

I. Konfiguracja serwera Wildfly 20:

1. Konfiguracja bazy danych:
 - a. Pobieramy i dodajemy sterownik do bazy danych np. postgresql poprzez kosciole zarządzanie Wildfly pod adresem: <http://localhost:9990/console/index.html>.


Add Deployment

Upload Deployment

Specify Names

1

2



postgresql-42.2.14.jar

Cancel

< Back

Next >

- b. Modyfikujemy plik standalone.xml dodając data source dla postgresql:

```
<subsystem xmlns="urn:jboss:domain:core-management:1.0"/>
<subsystem xmlns="urn:jboss:domain:databases:6.0">
  <datasources>
    <datasource jndi-name="java:jboss/datasources/ExampleDS" pool-name="ExampleDS" enabled="true" use-java-context="true" statistics-enabled="${wildfly.databases.statistics-enabled:${wildfly.statistics-enabled:false}}">
      <connection-url>jdbc:h2:mem:test;DB_CLOSE_DELAY=-1;DB_CLOSE_ON_EXIT=FALSE</connection-url>
      <driver>h2</driver>
      <security>
        <user-name>sa</user-name>
        <password>sa</password>
      </security>
    </datasource>
    <datasource jndi-name="java:PostgresDS" pool-name="PostgresDS" statistics-enabled="true">
      <connection-url>jdbc:postgresql://localhost:5432/soa_2020-eqzamin</connection-url>
      <driver-class>org.postgresql.Driver</driver-class>
      <driver>postgresql-42.2.14.jar</driver>
      <security>
        <user-name>postgres</user-name>
        <password>postgres</password>
      </security>
      <validation>
        <valid-connection-checker class-name="org.jboss.jca.adapters.jdbc.extensions.postgres.PostgreSQLValidConnectionChecker"/>
        <background-validation>true</background-validation>
        <exception-sorter class-name="org.jboss.jca.adapters.jdbc.extensions.postgres.PostgreSQLExceptionSorter"/>
      </validation>
    </datasource>
  </drivers>
  <drivers>
    <driver name="postgres" module="org.postgres">
      <driver-class>org.postgresql.Driver</driver-class>
    </driver>
    <driver name="h2" module="com.h2database.h2">
      <xa-datasource-class>org.h2.jdbcx.JdbcDataSource</xa-datasource-class>
    </driver>
  </drivers>
</subsystem>
```

2. Konfiguracja kolejki JMS:

a. Podmieniamy plik standalone.xml na standalone-full.xml, żeby dodać obsługę activemq.

b. Modyfikujemy plik standalone.xml dodając jms-queue:

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:10.0">
  <server name="default">
    <statistics enabled="${wildfly.messaging-activemq:statistics-enabled:${wildfly:statistics-enabled:false}}"/>
    <security-setting name="#">
      <role name="guest" send="true" consume="true" create-non-durable-queue="true" delete-non-durable-queue="true"/>
    </security-setting>
    <address-setting name="#" dead-letter-address="jms.queue.DLQ" expiry-address="jms.queue.ExpiryQueue" max-size-bytes="10485760" page-size-bytes="2097152" message-counter-history-day-limit="10"/>
    <http-connector name="http-connector" socket-binding="http" endpoint="http-acceptor"/>
    <http-connector name="http-connector-throughput" socket-binding="http" endpoint="http-acceptor-throughput">
      <param name="batch-delay" value="50"/>
    </http-connector>
    <in-vm-connector name="in-vm" server-id="0">
      <param name="buffer-pooling" value="false"/>
    </in-vm-connector>
    <http-acceptor name="http-acceptor" http-listener="default"/>
    <http-acceptor name="http-acceptor-throughput" http-listener="default">
      <param name="batch-delay" value="50"/>
      <param name="direct-deliver" value="false"/>
    </http-acceptor>
    <in-vm-acceptor name="in-vm" server-id="0">
      <param name="buffer-pooling" value="false"/>
    </in-vm-acceptor>
    <jms-queue name="ExpiryQueue" entries="java:/jms/queue/ExpiryQueue"/>
    <jms-queue name="DLQ" entries="java:/jms/queue/DLQ"/>
    <jms-queue name="SOA_LabQueue" entries="java:jboss/exported/jms/queue/SOA_2020-egzamin jms/queue/SOA_2020-egzamin" durable="true"/>
    <connection-factory name="InVmConnectionFactory" entries="java:/ConnectionFactory" connectors="in-vm"/>
    <connection-factory name="RemoteConnectionFactory" entries="java:jboss/exported/jms/RemoteConnectionFactory" connectors="http-connector"/>
    <pooled-connection-factory name="activemq-ra" entries="java:/JmsXA java:jboss/DefaultJMSConnectionFactory" connectors="in-vm" transaction="xa"/>
  </server>
</subsystem>
```

3. Konfiguracja dostępu do web service SOAP:

a. Modyfikujemy plik standalone.xml dodając security-domain:

```
<subsystem xmlns="urn:jboss:domain:security:2.0">
  <security-domains>
    <security-domain name="other" cache-type="default">
      <authentication>
        <login-module code="Remoting" flag="optional">
          <module-option name="password-stacking" value="useFirstPass"/>
        </login-module>
        <login-module code="RealmDirect" flag="required">
          <module-option name="password-stacking" value="useFirstPass"/>
        </login-module>
      </authentication>
    </security-domain>
    <security-domain name="SOA_2020-egzamin-security-domain" cache-type="default">
      <authentication>
        <login-module code="UsersRoles" flag="required">
          <module-option name="usersProperties" value="${jboss.server.config.dir}/application-users.properties"/>
          <module-option name="rolesProperties" value="${jboss.server.config.dir}/application-roles.properties"/>
          <module-option name="unauthenticatedIdentity" value="nobody"/>
        </login-module>
      </authentication>
    </security-domain>
    <security-domain name="jboss-web-policy" cache-type="default">
      <authorization>
        <policy-module code="Delegating" flag="required"/>
      </authorization>
    </security-domain>
  </security-domains>
</subsystem>
```

Ustawiamy users roles application-users.properties i application-roles.properties znajdujące się w standalone/configuration i konfigurowanie po stronie Wildfly skryptem add-user.bat/sh.

b. Dodajemy użytkowników i ich przynależność do grup za pomocą skryptu add-user.bat/sh:

```
C:\WINDOWS\system32\cmd.exe

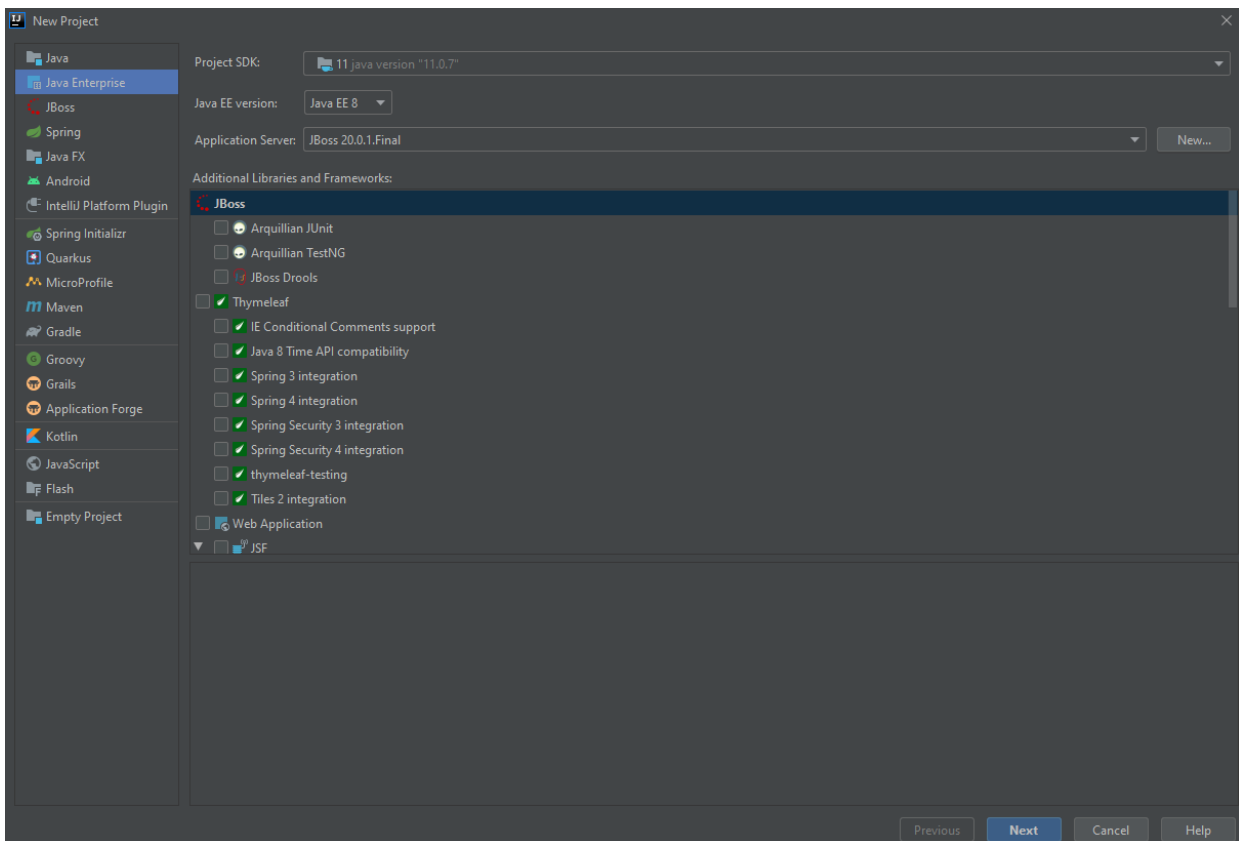
What type of user do you wish to add?
a) Management User (mgmt-users.properties)
b) Application User (application-users.properties)
(a): b

Enter the details of the new user to add.
Using realm 'ApplicationRealm' as discovered from the existing property files.
Username : example
Password recommendations are listed below. To modify these restrictions edit the add-user.properties configuration file.
- The password should be different from the username
- The password should not be one of the following restricted values {root, admin, administrator}
- The password should contain at least 8 characters, 1 alphabetic character(s), 1 digit(s), 1 non-alphanumeric symbol(s)
Password : JFLYDM0098: The password should be different from the username
Are you sure you want to use the password entered yes/no? yes
Re-enter Password :
What groups do you want this user to belong to? (Please enter a comma separated list, or leave blank for none)[ ]: exampleGroup
About to add user 'example' for realm 'ApplicationRealm'
Is this correct yes/no? yes
Added user 'example' to file 'C:\wildfly-20.0.1.Final\standalone\configuration\application-users.properties'
Added user 'example' to file 'C:\wildfly-20.0.1.Final\domain\configuration\application-users.properties'
Added user 'example' with groups exampleGroup to file 'C:\wildfly-20.0.1.Final\standalone\configuration\application-roles.properties'
Added user 'example' with groups exampleGroup to file 'C:\wildfly-20.0.1.Final\domain\configuration\application-roles.properties'
Is this new user going to be used for one AS process to connect to another AS process?
e.g. for a slave host controller connecting to the master or for a Remoting connection for server to server EJB calls.
yes/no? no
Press any key to continue . . .
```

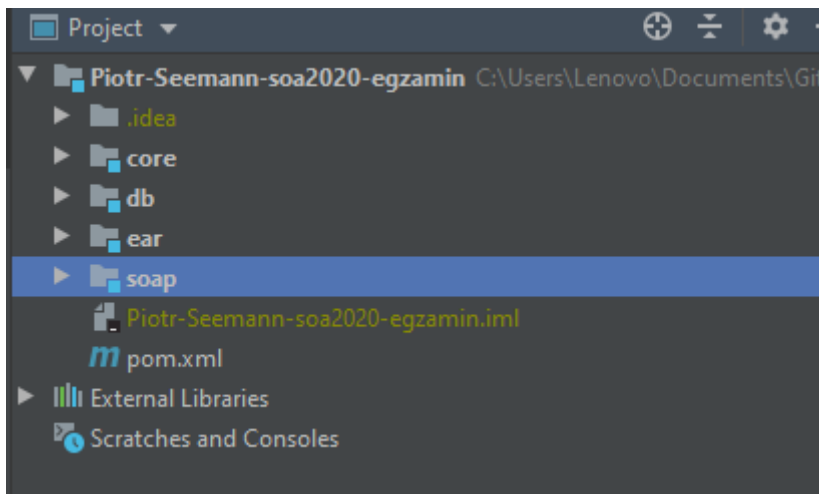
W powyższym przykładzie username: example, grupa: exampleGroup

II. Tworzenie aplikacji krok po kroku:

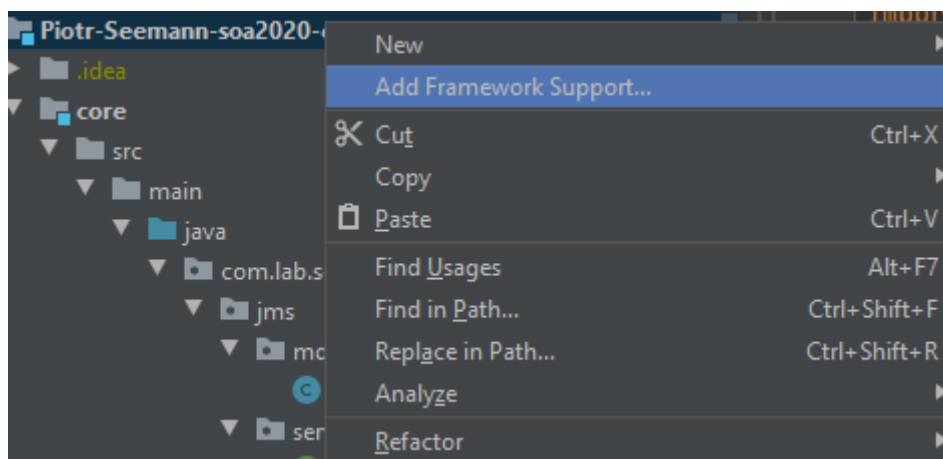
1. Tworzymy nowy projekt Java EE:



2. W projekcie dodajemy moduły:
 - soap – klasa web service SOAP
 - db – klasy JPA i DAO
 - core – klasy realizujące logikę aplikacji (m. in. JMS)
 - ear – wrapper całości



3. Do modułu głównego oraz każdego podmodułu dodajemy Maven framework support klikając na nim PPM oraz Web Application do modułu soap.



4. Konfigurujemy główny plik pom.xml dodając:
 - packaging type - pom
 - podmoduły
 - wybraną wersję Javy
 - zależności dla Javy EE oraz Lombok – przydatna biblioteka, która pozwala na automatyczne generowanie powtarzalnego kodu przy pomocy anotacji.
 - maven-wildfly-plugin, który pozwala na deployowanie aplikacji na serwer Wildfly zgodnie z konfiguracją Mavena, którą dodany w pliku pom w module ear, dlatego tutaj pomijamy konfigurację.

```
?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.lab.soa</groupId>
  <artifactId>Piotr-Seemann-soa2020-egzamin</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>pom</packaging>

  <name>${project.artifactId}</name>

  <modules>
    <module>soap</module>
    <module>db</module>
    <module>core</module>
    <module>ear</module>
  </modules>
```

```
  <properties>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <dependency>
      <groupId>javax</groupId>
      <artifactId>javaee-api</artifactId>
      <version>8.0</version>
      <scope>provided</scope>
    </dependency>

    <dependency>
      <groupId>org.projectlombok</groupId>
      <artifactId>lombok</artifactId>
      <version>1.18.12</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>

  <build>
    <finalName>${project.artifactId}</finalName>
    <plugins>
      <plugin>
        <groupId>org.wildfly.plugins</groupId>
        <artifactId>wildfly-maven-plugin</artifactId>
        <version>2.0.2.Final</version>
        <configuration>
          <skip>true</skip>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

5. Zaczynamy od obsługi bazy danych w module db, w pliku pom.xml dodajemy dependencje jdbc i hibernate oraz typ pakowania jar.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <parent>
    <groupId>com.lab.soa</groupId>
    <artifactId>Piotr-Seemann-soa2020-egzamin</artifactId>
    <version>1.0-SNAPSHOT</version>
  </parent>

  <artifactId>db</artifactId>

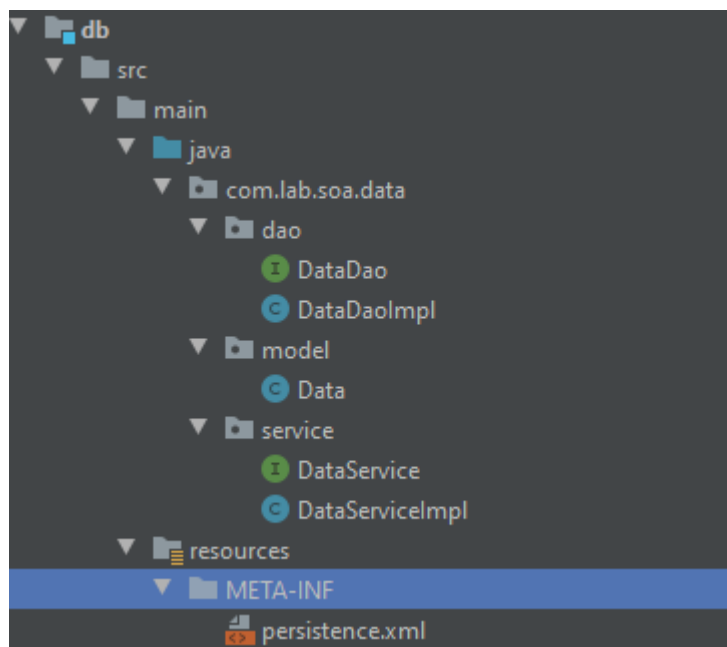
  <packaging>jar</packaging>

  <dependencies>
    <dependency>
      <groupId>org.clojure</groupId>
      <artifactId>java.jdbc</artifactId>
      <version>0.7.9</version>
    </dependency>

    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-core</artifactId>
      <version>5.4.15.Final</version>
    </dependency>
  </dependencies>

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

6. Tworzymy:
model danych - Data
dao do obsługi podstawowych zapytań do bazy dostępny przez interface DataDao
service pośredniczący między dao, a resztą aplikacji dostępny przez interface DataService
konfigurację bazy w pliku persistence.xml



7. W modelu Data dodajemy pola id, createData, messageReceivedData oraz data. Widać tutaj przydatność lomboka, który pozwala dodać konstruktory, gettery i settery oraz toString przy minimalnej ilości kodu.

```
package com.lab.soa.data.model;

import lombok.*;
import org.hibernate.annotations.CreationTimestamp;

import javax.persistence.*;
import java.util.Date;

@Getter
@Setter
@NoArgsConstructor
@RequiredArgsConstructor
@ToString
@Entity
@Table(name = "data")
public class Data {
    @Id
    @GeneratedValue
    @Column(name = "id", nullable = false)
    private long id;
    @CreationTimestamp
    @Temporal(TemporalType.TIMESTAMP)
    @Column(name = "create_date", updatable = false)
    private Date createDate;
    @Temporal(TemporalType.TIMESTAMP)
    @Column(name = "message_received_date")
    private Date messageReceivedDate;
    @Column(name = "data")
    @NonNull private String data;
}
```


8. W bezstanowy beanie dao na potrzeby projektu wystarczą nam metody get, create i update. Nazwa persistence context, który konfigurujemy dalej w pliku persistence.xml to SOA_2020-egzamin.

```
@Stateless
public class DataDaoImpl implements DataDao {

    private static final Logger LOG = Logger.getLogger(DataDao.class);

    @PersistenceContext(unitName = "SOA_2020-egzamin")
    EntityManager em;

    @Override
    public Data create(Data data) throws Exception {
        try {
            em.persist(data);
            LOG.info("Saved to database:\n" + data);
        } catch (PersistenceException e) {
            LOG.error((e.getMessage()));
            throw new Exception(e.getMessage());
        }
        return data;
    }

    @Override
    public Data get(long id) {

        Data obj = em.find(Data.class, id);
        LOG.info("Getting from database:\n" + obj);

        return obj;
    }

    @Override
    public Data update(Data data) throws Exception {
        try{
            em.merge(data);
            LOG.info("Updated in database:\n" + data);
        } catch (PersistenceException e) {
            LOG.error((e.getMessage()));
            throw new Exception(e.getMessage());
        }

        return data;
    }
}
```


9. W bezstanowy beanie service na potrzeby projektu dodajemy metody checkIsCompleted – która sprawdza czy operacja się skończyła i został dodany timestamp messageReceivedDate, create i update. Wstrzykujemy DataDao.

```
@Stateless
public class DataServiceImpl implements DataService {

    private static final Logger LOG = Logger.getLogger(DataDao.class);

    @Inject
    DataDao dataDao;

    @Override
    public long create(String data) throws Exception {
        Data obj = dataDao.create(new Data(data));

        return obj.getId();
    }

    @Override
    public boolean checkIsCompleted(long id) {
        Data obj = dataDao.get(id);

        if(obj == null){ return false; }

        return obj.getMessageReceivedDate() != null && !obj.getMessageReceivedDate().equals(obj.getCreateDate());
    }

    @Override
    public Data update(long id) throws Exception {
        Data obj = dataDao.get(id);
        obj.setMessageReceivedDate(new Date());

        return dataDao.update(obj);
    }
}
```

10. W pliku persistence.xml odpowiadający za obsługę połączenia z bazą danych ustawiamy transakcyjność obsługiwaną za nas przez JTA, driver, adres oraz dane logowania do bazy, a także hbm2ddl.auto: create, żeby tworzyć nową bazę przy każdym starcie aplikacji.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<persistence 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_2.xsd"
    version="2.2">

    <persistence-unit name="SOA_2020-egzamin" transaction-type="JTA">
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
        <properties>
            <property name="javax.persistence.jdbc.driver" value="org.postgresql.Driver" />
            <property name="javax.persistence.jdbc.url" value="jdbc:postgresql://localhost:5432/SOA_2020-egzamin?serverTimezone=UTC" />
            <property name="javax.persistence.jdbc.user" value="postgres" /> <!-- DB User -->
            <property name="javax.persistence.jdbc.password" value="postgres" /> <!-- DB Password -->
            <property name="hibernate.dialect" value="org.hibernate.dialect.PostgreSQLDialect"/>
            <property name="hibernate.hbm2ddl.auto" value="create" />
            <property name="hibernate.show_sql" value="true" /> <!-- Show SQL in console -->
            <property name="hibernate.format_sql" value="true" /> <!-- Show SQL formatted -->
        </properties>
    </persistence-unit>
</persistence>
```

11. Następnie w module soap będzie tworzyć webService soapowy, w pliku pom.xml dodajemy dependencje jaxws i security bez com.sun.tools, które nie są nam potrzebne, a nie działają z powodu buga, moduł core oraz typ pakowania war.

```
<artifactId>Piotr-Seemann-soa2020-egzamin</artifactId>
<version>1.0-SNAPSHOT</version>
</parent>

<artifactId>soap</artifactId>

<packaging>war</packaging>

<dependencies>
  <dependency>
    <groupId>com.sun.xml.ws</groupId>
    <artifactId>jaxws-ri</artifactId>
    <version>2.3.1</version>
    <type>pom</type>
  </dependency>

  <dependency>
    <groupId>org.jboss.ws</groupId>
    <artifactId>jbossws-api</artifactId>
    <version>1.1.2.Final</version>
  </dependency>

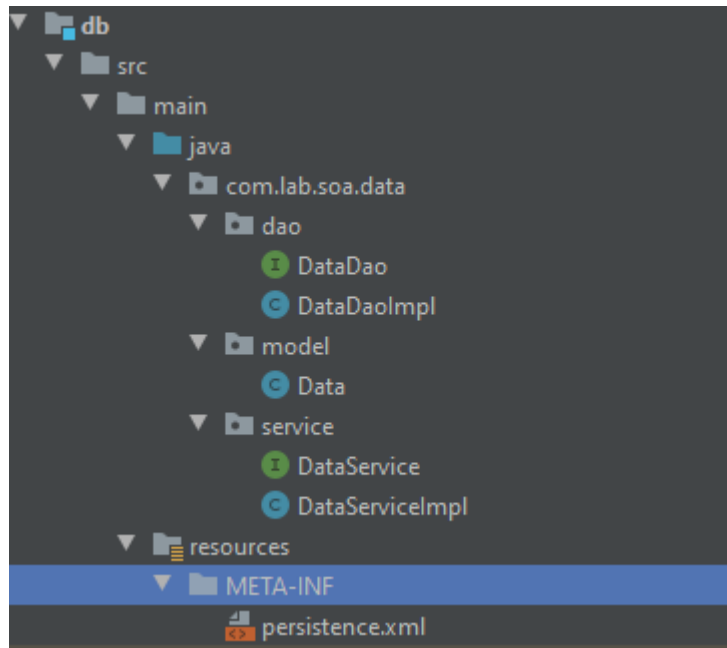
  <dependency>
    <groupId>org.jboss.as</groupId>
    <artifactId>jboss-as-security</artifactId>
    <version>7.2.0.Final</version>
    <exclusions>
      <exclusion>
        <artifactId>tools</artifactId>
        <groupId>com.sun</groupId>
      </exclusion>
    </exclusions>
  </dependency>

  <dependency>
    <groupId>com.lab.soa</groupId>
    <artifactId>core</artifactId>
    <version>1.0-SNAPSHOT</version>
    <scope>compile</scope>
  </dependency>
</dependencies>
```

12. Tworzymy:

webservice soapowy - DataWebService

webapp z konfiguracją web.xml, który powinien zostać automatycznie wygenerowany w punkcie 3.



13. W bezstanowy bean dodajemy anotacje `@WebService`, która tworzy service soapowy, `@WebContext`, która ustawia adres serwisu oraz typu autoryzacji oraz `@SecurityDomain`, która dodaje securitydomain zdefiniowany w `standalone.xml` i pozwala na dostęp do metod tylko użytkownikom zdefiniowanym w `application-users.properties`. Bean posiada dwie metody `push`, która dodaje dane do bazy oraz wywołuje czasochłonną asynchroniczną metodę z `actionService` oraz `check` która sprawdza czy metoda `action` się zakończyła i został dodany `update` do bazy. Metoda `push` jest `@PermitAll`, więc mogą jej użyć użytkownicy z dowolną rolą, a metoda `check` jest dostępna tylko dla użytkowników z rolą `soa2020` dodaną w `application-roles.properties`.

```
@Stateless
@WebService
@SecurityDomain("SOA_2020-egzamin-security-domain")
@WebContext(contextRoot = "/soap", urlPattern = "/dataWebService",
            authMethod = "BASIC", transportGuarantee = "NONE")
public class DataWebService {

    private static final Logger LOG = Logger.getLogger(DataWebService.class);

    @Inject
    DataService dataService;

    @Inject
    ActionService actionService;

    @WebMethod(action = "push")
    @PermitAll
    public long push(@WebParam(name = "data") String data) throws Exception {

        long id = dataService.create(data);

        LOG.info("Starting asynchronous action for data: " + data);
        actionService.action(id);

        return id;
    }

    @WebMethod(action = "check")
    @RolesAllowed("soa2020")
    public boolean check(@WebParam(name = "id") long id) {

        LOG.info("Checking if action is completed for id: " + id);
        return dataService.checkIsCompleted(id);
    }
}
```

14. Następnie w module core będzie odpowiedzialnym za logikę aplikacji jms oraz action symulującą czasochłonną metodę dodaje pom.xml z dependencjami JMS, logging do loggera oraz moduł db. Packaging ustawiamy na jar.

```
xs1:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>

<parent>
  <groupId>com.lab.soa</groupId>
  <artifactId>Piotr-Seemann-soa2020-egzamin</artifactId>
  <version>1.0-SNAPSHOT</version>
</parent>

<artifactId>core</artifactId>

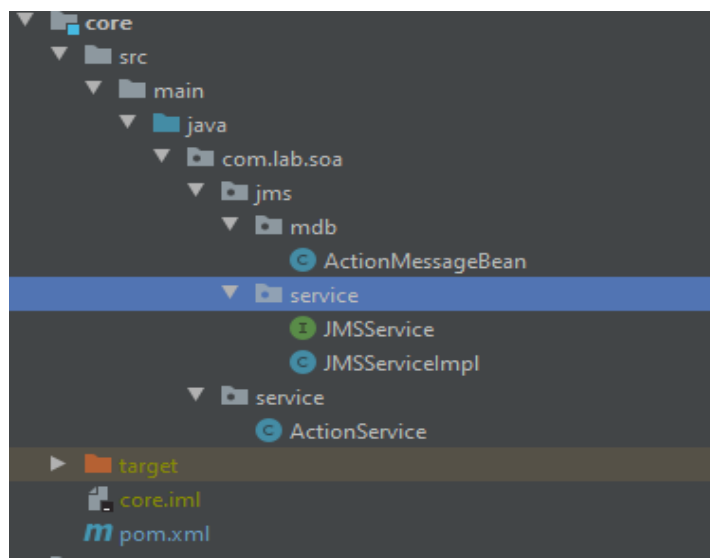
<packaging>jar</packaging>

<dependencies>
  <dependency>
    <groupId>javax.jms</groupId>
    <artifactId>javax.jms-api</artifactId>
    <version>2.0.1</version>
  </dependency>

  <dependency>
    <groupId>org.jboss.logging</groupId>
    <artifactId>jboss-logging</artifactId>
    <version>3.3.2.Final</version>
  </dependency>

  <dependency>
    <groupId>com.lab.soa</groupId>
    <artifactId>db</artifactId>
    <version>1.0-SNAPSHOT</version>
    <scope>compile</scope>
  </dependency>
</dependencies>
<build>
  <finalName>${parent.artifactId}-${project.artifactId}</finalName>
</build>
</project>
```

15. Tworzymy:
- symulujący czasochłonna akcję – ActionService
 - wysyłający akcję do kolejki JMS dostępny przez interfacce JMSService
 - listener oczekujący na komunikaty z kolei JM ActionMessageBean



16. W bezstanowym beanie JMSService mamy tylko jedną metodę sendMessage, która wysyła wiadomość do kolejki JMS. Dodajemy jako @Resource queue, sessionContext oraz ConnectionFactory zdefiniowane w stadalone.xml.

```
@MessageDriven(activationConfig = {
    @ActivationConfigProperty(propertyName = "acknowledgeMode", propertyValue = "Auto-acknowledge"),
    @ActivationConfigProperty(propertyName = "destinationType", propertyValue = "javax.jms.Queue"),
    @ActivationConfigProperty(propertyName = "destination", propertyValue = "java:/jms/queue/SOA_2020-egzamin")
})

public class ActionMessageBean implements MessageListener {

    private static final Logger LOG = Logger.getLogger(ActionMessageBean.class);

    @Inject
    DataService dataService;

    public void onMessage(Message msg) {
        TextMessage txtMsg;
        try {
            if (msg instanceof TextMessage) {
                txtMsg = (TextMessage) msg;
                String txt = txtMsg.getText();
                dataService.update(Long.parseLong(txt));
            }
        } catch (Exception e) {
            LOG.error((e.getMessage()));
        }
    }
}
```

17. W MessageDrivenBean nasłuchujemy na komunikaty w kolejce JMS i po przyjęciu komunikatu wykonujemy update w bazie przez metodę update interfejsu DataService.

```
@MessageDriven(activationConfig = {
    @ActivationConfigProperty(propertyName = "acknowledgeMode", propertyValue = "Auto-acknowledge"),
    @ActivationConfigProperty(propertyName = "destinationType", propertyValue = "javax.jms.Queue"),
    @ActivationConfigProperty(propertyName = "destination", propertyValue = "java:/jms/queue/SOA_2020-egzamin")
})

public class ActionMessageBean implements MessageListener {

    private static final Logger LOG = Logger.getLogger(ActionMessageBean.class);

    @Inject
    DataService dataService;

    public void onMessage(Message msg) {
        TextMessage txtMsg;
        try {
            if (msg instanceof TextMessage) {
                txtMsg = (TextMessage) msg;
                String txt = txtMsg.getText();
                dataService.update(Long.parseLong(txt));
            }
        } catch (Exception e) {
            LOG.error((e.getMessage()));
        }
    }
}
```


18. Bezstanowy bean `actionService` ma metodę `action`, która symuluje czasochłonną akcję, która jest wykonywana asynchronicznie anotacja `@Asynchronous`, a następnie wysyła komunikat do kolejki przez wstrzyknięty interfejs `JMSService`.

```
@Stateless
public class ActionService {

    private static final Logger LOG = Logger.getLogger(ActionService.class);

    @Inject
    JMSService JMSService;

    @Asynchronous
    public void action(long id) {
        int actionDuration = 5000 + (int) (45000 * Math.random());
        try {
            Thread.sleep(actionDuration);
        } catch (Exception e) { LOG.error(e.getMessage()); }

        JMSService.sendMessage(String.valueOf(id));
    }
}
```

19. Ostatni modułem jest moduł `ear`, gdzie pakujemy całą aplikację. Dodajemy wszystkie inne moduły jako `dependencies`/

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.lab.soa</groupId>
    <artifactId>ear</artifactId>
    <version>1.0-SNAPSHOT</version>

    <packaging>ear</packaging>

    <name>ear</name>

    <properties>
        <maven.compiler.source>11</maven.compiler.source>
        <maven.compiler.target>11</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <dependencies>
        <dependency>
            <groupId>${project.groupId}</groupId>
            <artifactId>core</artifactId>
            <version>${project.version}</version>
            <type>ejb</type>
        </dependency>
        <dependency>
            <groupId>${project.groupId}</groupId>
            <artifactId>db</artifactId>
            <version>${project.version}</version>
            <type>ejb</type>
        </dependency>
        <dependency>
            <groupId>${project.groupId}</groupId>
            <artifactId>soap</artifactId>
            <version>${project.version}</version>
            <type>war</type>
        </dependency>
    </dependencies>
</project>
```

20. Następni pluginem Mavena maven-ear-plugin dodajemy moduły core i db jako ejbModule oraz soap jako webModule. Dodajemy też wildfly-maven-plugin, który odpowiada za deployment naszej aplikacji.

```
build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-ear-plugin</artifactId>
      <version>3.0.2</version>
      <configuration>
        <defaultLibBundleDir>lib</defaultLibBundleDir>
        <skinnyWars>true</skinnyWars>
        <modules>
          <ejbModule>
            <groupId>${project.groupId}</groupId>
            <artifactId>core</artifactId>
          </ejbModule>
          <ejbModule>
            <groupId>${project.groupId}</groupId>
            <artifactId>db</artifactId>
          </ejbModule>
          <webModule>
            <groupId>${project.groupId}</groupId>
            <artifactId>soap</artifactId>
            <unpack>${unpack.wars}</unpack>
            <contextRoot>/soap</contextRoot>
          </webModule>
        </modules>
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.wildfly.plugins</groupId>
      <artifactId>wildfly-maven-plugin</artifactId>
      <version>2.0.2.Final</version>
      <configuration>
        <skip>>false</skip>
      </configuration>
    </plugin>
  </plugins>
</build>
```