Fortgeschrittene Programmierung

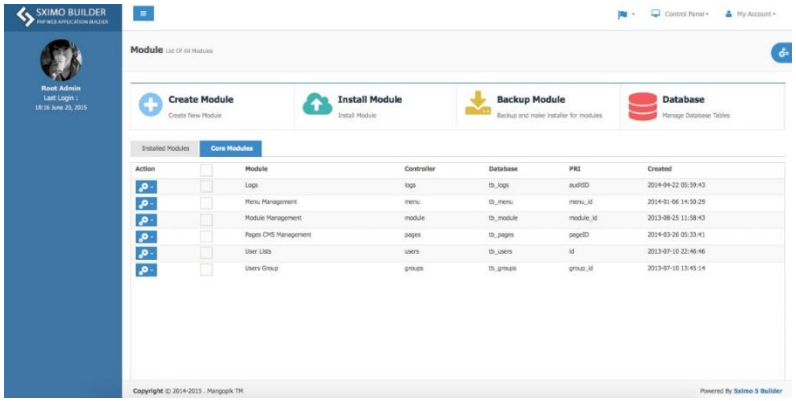
Vorlesungsfolien
Thomas Richter

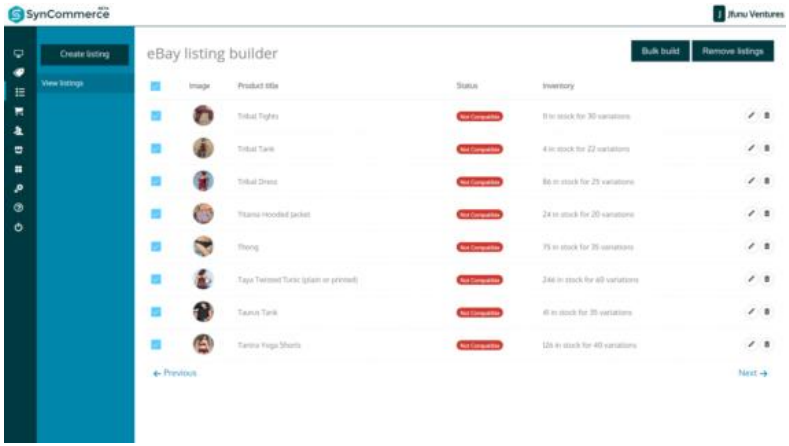
REST-Interfaces

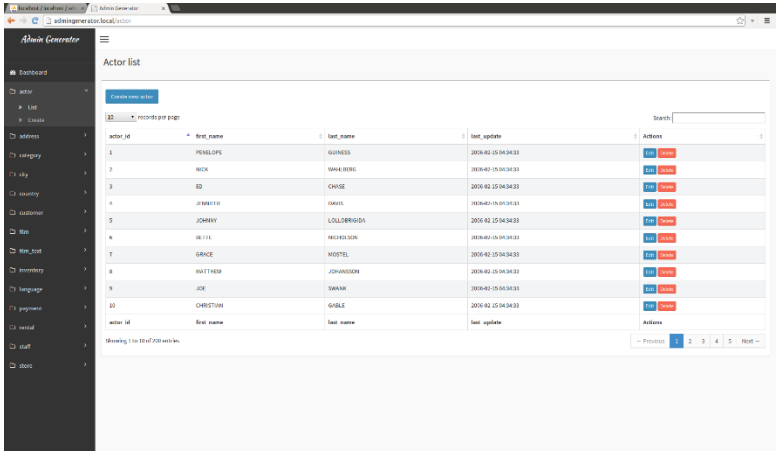
Einleitung: CRUD-Paradigma

- Informationssysteme bauen bzgl. Datenhaltung sehr oft auf wenigen Basisoperationen auf:
  - **C**reate: Erzeugen eines Datensatzes
  - **R**ead: Lesen / Abrufen eines oder mehrerer Datensätze
  - **U**pdate: Ändern eines Datensatzes
  - **D**elete: Löschen eines oder mehrerer Datensätze

- Diesem Paradigma folgende Webanwendungen ähneln sich oft in Look-and-Feel:







Quellen:

- https://s3.envato.com/files/157950544/screens/2-edit-countries.png
- http://www.webdesigndev.com/wp-content/uploads/2017/03/004019-Laravel-CRUD-CMS-Sximo-5-LTS-by-mangopik-_-CodeCanyon.jpg
- https://img0.etsystatic.com/121/0/11462924041/iag_570xN.8927016_5j1g.png
- http://crud-admin-generator.com/images/list.png

Einleitung: CRUD-Frameworks

- CRUD-Frameworks
  - kapseln die Persistenzschicht und bieten der Applikationslogik Zugriff auf Objektebene
  - liefern häufig Tools zur Datenmodellierung sowie Templates für (Web)Oberflächen
- Beispiele:
  - Java: Hibernate
  - PHP: CakePHP
  - Oracle Application Express (APEX)

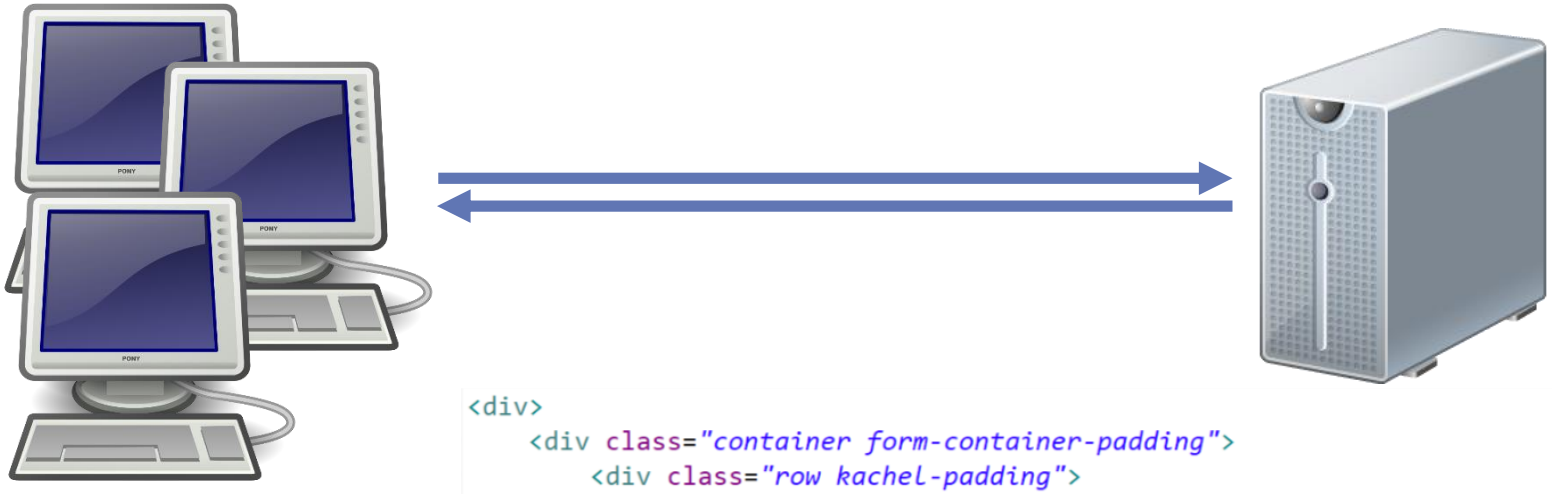
Verteilte Informationssysteme: Server

- Webbasierte Informationssysteme werden erst durch zentrale, von vielen Clients geteilte Datenbasis sinnvoll.

  → es kommen Server ins Spiel, auf den die Daten gespeichert und ggf. verarbeitet werden

- Technische Implementierung der Datenbasis (SQL-DB, No-SQL DB, Filesystem, …) für die Fachlichkeit häufig unerheblich

Webserver - Historie



Je Request wird eine HTML-Seite geladen, die zuvor vom Server dynamisch erzeugt wurde.

→ Latenz, Netzwerk-last, Serverlast

```
<div>
    <div class="container form-container-padding">
        <div class="row kachel-padding">
            <div class="kachel-space">
                <div class="hidden-xs kachel-label kachel-info">
                    <p><?php echo $welcomeMessage;?></p>
                </div>
            </div>
<?php
// Kategorien anzeigen
for ($i = 0; $i < count($categories); $i++) {
?>
<div class="col-xs-6 col-md-4 kachel-space">
    <a href="reporterData.php" class="btn btn-default kachel-auswahl" onclick='
        <img class="btnicon" src="<?php echo $projectFolder;?>/images/categoryI
        <div class="kachel-label"><?php echo $categories[$i];?></div>
    </a>
</div>
<?php
}
```

Wie spricht man den Server an, wenn man bestimmte Daten abrufen / einliefern / … will?

- Web Services
  - RPC-ähnliche APIs in diversen Geschmacksrichtungen:
    - SOAP mit singulärem Endpoint, Operationen im Request
      - POST /soapapi.php
    - Custom-made APIs mit Wildwuchs an Endpoints
      - GET /myapi/getProduct.php?productID=42
      - POST /myapi/searchProducts.php
      - POST /myapi/createNewProduct.php
      - POST /myapi/updateProduct.php
      - GET /myapi/findProduct.php?category=517
      - …
  - Sehr häufig POST bei Datenabruf (Suchformular, etc.)
  - POST ist unsichere Operation und kann daher nicht gecached werden (später mehr)

Representational State Transfer (REST)

- Roy Fielding: **Architectural styles and the design of network based software architectures.**
- REST: Wie schreibt man Software, die in einem Netzwerk funktioniert?
- Definiert Anforderungen an REST
  - Client-Server
  - Zustandslos (Server und Client kennen gegenseitig Status nicht)
  - Cacheable
  - Schichtenmodell (Proxies, Caches, Gateways, etc.)
  - Code on Demand (optional)
  - Einheitliche Schnittstelle (Uniform Interface)

Stateless Server / Caching

- Jeder Request enthält alle Informationen, die zur Beantwortung erforderlich sind
- Der Server selbst ist nicht zustandslos, er kennt nur den Zustand des (der) Clients nicht.
  - Keine Session-IDs etc.
- Verbesserte Skalierbarkeit
- Mehr Netzwerk-Traffic!
- Caching
  - Clientseitig: weniger Requests
  - Serverseitig: weniger Ressourcen, kürzere Antwortzeiten

Code on Demand

- Server liefert ggf. auch die Logik aus
- Client stellt nur Ausführungsumgebung bereit (JavaScript-Runtime, Rendering-Engine)
- Einfaches Deployment

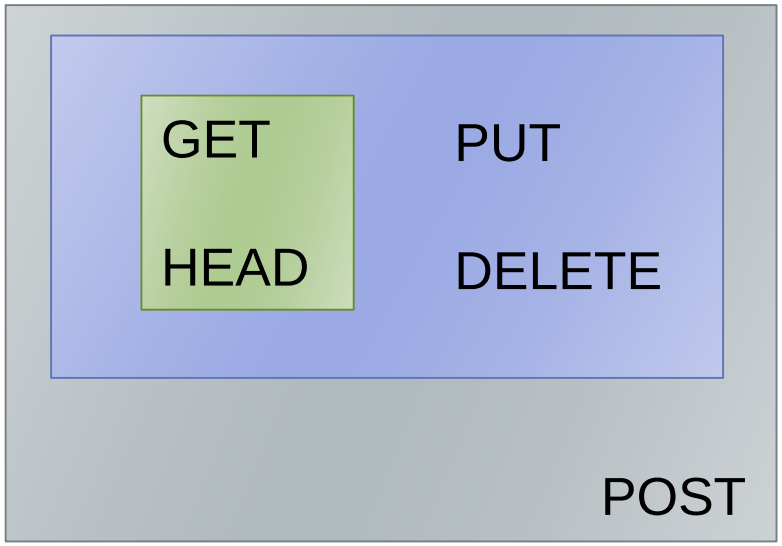
Uniform Interface

- Objekte heißen Ressourcen und werden durch URLs eineindeutig identifiziert
- Operationen auf Ressourcen werden durch HTTP-Methoden festgelegt
- Menge der HTTP-Methoden stellt alle auf einer Ressource ausführbaren Operationen dar
- Operationen sind implizit und nicht Teil des URL

- Daten werden mit Hypermedia repräsentiert: Webseiten sind keine Ressourcen sondern repräsentieren Ressourcen → eine Ressource kann verschiedene Repräsentationen haben

- (Im Service wird mit Links navigiert)

HTTP-Verben / Methoden



HTTP-Verben / Methoden

GET

HEAD

- GET und HEAD sind sicher
  - Verändern den Zustand einer Ressource nicht
  - Cacheable
- GET liefert eine Repräsentation einer Ressource samt ihrer Metadaten zurück
- Head liefert nur die Metadaten einer Repräsentation einer Ressource zurück

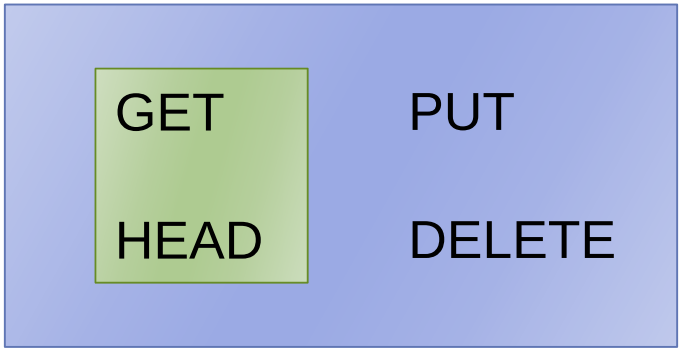
Repräsentationen

```
GET /articles/42
Host: myblog.local
Accept application/json

HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Allow: GET, POST, PUT, DELETE
{
  "id": "42",
  "titel": "mein erster Blogbeitrag",
  "datum": "14.03.2018 13:24:06",
  "text": "..."
}
```

Repräsentationen

```
GET /articles/42
Host: myblog.local
Accept application/xml

HTTP/1.1 200 OK
Content-Type: application/xml; charset=utf-8
Allow: GET, POST, PUT, DELETE
<?xml version="1.0" encoding="utf-8"?>
<article id="42" xl:type="simple"
xl.href="https://myblog.local/articles/42>
  <titel>mein erster Blogbeitrag</titel>
  <datum>14.03.2018 13:24:06</datum>
  <text>...</text>
</article>
```

HTTP-Verben / Methoden



- GET, HEAD, PUT und DELETE sind idempotent.
  - f(x) = f(f(x))
- Idempotente Aktionen können ohne Nebeneffekte wiederholt ausgeführt werden
- PUT (erstellt) oder ändert eine Ressource
- DELETE löscht eine Ressource

POST

- POST ist weder sicher noch idempotent
  - Vorsicht!
- POST fügt eine Unter-Ressource an eine Ressource an
- POST wird oft falsch verwendet, z. B. zum Übermitteln von Suchformularen
  - Problem: Ergebnis ist nicht cacheable und nicht adressierbar, da die Suchanfrage im Requestbody liegt

HTTP-Methoden: Regeln

- GET und HEAD sind verpflichtend
- Alle anderen Methoden sind optional, müssen jedoch der beschriebenen Semantik entsprechen, wenn sie implementiert werden
- Mit OPTIONS kann ermittelt werden, welche Methoden auf eine Ressource anwendbar sind

Antworten des Servers

- Sind selbsterklärend
  - Standard HTTP-Header
    - Content-Type: application/json; charset=utf-8
    - Allow: GET,HEAD,POST
  - Statuscodes:
    - 200 OK
    - 201 Created
    - 404 Not Found
    - 405 Method Not Allowed
    - 507 Insufficient Storage

CRUD, HTTP und REST

- Offenbar eignet sich HTTP, um REST-Interfaces zu implementieren, die eine CRUD-Funktionalität bereitstellen:
  - **C**reate: POST
  - **R**ead: GET
  - **U**pdate: PUT
  - **D**elete: DELETE

HTTP-basiertes REST Interface entwerfen

- Mit welchen Objekten haben wir zu tun?
- Wie sind diese organisiert?
- Bsp.: Webshop mit Produkten und Produktfotos
- https://myshop.local/products
- https://myshop.local/products/23
- https://myshop.local/products/23/photos
- https://myshop.local/products/23/photos/7
- https://myshop.local/products?brand=ibm&sort=asc

GET – Ressource(n) abrufen

```
GET /products HTTP/1.1
Host: myshop.local
Accept: application/json
```

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Content-Length: 87
[{"id":"23", "name":"Laptop", "price":459.99},
 {"id":"42", "name":"TV", "price":319.00}]
```

- Typische Statuscodes:
  - 200 OK
  - 404 Not Found: Das Produkt mit ID … wurde nicht gefunden

POST – Ressourcen erstellen

```
POST /products HTTP/1.1
Host: myshop.local
Content-Type: application/json
{"name":"Smartphone", "price":229.00}
```

```
HTTP/1.1 201 Created
Content-Type: application/json; charset=utf-8
Content-Length: 11
{"id":"57"}
```

- Typische Statuscodes:
  - 201 Created

PUT – Ressourcen ändern

```
PUT /products/23 HTTP/1.1
Host: myshop.local
Content-Type: application/json
{"id":"23", "name":"Laptop", "price":399.00}
```

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Content-Length: 44
{"id":"23", "name":"Laptop", "price":399.00}
```

- Typische Statuscodes:
  - 200 OK
  - 404 Not Found

DELETE – Ressourcen löschen

```
DELETE /products/23 HTTP/1.1
Host: myshop.local
Content-Type: application/json
```

```
HTTP/1.1 200 OK
Content-Type: text/plain; charset=utf-8
Content-Length: 18
Product 23 deleted
```

- Typische Statuscodes:
  - 200 OK
  - 404 Not Found

Filtern von Daten mittels GET

- https://myshop.local/products/brand/ibm

- Warum ist das eine schlechte Idee?

Filtern von Daten mittels GET

- https://myshop.local/products/brand/ibm

- Warum ist das eine schlechte Idee?

- brand ist keine Unterressource von product
- Neue Filter erfordern Anpassung des Clients
- Weitere Filter lassen sich nicht kombinieren, z. B. Kategorie:
  - https://myshop.local/products/brand/ibm/category/pc
  - Oder
  - https://myshop.local/products/category/pc/brand/ibm
  - Explizite Pfade müssten implementiert werden

Filtern von Daten mittels GET

- QueryStrings verwenden:
- https://myshop.local/products?brand=ibm
- https://myshop.local/products?brand=ibm&category=pc

- Filter lassen sich beliebig als Schlüssel/Wert-Paare im QueryString kombinieren
- API bleibt aus Sicht des Clients stabil, auch wenn neue Filter hinzukommen

Serverseitiges JavaScript 2

Express

Express

- HTTP-Klasse ist noch relativ umständlich. Z. B. müssen die verschiedenen HTTP-Methoden in der Callback-Funktion unterschieden werden (switch/case).
- Das Modul express vereinfacht das, indem für jede HTTP-Methode eigene Eventhandler registriert werden können.
- Zusätzlich kann jeder Eventhandler an einen bestimmten URI (einen Pfad) gebunden werden.
- Abhängig von HTTP-Methode und URI wird dann der richtige Eventhandler aufgerufen.

Express App erzeugen

```
var http = require("http");

// Express einbinden
var express = require("express");
// Express-App erzeugen
var app = express();

// HTTP Server erzeugen
var server = http.createServer(app);

// Server an Port binden
server.listen(3000);
```

Express Server Hello World

```
var http = require("http");

var express = require("express");
var app = express();
var server = http.createServer(app);
server.listen(3000);

app.get("/hallo", function(req, res) {
    res.contentType("text/html");
    res.status(200).send("Das ist die Route /hallo");
});

app.get("*", function(req, res) {
    res.contentType("text/html");
    res.status(200).send("Das ist die Wildcard-Route: *");
});
```

- Die Reihenfolge der Eventhandler ist von Bedeutung, wenn mehrere Handler dieselbe Route abdecken: der erste passende Eventhandler wird aufgerufen.

Webapp ausliefern

- Mittels Express kann sehr einfach die Webapp an den Browser ausgeliefert werden:

```
var app = express();

// statischen Fileserver für das Verzeichnis /app einrichten
// __dirname ist das Verzeichnis des aktuell laufenden Skripts
app.use(express.static(__dirname + '/app'));
```

- Das angegebene Verzeichnis (hier: app) wird damit zum DocumentRoot des Servers. Dateien, die dort liegen sind dann mit https://myserver/datei abrufbar (ohne app im URL). Standardmäßig wird die index.html ausgeliefert.

Pfade und Parameter

```
// GET-Request: Alle Produkte ausliefern
app.get("/products", function(req, res) {
    logUrl(req);

    res.contentType("application/json");
    res.status(200).send(JSON.stringify(articles));
});

// GET-Request: Ein Produkt mit ID ausliefern
// id wird zum Attribut des Objektes req.params
app.get("/products/:id", function(req, res) {

    // Produkt-Objekt anhand ID in Datenbank finden
    var product = getProductById(req.params.id);

    res.contentType('application/json');
    res.status(20).send(JSON.stringify(product));
});
```

Inhaltstypen / Modul body-parser

- Wenn im Request-Body application/json gesendet wird, ist das gesendete Objekt bei Verwendung des Moduls body-parser in `req.body` enthalten:

```
// Body-Parser für Requests einbinden
var bodyParser = require("body-parser");
// Parsen von JSON (application/json) aktivieren
app.use(bodyParser.json());
```

```
// POST-Request: Neues Produkt anlegen
app.post("/products", function(req, res){
    // req.body ist das vom Client gesendete Objekt
    console.log(req.body.price);
    // ...
});
```

Inhaltstypen Antwort

- Der Inhaltstyp der Antwort wird explizit gesetzt und ggf. das Antwortobjekt in JSON umgewandelt:

```
res.contentType("application/json");
res.status(200).send(JSON.stringify(product));
```

REST-Interfaces mit Express

- Es lassen sich für die HTTP-Methoden GET, POST, PUT, DELETE Eventhandler bei der Express-App registrieren:

```
// GET-Request: Alle Produkte ausliefern
app.get("/products", function(req, res) { ... });

// POST-Request: Neues Produkt anlegen
app.post("/products", function(req, res){ ... });

// PUT-Request: Produkt editieren
app.put("/products/:id", function(req, res){ ... });

// DELETE-Request: Produkt löschen
app.delete("/products/:id", function(req, res){ ... });
```

HTTP Status

- Der HTTP-Status wird mit der Funktion status gesetzt:

```
res.contentType("application/json");
res.status(200).send(JSON.stringify(product));
```

Exkurs: Persistierung

- Üblicherweise wird im MEAN-Stack eine NoSQL-Datenbank (z. B. MongoDB) eingesetzt. Wir verwenden aus Zeitgründen eine einfache Persitierung im Filesystem:

- Das Array mit allen Blogartikeln wird einfach als JSON in eine Datei geschrieben und beim Start des Servers von dort geladen.

- Alternativ kann auch eine Map statt eines Arrays verwendet werden. Die Persistierung erfolgt auf dieselbe Weise.

Exkurs: Persistierung im Filesystem

```
var articles = [];
//FileSystem Modul einbinden
var fs = require("fs");

// Einlesen aus Datei (falls vorhanden)
// __dirname ist das Verzeichnis des aktuell laufenden Skripts
var filename = __dirname + '/articles.json';
try {
    var filedata = fs.readFileSync(filename);
    articles = JSON.parse(filedata);
} catch (err) {
    console.log('Keine Datensätze gelesen');
}

// Schreibt das Array in die Datei (z. B. nach POST oder DELETE)
function updateFile() {
    fs.writeFileSync(filename, JSON.stringify(articles));
}
```

Interface testen

- Demo mit Postman