Node.js: REST Web-Services

- Begriff Web-Service und dessen Eigenschaften verstehen
- Verschiedene Anwendungsszenarien begreifen
- Eigenschaften von REST verstehen
- Einen REST-Web-Service auf einem Node-Server zur Verfügung stellen können
- Daten von einem Web-Service beziehen aber auch Daten an ihn schicken, ändern und löschen können
- HTTP-Methoden und HTTP-Response Statuscodes richtig einsetzen können
- Als Übertragungsformat neben JSON auch XML verwenden können
- Auf den Web-Service über Java, PHP und jQuery zugreifen können
- Mit dem Cross-Origin Resource Sharing (CORS) auf einem Node-Server umgehen können

Web-Services sind Dienste die ein Web-Server für Clients anbietet:

- Abrufen von Ressourcen (Dateien, Objekte)
- Aufrufen entfernter Methoden (mit Parameter) mit Rückgabe
- Instanziieren von Objekten am Server, die Zustände und Verhalten haben und die mit Client kommunizieren
- Auch Kommunikation zw. Servern wird realisiert

Eigenschaften

- Kommunikation erfolgt über normales HTTP-Protokoll (kein zusätzlicher Port notwendig)
- Plattformübergreifend

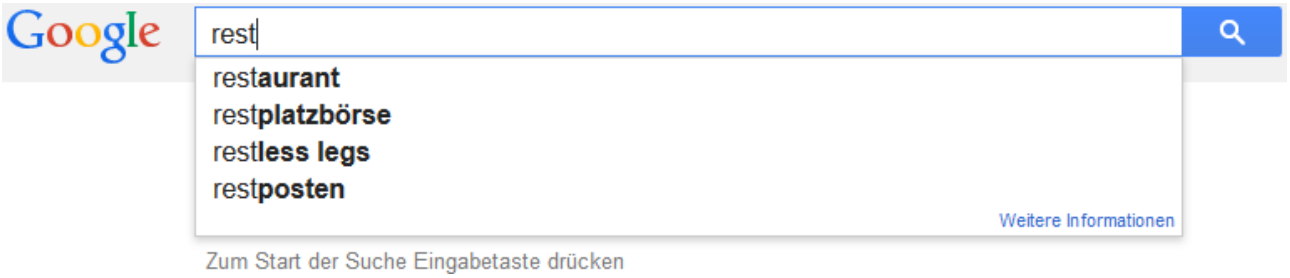
SOAP (*Simple Object Access Protocol*)

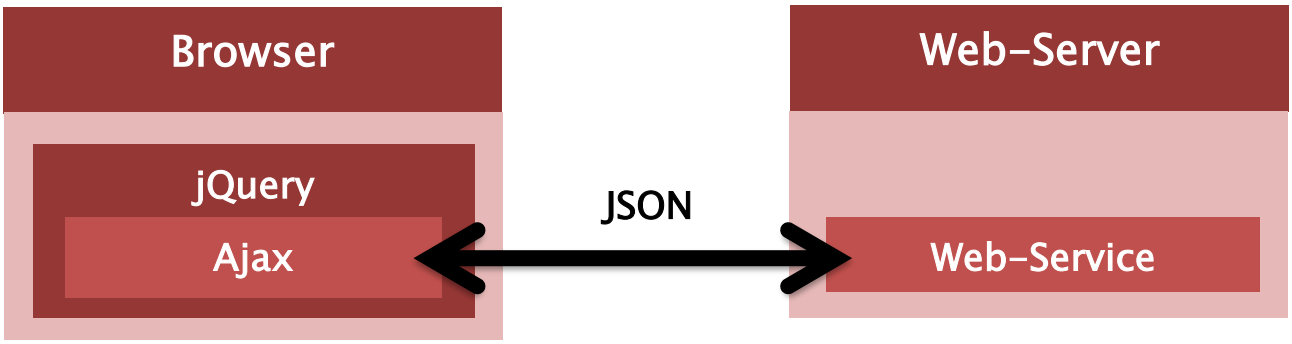
- Überträgt XML-Nachrichten
- Ermöglicht entfernte Methodenaufrufe
- Parameter und Rückgaben exakt im XML-Format beschrieben
- Generatoren erstellen Zugriffsklassen für unterschiedliche Programmiersprachen
- Komplex aber mächtig

REST (*Representational State Transfer*)

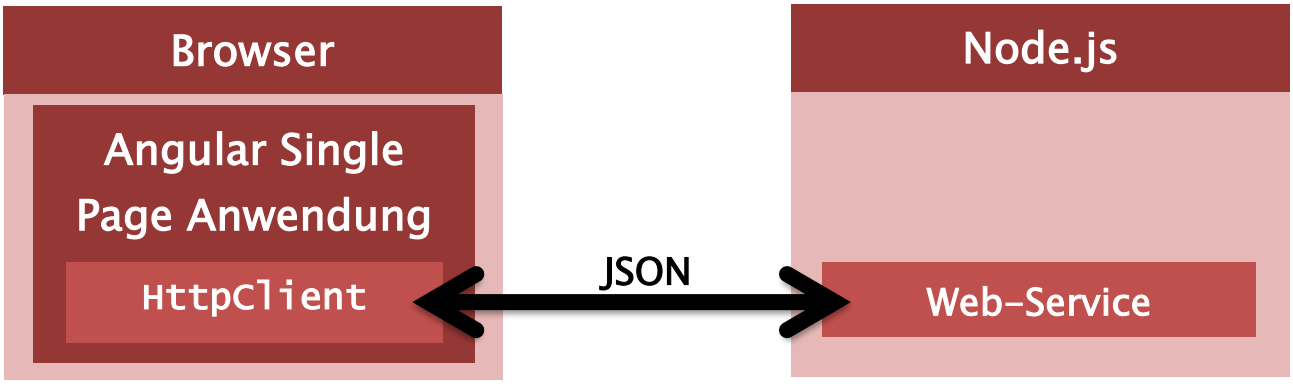
- Anfrage wird über HTTP verschickt
- URL bestimmt die angefragte Ressource
- URL enthält codierte Parameter
- Nur wenige Operationen möglich (GET, POST, PUT, DELETE)
- Ressource hat beliebiges Format (HTML, Text, XML, JSON, Bild, MP3, usw.)
- Einfach aber nicht so flexibel

Anwendungsszenarien

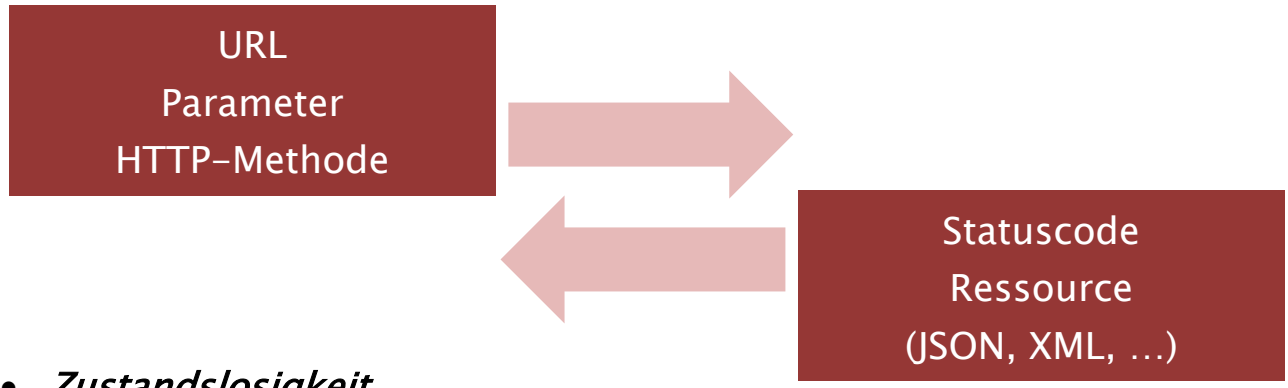




- Einsatz von **JavaScript** ermöglicht Absetzen von Anfragen und Anzeigen von Ergebnissen ohne komplett neuen Seitenaufbau
- **jQuery** JavaScript-Bibliothek ermöglicht komfortable DOM-Navigation und -Manipulation
- **Ajax** (*Asynchronous JavaScript and XML*) ermöglicht HTTP-Anfragen zw. Browser und Server ohne Seite komplett neu zu laden. JQuery verfügt über Ajax-Schnittstelle



REST Funktionsweise und Eigenschaften

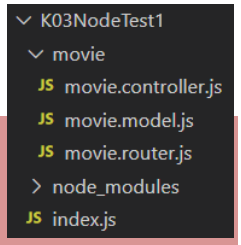


- *Zustandslosigkeit*
  Jede Anfrage muss sämtliche erforderlichen Informationen enthalten
- *Ressourcen*
  Jede Ressource ist über eindeutigen URL abrufbar
  z.B. `http://localhost:8080/movie/1`
- *HATEOAS (Abk. Hypermedia as the Engine of Application State)*
  Über versch. Links wird dem Konsumenten des Service mitgeteilt, welche Zustandsänderungen mit der angeforderten Ressource noch möglich sind (darauf wird hier verzichtet)

TIPP: Zum Testen der Schnittstellen des Web-Services dienen *Postman* (grafische Oberfläche) (`https://www.postman.com`) oder *cURL*[1] (Kommandozeilentool) (`https://curl.haxx.se`)

[1] cURL wird auch weiter hinten verwendet um mit PHP auf einen Web-Service zuzugreifen

Der Server mit seinen Bestandteilen



index.js

```
const express = require('express');
const app = express();

const bodyParser = require('body-parser');
app.use(bodyParser.json()[2]);

const movieRouter = require('./movie/movie.router');
app.use('/movie', movieRouter);

app.get('/', (request, response) => response.redirect('/movie'));

app.listen(8080, () =>
  console.log('Web-Service listen on port 8080'));
```

movie.router.js

```
const express = require('express');
const router = express.Router();

const { listAction, viewAction } = require('./movie.controller');

router.get('/', listAction);[3]
router.get('/:id', viewAction);

module.exports = router;
```

Mögliche Routen `/movie`, `/movie?sort=asc|desc`, `/movie/1`

movie.controller.js

```
const movieModel = require('./movie.model');

function listAction(request, response) {

  const sort = request.query.sort ? request.query.sort : '';

  movieModel.getAll(sort, 'sepp')
    .then(movies => response.json(movies))
    .catch(error => response.status(
      error === 'Database error' ? 500 : 400).json(error));
}

function viewAction(request, response) {

  movieModel.get(request.params.id, 'sepp')
    .then(movie => response.json(movie))
    .catch(error => response.status(
      error === 'Database error' ? 500 : 400).json(error));
}

module.exports = { listAction, viewAction };
```

`status()`	Setzen des richtigen Response Statuscodes (siehe hinten)
`json()`	Codieren des Ergebnisses nach JSON und schickt Statuscode 200

---

[2] Dadurch können über POST JSON-Objekte übertragen werden (siehe hinten)

[3] **ACHTUNG:** Reihenfolge der Routen im Router ist entscheidend insbesondere bei der Authentifizierung (siehe Übung)

movie.model.js

```
…
async function getAll(sort = null, username = null) {
  const sql =`
    SELECT m.id, title,year,published,CONCAT(…) as fullname,
      u.username AS owner
    FROM movies m, users u
    WHERE m.owner = u.id
      ${username ? 'AND (u.username = ? OR published = true)' :
      'AND published = true'}
    ORDER BY title ${!sort || sort === 'asc' ? 'ASC' : 'DESC'};
  `;
  const database = new Database(connectionProperties);
  try {
    const result = await database.queryClose(sql, [ username ]);
    return result.length === 0 ?
      Promise.reject('No movies found') : Promise.resolve(result);
  } catch (error) {
    return Promise.reject('Database error');
  }
}
async function get(id, username) {
  if (!username) {
    return Promise.reject('User not set');
  } else {
    try {
      const database = new Database(connectionProperties);
      const sql = ` … `;
      const result=await database.queryClose(sql, [id, username]);
      if (result.length === 0) {
        return Promise.reject('Movie not found');
      } else {
        return Promise.resolve(result[0]);
      }
    } catch (error) {
      return Promise.reject("Database error");
    }
  }
}
```

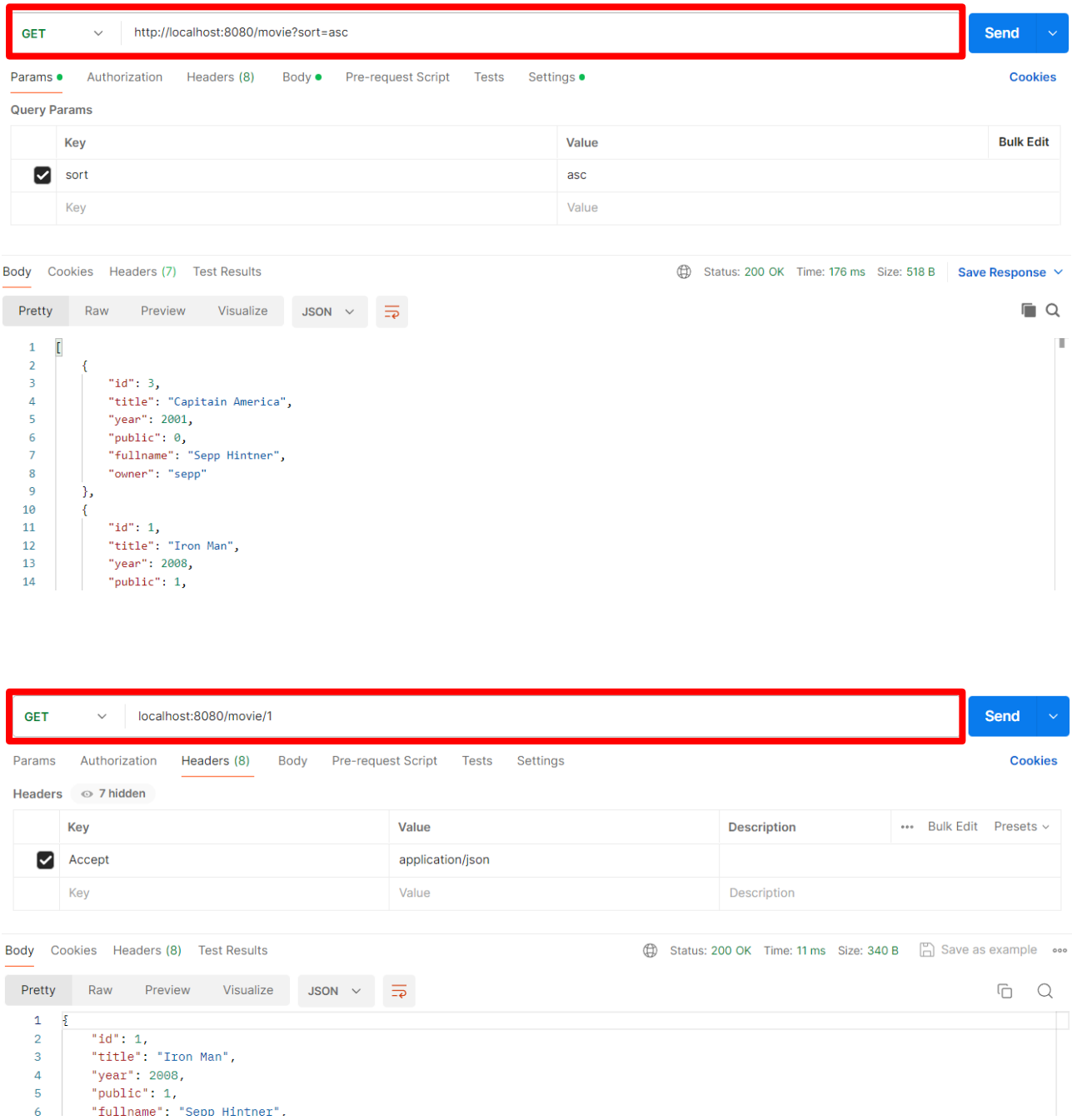
Route `/movie` mit Methode `getAll()` liefert

- Liste der Filme ➔ 200
- Fehler `'Database error'` falls DBMS nicht antwortet ➔ 500
- Fehler `'No movies found'` falls keine Filme gefunden werden konnten ➔ 400

Route `/movie/:id` mit Methode `get()` liefert

- Den gefundenen Film ➔ 200
- Fehler `'Database error'` falls DBMS nicht antwortet ➔ 500
- Fehler `'User not set'` oder `'Movie not found'` ➔ 400

Testen mit Postman



Was mit Ressource passieren soll wird über **HTTP-Methode** definiert:

HTTP-Methode	Operation	Vergleichbar mit
POST	Legt neue Ressource an	INSERT
PUT	Aktualisiert Ressource	UPDATE
DELETE	Löscht Ressource oder Sammlung von Ressourcen	DELETE
GET	Listet Ressourcen auf oder holt konkrete Ressource	SELECT

Ergebnisrückgabe anhand von

- *HTTP-Response Statuscode* liefert Erfolg oder Fehler[4]

```
200   OK
201   Created
202   Accepted

400   Bad Request
401   Unauthorized
404   Not Found
406   Not Acceptable
409   Conflict

500   Internal Server  Error
501   Not Implemented
…
```

- *und Objekte* im JSON- oder XML-Format

Ändern des Ausgabeformates auf XML

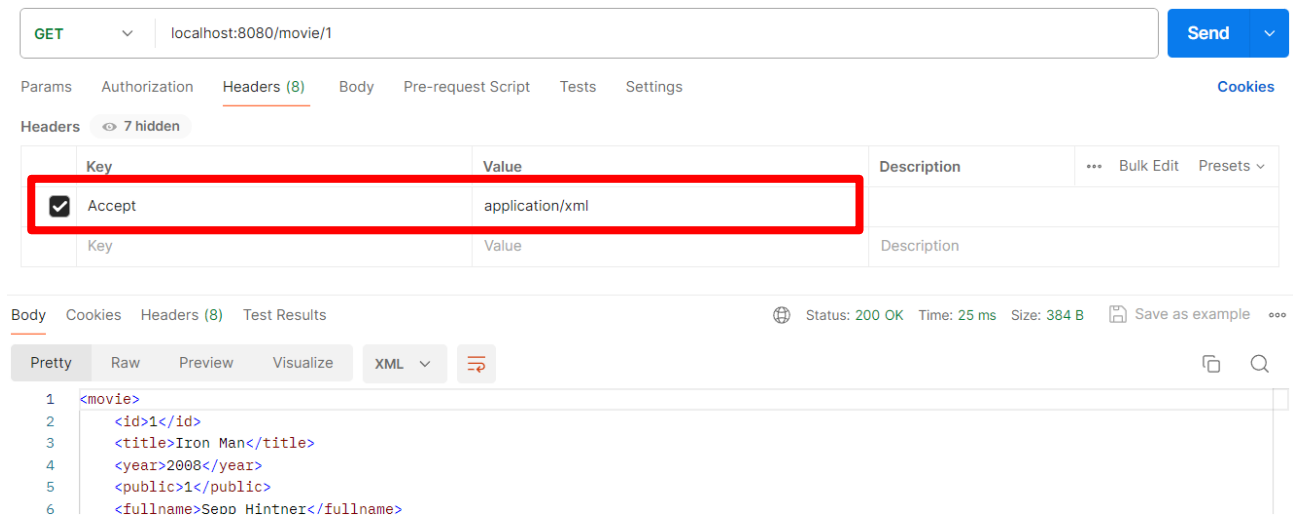
Client fordert durch Setzen des Headers `Accept application/xml` gewünschtes Format an

```
$ npm install jsontoxml
```

[4] In Spezifikation RFC 7231 definiert

movie.controller.js

```
…
const jsonXml = require('jsontoxml');
function listAction(request, response) {
  const sort = request.query.sort ? request.query.sort : '';
  movieModel.getAll(sort, 'sepp')
    .then(movies => response.format({
      'application/xml': () => {
        movies = movies.map(movie => ({ movie, }))[5];
        response.send(`<movies>${jsonXml(movies)}</movies>`[6])
      },
      'application/json': () => response.json(movies),
      'default': () =>  response.json(movies)
      }))
    .catch(error => response.format({
      'application/xml': () =>
        response.status(
          error==='Database error'?500:400).send(error),
      'application/json': () =>
        response.status(
          error === 'Database error' ? 500 : 400).json(error),
      'default': () =>
        response.status(
          error === 'Database error' ? 500 : 400).json(error)
      }));
}
module.exports = { listAction };
```



[5] Überführen in die Struktur { movie: { … } }, { movie: { … } }

[6] Überführen in die Struktur <movies><movie>…</movie><movie>…</movie></movies>

Definition des Web-Services (für Übung 1)

Methode	Route	Statuscode/Error
GET	/movie /movie?sort=asc /movie?sort=desc	200 liefert Liste der Filme 400 No movies found 500 Database error
GET	/movie/:id	200 liefert gefundenen Film 400 User not set, Movie not found 500 Database error
POST	/movie	200 liefert eingetragenen Film mit neuer id und fullname 400 User not set, User not found, Title exists 500 Database error
PUT	/movie/:id	200 liefert geänderten Film 400 User not set, User not found, Title exists, Movie exists, Movie not found 500 Database error
DELETE	/movie/clear	200 Leerer Body bei Erfolg 400 User not set, Movies not found 500 Database error
DELETE	/movie/:id	200 Leerer Body bei Erfolg 400 User not set, Movie not found 500 Database error

Hinzufügen/Ändern eines neuen Films (JSON oder XML) über POST

```
{
  "title": "Troy",
  "year": "2004",
  "published": "false",
  "owner": "sepp"[7]
}
```

```
<movie>
  <title>Troy</title>
  <year>2004</year>
  <published>false</published>
  <owner>sepp</owner>
</movie>
```

ANNAHMEN: Eigenschaften werden als Zeichenketten übertragen

[7] **ACHTUNG**: Als owner wird der Benutzername übergeben und nicht die Id des Benutzers

```
npm install express-xml-bodyparser
```

Dadurch können über POST auch Daten im XML-Format übertragen werden

index.js

```
…
const xmlparser = require('express-xml-bodyparser');
app.use(xmlparser({ explicitRoot: false }));
```

`explicitRoot` Root-Element (<movie>) wird entfernt

movie.router.js

```
…
router.post('/', insertAction);
router.put('/:id', updateAction);
…
```

movie.controller.js

```
function insertAction(request, response) {
  const movie = {
    id: -1,
    title: request.body.title,
    year: parseInt(request.body.year, 10),
    published: request.body.published === "true" ? true : false,
    owner: request.body.owner
  };
  movieModel.insert(movie, 'sepp')
    .then(movie => response.format({ … }))
    .catch(error => response.format({ … }));
}
function updateAction(request, response) {
  const id = parseInt(request.params.id, 10);
  const movie = { … };
  movieModel.update(id, movie, 'sepp')
    .then(movie => response.format({ … }))
    .catch(error => response.format({ … }));
}
```

Zugriff auf den Web-Service über Java

Anlegen eines Maven-Projektes und Ergänzen von pom.xml

```
<dependencies>
  <dependency>
    <groupId>org.glassfish.jersey.core</groupId>
    <artifactId>jersey-client</artifactId>
    <version>2.19</version>
  </dependency>
  <dependency>
    <groupId>org.glassfish.jersey.media</groupId>
    <artifactId>jersey-media-json-jackson</artifactId>
    <version>2.17</version>
  </dependency>
  <dependency>
    <groupId>org.glassfish.jersey.core</groupId>
    <artifactId>jersey-server</artifactId>
    <version>2.19</version>
  </dependency>
    <dependency>
    <groupId>javax.xml.bind</groupId>
    <artifactId>jaxb-api</artifactId>
    <version>2.3.0</version>
  </dependency>
  <dependency>
    <groupId>com.sun.xml.bind</groupId>
    <artifactId>jaxb-core</artifactId>
    <version>2.3.0</version>
  </dependency>
  <dependency>
    <groupId>com.sun.xml.bind</groupId>
    <artifactId>jaxb-impl</artifactId>
    <version>2.3.0</version>
  </dependency>
</dependencies>
```

JAX-RS (*Java API for RESTful Web Services*)

Java API zur Deklaration von REST-basierten Web-Services. Ihre Referenzimplementierung ist **Jersey**

Jackson dient zum Umwandeln von JSON und XML in Java-Objekte

Mit **jaxb** wird Klasse mit `@XmlRootElement` annotiert (siehe hinten)

Erstellen der Movie-Klasse:

```
@XmlRootElement
public class Movie {

  protected Integer id = -1;
  protected String title = null;
  …
  public Movie() { }

  // Es folgen die Getter- und Setter-Methoden

  public String toString() {
    return id + ", " + title + ", " + year + ", " +
      published + ", " + fullname + ", " + owner,
  }
}
```

Klasse muss Defaultkonstruktor besitzen

GET mit Parametern

```
try {
  List<Movie> movies = ClientBuilder.newClient()
    .target("http://localhost:8080")
    .path("movie")
    .queryParam("sort", "desc")
    .request()
    .accept(MediaType.APPLICATION_XML)                    // Optional
    .get(new GenericType<List<Movie>>() {});

  for(Movie m: movies)
    System.out.println(m);

} catch (ClientErrorException e) {
  System.out.println(e.getClass().getCanonicalName());
  System.out.println(e.getMessage());
  System.out.println(e.getResponse().getStatus());
  System.out.println(e.getResponse().readEntity(String.class));
}
```

GET mit variablem URL-Teil

```
Movie movie = ClientBuilder.newClient()
  .target("http://localhost:8080")
  .path("movie/{id}")
  .resolveTemplate("id", 1)
  .request()
  .get(new GenericType<Movie>() {});
```

POST

```
Movie movie = new Movie();
movie.setTitle("Troy"); movie.setYear(2004);
movie.setPublished(false); movie.setOwner("sepp");

Movie ret = ClientBuilder.newClient()
  .target("http://localhost:8080")
  .path("movie")
  .request()
  .post(Entity.entity(movie,MediaType.APPLICATION_JSON),
    new GenericType<Movie>() {});                        // Rückgabetyp
```

PUT

```
movie.setTitle("Troy 2");

Movie ret = ClientBuilder.newClient()
  .target("http://localhost:8080")
  .path("movie/{id}")
  .resolveTemplate("id", 18)
  .request()
  .put(Entity.entity(movie,MediaType.APPLICATION_JSON),
    new GenericType<Movie>() {});
```

DELETE

```
Response response = ClientBuilder.newClient()
  .target("http://localhost:8080")
  .path("movie/{id}")
  .resolveTemplate("id", 1)
  .request()
  .delete(Response.class);

System.out.println(response);
```

```
InboundJaxrsResponse
{context=ClientResponse{method=DELETE,
uri=http://localhost:8080/movie/18,
status=200, reason=OK}}
```

Zugriff auf den Web-Service über PHP und cURL über JSON

cURL (Abk. Client for URLs) ist Programmbibliothek zur Übertragung von Dateien in Rechnernetzen. In PHP standardmäßig vorhanden

Voraussetzung JSON serialisierbares Objekt

```
<?php
class Movie implements JsonSerializable
{
  private $id = 0;
  private $title = null;
  …
  public function __construct($data = null) {
    $this->id = $data["id"];
    $this->title = $data["title"];
    …
  }
  // Es folgen die Getter- und Setter-Methoden
  public function jsonSerialize() {
    return [
      "id" => $this->id,
      "title" => $this->title,
      …
    ];
  }
  public function __toString() {
    return $this->id . ", " . …;
}
```

GET

```
$ch = curl_init();
if ($ch) {
  curl_setopt($ch, CURLOPT_URL, "http://localhost:8081[8]/movie");
  curl_setopt($ch,CURLOPT_HTTPHEADER,["Accept:application/json"]);
  curl_setopt ( $ch, CURLOPT_RETURNTRANSFER, true);
  $res = curl_exec ($ch);
  if (curl_getinfo($ch, CURLINFO_RESPONSE_CODE) != 200)
    echo $res; // Ausgabe der Fehlermeldung
  else {
    $arr = json_decode($res, true);
    foreach ($arr as $value)
      echo new Movie($value);
  }
  curl_close($ch);
}
```

CURLOPT_RETURNTRANSFER
Response wird als String zurückgegeben und nicht direkt ausgegeben

json_decode()
JSON-String wird in ein assoziatives PHP-Array umgewandelt:

{ "id":1, "title":"Iron Man", … } in

["id" =>1, "title" => "Iron Man", …]

POST

```
$movie = new Movie([
  "id" => -1, "title" => "Troy", "year" => 2004,
  "published" => false, "owner" => "sepp", "fullname" => null
]);
$ch = curl_init();
if ($ch) {
  curl_setopt($ch, CURLOPT_URL, "http://localhost:8081/movie");
  curl_setopt($ch, CURLOPT_CUSTOMREQUEST, "POST");
  curl_setopt($ch, CURLOPT_HTTPHEADER, [
    "Content-Type: application/json",
    "Accept: application/json"
  ]);
  curl_setopt($ch, CURLOPT_POSTFIELDS,
    json_encode($movie->jsonSerialize()));
  curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
  $res = curl_exec($ch);
  …
}
```

Nur POST-, PUT- und DELETE-Requests müssen mit
CURLOPT_CUSTOMREQUEST übertragen werden

[8] Auf Port 8080 läuft PHP-Web-Server

PUT

```
…
curl_setopt($ch, CURLOPT_URL,
  "http://localhost:8081/movie/" . $movie->getId());
curl_setopt($ch, CURLOPT_CUSTOMREQUEST, "PUT");
…
```

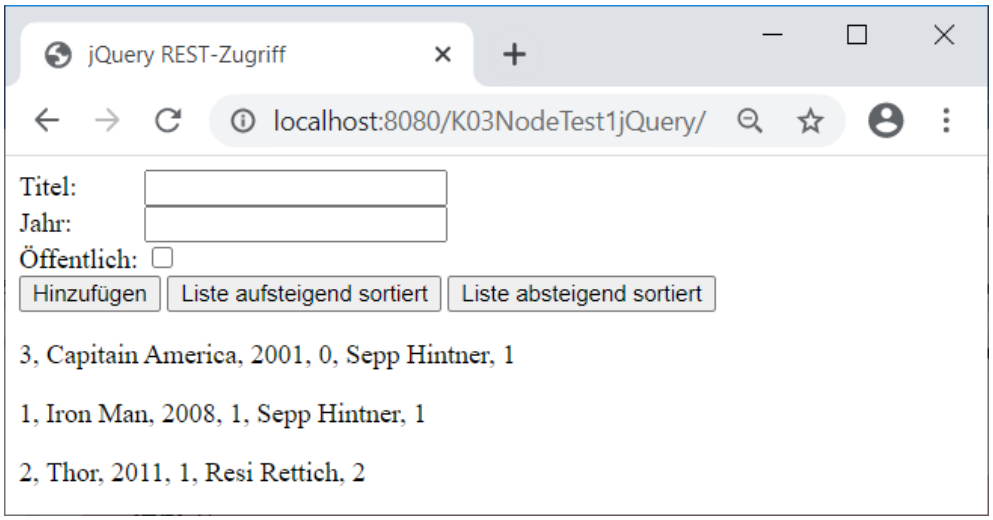
DELETE

```
…
curl_setopt($ch, CURLOPT_URL,
  "http://localhost:8081/movie/" . $movie->getId());
curl_setopt($ch, CURLOPT_CUSTOMREQUEST, "DELETE");
…
```

Zugriff auf Web-Service über jQuery

HTML-Gerüst

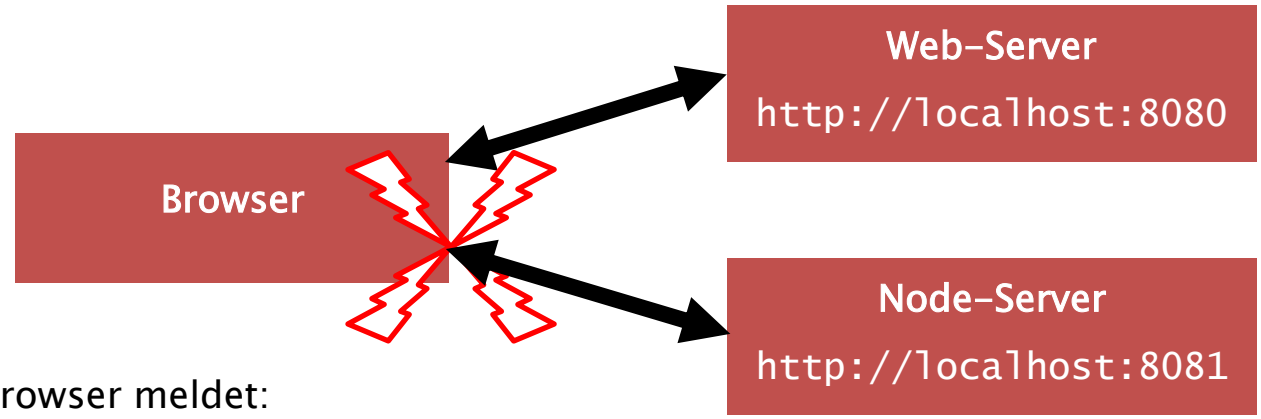
```
<div>
  <label for="title">Titel:</label>
  <input id="title" type="text">
</div>
<div>
  <label for="year">Jahr:</label>
  <input id="year" type="text">
</div>
<div>
  <label for="published">Öffentlich:</label>
  <input id="published" type="checkbox">
</div>
<button id="insert" type="button">Hinzufügen</button>
<button id="asc" type="button">Liste aufsteigend sortiert</button>
<button id="desc" type="button">Liste absteigend sortiert</button>
<div id="output"></div>
```



JavaScript- mit jQuery-Code

```
<script src="lib/jquery-3.7.1.min.js"></script>
<script>
class Movie {
  constructor(id = null, title = null, year = null, …) {
    this.id = id; this.title = title; this.year = year; …
  }
  toString() {
    return this.id + ", " + this.title + ", " + this.year + …
  }
}
//Nachdem HTML-Dokument vollständig geladen wurde startet read()
$(document).ready(function() {
  $("#asc, #desc").click(function() {
    $("#output").empty();
    $.ajax({
      url: "http://localhost:8081/movie?sort="+$(this).attr("id"),
      type: "GET",
      // Erwarteter Rückgabetyp
      dataType: "json",
      success: function(data) {
        $.each(data, function(i, data1) {
          $("#output").append($("<p>").append(
            Object.assign(new Movie(), data1).toString()));
        });
      },
      error: function(error) {
        $("#output").append($("<p>").append(error.responseJSON));
      }
    });
  });
  $("#insert").click(function() {
    $("#output").empty();
    const movie =
      new Movie(-1,$("#title").val(), $("#year").val(),
        $("#published").is(":checked")?"true":"false", "sepp");
    $.ajax({
      url: "http://localhost:8081/movie/",
      type: "POST",
      // Konvertierung in JSON-String
      data: JSON.stringify(movie),
      // Typ der gesendeten Daten
      contentType: "application/json; charset=utf-8",
      dataType: "json",
      success: function(data) {
        $("#output").append($("<p>").append(
          Object.assign(new Movie(), data).toString()));
      },
      error: function(error) {
        $("#output").append($("<p>").append(error.responseJSON));
      }
    });
  });
});
</script>
```

PROBLEM: Cross-Origin Resource Sharing (CORS)



Browser meldet:

```
Access to XMLHttpRequest at 'http://localhost:8081/movie?sort=as (index):1
c' from origin 'http://localhost:8080' has been blocked by CORS policy: No
'Access-Control-Allow-Origin' header is present on the requested resource.
```

CORS ist ein Mechanismus, der Web-Browsern Cross-Origin-Requests ermöglicht. Zugriffe dieser Art sind normalerweise durch die Same-Origin-Policy (SOP) untersagt und werden von Browsern unterbunden[9].

Erlauben aller Zugriffe auf Node-Server

```
app.use((request, response, next) => {
  response.header("Access-Control-Allow-Origin", "*");
  response.header("Access-Control-Allow-Methods",
    "GET, PUT, POST, DELETE");
  response.header("Access-Control-Allow-Headers",
    "Origin, X-Requested-With, Content-Type, Accept,
    Authorization");
  next();
});
```

Node-Server teilt Browser beim Senden der Antwort durch Setzen des Headers mit, dass er obige Zugriffe erlaubt

[9] Quelle: https://de.wikipedia.org/wiki/Cross-Origin_Resource_Sharing