JUST DO IT
HÖHERE ABTEILUNG FÜR INFORMATIONSTECHNOLOGIE

tgm
Die Schule der Technik

Dezsys 07

SOA and Restful Webservice

Systemtechnik Labor
5BHITT 2015/16

Selina Brinnich / Niklas Hohenwarter

Version 1.0

Note:

Betreuer: Thomas Micheler

Begonnen am 11. Dezember 2015

Beendet am 02. Jänner 2016

Inhaltsverzeichnis

1 Einführung

Das neu eröffnete Unternehmen iKnow Systems ist spezialisiert auf Knowledgemanagement und bietet seinen Kunden die Möglichkeiten Daten und Informationen jeglicher Art in eine Wissensbasis einzupflegen und anschließend in der zentralen Wissensbasis nach Informationen zu suchen (ähnlich wikipedia).

Folgendes ist im Rahmen der Aufgabenstellung verlangt:

- Entwerfen Sie ein Datenmodell, um die Eintraege der Wissensbasis zu speichern und um ein optimitiertes Suchen von Eintraegen zu gewaehrleisten. [2Pkt]
- Entwickeln Sie mittels RESTful Webservices eine Schnittstelle, um die Wissensbasis zu verwalten. Es muessen folgende Operationen angeboten werden:
  - Hinzufuegen eines neuen Eintrags
  - Aendern eines bestehenden Eintrags
  - Loeschen eines bestehenden Eintrags
- Alle Operationen muessen ein Ergebnis der Operation zurueckliefern. [3Pkt]
- Entwickeln Sie in Java ein SOA Webservice, dass die Funktionalitaet Suchen anbietet und das SOAP Protokoll einbindet. Erzeugen Sie fuer dieses Webservice auch eine WSDL-Datei. [3Pkt]
- Entwerfen Sie eine Weboberflaeche, um die RESTful Webservices zu verwenden. [3Pkt]
- Implementieren Sie einen einfachen Client mit einem User Interface (auch Commandline UI moeglich), der das SOA Webservice aufruft. [2Pkt]
- Dokumentieren Sie im weiteren Verlauf den Datentransfer mit SOAP. [1Pkt]
- Protokoll ist erforderlich! [2Pkt]

Zum Testen bereiten Sie eine Routine vor, um die Wissensbasis mit einer 1 Million Datensaetze zu fuellen. Die Datensaetze sollen mindestens eine Laenge beim Suchbegriff von 10 Zeichen und bei der Beschreibung von 100 Zeichen haben! Ist die Performance bei der Suche noch gegeben?

2 Ergebnisse

2.1 Testdaten / Datenmodell

Eine Herausforderung bei dieser Aufgabe ist es genug Testdaten zu generieren. Zu Beginn war geplant, ein Wörterbuch zu verwenden, jedoch wurde keine Wörterbuch Datenbank mit einer Million einträgen gefunden. Aus diesem Grund war es nötig Testdaten selbst zu generieren. Die generierten Daten haben eine Dateigröße von ca 180MB und werden daher nicht mit auf eLearning abgegeben.

Es wird von einer Personendatenbank ausgegangen. Diese Datenbank speichert die Email-Adresse und Biografie einer Person. Als DBMS wird MySQL verwendet.

```
CREATE TABLE search(
     email     VARCHAR(100),
     bio       VARCHAR(250),
     PRIMARY KEY(email)
);
```

Als Testdaten Generator wurde der databasetestdata[1] Generator verwendet. Hier wurden die generierten Daten als JSON exportiert. Da jedoch SQL als Datenformat benötigt wird, müssen die Daten konvertiert werden. Hierzu wurde ein einfacher Converter in Python geschrieben, welcher im Ordner */sqldata/converter* zu finden ist.

```
def json_to_sql(filein, fileout):
    # Load files
    input = open(filein, 'r')
    output = open(fileout, 'a')

    #load data to list
    data = json.load(input)

    #write data in sql format to file
    for d in data:
        d['bio'] = d['bio'].replace("\n", " ")
        output.write("INSERT INTO search VALUES('"+d['email']+"',
'"+d['bio']+"');\n")
```

2.2 Datenbankverbindung

Um auf die Datenbasis zuzugreifen, wird JDBC verwendet. Als Anhaltspunkt wurde das Tutorial von Tutorialspoint[2] verwendet. Die nötigen Dependencies wurden mit Maven hinzugefügt.

Der erste Schritt besteht darin, ein Interface mit allen Zugriffsmöglichkeiten für die gewünschte Tabelle zu erstellen. Dieses nennen wir SearchDAO.

```
public interface SearchDAO {
    /**
     * Initialisiert die Datenbankverbindung
     */
    public void setDataSource(DataSource ds);
    /**
     * Hiermit koennen neue Datensaetze eingefuegt werden
     */
    public boolean create(String email, String bio);
    /**
     * Abfrage eines einzelnen Datensatzes anhand der Email Adresse
     */
    public Search getPerson(String email);
    /**
     * Auflistung aller Datensaetze, die einem Muster enstprechen
     */
    public List<Search> listPersonsFilterBy(String s);
    /**
     * Auflistung aller Datensaetze, die einem Muster enstprechen (Seitenweise)
     */
    public List<Search> listPersonsFilterBy(String s, int min, int max);
    /**
     * Auflistung aller Datensaetze
     */
    public List<Search> listPersons();
    /**
     * Auflistung aller Datensaetze (Seitenweise)
     */
    public List<Search> listPersons(int min, int max);
    /**
     * Loechen eines Datensatzes
     */
    public boolean delete(String email);
    /**
     * Zum veraendern eines Datensatzes
     */
    public boolean update(String email, String bio);
}
```

1 DAO Modell

Danach muss ein Model für die Daten erstellt werden. Dieses muss alle getter & setter für die Attribute haben.

```java
public class Search {
    private String email;
    private String bio;

    /**
     * Getter fuer die E-Mail Adresse
     * @return E-Mail Adresse
     */
    public String getEmail() {
        return email;
    }

    /**
     * Getter fuer die Bio
     * @return Bio
     */
    public String getBio() {
        return bio;
    }
```

2 Datenmodell in Java

Im nächsten Schritt muss der SearchMapper erstellt werden. Dieser repräsentiert einen Datensatz in der Datenbank(row).

```java
/**
 * Repreasentiert einen Datensatz bzw. holt diesen aus der Datenbank
 * und speichert ihn als Search Objekt
 *
 * @author Niklas Hohenwarter
 * @version 2015-12-30
 * @see "http://www.tutorialspoint.com/spring/spring_jdbc_example.htm"
 */
public class SearchMapper implements RowMapper<Search> {
    /**
     * Mapt eine Row auf ein Search Objekt
     * @param rs Result Set der Quary
     * @param rowNum Nummer der Zeile in der DB
     * @return Search Objekt
     * @throws SQLException
     */
    public Search mapRow(ResultSet rs, int rowNum) throws SQLException {
        Search search = new Search();
        search.setEmail(rs.getString("email"));
        search.setBio(rs.getString("bio"));
        return search;
    }
}
```

3 *Java Mapper*

Die letzte Klasse welche erstellt werden muss ist die SearchJDBCTemplate Klasse. Diese bringt alle vorher erstellten Klassen zusammen und greift mittels SQL auf die Datenbank zu. Da die Klasse sehr lang ist folgt nur der Teil, in welchem ein neuer Datensatz erstellt wird.

```java
public class SearchJDBCTemplate implements SearchDAO {
    private DataSource dataSource;
    private JdbcTemplate jdbcTemplateObject;

    /**
     * Setzt die Datasource
     * @param dataSource DataSource
     */
    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
        this.jdbcTemplateObject = new JdbcTemplate(dataSource);
    }

    /**
     * Erstellt einen neuen Datensatz
     * @param email Email
     * @param bio Bio
     */
    public void create(String email, String bio) {
        String SQL = "insert into search (email, bio) values (?, ?)";

        jdbcTemplateObject.update(SQL, email, bio);
        System.out.println("Created Record Email = " + email + " Bio = " + bio);
    }
```

4 *Create für einen Datensatz im JDBCTemplate*

Nun muss die Beans.xml erstellt werden. Dort werden die DataSource und die Zugangsdaten für die Datenbank festgelegt.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd ">

    <!-- Datenbank -->
    <bean id="dataSource"
          class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
        <property name="url" value="jdbc:mysql://192.168.1.116:3306/search"/>
        <property name="username" value="search"/>
        <property name="password" value="search"/>
    </bean>

    <!-- Template Bean -->
    <bean id="studentJDBCTemplate"
          class="brinnichHohenwarter.db.SearchJDBCTemplate">
        <property name="dataSource"  ref="dataSource" />
    </bean>

</beans>
```

5 *Beans XML*

Nun muss die Bean noch initialisiert werden und dannach kann auf die Daten zugegriffen werden. Zum Testen der Verbindung wird in der Main auf die Daten zugegriffen.

```java
public static void main(String[] args) {
    SpringApplication.run(Application.class, args);

    ApplicationContext context =
            new ClassPathXmlApplicationContext("beans.xml");

    SearchJDBCTemplate searchJDBCTemplate =
            (SearchJDBCTemplate)context.getBean("studentJDBCTemplate");

    searchJDBCTemplate.create("info@niklashohenwarter.com", "Tech guy");
    System.out.println("----Listing Record with Email = info@niklashohenwarter.com -----" );
    Search search = searchJDBCTemplate.getPerson("info@niklashohenwarter.com");
    System.out.print("Email : " + search.getEmail() );
    System.out.print(", Bio : " + search.getBio() );
}
```

6 *Test des DB Zugriffs*

```
2015-12-30 13:39:22.201  INFO 9012 --- [           main] o.s.b.f.xml.XmlBeanDefinitior
2015-12-30 13:39:22.317  INFO 9012 --- [           main] o.s.j.d.DriverManagerDataSour
Created Record Email = info@niklashohenwarter.com Bio = Tech guy
----Listing Record with Email = info@niklashohenwarter.com -----
Email : info@niklashohenwarter.com, Bio : Tech guy
```

7 DB Zugriff in der Spring Konsole

2.3 REST API

Für die Erstellung eines Restful Webservices in Spring muss ein RestController erstellt werden. Dazu wird eine Java-Klasse mit der Annotation *@RestController* versehen.

In diesem RestController muss nun eine Datenbankverbindung hergestellt werden. Das wird im Konstruktor gemacht:

```
private SearchJDBCTemplate searchJDBCTemplate;

public SearchController(){
    ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");

    this.searchJDBCTemplate = (SearchJDBCTemplate)context.getBean("studentJDBCTemplate");
}
```

8 Initialisieren der Datenbankverbindung (Rest)

Anschließend können für Java-Methoden URLs definiert werden, über die diese Methoden aufgerufen werden können:

```
@RequestMapping(value="/api/add/'{email}'/'{bio}'", method= RequestMethod.GET)
public String add(@PathVariable String email, @PathVariable String bio){
    boolean res = this.searchJDBCTemplate.create(email, bio);
    if(res){
        return "Person successfully inserted!";
    }else{
        return "An error occured! Person could not be inserted!";
    }
}
```

9 Methodendefinition Restful Webservice

Mit der Annotation *@RequestMapping* kann eine bestimmte URL und eine Methode (GET, POST, PUT, ...) definiert werden. In der URL können zudem beliebige Parameter vorkommen (hier: email & bio). Diese Parameter werden der Methode über die Annotation @PathVariable übergeben. Zurückgegeben wird hier ein String, der nach Aufrufen dieser URL im Browser als Text angezeigt wird.

2.4 REST GUI

Um das Restful Webservice über eine GUI verwenden zu können, müssen noch ein paar zusätzliche Methoden im RestController geschrieben werden.

```
@RequestMapping(value="/api/add", method= RequestMethod.POST, produces = "application/json")
public String addGui(@RequestBody Search person){
    return "{\"success\":"+this.searchJDBCTemplate.create(person.getEmail(), person.getBio())+"}";
}
```

10 Verwendung von HTTP POST (Rest)

Hier ist die URL etwas anders als beim Beispiel im vorherigen Kapitel. Das liegt daran, dass die Parameter nun nicht mehr als GET-Parameter übergeben werden, sondern nun die Methode POST verwendet wird. Zudem wird hier mit *produces = ...* noch definiert, dass kein normaler Text, sondern ein JSON-String zurückgegeben wird. Die Methode bekommt als PUT-Parameter ebenfalls ein JSON-Objekt. Dieses enthält die Parameter *email* und *bio*. Mit der Annotation *@RequestBody* werden diese Parameter intern in ein *Search*-Objekt umgewandelt (da dieses ebenfalls nur diese beiden Parameter besitzt).

Anschließend muss neben dem RestController noch ein Controller definiert werden, der für die View zuständig ist. In diesem wird genau wie beim RestController im Konstruktor die Datenbankverbindung hergestellt.

Nun können Methoden definiert werden, die dann ein HTML-Template zurückgeben:

```
@RequestMapping(value = "/add", method= RequestMethod.GET, produces = "text/html")
public String add() { return "add"; }
```

11 Rückgabe eines HTML-Templates

Das HTML-File, das hier zurückgegeben wird, heißt „*add.html*". Dieses befindet sich im Ordner */src/main/resources/templates*, wobei der Ordner *resources* auch als Resource-Ordner markiert werden muss. Alle HTML-Templates müssen in diesen Ordner, damit der Controller darauf zugreifen kann. Alle JavaScript-Files, die für die Oberfläche benötigt werden, kommen in den Ordner */src/main/resources/static*.

Im HTML-File muss innerhalb des HTML-Tags folgender Namespace definiert werden:

```
<html xmlns:th="http://www.thymeleaf.org">
```

Im Head kann auf die vorhin erwähnten Javascript-Files folgendermaßen zugegriffen werden:

```
<script type="text/javascript" src="/javascript.js"></script>
```

Um dem Template Variablen übergeben zu können, muss im Controller folgendermaßen vorgegangen werden:

```
@RequestMapping(value = "/edit/'{email}'", method= RequestMethod.GET, produces = "text/html")
public String edit(@PathVariable String email, Model model){
    Search person = this.searchJDBCTemplate.getPerson(email);
    model.addAttribute("person", person);
    return "edit";
}
```

12 Übergeben von Variablen an HTML-Templates

Hier wird der Java-Methode zusätzlich zu der PathVariable noch ein Model übergeben. Mithilfe von *model.addAttribute(name, object)* kann dem Model eine Variable hinzugefügt werden. In diesem Fall ist das ein Search-Objekt, es kann aber auch bspw. eine Liste zurückgegeben werden. Im HTML-File kann anschließend auf diese Variable folgendermaßen zugegriffen werden:

```
<span th:text="${person.email}"></span>
```

Über den vorher definierten Namespace wird Thymeleaf eingebunden, was dazu verwendet werden kann solche Variablen zu parsen. Durch vorher erwähnte Listen in Variablen kann jedes Element einzeln durchgegangen werden:

```
<div th:each="person : ${persons}">
```

Anschließend kann wieder mit *${person.email}* auf das einzelne Objekt zugegriffen werden.

Innerhalb des Controllers können außerdem Redirects angegeben werden, um beispielsweise von der Standard-URL „/" auf eine andere URL weiterzuleiten:

```
@RequestMapping(value = "/", method= RequestMethod.GET, produces = "text/html")
public String index() { return "redirect:page=1"; }
```

13 Redirect bei Restful Webservice

Dazu wird hier einfach ein String im Format "redirect:<url>" zurückgegeben.

2.5 SOA Webservice

Bei der Realisierung wird auf das Example von Spring zurückgegriffen[3]. Im ersten Schritt werden die Dependencies in das bereits vorhandene pom.xml hinzugefügt.

Der nächste Schritt ist es ein XSD File zu definieren, welches die aufrufbaren Methoden und deren Parameter definiert.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:tns="http://localhost/soa/search"
           targetNamespace="http://localhost/soa/search" elementFormDefault="qualified">

    <xs:element name="getPersonsRequest">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="name" type="xs:string"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:element name="getPersonResponse">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="search" type="tns:search"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:complexType name="search">
        <xs:sequence>
            <xs:element name="email" type="xs:string"/>
            <xs:element name="bio" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```

14 Service XSD

Dannach muss in Maven ein Plugin definiert werden, welches aus dem XSD automatisch Java Code erzeugt.

```
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>jaxb2-maven-plugin</artifactId>
    <version>1.6</version>
    <executions>
        <execution>
            <id>xjc</id>
            <goals>
                <goal>xjc</goal>
            </goals>
        </execution>
    </executions>
    <configuration>
        <schemaDirectory>${project.basedir}/src/main/resources/</schemaDirectory>
        <outputDirectory>${project.basedir}/src/main/java/brinnichHohenwarter/soa</outputDirectory>
        <clearOutputDir>false</clearOutputDir>
    </configuration>
</plugin>
```

15 *JAXB Konfiguration*

Um die Files zu generieren muss Maven explizit gesagt werden dies zu tun. Dannach sind die Files im definierten Package. Dannach muss man im Java Code noch die Package deklaration ändern, da diese falsch ist.

Im nächsten Schritt muss man ein sogenanntes Repository erstellen. Über dieses wird auf die Daten zugegriffen.

```java
@Component
public class SearchRepository {

    private SearchJDBCTemplate searchJDBCTemplate;

    @PostConstruct
    public void initData() {
        ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");

        this.searchJDBCTemplate = (SearchJDBCTemplate)context.getBean("studentJDBCTemplate");
    }

    public List<Search> findPerson(String email) {

        List<brinnichHohenwarter.db.Search> dbresult = searchJDBCTemplate.listPersonsFilterBy(email);
        ArrayList<Search> result = new ArrayList<Search>();

        for(int i = 0; i < dbresult.size(); i++){
            Search entry = new Search();
            entry.setEmail(dbresult.get(i).getEmail());
            entry.setBio(dbresult.get(i).getBio());
            result.add(entry);
        }

        return result;
    }
}
```

16 *SOA Repositroy*

Dannach muss SOA noch als WebService definiert und konfiguriert werden.

```java
@EnableWs
@Configuration
public class WebServiceConfig extends WsConfigurerAdapter {
    @Bean
    public ServletRegistrationBean messageDispatcherServlet(ApplicationContext applicationContext) {
        MessageDispatcherServlet servlet = new MessageDispatcherServlet();
        servlet.setApplicationContext(applicationContext);
        servlet.setTransformWsdlLocations(true);
        return new ServletRegistrationBean(servlet, "/soa/*");
    }

    @Bean(name = "search")
    public DefaultWsdl11Definition defaultWsdl11Definition(XsdSchema searchschema) {
        DefaultWsdl11Definition wsdl11Definition = new DefaultWsdl11Definition();
        wsdl11Definition.setPortTypeName("SearchPort");
        wsdl11Definition.setLocationUri("/soa");
        wsdl11Definition.setTargetNamespace("http://localhost/soa/search");
        wsdl11Definition.setSchema(searchschema);
        return wsdl11Definition;
    }

    @Bean
    public XsdSchema searchShema() {
        return new SimpleXsdSchema(new ClassPathResource("search.xsd"));
    }
}
```

17 *Webservice Config*

Dannach sollte alles funktionieren. Dies kann mit einer SOAP Anfrage, welche mittels curl übermittelt wird getestet werden.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:gs="http://localhost/soa/search">
    <soapenv:Header/>
    <soapenv:Body>
        <gs:getPersonsRequest>
            <gs:email>info@niklashohenwarter.com</gs:email>
        </gs:getPersonsRequest>
    </soapenv:Body>
</soapenv:Envelope>
```

```
niklas@apico ~/Desktop % curl --header "content-type: text/xml" -d @req.xml http://localhost:8080/so
a
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"><SOAP-ENV:Header/><SOA
P-ENV:Body><ns2:getPersonResponse xmlns:ns2="http://localhost/soa/search"><ns2:search><ns2:email>inf
o@niklashohenwarter.com</ns2:email><ns2:bio>Tech guy</ns2:bio></ns2:search></ns2:getPersonResponse><
/SOAP-ENV:Body></SOAP-ENV:Envelope>%
```

18 *SOA Suchantwort*

Die WDSL Datei wurde automatisch generiert und kann unter der URL http://localhost:8080/soa/search.wsdl abgerufen werden.

```
−<wsdl:definitions targetNamespace="http://localhost/soa/search">
  −<wsdl:types>
    −<xs:schema elementFormDefault="qualified" targetNamespace="http://localhost/soa/search">
      −<xs:element name="getPersonsRequest">
        −<xs:complexType>
          −<xs:sequence>
              <xs:element name="email" type="xs:string"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      −<xs:element name="getPersonResponse">
        −<xs:complexType>
          −<xs:sequence>
              <xs:element maxOccurs="unbounded" name="search" type="tns:search"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      −<xs:complexType name="search">
        −<xs:sequence>
            <xs:element name="email" type="xs:string"/>
            <xs:element name="bio" type="xs:string"/>
          </xs:sequence>
        </xs:complexType>
      </xs:schema>
    </wsdl:types>
  −<wsdl:message name="getPersonsRequest">
      <wsdl:part element="tns:getPersonsRequest" name="getPersonsRequest"> </wsdl:part>
    </wsdl:message>
  −<wsdl:message name="getPersonResponse">
      <wsdl:part element="tns:getPersonResponse" name="getPersonResponse"> </wsdl:part>
    </wsdl:message>
  −<wsdl:portType name="SearchPort">
    −<wsdl:operation name="getPersons">
        <wsdl:input message="tns:getPersonsRequest" name="getPersonsRequest"> </wsdl:input>
      </wsdl:operation>
    −<wsdl:operation name="getPerson">
        <wsdl:output message="tns:getPersonResponse" name="getPersonResponse"> </wsdl:output>
      </wsdl:operation>
    </wsdl:portType>
  −<wsdl:binding name="SearchPortSoap11" type="tns:SearchPort">
      <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
```

19 *Auszug aus der WSDL*

2.6 SOA Client

Der Client basiert ebenfalls auf dem offizielen Spring Example[4]. Der Client muss allerdings ein eigenes Projekt sein, da ansonsten Fehler entstehen. Im pom.xml welches aus dem Tutorial übernommen wurde muss der inhalt des <configuration> tags geändert werden.

```
<configuration>
    <schemaLanguage>WSDL</schemaLanguage>
    <generatePackage>search.wsdl</generatePackage>
    <schemas>
        <schema>
            <url>http://localhost:8080/soa/search.wsdl</url>
        </schema>
    </schemas>
</configuration>
```

20 pom.xml SOA Client

Dannach kann Maven aufgrund des angegebenen WSDLs Code generieren. Nun wird ein Client benötigt, welcher auf das Service mittels SOAP anfrage zugreift.

```
public class SearchClient extends WebServiceGatewaySupport {

    private static final Logger log = LoggerFactory.getLogger(SearchClient.class);

    public GetPersonResponse getPersonByEmail(String email) {

        GetPersonsRequest request = new GetPersonsRequest();
        request.setEmail(email);

        log.info("Requesting bio for " + email);

        GetPersonResponse response = (GetPersonResponse) getWebServiceTemplate()
                .marshalSendAndReceive(
                        "http://localhost:8080/soa",
                        request,
                        new SoapActionCallback("http://localhost:8080/soa"));

        return response;
    }

    public void printResponse(GetPersonResponse response) {

        List<Search> personReturn = response.getSearch();

        if (personReturn != null) {
            for(int i = 0; i < personReturn.size(); i++){
                Search res = personReturn.get(i);
                log.info("Email: " + res.getEmail() + "\nBio: " + res.getBio());
            }
        } else {
            log.info("Nothing found");
        }
    }

}
```

21 *SOA Client*

Für den Client muss man nun noch den Marshaller konfigurieren.

```
@Configuration
public class SearchConfiguration {

    @Bean
    public Jaxb2Marshaller marshaller() {
        Jaxb2Marshaller marshaller = new Jaxb2Marshaller();
        marshaller.setContextPath("search.wsdl");
        return marshaller;
    }

    @Bean
    public SearchClient searchClient(Jaxb2Marshaller marshaller) {
        SearchClient client = new SearchClient();
        client.setDefaultUri("http://localhost/soa");
        client.setMarshaller(marshaller);
        client.setUnmarshaller(marshaller);
        return client;
    }

}
```

22 *Marshaller Config*

Dannach muss man noch eine Main für den Client schreiben.

```
@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

    @Bean
    CommandLineRunner lookup(SearchClient searchClient) {
        return args -> {
            String email = "";

            if (args.length > 0) {
                email = args[0];
            }
            GetPersonResponse response = searchClient.getPersonByEmail(email);
            searchClient.printResponse(response);
        };
    }

}
```

23 *Starter für den Client*

Beim Start muss im args[] Array der Suchbegriff bzw. die E-Mail Adresse nach welcher gesucht werden soll übergeben werden.

```
2016-01-02 16:28:37.406  INFO 11670 --- [           main] b.soaclient.SearchClient                 : Requesting bio for info@niklashohenwarter.com
2016-01-02 16:28:37.690  INFO 11670 --- [           main] b.soaclient.SearchClient                 : Email: info@niklashohenwarter.com
Bio: Tech guy
2016-01-02 16:28:37.692  INFO 11670 --- [           main] b.soaclient.Application                  : Started Application in 2.484 seconds (JVM running for 2.946)
```

24 *Client Test*

2.7 SOA Datentransfer

Um den Datentransfer zu dokumentieren wurde Wireshark verwendet. Als Beispiel greift der Client auf den SOA Webservice zu und sucht nach der E-Mail Adresse info@niklashohenwarter.com

No.	Time	Source	Destination	Protocol	Length	Info
18	3.068949099	127.0.0.1	127.0.0.1	HTTP/XML	349	POST /soa HTTP/1.1
22	3.416634365	127.0.0.1	127.0.0.1	HTTP/XML	401	HTTP/1.1 200 OK

25 Kommunikation zwischen Client und Service

```
▸ Frame 18: 349 bytes on wire (2792 bits), 349 bytes captured (2792 bits) on interface 0
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▸ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▸ Transmission Control Protocol, Src Port: 50066 (50066), Dst Port: 8080 (8080), Seq: 335, Ack: 1, Len: 283
▸ [2 Reassembled TCP Segments (617 bytes): #16(334), #18(283)]
▸ Hypertext Transfer Protocol
▾ eXtensible Markup Language
  ▾ <SOAP-ENV:Envelope
      xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
      <SOAP-ENV:Header/>
    ▾ <SOAP-ENV:Body>
      ▾ <ns2:getPersonsRequest
          xmlns:ns2="http://localhost/soa/search">
        ▾ <ns2:email>
            info@niklashohenwarter.com
            </ns2:email>
          </ns2:getPersonsRequest>
        </SOAP-ENV:Body>
      </SOAP-ENV:Envelope>
```

26 SOAP Request

```
▸ Frame 22: 401 bytes on wire (3208 bits), 401 bytes captured (3208 bits) on interface 0
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▸ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▸ Transmission Control Protocol, Src Port: 8080 (8080), Dst Port: 50066 (50066), Seq: 231, Ack: 618, Len: 335
▸ [2 Reassembled TCP Segments (565 bytes): #20(230), #22(335)]
▸ Hypertext Transfer Protocol
▾ eXtensible Markup Language
  ▾ <SOAP-ENV:Envelope
      xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
      <SOAP-ENV:Header/>
    ▾ <SOAP-ENV:Body>
      ▾ <ns2:getPersonResponse
          xmlns:ns2="http://localhost/soa/search">
        ▾ <ns2:search>
          ▾ <ns2:email>
              info@niklashohenwarter.com
              </ns2:email>
          ▾ <ns2:bio>
              Tech guy
              </ns2:bio>
            </ns2:search>
          </ns2:getPersonResponse>
        </SOAP-ENV:Body>
      </SOAP-ENV:Envelope>
```

27 SOAP Response

3 Quellen

[1] DatabaseTestData, Generate test data for your database, http://www.databasetestdata.com/, zuletzt besucht: 13.12.15

[2] Tutorialspoint, Spring JDBC Example, http://www.tutorialspoint.com/spring/spring_jdbc_example.htm, zuletzt besucht: 30.12.15

[3] Spring, Producing a SOAP web service, http://spring.io/guides/gs/producing-web-service/, zuletzt besucht: 01.01.16

[4] Spring, Consuming a SOAP Webservice, http://spring.io/guides/gs/consuming-web-service/, zuletzt besucht: 02.01.16

4 Abbildungsverzeichnis

5 Zeitaufzeichnung

Datum	Dauer	Name	Beschreibung
11.12.15	2h	Brinnich	Einlesen in Spring
11.12.15	2h	Hohenwarter	Testdaten generieren
13.12.15	2h	Hohenwarter	JSON to SQL Protokoll & Repo Draft
18.12.15	2h	Brinnich	REST Web GUI Draft
18.12.15	2h	Hohenwarter	REST API Draft / Data Source
30.12.15	4h	Brinnich	REST Web GUI REST API Anpassungen
30.12.15	2h	Hohenwarter	Datenbankverbindung
01.01.16	1h	Hohenwarter	SOA Service
01.01.16	1h	Brinnich	REST finalisierung
02.01.16	2h	Brinnich	REST Dokumentation Bugfixing
02.01.16	2h	Hohenwarter	SOA Service SOA Client