Webentwicklung

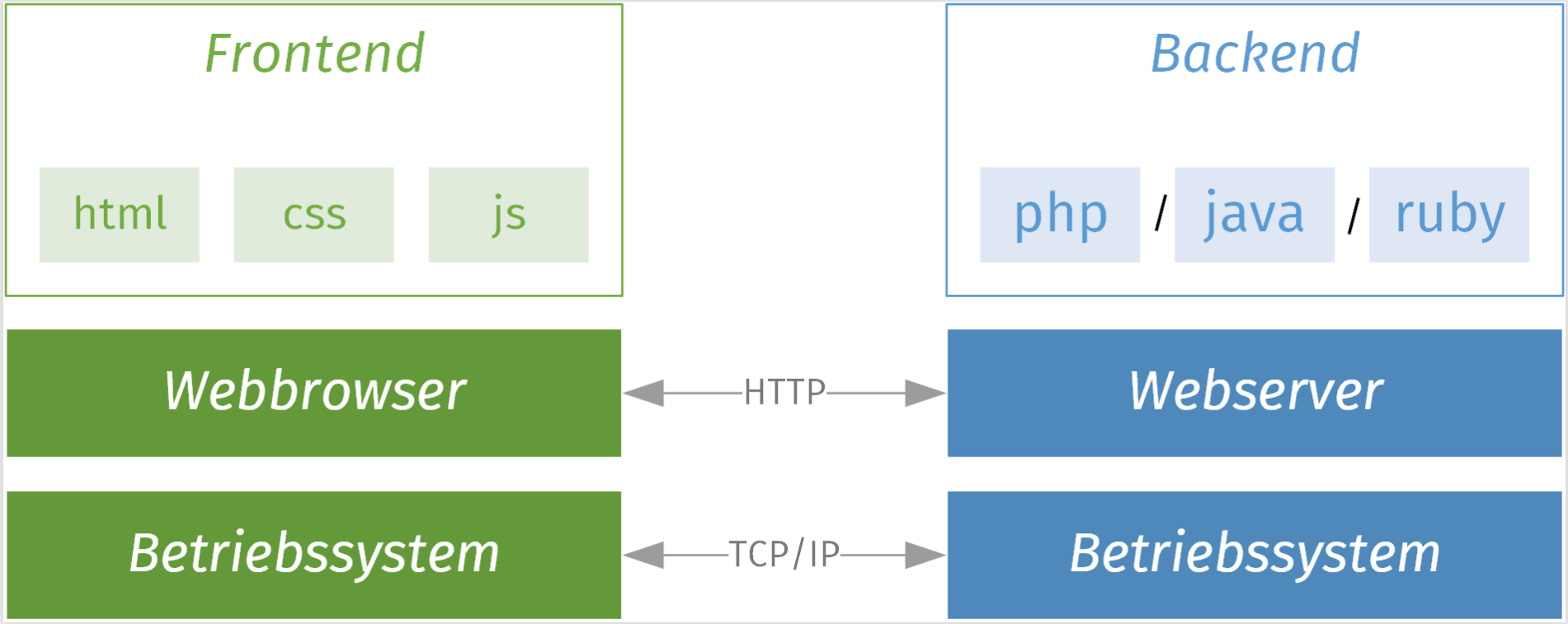
Querschnittsthema: Webservices und AJAX

Inhalt dieser Einheit

1. Service-Orientierte Architektur
2. XML-basierte Webservices
3. REST-basierte Webservices
4. Umsetzung in PHP
5. AJAX

Frontend & Backend

- Kommunikation über HTTP



- Inhalte der Nachrichten (bisher):
  - Client: Requests (`GET` und `POST`)
  - Server: Responses (HTML, JavaScript, CSS, andere Ressourcen)

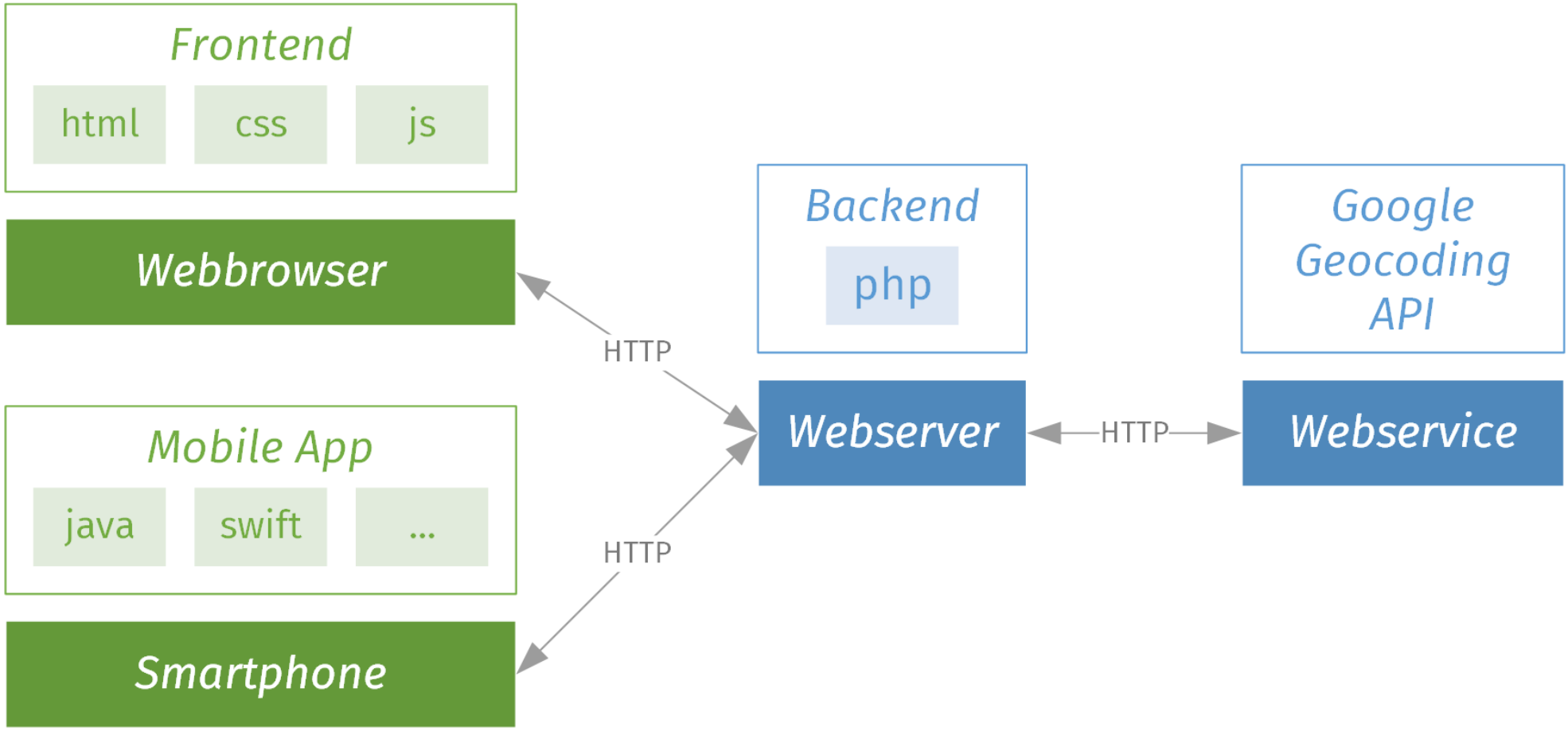
Webservices

- Heute zwei Belange:
  - Konsequentere Trennung von Inhalt und Darstellung
  - Programmatische Nutzung von Diensten über APIs
- **Anwendungsfall 1**:
  - Lauftagebuch als Webanwendung; HTML & CSS *responsive*
  - Jetzt: Smartphone-App (Schrittzähler, Zeitmessung, GPS, …), automatische Eintragung im Lauftagebuch
  - Idee: Backend wiederverwenden
- **Anwendungsfall 2**:
  - Backend: GPS-Daten → Städte/Regionen
    - (z.B. für Feature *Aktivster Läufer der Stadt*)
  - Idee: externen Dienst nutzen
    - (z.B. Google Geocoding API)

Service-Oriented Architecture

Wer macht hier eigentlich was?

Architektur



Allgemeine Idee

- Technisch in *beiden* Anwendungsfällen:
  1. Dienst über URI erreichbar
  2. Anfragen und Antworten: HTTP(S)
     - (bisher nur `GET` und `POST` für HTML-, JavaScript- und CSS-Code)
- weitere Eigenschaften von **Webservices**:
  3. Nachrichten in Datenformat (z.B. XML oder JSON)
  4. Beschreibung der Schnittstelle
- Heute: zwei Arten von Webservices
  - SOAP: XML für Nachrichten und Schnittstellenbeschreibung
  - REST: Kommunikation direkt in HTTP, verbale Beschreibung

XML-basierte Webservices

mit SOAP und WSDL

Remote Procedure Call (RPC)

- Idee: entfernte Objekte wie lokale behandeln
  - Lokal in Anwendungslogik (z.B. Banking-App):

```
float balance = account.getBalance();
if (balance > amount) {
    account.transfer(otherAccount, amount);
}
```

    - Objekt `account` ist “verdrahtet”, sodass jeder Methodenaufruf als Nachricht übers Netzwerk geht
  - Tatsächliche Implementierung serverseitig (bei der Bank):

```
class Account {
   public float getBalance() { /* ... */ }
   public void transfer(Account target, float amount) { /* ... */ }
}
```

    - Rückgabewert in Nachricht zurück an Client
- Nachrichtenkanal für Geschäftslogik transparent

XML-RPC und SOAP

- 1998: XML-RPC
  - Remote Procedure Call mit XML-Nachrichten
- 2003: Nachfolger SOAP, Version 1.2 als W3C-Standard
  - damals: *Simple Object Access Protocol*
- SOAP definiert XML-Format
  - Wurzel-Element `Envelope`, darin `Header` und `Body`
- SOAPs XML-Nachrichten sind protokollunabhängig
  - z.B: TCP, UDP oder auch SMTP, meist aber **HTTP**
- Nachrichten werden meist programmatisch erzeugt

SOAP-Nachrichten in HTTP

- technisch: XML-Dokument im HTTP-Body
  - `getBalance()`-Aufruf: HTTP-POST-Request

```
POST /banking HTTP/1.1
Host: meine-bank.de

<?xml version="1.0"?>
<soap:Envelope>
<soap:Body><b:GetBalance>...</b:GetBalance></soap:Body>
</soap:Envelope>
```

  - `getBalance()`-Rückgabewert: HTTP-Response

```
HTTP/1.1 200 OK

<?xml version="1.0"?>
<soap:Envelope>
<soap:Body><b:GetBalanceResponse>...</b:GetBalanceResponse></soap:Body>
</soap:Envelope>
```

Schnittstellen-Beschreibung

- Beschreibung der verfügbaren Operationen
  - sowie ihrer Parameter und Rückgabewerte
  - in Form von erlaubten SOAP-Nachrichten, z.B.
    - Input: Nachricht `GetBalance` mit einem `Account`-Parameter
    - Output: Nachricht `GetBalanceResponse` mit einem `float`-Wert
    - Adresse `/banking` an die der Aufruf geschickt werden soll
- Beschreibung: Maschinen-lesbar als XML-Datei
  - Format: **WSDL** (*Web Services Description Language*)
  - erlaubt komplexe Datentypen
  - erlaubt Code-Generierung für *Remote Procedure Call*
    - Eingabe: WSDL-Datei
    - Ausgabe: Klassen, die gesamte HTTP-Kommunikation kapseln

XML-basierte Webservices

- **Vorteile**
  - Ein- und Ausgabeformate exakt spezifiziert
  - Typsicherheit, unabhängig von Programmiersprachen
  - ermöglicht automatisierte Code-Generierung
- **Nachteile**
  - XML-Formate SOAP und WSDL sind schwergewichtig
- **Verbreitung**
  - Anfang der 2000er einigermaßen beliebt
  - aktuell: kaum noch öffentlich erreichbare SOAP-Dienste
    - Firmen-intern mag es noch einige geben

REST-basierte Webservices

HTTP-Methoden, JSON (und auch XML)

Erinnerung: SOAP-Nachrichten

- SOAP ist für div. Szenarien ausgelegt, daher umfangreich
  - Anfrage, gekürzt (z.B. im Body einer HTTP-`POST`-Anfrage):

```
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
  <soap:Body>
    <b:GetBalance xmlns:b="http://meine-bank.de/banking">
      <b:Account>DE5200...</b:Account>
    </b:GetBalance>
  </env:Body>
</soap:Envelope>
```

  - Antwort, gekürzt (z.B. im HTTP-Response-Body):

```
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
  <soap:Body>
    <b:GetBalanceResponse xmlns:b="http://meine-bank.de/banking">
      <b:Balance>251.08</b:Balance>
    </b:GetBalance>
  </env:Body>
</soap:Envelope>
```

    - Ausnahmen (Exceptions) würden ebenfalls im Body stehen

Alternatives Design

- Etwas schlankeres Format:
  - `GET`-Anfrage (kein Body nötig)

```
GET /accounts/DE5200.../balance HTTP/1.1
Host: meine-bank.de
```

  - HTTP-Antwort

```
HTTP/1.1 200 OK
Content-length: 6

251.08
```

- HTTP-Protokoll “richtig” nutzen:
  - HTTP-Methode `GET` für die Art der Anfrage
  - Anfrageziel ist *Ressource* (Bankkonto, bzw. Stand)
  - HTTP-Status-Code `200` für Ergebnis der Anfrage
  - HTTP-Body für eigentliche Antwort

Idee von REST

- **Kern**:
  1. URL repräsentiert Inhalt *(Ressource)*
     - Name: ***Re**presentational **S**tate **T**ransfer*
  2. Umgang mit Inhalten über HTTP-Methoden
- Eigenschaften:
  - **Zustandslosigkeit**:
    - Jede Nachricht enthält alle zum Verständnis notwendigen Informationen / ist in sich geschlossen.
  - **Operationen**:
    - Können auf Ressourcen angewendet werden
    - beispielsweise `GET`, `POST`, `PUT` und `DELETE`
  - **Unterschiedliche Repräsentationen**:
    - z.B. HTML (kennen wir schon), XML, JSON

Beispielhafte REST-API

Methode	Pfad	Effekt
`GET`	`/users`	Liste aller Nutzer
`GET`	`/users/12`	Ein bestimmter Nutzer
`POST`	`/users`	Erstellt einen neuen Nutzer
`PUT`	`/users/12`	Aktualisiert Nutzer #12
`PATCH`	`/users/12`	Aktualisiert Nutzer #12 teilweise
`DELETE`	`/users/12`	Löscht Nutzer #12

HTTP-Methoden (1/2)

- **GET**:
  - fordert die angegebene Ressource vom Server an
  - keine Nebeneffekte, Zustand auf dem Server wird nicht verändert (GET ist “sicher”)
- **POST**:
  - fügt neue Ressource unterhalb der angegebenen Ressource ein
    - (neue Ressource noch ohne URI, daher übergeordnete R. ansprechen)
  - Antwort: URI der neuen Ressource
  - nicht idempotent: Neuer Aufruf heißt neue Ressource
- **PUT**:
  - legt neue Ressource an
  - existiert Ressource bereits, wird sie aktualisiert
  - ist idempotent: zweiter Aufruf hat keinen weiteren Effekt

Quelle: https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html

HTTP-Methoden (2/2)

- **PATCH**:
  - Teil der angegeben Ressource wird geändert
    - man gibt nur die Attribute an, die man ändern möchte
    - (Methode wurde nachgerüstet in RFC 5789)
- **DELETE**:
  - löscht die angegebene Ressource
    - muss nicht sofort passieren, sondern kann vermerkt werden
  - ist idempotent
- **HEAD**:
  - fordert Metadaten zu einer Ressource vom Server an
  - keine Nebeneffekte, damit “sicher”
- **OPTIONS**:
  - welche Methoden stehen auf einer Ressource zur Verfügung?
  - keine Nebeneffekte, damit “sicher”

Quelle: https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html

Einschub: Microservices

- Wie SOAP: REST abstrahiert von Programmiersprachen
  - Konsequenz: In vielen Unternehmen werden große Softwaresysteme durch “Mikroservices” realisiert
- Jedes Team baut ein abgeschlossenes Software-System
  - Kommuniziert mit anderen Teil-Systemen über HTTP
    - oft: REST mit JSON
  - viele Möglichkeiten, z.B. pro Team:
    - eigene Versionen von Bibliotheken
    - eigene Infrastruktur (Webserver)
    - eigene Programmiersprache
    - …
- prominente Beispiele (aus Berlin):
  - Zalando SE, Soundcloud, ImmobilienScout 24

Formate

- REST ist nur ein *Paradigma*, aber kein *Protokoll*
- verschiedene Datenformate sind möglich
  - HTML-Darstellung einer Ressource (“altes Web”)
  - **XML**
    - eigenes, handgeschmiedetes Format
    - existierendes Format, wie etwa RSS (für Artikel und Podcasts)
  - **JSON**: *JavaScript Object Notation*
  - Plain Text
  - CSV (Comma-separated Values)
  - …

XML vs. JSON

```xml
<Kreditkarte
   Nummer="1234-5678-9012-3456">
   <Inhaber
      Name="Mustermann"
      Alter="42" Partner="null">
      <Hobbys>
         <Hobby>Golfen</Hobby>
         <Hobby>Lesen</Hobby>
      </Hobbys>
      <Kinder />
   </Inhaber>
</Kreditkarte>
```

```json
{
  "Nummer": "1234-5678-9012-3456",
  "Inhaber": {
    "Name": "Mustermann",
    "Hobbys": [ "Golfen", "Lesen" ],
    "Alter": 42,
    "Kinder": [],
    "Partner": null
  }
}
```

Quelle: wikipedia.org

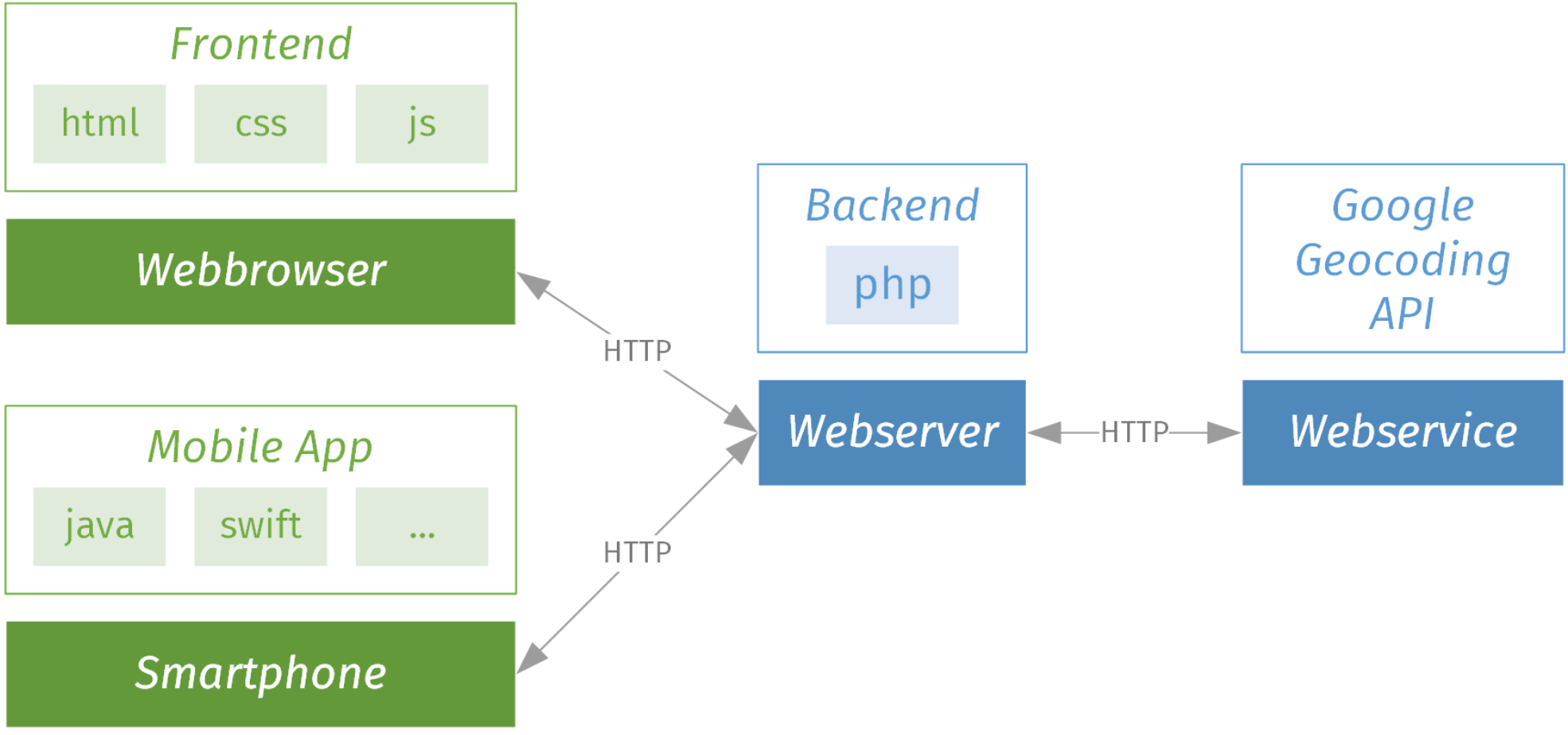
Bewertung REST-APIs

- **Vorteile:**
  - tendenziell schlankes Format
  - einfach zu erzeugende Ausgabe
  - oft einfach zu lesen
- **Nachteile:**
  - Ausgabe nicht streng typisiert, besonders bei JSON-Ausgabe
    - kennt nur primitive JavaScript-Typen, Listen und Arrays
  - Schnittstellenbeschreibung muss händisch geschrieben werden
  - … und manuell verstanden, da i.d.R. nicht maschinenlesbar
- **Verbreitung:**
  - Wenn API im Web: REST (Google APIs, GitHub, …)
    - manchmal mit XML, meistens mit JSON

Umsetzung

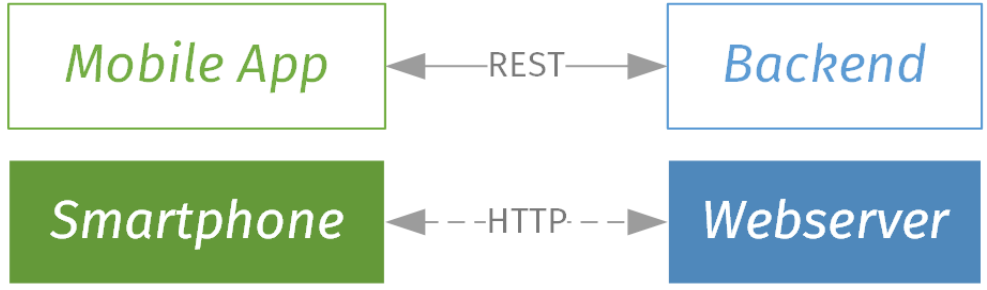
Umgang mit REST-APIs in PHP

Erinnerung: Anwendungsfälle



REST-API für Backend

- Zwei Aspekte
  1. Implementieren der Endpunkte/Routen
     - (bisher: `GET` und `POST`)
  2. Auswahl eines Datenformats
     - (bisher: HTML-Code)
- Beides geht in Symfony sehr leicht



REST-API mit Symfony

- HTTP-Methode mit `@Route` am Controller prüfen
- Gleiches URL-Muster, aber verschiedene Methoden:

```
/**
 * @Route("/users/{id}", methods={"GET"})
 */
public function getUser(User $user) { /* ... */ }

/**
 * @Route("/users/{id}", methods={"DELETE"})
 */
public function deleteUser(User $user) { /* ... */ }
```

  - 1. Anfrage an `getUser()`, 2. Anfrage an `deleteUser()`:

```
GET /users/42 HTTP/1.1
DELETE /users/42 HTTP/1.1
```

Quelle: https://symfony.com/doc/current/routing/requirements.html

Verschiedene Ausgabe-Formate

- Festlegung im Controller:
  1. Template-Engine Twig ist textbasiert
     - bisher: Templates, die HTML erzeugen

```
return $this->render('user-list.html.twig', ['users' => $users]);
```

     - analog: Templates, die z.B. XML erzeugen (oder theoretisch auch JSON)

```
return $this->render('user-list.xml.twig', ['users' => $users]);
```

  2. Ohne Template-Engine (für JSON):
     - Mit Symfony-Controller-Hilfsfunktion:

```
return $this->json($users);
```

Ausgabe-Format festlegen

- Symfony kennt verschiedene Formate
  - Setzen über Platzhalter `_format` in Route:

```
/**
 * @Route("/users/{id}.{_format}", defaults={"_format"="html"})
 */
public function showUser(User $user) { /* ... */ }
```

  - 1. Anfrage: `_format` ist `html`, 2. Anfrage: `rss`

```
GET /users/42.html HTTP/1.1
GET /users/42.rss HTTP/1.1
```

  - Abfrage über Request-Objekt; Nutzen für Template-Wahl:

```
$format = $request->getRequestFormat();
// Format nutzen, um passendes Template zu laden
$this->render('user-profile.'.$format.'.twig', ['user' => $user]);
```

  - (bekannte Formate: automatisch korrekter `Content-Type`)

Quelle: https://symfony.com/doc/current/templating/formats.html

Passendes Template laden

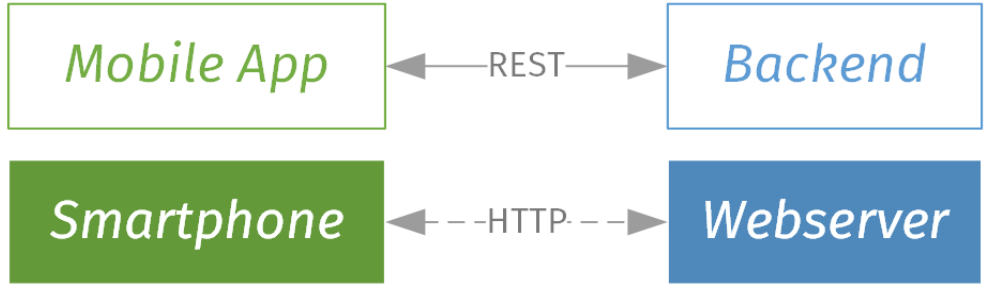
- Automatisch per `@Template`-Annotation
  - Annotationen am Controller

```
/**
 * @Route("/users/{id}.{_format}", defaults={"_format"="html"})
 * @Template
 */
public function showUser(User $user) { }
```

  - `_format` wird zur Namensbestimmung verwendet
    - `GET /users/12` nutzt `show_user.html.twig` (wg. `defaults` in `@Route`)
    - `GET /users/12.html` nutzt `show_user.html.twig`
    - `GET /users/12.json` nutzt `show_user.json.twig`
    - `GET /users/12.myfile` nutzt `show_user.myfile.twig`

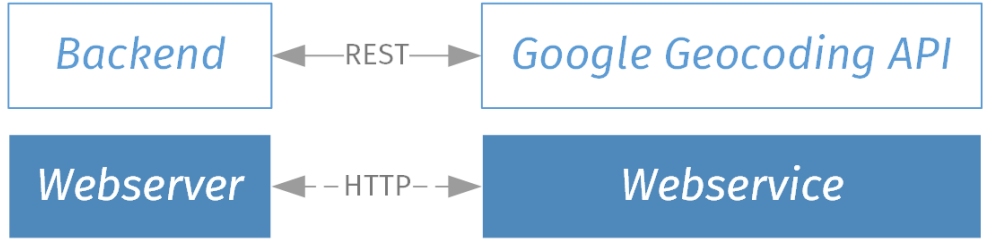
Fertig: REST-API für Backend

1. Implementieren der Endpunkte/Routen ✓
   - Durch `@Route` mit `methods`
2. Auswahl eines Datenformats ✓
   - Format in URL setzen (`_format`)
   - Auslesen und für Template-Auswahl benutzen



Jetzt: REST-API benutzen

- API-Doku lesen
  - URL- und Anfrageformat
  - Ausgabeformat
    - (beides war mit SOAP/WSDL nicht nötig, weil maschinenlesbar)
  - Beispiele:
    - Google Geocoding API
    - GitHub API
- Bibliotheken, um HTTP-Anfragen zusammenzubauen
  - für PHP z.B. Guzzle



Bsp: REST-API benutzen

- Nutzung der Google Geocoding API aus PHP heraus:

```php
$client = new GuzzleHttp\Client();
$res = $client->request(
    'GET',
    'https://maps.googleapis.com/maps/api/geocode/json',
    ['query' => [
      'latlng' => '40.714224,-73.961452',
      'key' => $api_key
    ]]
);
$result = json_decode($res->getBody());
```

  - (Alternativ: Query-Parameter in URL angeben)
- Was tun wenn es keine REST-API gibt?
  - Web-Scraping: HTML-Ausgabe laden und zerlegen
    - z.B. in PHP mit Goutte

REST-API aus Frontend benutzen?

- Wenn wir schon eine REST-API anbieten: Warum nutzen wir die nicht auch im Frontend?
- Bisher: teilweise auch schon “REST-APIs”:
  - Ansprechbar direkt aus dem Frontend:
    - GET-Adressen: Erreichbar über Links
    - POST-Adressen: Erreichbar über HTML-Formulare
- Gibt es noch weitere Möglichkeiten?

Symfony: Formular-Methode

- für eigene Formulare (Doku):
  - bei der Erzeugung im Controller:

```
$formBuilder->setMethod('DELETE')
```

  - oder beim Rendern im Template:

```
{{ form_start(form, {'method': 'DELETE'}) }}
```

- (Technisch: dennoch HTTP-POST)
  - intern: `<input name='_method' type='hidden' value='DELETE'>`
  - o.g. Annotationen (`@Route{"...", methods=...}`) funktionieren

Offenes Problem

- Requests dieser Art ändern den Inhalt im Browser
  - Klick auf Link: Neue Seite wird geladen
  - Formular-Submit: Neue Seite wird geladen
- Hintergrund:
  - Historisches “Request/Response”-Paradigma
- Alternative: Requests im Hintergrund ausführen

Asynchronous JavaScript and XML

Nicht immer asynchron, nicht immer XML

AJAX, eine JavaScript-Technik

- nutzt `XMLHttpRequest`-Objekte für Anfragen
  - (analog zu Guzzles Request-Objekt, nur im Frontend)
- Google-API-Beispiel von oben:

```
var url = 'https://maps.googleapis.com/maps/api/geocode/json';
var query = '?latlng=40.714224,-73.961452+key=' + api_key;
var request = new XMLHttpRequest();
request.open('GET', url + query);
request.onreadystatechange = function() {
   if (request.readyState === XMLHttpRequest.DONE &&
       request.status === 200) {
      // mach etwas mit request.response
   }
};
request.send();
```

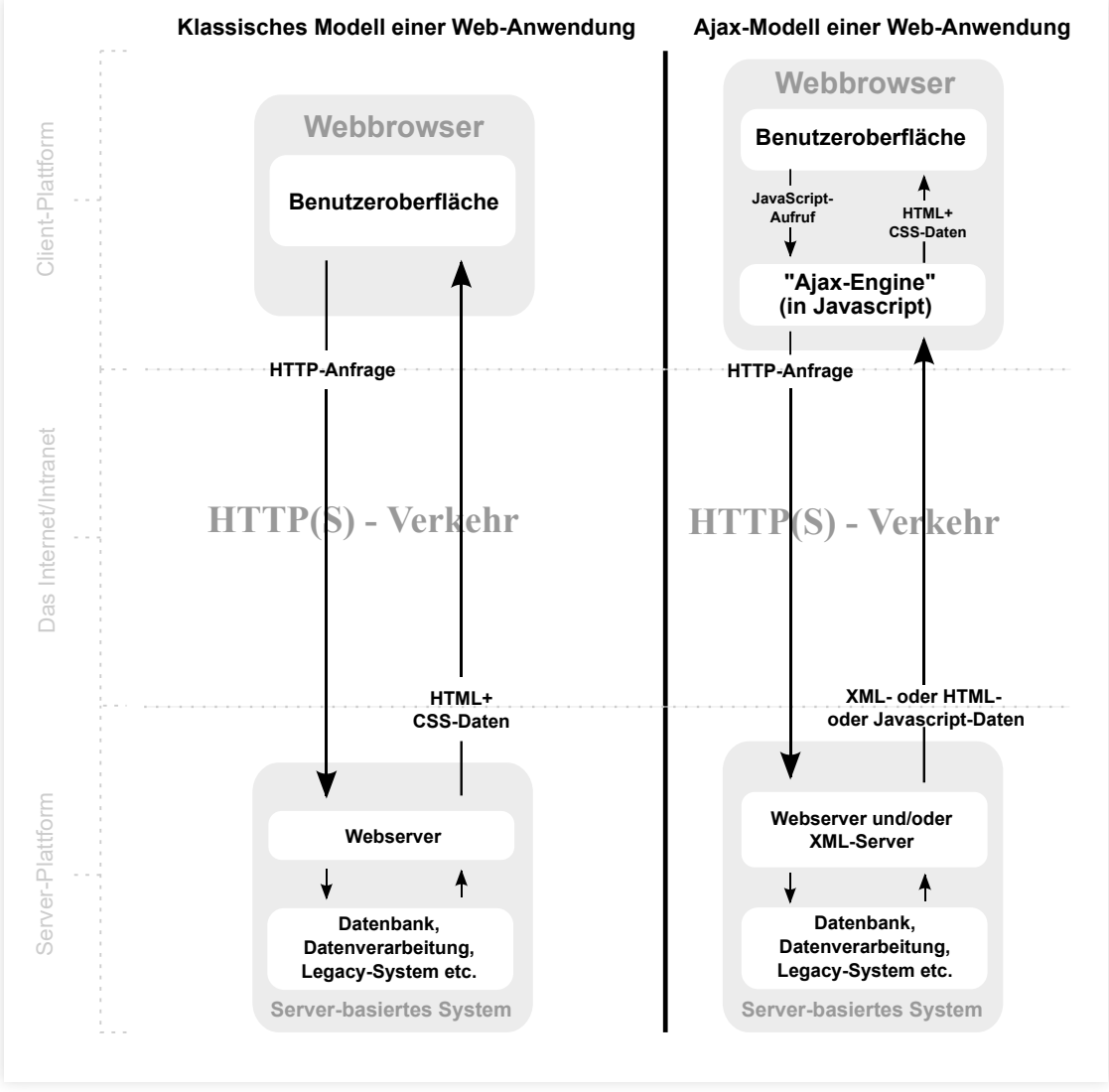
Quelle: https://developer.mozilla.org/en-US/docs/Web/Guide/AJAX/Getting_Started

`XMLHttpRequest`-Eigenschaften

- `open()`:
  - Bereit Anfrage vor, gibt HTTP-Methode und URL an
- `onreadystatechange`
  - Callback, wird aufgerufen, wenn sich der `readyState` ändert
    - oft nur Status `4`, bzw. `DONE` interessant
- `status`
  - HTTP-Status-Code
- `response`
  - HTTP-Response, z.B. ein String oder auch ein JavaScript-Objekt
- `send()`:
  - Anfrage absenden

Quelle: https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest

AJAX konzeptionell



Quelle: https://commons.wikimedia.org/wiki/File:Ajax-vergleich.svg

Toll, aber

Was **macht** man jetzt damit?

AJAX-Einsatzmöglichkeiten

- Einige Szenarien:
  - Nicht komplette Seite neu laden, sondern nur Teile ersetzen
    - z.B. beim Hinzufügen/Bearbeiten eines Eintrags einer langen Liste
    - z.B. nach erfolgreichem Login: Nur Login-Formular austauschen
  - Regelmäßig Zwischenstände ans Backend “flüstern”
    - z.B. für “Entwürfe”-Feature eines Webmailers
  - Regelmäßig beim Server nach Updates fragen
    - z.B. für Posteingang bei einem Webmailer
  - mehr Interaktion im Browser
    - z.B. für Autovervollständigung bei Eingabefeldern
    - z.B. für serverseitige Validierung noch während der Eingabe

Szenarien konzeptionell

- Für Umsetzung entscheidend:
  - Trennung Frontend/Backend: Was wird wo “berechnet”?
- Oder etwas konkreter:
  - Was steckt in der Server-Antwort:
    1. Fertig “gerenderte” HTML-Schnipsel? oder
    2. JSON-String mit Rohdaten?
- Beispiel *Neuer Listen-Eintrag*: HTTP-Antworten des Servers in HTML vs. JSON

```
<tr><td>2017-12-29</td><td>12 km</td><td>01:02:21</td></tr>
```

```
{"date":"2017-12-29", "distance":12, "time":"01:02:21"}
```

Abwägung der Umsetzungsideen

- **Vor- und Nachteile:**
  - Template serverseitig oft schon vorhanden, rendern ist dann dort leichter als im Frontend
    - rendert man an zwei Stellen, drohen Inkonsistenzen bei Template-Änderungen (z.B. neue Spalte)
    - (wenn Markup trivial ist, ist das kein gewichtiger Punkt)
  - HTML-Schnipsel in DOM einfügen ist leicht implementiert
  - JSON-Format erleichtert die Entwicklung weiterer Clients
  - JSON-Format ist kompakter als gerendertes HTML (Datenvolumen)
  - DOM client-seitig aus Rohdaten erzeugen ist evtl. rechenintensiv (Performance, Akkuleistung)

Ajax mit jQuery

```
jQuery.ajax( url, settings );

jQuery.post( url, data, success(data, textStatus, jqXHR), dataType );
jQuery.get( url, data, success(data, textStatus, jqXHR), dataType );

jQuery.getJSON( url, data, success( data, textStatus, jqXHR ) );
jQuery.getScript( url, success(script, textStatus, jqXHR) );

$('.element').load(
   url, data, complete(responseText, textStatus, XMLHttpRequest)
);
```

Quelle: http://api.jquery.com/category/ajax/

jQuery: `ajax`

- Low-Level-Zugriff auf alle AJAX-Funktionen:

```
$.ajax({
    url: 'api/createUser.php',
    data: {
        name: user.name,
        age: 42
    },
    datatype: 'json',
    type: 'POST',
    success: function(data) {
        console.log('Response', data);
    }
});
```

  - statt `POST`: *alle* HTTP-Methoden sind möglich
    - (Erinnerung: Links und Formulare resultieren immer in `GET` oder `POST`)

jQuery: `get` und `post`

- Kurzformen von `$.ajax()` für HTTP GET und POST.

```
$.get('test.php',
  { name: 'John', time: '2pm' },
  function(data) {
    $('.result').html(data);
  }
);
```

jQuery: `getJSON`

- Wenn man eine JSON-Antwort erwartet:

```
// options.json: { 'one': 'Einer', 'two': 'Zwei', 'three': 'Drei' }
$.getJSON('ajax/options.json', function(data) {
    var options = '';
    $.each(data, function(key, val) {
        options += '<option value="' + key + '">' + val + '</option>';
    });
    $('.dropdown').html(options);
});
```

  - Serverseitig: i.d.R. *keine* statische Ressource
    - sondern Programm, z.B. Symfony-Controller mit JSON-Output

jQuery: Ajax Promises

- Alternative API neben `onreadystatechanged`:

```
var jqxhr = $.get('example.php')
  .done(function() { alert('success'); })
  .fail(function() { alert('error'); })
  .always(function() { alert('finished'); });
```

  - (*Promises*: Teil von ES6, für uns im Detail nicht wichtig)
    - erleichtern asynchrone Programmierung
    - z.B. durch Verkettung von Callbacks

Quelle: http://api.jquery.com/jQuery.ajax/#jqXHR

Zusammenfassung

- Idee von Service-Oriented Architecture
  - und Micro-Services
- WebService-Arten: XML- vs. REST-basiert
  - Vor- und Nachteile
- REST-APIs
  - Semantik verschiedener HTTP-Methoden
  - Begriffe/Bedeutung: *Sichere* und *Idempotente* Methode
- AJAX
  - `XmlHttpRequest`-Objekt in JavaScript
  - Einsatzmöglichkeiten
  - Abwägung: HTML-Antwort vs. JSON-Antwort

Literatur

- AJAX beim Mozilla Developer Network
- REST-APIs ausführlich
- Eigene REST-API entwickeln:
  - API-Keys mit Symfony
  - FOSRestBundle: Leichter REST-APIs für Symfony erstellen
    - (wirkt noch nicht 100%ig kompatibel mit Symfony 4)

Danke!