JUST DO IT
HÖHERE ABTEILUNG FÜR INFORMATIONSTECHNOLOGIE

tgm
Die Schule der Technik

Laborprotokoll

09 - Web Services in Java

Systemtechnik Labor
5BHITT 2015/16, Gruppe Y

Mathias Ritter

Version 1.0

Note:

Betreuer: Prof. Borko

Begonnen am 11. März 2016

Beendet am 31. März 2016

Inhaltsverzeichnis

1 Einführung

Diese Übung zeigt die Anwendung von mobilen Diensten in Java.

1.1 Ziele

Das Ziel dieser Übung ist eine Webanbindung zur Benutzeranmeldung in Java umzusetzen. Dabei soll sich ein Benutzer registrieren und am System anmelden können.

Die Kommunikation zwischen Client und Service soll mit Hilfe von JAX-RS (Gruppe1+2) umgesetzt werden.

1.2 Voraussetzungen

- Grundlagen Java und Java EE
- Verständnis über relationale Datenbanken und dessen Anbindung mittels JDBC oder ORM-Frameworks
- Verständnis von Restful Webservices

1.3 Aufgabenstellung

Es ist ein Webservice mit Java zu implementieren, welches eine einfache Benutzerverwaltung implementiert. Dabei soll die Webapplikation mit den Endpunkten /register und /login erreichbar sein.

Registrierung
Diese soll mit einem Namen, einer eMail-Adresse als BenutzerID und einem Passwort erfolgen. Dabei soll noch auf keine besonderen Sicherheitsmerkmale Wert gelegt werden. Bei einer erfolgreichen Registrierung (alle Elemente entsprechend eingegeben) wird der Benutzer in eine Datebanktabelle abgelegt.

Login
Der Benutzer soll sich mit seiner ID und seinem Passwort entsprechend authentifizieren können. Bei einem erfolgreichen Login soll eine einfache Willkommensnachricht angezeigt werden.

Die erfolgreiche Implementierung soll mit entsprechenden Testfällen (Acceptance-Tests bez. aller funktionaler Anforderungen mittels JUnit) dokumentiert werden. Es muss noch keine grafische Oberfläche implementiert werden! Verwenden Sie auf jeden Fall ein gängiges Build-Management-Tool (z.B. Maven). Dabei ist zu beachten, dass ein einfaches Deployment möglich ist (auch Datenbank mit z.B. file-based DBMS).

Bewertung: 16 Punkte
- Aufsetzen einer Webservice-Schnittstelle (4 Punkte)
- Registrierung von Benutzern mit entsprechender Persistierung (4 Punkte)
- Login und Rückgabe einer Willkommensnachricht (3 Punkte)
- AcceptanceTests (3 Punkte)
- Protokoll (2 Punkte)

2 Ergebnisse

Die Aufgabe wurde mittels Java EE und Spring umgesetzt. Bei der Implementierung haben mir zwei Github-Repositories weitergeholfen [1, 2]. In den folgenden Unterkapiteln wird die Umsetzung genauer erläutert.

2.1 Definition einer User-Entity

Für den User muss eine neue Entity definiert werden, welche in der Datenbank angelegt werden. Dazu wird eine Klasse mit der Annotation „@Entity" erstellt. Die notwendigen Spalten werden als Attribute erstellt.

Der User enthält als Primary Key eine E-Mail-Adresse. Dafür wird die Annotation „@Id" über das entsprechende Attribut geschrieben. Außerdem soll diese niemals leer sein, daher wird „@NotBlank" verwendet. Bei der E-Mail-Adresse soll zusätzlich überprüft werden, ob es sich um eine gültige E-Mail-Adresse handelt. Dazu wird die Annotation „@Email" verwendet:

```
@Id
@NotBlank(message="{NotBlank.user.email}")
@Email(message="{Email.user.email}")
private String email;
```

Das Passwort wird als Hash in der Datenbank gespeichert, aber im Klartext übertragen. Daher sind hierfür zwei Attribute erforderlich, aber nur eines soll persistiert werden. Da das Passwort unbedingt erforderlich ist, wird „@NotNull" über dem Passworthash verwendet.
Um eine Persistierung des Passworts in Klartext zu vermeiden, wird „@Transient" verwendet. Außerdem wird bei diesem Attribut die Passwortlänge von mindestens 5 Zeichen sichergestellt. Weiters wird die JSON-Property auf „password" bei der Serialisierung/Deserialisierung umbenannt:

```
@NotNull(message="{NotNull.user.passwordHash}")
private byte[] passwordHash;

@Transient
@Size(min=5, message="{Size.user.passwordRaw}")
@JsonProperty("password")
private String passwordRaw;
```

2.2 Persistierung der User-Entity

Zur Persistierung wird H2 verwendet. Die Konfiguration erfolgt in der Datei „application.properties" im Ressourcenordner. Dabei wurden folgende Einstellungen getroffen:

```
spring.datasource.url=jdbc:h2:./db/main;Mode=Oracle;DB_CLOSE_ON_EXIT=FALSE
spring.datasource.platform=h2
spring.datasource.continue-on-error=true
spring.jpa.hibernate.ddl-auto=update
```

Die Speicherung der DB erfolgt unter „db/main". Durch die Einstellung „ddl-auto=update" wird das Schema der Datenbank beim Starten immer den Modelklassen bzw. Entities angepasst. Die Daten bleiben außerdem erhalten.

Außerdem wurde die H2 Webkonsole auf der URL „/console" eingebunden. Diese Konfiguration wurde in der Klasse „WebConfiguration" durchgeführt, wobei die Klassenannatation „@Configuration" erforderlich war.

2.3 Validierung der User-Entity

Zur Validierung der User-Entity wird Beanvalidation verwendet.

Um die Fehlermeldung anzupassen, welche bei gescheiterter Validierung ausgegeben wird, werden diese in einer externen Datei definiert. Dadurch wird erreicht, dass diese nicht direkt im Code stehen. Im Code wird die verwendete Property in geschwungenen Klammern angegeben (siehe Code-Listings in vorherigem Kapitel). Die Definition dieser Fehlermeldungen erfolgt in der Datei „ValidationMessages.properties" im Ressourcenordner:

```
Size.user.passwordRaw=The password must be at least 5 characters long
```

Die Validierung wird per Angabe der Annotation „@Valid" ausgelöst. Falls das Objekt nicht valide ist, wird eine „ConstraintViolationException" ausgelöst. Die Validierung findet in der Klasse des RESTful Endpoints statt. Das Exceptionhandling wird von ExceptionMappern übernommen.

2.4 Konfiguration von Jersey

Damit Jersey in Kombination mit Spring verwendet werden kann, ist eine Konfigurationsklasse erforderlich. Mit Hilfe dieser Klasse werden die RESTful Endpoints registriert. Außerdem können ExceptionMapper angegeben werden, welche bei Auftreten einer Exception im Endpoint aufgerufen werden und eine JSON-Errorresponse zurückgeben. Weiters ist eine Einstellung zur Aktivierung der Beanvalidation erforderlich:

```
@Configuration
public class JerseyConfiguration extends ResourceConfig {
    public JerseyConfiguration() {
        this.register(UserEndpoint.class);
        this.register(JsonMappingExceptionMapper.class);
        this.property(ServerProperties.BV_SEND_ERROR_IN_RESPONSE, true);
    }
}
```

2.5 Definition des Repositories

Um User-Objekte zu persistieren bzw. abzurufen, ist eine Schnittstelle erforderlich. Diese wird in Spring mittels Repositories realisiert. Da nur einfache CRUD-Funktionalität (Create, Read, Update, Delete) benötigt wird, wird „CrudRepository" implementiert. Das betreffende Objekt, somit der User, und dessen ID, somit String (E-Mail-Adresse) wird als generischer Parameter angegeben:

```
@Repository
public interface UserRepository extends CrudRepository<User, String> {}
```

Um Spring mitzuteilen, dass es sich um ein Repository handelt, ist „@Repository" erforderlich. Eine Implementierung dieses Interfaces ist nicht erforderlich und wird von Spring zur Verfügung gestellt.

2.6 Definition des RESTful Endpoints

Der RESTful Endpoint zur Registrierung und zum Login wurde in der Klasse „UserEndpoint" implementiert. Über der Klasse muss mit „@Produces" angegeben werden, welcher Medientyp zurückgegeben wird. Als Medientyp wurde JSON gewählt.

Der Endpoint benötigt das Repository, um auf User-Objekte zuzugreifen bzw. diese zu persistieren. Das Repository wird mittels Dependency-Injection übergeben.

```
@Autowired
private UserRepository repo;
```

Die Response des Endpoints auf alle Anfragen besteht aus einer „ResponseMessage". Die „ResponseMessage" besteht aus den beiden Attributen „message" und „status". „message" erhält eine Nachricht, wie z.B. eine Fehlermeldung im Fehlerfall. „status" enthält den HTTP-Statuscode.

Registrierung

Für die Registrierung wurde die URI „/register" gewählt. An diese URI muss ein POST-Request mit folgendem Body gesendet werden, um einen neuen User zu registrieren:

```
{
	"email": "<E-MAIL>",
	"password": "<PASSWORD>"
}
```

Um Anfragen an diese URI zu verarbeiten, wird eine Methode definiert, und dieser Pfad als Annotation angegeben. Außerdem wird ebenfalls die Methode mit „@POST" spezifiziert. Im Body der Anfrage muss das User-Objekt in JSON angegeben werden. Dieses wird als Parameter angegeben:

```
@POST
@Path("/register")
public Response register(@Valid User user) {
	// Implementation...
}
```

Bei der Registrierung wird das User-Objekt aufgrund der im Model angegebenen Annotations validiert. Zusätzlich wird überprüft, ob ein User mit der angegebenen E-Mail-Adresse bereits existiert. Im Fehlerfall wird der Statuscode „400" (Bad Request) zurückgegeben. Sollte die Registrierung erfolgreich verlaufen sein, wird „201" (Created) zurückgegeben.

Login

Für das Login wurde die URI „/login" gewählt. An diese URI muss ein POST-Request mit folgendem Body gesendet werden:

```
{
	"email": "<E-MAIL>",
	"password": "<PASSWORD>"
}
```

Ähnlich wie die Registrierung wird die Methode mit POST angegeben und ein User-Objekt im Body übergeben:

```
@POST
@Path("/login")
public Response login(@Valid User user) {
	// Implementation...
}
```

Genauso wie bei der Registrierung erfolgt eine Validierung des User-Objekts. Bei fehlgeschlagener Validierung wird der Statuscode „400" (Bad Request) zurückgegeben. Zusätzlich wird das angegebene Passwort überprüft. Sollte dieses nicht übereinstimmen, wird „403" (Forbidden) zurückgegeben.

2.7 ExceptionMapper (Handling von Exceptions)

Sollte es während der Abarbeitung eines Requests zu einer Exception kommen, z.B. weil bei der Validierung ein Constraint verletzt wurde, wird ein registrierter ExceptionMapper zur Verarbeitung verwendet.

Zuerst wurde ein Basis-Exceptionmapper implementiert, welche für alle Klassen, welche von Exception erben (somit für alle Exceptions) zuständig ist. Dafür muss die Klasse „ExceprtionMapper" implementiert und als generisches Argument die Klasse „Exception" angegeben werden. Implementiert wird die Methode „toResponse(Exception e)". Als Response wird ein Internal Server Error zurückgegeben:

```
public Response toResponse(Exception e) {
    return ResponseUtil.internalServerError(e.getMessage());
}
```

Zusätzlich zu diesem allgemeinen Exceptionmapper wurden zwei weitere spezielle Exceptionmapper definiert. Ein Exceptionmapper ist für „ConstraintViolationException" zuständig, welche beim Validieren auftreten kann. Der andere ist für „JsonMappingException" zuständig, welche beim Mapping von JSON auf Java-Objekte auftreten kann. Beide führen zu einer Response in Form eines Bad Requests.

2.8 Testfälle

Es wurden Acceptance Tests des RESTful Endpoints durchgeführt. Details dazu sind im Testbericht zu finden, welche sich ebenfalls in diesem jar-Archiv befindet.

2.9 Ausführen der Applikation

Durch den Einsatz von Spring Boot ist in der jar-Datei bereits ein Applicationserver integriert. Daher kann die Anwendung mit „java -jar" gestartet werden.

Standardmäßig startet der Server auf Port 8080, dies kann allerdings mittels des Parameters „--server.port=<PORT>" geändert werden.

URL für die Registrierung: „http://127.0.0.1:8080/register"
URL für das Login: „http://127.0.0.1:8080/login"

3 Repository

Das Repository ist verfügbar unter: https://github.com/mritter-tgm/DezSys09

4 Literaturverzeichnis

[1] Github-Repostory der Übung „DezSys09"; Rene Hollander [Online]
Verfügbar unter: https://github.com/ReneHollander/
dezsys09-webservices
[Zuletzt abgerufen am 30.03.2016]

[2] Github-Repostory der Übung „DezSys09"; Stefan Geyer [Online]
Verfügbar unter: https://github.com/sgeyer-tgm/DezSys09
[Zuletzt abgerufen am 30.03.2016]