JUST DO IT
HÖHERE ABTEILUNG FÜR INFORMATIONSTECHNOLOGIE

tgm
Die Schule der Technik

Laborprotokoll

SOA and RESTful Webservice

Systemtechnik Labor
5BHITT 2015/16

Stefan Erceg & Martin Kritzl

Version 1.0

Note:

Betreuer: Prof. Borko & Micheler

Begonnen am 4. Dezember 2015

Beendet am 7. Januar 2016

Inhaltsverzeichnis

1 Aufgabenstellung

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 Einträge der Wissensbasis zu speichern und um ein optimiertes Suchen von Einträgen zu gewährleisten. [2Pkt]

- Entwickeln Sie mittels RESTful Webservices eine Schnittstelle, um die Wissensbasis zu verwalten. Es müssen folgende Operationen angeboten werden:
  - **Hinzufügen** eines neuen Eintrags
  - **Ändern** eines bestehenden Eintrags
  - **Löschen** eines bestehenden Eintrags

  Alle Operationen müssen ein Ergebnis der Operation zurückliefen. [3Pkt]

- Entwickeln Sie in Java ein SOA Webservice, dass die Funktionalität Suchen anbietet und das SOAP Protokoll einbindet. Erzeugen Sie für dieses Webservice auch eine WSDL-Datei. [3Pkt]

- Entwerfen Sie eine Weboberfläche, um die RESTful Webservices zu verwenden. [3Pkt]

- Implementieren Sie einen einfachen Client mit einem User Interface (auch Commandline UI möglich), der das SOA Webserviceaufruft. [2Pkt]

- Dokumentieren Sie im weiteren Verlauf den Datentransfer mit SOAP. [1Pkt]

- Protokoll ist erforderlich! [2Pkt]

Info:

Gruppengröße: 2 Mitglieder
Punkte: 16

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

Links:

JEE Webservices:
http://docs.oracle.com/javaee/6/tutorial/doc/gijti.html

Apache Web Services Project:
http://ws.apache.org/

Apache Axis/Axis2:
http://axis.apache.org
http://axis.apache.org/axis2/java/core/

IBM Article: Java Web services - JAXB and JAX-WS in Axis2:
http://www.ibm.com/developerworks/java/library/j-jws8/index.html

2 Ergebnisse

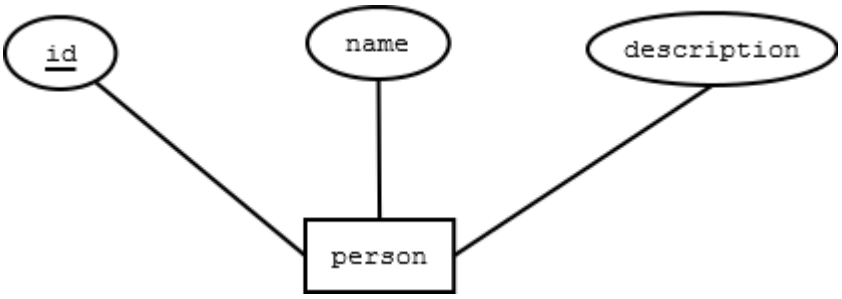
2.1 Auswahl eines Datenbankmanagementsystems

Zu Beginn wurde entschieden, ob eine relationale oder eine NoSQL-Datenbank verwendet wird. Da bei dieser Übung nur eine Tabelle existiert und somit keine Relationen vorhanden sind, wurde eine NoSQL-Datenbank ausgewählt. Es werden ebenfalls keine Aggregationen verwendet und somit wird die Performance gesteigert.

Als konkretes NoSQL-DBMS haben wir MongoDB ausgewählt, da es dazu genügend Dokumentation gibt und wir ebenfalls Erfahrungen mit diesem System machen wollten.

2.2 Erstellung des Datenbankentwurfs

Das ER-Diagramm wurde mit dem Programm „Dia“ erstellt.



Unsere Datenbank besteht aus einer einzigen Tabelle, welche eine Person darstellt. Der Primary Key ist eine ID, weiters werden zu jeder Person der Name und eine Beschreibung gespeichert.

2.3 Datengenerierung

Zum Generieren der Datenbankeinträge haben wir „generatedata.com“ verwendet. Das ZIP-File dazu kann man sich auf folgender Seite herunterladen: [1].

Da auf unseren Laptops bereits ein Apache Server lief, mussten wir den entpackten Ordner „generatedata“ lediglich in den Ordner „Apache Software Foundation/Apache2.2/htdocs“ hinzufügen, um über den Browser die Seite http://localhost/generatedata-3.1.4/index.php aufrufen und somit die Testdaten generieren zu können.

Da wir den Personen vor dem Generieren der Daten immer 2 Vornamen geben, besitzt der Name jeder Person mindestens 10 Zeichen (so wie es gewünscht war). Dem Attribut „description“ wurden zu jedem Eintrag Lorem Ipsum-Texte mit mindestens 100 Zeichen zugewiesen. Dazu wurde bei „Data Type“ „Fixed Number of Words“ ausgewählt und bei Options „Generate #100 words“ eingestellt.

2.4 Implementierung von REST

2.4.1 Allgemeines

Wie wir bereits in dem Referat von unseren Mitschülern Stefan Geyer & Mathias Ritter gehört haben, stellt REST (REpresentational State Transfer) ein Architekturmodell dar, welches im World Wide Web genutzt wird. Mittels REST können die HTTP-Methoden GET, PUT, POST und DELETE realisiert werden und dies haben wir auch implementiert.

Vor jeder Methode ist die Angabe der Annotation „RequestMapping“ wichtig, welche folgende Eigenschaften besitzt:

- `value = /persons` bzw. `/persons/{id}`
- `method =` z.B.: `RequestMethod.GET`
- `produces = application/json`

2.4.2 Umsetzung der HTTP-Methoden

- ***GET***

  Um nach bestimmten Einträgen suchen zu können, ist es notwendig eine Methode zu erstellen, welche als Rückgabewert ein „ResponseEntity"-Objekt besitzt. In unserem Fall wird nach dem Namen gefiltert und dieser wird als Parameter der Methode angegeben:

```
@RequestParam(value = "name", defaultValue = "") String name
```

  Folgendermaßen holen wir uns dann in der Methode das "ResponseEntity"-Objekt:

```
if (name.length() == 0)
    return new ResponseEntity<>(service.findTop50(), HttpStatus.OK);
else
    return new ResponseEntity<>
      (service.findTop50ByNameContainingIgnoreCase(name), HttpStatus.OK);
```

- ***PUT***

  Um einen bestimmten Eintrag zu ändern, wird nach der ID der Person, deren Eigenschaften geändert werden sollen, gefiltert und diese wird als Parameter der Methode angegeben:

```
@PathVariable String id, @RequestBody PersonRecord newDataRecord
```

  Der Name und die Beschreibung der Person werden dann folgendermaßen überschrieben:

```
PersonRecord dataRecord = service.findById(id);
dataRecord.setName(newDataRecord.getName());
dataRecord.setDescription(newDataRecord.getDescription());

return new ResponseEntity<>(service.update(dataRecord), HttpStatus.OK);
```

- ***POST***

  Um eine bestimmte Person hinzuzufügen, ist folgender Parameter bei der Methode anzugeben:

  ```
  @RequestBody PersonRecord personRecord
  ```

  Damit die Person dann tatsächlich zur Personenliste hinzugefügt wird, sind in der Methode lediglich folgende 2 Codezeilen notwendig:

  ```
  service.create(personRecord);
  return new ResponseEntity<>(personRecord, HttpStatus.CREATED);
  ```

- ***DELETE***

  Um einen bestimmten Personeneintrag zu löschen, muss nach der ID der Person, welche gelöscht werden soll, gefiltert werden und diese wird als Parameter der Methode angegeben:

  ```
  @PathVariable String id
  ```

  Damit die Person dann tatsächlich von der Personenliste entfernt wird, sind in der Methode lediglich folgende 2 Codezeilen notwendig:

  ```
  service.delete(id);
  return new ResponseEntity<>(HttpStatus.OK);
  ```

2.5 Implementierung der Web-Oberfläche

2.5.1 Listendarstellung der eingetragenen Personen

Damit alle Personeneinträge auf der Hauptseite dargestellt werden, wird als Parameter der Methode ein Model angegeben, auf welches später im HTML-File zugegriffen wird:

```
@RequestParam(value = "name", defaultValue = "") String name, Model model
```

Optional könnte man durch die Angabe des Strings “name” im Parameter auch nach dem Namen von Personen filtern und nur die Personen, die diesen Namen besitzen, würden auf der Hauptseite dargestellt werden:

```
List<PersonRecord> personRecords =
restController.findDataRecordsByName(name).getBody();
```

Um die Personeneinträge zum Model hinzuzufügen, ist folgender Befehl notwendig:

```
model.addAttribute("personRecords", personRecords);
```

Zum Schluss wird in der Methode der Name des HTML-Files, die für die Listendarstellung aller Personen zuständig ist, zurückgegeben:

```
return "showPersons";
```

Folgendermaßen schreiben wir dann alle Personeneinträge in das HTML-File „showPersons.html“ und zeigen somit den Namen und die Beschreibung jeder Person in einer Tabelle dar:

```
<th:block th:each="personRecord : ${personRecords}">
    <tr th:id="${personRecord.id}">
        <td th:text="${personRecord.name}"></td>
        <td th:text="${personRecord.description}"></td>
    </tr>
</th:block>
```

2.5.2 Hinzufügen einer Person

Falls man auf der Hauptseite auf den Button „neue Person hinzufügen“ klickt, wird eine Unterseite geöffnet, mittels welcher es möglich ist, den Namen und die Beschreibung der neuen Person einzugeben und diese zu speichern.

Damit das HTML-File „addPerson.html“ unter der URL „/persons/add“ erreichbar ist, ist folgende Methode notwendig:

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

Nachdem der User auf der Seite zum Hinzufügen einer Person den Button “hinzufügen” geklickt hat, holen wir uns in JavaScript zuerst den vom User geschriebenen Namen und die Beschreibung und speichern dann diese Values in eine Variable ab:

```
var data = {
    name: $("#name").val(),
    description: $("#description").val()
};
```

Mittels AJAX werden dann die Daten übernommen, in ein JSON-Format umgewandelt und allgemein in der Liste aller Personeneinträge abgelegt:

```
$.ajax({
    url: "/persons",
    data: JSON.stringify(data),
    contentType: "application/json",
    processData: false,
    type: 'POST',
    success: function (data) {
        $("#successAlert").show();
    },
    error: function (data) {
        $("#errorAlert").show();
    }
});
```

2.5.3 Bearbeiten oder Löschen einer Person

Das Bearbeiten bzw. Löschen einer Person funktioniert vom Prinzip her genauso wie das Hinzufügen (siehe Punkt 2.5.2), nur das man mittels AJAX eine andere HTTP-Methode angibt, z.B. zum Bearbeiten `type: 'PUT'`.

Zum Bearbeiten der Eigenschaften einer Person haben wir ebenfalls eine zusätzliche HTML-Seite erstellt, welche sich „editPerson.html" nennt und über die URL „/persons/{ID der Person}" erreichbar ist. Folgendermaßen holen wir uns im HTML-File den Namen der Person, welche man bearbeiten möchte:

```
<input type="text" class="form-control" id="name"
th:value="${personRecord.name}"/>
```

2.6 SOAP-Kommunikation

Auf der Seite des Clients werden Daten wie der Host und der Port des SOAP-Services genauso wie der zu suchende Name und die Ausgabedatei angegeben. Diese werden mit einem CLIParser aufbereitet.

```
CLIHandler cliHandler = new SOAPCLIHandler();
cliHandler.parse(args);
```

Aufgrund dieser Angaben wird zunächst eine Verbindung zum SOAP-Service aufgebaut.

```
SOAPConnectionFactory connectionFactory =
SOAPConnectionFactory.newInstance();
SOAPConnection connection = connectionFactory.createConnection();
```

Im Weiteren wird eine SOAP-Message erstellt, die den zu suchenden Namen enthält.

```
…
OAPElement nameElement = requestElement.addChildElement("name", "ek");
nameElement.addTextNode(this.name);
…
```

Diese wird dann an das SOAP-Service geschickt.

```
SOAPMessage message = connection.call(this.messageCreator.create(),
this.webServiceURL);
```

Dort wird erkannt, dass eine Personenanfrage angefordert ist und sucht aufgrund dessen nach passenden Einträgen in der Datenbank. Wenn dies erledigt ist wird eine Antwort erstellt die wieder an den Client übermittelt wird.

```
@PayloadRoot(namespace = NAMESPACE_URI, localPart = "getPersonRequest")
@ResponsePayload
public GetPersonResponse getPersonRecord(@RequestPayload GetPersonRequest request)
{
    GetPersonResponse response = new GetPersonResponse();

response.setPersonRecord(dataRecordRepository.findTop50ByNameContainingIgnoreCase(
request.getName()));
    return response;
}
```

Dort wird die Nachricht in einen String umformatiert und leserlich formatiert.

```
String messageString = SOAPMessageUtils.soapMessageToString(message);
```

Je nach Benutzereingabe wird diese Antwort in ein File oder in die Konsole geschrieben.

```
System.out.println(messageString);
…
writer.write(messageString);
```

Im Folgenden ist eine beispielhafte Antwort des SOAP-Services.

```
<?xml version="1.0" encoding="UTF-16"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Header/>
    <SOAP-ENV:Body>
        <ns3:getPersonResponse xmlns:ns3="http://erceg_kritzl/soa/">
            ...
            <ns3:personRecord>
                <id>568ab2e8545f0d046770865b</id>
                <name>Petra Jamalia Hogan</name>
                <description> Beschreibung … </description>
            </ns3:personRecord>
         ...
        </ns3:getPersonResponse>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

2.7 Verwendung

2.7.1 Requirements

Zum Ausführen der Webapplikation sind folgende Programme notwendig:

- Vagrant
- VirtualBox
- Maven
- Java
- Eclipse oder IntelliJ

2.7.2 Ausführen

1. In den „vagrant" Ordner der Projektstruktur gehen und in der Konsole folgende Befehle ausführen:
   - vagrant plugin install vagrant-vbguest
   - vagrant up
2. Im Root-Verzeichnis des Projektes folgende Befehle mit der Konsole ausführen:
   - mvn clean
   - mvn install
3. Die Ausführung des Web-Services kann durch Ausführen der Klasse „erceg_kritzl.Application" im Ordner Web-Rest mittels IntelliJ oder Eclipse erfolgen. Leider war keine Zeit mehr für ein Deployment
4. Die Jar-Datei des SOAP-Clients ist unter „Client-SOAP\artifact" zu finden
   - java –jar Client-SOAP.jar –h localhost –n <Suchbegriff>
   - Ein Outputfile kann mit „-o" angegeben werden: -o <Dateiname>
   - Der Port kann mit „-p" angegeben werden: -p <Portnummer>

3 Quellenverzeichnis

[1] Github-User „benkeen" (2012, 2015).
Github-Repository „generatedata" – Releases [Online].
Available at: https://github.com/benkeen/generatedata/releases
[zuletzt abgerufen am 11.12.2015]