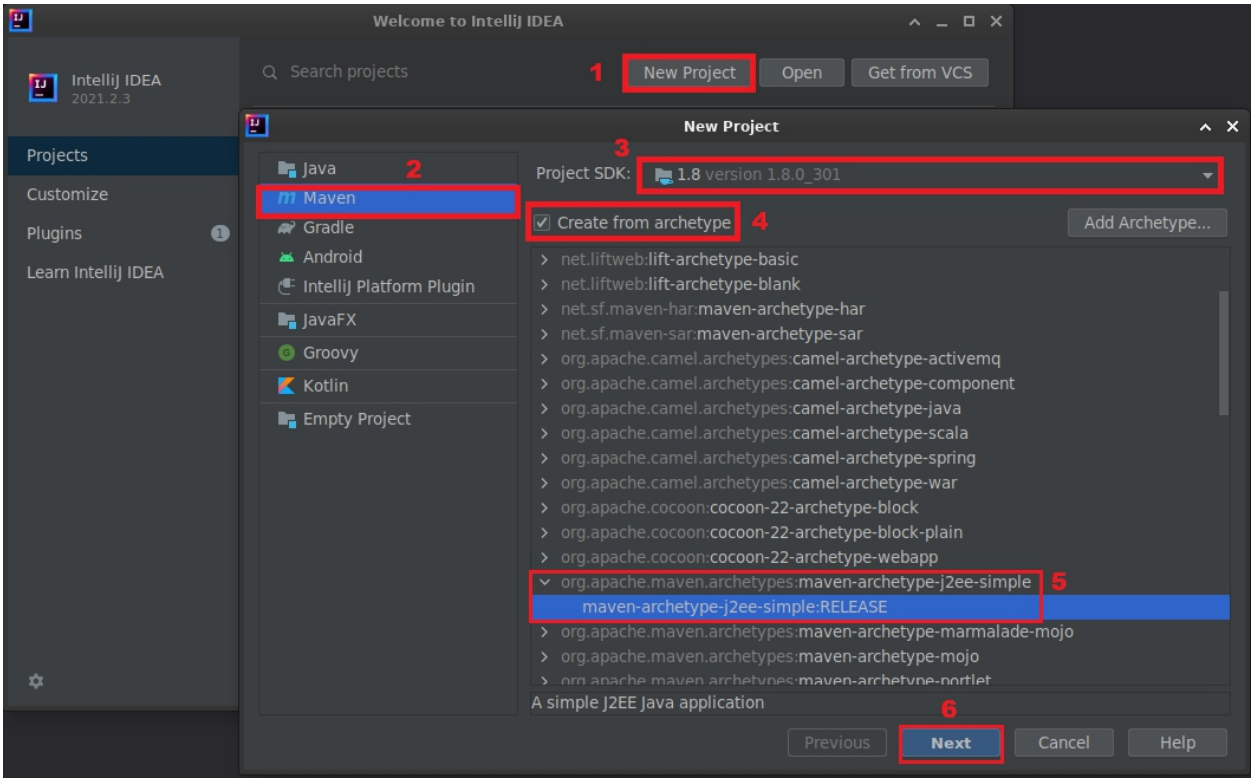
După terminarea orei de laborator, deschideţi consola Glassfish: http://localhost:4848, daţi click pe `Applications` în partea stângă, bifaţi tot în afară de `cargocpc` şi apăsaţi pe butonul `Undeploy`.

1.1. Creare şi configurare proiect JEE complet folosind IntelliJ IDEA Community

1.1.1. Creare proiect IntelliJ

Deschideţi IntelliJ IDEA Community, iar în meniul din partea dreaptă alegeţi „**Create new project**”.

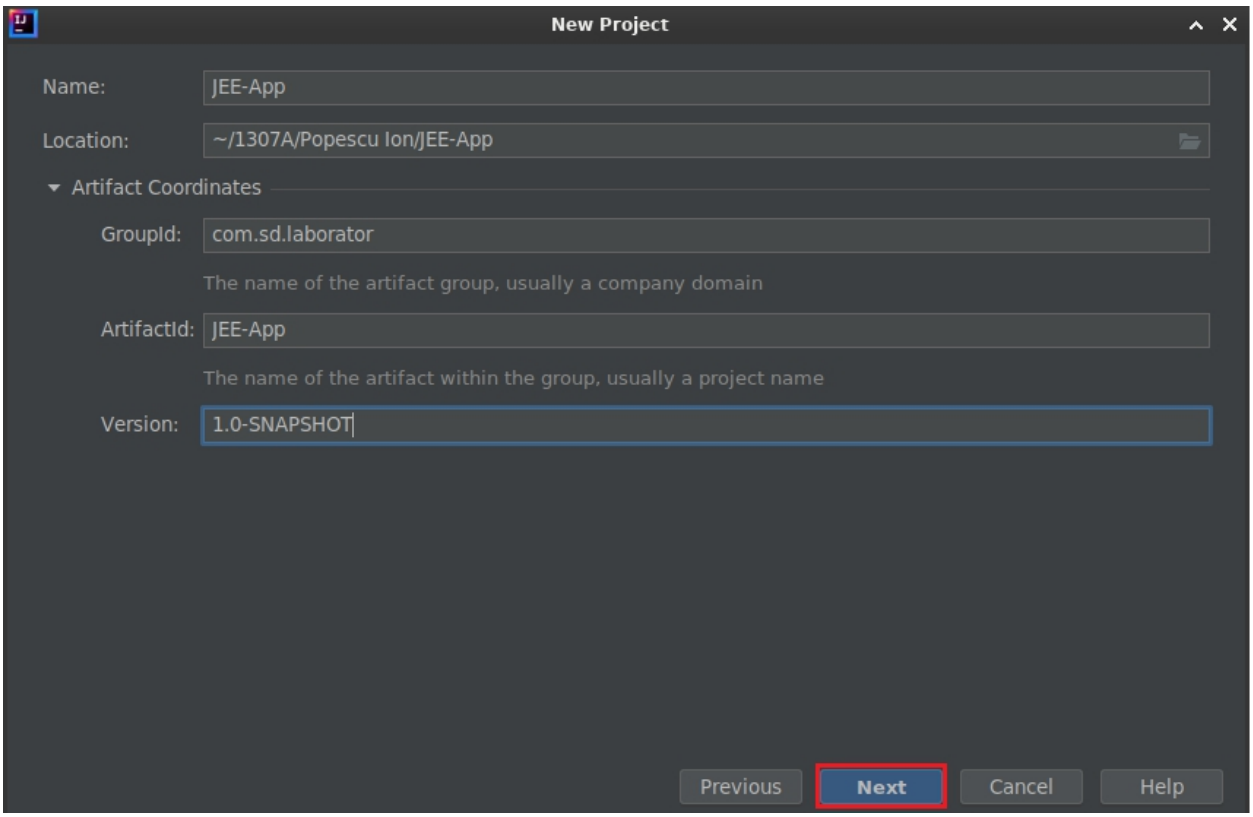
În fereastra de selecţie a tipului de proiect, alegeţi „**Maven**” în partea stângă, apoi selectaţi versiunea de **Java SDK 1.8**. Bifaţi „**Create from archetype**”, iar din lista de arhetipuri disponibile, expandaţi **`org.apache.maven.archetypes:maven-archetype-j2ee-simple`** şi selectaţi **`maven-archetype-j2ee-simple:RELEASE`**. Click pe „**Next**”.



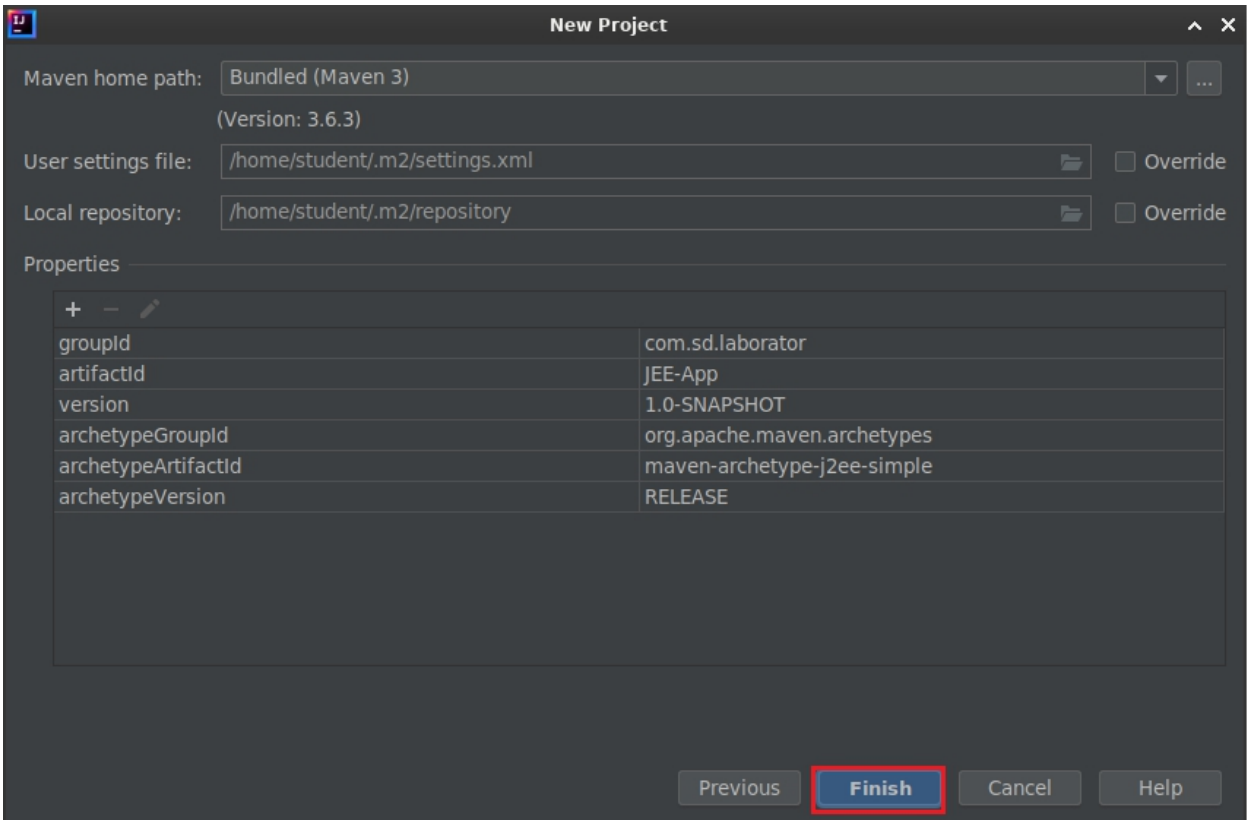
În continuare, se aleg numele şi locaţia proiectului pe disc, precum şi detaliile artefactului EAR rezultat. Secţiunea „**Artifact Coordinates**” poate fi lăsată cu valorile implicite, sau puteţi completa, dacă doriţi, **`GroupId`**-ul cu o valoare personalizată, cum ar fi **`com.sd.laborator`**.

În acest exemplu, proiectul se va numi „**JEE-App**”, iar locaţia va fi **`~/1307A/Popescu Ion/JEE-App`**.

Click pe „**Next**” după completarea datelor menţionate.



În următoarea fereastră, se lasă totul neschimbat şi se apasă „**Finish**”.



Aşteptaţi ca Maven să termine de generat structura proiectului şi să aducă primele dependenţe definite în configuraţia arhetipului.

1.1.2. Configurare proiect Maven

Proiectul Maven generat de arhetipul pentru JEE este organizat în module, fiecare modul conţinând anumite tipuri de componente, după cum urmează:

- **`ear`** → generează artefactul EAR ce încapsulează toate celelalte module ale proiectului sub formă de arhivă **E**nterprise **AR**chive.
- **`ejbs`** → conţine codul de *business* al claselor *Enterprise Java Beans*. Aici veţi pune clasele EJB, organizate eventual în pachete.
- **`primary-source`** → conţine clase adiţionale utilizate în proiect (clase care nu reprezintă neapărat componente specifice JEE)
- **`projects`** → conţine eventuale subproiecte ale proiectului JEE de bază. Are deja adăugat un schelet de subproiect denumit **`logging`**.
- **`servlets`** → conţine un submodul numit **`servlet`**, care reprezintă punctul de intrare al componentei web (nivelul *web tier*). Aici puteţi adăuga clase servlet, pagini JSP, etc.

Calea implicită de acces a folder-ului rădăcină a componentei web este `/servlet`. Aşadar, pentru a vizualiza pagina index.jsp creată automat, după încărcarea aplicaţiei, se accesează URL-ul: http://localhost:8080/servlet/.

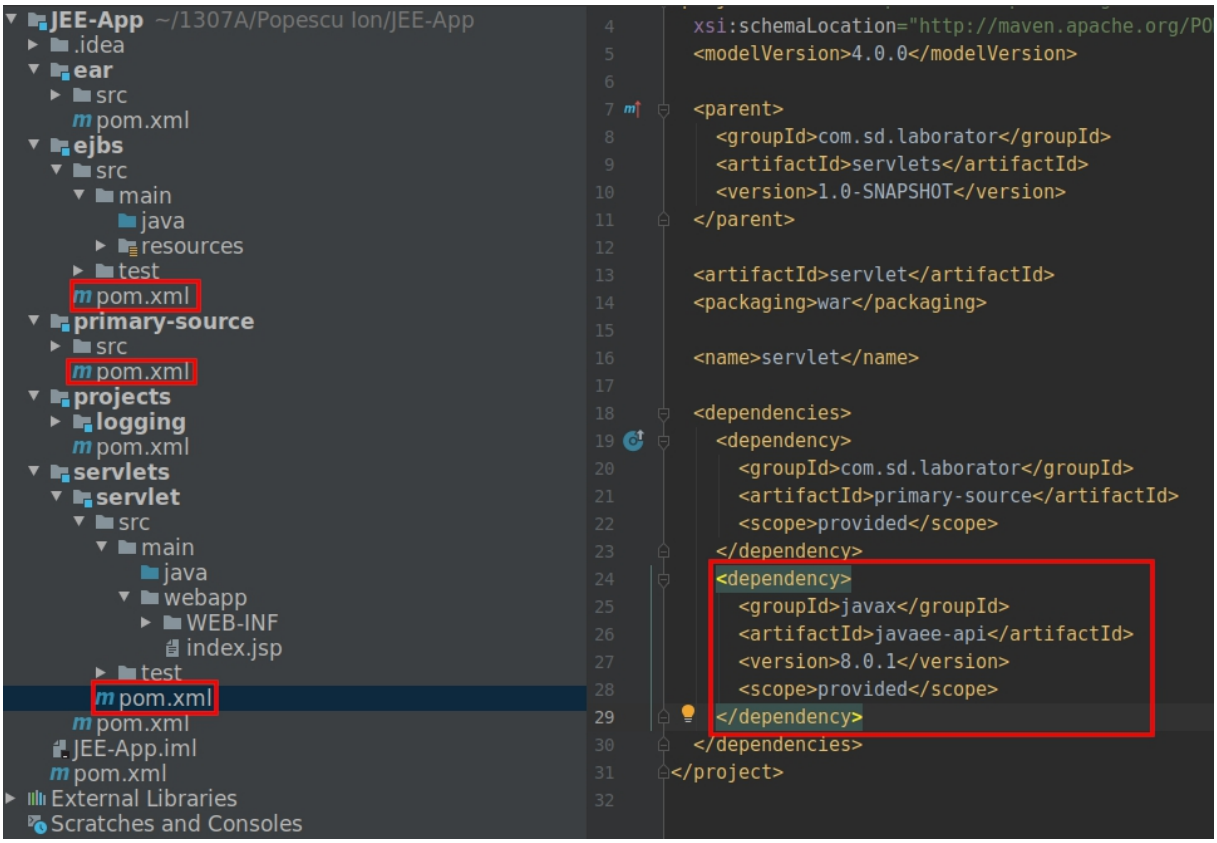
Structura de proiect arată ca în figură:

Observaţi că fiecare modul are fişierul său de configurare **`pom.xml`** (**Project Object**

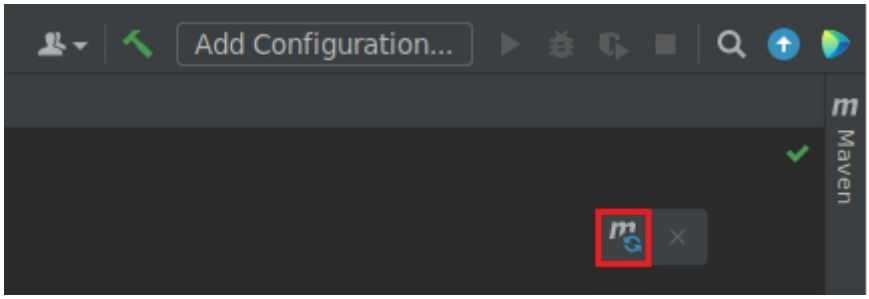
Model), deoarece se pot declara dependenţe între modulele componente, respectiv, pentru fiecare modul în parte se pot face configurări personalizate, se pot declara dependenţe şi folosi *plugin*-uri personalizate etc.

Adăugaţi dependenţa **JavaEE API** (https://mvnrepository.com/artifact/javax/javaee-api/8.0) **în fiecare fişier `pom.xml` al modulelor `ejbs`, `primary-source` şi `servlet` (atenţie, nu `servlets`!)**. Aşadar, ca subordonat al *tag*-ului **`<dependencies>`** din fişierele **`pom.xml`**, adăugaţi următorul element:

```
<dependency>
 <groupId>javax</groupId>
 <artifactId>javaee-api</artifactId>
 <version>8.0.1</version>
 <scope>provided</scope>
</dependency>
```



După modificările făcute în fişierele **`pom.xml`**, nu uitaţi să sincronizaţi modificările, apăsând iconița din dreapta-sus a IDE-ului când IntelliJ vă cere acest lucru:



1.1.3. Completare descriptori XML lipsă

Arhetipul din care proiectul Maven a fost creat nu adaugă conţinut în descriptorii XML ai componentelor web, respectiv EJB. Aceştia trebuie adăugaţi manual, astfel:

Dechideţi **`servlets/servlet/src/main/webapp/WEB-INF/web.xml`** şi adăugaţi următorul conţinut:

```
<!DOCTYPE web-app PUBLIC
 "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
 "http://java.sun.com/dtd/web-app_2_3.dtd" >

<web-app>
  <display-name>Web Component App</display-name>
</web-app>
```

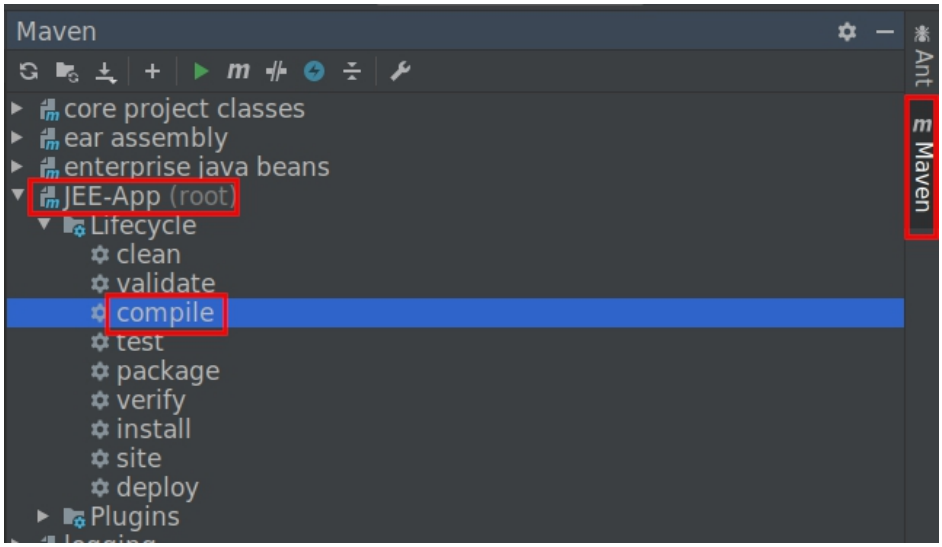
Apoi deschideţi **`ejbs/src/main/resources/META-INF/ejb-jar.xml`** şi adăugaţi următorul conţinut:

```
<ejb-jar xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
       http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd"
         version="3.0">
</ejb-jar>
```

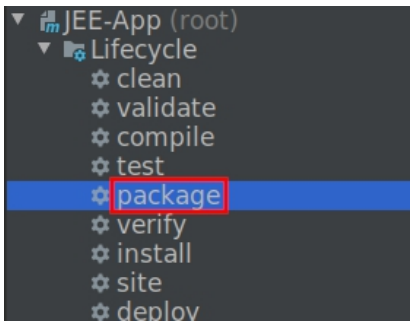
1.2. Compilarea şi împachetarea aplicaţiei JEE

Operaţiile care pot fi făcute asupra proiectului sunt disponibile sub formă de *Maven lifecycles* în partea dreaptă a ferestrei IntelliJ, în conţinutul panoului **Maven**.

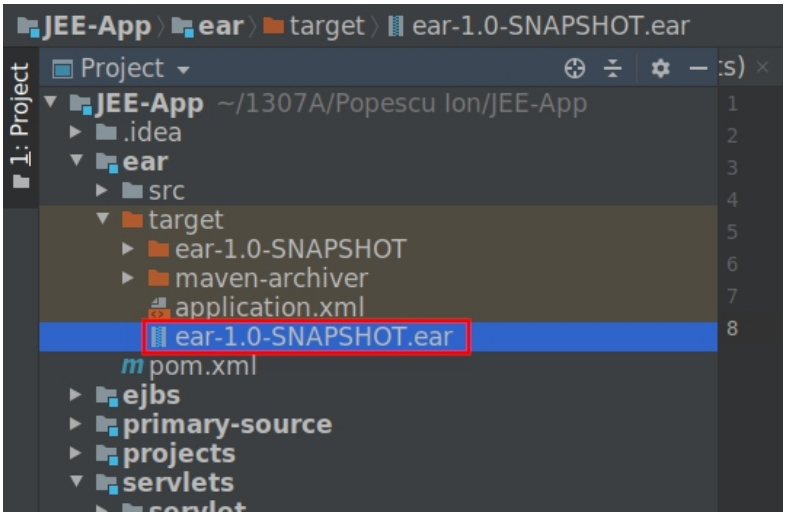
Pentru a compila proiectul, se expandează panoul Maven şi se utilizează *lifecycle*-ul **`compile`** al modulului Maven care cuprinde întregul proiect, adică cel denumit exact ca şi proiectul IntelliJ: **`<nume_proiect> (root)`**.



Tot din acelaşi modul Maven se poate împacheta structura aplicaţiei rezultate în urma compilării (conţinutul folder-ului **`target`**) folosind *lifecycle*-ul **`package`**.

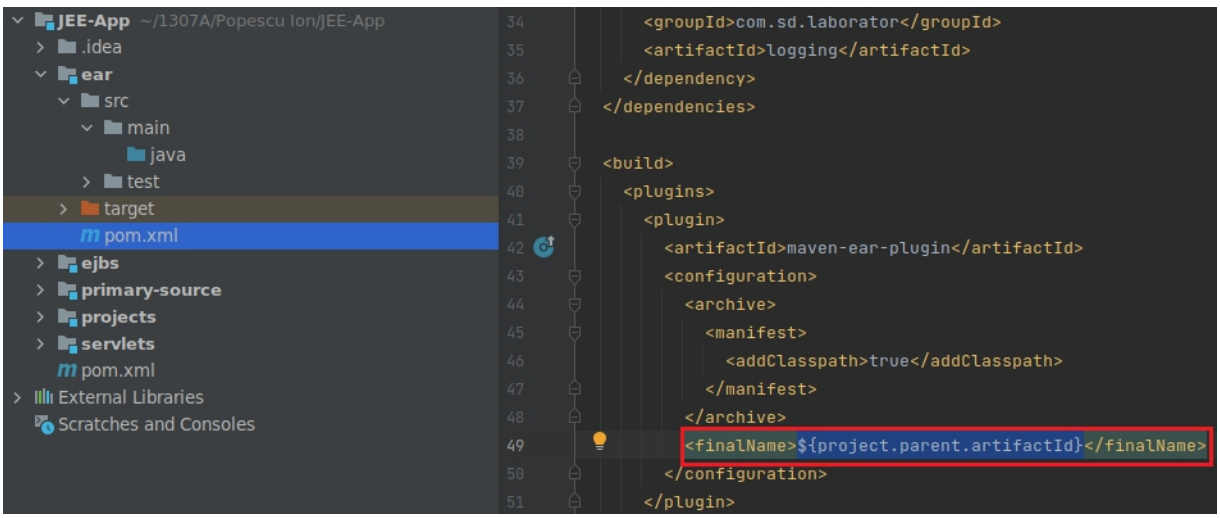


Acest pas va genera un fişier **EAR** (**E**nterprise **AR**chive) ca subordonat al folder-ului **`target`** din modulul **`ear`**, vizibil în structura de proiect din partea stângă a ferestrei IntelliJ. Artefactul va fi denumit, în mod implicit, sub forma **`ear-<versiune>.ear`**.



Numele artefactului rezultat este stabilit în mod implicit conform regulii: **`<nume_modul_ear>-<versiune>.ear`**. Pentru a vă uşura munca atunci când veţi încărca aplicaţia pe server, configuraţi modulul ear astfel încât să exporte artefactul EAR sub un nume mai „prietenos”, şi anume cel al proiectului părinte. Deschideţi **`ear/pom.xml`** şi adăugaţi următoarea configurare ca şi subordonat al *tag*-ului **`<configuration>`** pentru *plugin*-ul **`maven-ear-plugin`**:

```
<finalName>${project.parent.artifactId}</finalName>
```



După ce împachetaţi din nou aplicaţia (cu *lifecycle*-ul **package**), veţi observa că rezultă un artefact EAR sub numele de **JEE-App.ear** (mult mai sugestiv ca cel vechi, implicit). Cel vechi poate fi şters.

1.3. Curăţarea proiectului

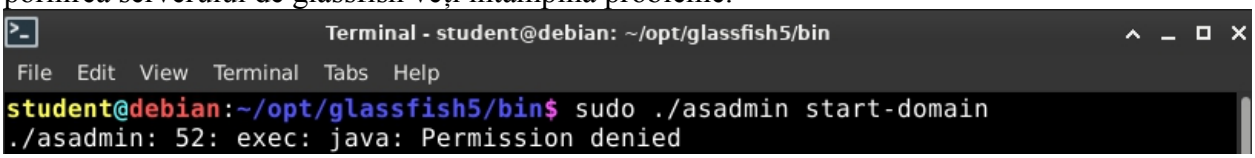
Tot din panoul Maven se poate curăţa folder-ul cu fişiere compilate prin utilizarea lifecycle-ului **clean**. Acesta va şterge tot conţinutul folder-elor **target** al modulelor componente. Dacă se doreşte curăţarea doar a anumitor module, se execută *lifecycle*-ul **clean** din secţiunea corespunzătoare acestora din panoul Maven.

1.4. Pornirea server-ului GlassFish

Asigurați-vă că variabila PATH deține locația la JDK1.8 prin rularea comenzii:

```
java -version
```

Dacă rezultă răspunsul: "command not found", setați variabila respectivă (vezi laborator 1 pentru instalarea JDK1.8 și actualizarea variabilei PATH și JAVA_HOME). În caz contrar, la pornirea serverului de glassfish veți întâmpina probleme.

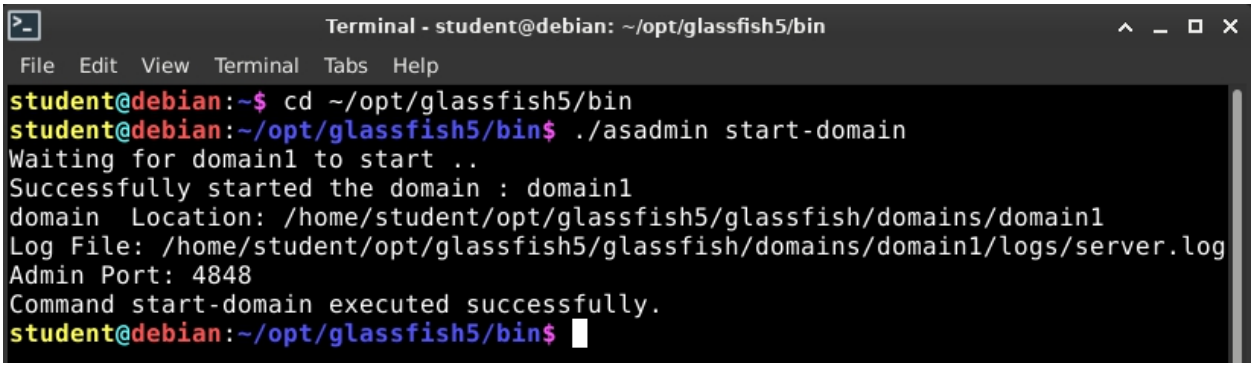


După aceasta, dintr-o sesiune de terminal, executaţi următoarea comandă:

```
<LOCAŢIE_SERVER_GLASSFISH>/bin/asadmin start-domain
```

De exemplu, în acest caz, server-ul fiind localizat în **/home/student/opt/glassfish5**, comanda este:

```
~/opt/glassfish5/bin/asadmin start-domain
```



1.5. Oprirea server-ului GlassFish

Într-o sesiune de terminal, se execută comanda:

```
<LOCAŢIE_SERVER_GLASSFISH>/bin/asadmin stop-domain
```

În acest caz, ar fi:

```
~/opt/glassfish5/bin/asadmin stop-domain
```

1.6. Încărcarea proiectului (deploy) pe server-ul GlassFish

Deoarece s-a utilizat un arhetip de proiect JEE modular, încărcarea artefactului EAR pe server-ul de aplicaţii *enterprise* se poate face mai facil din linia de comandă, decât cu *plugin*-ul Cargo, utilizat în laboratorul 1.

Atenţie: nu puteţi încărca o aplicaţie în format EAR pe server-ul GlassFish dacă nu există cel puţin un tip de *Enterprise Bean* creat şi descris în `ejb-jar.xml`. Deci, nu veţi putea utiliza efectiv comenzile descrise în cele ce urmează decât după ce adăugaţi *bean*-uri.

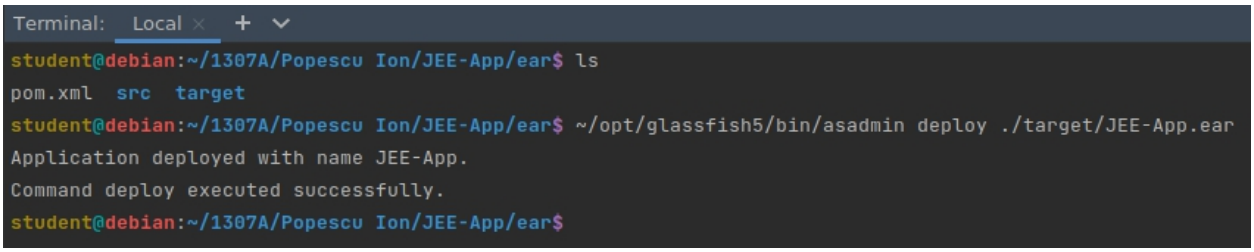
După ce aţi compilat şi împachetat aplicaţia sub formă de artefact EAR, click dreapta pe folder-ul **`ear`** → **Open in Terminal**. Se va deschide o sesiune de terminal în partea de jos a ferestrei IntelliJ.

Încărcarea proiectului (prima dată) se face cu următoarea comandă:

```
<LOCAŢIE_SERVER_GLASSFISH>/bin/asadmin deploy
/CALE/CĂTRE/ARTEFACT/<NUME_ARTEFACT>.ear
```

De exemplu:

```
~/opt/glassfish5/bin/asadmin deploy ./target/JEE-App.ear
```



1.7. Ştergerea proiectului (undeploy) de pe server-ul GlassFish

Dacă se doreşte ca aplicaţia enterprise să fie ştearsă de pe server, se poate folosi comanda:

```
<LOCAŢIE_SERVER_GLASSFISH>/bin/asadmin undeploy <NUME_APLICAŢIE>
```

Exemplu:

```
~/opt/glassfish5/bin/asadmin undeploy JEE-App
```

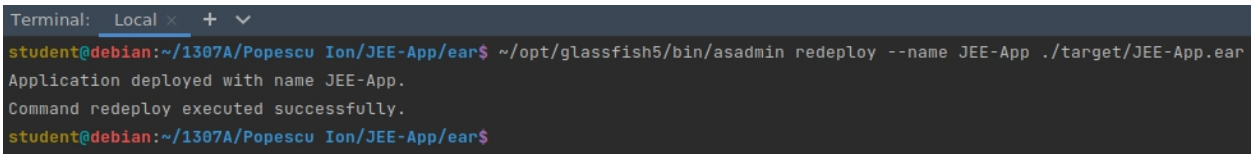
1.8. Reîncărcarea proiectului (redeploy) pe server-ul GlassFish

După ce încărcaţi pentru prima dată o aplicaţie JEE pe server, dacă actualizaţi fişierele sursă şi doriţi să vedeţi modificările, trebuie să reîncărcaţi artefactul nou rezultat în urma împachetării. Deoarece artefactul are acelaşi nume după reîmpachetare (dacă nu îl modificaţi), server-ul nu vă permite să suprascrieţi aplicaţia veche, decât folosind o comandă de **reîncărcare** (**`redeploy`**):

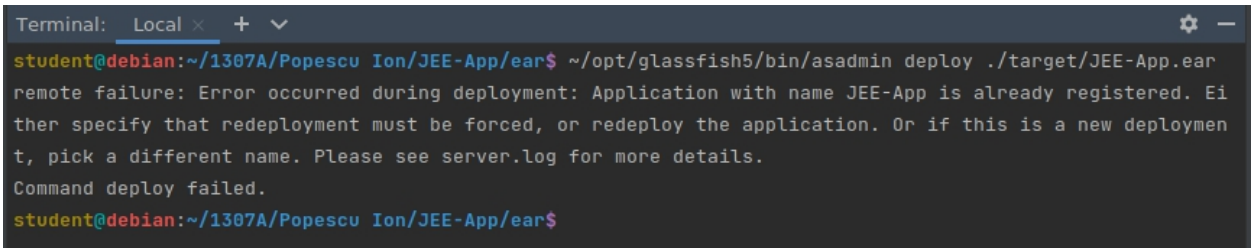
```
<LOCAŢIE_SERVER_GLASSFISH>/bin/asadmin redeploy <NUME_APLICAŢIE>
/CALE/CĂTRE/ARTEFACT/<NUME_ARTEFACT>.ear
```

Exemplu:

```
~/opt/glassfish5/bin/asadmin redeploy --name JEE-App ./target/JEE-
App.ear
```



Dacă încercaţi să folosiţi comanda **deploy** pentru un artefact deja existent (cu acelaşi nume), veţi primi o eroare de tipul:



Stateless Session Beans

Stateless Session Bean-ul este un *enterprise bean* care, de obicei, efectuează operaţii independente de clientul apelant (nu creează nicio legătură bean ↔ client, nu păstrează starea conversaţiei dintre cele 2 părţi participante).

Pentru a exemplifica acest tip de *enterprise bean*, veţi expune clientului o interfaţă simplă prin care poate prelua data curentă de la server şi prin care poate aduna 2 numere întregi. Această interfaţă trebuie cunoscută atât de server-ul de aplicaţii *enterprise* (ca să ştie ce metode trebuie să expună în momentul încărcării aplicaţiei), cât şi de clientul apelant (pentru ca el să ştie ce metode sunt disponibile pentru apel).

Folosiţi în continuare proiectul JEE creat în paşii anteriori.

1.9. Codul de business

Adăugaţi un pachet nou în folder-ul **ejbs/src/main/java**, denumit **interfaces**. În pachetul respectiv, creaţi o interfaţă Java numită **StatelessSessionBeanRemote**, cu următorul conţinut:

```
package interfaces;

public interface StatelessSessionBeanRemote {
    String getCurrentTime();
    Integer addNumbers(Integer a, Integer b);
}
```

Acum, veţi crea implementarea conform interfeţei, corpul efectiv al *stateless session bean*-ului. În folder-ul **ejbs/src/main/java**, creaţi un pachet denumit **ejb** (**E**nterprise **J**ava **B**eans). În pachetul respectiv, creaţi o clasă Java denumită **StatelessSessionBeanImpl**, cu următorul conţinut:

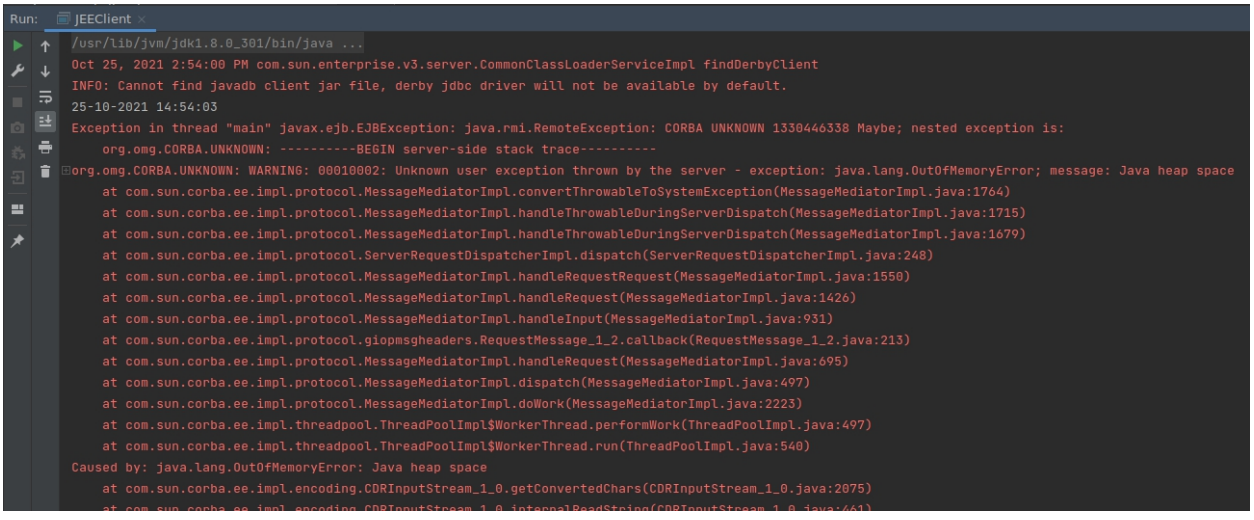
```
package ejb;

import interfaces.StatelessSessionBeanRemote;

import java.io.Serializable;
import java.text.SimpleDateFormat;
import java.util.Date;

public class StatelessSessionBeanImpl implements
StatelessSessionBeanRemote, Serializable {
    public StatelessSessionBeanImpl() {
```

```
        System.out.println("[Glassfish] S-a instanţiat un stateless
session bean: " +
                StatelessSessionBeanImpl.class.getName());
    }

    public String getCurrentTime() {
        System.out.println("[Glassfish] S-a apelat metoda
getCurrentTime()");
        Date date = new Date();
        SimpleDateFormat formatter = new SimpleDateFormat("dd-MM-yyyy
HH:mm:ss");
        return formatter.format(date);
    }

    public Integer addNumbers(Integer a, Integer b) {
        System.out.println("[Glassfish] S-a apelat metoda addNumbers("
+ a + ", " + b + ")");
        return a + b;
    }
}
```

Clasa ce reprezintă acest bean trebuie să implementeze interfaţa **`Serializable`**, deoarece codul va fi apelat prin RMI (**Remote Method Execution**) de la o aplicaţie client complet decuplată de server, şi atunci entităţile conţinute de acea clasă (inclusiv clasa însăşi) trebuie să fie serializabile.

Atenţie: nu folosiţi tipuri primitive de date în *enterprise beans* care vor fi apelate prin RMI (*bean*-uri nelocale)! Tipurile primitive de date (`int`, `char`, `double` etc.) nu sunt serializabile. Folosiţi, în schimb, clasele lor corespondente Java: `Integer`, `Char`, `Double` etc. De aceea, pentru metoda `addNumbers`, s-au folosit parametri de tip `Integer`, şi nu `int`, atât în interfaţă, cât şi în implementare.

Dacă *bean*-urile *enterprise* conţin tipuri primitive de date, clientul apelant va primi o eroare de acest tip:



În continuare, folosind descriptori XML, trebuie să îi specificăm server-ului de aplicaţii *enterprise* ce reprezintă clasa creată mai sus (un *bean stateless*), cum se expune aceasta clientului, unde poate găsi interfaţa pe care o implementează, respectiv unde găseşte clasa cu implementarea. Aşadar, în folder-ul **`ejbs/src/main/resources/META-INF`**, adăugaţi

următorul conţinut în fişierul **ejb-jar.xml**:

```
<ejb-jar xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
       http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd"
         version="3.0">
    <enterprise-beans>
        <session>
            <ejb-name>StatelessSessionBeanExample</ejb-name>
            <mapped-name>ssb-example</mapped-name>
            <business-
local>interfaces.StatelessSessionBeanRemote</business-local>
            <business-
remote>interfaces.StatelessSessionBeanRemote</business-remote>
            <ejb-class>ejb.StatelessSessionBeanImpl</ejb-class>
            <session-type>Stateless</session-type>
        </session>
    </enterprise-beans>
</ejb-jar>
```

În acest XML, se declară faptul că aplicaţia conţine un *session bean* (tag-ul **<session>**), cu următoarele proprietăţi:

- un nume descriptiv (nu are legătură cu numele clasei!): *tag*-ul **<ejb-name>**
- identificatorul sub care *bean*-ul este mapat în JNDI (*Java Naming and Directory Interface*): tag-ul **<mapped-name>**
- interfeţele care sunt implementate de clasa ce reprezintă *bean*-ul (locală - **business-local**, respectiv de la distanţă - **business-remote**), date sub forma **<pachet>.<nume_clasă>**.
- numele clasei care conţine implementarea *bean*-ului, dat sub forma **<pachet>.<nume_clasă>**: tag-ul **<ejb-class>**
- tipul de *bean*: tag-ul **<session-type>**

Faceţi următoarele operaţiuni asupra proiectului: **clean** → **compile** → **package** → **deploy** **de la consolă** (nu uitaţi să porniţi server-ul GlassFish, dacă nu este deja pornit).

Dacă totul a decurs bine, veţi regăsi în consola de administrare GlassFish *bean*-ul creat printre celelalte componente deja existente pe server:

Modules and Components (5)

Module Name	Engines	Component Name	Type	Action
com.sd.laborator-ejbs-1.0-SNAPSHOT.jar	[ejb]	----------	----------	
com.sd.laborator-ejbs-1.0-SNAPSHOT.jar		StatelessSessionBeanExample	StatelessSessionBean	
com.sd.laborator-servlet-1.0-SNAPSHOT.war	[web]	----------	----------	
com.sd.laborator-servlet-1.0-SNAPSHOT.war		default	Servlet	
com.sd.laborator-servlet-1.0-SNAPSHOT.war		jsp	Servlet	

1.10. Extragerea numelui JNDI al bean-ului

Server-ul de aplicaţii enterprise expune bean-ul respectiv printr-un nume JNDI mapat conform a ceea ce s-a specificat în descriptorul XML (conţinutul tag-ului **<mapped-name>**).

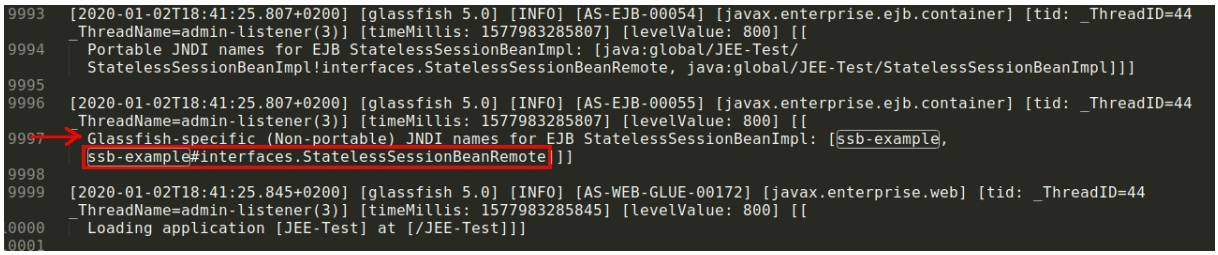
Însă, un client care caută şi utilizează *bean*-ul de pe server are nevoie de numele complet JNDI stabilit de GlassFish, conform specificaţiilor proprii. Acest nume poate fi găsit în log-ul server-ului, disponibil la locaţia următoare:

```
<LOCAŢIE_SERVER_GLASSFISH>/glassfish/domains/domain1/logs/server.log
```

În acest caz, log-ul se află în locaţia:

```
/home/student/opt/glassfish5/glassfish/domains/domain1/logs/server.log
```

Deschideţi log-ul cu un editor de text şi căutaţi numele mapat *bean*-ului pe care l-aţi creat (în acest caz, căutaţi „**ssb-example**”).



Notaţi numele complet JNDI al bean-ului atribuit de server (**de tip Glassfish-specific, nu cel portabil!**), veţi avea nevoie de el în aplicaţia client. În acest caz:

```
ssb-example#interfaces.StatelessSessionBeanRemote
```

La modul general, numele JNDI specific GlassFish al unui bean enterprise atribuit de de server este:

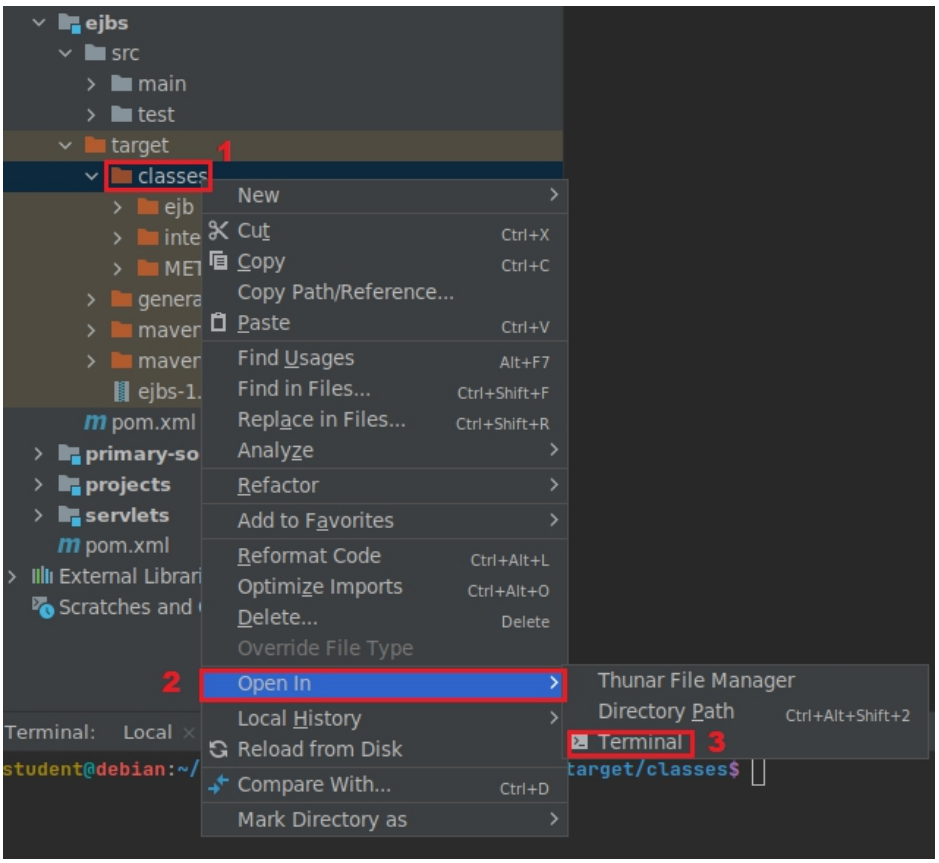
```
<NUME_MAPAT_ÎN_DESCRIPTOR>#<NUME_PACHET>.<NUME_INTERFAŢĂ>
```

1.11. Aplicaţie client

Clientul va fi, în acest caz, complet decuplat de server, deci aplicaţia client care va utiliza bean-ul creat anterior va fi executată sub propriul JVM. Aşadar, clientul trebuie să aibă acces (cumva) la interfaţa *bean*-ului, pentru a putea apela metodele din acesta, de la distanţă.

O soluţie în acest sens ar fi să împachetaţi sub formă de librărie JAR (**J**ava **AR**chive) codul obiect rezultat după compilarea fişierului sursă ce conţine interfaţa *bean*-ului. Acest fişier nu conţine implementarea, deci are sens să fie distribuit şi clientului spre utilizare (de aici decuplarea).

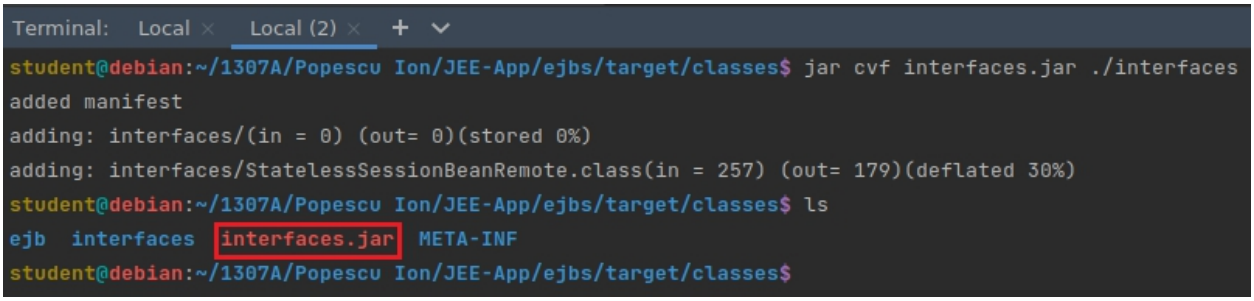
Deschideţi un terminal în folder-ul **ejbs/target/classes**. Din IntelliJ, se poate face cu click dreapta pe acest folder în structura de proiect din partea stângă → **Open in Terminal**. Se va deschide un terminal în partea de jos a ferestrei IntelliJ.



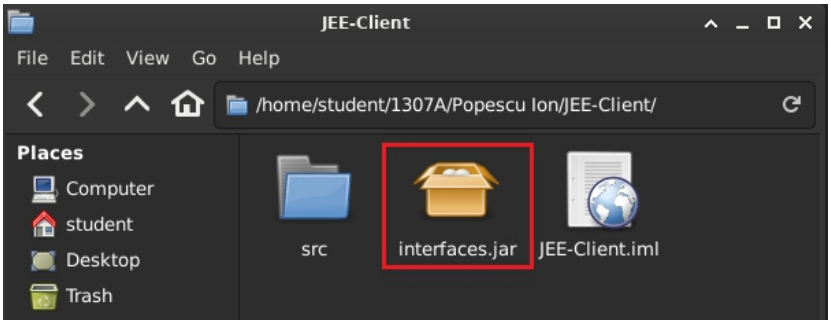
Executaţi următoarea comandă:

```
jar cvf interfaces.jar ./interfaces
```

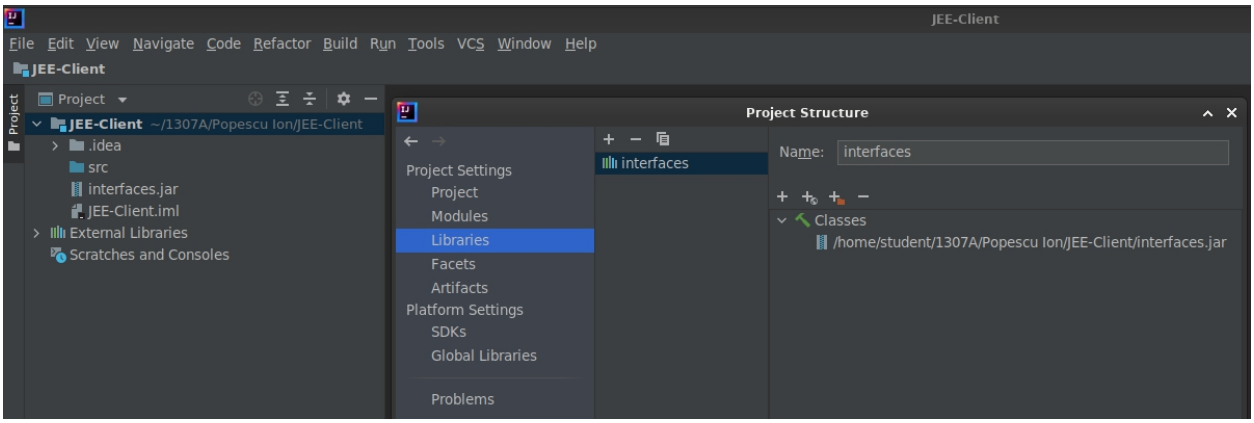
Comanda va împacheta conţinutul folder-ului **`interfaces`** într-un fişier JAR.



Creaţi un proiect nou IntelliJ de tip Java (simplu, fără manager de proiect Maven, etc.). Denumiţi-l, spre exemplu, **`JEE-Client`**. Copiaţi fişierul **`interfaces.jar`** în folder-ul rădăcină al proiectului.



Acum, trebuie să adăugaţi acest fişier ca librărie de care proiectul **`JEE-Client`** depinde (ca aplicaţia client să poată utiliza interfaţa **`StatelessSessionBeanRemote`**). Accesaţi: **File → Project Structure... → Libraries** → click pe pictograma în formă de plus (**New Project Library**) → **Java** → căutaţi şi selectaţi fişierul **`interfaces.jar`** copiat în pasul anterior → în fereastra „**Choose modules**” apăsaţi doar **OK** → apăsaţi **OK** în fereastra cu setările proiectului.



Creaţi o clasă Java în folder-ul **`src`** cu surse, denumită **`JEEClient`**, având următorul conţinut:

```
import interfaces.StatelessSessionBeanRemote;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class JEEClient {
    public static void main(String[] args) throws NamingException {
        Context ctx = new InitialContext();
        StatelessSessionBeanRemote ssb = (StatelessSessionBeanRemote)
ctx.lookup("ssb-example#interfaces.StatelessSessionBeanRemote");
        System.out.println(ssb.getCurrentTime());
        System.out.println("1 + 3 = " + ssb.addNumbers(1, 3));
    }
}
```

În codul de mai sus, se importă interfaţa *bean*-ului de tip *stateless session*. Apoi, se iniţializează un context iniţial pentru operaţia de căutare în JNDI ce urmează (este necesară această iniţializare, deoarece toate operaţiile de căutare în JNDI sunt relative la un context).

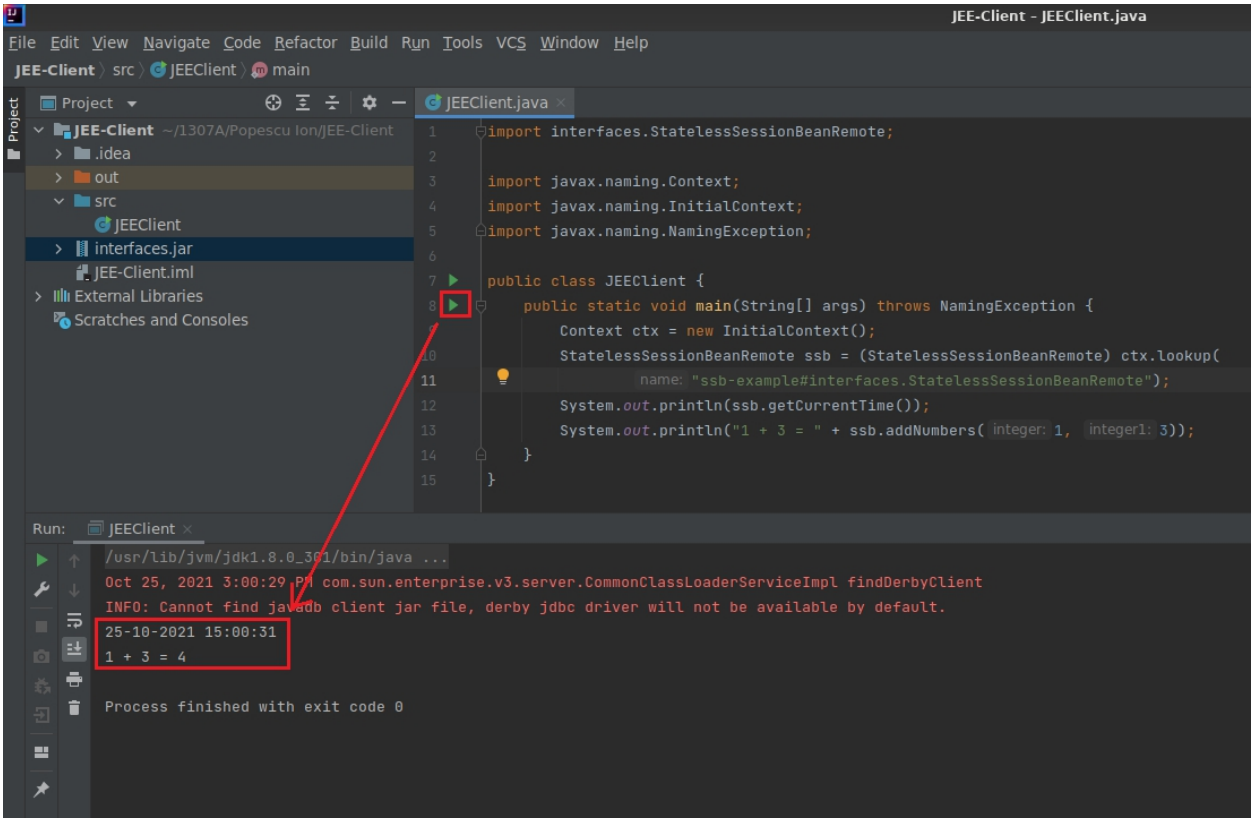
Apoi, se caută în JNDI bean-ul expus sub numele pe care l-aţi găsit şi notat anterior. După ce s-a „pus mâna” pe obiectul stub care apelează metodele remote ale *bean*-ului, se fac apelurile efective de metode: se preia data şi ora curentă, respectiv se face calculul complicat care adună numerele 1 şi 3.

Totuşi, aplicaţia client nu este pregătită de utilizare, deoarece nu sunt disponibile implementările claselor client specifice GlassFish. Acestea se regăsesc în fişierul **`gf-client.jar`** din folder-ul **`glassfish/lib`** al server-ului GlassFish (în acest caz, locaţia este: **`/home/student/opt/glassfish5/glassfish/lib/gf-client.jar`**).

Adăugaţi acest fişier JAR ca dependenţă la proiect, exact cum aţi adăugat şi pachetul **`interfaces.jar`**.

1.12. Testare stateless session bean

Executaţi aplicaţia client prin apăsarea butonului verde din dreptul funcţiei **`main()`**. IntelliJ va crea automat o configuraţie de execuţie Java.



După apel, dacă consultaţi log-ul server-ului Glassfish, puteţi observa mesajele care au fost generate din conţinutul *bean*-ului, după apelarea metodelor încapsulate de acesta:

```
10005  [2020-01-02T18:41:44.755+0200] [glassfish 5.0] [INFO] [] [] [tid: _ThreadID=107 _ThreadName=Thread-8] [timeMillis: 1577983304755] [
       levelValue: 800] [[
10006    [Glassfish] S-a instanţiat un stateless session bean: ejb.StatelessSessionBeanImpl]]
10007
10008  [2020-01-02T18:41:44.755+0200] [glassfish 5.0] [INFO] [] [] [tid: _ThreadID=107 _ThreadName=Thread-8] [timeMillis: 1577983304755] [
       levelValue: 800] [[
10009    [Glassfish] S-a apelat metoda getCurrentTime()]]
10010
10011  [2020-01-02T18:41:44.763+0200] [glassfish 5.0] [INFO] [] [] [tid: _ThreadID=107 _ThreadName=Thread-8] [timeMillis: 1577983304763] [
       levelValue: 800] [[
10012    [Glassfish] S-a apelat metoda addNumbers(1, 3)]]
10013
10014
```

Pentru a observa în timp real apelurile efectuate pe server, deschideţi un terminal şi executaţi următoarea comandă:

```
tail -f <LOCAŢIE_GLASSFISH>/glassfish/domains/domain1/logs/server.log
```

Comanda va afişa în timp real ultimele linii din fişierul log, pe măsură ce acesta este populat. Puteţi deschide unul sau mai mulţi clienţi deodată şi observaţi cum *bean*-ul creat este (re)utilizat pentru clienţi diferiţi.

Atenţie: chiar dacă bean-ul creat este de tip *Stateless Session*, nu înseamnă că server-ul va utiliza o singură instanţă a acestuia mereu! Presupunând că în *container*-ul de *business* există, pentru moment, o singură instanţă a *bean*-ului, iar un client încă „ocupă” acest *bean* (deoarece o metodă apelată încă se execută), atunci server-ul va crea o nouă instanţă a *bean*-ului respectiv şi o va oferi spre utilizare unui alt client. De aici ideea de „Stateless Session”: nu contează care din *bean*-uri este folosit de care client, deoarece nu există ideea de stare menţinută cu un anumit client.

Stateful Session Beans

Spre deosebire de *stateless session beans*, aceste tipuri de *beans* menţin o legătură între clientul apelant şi *bean*-ul care a fost apelat. Aşadar, dacă n clienţi accesează un *stateful session bean*, server-ul este obligat să instanţieze pentru fiecare un *bean* separat, deoarece fiecare *bean* are starea proprie, şi nu este permis ca un client să acceseze informaţii din starea unui *bean* ce aparţine de alt client.

De exemplu, presupunând că pe server există un *bean* ce expune o metodă care afişează lista de e-mail-uri a unui utilizator, cum ar fi ca un alt client să apeleze această metodă din *bean* şi să primească lista dvs. de mail-uri, în loc de lista proprie?

Pentru a ilustra funcţionarea unui *stateful session bean*, veţi crea o aplicaţie de gestiune a unui cont bancar „deschis" în momentul în care un client accesează prima dată una din operaţiunile disponibile:

- depunere numerar
- retragere numerar
- interogare sold

1.13. Codul de business

Mai întâi, creaţi interfaţa de *business* a *bean*-ului, care expune unui client operaţiunile ce pot fi efectuate. Adăugaţi un fişier de tip interfaţă Java în **`ejbs/src/main/java/interfaces`** şi denumiţi-l **`BankAccountBeanRemote`**. Adăugaţi următorul conţinut:

```
package interfaces;

public interface BankAccountBeanRemote {
    Boolean withdraw(Integer amount);
    void deposit(Integer amount);
    Integer getBalance();
}
```

Apoi, creaţi implementarea corespunzătoare bean-ului în pachetul **`ejb`** din acelaşi modul (**`ejbs/src/main/java/`**). Denumiţi clasa Java **`BankAccountBeanImpl`**, spre exemplu.

```
package ejb;

import interfaces.BankAccountBeanRemote;

import java.io.Serializable;

public class BankAccountBeanImpl implements BankAccountBeanRemote,
Serializable {
    private Integer availableAmount = 0;

    public Boolean withdraw(Integer amount) {
        if(availableAmount >= amount) {
            availableAmount -= amount;
            return true;
        } else {
            return false;
        }
    }
```

```
    public void deposit(Integer amount) {
        availableAmount += amount;
    }

    public Integer getBalance() {
        return availableAmount;
    }
}
```

Înregistraţi clasa creată ca şi *stateful session bean* în descriptorii XML din fişierul **ejb-jar.xml**, din **ejbs/src/main/resources/META-INF**. Adăugaţi acest descriptor ca şi subordonat al tag-ului **<enterprise-beans>**:

```
<session>
    <ejb-name>BankAccountBean</ejb-name>
    <mapped-name>bankaccount</mapped-name>
    <business-local>interfaces.BankAccountBeanRemote</business-local>
    <business-remote>interfaces.BankAccountBeanRemote</business-
remote>
    <ejb-class>ejb.BankAccountBeanImpl</ejb-class>
    <session-type>Stateful</session-type>
</session>
```

De această dată, veţi accesa *bean*-ul din *web tier*, spre deosebire de capitolul anterior, unde aţi creat o aplicaţie client Java separată.

1.14. Codul din web tier

În continuare, creaţi meniul de acces pentru client, ca şi pagină JSP. Puteţi folosi pagina **index.jsp** deja existentă ca şi meniu.

Modificaţi **servlets/servlet/src/main/webapp/index.jsp** astfel:

```
<%@ page contentType="text/html; charset=UTF-8" %>
<html>
   <head>
      <title>Meniu principal</title>
      <meta charset="utf-8" />
   </head>
   <body>
      <h1>Meniu principal</h1>
      <h3>Gestiune cont bancar</h3>
      <form action="./process-bank-operation" method="post">
            <fieldset label="operatiuni">
                  <legend>Alegeti operatiunea dorita:</legend>
                  <select name="operation">
                        <option value="deposit">Depunere numerar</option>
                        <option value="withdraw">Retragere
numerar</option>
                        <option value="balance">Interogare sold</option>
                  </select>
                  <br />
                  <br />
                  Introduceti suma: <input type="number" name="amount"
/>
                  <br />
                  <br />
```

```
                <button type="submit">Efectuare</button>
            </fieldset>
        </form>
    </body>
</html>
```

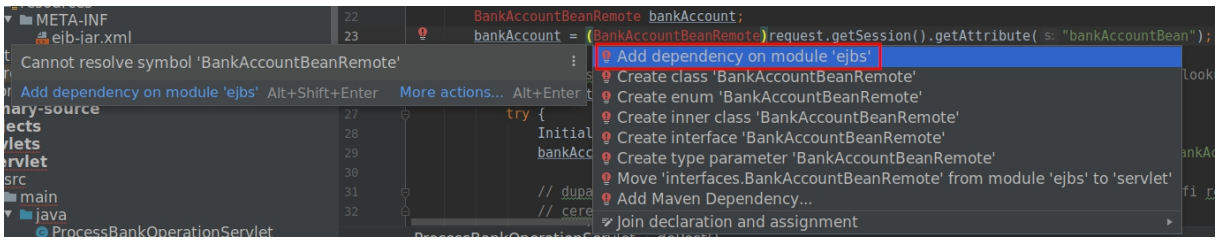
Pagina conţine un formular simplu în care utilizatorul poate selecta operaţiunea dorită şi poate introduce o sumă de bani într-un câmp numeric, utilizat la operaţiunile de retragere / depunere.

Ţinta formularului este un servlet mapat pe ruta de acces **`./process-bank-operation`**, către care datele din formular sunt trimise prin metoda **POST**. Aşadar, acum veţi crea servlet-ul respectiv: o clasă Java denumită **`ProcessBankOperationServlet`**, plasată în modulul **`servlet`**, în următorul folder: **`servlets/servlet/src/main/java`**. Adăugaţi codul următor:

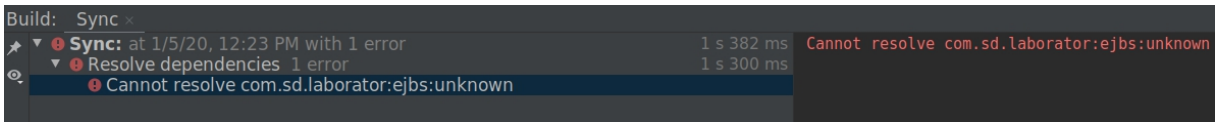
```
import interfaces.BankAccountBeanRemote;

import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

public class ProcessBankOperationServlet extends HttpServlet {
    @Override
    protected void doPost(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {
        // preluare parametri din cererea HTTP
        String operation = request.getParameter("operation");
        String amountString = request.getParameter("amount");

        // nu conteaza suma introdusa in campul numeric daca
operatiunea este de tip "Sold cont"
        Integer amount = (!amountString.equals("")) ?
Integer.parseInt(amountString) : 0;

        // se incearca preluarea bean-ului folosind obiectul
HttpSession, care pastreaza o sesiune HTTP intre client si server
        BankAccountBeanRemote bankAccount;
        bankAccount =
(BankAccountBeanRemote)request.getSession().getAttribute("bankAccountB
ean");

        // daca nu exista nimic pastrat in sesiunea HTTP, inseamna ca
bean-ul se preia prin JNDI lookup
        if (bankAccount == null) {
            try {
                InitialContext ctx = new InitialContext();
                bankAccount = (BankAccountBeanRemote)
ctx.lookup("bankaccount#interfaces.BankAccountBeanRemote");

                // dupa preluarea bean-ului prin JNDI, obiectul se
stocheaza in sesiune pentru a fi refolosit ulterior
                // cererile urmatoare vor utiliza obiectul remote
```

```
stocat in sesiune
                request.getSession().setAttribute("bankAccountBean",
bankAccount);
            } catch (NamingException e) {
                e.printStackTrace();
                return;
            }
        }

        Integer accountBalance = null;
        String message = "";

        // in functie de operatia selectata de client, se apeleaza
metoda corespunzatoare din obiectul remote
        if (operation.equals("deposit")) {
            bankAccount.deposit(amount);
            message = "In contul dvs. au fost depusa suma: " + amount
+ ".";
        } else if (operation.equals("withdraw")) {
            if (bankAccount.withdraw(amount)) {
                message = "Din contul dvs. s-a retras suma de: " +
amount + ".";
            } else {
                message = "Operatiunea a esuat! Fonduri insuficiente.";
            }
        } else if (operation.equals("balance")) {
            accountBalance = bankAccount.getBalance();
        }

        message += "<br /><br />";
        if (accountBalance != null) {
            message += "Sold cont: " + accountBalance;
        }
        message += "<br /><a href='./'>Inapoi la meniul principal</a>";

        // dupa construirea mesajului raspuns, acesta este trimis ca
si continut HTML inapoi la clientul apelant
        response.setContentType("text/html");
        response.getWriter().print(message);
    }
}
```

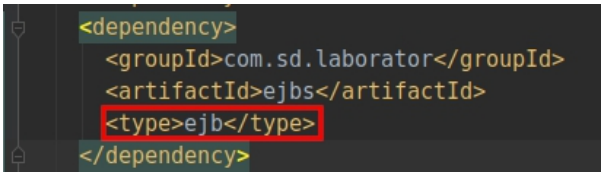
După ce scrieţi codul respectiv în fişierul sursă, veţi observa o problemă: nu aveţi acces din acest modul la pachetul cu *bean*-uri *enterprise* (**`ejbs`**), şi deci nu se poate importa interfaţa **`BankAccountBeanRemote`**. Pentru a rezolva acest lucru, modulul Maven **`ejbs`** trebuie adăugat ca dependenţă la modulul Maven **`servlet`**. IntelliJ ajută utilizatorul în acest sens: daţi click pe **`BankAccountBeanRemote`** marcat cu roşu, apăsaţi **ALT+ENTER** şi alegeţi „**Add dependency on module ejbs**”.



Nu este suficient, deoarece IntelliJ nu poate detecta singur ce tip de modul este cel importat.



Aşa încât, modificaţi manual fişierul **`servlets/servlet/pom.xml`**, iar pentru dependenţa adăugată de IntelliJ, specificaţi şi atributul **`<type>ejb</type>`** (deoarece se importă un modul de tip *Enterprise Java Bean*, conform schemei DTD al fişierului POM).



Nu uitaţi de „**Import changes**” pentru a sincroniza proiectul Maven şi modulele sale.

Corpul servlet-ului conţine cod de tratare a cererilor de tip POST, deoarece formularul care îl ţinteşte trimite datele prin această metodă HTTP (se putea folosi şi GET, spre exemplu).

Bean-ul de tip *stateful* se preia prin ***JNDI lookup*** prima dată când clientul accesează servlet-ul. În acel moment, server-ul îi creează şi asociază o instanţă a *bean*-ului în contextul de execuţie, iar această instanţă este reţinută de servlet în sesiunea HTTP, deoarece este nevoie de ea la fiecare cerere. **Dacă nu aţi fi reţinut instanţa primită, cererile ulterioare vor prelua instanţe noi de *bean* de la server** (**chiar dacă este *stateful*! Starea este păstrată pe aceeaşi instanţă. În momentul în care se face *lookup* din nou, e ca şi cum aţi cere o instanţă nouă, deci aţi pierdut starea**).

Cererile ulterioare primei vor prelua *bean*-ul din sesiune, iar apelurile metodelor acestuia vor ţine cont de starea anterioară, deoarece clientul este acelaşi, iar server-ul *enterprise* va folosi aceeaşi instanţă din memorie pentru a deservi clientul respectiv. Dacă un alt client cere o instanţă de *bean*, server-ul va crea o instanţă nouă şi i-o va asocia lui, păstrându-i starea în cererile ulterioare, ş.a.m.d, pentru fiecare client **diferit**.

Nu uitaţi să creaţi descriptorul XML pentru servlet, pentru ca server-ul *enterprise* să îl mapeze pe ruta dorită şi să îl gestioneze corespunzător.

Deschideţi **`servlets/servlet/src/main/webapp/WEB-INF/web.xml`** şi adăugaţi o nouă mapare pentru servlet-ul **`ProcessBankOperationServlet`**:

```
<servlet>
    <servlet-name>ProcessBankOperation</servlet-name>
    <servlet-class>ProcessBankOperationServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>ProcessBankOperation</servlet-name>
```

```
    <url-pattern>/process-bank-operation</url-pattern>
</servlet-mapping>
```

Urmează paşii cu care deja v-aţi obişnuit: **compile → package → redeploy de la consolă**.

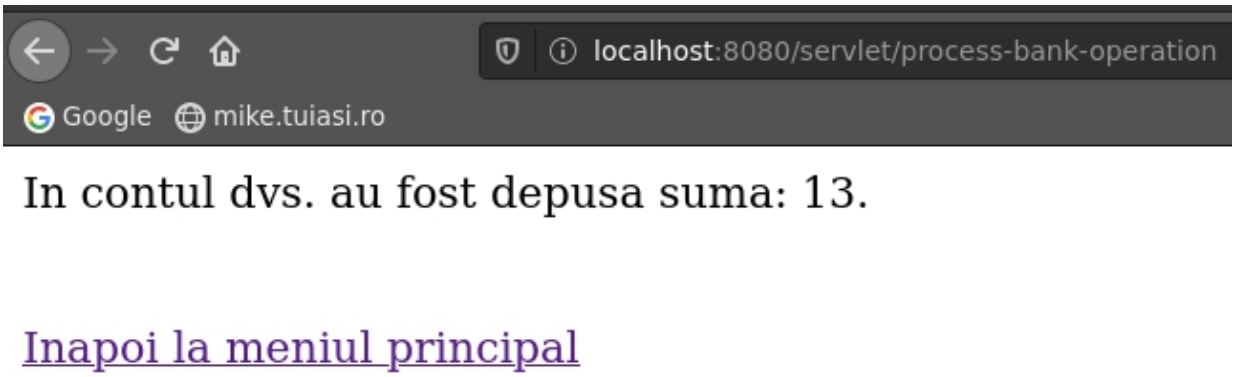
1.15. Testare stateful session bean

Testarea *bean*-ului de tip *stateful* presupune existenţa a 2 clienţi diferiţi (sesiuni diferite HTTP). Aveţi 2 variante, întrucât server-ul este local: folosiţi 2 browsere diferite, sau folosiţi 1 singur browser, o dată din modul obişnuit de navigare, şi a doua oară din modul **incognito**.

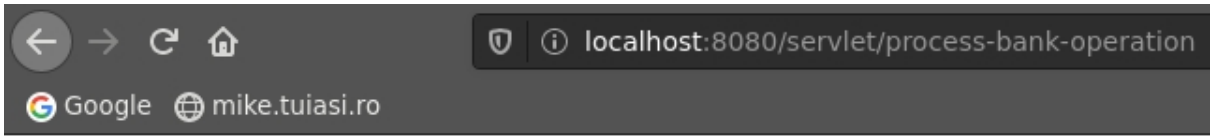
Deschideţi un browser şi navigaţi la adresa:

http://localhost:8080/servlet/

Selectaţi **Depunere numerar**, introduceţi o sumă şi apoi apăsaţi pe „**Efectuare**”. În acest moment, clientului curent (browser-ul web, în acest caz) a primit o instanţă de *stateful session bean*, iar aceasta a fost stocată în sesiune.



Înapoi la meniul principal şi verificaţi soldul. Valoarea este preluată din *bean*-ul existent în memoria server-ului *enterprise*.

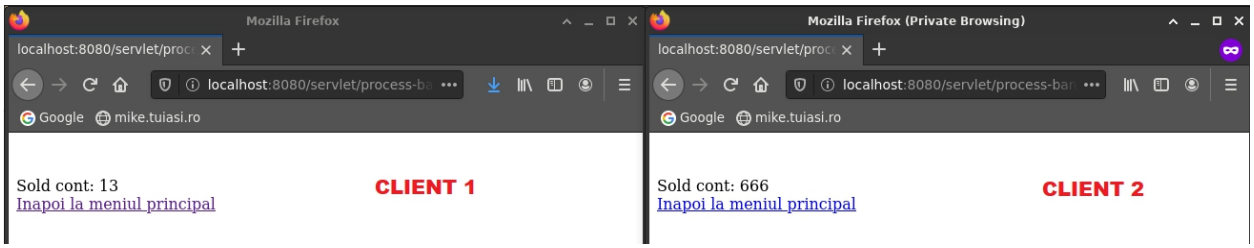


Sold cont: 13
Inapoi la meniul principal

Puteţi testa alte operaţiuni de retragere / depunere pentru a confirma că starea *bean*-ului este păstrată de la un apel la altul.

Acum deschideţi un al 2-lea browser (sau modul incognito dacă folosiţi acelaşi browser: **CTRL+SHIFT+N** pentru Google Chrome sau **CTRL+SHIFT+P** pentru Mozilla Firefox).

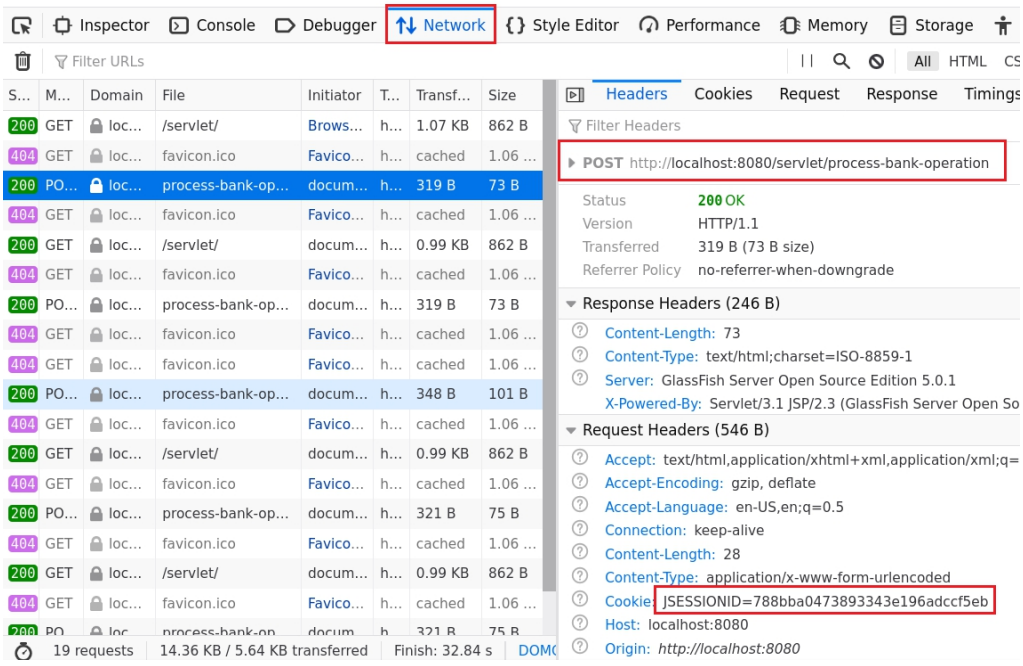
Faceţi o depunere şi observaţi că soldul diferă de cel al primului client, deoarece acesta este un client nou, deci server-ul va utiliza o altă instanţă de *bean* atunci când se face căutarea (*lookup*) în JNDI.



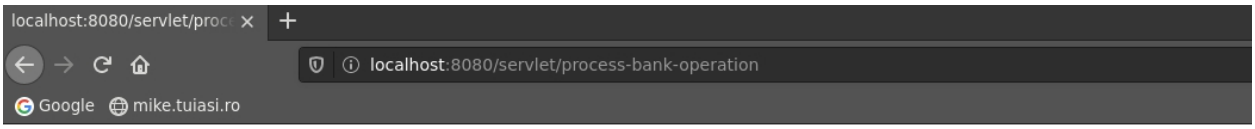
1.15.1. Cookie-ul de sesiune

Sesiunea clientului este stocată într-un cookie în browser-ul client-ului. Acest cookie este axat la fiecare request către servlet, și în așa fel este detectată sesiunea corespunzătoare.

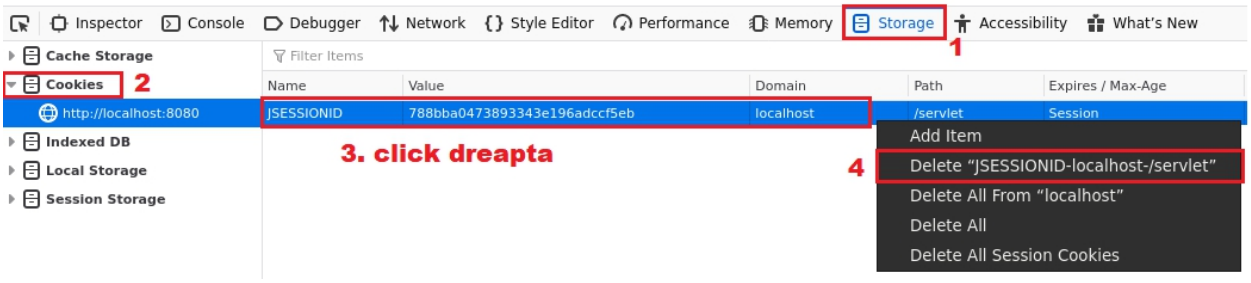
Pentru a verifica acest lucru, deschideți bara de Network (F12) și inspectați request-urile făcute. Veți observa că în header-ul Cookie al request-urilor este axat JSESSIONID.



Pentru a verifica faptul că sesiunea este stocată în cookie, deschideți pagina de interogare a sold-ului. Ulterior, intrați din nou în toolbar-ul de Inspect la secțiunea Storage -> Cookies și stergeți cookie-ul JSESSIONID. Dați refresh la pagină și veți observa că sold-ul a devenit 0 (s-a resetat sesiunea), și un alt cookie de JSESSIONID a fost generat.



Sold cont: 123
Inapoi la meniul principal



Exercițiu: Încercați acum din alt browser sau din incognito sa accesați pagina de interogare a sold-ului. Setați manual valoarea Cookie-ul JSESSIONID cu valoarea din sesiunea curentă și reîncărcați pagina. Ce ați observat? Prezintă acest lucru o problemă și de ce?

Mai multe despre cookie-uri veți afla la disciplina de Programare Web.

JPA Entities

Entităţile JPA (*JPA Entities*) au fost precedate, ca tehnologie, de *Entity Beans*. Începând de la standardul EJB 3.0, *Entity Bean*-urile sunt marcate ca învechite şi, de aceea, veţi folosi tehnologia nouă în acest capitol al laboratorului.

1.16. Configurare persistenţă

Pentru a folosi *Java Persistence API* în proiectul JEE, trebuie să adăugaţi un descriptor de persistenţă într-un fişier denumit **persistence.xml**, ce rezidă în folder-ul **ejbs/src/main/resources/META-INF**. Creaţi fişierul şi adăugaţi următorul conţinut:

```
<persistence xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
             version="2.0"
             xmlns="http://java.sun.com/xml/ns/persistence">
    <persistence-unit name="bazaDeDateSQLite" transaction-
type="RESOURCE_LOCAL">

<provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
        <class>ejb.StudentEntity</class>
        <properties>
            <property name="javax.persistence.jdbc.driver"
value="org.sqlite.JDBC" />
            <property name="javax.persistence.jdbc.url"
value="jdbc:sqlite:/home/student/studenti.db" />
            <property name="eclipselink.logging.level" value="ALL" />
            <property name="eclipselink.ddl-generation" value="create-
tables" />
        </properties>
    </persistence-unit>
</persistence>
```

În acest XML s-a definit o aşa-numită unitate de persistenţă (*persistence unit*), adică o grupare logică a tuturor entităţilor gestionate de instanţele unui **EntityManager** (**detalii despre această clasă în cele ce urmează**) în aplicaţia JEE. Unitatea de persistenţă poate folosi o bază de date SQL / NoSQL, o bază de date *embedded* etc. În acest caz s-a utilizat o bază de date SQLite, în care tranzacţiile se fac doar cu „resurse locale”, deoarece această bază de date este complet încapsulată într-un fişier cu extensia **db**. S-a marcat acest lucru prin atributul **transaction-type="RESOURCE_LOCAL"**.

JPA trebuie configurat în aşa fel încât să ştie ce furnizor şi *driver* de persistenţă să folosească atunci când utilizatorul lucrează cu acest API şi cu datele ce se transmit. În acest caz, furnizorul (*provider*-ul) de persistenţă este interfaţa **PersistenceProvider** din implementarea JPA **EclipseLink** - disponibilă implicit odată cu server-ul GlassFish.

Pentru unitatea de persistenţă se specifică, la început (după tag-ul **<provider>**), care sunt clasele ce vor fi gestionate de **EntityManager** - clasele de domeniu, adică cele care sunt mapate în baza de date sub formă de tabele.

Apoi, se dau proprietăţile furnizorului de persistenţă: driver-ul folosit, calea pe disc către fişierul unde este încapsulată baza de date SQLite (în acest caz, **/home/student/studenti.db**), nivelul de *logging* şi, la final, este specificat faptul că

EclipseLink va crea schema bazei de date folosind o interogare de tip „**`CREATE TABLE...`**” în momentul în care aplicaţia este încărcată pe server.

Atenţie: fişierul pe care îl specificaţi ca şi bază de date SQLite trebuie să se afle într-o locaţie în care utilizatorul care a pornit server-ul GlassFish are drepturi de scriere!

Pentru ca aplicaţia JEE să poată utiliza *driver*-ul JDBC pentru SQLite (clasa **`org.sqlite.JDBC`**), acest driver trebuie adăugat ca dependenţă în modulul Maven în care este folosit, adică în modulul **`ejbs`**. Deschideţi fişierul **`ejbs/pom.xml`** şi adăugaţi următoarea dependenţă:

```
<dependency>
  <groupId>org.xerial</groupId>
  <artifactId>sqlite-jdbc</artifactId>
  <version>3.35.0.1</version>
</dependency>
```

1.17. Adăugare entitate JPA

Creaţi o entitate JPA denumită **`StudentEntity`**, ca şi clasă Java situată în **`/ejbs/src/main/java/ejb`**. Codul clasei este următorul:

```
package ejb;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Entity
public class StudentEntity {
    @Id
    @GeneratedValue
    private int id;
    private String nume;
    private String prenume;
    private int varsta;

    public StudentEntity() {
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getNume() {
        return nume;
    }

    public void setNume(String nume) {
        this.nume = nume;
    }

    public String getPrenume() {
```

```
        return prenume;
    }

    public void setPrenume(String prenume) {
        this.prenume = prenume;
    }

    public int getVarsta() {
        return varsta;
    }

    public void setVarsta(int varsta) {
        this.varsta = varsta;
    }
}
```

Observaţi că entitatea JPA seamănă cu un **POJO** (*Plain Old Java Object*) ce conţine adnotări corespunzătoare, unde este cazul: adnotarea **`@Entity`** marchează faptul că această clasă este o entitate JPA, adnotarea **`@Id`** indică după care câmp vor fi identificate înregistrările din tabela rezultată în baza de date (cheia primară), adnotarea **`@GeneratedValue`** marchează faptul că acel câmp adnotat (în acest caz, câmpul **`id`**) conţine o valoare generată automat la inserarea unei înregistrări - nu trebuie dat explicit de utilizator.

Câmpurile încapsulate trebuie să fie tipuri de date simple (primitive sau nu), şi să fie serializabile.

1.18. Codul din web tier

Pentru a ilustra lucrul cu această entitate JPA simplă, veţi folosi acelaşi formular de introducere a datelor unui student, utilizat în laboratorul 1. Adăugaţi codul acestuia în fişierul **`servlets/servlet/src/main/webapp/formular.jsp`**.

```
<html xmlns:jsp="http://java.sun.com/JSP/Page">
   <head>
      <title>Formular student</title>
      <meta charset="UTF-8" />
   </head>
   <body>
      <h3>Formular student</h3>
      Introduceti datele despre student:
      <form action="./process-student" method="post">
            Nume: <input type="text" name="nume" />
            <br />
            Prenume: <input type="text" name="prenume" />
            <br />
            Varsta: <input type="number" name="varsta" />
            <br />
            <br />
            <button type="submit" name="submit">Trimite</button>
      </form>
   </body>
</html>
```

Formularul va prelua aceste date şi le va trimite unui servlet spre procesare. Servlet-ul preia datele din cererea HTTP şi le persistă într-o bază de date *embedded* de tip **SQLite** (încapsulată într-un fişier pe disc). Creaţi acest servlet în

servlets/servlet/src/main/java/ProcessStudentServlet.java, cu următorul conţinut:

```
import ejb.StudentEntity;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

public class ProcessStudentServlet extends HttpServlet {
    @Override
    protected void doPost(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {
        // se citesc parametrii din cererea de tip POST
        String nume = request.getParameter("nume");
        String prenume = request.getParameter("prenume");
        int varsta = Integer.parseInt(request.getParameter("varsta"));

        // pregatire EntityManager
        EntityManagerFactory factory =
Persistence.createEntityManagerFactory("bazaDeDateSQLite");
        EntityManager em = factory.createEntityManager();

        // creare entitate JPA si populare cu datele primite din
formular
        StudentEntity student = new StudentEntity();
        student.setNume(nume);
        student.setPrenume(prenume);
        student.setVarsta(varsta);

        // adaugare entitate in baza de date (operatiune de
persistenta)
        // se face intr-o tranzactie
        EntityTransaction transaction = em.getTransaction();
        transaction.begin();
        em.persist(student);
        transaction.commit();

        // inchidere EntityManager
        em.close();
        factory.close();

        // trimitere raspuns inapoi la client
        response.setContentType("text/html");
        response.getWriter().println("Datele au fost adaugate in baza
de date." +
                "<br /><br /><a href='./'>Inapoi la meniul
principal</a>");
    }
}
```

Clasa **EntityManager** este folosită, după cum este şi denumită, la gestiunea tuturor entităţilor persistente (*JPA Entities*) de care este nevoie în aplicaţia JEE. O instanţă **EntityManager** se obţine de la **EntityManagerFactory**, pe baza numelui unităţii de persistenţă.

Creaţi încă un servlet folosit pentru afişarea listei de studenţi disponibilă în baza de date. Denumiţi servlet-ul **FetchStudentListServlet** şi adăugaţi următorul conţinut în fişierul sursă:

```
import ejb.StudentEntity;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.TypedQuery;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.List;

public class FetchStudentListServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {
        // pregatire EntityManager
        EntityManagerFactory factory =
Persistence.createEntityManagerFactory("bazaDeDateSQLite");
        EntityManager em = factory.createEntityManager();

        StringBuilder responseText = new StringBuilder();
        responseText.append("<h2>Lista studenti</h2>");
        responseText.append("<table
border='1'><thead><tr><th>ID</th><th>Nume</th><th>Prenume</th><th>Vars
ta</th></thead>");
        responseText.append("<tbody>");

        // preluare date studenti din baza de date
        TypedQuery<StudentEntity> query = em.createQuery("select
student from StudentEntity student", StudentEntity.class);
        List<StudentEntity> results = query.getResultList();
        for (StudentEntity student : results) {
            // se creeaza cate un rand de tabel HTML pentru fiecare
student gasit
            responseText.append("<tr><td>" + student.getId() +
"</td><td>" +
                    student.getNume() + "</td><td>" +
student.getPrenume() +
                    "</td><td>" + student.getVarsta() + "</td></tr>");
        }

        responseText.append("</tbody></table><br /><br /><a
href='./'>Inapoi la meniul principal</a>");

        // inchidere EntityManager
```

```
        em.close();
        factory.close();

        // trimitere raspuns la client
        response.setContentType("text/html");
        response.getWriter().print(responseText.toString());
    }
}
```

Observaţi că preluarea listei de studenţi se face printr-o interogare care seamănă cu limbajul SQL. Acest limbaj se numeşte **JPQL** (***Java Persistence Query language***).

Nu uitaţi de maparea servleţilor prin descriptori XML. Modificaţi **`web.xml`** astfel:

```
...

<servlet>
   ...
</servlet>
<servlet>
    <servlet-name>ProcessStudent</servlet-name>
    <servlet-class>ProcessStudentServlet</servlet-class>
</servlet>
<servlet>
    <servlet-name>FetchStudentList</servlet-name>
    <servlet-class>FetchStudentListServlet</servlet-class>
</servlet>

<servlet-mapping>
    ...
</servlet-mapping>
<servlet-mapping>
    <servlet-name>ProcessStudent</servlet-name>
    <url-pattern>/process-student</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>FetchStudentList</servlet-name>
    <url-pattern>/fetch-student-list</url-pattern>
</servlet-mapping>

...
```

Modificaţi meniul principal din pagina **`index.jsp`** ca să includeţi ancore către formularul de adăugare a unui student, respectiv către servlet-ul care afişează lista studenţilor:

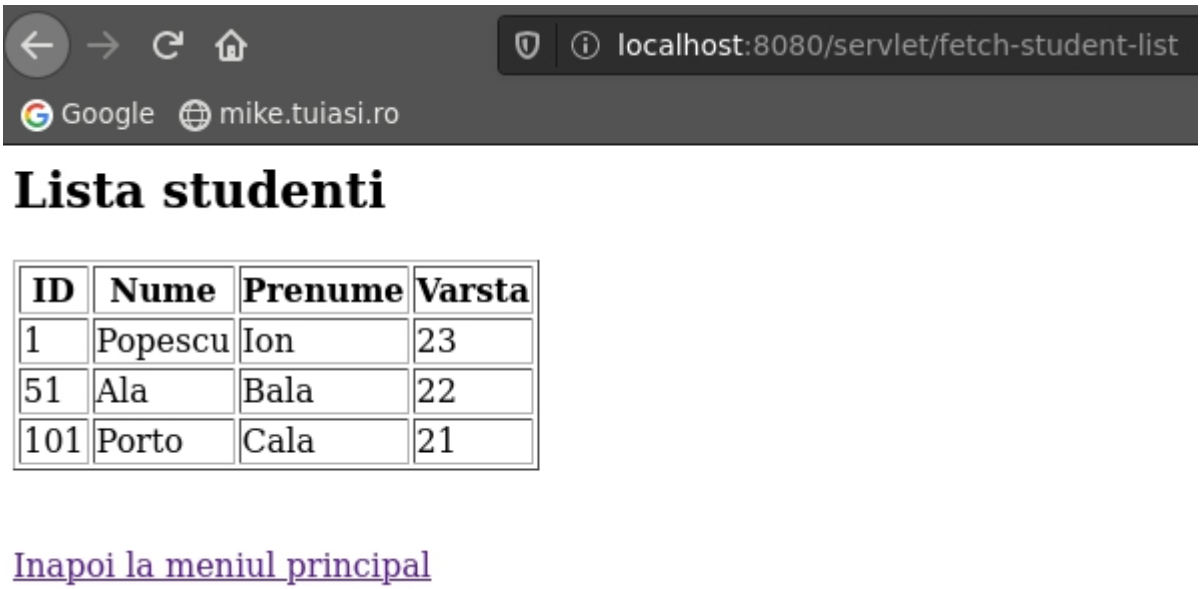
```
...
...
</form>
<hr />
<h3>Baza de date cu studenti</h3>
<a href="./formular.jsp">Adaugare student</a>
<br />
<a href="./fetch-student-list">Afisare lista studenti</a>
...
...
```

1.19. Testare entitate JPA

Curăţaţi aplicaţia (**`clean`**), apoi compilaţi (**`compile`**), împachetaţi aplicaţia (**`package`**) în artefact EAR şi încărcaţi-o pe server (**`redeploy`** dacă aţi mai încărcat-o anterior, sau **`deploy`** dacă e prima dată când încărcaţi - nu uitaţi, **de la consolă**).

Navigaţi la http://localhost:8080/servlet.

Adăugaţi studenţi în baza de date din meniul corespunzător, apoi testaţi dacă lista de studenţi poate fi preluată din baza de date (din fişierul **`studenti.db`**).



Aplicaţii şi teme

- **Temă de laborator**

Adăugaţi operaţiile de actualizare şi de ştergere a unui student din baza de date SQLite:

– modificaţi meniul principal din pagina **`index.jsp`** pentru a indica aceste operaţii

– adăugaţi încă 2 servleţi corespunzători acestor operaţii. În aceşti servleţi, folosiţi clasa **`EntityManager`** şi limbajul **JPQL** (*Java Persistence Query Language*) pentru a implementa operaţiile cerute. Sugestie:

– actualizare: căutaţi studentul în baza de date după nume şi / sau prenume, apoi preluaţi identificatorul acestuia (ID-ul). Având acest ID, faceţi o operaţie de UPDATE folosind JPQL cu noile date preluate de la utilizator (eventual dintr-un formular JSP).

– ştergere: căutaţi studentul în baza de date după nume şi / sau prenume, apoi preluaţi identificatorul acestuia (ID-ul). Având acest ID, faceţi o operaţie de DELETE folosind JPQL, după ID-ul preluat.

– **Alternativă**: atunci când se preiau studenţii pentru afişare, puteţi prelua şi ID-ul din baza de date şi să îl setaţi ca parametru URL pentru servlet-ul apelat la click pe un buton de ştergere / actualizare pus în dreptul fiecărui student, asemănător cu:

```
<a href="./update-student?id=ID&nume=NUME&prenume=...">Actualizeaza</a>
<a href="./delete-student?id=ID">Sterge</a>
```

- **Temă pe acasă**

Creați într-un thread sau proces separate un mecanism de monitorizare a bazei de date care pe baza unei reguli de tipul dacă valoare dintr-un câmp numeric (trimis ca parametru) iese din intervalul [a,b] unde a,b trimiși ca parametri să genereze o pagina de alarmare unde să afișeze și numele câmpului și valoarea care a condus la depășire. Se vor monitoriza măcar două câmpuri din baza de date.

Bibliografie

Enterprise Java Beans - https://docs.oracle.com/cd/E24329_01/web.1211/e24446/ejbs.htm
Accesarea bean-urilor - https://docs.oracle.com/javaee/7/tutorial/ejb-intro004.htm
Introducere în Java Persistence API - https://docs.oracle.com/javaee/7/tutorial/persistence-intro.htm
JPA Entities - https://docs.oracle.com/cd/E16439_01/doc.1013/e13981/undejbs003.htm
Persistence Units - https://docs.oracle.com/cd/E19798-01/821-1841/bnbrj/index.html
JPQL Language reference - https://docs.oracle.com/html/E13946_04/ejb3_langref.html
Entity Beans - https://docs.oracle.com/cd/B14099_19/web.1012/b15505/entity.htm
Message-Driven Beans - https://docs.oracle.com/javaee/6/tutorial/doc/gipko.html