Middleware – Cloud Computing – Übung

Web-Services: RESTful Web-Services in Java

Wintersemester 2025/26

Paul Bergmann, Christian Berger

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Systemsoftware)

https://sys.cs.fau.de

Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Technische Fakultät

Überblick

Implementierung von RESTful Web-Services in Java

Implementierung von RESTful Web-Services in Java

RESTful Web-Services in Java

- Java API for RESTful Web Services (JAX-RS)
  - Schnittstellen für die Implementierung von HTTP auf Client-Seite
  - Entwicklung und Ausführung der Server-Seite
    - Implementierung als Java-Anwendung
    - Bereitstellung über einen Java-internen Web-Server
  - Annotationen als zentrales Hilfsmittel
    - Verknüpfung von HTTP-Operationen und Java-Methoden
    - Zuordnung von URI-Pfaden zu Methoden
  - Packages: `javax.ws.rs.*`
  - Tutorial: `https://javaee.github.io/tutorial/jaxrs.html`
- Jersey
  - Framework zur Entwicklung JAX-RS-basierter Web-Services
  - Bietet verschiedene Web-Server-Implementierungen. In der Übung: Grizzly
  - Packages: `org.glassfish.jersey.*`
  - Projektseite: `https://eclipse-ee4j.github.io/jersey/`
- Bibliotheken im CIP-Pool (Java 11+): `/proj/i4mw/pub/aufgabe1/`

Grundgerüst

- Server-Seite
  - Festlegung des Server-Pfads per `@Path`-Annotation
  - `@Singleton` verhindert die Erzeugung einer neuen Instanz für jeden Aufruf

```
@Singleton                                                    // Server
@Path("queue")
public class MWQueueServer { // Warteschlangen-Service
  private List<String> queue = new LinkedList<String>();
  // [...] Default-Konstruktor, falls weitere Konstruktoren existieren
  // [...] Methodenimplementierungen (siehe nachfolgende Folien)
}
```

```
public static void main(String[] args) {
  URI uri = UriBuilder.fromUri("http://[::]/").port(12345).build();
  ResourceConfig config = new ResourceConfig(MWQueueServer.class);
  GrizzlyHttpServerFactory.createHttpServer(uri, config);
}
```

- Client-Seite

```
URI uri = UriBuilder.fromUri("http://localhost/").port(12345).build();                    // Client
WebTarget client = ClientBuilder.newClient().target(uri).path("queue");
// [...] Methodenaufrufe (siehe nachfolgende Folien)
```

Bereitstellung von Methoden

- Server-Seite
  - Implementierung der Funktionalität mittels `public`-Methoden
  - Festlegung der HTTP-Zugriffsmethode über entsprechende Annotation
  - Spezifische Unterpfade für Methoden möglich
  - Repräsentation der Antwort durch `Response`-Objekt

```
@GET
@Path("size")
public Response getSize() {
  return Response.ok(queue.size()).build();
}
```

- Client-Seite
  - Festlegung des Unterpfads mittels `path()`-Methode
  - Auswahl der HTTP-Operation über entsprechende Methode (hier: `get()`)
  - Deserialisieren des Rückgabewerts per `readEntity()` am `Response`-Objekt

```
WebTarget client = [...];
Response response = client.path("size").request().get();
Integer size = response.readEntity(Integer.class);
response.close();
```

- Konzept
  - Übergabe von Aufrufparametern als Teil des Pfads
  - Interpretation dieser Pfadelemente auf Server-Seite

- Server-Seite
  - Kennzeichnung der im Pfad kodierten Variablen mit „`{...}`“
  - Zugriff auf Pfadparameter mit `@PathParam` und Variablenname

```
@GET
@Path("{index}")
public Response get(@PathParam("index") int i) {
  return Response.ok(queue.get(i)).build();
}
```

- Client-Seite

```
WebTarget client = [...];
Response response = client.path("1").request().get();
String value = response.readEntity(String.class);
response.close();
```

- Konzept
  - Übergabe von Aufrufparametern im Query-Teil der URI
  - Beispiel: `http://localhost:12345/queue/index-of?value=example`

- Server-Seite
  - Zugriff auf Anfrageparameter über `@QueryParam`-Annotation
  - Angabe von Standard-Werten mittels `@DefaultValue`-Annotation

```
@GET
@Path("index-of")
public Response indexOf(@QueryParam("value") @DefaultValue("") String v) {
  return Response.ok(queue.indexOf(v)).build();
}
```

- Client-Seite

```
WebTarget client = [...];
Response response = client.path("index-of").queryParam("value", "example").request().get();
Integer index = response.readEntity(Integer.class);
response.close();
```

- Konzept
  - Übergabe eines Aufrufparameters im Body der HTTP-Anfrage
  - Einsatz der HTTP-Operationen `PUT` oder `POST`
- Server-Seite
  - Spezifizierung eines einzelnen Parameters
  - Automatische Konvertierung der Daten durch die Laufzeitumgebung

```
@PUT
@Path("tail")
public Response add(String value) {
  queue.add(value);
  return Response.ok().build();
}
```

- Client-Seite
  - Übergabe des Werts und Festlegung des Formats mittels `Entity`-Objekt
  - Beispiele: Text (`Entity.text()`) oder JSON (`Entity.json()`)

```
WebTarget client = [...];
client.path("tail").request().put(Entity.text("example")).close();
```

- Übertragung von Generics-Datentypen
  - Grundsätzliche Vorgehensweise wie bei Java-Standarddatentypen
  - Sonderbehandlung bei Deserialisierung

- Server-Seite

```
@GET
public Response list() {
  ArrayList<String> queueCopy = new ArrayList(queue);
  return Response.ok(queueCopy).build();
}
```

- Client-Seite
  - Standardansatz mangels `class`-Objekten für Generics nicht möglich
  - Bereitstellung der Typ-Information mittels `GenericType`-Hilfsobjekt

```
WebTarget client = [...];
Response response = client.request().get();
GenericType<List<String>> type = new GenericType<List<String>>() {};
List<String> list = response.readEntity(type);
response.close();
```

- Nutzerdefiniertes Objekt als Rückgabewert

```
@POST
@Path("find")
public Response find(String prefix) {
  MWQueueElement element = [...]; // Bestimmung des Ergebnisses
  return Response.ok(element).build();
}
```

```
Response response = client.path("find").request().post(Entity.text("t"));
MWQueueElement element = response.readEntity(MWQueueElement.class);
response.close();
```

- Nutzerdefiniertes Objekt als Aufrufparameter

```
@POST
public Response insert(MWQueueElement element) {
  queue.add(element.getIndex(), element.getValue());
  return Response.ok().build();
}
```

```
MWQueueElement element = new MWQueueElement(1, "test");
client.request().post(Entity.json(element)).close();
```

- Übertragung nutzerdefinierter Objekte mittels JSON
  - Default-Konstruktor erforderlich, falls weitere Konstruktoren existieren
  - Getter- und Setter-Methoden für zu übertragende private Attribute nötig

```
public class MWQueueElement {
  private int index;
  private String value;

  public MWQueueElement() {}
  public MWQueueElement(int index, String value) {
    this.index = index;
    this.value = value;
  }

  public int getIndex()              { return index; }
  public void setIndex(int index)    { this.index = index; }
  public String getValue()           { return value; }
  public void setValue(String value) { this.value = value; }
}
```

Status- bzw. Fehlermeldungen

- Konzept
  - Keine direkte Weitergabe von Exceptions
  - Abbildung von Fehlern auf HTTP-Status-Codes
- Server-Seite (Alternativen)
  - Werfen einer `WebApplicationException` mit entsprechendem Status-Code

```
if([...]) throw new WebApplicationException(Status.BAD_REQUEST);
```

  - Konfigurierung des Status-Codes durch Methode am Antwortobjekt

```
if([...]) return Response.serverError().build();
```

- Client-Seite

```
Response response = [...];
switch(Status.fromStatusCode(response.getStatus())) {
  case OK:
    [...] // Verarbeitung des Ergebnisses
  case BAD_REQUEST:
    [...] // Reaktion auf Fehler
  [...] // Behandlung weiterer Status-Codes
}
```

- Problem: Keine Anzeige von Exceptions **auf Server-Seite**
- Abfangen und Darstellen mittels Exception-Handler
  - Kennzeichnung als `@Provider`
  - Propagieren des Fehler-Status-Codes bei `WebApplicationExceptions`

```
@Provider
public class MWErrorHandler implements ExceptionMapper<Throwable> {
  public Response toResponse(Throwable error) {
    // Ausgabe der Exception
    error.printStackTrace();

    // Propagieren der Exception
    if(error instanceof WebApplicationException) {
      return ((WebApplicationException) error).getResponse();
    } else return Response.serverError().build();
  }
}
```

- Handler-Registrierung als Teil der Web-Server-Konfiguration

```
ResourceConfig config = new ResourceConfig(MWQueueServer.class);
config.register(MWErrorHandler.class);
```

- HTTP-Debugging auf der Kommandozeile mittels cURL
- Zentrale Parameter (siehe Manpage: `man curl`)
  - `-v` Ausgabe des vollständigen Nachrichtenaustauschs
  - `-X {GET,PUT,...}` Festlegung der HTTP-Operation
  - `-d <data>` Übergabe von Daten im HTTP-Body
  - `-u <username>` Angabe eines Logins [→ Passworteingabe bei anschließender Abfrage]

```
$ curl -v -X PUT -d "example" http://localhost:12345/queue/tail
[...]
> PUT /queue/tail HTTP/1.1
> Host: localhost:12345
> User-Agent: curl/7.52.1
> Accept: */*
> Content-Length: 7
> Content-Type: application/x-www-form-urlencoded
[...]
< HTTP/1.1 200 OK
< Date: Mon, 16 Oct 2017 10:45:03 GMT
< Content-length: 0
[...]
```