Web-Entwicklung

Serverseitige Anwendungen II: RESTful HTTP

Inhalte der Vorlesung

- Serverseitige Anwendungen: RESTful HTTP
  - Einführung
    - Representional State Transfer
    - Grundprinzipien
  - Entwurf
  - Realisierung mit Express
  - Ausblick

Einführung

Representional State Transfer (REST)

- Nachträgliche Abstraktion eines konzeptionellen Modells aus der HTTP-Spezifikation (Roy Fielding, 2000)
- Architekturstil zur Modellierung von Schnittstellen (APIs) für verteilte Client-/Server-Systeme
  - Vgl. CORBA, RPC, SOAP, …
- Grundprinzipien sind prinzipiell unabhängig von eingesetzten Technologien, Protokollen und Datenformaten
- RESTful HTTP ist aber die praktisch einzig relevante konkrete Ausprägung
  - Nicht jede HTTP-API ist automatisch RESTful
  - *Accidentally RESTful:* Trotz Verstöße gegen einzelne Grundprinzipien kann eine HTTP-API im wesentlichen RESTful sein.

REST-Grundprinzipien

- Verwaltung von Ressourcen mit eindeutiger Kennung
- Verwendung von einheitlichen Standardmethoden für alle Ressourcen
- Unterstützung unterschiedlicher Repräsentationen für Ressourcen
- Zustandslose Kommunikation
- Verknüpfung von Ressourcen

Verwaltung von Ressourcen mit eindeutiger Kennung

- Architekturstil beschreibt Modellierung einer API für persistent datenhaltende Dienste
  - *CRUD: Create – Read – Update – Delete*
- API exponiert (Repräsentationen von) Ressourcen
  - Ressourcen können sowohl Objekte als auch Prozesse sein
    - Geschäftsobjekte (z.B. "Kunde", "Bestellung")
    - Geschäftsprozesse (z.B. "Stornierung einer Bestellung")
    - Supportprozesse (z.B. "Erstellen eines Backups")
  - Repräsentationen
    - Projektion und Aggregation
    - Unterschiedliche Datenformate
- Jede Ressource besitzt eine global eindeutige ID
  - Globales Namensschema bei RESTful HTTP: URI
    - siehe Modul "Web-Technologien"

Verwendung von einheitlichen Standardmethoden

- Interaktion mit allen Ressourcen erfolgt über den gleichen Satz von Standardmethoden
  - RESTful HTTP: HTTP-Verben `POST`, `GET`, `PUT`, `DELETE`, ...
- Beschränkung auf diese Methoden nur scheinbar ein Nachteil, da auch Prozesse (z.B. "Bestellung stornieren") als Objekte modelliert werden können
  - `DELETE /orders/4711`
  - `POST /cancellations`
    `{ orders: [4711, 1377], reason: "Passt nicht." }`
- Gutes Verständnis und korrekter Einsatz des Protokolls (HTTP) sind essentiell
  - Verben, Status-Codes, Header-Felder
  - siehe Modul "Web-Technologien"

Unterstützung unterschiedlicher Repräsentationen

- Clients teilen bei jeder Anfrage mit, welche Repräsentationen der Ressourcen (=Datenformate) sie verarbeiten können
- Server liefert – falls unterstützt – Ressource im gewünschten Format
- HTTP: *Content Negotiation* mithilfe von MIME-Typen
  - Bei Anfrage im Header `Accept`
  - Bei Antwort im Header `Content-Type`
  - Bei nicht-unterstütztem Format: Statuscode `406`

```
GET /orders/4711 HTTP/1.1
Host: shop.com
Accept: application/json
Accept-Encoding: utf-8

HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
...

GET /orders/4711 HTTP/1.1
Host: shop.com
Accept: text/xml
...

GET /orders/4711 HTTP/1.1
Host: shop.com
Accept: text/html
...

HTTP/1.1 406 Not Acceptable
...
```

Zustandslose Kommunikation

- Jede Anfrage enthält alle Informationen, die benötigt werden, um die Anfrage zu verstehen
- Sitzungszustand oder –historie wird niemals auf dem Server gespeichert
- Lose Kopplung von Client und Server ermöglicht hohe Skalierbarkeit (z.B. Lastverteilung auf mehrere Server-Instanzen)
- Verwaltung und Kommunikation eines Sitzungszustands (z.B. Authentifizierungs-Token oder Warenkorb-Inhalt) trotzdem oft nötig
  - Speicherung ausschließlich Client-seitig, z.B. mithilfe der Local Storage API
  - Mitsenden der benötigten Information in jeder Anfrage (als Teil der URI oder in einem Header)

Verknüpfung von Ressourcen

- *Hypermedia As The Engine Of Application State (HATEOAS)*
- Umsetzung dieses Prinzip gilt als höchster Reifegrad für REST-Systeme
  - *Richardson Maturity Model (RMM)* Level 3
- Ziel: API soll von einem generischen Client, d.h. ohne implizites Wissen des Client über die konkrete API, benutzt werden können
  - Vgl. HTML-Formulare im Browser
  - I.d.R. nur eine Ressource als Einstiegspunkt bekannt
- Verknüpfung von Ressourcen und Operationen auf Ressourcen über das global eindeutige Namensschema sowie den einheitlichen Methodensatz
  - Server fügt bei jeder Anfrage verknüpfte Ressourcen und Operationen (sog. Relationen) in die Antworten ein
  - Damit auch Verknüpfungen über die Grenzen des Systems hinweg möglich

Verknüpfung von Ressourcen

```
GET /orders/4710 HTTP/1.1

HTTP/1.1 200 OK
...
{
   customer: {
      href: "http://crm.service.com/customers/42"
   },
   items: [ {
      product: { href:
         "http://shop.com/products/1337" },
      amount: 3
   }, {
      product: { href:
         "http://shop.com/products/2048" },
      amount: 1
   }],
   _links: {
      self: {
         href: "http://shop.com/orders/4710"
      }
      update: {
         method: "PUT",
         href: "http://shop.com/orders/4710"
      },
      cancel: {
         method: "POST",
         href: "http://shop.com/cancellations"
      },
      list: {
         href: "http://shop.com/orders"
      }
   }
}
```

Verknüpfung von Ressourcen

```
GET /orders/4711 HTTP/1.1

HTTP/1.1 200 OK
...
{
   customer: {
      href: "http://crm.service.com/customers/21"
   },
   items: [ {
      product: { href:
         "http://shop.com/products/1024" },
      amount: 1
   }],
   _links: {
      self: {
         href: "http://shop.com/orders/4711"
      },
      list: {
         href: "http://shop.com/orders"
      }
   }
}
```

Entwurf

Vorgehen

- Fachliche Anforderungserhebung ist abgeschlossen
- Anschließend Entwurf der RESTful API
    1. Identifikation von Ressourcen
    2. URI-Entwurf
    3. Relationsentwurf
    4. Auswahl der unterstützten Repräsentationsformate
    5. Modellierung der Ressourcen-Repräsentationen
    6. Einfügen der Relationen

Dokumentation

- URI-Design: Tabelle mit
  - Ressourcen
  - relativen URIs
    - URI-Templates (https://tools.ietf.org/html/draft-gregorio-uritemplate-05), z.B. `/orders/{id}`
    - Express-Notation, z.B. `/orders/:id`
  - explizit unterstützten HTTP-Verben
    - Implizit: `HEAD`, `OPTIONS`
- Relations-Design: Zustandsdiagramm mit
  - Ressourcen (Knoten)
  - Relationen (Kanten)

Dokumentation

- Bei APIs, die öffentlich produktiv eingesetzt werden sollen, sollte der Einsatz von semiformalen Methoden und Werkzeugen zur Dokumentation von APIs in Betracht gezogen werden
  - OpenAPI/Swagger (http://swagger.io)
  - API Blueprint (https://apiblueprint.org)
  - ...
- Vorteile
  - Relativ weit verbreitete De-facto-Standards
  - Umfangreiche Werkzeug-Unterstützung, z.B.
    - Editoren (https://editor.swagger.io)
    - Generatoren
      - HTML-Dokumentation (https://swagger.io/tools/swagger-ui)
      - Client- und Server-Stubs für zahlreiche Umgebungen (https://swagger.io/tools/swagger-codegen)
      - Automatisierte Test-Suites

Beispiel

- Stark vereinfachtes Beispielsystem aus dem Bestellwesen zur Demonstration von Entwurf und Implementierung von RESTful HTTP-APIs
- Ergebnis der Anforderungserhebung
  - Verwaltung von Kunden: Anlegen, Aktualisieren, Löschen
  - Verwaltung von Produkten: Anlegen, Aktualisieren, Löschen
  - Verwaltung von Bestellungen: Anlegen, Aktualisieren des Status (storniert, versendet)

Identifikation von Ressourcen

- Primärressourcen
  - Primärressourcen oft übereinstimmend mit den Kern-Geschäftsobjekten oder den Datenbank-Entitäten (z.B. Kunden, Produkte, Bestellungen, …)
- Subressourcen sind Teile einer Primärressource, die von der REST-Schnittstelle nicht als Ressourcen mit eigener Identität exponiert werden (z.B. Bestellpositionen, Adressen)
  - In der Datenhaltung können diese sehr wohl eigenständige Datensätze darstellen

Identifikation von Ressourcen

- Listenressourcen
  - Listen von Primärressourcen
  - Dient auch als Endpunkt zur Erstellung neuer Primärressourcen, z.B.

```
POST /orders HTTP/1.1
{
   customer: { href: "http://shop.com/customers/42" },
   items: [ { product: { href: "http://shop.com/products/1337" }, amount: 3 }
}
```

  - Verfeinerung möglich
    - Filterung, z.B. `GET /orders?state=cancelled`
    - Paginierung, z.B. `GET /orders?count=20&page=3`

Identifikation von Ressourcen

Ressource		
Kunde		
Liste der Kunden		
Produkt		
Liste der Produkte		
Bestellung		
Liste der Bestellungen		
Liste der eingegangenen Bestellungen		
Liste der stornierten Bestellungen		
Liste der ausgelieferten Bestellungen		
Stornierung		
Liste der Stornierungen		

URI-Entwurf

- Festlegung der relativen URIs für alle Ressourcen
  - Ressourcen sind Objekte, also sollten keine Verben sondern Substantive enthalten
  - Wahl des Bezeichners ergibt sich oft aus dem fachlichen Entwurf
- Festlegung der unterstützten HTTP-Verben für alle Ressourcen
  - Verwendung des spezifischen HTTP-Verbs statt Beschränkung auf `GET` und `POST`
- Beispiel

```
GET /deleteOrder?id=4711

DELETE /orders/4711

POST /cancellations
{ orders: [4711, 1377], reason: "Passt nicht." }
```

HTTP 1.1-Verben

GET	Lesen einer Ressource
HEAD	Liefert nur den HTTP-Header einer Ressource ohne die eigentliche Repräsentation der Ressource im HTTP-Body z.B. zur Prüfung der Existenz, der Datenmenge oder des Änderungsdatums einer Ressource
POST	Anlegen einer neuen Ressource mit einer vom Server bestimmten URI; i.d.R. Verwendung auf Listenressourcen
PUT	Aktualisierung einer bestehenden Ressource (oder - weniger empfehlenswert - Anlegen einer neuen Ressource mit einer vom Client bestimmten URI, d.h. Verwendung auf URI einer bisher nicht existenten Ressource)
DELETE	Löschen einer Ressource Logisches Löschen: Die korrespondierenden Daten in der Datenhaltung müssen nicht zwingend physisch gelöscht werden, auch Anpassung der Ressource (z.B. Setzen eines Flags) möglich
OPTIONS	Liefert erlaubte HTTP-Verben einer Ressource
TRACE	
CONNECT	

URI-Entwurf

Ressource	Relative URI	Unterstützte HTTP-Verben
Kunde	`/customers/:id`	`GET, PUT, DELETE`
Liste der Kunden	`/customers`	`GET, POST`
Produkt	`/products/:id`	`GET, PUT, DELETE`
Liste der Produkte	`/products`	`GET, POST`
Bestellung	`/orders/:id`	`GET, PUT`
Liste der Bestellungen	`/orders`	`GET, POST`
Liste der eingegangenen Bestellungen	`/orders?state=created`	`GET`
Liste der stornierten Bestellungen	`/orders?state=cancelled`	`GET`
Liste der ausgelieferten Bestellungen	`/orders?state=shipped`	`GET`
Stornierung	`/cancellations/:id`	`GET`
Liste der Stornierungen	`/cancellations`	`GET, POST`

URI-Entwurf

Ressource	Relative URI	Unterstützte HTTP-Verben
Kunde	`/customers/:id`	`GET, PUT, DELETE`
Liste der Kunden	`/customers`	`GET, POST`
Produkt	`/products/:id`	`GET, PUT, DELETE`
Liste der Produkte	`/products`	`GET, POST`
Bestellung	`/orders/:id`	`GET, PUT`
Liste der Bestellungen	`/orders`	`GET, POST`
Liste der eingegangenen Bestellungen	`/orders?state=created`	`GET`
Liste der stornierten Bestellungen	`/orders?state=cancelled`	`GET`
Liste der ausgelieferten Bestellungen	`/orders?state=shipped`	`GET`
Stornierung	`/cancellations/:id`	`GET`
Liste der Stornierungen	`/cancellations`	`GET, POST`

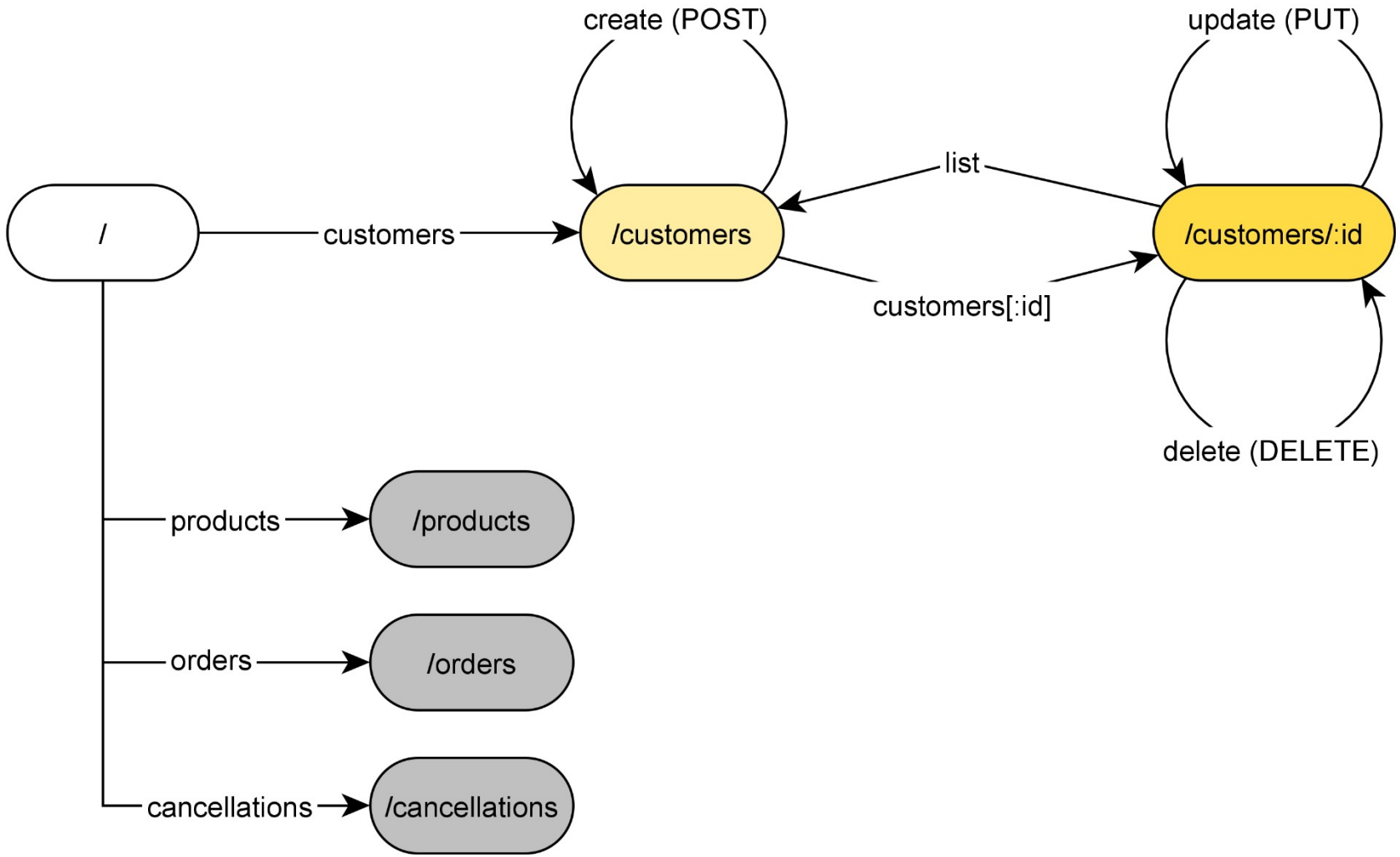
Relationsentwurf

- Grafische Notation als Zustandsdiagramm, z.B. in Anlehnung an die UML
- Knoten → Ressourcen
  - Beschriftung: Relative URI
- Kanten → Relationen
  - Beschriftung
    - Ggf. Bedingung für das Einfügen der Relation (sog. *guards*)
    - Name der Relation (beliebig wählbar)
    - HTTP-Verb (Standard: GET)
  - Beginn einer Relation in einer Ressource

    → Relation wird in Repräsentation der Ressource eingefügt
  - Ende einer Relation in einer Ressource

    → Relation referenziert Ressource, d.h. Anfrage ist an jene Ressource zu stellen (Nicht: Antwort enthält jene Ressource)

Relationsentwurf



Auswahl der unterstützten Repräsentationsformate

- Es existieren unzählige universelle, spezifische, proprietäre, strukturierte, unstrukturierte, Text- und Binär-Formate
  - Evtl. ergibt sich Wahl der Formate aus der Anforderungserhebung
- Universelle, hierarchisch strukturierte Textformate
  - Aktuell: JSON (`application/json`)
  - Klassiker: XML (`text/xml` bzw. `application/<derivat>+xml`)
  - Praktisch: HTML (`text/html`)
    - Erlaubt – korrekt umgesetzt – Verwendung der API im Browser
- In der Praxis oft Beschränkung auf ein Format

Modellierung der Ressourcen-Repräsentationen

- Für jede Kombination aus Ressource und Repräsentationsformat muss eine Repräsentationsvorlage (Template oder Template-Funktion) oder zumindest eine Beispielrepräsentation modelliert werden
- Herausforderungen ähnlich wie bei der "klassischen" Datenmodellierung für RDBMS, z.B.
  - Primär- vs. Subressource
  - Fachliche vs. technische Schlüssel
  - Minimale Redundanz vs. minimale Anfragenanzahl

Modellierung der Ressourcen-Repräsentationen

`/customers/:id`

```
{
  "customer": {
    "name": "Doe, John",
    "email": "john@doe.com",
    "address": {    // Subressource
      "line1": "Main Street 14",
      "line2": null,
      "line3": null,
      "postalcode": "1000",
      "city": "Washington, D.C.",
      "country": "USA"
    }
  }
}
```

Modellierung der Ressourcen-Repräsentationen

/customers

```
{
  "customers": [
    {
      "name": "Wurst, Hans",  // redundant
      "href": "http://shop.com/customers/0"
    },
    {
      "name": "Doe, John",
      "href": "http://shop.com/customers/1"
    },
    {
      "name": "Doe, Jane",
      "href": "http://shop.com/customers/2"
    }
  ]
}
```

Modellierung der Ressourcen-Repräsentationen

```
/orders/:id
```

```
{
  order: {
    customer: { href: "http://shop.com/customers/42" },
    items: [{       // redundanzfrei
      product: { href: "http://shop.com/products/1337" },
      amount: 3
    }, {
      product: { href: "http://shop.com/products/2048" },
      amount: 1
    }],
    status: "created"
    creationDate: "…",
    shippedDate: null
  }
}
```

Modellierung der Ressourcen-Repräsentationen

/orders

```
{
  orders: [
    { href: "http://shop.com/orders/1" },
    { href: "http://shop.com/orders/2" },
    ...
    { href: "http://shop.com/orders/7" },
    ...
    { href: "http://shop.com/orders/12" },
    { href: "http://shop.com/orders/13" },
    ...
  ]
}
```

Modellierung der Ressourcen-Repräsentationen

```
/orders?state=cancelled
```

```
{
  orders: [
    { href: "http://shop.com/orders/7" }
  ]
}
```

Modellierung der Ressourcen-Repräsentationen

```
/cancellations/:id
```

```
{
  cancellations: [{
    href: "http://shop.com/orders/7",
    reason: "Passt nicht."
  }]
}
```

Einfügen der Relationen

- In allen Ressourcen-Repräsentationen müssen die Relationen gemäß Relationsentwurf ergänzt werden
- Noch kein allgemein akzeptiertes Format etabliert
  - JSON for Linked Documents (JSON-LD)
  - Hypertext Application Language (HAL)
  - Collection+JSON
  - SIREN
  - JSON:API
  - …
- Hier: HAL (mit proprietärer, rückwärtskompatibler Ergänzung um Eigenschaft `method`)
  - Zusätzliche Eigenschaft `_links`
    - Schlüssel: Name der Relation
    - Wert: Objekt mit Schlüsseln
      - `href` (URI)
      - `method` (HTTP-Verb, entfällt bei `GET`)

Einfügen der Relationen

```
GET /customers
```

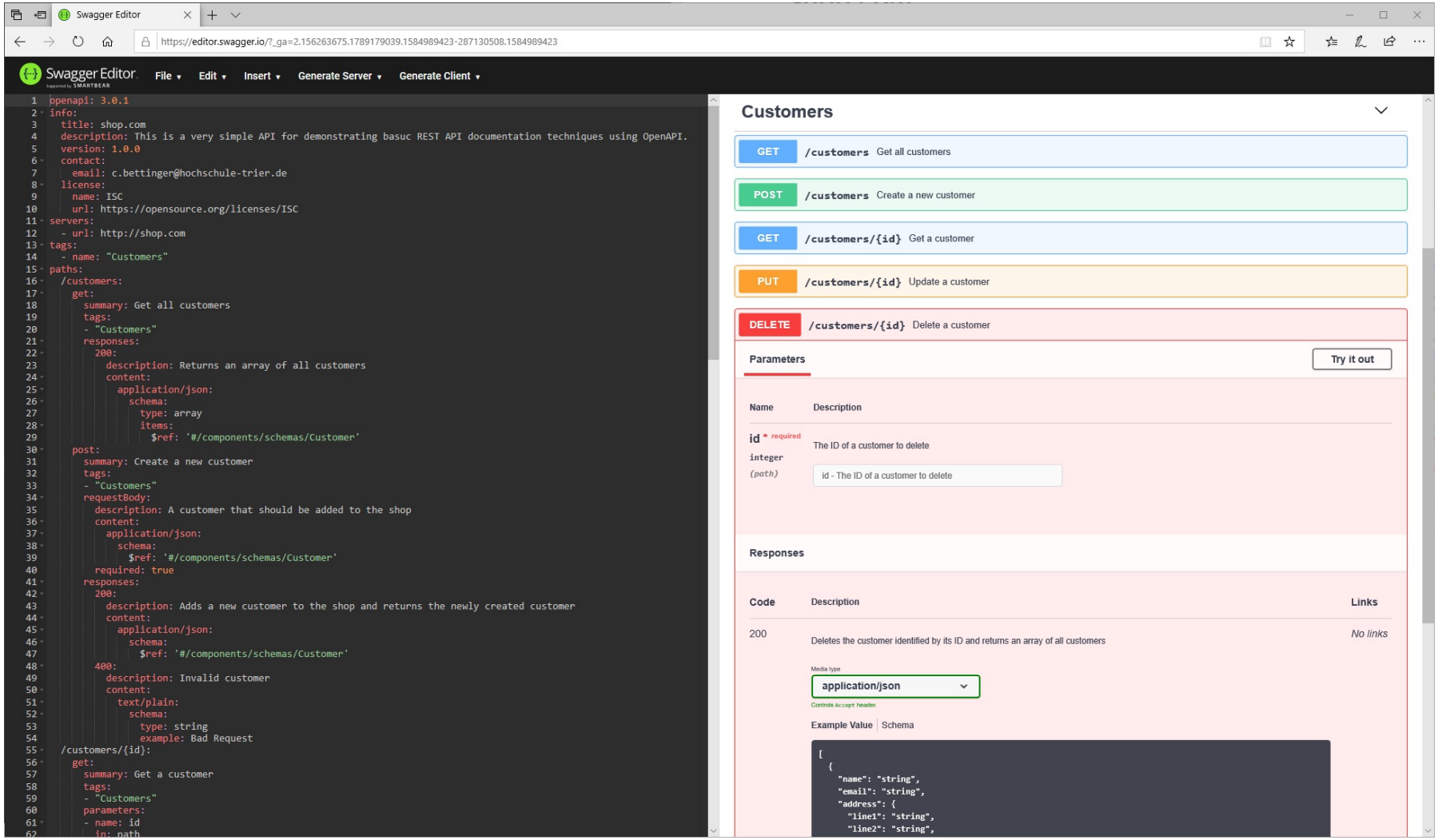
```
{
  "customers": [ { … }, { … }, { … }, … ],
  "_links": {
    "self": { "href": "http://shop.com/customers" },
    "create": { "method": "POST", "href": "http://shop.com/customers" }
  }
}
```

Einfügen der Relationen

```
GET /customers/:id
```

```
{
  "customer": { … },
  "_links": {
    "self": { "href": http://shop.com/customers/:id },
    "update": { "method": "PUT", "href": http://shop.com/customers/:id },
    "delete": { "method": "DELETE", "href": http://shop.com/customers/:id },
    "list": { "href": http://shop.com/customers }
  }
}
```

OpenAPI – Swagger Editor



OpenAPI: Metadaten

```
openapi: 3.0.1
info:
  title: shop.com
  description: This is a very simple API for demonstrating basuc REST API documentation techniques using OpenAPI.
  version: 1.0.0
  contact:
    email: c.bettinger@hochschule-trier.de
  license:
    name: ISC
    url: https://opensource.org/licenses/ISC
servers:
  - url: http://shop.com
tags:
  - name: "Customers"
```

OpenAPI: Schemata

```
components:
  schemas:
    Customer:
      type: object
      properties:
        name:
          type: string
        email:
          type: string
        address:
           type: object
```

```
properties:
      line1:
        type: string
      line2:
        type: string
      line3:
        type: string
      postalcode:
        type: string
      city:
        type: string
      country:
        type: string
```

OpenAPI: Pfade und Operationen

```yaml
paths:
  /customers:
    get:
      summary: Get all customers
      tags:
      - "Customers"
      responses:
        200:
          description: Returns an array of all customers
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/Customer'
    post:
      summary: Create a new customer
      tags:
      - "Customers"
      requestBody:
        description: A customer that should be added to the shop
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Customer'
        required: true
      responses:
        200:
          description: Adds a new customer to the shop and
               returns the newly created customer
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Customer'
        400:
          description: Invalid customer
          content:
            text/plain:
              schema:
                type: string
                example: Bad Request
```

OpenAPI: Pfade und Operationen

```
/customers/{id}:
  get:
    summary: Get a customer
    tags:
    - "Customers"
    parameters:
    - name: id
      in: path
      description: The ID of a customer to return
      required: true
      schema:
        type: integer
        format: int64
```

```
responses:
  200:
    description: Returns a customer identified
        by its ID
    content:
      application/json:
        schema:
          $ref: '#/components/schemas/Customer'
  404:
    description: Invalid customer ID
    content:
      text/plain:
        schema:
          type: string
          example: Not found
```

Realisierung mit Express

Node.js und Express

- Kombination aus Node.js und Express bietet alle nötigen Features zur Umsetzung von RESTful HTTP-APIs
  - Siehe Vorlesungseinheit "Serverseitige Anwendungen mit Node.js"
- Bei sorgfältigem Entwurf leichte Umsetzung
- Relationsentwurf und damit die Repräsentationen der Ressourcen enthalten nur zulässige Relationen – ein produktiv eingesetzter Server sollte aber robust gegenüber jeglicher Interaktion mit den Ressourcen implementiert sein (z.B. Anfrage zum Löschen nicht existierender Ressourcen)
  - Antwort mit HTTP-Statuscodes `4XX`
  - Express erzeugt implizit Handler für `OPTIONS`- und `HEAD`-Anfragen

Setup

```
const Customers = require("./Customers");

const PORT = 8080;
const BASE_URI = `http://localhost:${PORT}`;

const server = express();
server.use(bodyParser.json());

…

server.listen(PORT, () => {
  console.log("HTTP server listening on port %d.", PORT);
});
```

Einstiegspunkt

```
server.get("/", (request, response) => {
    response.json({
        _links: {
            self: { href: `${BASE_URI}` },
            customers: { href: `${BASE_URI}/customers` },
            products: { href: `${BASE_URI}/products` },
            orders: { href: `${BASE_URI}/orders` },
            cancellations: { href: `${BASE_URI}/cancellations` }
        }
    });
});
```

Listenressourcen

```
server.get("/customers", (request, response) => {
  response.json(createCustomerListBody ());
});
```

```
function createCustomerListBody () {
  return {
    customers: Customers.getAll().map(id => {
      return {
        name: Customers.get(id).name,
        href: `${BASE_URI}/customers/${id}`
      };
    }),
    _links: {
      self: {
        href: `${BASE_URI}/customers`
      },
      create: {
        method: 'POST',
        href: `${BASE_URI}/customers`
      }
    }
  };
}
```

Primärressourcen

```
server.get("/customers/:id", (request, response) => {
  const id = request.params.id;
  if (!Customers.exists(id)) {
    response.sendStatus(404);
  }
  else {
    response.json(createCustomerBody(id));
  }
});
```

Primärressourcen

```
function createCustomerResponse (id) {
  if (Customers.exists(id)) {
    return {
      customer: Customers.get(id),
      _links: {
        self: {
          href: `${BASE_URI}/customers/${id}`
        },
        update: {
          method: 'PUT',
          href: `${BASE_URI}/customers/${id}`
        },
        delete: {
          method: 'DELETE',
          href: `${BASE_URI}/customers/${id}`
        },
        list: {
          href: `${BASE_URI}/customers`
        }
      }
    };
  } else {
    return null;
  }
}
```

Primärressourcen

```
server.put("/customers/:id", (request, response) => {
    const id = request.params.id;
    if (!Customers.exists(id)) {
        response.sendStatus(404);
    }
    else {
        const updatedCustomer = request.body;
        Customers.update(id, updatedCustomer.name, updatedCustomer.email,
              updatedCustomer.address.country, updatedCustomer.address.postalcode,
              updatedCustomer.address.city, updatedCustomer.address.line1,
              updatedCustomer.address.line2, updatedCustomer.address.line3);
        response.json(createCustomerBody(id));
    }
});
```

Primärressourcen

```
server.delete("/customers/:id", (request, response) => {
    const id = request.params.id;
    if (!Customers.exists(id)) {
        response.sendStatus(404);
    }
    else {
        Customers.delete(id);
        response.json(createCustomerListBody());
    }
});
```

Erzeugen von Primärressourcen

- Als Ziel einer Relation zum Erzeugen von Primärressourcen dient i.d.R. nicht die URI der Primärressource, sondern die der dazugehörigen Listenressource
  - Einfügen einer untergeordneten Ressource mit einer vom Server bestimmtem Kennung via `POST`
- HTTP-Body der Anfrage enthält vollständigen Datensatz
- Antwort enthält
  - HTTP-Statuscode `201 (Created)` bzw. `400 (Bad Request)`
  - Ggf. URI der neu erzeugten Primärresource im HTTP-Header `Location`
  - Ggf. Repräsentation der neu erzeugten Primärressource im HTTP-Body

Erzeugen von Primärressourcen

```
server.post("/customers", (request, response) => {
    const newCustomer = request.body;
    if (!(newCustomer.name && newCustomer.email && newCustomer.address.country &&
          newCustomer.address.postalcode && newCustomer.address.city && newCustomer.address.line1)) {
        response.sendStatus(400);
    }
    else {
        const id = Customers.create(newCustomer.name, newCustomer.email, newCustomer.address.country,
            newCustomer.address.postalcode, newCustomer.address.city, newCustomer.address.line1,
            newCustomer.address.line2, newCustomer.address.line3);
        response.location(`${BASE_URI}/customers/${id}`).status(201)
            .json(createCustomerBody(id));
    }
});
```

Ausblick

Ausblick

- Unterstützung von mehr als einem Repräsentationsformat
  - Mehrere Body-Parser
  - Mehrere Formatter/Templates
- Sicherheitsaspekte
  - Betrieb eines HTTPS-Servers
  - Authentifizierung und Autorisierung
    - Unzählige "Strategien"
      - 80%-Lösung: HTTPS+HTTP Basic Authentication
      - OAuth 2.0-Protokoll zur Authentifizierung über Drittanbieter (z.B. Google, Amazon, Microsoft, ...)
    - Express-Middleware http://passportjs.org

Passport.js: HTTP Basic Authentication

```
passport.use(new BasicStrategy(
    (username, password, done) => {
        User.get({ username }, function (error, user) {
            if (error) { return done(error); }
            if (!user) { return done(null, false); }
            if (!user.isValidPassword(password)) { return done(null, false); }
            return done(null, user);
        });
    }
));
```

Passport.js: HTTP Basic Authentication

```
server.delete('/customers/:id',
    passport.authenticate('basic', { session: false }),
    (request, response) => {
        const id = request.params.id;
        if (!Customers.exists(id)) {
            response.sendStatus(404);
        }
        else {
            Customers.delete(id);
            response.status(200).json(createCustomerBody(id));
        }
    }
);
```

Fragen?