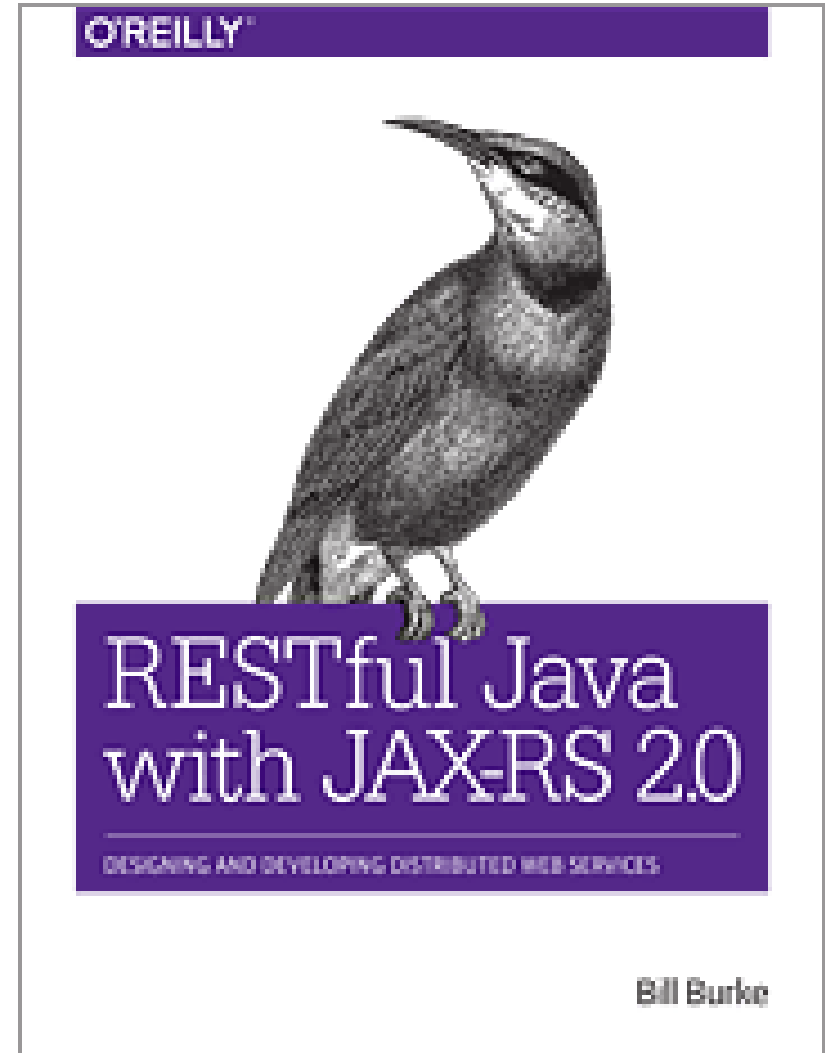
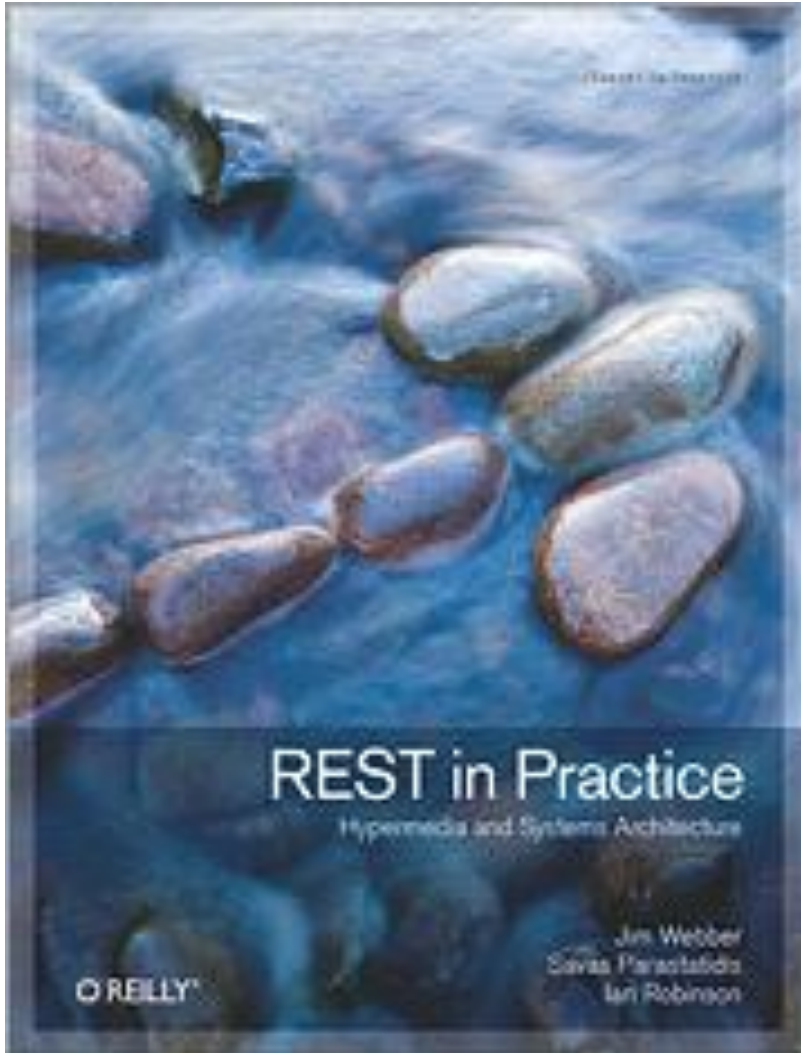


RESTful WebServices



- Die in diesem Seminar verwendete Werkzeuge und Frameworks sind Open Source
 - LPGL Lizenzmodell
- Dies ist ein Programmier-Seminar
 - Damit werden die Inhalte durch Übungen vertieft und verinnerlicht
 - Musterbeispiele werden zur Verfügung gestellt
 - Diese können am Ende des Seminars als ZIP-Datei kopiert werden
 - USB-Stick oder ähnliches
- Dokumentation und Ressourcen stehen auch im Internet zur Verfügung
- Konventionen
 - Befehle werden in `Courier-Schriftart` dargestellt
 - Dateinamen werden in *`kursiver Courier-Schriftart`* dargestellt
 - Links werden in `unterstrichener Courier-Schriftart` dargestellt
 - Zitate werden in *"Anführungszeichen kursiv"* formatiert, die Quellenangabe steht eingerückt darunter

© Javacream

Javacream

Dr. Rainer Sawitzki

Alois-Gilg-Weg 6

81373 München

Alle Rechte, einschließlich derjenigen des auszugsweisen Abdrucks, der fotomechanischen und elektronischen Wiedergabe vorbehalten.

Übersicht	6
Services	25
Server	39
Client	69
Weitere Themen	85

1 ÜBERSICHT

1.1

DER REST-ANSATZ

- Doktorarbeit von Roy Fielding, 2000
- Siehe <http://en.wikipedia.org/wiki/REST>

Representational state transfer

From Wikipedia, the free encyclopedia
(Redirected from [REST](#))

"REST" redirects here. For other uses, see [Rest](#).

Representational state transfer (REST) is a software architectural style consisting of a coordinated set of architectural constraints applied to components, connectors, and data elements, within a distributed [hypermedia](#) system. REST ignores the details of component implementation and protocol syntax in order to focus on the roles of components, the constraints upon their interaction with other components, and their interpretation of significant data elements.^{[1][2]}

The term *representational state transfer* was introduced and defined in 2000 by [Roy Fielding](#) in his doctoral dissertation at [UC Irvine](#).^{[1][3]} REST has been applied to describe desired web architecture, to identify existing problems, to compare alternative solutions, and to ensure that protocol extensions would not violate the core constraints that make the web successful. Fielding used REST to design [HTTP 1.1](#) and [Uniform Resource Identifiers \(URI\)](#).^{[4][5]} The REST architectural style is also applied to the development of [web services](#)^[6] as an alternative to other distributed-computing specifications such as [SOAP](#).

Contents [\[hide\]](#)

- 1 History
- 2 Software architecture
 - 2.1 Components
 - 2.2 Connectors
 - 2.3 Data
- 3 Architectural properties
- 4 Architectural constraints
 - 4.1 Client-server
 - 4.2 Stateless
 - 4.3 Cacheable
 - 4.4 Layered system
 - 4.5 Code on demand (optional)
 - 4.6 Uniform interface

- Die Arbeitsweise des Internet abstrahiert
 - Was sind Ressourcen?
 - Wie werden Ressourcen identifiziert?
 - Was sind Ressourcen-Operationen?
 - Wie werden Services beschrieben?
 - Was sind Stateless Operationen?
 - Voraussetzungen und Umsetzung von Caching-Mechanismen
 - Optional: Übertragung von Skript-Logik auf den Client
- Grundlegendes Konzept sind die verlinkten HyperText-Dokumente
- Nicht überraschend: „Das Internet ist ein Beispiel für die Implementierung eines REST-basierten Systems“

1.2

REST UND HTTP

- Eine umfassende Spezifikation des w3w-Konsortiums
 - Siehe <http://en.wikipedia.org/wiki/Http>

Hypertext Transfer Protocol

From Wikipedia, the free encyclopedia
(Redirected from [Http](#))

The **Hypertext Transfer Protocol (HTTP)** is an [application protocol](#) for distributed, collaborative, [hypermedia](#) information systems.^[1] HTTP is the foundation of data communication for the [World Wide Web](#).

[Hypertext](#) is structured text that uses logical links ([hyperlinks](#)) between [nodes](#) containing text. HTTP is the protocol to exchange or transfer hypertext.

The standards development of HTTP was coordinated by the [Internet Engineering Task Force \(IETF\)](#) and the [World Wide Web Consortium \(W3C\)](#), culminating in the publication of a series of [Requests for Comments \(RFCs\)](#), most notably [RFC 2616](#) (June 1999), which defines HTTP/1.1, the version of HTTP in common use.

Contents [\[hide\]](#)

- 1 Technical overview
- 2 History
- 3 HTTP session
- 4 Request methods
 - 4.1 Safe methods
 - 4.2 Idempotent methods and web applications
 - 4.3 Security
- 5 Status codes
- 6 Persistent connections
- 7 HTTP session state
- 8 Encrypted connections
- 9 Request message
- 10 Response message
- 11 Example session
 - 11.1 Client request

Internet protocol suite

- Application layer**
BGP • DHCP (DHCPv6) • DNS • FTP • **HTTP** • IMAP • IRC • LDAP • MGCP • NNTP • NTP • POP • RPC • RTP • RTSP • RIP • SIP • SMTP • SNMP • SOCKS • SSH • Telnet • TLS/SSL • XMPP • *more...*
- Transport layer**
TCP • UDP • DCCP • SCTP • RSVP • *more...*
- Internet layer**
IP (IPv4 • IPv6) • ICMP • ICMPv6 • ECN • IGMP • IPsec • *more...*
- Link layer**
ARP/InARP • NDP • OSPF • Tunnels (L2TP) • PPP • Media access control (Ethernet • DSL • ISDN • FDDI • DOCSIS) • *more...*

V • T • E

- Definition von URIs
 - Pfad
 - Parameter
- http-Request und http-Response
 - Daten-Container mit Header und Body
 - Encodierung
- Umfassender Satz von Header-Properties
 - Content-Length
 - Accepts
 - Content-Type

- http-Methoden
 - PUT
 - GET
 - POST
 - DELETE
 - OPTIONS
 - HEAD
- Statuscodes für Aufrufe
 - 404: „Not found“
 - 204: „Created“
 - ...

- Definition der Datentypen des Internet
 - Nicht zu verwechseln mit einem XML-Schema
 - Ein MimeType ist „nur“ eine strukturierte Zeichenkette
 - Eigene Erweiterungen sind möglich

- REST hat mit http prinzipiell nichts zu tun
 - REST ist eine abstrakte Architektur
 - http ist ein konkretes Kommunikationsprotokoll
- Aber
 - http passt als Kommunikations-Protokoll der „Referenz-Implementierung“ Internet natürlich perfekt zum REST-Stil

- http Methoden und Ressourcen-Operationen
 - PUT
 - Neu-Anlegen einer Ressource
 - Aktualisierung
 - GET
 - Lesen einer Ressource
 - POST
 - Aktualisierung
 - Neuanlage
 - DELETE
 - Löschen
- Idempotenz
 - Idempotente Operationen dürfen beliebig oft aufgerufen werden und verursachen keine Nebeneffekte
 - REST verlangt eine idempotente Implementierung für PUT und DELETE

- Mit PUT
 - Der Client muss die Ressourcen-ID mit angeben
 - Rückgabe ist ein Statuscode „201: Created“
- Mit POST
 - Der Server entscheidet, ob er eine neue Ressource anlegen muss
 - Falls ja:
 - Statuscode „201: Created“
 - Gesetzter `Location`-Header mit URI der eben angelegten Ressource
 - Optional: Body enthält die angelegte Ressource

- Mit PUT
 - Statuscode „200: OK“ oder „204: No content“
 - PUT ist idempotent (!)
- Mit POST
 - POST wird für nicht-idempotente Updates benutzt

- Mit DELETE
 - Statuscode „200: OK“ oder „204: No content“
 - PUT ist idempotent (!)
- Konzeptionell muss unterschieden werden:
 - Ein „echtes“ DELETE löscht die Ressource
 - Ein fachliches Löschen (z.B. Storno) ist eigentlich ein Update der Ressource
 - Ein überladen des http-DELETE ist für diese Zwecke jedoch durchaus legitim
 - `DELETE order/ISBN42?cancel=true`

1.3

REST UND WEB ANWENDUNGEN IM BROWSER

- Die Datenerfassung im Browser erfolgt klassischerweise in HTML-Formularen
- Ein Formular-Submit ohne Einsatz von JavaScript unterstützt jedoch nur die http-Methoden GET und POST!
 - Damit müssen DELETE und PUT-Operationen irgendwie nachgebildet werden
 - Dieses Problem kann auch im Zusammenspiel mit Firewalls auftreten, die nur GET- und POST-Requests durchlassen
 - Beispielsweise durch die Übertragung eines Request-Parameters im Rahmen eines POST-Requests
 - <http://host:port/application/resource/id?method=delete>
 - <http://host:port/application/resource/id?method=create>
 - Der Server steuert in Abhängigkeit des `method`-Parameters die Anwendungslogik

- Dieser Workaround passt nicht auf die http-Spezifikation
 - Statuscodes
 - Idempotenz und Caching
- Programmierer benutzen Request-Parameter „für Alles“
 - <http://host:port/application/resource?id=00815&method=create&title=BookTitle&price=19.99>
 - Damit geht der REST-Stil komplett verloren
- Im Internet entstand eine Parallel-Welt, bestehend aus
 - den originalen RESTful „statischen“ Ressourcen-Zugriffen
 - Dynamischen Action-Urls, die verkappt einen Remote Procedure Call via http definieren

- Im Zusammenhang mit der zunehmenden Verbreitung und Akzeptanz von JavaScript wurden RESTful Web Services konsequenter und strukturierter benutzt
 - AJAX-Requests unterstützen in allen Browsern alle http-Methoden
 - Das JSON-Format ermöglicht einen kompakten, Sprach-übergreifenden Datenaustausch

Beispiel: http-Aufrufe mit jQuery

```
loadPerson = function(url){
    jQuery.getJSON(url, callback);
};
deletePerson = function(url){
    jQuery.ajax({
        type: "DELETE",
        url: url,
        contentType: "application/json"
    }); };
createPerson = function(url, data){
    jQuery.ajax({
        type: "PUT",
        url: url,
        contentType: "application/json",
        data: data
    }); };
updatePerson = function(url, data){
    jQuery.ajax({
        type: "POST",
        url: url,
        contentType: "application/json",
        data: data
    }); };
};
```


2

SERVICES

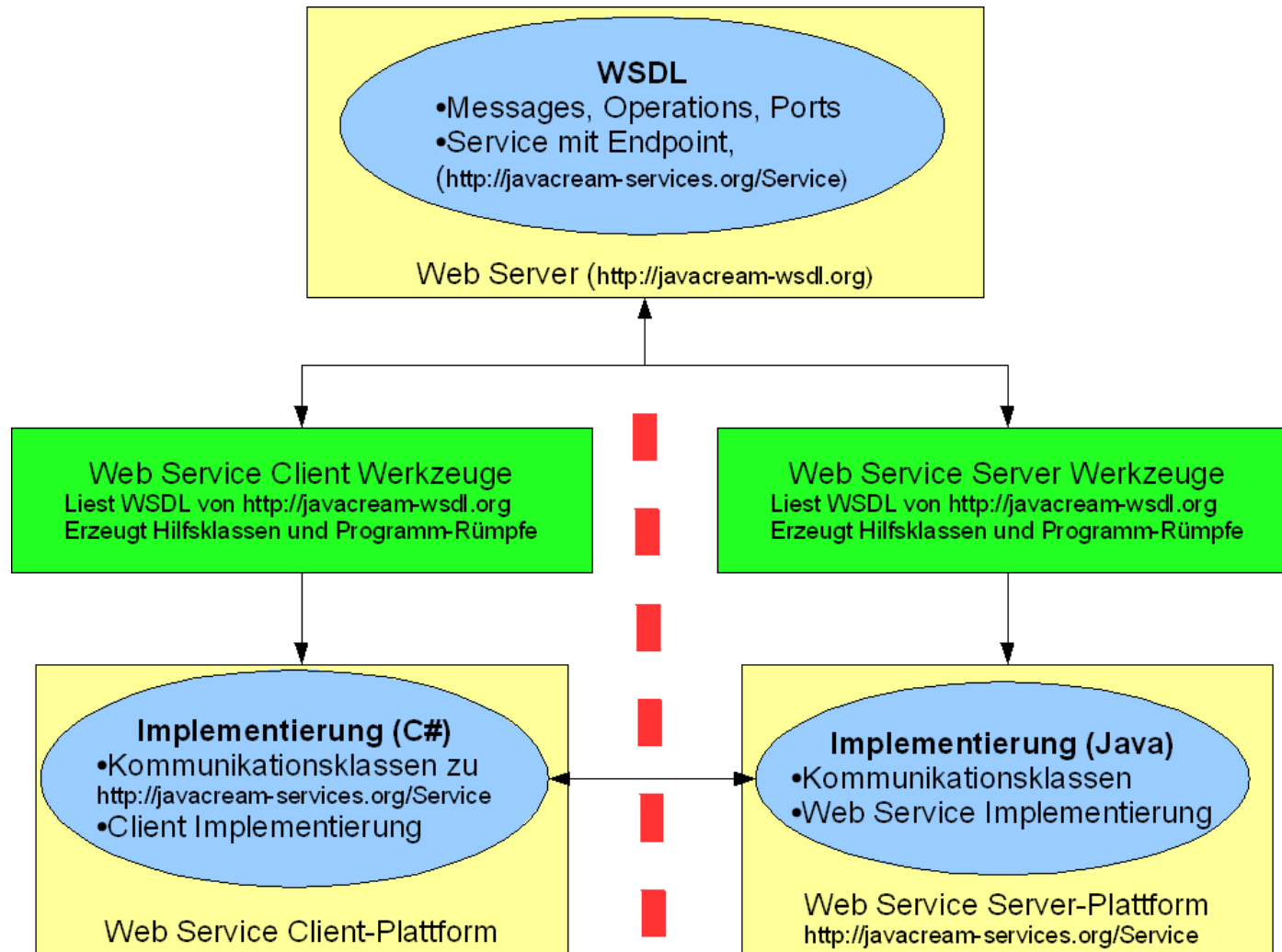
2.1

„KLASSISCHE“ WEB SERVICES MIT WSDL UND SOAP

- SOAP-basierte Web Services verlangen eine Schnittstellen-Beschreibung
 - Formulierung in der Web Services Description Language, WSDL
 - WSDL ist ein XML-Schema zur Beschreibung eines Interfaces bestehend aus
 - Typen
 - Operationen
 - Ports
 - Eine WSDL ist damit ein XML-Dokument
- Ein SOAP-Envelope definiert den Aufruf
 - SOAP ist ebenfalls ein XML-Schema
 - Das SOAP-Dokument wird an die Zieladresse des Web Services, den Endpoint gesendet

- Definiert wird eine Service-Beschreibung in Form einer WSDL
- Realisierung
 - Für die Server-Implementierung werden Code-Rümpfe erzeugt,
 - für den Client fertige Stub-Klassen
- Beim Aufruf werden die SOAP-Envelopes über das Netzwerk ausgetauscht

Beispiel eines SOAP-WebServices



Das Web interessiert sich nicht für SOAP!

- Keine einzige Browser-Implementierung erzeugt beispielsweise beim Formular-Submit einen SOAP-Envelope
- JavaScript könnte dies zwar prinzipiell, es gibt aber hierfür keine einzige ernst zu nehmende Utilities-Bibliothek
- Hardware und Software von Systemen sind auf http-Requests ausgerichtet
 - Firewalls
 - Load Balancer und Router
 - Cache-Lösungen

2.2

RESTFUL WEB SERVICES

- Ein RESTful Web Service benötigt in erster Näherung keine gesonderte Beschreibung
 - PUT, GET, POST, DELETE decken alle relevanten Ressourcen-Operationen ab
 - Datentypen werden durch MIME-Types bestimmt
- Die einzige notwendige Information ist die Endpoint-Adresse
- In der Praxis werden jedoch in den meisten Fällen zusätzliche Informationen benötigt
 - Ein GET auf den Endpoint liefert beispielsweise eine Liste aller Ressourcen
 - Aber:
 - Liste aller Informationen oder Referenzen?
 - Bestimmung der Sortierreihenfolge?
 - ...

- Analog zur WSDL wird eine detailliertere Service-Beschreibung geliefert
 - Web Application Description Language (WADL)
 - http://en.wikipedia.org/wiki/Web_Application_Description_Language
 - Rest Service Description Language (RSDL)
 - <http://en.wikipedia.org/wiki/RSDL>

- Hypertext as the Engine of Application State
 - <http://en.wikipedia.org/wiki/HATEOAS>
- Grundidee: Jeder Response enthält eine Menge von Verlinkungen, die die weiteren sinnvollen Aktionen definieren
 - „Hört sich an, wie eine Web Anwendung!“
 - Aber nun konsequent mit PUT, DELETE!

- Die WADL enthält analog zur WSDL eine detailliertere Service-Beschreibung
- http://en.wikipedia.org/wiki/Web_Application_Description_Language

2.3

REALISIERUNG MIT JAX-RS

- Ab der JEE 7 verpflichtender Teil der Java Enterprise Edition (JEE)
 - Ältere Versionen werden durch separate Bibliotheken unterstützt
- Die Semantik des RESTful Web Services wird auf Java-Methoden gemapped
 - Dazu definiert das JAX-RS-API vorwiegend Annotationen
 - Encoder und Decoder unterstützen die Umwandlung von/nach Java-Objekten
 - Eine JSON-Bibliothek ist Bestandteil der Standard-Bibliothek

[All Classes](#)

Packages

[javax.ws.rs](#)

[javax.ws.rs.client](#)

[javax.ws.rs.container](#)

[javax.ws.rs.core](#)

[javax.ws.rs.ext](#)

Annotation Types

[ApplicationPath](#)

[BeanParam](#)

[ConstrainedTo](#)

[Consumes](#)

[CookieParam](#)

[DefaultValue](#)

[DELETE](#)

[Encoded](#)

[FormParam](#)

[GET](#)

[HEAD](#)

[HeaderParam](#)

[HttpMethod](#)

The Java API for RESTful Web Services provides portable APIs for developing, exposing and accessing Web applications d compliance with principles of REST architectural style.

See:

[Description](#)

Packages

javax.ws.rs	High-level interfaces and annotations used to create RESTful service resources.
javax.ws.rs.client	The JAX-RS client API
javax.ws.rs.container	Container-specific JAX-RS API.
javax.ws.rs.core	Low-level interfaces and annotations used to create RESTful service resources.
javax.ws.rs.ext	APIs that provide extensions to the types supported by the JAX-RS API.

The Java API for RESTful Web Services provides portable APIs for developing, exposing and accessing Web applications d compliance with principles of REST architectural style.

Web resources

3

SERVER

3.1

COMPONENTS

- Als Komponenten werden annotierte Klassen identifiziert
 - `@Provider`
 - `@Path`
 - Implizit ebenfalls ein Provider
- Instanzen von Komponenten werden vom Context der JAX-RS-
Runtime verwaltet
 - Lifecycle
 - Dependency Injection
- Methoden von Components werden für einen RESTful-Request
aufgerufen
 - Im einfachsten Fall identifiziert JAX-RS exakt eine einzige Component
Method
 - Filter und Interceptors erweitern dies zu einer Sequenz von Aufrufen

- Prinzipiell unabhängig vom CDI-Framework
 - Eine Integration mit CDI ist aber nahtlos möglich
- Methoden von Komponenten können bei Bedarf annotierte Parameterlisten definieren, um Zugriff auf notwendige Referenzen zu bekommen
 - Request-Parameter
 - Informationen aus dem Context selbst
 - Dazu dient die `@Context`-Annotation
 - Annotiert wird ein Parameter oder ein Attribut
 - Gültige Typen sind beispielsweise
 - Der komplette `Request`
 - Die `HttpHeader`
 - `UriInfo`
 - Konfigurationsinformationen über `Configurable`

Code-Beispiel: Eine annotierte Java-Klasse

```
@Path("/people")
@Produces(MediaType.APPLICATION_JSON)
public class RestPeopleWebController {
    private PeopleController peopleController = new PeopleController();

    @GET
    @Path("/{id}")
    public Person findPerson( @PathParam("id") Long id) {
        return peopleController.findPersonById(id);
    }

    @GET
    public List<Person> findPeople() {
        return peopleController.findPeople();
    }

    @PUT
    @Path("/{id}")
    @Consumes(MediaType.APPLICATION_JSON)
    public void insertPerson(@PathParam("id") Long id, Person p){
        peopleController.insertPerson(id, p);
    }

    @POST
    @Path("/{id}")
    @Consumes(MediaType.APPLICATION_JSON)
    public void updatePerson(@PathParam("id") Long id, Person p){
        peopleController.updatePerson(id, p);
    }

    @DELETE
    @Path("/{id}")
    public void deletePersonById(@PathParam("id") Long id){
        peopleController.deletePersonById(id);
    }
}
```

3.2

DEPLOYMENT

- Das Deployment des RESTful Web Services erfolgt als Web-Anwendung
 - WAR-Datei mit der bekannten Verzeichnis-Struktur
 - WEB-INF
 - WEB-INF/classes
 - WEB-INF/lib
 - Konfigurationsmöglichkeiten über `web.xml`
 - Verwendung vorhandener Servlet-Filter-Implementierungen

- Eine Application-Implementierung definiert die zu verwendenden Services

```
@ApplicationPath("/rest")
public class RestApplication extends Application{
    private HashSet<Class<?>> classes;
    {
        classes = new HashSet<>();
        classes.add(RestBooksServiceFacade.class);
    }
    @Override
    public Set<Class<?>> getClasses() {
        return classes;
    }
}
```

- Mehrere Implementierungen gleichzeitig sind möglich

- Scannen nach Klassen mit Component-Annotationen
- Aktivierung des Scanners durch „leere“ `Application`

```
@ApplicationPath("/rs")
public class AnnotationScannerApplication extends
    Application {
    public Set<Class<?>> getClasses() {
        return Collections.emptySet();
    }
    @Override
    public Set<Object> getSingletons() {
        return Collections.emptySet();
    }
}
```

3.3

REQUEST-VERARBEITUNG

- Die http-Requests werden durch `@Path`-Annotationen auf Java-Signaturen gemappt
- Die verschiedenen Mappings müssen aber nicht disjunkt sein
 - Bei überlappenden Anweisungen wird ein komplexerer Algorithmus zur Bestimmung der aufzurufenden Methode durchlaufen
 - Prinzipiell: Das speziellste Mapping gewinnt
 - Details sind der JAX-RS-Spezifikation zu entnehmen

- `@PathParam`
- `@QueryParam`
- `@FormParam`
- `@CookieParam`
- `@HeaderParam`
- `@MatrixParam`
- `@BeanParam`
 - Hier wird eine Java-Bean-Klasse angegeben, deren Properties mit obigen Annotationen versehen sind
- `@DefaultValue`
 - Zusätzliche Annotation zum Setzen eines optionalen Default-Wertes

- Alle Informationen werden über annotierte Parameter via Dependency Injection gesetzt
 - `public String javaService(@QueryParam("order") String order, @PathParam Long id)`
- Bei Bedarf kann der Java-Aufruf aber auch noch als (zusätzlichen) Parameter eine Referenz auf den konkreten Request definieren
 - `javax.ws.core.Request`

3.4

CONTENT HANDLER

- Der Rückgabewert des Java-Aufrufs muss in den benötigten MIME-Type encodiert werden
 - Dies kann automatisch durch Content Handler erfolgen
 - Low-Level Content Handler sind Bestandteil der Spezifikation
- Der Content Type im Header des Responses wird durch die `@Produces`-Annotation gesetzt
 - In `javax.ws.rs.core.MediaType` sind statische Konstanten für gebräuchliche Types definiert
 - `MediaType.APPLICATION_JSON`
 - `MediaType.TEXT_PLAIN`
 - Die `Produces`-Annotation wird für das Auflösen der aufzurufenden Component-Methode berücksichtigt
 - Der Client sendet den http-Header `Accepts`

- Deklaration eines Parameters einer Component-Methode
 - Dieser wird injected
 - Eine Annotation dieses Parameters ist nicht notwendig
- Unterstützt werden:
 - `java.io.InputStream` und `java.io.Reader` zum direkten Lesen des Request Message Bodies
 - `java.io.File` als Rückgabe einer Datei
 - Der Dateityp ist dann natürlich als Content Type anzugeben
 - `byte[]`, `char[]` und `String` sind zum Lesen (Parameter) und Schreiben (Rückgabewert) erlaubt
 - `javax.xml.transform.Source` für Requests und/oder Responses im XML-Format

- `javax.ws.rs.core.StreamingOutput`
 - Direktes Streamen einer Antwort
 - Wrapper-Interface um einen `java.io.OutputStream`
public void `write(OutputStream output)` **throws**
`IOException, WebApplicationException`
 - Rückgabewert einer Component-Methode
- `javax.ws.rs.core.MultivaluedMap`
 - Für Formular-Eingaben

- Mit JAXB erfolgt ein Mapping von Java-Objekten nach XML und umgekehrt
- Ein Content Handler steht zur Verfügung, der als Rückgabebetyp Java-Klassen mit JAXB-Annotationen akzeptiert
 - Die Methode ist mit `@Produces(MediaType.APPLICATION_XML)` zu annotieren

- Die Erzeugung einer JSON-Antwort kann mit Hilfe des JSON-APIs in `javax.json` erfolgen
 - `JsonObject`
 - `JsonArray`
 - `JsonObjectBuilder` und `JsonArrayBuilder`
 - `JsonParser`
- Die `jackson`-Bibliothek enthält einen Content Handler, der automatisch ein Marshalling von Java-Objekten nach JSON und umgekehrt durchführt

- Eigene Content Handlers implementieren
 - `javax.ws.rs.ext.MessageBodyWriter`
 - `javax.ws.rs.ext.MessageBodyReader`
- Die implementierende Klasse wird als Component definiert und einem Content Type zugeordnet

```
@Provider
@Produces("application/xml")
public class MyJAXBMarshaller implements MessageBodyWriter{...}
```

oder

```
@Provider
@Consumes("application/xml")
public class MyJAXBUnmarshaller implements MessageBodyReader{...}
```

3.5

CONTENT NEGOTIATION

- Bisher:
 - Festlegung des Content Types der Antwort
- Erweiterung:
 - Wird auch beim Matching-Prozess zum Auflösen der aufzurufenden Java-Methode benutzt
 - Dazu wird natürlich der `Accept-Header` ausgewertet
 - Aktuell noch keine Annotation-basierte Unterstützung für `Accept-Language` und `Accept-Encoding`
 - Algorithmus analog zu `@Path`
 - Der speziellste Treffer gewinnt

- `@Produces` akzeptiert ein Array von Content Types
 - Content Handler können je nach aktuellem `Accept`-Header automatisch den angeforderten Typ liefern
- Komplexes Dispatching kann auch programmatisch erfolgen
 - Zugriff auf den http-Header durch Injection
 - ```
publicBook findBookByIsbn(@Context HttpHeaders headers,
 @PathParam("isbn")String isbn) {...}
```

- Empfohlene Namenskonvention:
  - `application/vnd.<company>.<type-name>`
    - `application.vnd.javacream.book`
- Ein „Parent“-Mechanismus wird unterstützt
  - `application/vnd.<company>.<type-name>+parent`
    - `application.vnd.javacream.book+xml`
- Zusätzliche Meta-Informationen durch key=value-Paare
  - `application/vnd.<company>.<type-name>+parent;key=value`
    - `application.vnd.javacream.book+xml;version=1.0`

- Browser senden ohne zusätzliche Plugins ausschließlich ihren vordefinierten `Accept`-Header
  - Damit ist eine Content Negotiation über Links und Formulare nicht automatisch möglich
- Lösungen
  - Senden der Informationen über Request-Parameter und dispatching im Programmcode
    - Path, Query
  - Die meisten Provider unterstützen Content Negotiation durch die „File“-Endung des Requests
    - Diese Endung wird vor der eigentlichen Verantwortung automatisch aus dem Request entfernt
    - Die Provider-Laufzeit enthält ein Mapping zwischen Endung und echtem MIME-Type

3.6

## **RESPONSE UND EXCEPTIONS**



- RESTful Web Services mappen den Status der Request-Verarbeitung auf die http-Status Codes
  - <http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html>
- Diese sind von der JAX-RS-Spezifikation direkt übernommen worden
  - und werden von den Component Methods automatisch dem Response zugeordnet

- Um eine Fehler-Situation signalisieren zu können enthält das API die `javax.ws.rs.WebApplicationException`
  - Eine Runtime-Exception
  - Erzeugung beispielsweise mit einem Status-Code-Wert
  - Vorsicht: Es liegt in der Verantwortung des Programmierers, nur Codes zu benutzen
    - Ein PUT darf nicht in der Antwort einen „404: Not Found“-Status setzen

- Zur Vereinfachung existiert eine ganze Hierarchie vordefinierter Exception-Klassen

| Exception                                 | Status code | Description                               |
|-------------------------------------------|-------------|-------------------------------------------|
| <code>BadRequestException</code>          | 400         | Malformed message                         |
| <code>UnauthorizedException</code>        | 401         | Authentication failure                    |
| <code>ForbiddenException</code>           | 403         | Not permitted to access                   |
| <code>NotFoundException</code>            | 404         | Couldn't find resource                    |
| <code>NotAllowedException</code>          | 405         | HTTP method not supported                 |
| <code>NotAcceptableException</code>       | 406         | Client media type requested not supported |
| <code>NotSupportedException</code>        | 415         | Client posted media type not supported    |
| <code>InternalServerErrorException</code> | 500         | General server error                      |
| <code>ServiceUnavailableException</code>  | 503         | Server is temporarily unavailable or busy |

- Mit Hilfe des `javax.ws.rs.core.ResponseBuilder`s kann der Programmierer den `javax.ws.rs.core.Response` feingranular erzeugen

```
public Response getBook() {
 String book = ...;
 ResponseBuilder builder = Response.ok(book);
 builder.language("fr")
 .header("Some-Header", "some value");
 return builder.build();
}
```

- Der `javax.ws.rs.core.Response` enthält
  - Das vom Content Handler zu konvertierende Java-Objekt
    - Die „Entity“
  - Den Statuscode
  - Die zu setzenden Header als `MultivaluedMap`

4

# CLIENT

4.1

## **CLIENT API**

- JAX-RS stellt ein generisches Client-API zur Verfügung
  - `Client client = ClientBuilder.newClient();`
- Mit Hilfe des `client` werden RESTful-Zugriffe ausgeführt
  - Fluentes API
  - `Response res = client.target("http://example.org/hello").request("text/plain").get();`
  - `WebTarget messages = client.target("http://example.org/messages/{id}");  
WebTarget msg123 = messages.resolveTemplate("id", 123);  
WebTarget msg456 = messages.resolveTemplate("id", 456);`

- `java.net.URL`
  - REST ist normales HTTP
- Apache HttpClient
  - Open Source Utility-Bibliothek zur vereinfachten Programmierung von HttpClient
  - Bestandteil der Java-Umgebung von Android-basierten Geräten
- RESTEasy
  - Open Source-Bibliothek von JBoss/RedHat mit extrem einfacher Client-Programmierung



4.2

## **FILTER UND INTERCEPTORS**

## ■ Filter

### ■ Querschnittsfunktionen, die den Zugriff auf Header-Informationen benötigen

- `javax.ws.rs.container.ContainerRequestFilter`
- `javax.ws.rs.container.ContainerResponseFilter`
- `javax.ws.rs.client.ClientRequestFilter`
- `javax.ws.rs.client.ClientResponseFilter`

### ■ Der Request-Filter ist in zwei Phasen aufgeteilt

- `javax.ws.rs.container.PreMatching-Annotation`
- Nicht-annotiert: „PostMatching“

## ■ Interceptors

### ■ Querschnittsfunktionen, die den Body verändern

- `javax.ws.rs.ext.ReaderInterceptor`
- `javax.ws.rs.ext.WriterInterceptor`

```
@PreMatching
public class DelaySimulationFilter implements ContainerRequestFilter {
 @Override
 public void filter(ContainerRequestContext containerRequestContext)
 throws IOException {
 String delayString = containerRequestContext.getUriInfo()
 .getQueryParameters().getFirst("delay");
 System.out.println("Delaying request, delay=" + delayString);
 if (delayString != null) {
 long delay = Long.parseLong(delayString);
 try {
 Thread.sleep(delay);
 } catch (InterruptedException e) {
 e.printStackTrace();
 }
 }
 }
}
```

```
@Provider
public class GZIPEncoder implements WriterInterceptor
{
 public void aroundWriteTo(WriterInterceptorContext
 ctx)
 throws IOException, WebApplicationException {
 GZIPOutputStream os = new
 GZIPOutputStream(ctx.getOutputStream());
 ctx.getHeaders().putSingle("Content-Encoding",
 "gzip");
 ctx.setOutputStream(os);
 ctx.proceed();
 }
}
```

- `javax.annotation.Priority`-Annotation
  - `@Provider @Priority(42)`
- Einige Konstanten definiert in `javax.ws.rs.Priorities`
  - `@Priority(Priorities.AUTHENTICATION)`

- Anwendung eines Filters/Interceptors pro Methode
- Definition einer `@NameBinding`-Annotation

```
@NameBinding
@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface MyBinding {}
```
- Benutzung der Annotation auf Klassen- oder Methodenebene

- Programmatische Konfiguration des JAX-RS-Kontextes

@Provider

```
public class MyFeature implements DynamicFeature {
 public void configure(ResourceInfo ri, FeatureContext ctx) {
 MyAnnotation ann =
 ri.getResourceMethod().getAnnotation(MyAnnotationclass
);
 if (ann == null) return;
 MyFilter filter = new MyFilter();
 ctx.register(filter);
 }
}
```

- MyFilter ist kein @Provider!

4.3

## **ASYNCHRONE AUFRUFE**



- Das Client-API unterstützt Aufrufe unter Verwendung von `java.util.concurrent` mit `Future<T>`

```
Client client = ClientBuilder.newClient();
Future<Response> futureResponse =
 client.target(uri).request().async().get();
//...
Response response = futureResponse.get();
//...
```

- **Dazu dient das Interface**

```
javax.ws.rs.client.InvocationCallback
public class MyCallback implements
 InvocationCallback<Response> {
 public void completed(Response response) {
 //...
 }
 public void failed(Throwable throwable) {
 //...
 }
}
```

- **Benutzung**

```
client.target(uri).request().async().get(new
 MyCallback());
```

- Auch der Server kann die Verarbeitung asynchron durchführen
- Dies muss der Programmierer vorsehen:
  - void-Methode
  - Injection eines Parameter vom Typ `javax.ws.rs.container.AsyncResponse`
    - `public void myService( @Suspended AsyncResponse asyncResponse)`
  - **Benutzung des AsyncResponse in separatem Thread**

```
new Thread() {
 public void run() {
 asyncResponse.resume(result);
 }
}.start();
```

- Ein AsyncResponse kann abgebrochen werden
  - `cancel()`
- Timeout
  - `setTimeout(timeValue, timeUnit)`
- Registrierung von Callback-Handlern
  - `javax.ws.rs.container.TimeoutHandler`
  - `javax.ws.rs.container.CompletionCallback`
  - `javax.ws.rs.container.ConnectionCallback`

5

## **WEITERE THEMEN**

5.1

## **HATEOAS**

- Einbetten von Links in der Antwortstruktur
  - XML
  - JSON
- Die Antwort enthält damit
  - die Ergebnisse sowie
  - die nächsten sinnvollen Aktionen

- Standard für Links in XML
  - `<link href= "..." hreflang= "..." rel= "..." type= "..." />`
  - **href**
    - Ziel der Verlinkung
  - **rel:**
    - Logische Bedeutung der Verlinkung als beliebige Zeichenkette
      - next, previous, edit
  - **type**
    - Media Type
  - **hreflang**
    - Sprache



- Alternative zu den Atom Links
- Link-Header
  - Link:  
`<http://resource.url;rel=next,type=application/xml`

- `javax.ws.rs.core.UriBuilder`
  - Hilfsklasse zur Erzeugung von URIs
    - Absolut
    - Relativ
    - Mit Query-Parametern
    - Mit Templates
- `javax.ws.rs.core.Link`
  - Hilfsklasse zur Erzeugung von Atom Links und Link-Headern

5.2

## **SECURITY**

- Anbindung an ein Benutzerverwaltungssystem erfolgt administrativ
  - Konfiguration von „Realm“
- Dann stehen zur Verfügung
  - Security Constraints der `web.xml`
    - Authentication
    - Encryption
    - Authorization
  - Security Annotations
    - `@RolesAllowed`
    - `@PermitAll`
  - Authentication Schemas
    - BASIC
    - DIGEST
    - CLIENT\_CERT
    - FORM

- `javax.ws.rs.core.SecurityContext` mit Zugriff auf
  - Principal
  - Rolle des authentifizierten Benutzers
  - Das Authentifizierungsschema
- Injection via `@Context`-Annotation

- Der `ClientBuilder` enthält folgende relevanten Methoden für eine SSL-Verbindung
  - `keyStore(final KeyStore keyStore, final String password)`
    - Der `KeyStore` enthält das Client-Zertifikat
  - `keyStore(final KeyStore keyStore, final String password)`
    - Der `KeyStore` enthält den Schlüssel zur Verifizierung des Server-Zertifikats
- Provider können zusätzliche `RequestFilter` zur Verfügung stellen
  - Diese werden beim Client mit der Methode `register(...)` angemeldet
  - Beispiel
    - `org.jboss.resteasy.client.jaxrs.BasicAuthentication`

5.3

## **CACHING**

- Proxy-Server versuchen aus Performance-Gründen Aufrufe zu cachen
  - Expires-Header
    - Kann über den Response bzw. im ResponseBuilder gesetzt werden
  - Ab HTTP 1.1 Cache-Control mit reichhaltigen Properties
    - Klasse `javax.ws.rs.core.CacheControl`
    - Kann über den Response bzw. im ResponseBuilder gesetzt werden



- Conditional GET
  - Last-Modified-Header in der Antwort des Servers
  - GET-Anfrage mit If-Modified-Since-Header
    - Bei Änderung „200: OK“ mit geändertem Inhalt
    - Sonst: „304: Not modified“
- JAX-RS
  - Auswertung im `javax.ws.rs.core.Request`

- ETag
  - Hash-Wert, der vom Server eindeutig aus dem Inhalt des Responses berechnet wird
  - ETag-Header in der Antwort des Servers
  - GET-Anfrage mit `If-None-match-Header`
    - Bei Änderung „200: OK“ mit geändertem Inhalt
    - Sonst: „304: Not modified“
- JAX-RS
  - `javax.ws.rs.core.EntityTag`
  - Auswertung im `javax.ws.rs.core.Request`

- Vermeidung inkonsistenter Zustände durch konkurrierende Client-Zugriffe
- Header-Property `If-Unmodified-Since`
  - Im Fehlerfall „412: Precondition Failed“
- Auswertung in JAX-RS ebenfalls wieder über den `Request`

5.4

## **WERKZEUGE UND PROVIDER**

- Arbeitsweise und Umfang der Generatoren ist nicht Bestandteil der JAX RS-Spezifikation
- Prinzipiell kann erzeugt werden:
  - WADL aus annotierter Java-Klasse
  - Code-Rümpfe für Server-Implementierung aus WADL
  - Client-Stubs aus WADL
  - Client-Stubs aus annotierter Java-Klasse

- Die Annotationen werden durch eine Laufzeitumgebung ausgewertet
  - Diese wird von einem Provider bereit gestellt
    - Jersey
    - RestEasy
    - Apache CXF
  - Ab der JEE 7 ist JAX-RS verpflichtender Bestandteil eines Applikationsservers



RESTful Web Services in Java.

## About

Developing RESTful Web services that seamlessly support exposing your data in a variety of representation media types and abstract away the low-level details of the client-server communication is not an easy task without a good toolkit. In order to simplify development of RESTful Web services and their clients in Java, a standard and portable [JAX-RS API](#) has been designed. Jersey RESTful Web Services framework is open source, production quality, framework for developing RESTful Web Services

- Ein von JBoss/RedHat implementierter JAX-RS-Stack
- Insbesondere interessant das Client-API
  - Hier wird einfach das Konzept der Annotationen auf Server-Seite für den Client wiederverwendet
    - Annotiert wird ein Interface
    - Eine Stub-Implementierung wird von RESTEasy dynamisch generiert