

Middleware – Cloud Computing – Übung

Christopher Eibel, Michael Eischer, Tobias Distler

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)

www4.cs.fau.de

Wintersemester 2018/19



Web-Services

RESTful Web-Services

Implementierung von RESTful Web-Services

Registrierung von Web-Services

Aufgabe 1



■ Web-Service

- Software-System zur Interaktion zwischen Rechnern über ein Netzwerk
- Plattformunabhängigkeit durch Einsatz von Web-Standards (z. B. HTTP)
- Unterschiedliche Ausprägungen (Beispiele)
 - SOAP
 - **REST**
 - XML-RPC

■ RESTful Web-Service

- Umsetzung des Prinzips des *Representational State Transfer (REST)*
 - Identifizierung von Ressourcen mittels *Universal Resource Identifiers (URIs)*
 - Darstellung von Ressourcenzuständen anhand von *Repräsentationen*
- HTTP als Anwendungsprotokoll
- Literatur



Roy Fielding

Architectural Styles and the Design of Network-based Software Architectures,
Dissertation, 2000.



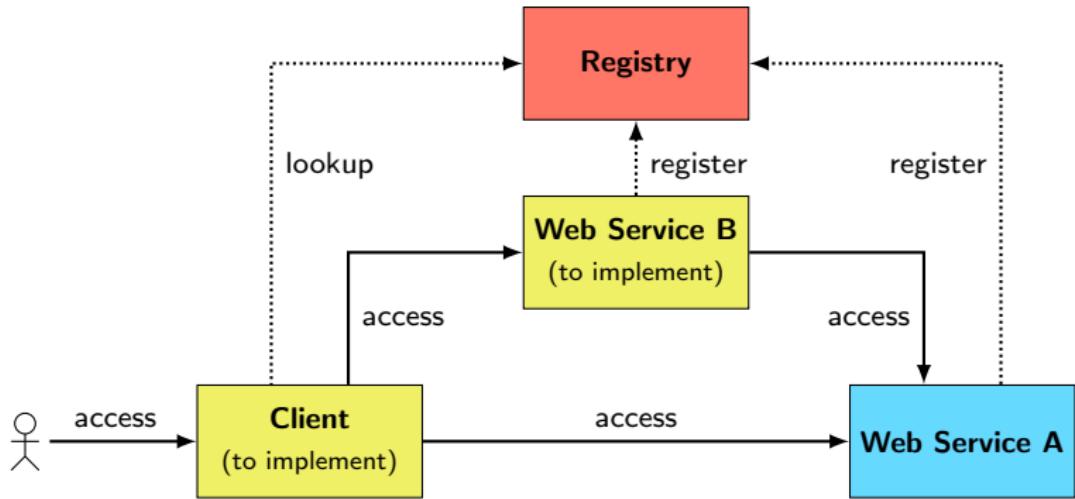
HTTP als Anwendungsprotokoll

- Übertragung von Daten in HTTP-Anfragen und -Antworten
 - Header für Metadaten
 - Body für Nutzdaten (optional)
- Ausführung von Aktionen mittels HTTP-Operationen (Beispiele)
 - GET Lesezugriff auf eine Ressource
 - PUT Schreibzugriff auf eine Ressource
 - DELETE Löschen einer Ressource
 - POST Übermittlung von Daten an eine Ressource
- Senden von Statusmeldungen durch HTTP-Status-Codes (Beispiele)
 - 200 OK Erfolgreiche Bearbeitung
 - 400 Bad Request Fehlerhafte Anfragenachricht
 - 404 Not Found Ressource existiert nicht
- Literatur
 -  Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter et al.
Hypertext Transfer Protocol – HTTP/1.1, RFC 2616, 1999.



Aufgabe 1: Web-Services

- Nutzung eines existierenden Web-Services
- Bereitstellung eines eigenen RESTful Web-Services



Web-Services

RESTful Web-Services

Implementierung von RESTful Web-Services

Registrierung von Web-Services

Aufgabe 1



- 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://docs.oracle.com/javaee/7/tutorial/jaxrs.htm>
- Jersey
 - Framework zur Entwicklung JAX-RS-basierter Web-Services
 - Bietet verschiedene Web-Server-Implementierungen. In der Übung: Grizzly
 - Packages: org.glassfish.jersey.*
 - Projektseite: <https://jersey.github.io/>
 - Bibliotheken im CIP-Pool (Java 8+): /proj/i4mw/pub/aufgabe1/



■ Server-Seite

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

```
@Singleton  
@Path("queue")  
public class MWQueueServer {  
    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();  
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 und Arrays
 - Grundsätzliche Vorgehensweise wie bei Java-Standarddatentypen
 - Sonderbehandlung bei Deserialisierung
- Server-Seite

```
@GET  
public Response list() {  
    String[] array = new String[queue.size()];  
    queue.toArray(array);  
    return Response.ok(array).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<String[]> type = new GenericType<String[]>() {};  
String[] array = response.readEntity(type);  
response.close();
```



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

```
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; }  
}
```



■ Nutzerdefiniertes Objekt als Aufrufparameter

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

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

■ Nutzerdefiniertes Objekt als Rückgabewert

```
@POST // Server-Seite
@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(); // Client-Seite
```



- Konzept
 - Abbildung von Fehlern auf HTTP-Status-Codes
 - Keine direkte Weitergabe von Exceptions
- Server-Seite (Alternativen)
 - Werfen einer WebApplicationException mit entsprechendem Status-Code
 - Konfigurierung des Status-Codes durch Methode am Antwortobjekt

```
if([...]) throw new WebApplicationException(Status.BAD_REQUEST);
if([...]) return Response.serverError().build();
```

- Client-Seite

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



- Problem: Keine Anzeige von Exceptions **auf Server-Seite**
- Auffangen 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
[...]
```



Web-Services

RESTful Web-Services

Implementierung von RESTful Web-Services

Registrierung von Web-Services

Aufgabe 1



- Problemstellung
 - Große Anzahl verschiedener Web-Services in einem System
 - Adressen von Diensten können sich mit der Zeit ändern
 - Clients benötigen aktuelle Web-Service-Adressen
- Möglicher Lösungsansatz: Einsatz einer Registry
 - Verzeichnisdienst zur Verwaltung von Metadaten verfügbarer Dienste
 - Registry-Adresse ist im System global bekannt
 - Treffpunkt zwischen Dienstanbieter und Dienstnehmer
 - Dienstanbieter registriert Web-Service-Adresse unter einem Namen
 - Dienstnehmer findet Web-Service-Adresse mittels Suchanfrage nach Namen
 - Keine Einbeziehung der Registry in anschließende Client-Dienst-Interaktion
- I4-Registry
 - Implementierung als RESTful Web-Service
 - Bereitstellung auf einem Rechner des Lehrstuhls
 - Registry-URI in der Datei `/proj/i4mw/pub/aufgabe1/registry.address`



- Hierarchische Verwaltung von Einträgen
 - Gruppe (String): Eine für jede Übungsgruppe (z. B. „gruppe0“)
 - Dienst (String)
 - Zuordnung zu einer Gruppe
 - Mehrere Dienste pro Gruppe möglich
 - Schlüssel-Wert-Paar (jeweils String)
 - Zuordnung zu einem Dienst
 - Mehrere Schlüssel-Wert-Paare pro Dienst möglich
- Schnittstelle

■ GET	/registry	Auflistung der Gruppen
■ GET	/registry/{group}	Auflistung der Dienste
■ PUT	/registry/{group}/{service}	Erstellen eines Diensts
■ DELETE	/registry/{group}/{service}	Löschen eines Diensts
■ GET	/registry/{group}/{service}	Auflistung der Schlüssel
■ GET	/registry/{group}/{service}/{key}	Ausgabe eines Werts
■ PUT	/registry/{group}/{service}/{key}	Setzen eines Werts
■ DELETE	/registry/{group}/{service}/{key}	Löschen eines Werts



Web-Services

RESTful Web-Services

Implementierung von RESTful Web-Services

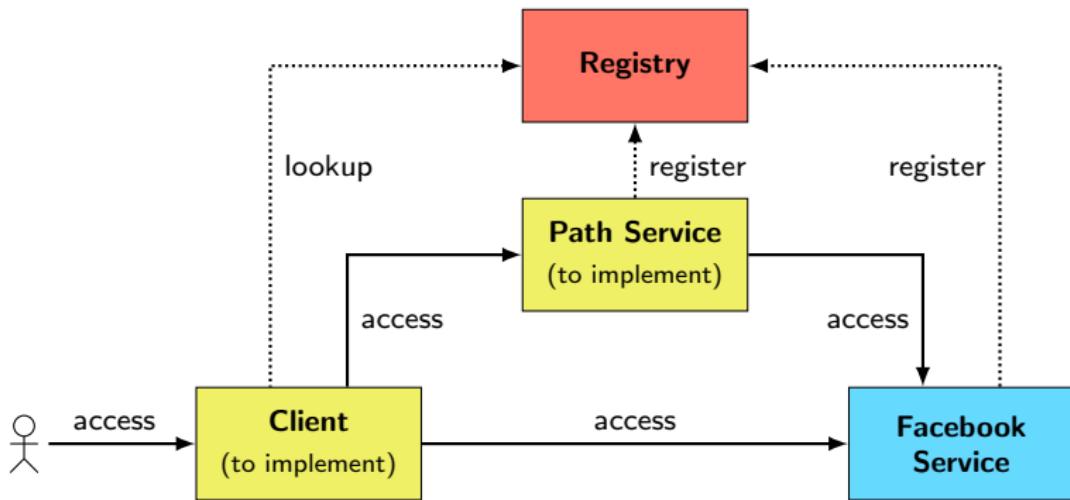
Registrierung von Web-Services

Aufgabe 1



Aufgabe 1: Web-Services

- Bereitstellung eines eigenen RESTful Web-Services
- Teilaufgaben
 - Kommandozeilen-Client für Registry-Zugriff
 - Web-Service zur Erweiterung eines bereits bestehenden Web-Services
 - Client zum Zugriff auf beide Web-Services



■ Zu implementierende Kommandos

- | | |
|-------------------------------|--|
| ■ <code>list-groups</code> | Auflisten aller existierender Gruppen |
| ■ <code>list-services</code> | Auflisten aller Dienste einer Gruppe |
| ■ <code>create-service</code> | Erstellen eines neuen Diensts |
| ■ <code>delete-service</code> | Löschen eines existierenden Diensts |
| ■ <code>list-keys</code> | Auflisten aller Schlüssel eines Diensts |
| ■ <code>get-value</code> | Lesen eines zu einem Schlüssel gehörigen Werts |
| ■ <code>put-value</code> | Speichern eines Schlüssel-Wert-Paars |
| ■ <code>delete-value</code> | Löschen eines Schlüssel-Wert-Paars |

■ Registry-Zugang

- Lesezugriff auf alle Einträge, Schreibzugriff beschränkt auf eigene Gruppe
- Nutzername (`user`): Eigener Gruppenname (z. B. „gruppe0“)
- Passwort (`pwd`): Siehe Bestätigungs-E-Mail nach Gruppenanmeldung

```
WebTarget client = [...];
HttpAuthenticationFeature af = HttpAuthenticationFeature.basic(user, pwd);
client.register(af);
```

[Hinweis: Nur der erste Aufruf von `register(af)` setzt Nutzername und Passwort; weitere Aufrufe haben keinen Effekt.]



- Überblick
 - Verwaltung von Nutzern und ihrer Freundschaftsbeziehungen zu anderen
 - Zusammenhängender Graph aus Freundschaftsbeziehungen
 - Bereitstellung auf einem Lehrstuhlrechner
- Verwaltete Informationen für jeden Nutzer
 - *ID* Eindeutige Kennzeichnung des Nutzers
 - *Name* (Klar-)Name des Nutzers
 - *Freunde* Liste mit den IDs von Freunden des Nutzers
- Basisschnittstelle
 - GET /facebook/search?string=<s> Suche nach Nutzern, deren Name die Zeichenkette <s> enthält (maximal 1000 Ergebnisse)
 - GET /facebook/names/{id} Ausgabe des Klarnamens zu einer ID
 - GET /facebook/friends/{id} Ausgabe aller Freunde einer ID



- Gebündelte Abfrage von Klarnamen
 - Methode POST
 - Pfad /facebook/names
 - Anfrage-Body String-Array `ids` der abzufragenden IDs
 - Antwort-Body String-Array `names` der Klarnamen, wobei `names[i]` den Klarnamen von `ids[i]` repräsentiert
- Gebündelte Abfrage von Freundschaftsbeziehungen
 - Methode POST
 - Pfad /facebook/friends
 - Anfrage-Body String-Array `ids` der abzufragenden IDs
 - Antwort-Body Map<String, HashSet<String>> `friends` der angeforderten Freundschaftsbeziehungen
 - Schlüssel in `friends` entsprechen den abgefragten IDs
 - Werte in `friends` enthalten die jeweiligen Freund-IDs



- Überblick
 - Ermittlung der kürzesten Verbindung zwischen zwei Facebook-Nutzern
 - Im Rahmen von Aufgabe 1 selbst zu implementieren
- Ausgabe der kürzesten Verbindung zwischen {startID} und {endID}
 - Methode GET
 - Pfad /path/{startID}/{endID}
 - Antwort-Body MWPath-Objekt mit Pfad path und Aufrufstatistiken

```
public class MWPath {  
    String[] path;  
    [...] // Statistiken  
}
```

- Implementierung
 - Rückgriff auf den Facebook-Dienst
 - Bestimmung des kürzesten Pfads
 - Bereitgestellt: Implementierung des Dijkstra-Algorithmus
 - Zu implementieren: Zusammenstellung der Eingabemenge von IDs



Client für Web-Services

- Zugriff auf Pfad- und Facebook-Dienst per Kommandozeilen-Client
- Zu implementierende Kommandos
 - search Suche nach Nutzern
 - friends Ausgabe der Namen aller Freunde eines Nutzers
 - path Kürzester Pfad (Nutzernamen) zwischen zwei Nutzern

