REST-Services: Best Practices

Web-Engineering II

Prof. Dr. Sebastian von Klinski

Zielsetzung

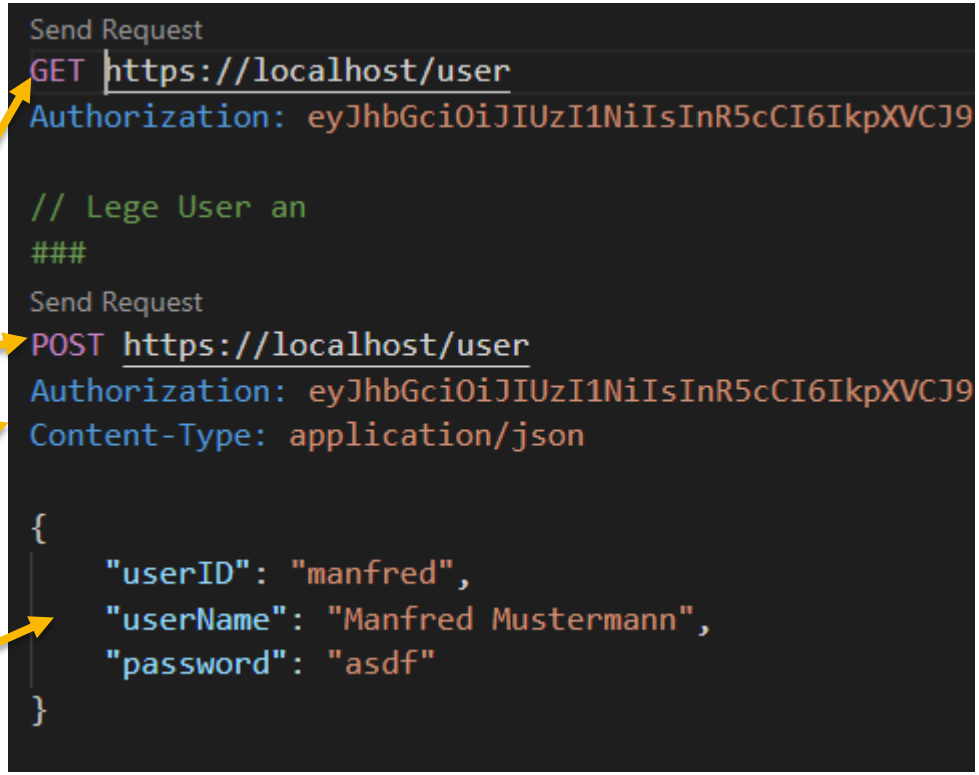
- REST-Services sollen flexibel durch unterschiedliches Clients genutzt werden
- Wesentliche Aspekte
  - IT-Sicherheit
  - Performante Ausführung
    - Anfragen müssen schnell beantwortet werden
    - Möglichst wenige Anfragen, um notwendige Daten zu bekommen
  - Einfache Nutzung:
    - Möglichst einheitliche Umsetzung → geringe Einarbeitung in Projekten

Grundlegender Aufbau: Request

HTTP-Methode

Header

Body (JSON)



Grundlegender Aufbau: Response

HTTP-Status

Header

Body (JSON)

```
HTTP/1.1 200 OK
X-Powered-By: Express
Access-Control-Allow-Origin: *
Access-Control-Allow-Headers: Origin, X-Requested-With, Content-Type, Accept
Access-Control-Expose-Headers: Authorization
Content-Type: application/json; charset=utf-8
Content-Length: 164
ETag: W/"a4-gLjIYRVmmH9dDTVV3THUhMDb7xQ"
Date: Sat, 08 Jan 2022 16:31:33 GMT
Connection: close

[
  {
    "userID": "admin",
    "userName": "Default Administrator Account",
    "isAdministrator": true
  },
  {
    "userID": "manfredTest",
    "userName": "Manfred Müller",
    "isAdministrator": false
  }
]
```

Relevante Aspekte bei Umsetzung

- Message-Protokoll (XML, JSON, andere)
  - Formatierung von Body
- Struktur des Requests
  - HTTP-Methode: GET, POST, PUT, DELETE
  - Struktur der URL
- Response
  - HTTP-Status-Code
    - Konsistente und aussagekräftige Status-Codes
  - Body
    - Enthält übertragene Daten
    - Minimierung von Anfragen, indem möglich alle notwendige Daten übertragen werden
- Umsetzung von Suchfunktionen
- Weiteres: Authentifizierung/ IT-Sicherheit/ Fehlermanagement

Message-Protokoll

Optionen: XML und JSON

- Message-Protokoll legt Format von Daten im Body fest
- Gängige Optionen: XML und JSON

```
POST /api/2.2/auth/signin HTTP/1.1
Content-Type:text/xml

<tsRequest>
  <credentials name="administrator" password="passw0rd">
    <site contentUrl="" />
  </credentials>
</tsRequest>
```

XML-REST-Call

```
POST /api/2.2/auth/signin HTTP/1.1
Content-Type:application/json

{
  "credentials": {
    "name": "administrator",
    "password": "passw0rd",
    "site": {
      "contentUrl": ""
    }
  }
}
```

JSON-REST-Call

Message-Protokoll

- XML
  - Umfangreiche Möglichkeiten für Validieren (XSS)
  - Sehr klare Struktur
  - Erzeugen und Parsen in Java-Script aufwändig
  - Sehr schnell sehr große Dokumente
  - Insbesondere auf Client-Seite Erzeugen von XML und Manipulation aufwändig
- JSON
  - Im Vergleich zu XML knappe Syntax und deutlich kleinere Dokumente/ Nachrichten
  - JavaScript:
    - eingebaute Funktionen zum Lesen
    - Erzeugen/ Marshalling und Un-Marshalling (JSON ⇔ Objekt/ Array)

→ In der Regel wird JSON als Message-Protokoll verwendet

Message-Protokoll

- Message-Protokoll wird im Header angegeben
- Message-Protokoll sollte bei Requests und Response festgelegt sein!

```
POST /api/2.2/auth/signin HTTP/1.1
Content-Type:application/json

{
   "name": "administrator",
   "password": "passw0rd"
}
```

Requests

Beispiele für Requests

Auflisten aller User

```
GET https://localhost/users
Authorization: {{adminToken}}
Content-Type: application/json
```

Anlegen eines Users

```
POST https://localhost/users
Authorization: {{adminToken}}
Content-Type: application/json

{
    "userID": "manfred",
    "userName": "Manfred Mustermann",
    "password": "asdf"
}
```

Ändern eines Users

```
PUT https://localhost/users/manfred
Authorization: {{adminToken}}
Content-Type: application/json

{
    "userName": "Manfred Müller"
}
```

Löschen eines User

```
DELETE https://localhost/users/manfred
Authorization: {{adminToken}}
```

Wesentliche Aspekte: HTTP-Methode, URL und Body-Content

HTTP-Methoden

- GET
  - Zum Abrufen von Ressourcen/ Daten
  - Sollten nie Daten ändern
  - Sollten nie Body übertragen
- POST
  - Zum Anlegen von (neuen) Ressourcen/ Daten
- PUT
  - Zum Ändern von existierenden Daten
- DELETE
  - Zum Löschen von Daten
  - Sollten nie Body übertragen
- PATCH
  - Partielle Updates von Entitäten

HTTP-Methoden: PUT versus PATCH

- Die klassische Festlegungen für PUT und PATCH sind problematisch
  - PUT für Anlegen oder vollständiges Ändern von Ressourcen
  - PATCH für partielles Ändern von Ressourcen
- PUT zum Anlegen wäre nicht sinnvoll, weil dafür POST vorgesehen ist
- Ein partielles Aktualisieren kann auch mit PUT umgesetzt werden, indem nur zu ändernde Attribute übertragen werden
- In Regel sind alle Update partielle Updates
  - Bei Entitäten gibt es häufig Attribute, die durch den Server abgeleitet werden (creatingUser, creationDate, status, etc.)
  - Selbst bei einem PUT würde man diese nicht überschreiben
- Unter diesen Aspekten wäre gleichzeitige Umsetzung von PUT und PATCH nicht sinnvoll, da weitgehend gleiche Umsetzung für 2 Endpoints

HTTP-Methoden: PUT versus PATCH

- PATCH wurde unter anderem eingeführt, um Bandbreite zu sparen
  - Bandbreite ist heute kein Thema mehr
- Bei NoSQL-Datenbanken kann selektives leichter umgesetzt werden, weil es direkt an ORD-Bridge übergeben werden kann

```
PATCH /products/234234
{ “stock: { “color": “red", "instock": “34" } }
```

- Persönliche Empfehlung
  - Bei einfachen Entitäten partielle und vollständige Updates mit PUT umsetzen
  - Wenn NoSQL-Datenbank im Backend und komplexe Updates von Sub-Collections notwendig, PATCH ergänzen

URL

- Bezeichnung und Struktur sollte systematisch sein
- Für das Verwalten von Daten (User, Artikel, Nachrichten, etc.) sollten Substantive verwendet werden
- Es sollte der Plural verwendet werden
  - Beispiele: „/users“, „/articles“, „/messages“
  - Endpoints verwalten stets Sammlungen von Entitäten/ Ressourcen → Plural verwenden
- Für gängie CRUD-Methoden sollten keine Verben benutzt werden
  - Z.B. /getUser
  - Stattdessen sollten HTTP-Methoden genutzt werden (z.B. GET)
- Für Operationen/ Berechnungen sollten Verben benutzt werden
  - Beispiele: „/login“, „/logout“, „calculateTemperatur“, etc.
- Es sollte Camel-Case verwendet werden (z.B. „/calculateTemperatur“)

Body

- Body sollte nur bei POST und PUT übergeben werden
- Struktur bildet in der Regel Objekt ab (Attribute und Werte)

```
POST https://localhost/users
Authorization: {{adminToken}}
Content-Type: application/json

{
    "userID": "manfred",
    "userName": "Manfred Mustermann",
    "password": "asdf"
}
```

Requests

- Gängige Request
  - CRUD-Methoden (Create, Read, Update, Delet)
  - Suchfunktionen
  - Anwendungsprozess (z.B. Sende E-Mail an alle User)
- CRUD-Methoden
  - Alle Entitäten abrufen (z.B. alle User)
  - Eine Entität abrufen (z.B. speziellen User)
  - Eine Entität anlegen (z.B. neuen User anglegen)
  - Eine Entität ändern (z.B. beim User Passwort ändern)
  - Eine Entität löschen (z.B. User löschen)
- Suchfunktionen
  - In der Regel Abfragen mit gestaffelten Suchkriterien (z.B. Suche alle User, die nicht aktiv sind)

CRUD Methoden: Auflisten von allen Objekten

```
GET https://localhost/users
Authorization: {{adminToken}}
Content-Type: application/json
```

- HTTP-Methode GET
- Header
  - Content-Type sollte angeben, in welchem Format Ergebnis zurückgegeben werden soll
  - Authorization-Header für Requests, die Authentifizierung erfordern
- Beispiel für Umsetzung

```
app.get('/users', (req, res) => {
  …
  res.json(users);
});
```

CRUD Methoden: Abrufen von konkreten Objekt

- HTTP-Methode GET
- URL: Parameter für User-ID/ Unique-Identifier wird an URL angehängt
- Header
  - Content-Type sollte angeben, in welchem Format Ergebnis zurückgegeben werden soll
  - Authorization-Header für Requests, die Authentifizierung erfordern
- Beispiel für Umsetzung

```
### Abrufen eines konkreten Users per User-ID
Send Request
GET https://localhost/users/admin
Authorization: {{adminToken}}
```

```
app.get('/users/:userID', (req, res) => {
  …
  res.json(users);
});
```

CRUD Methoden: Abrufen von konkreten Objekt (2)

- Beispiel 2: Es soll ein konkretes Forum abgerufen werden
- Struktur: An URL wird ID des Forums angehangen

```
GET http://localhost:8080/forums/{{forumID}}
Content-Type: application/json
```

- Beispiel: Es soll das Forum mit der ID 619a5fb7b983591ed85a5799 abgerufen werden

```
GET http://localhost:8080/forums/619a5fb7b983591ed85a5799
Content-Type: application/json
```

```
router.get('/:forumID', function (req, res) {
  const forumID = req.params.forumID;
  …
  res.status(200).send(forum);
});
```

CRUD Methoden: Anlegen von Objekt

- HTTP-Methode POST
- Header
  - Content-Type von übergebenen Daten im Body
  - Authorization-Header für Requests, die Authentifizierung erfordern
- Beispiel für Umsetzung

```
POST https://localhost/users
Authorization: {{adminToken}}
Content-Type: application/json

{
    "userID": "manfred",
    "userName": "Manfred Mustermann",
    "password": "asdf"
}
```

```
app.post('/users', (req, res) => {
  …
  res.json(createdUser);
});
```

CRUD Methoden: Ändern von Objekt

```
PUT https://localhost/users/manfred
Authorization: {{adminToken}}
Content-Type: application/json

{
    "userName": "Manfred Müller"
}
```

- HTTP-Methode PUT
- URL: Parameter für User-ID/ Unique-Identifier wird an URL angehängt
- Header
  - Content-Type von übergebenen Daten im Body
  - Authorization-Header für Requests, die Authentifizierung erfordern
- Beispiel für Umsetzung

```
app.put('/users/:userID', (req, res) => {
  const { userID } = req.params;
        …
  res.json(modifiedUser);
});
```

CRUD Methoden: Löschen von Objekt

- HTTP-Methode DELETE
- URL: Parameter für User-ID/ Unique-Identifier wird an URL angehängt
- Header
  - Authorization-Header für Requests, die Authentifizierung erfordern
- Beispiel für Umsetzung

```
DELETE https://localhost/users/manfred
Authorization: {{adminToken}}
```

```
app.delete('/users/:userID', (req, res) => {
  const { userID } = req.params;
        …
  res.json(„Deleted User“);
});
```

Response (Antworten)

Response/ Antworten

- Die zurückgegebenen Antworten von REST-Server sollten ebenfalls so weit wie möglich standardisiert sein
- Systeme, bzw. Anwendungen müssen Antworten verstehen
- Keine Freitext-Antworten schicken (z.B. „Hat nicht funktioniert“)
- Besser
  - Standardisierte HTTP-Codes
  - Standardisierte Nachrichten im Body im JSON-Format

HTTP-Status-Codes: Grundlegende Struktur

- Übersicht: https://de.wikipedia.org/wiki/HTTP-Statuscode#1xx_%E2%80%93_Informationen
- **1xx**
  - Informations-Codes
  - Server meldet, dass Bearbeitung noch läuft
- **2xx**
  - Die Anfrage war erfolgreich, die Antwort kann verwertet werden.
- **3xx**
  - Um eine erfolgreiche Bearbeitung der Anfrage sicherzustellen, sind weitere Schritte seitens des Clients erforderlich.
- **4xx**
  - Die Ursache des Scheiterns der Anfrage liegt (eher) im Verantwortungsbereich des Clients
- **5xx**
  - Die Ursache des Scheiterns der Anfrage liegt jedoch eher im Verantwortungsbereich des Servers.

2XX-Success-Codes

- **200 OK**
  - Die Operation wurde korrekt ausgeführt
- **201 Created**
  - Wenn durch die Operation eine Ressource angelegt wurde
  - Z.B. beim Anlegen von Daten bei POST-Requests
- **202 Accepted**
  - Für asynchrone Anfragen
- **204**
  - Alles ok, aber kein Content
  - z.B. bei Delete

4XX-Error-Codes: Client-Fehler

- **400 Bad Request**
  - Anfrage ist fehlerhaft
- **401 Unauthorized**
  - Client ist nicht berechtigt, die Aktion auszuführen (z.B. weil Client nicht authentifiziert ist)
- **403 Forbidden**
  - Authentifzierter User hat keine Berechtigung, die Aktion auszuführen
- **404 Not Found**
  - Die Ressource gibt es nicht
- **415 Nicht unterstützter Medientyp**
  - Wenn der Medientyp (z.B. JSON oder XML) nicht untertützt wird

5XX-Error-Codes: Server-Fehler

- **500 Internal server error**
  - Generische Fehlermeldung des Servers
- **502 Bad Gateway**
  - Es gab einen Fehler von einem angegliederten Server
- **503 Service Unavailable**
  - Es gab einen unerwarteten Fehler auf dem Server wie zu lange Antwortzeiten eines anderen Dienstes, fehlender Zugang zur Datenbank etc.

Body-Struktur: Web-Message-Body-Style

- Grundsätzlich unterschieden werden
  - Einfache REST-Antworten (bare oder simple)
  - Eingebettete REST-Antworten (wrapped)
- Einfache REST-Antworten
  - Objekte als einfache JSON-Objekte oder
  - Arrays von JSON-Objekten
  - Üblicherweise empfohlen
- Wrapped REST-Antworten
  - Es werden nachgeordnete Objekte und weiterführende Daten mitgeliefert
  - Nachteil: engere Kopplung von Client und Server
  - Vorteil: weniger Anfragen, dynamische Client-Steuerung

Web-Message-Body-Style Simple: Objekt

Einfaches JSON-Objekt

```
HTTP/1.1 200 OK
X-Powered-By: Express
Access-Control-Allow-Origin: *
Access-Control-Allow-Headers: Origin, X-Requested-With, Content-Type, Accept
Access-Control-Expose-Headers: Authorization
Content-Type: application/json; charset=utf-8
Content-Length: 77
ETag: W/"4d-xGn7SdN/PmUnDa964eXIXP/poX0"
Date: Sun, 23 Jan 2022 16:38:48 GMT
Connection: close

{
  "userID": "manfred2",
  "userName": "Manfred Mustermann",
  "isAdministrator": false
}
```

Web-Message-Body-Style Simple: Array

- Bei einfachen Antworten sollte direkt der Array zurückgegeben werden

```
HTTP/1.1 200 OK
X-Powered-By: Express
Access-Control-Allow-Origin: *
Access-Control-Allow-Headers: Origin, X-Requested-With, Content-Type, Accept
Access-Control-Expose-Headers: Authorization
Content-Type: application/json; charset=utf-8
Content-Length: 164
ETag: W/"a4-gLjIYRVmmH9dDTVV3THUhMDb7xQ"
Date: Sun, 23 Jan 2022 16:35:21 GMT
Connection: close

[
  {
    "userID": "admin",
    "userName": "Default Administrator Account",
    "isAdministrator": true
  },
  {
    "userID": "manfredTest",
    "userName": "Manfred Müller",
    "isAdministrator": false
  }
]
```

Web-Message-Body-Style Wrapped

- Antworten von REST-Server können umfangreich erweitert werden, um weitergehende Funktionen zu ermöglichen und Performance zu optimieren
- Weitergehende Funktionen
  - Detaillierte Fehlermeldungen
  - Weitere Aktionen (siehe HATEOAS)
  - Angaben zu Rechten (Schreiben, Löschen, etc.)
- Performance optimieren
  - Nachgeordnete Daten mit einbinden (z.B. Forum-Thread mit Nachrichten)

Web-Message-Body-Style

- Der Body-Message-Style sollte per Header übergeben werden, weil ansonsten alle URLs gedoppelt werden müssten
- Beispiel

```
### Forum mit WebMessageBodyStyle Wrapped
Send Request
GET https://localhost/forumThreads/{{forumID}}
WebMessageBodyStyle: Wrapped
```

Web-Message-Body-Style Wrapped

- Beispiel für erfolgreiche Antwort

```
HTTP/1.1 200 OK
X-Powered-By: Express
Access-Control-Allow-Origin: *
Access-Control-Allow-Headers: Origin, X-Requested-With, Content-Type, Accept
Access-Control-Expose-Headers: Authorization
WebMessageBodyStyle: Wrapped
Content-Type: application/json; charset=utf-8
Content-Length: 388
ETag: W/"184-m4BSaOB27sAjP+5jpCzJFF7OREE"
Date: Mon, 24 Jan 2022 19:09:27 GMT
Connection: close

{
  "Success": "Retrieved Forum Thread by ID",
  "ResultObjectType": "ForumThread",
  "ResultObject": {
    "_id": "61eef67ac929ee88282d27ee",
    "forumName": "Mein erstes Forum",
    "forumDescription": "Das ist ein erstes Forum, das ich im Rahmen der Tests angelegt habe",
    "ownerID": "admin",
    "createdAt": "2022-01-24T18:56:58.982Z",
    "updatedAt": "2022-01-24T18:56:58.982Z",
    "__v": 0
  },
  "Rights": {
    "Edit": true,
    "Delete": false
  }
}
```

Web-Message-Body-Style Wrapped

- Beispiel für Fehlerantwort

```
HTTP/1.1 404 Not Found
X-Powered-By: Express
Access-Control-Allow-Origin: *
Access-Control-Allow-Headers: Origin, X-Requested-With, Content-Type, Accept
Access-Control-Expose-Headers: Authorization
WebMessageBodyStyle: Wrapped
Content-Type: application/json; charset=utf-8
Content-Length: 260
ETag: W/"104-VFRrBieaz/DxpWdVVC2jXehtXZE"
Date: Mon, 24 Jan 2022 19:11:16 GMT
Connection: close

{
  "Error": "Es ist ein Fehler beim Abrufen des Forums 234523452345 aufgetreten: Did not find forum threa
  "userErrorMessage": "Das Forum konnte nicht abgerufen werden.",
  "errorCode": 23,
  "errorHelpLink": "localhost://errordescriptions/23.html"
}
```

Web-Message-Body-Style

- Üblicherweise wird empfohlen, einfache JSON-Objekte zurückzugeben
- Komplexe Antwortstruktur führt zu enger Kopplung zwischen Client und Server
- Wechsel zwischen Services wäre schwieriger
- Aber…
  - Wenn es viele kleine Entitäten gibt, kann Kommunikation sehr aufwändig sein
  - Normalisierung bei relationalen Daten ist meist nicht empfehlenswert!
  - Vielen Anfragen mit kleinen Antworten brauchen viel Zeit
  - Eine Anfrage mit viel Daten braucht wenig Zeit
- Messge-Body-Style ist daher ein Kompromiss zwischen Performance und Kopplung!

Suchfunktionen

Suchfunktionen

- Suche nach Daten sollte möglichst einfach gestaltet werden
- Gängige Anforderung
  - Abrufen von Objekten, die einen bestimmten Wert haben
    - Beispiel: Es sollen alle User abgerufen werden, die Administrator sind
  - Abrufen von nachgeordneten Objekten
    - Beispiel: Es sollen die Nachrichten in einem Forum abgerufen werden
  - Pagination
    - Es wird nur ein Teil alle Entitäten abgerufen

Abrufen von Objekten mit vorgegebenen Werten

- Beispiel: Es sollen die User abgerufen werden, die „manfred“ in der User-ID haben und nicht Administrator sind

```
### Auflisten aller Nutzer mit Token(Admin)
Send Request
GET https://localhost/users?userID="manfred"&isAdministrator=false
Authorization: {{adminToken}}
```

- Suche kann auf zwei Arten umgesetzt werden
  - Nutzen der Suchfunktionen von Mongoose
  - Abrufen von allen Entitäten und dann filtern der Ergebnisse
- Empfohlen Vorgehensweise hängt von Suchwerten und Komplexität der Anfrage ab
- In der Regel ist Suche über Mongoose schneller

Abrufen von Objekten mit vorgegebenen Werten

- Beispiel: Es sollen die User abgerufen werden, die „manfred“ in der User-ID haben und nicht Administrator sind

```
async function findUsers(queryParameters, callback) {
    var users;
    if (queryParameters.userID) {
        const userID = queryParameters.userID
        const regex = new RegExp(userID, 'i') // i for case insensitive
        users = await User.find({ userID: { $regex: regex } })
    }
    else {
        users = await User.find()
    }
    if (queryParameters.isAdministrator != null) {
        if (queryParameters.isAdministrator == "true") {
            users = users.filter(user => user.isAdministrator == true);
        }
        else {
            users = users.filter(user => user.isAdministrator === false);
        }
    }
    …
```

Abrufen von Objekten mit vorgegebenen Werten (2)

- Beispiel 2 (siehe https://stackoverflow.blog/2020/03/02/best-practices-for-rest-api-design/)

```
app.get('/employees', (req, res) => {
  const { firstName, lastName, age } = req.query;
  let results = [...employees];
  if (firstName) {
    results = results.filter(r => r.firstName === firstName);
  }
  if (lastName) {
    results = results.filter(r => r.lastName === lastName);
  }
  if (age) {
    results = results.filter(r => +r.age === +age);
  }
  res.json(results);
});
```

Abrufen von nachgeordneten Objekten

- Beispiel: Es sollen die Nachrichten in einem Forum-Thread abgerufen werden

```
### Forumnachrichten für einen bestimmten Forum-Thread abrufen
Send Request
GET https://localhost/forumThreads/{{forumID}}/forumMessages
```

- HTTP-Methode GET
- URL: Parameter für Unique-Identifier wird an URL angehängt, anschließend die gesuchte Entität
- Beispiel für Umsetzung

```
router.get('/:forumID/forumMessages', function (req, res) {
  const forumID = req.params.forumID;
                …
});
```

Abrufen von nachgeordneten Objekten (2)

- Beispiel: Abrufen von den Kommentaren eines konkreten Artikels

```
app.get('/articles/:articleId/comments', (req, res) => {
  const { articleId } = req.params;
  const comments = [];
  …
  res.json(comments);
});
```

- Es sollten nicht zu viele nachgeordnete Ebene abgebildet werden!
- Komplexität und Nutzbarkeit wird dann beeinträchtigt
- Nicht gut: /articles/:articleId/comments/:commentId/author
- Maximal 2-3 Ebenen

REST-Routen bei Node.js

- Express vergleicht die URLs des Request mit den definierten Routen von oben nach unten
- Beispiel: https://localhost/forumThreads/getByOwnerID
  - Es sind unter anderem die folgenden Routen implementiert
    - Route A: router.get('/:forumID', function (req, res) {
    - Route B: router.get('/getByOwnerID', AuthenticationUtils.isAuthenticated, function (req, res) {
  - Wenn Route A vor Route B ist, wird die Route „getByOwnerID“ nie aufgerufen, weil vorher immer die Route „/forumID“ gezogen wird!
  - Daher:
    - Routen mit spezifischen Namen müssen vor Routen mit dynamischen IDs definiert werden!

Sortieren

- Auch Sortierangaben können über URL angegeben werden
- Beispiel

```
http://example.com/articles?sort=+author,-datepublished
```

- + bedeutet in ansteigender Reihenfolge (A ist erster Buchstabe)
- - bedeutet in abfallender Reihenfolge (letzte Änderungen zuerst)

Pagination

- Produktivsysteme haben häufig so viele Daten, dass ein Abrufen alle Entitäten nicht mehr sinnvoll ist
  - Beispiel: Ein System hat ca. 5000 User
  - Das User-Management ruft die User immer in Blöcken a 20 User ab.
- Die Umsetzung kann über die Attribute „offset“ und „limit“ beschrieben werden
  - Offset: Beschreibt den Block
  - Limit: Beschreibt die Größe des Block

```
GET /users?offset=7&limit=20
```

Es soll der 7ten Block mit 20 Usern abgerufen werden

Pagination

- Beispiel

```
GET /users?offset=7&limit=20
```

- Es soll der 7ten Block mit 20 Usern abgerufen werden

- Mögliche Antwort
  - Wrapped Response
  - Angaben zum Block und Gesamtanzahl
  - Daten im Attribut „data“

```
{
  "pagination": {
    "offset": 7,
    "limit": 20,
    "total": 3465,
  },
  "data": [
    //...
  ],
 …
  }
}
```

Weiteres: Authentifizierung, IT-Sicherheit, Fehlermanagement, etc.

IT-Sicherheit

- Vertraulichkeit
  - Netzwerkprotokoll: HTTPS (HTTP keine Option)
- Authentifizierung
  - Token-basierte Authentifizierung
  - Für Single-Sign-On: OAuth2
- Autorisierung
  - Einfache Autorisierung fast immer notwendig
  - Ggfls. rollenbasierte Autorisierung
- Für jede Route sollte geprüft werden:
  - Ist Authentifizierung notwendig?
  - Ist Autorisierung notwendig?
    - Dürfen alle authentifizierte User die Aktion durchführen?
    - Dürfen nur bestimmte User die Aktion durchführen?

Fehlermanagement

- Häufige Fehler
  - Wenn unerwarteter Fall eintritt, wird keine Antwort geschickt
  - Es werden mehrere Antworten geschickt
- Es sollten immer alle möglichen Fälle abgefangen werden

```
router.get('/:userID', AuthenticationUtils.isAuthenticated, function (req, res) {
  const userID = req.params.userID;
userService.getUserByUserID(userID, function (err, result) {
    if (err) {
      res.status(500).json({ "Error": err });
    }
    else {
      if (result) {
        res.status(200).send(result);
      }
      else {
        res.status(404).json({ "Error": "Es konnten keine User gefunden werden." });
      }}});
});
```

Versionierung

- REST-Services, die über eine längere Zeit genutzt werden, verändern sich über die Zeit hinweg
  - Zusätzliche Entitäten
  - Assoziationen werden verändert,
  - Etc.
- Wenn bereits Anwendungen auf REST-Server zugreifen, ist Anpassung von REST-Services nicht unproblematisch
  - Nachfolgende Systeme funktionieren nicht mehr, weil Nachrichtenstruktur und Service-Struktur sich verändert haben
- Mögliche Umgangsweise mit Änderungen/ Versionen
  - Keine Versionsverwaltung
  - Versionen über URLs abbilden
  - Versionen über Header-Parameter

Versionierung: Keine Versionierung

- Endpoints werden ohne Versionsangaben weiterentwickelt
- Vertretbar, wenn es im wesentlichen nur einen Client gibt (z.B. Web-Anwendung mit REST-Server)
- In der Regel keine Fehler im Client, solange nur Felder/ Daten hinzukommen (z.B. beim User-Objekt wird das Feld E-Mail hinzugefügt)
- Problematisch für Clients ist…
  - Wenn Felder weggenommen werden (weil die Daten in eine anderen Entität verschoben wurden)
  - Wenn die Beziehungen zwischen Entitäten sich ändern
  - Wenn Entitäten umbenannt werden oder ganz entfernt werden

→ Nur vertretbar, wenn REST-Server und Client ein Produkt sind und die Entwicklungsteams eng zusammenarbeiten

Versionierung: URI-Versionsverwaltung

- Gängiger Ansatz für Versionierung: es wird eine Versionsnummer in URI aufgenommen

```
https://api.kundendatenbank.com/v1/customers/
https://api.kundendatenbank.com/v2/customers/
```

- Vorteile
  - Alte Services können erhalten bleiben, neue werden unter veränderter URL angeboten (zumindest für Übergangsphase)
  - Einfaches Hinzufügen und Entfernen von weiteren Versionen
- Nachteile
  - Schnelle Vermehrung der Endpoints
  - Duplizierung des Codes, selbst wenn nur geringe Änderungen bestehen
  - Wenn Version nicht mehr unterstützt wird, fällt Endpoint-Version ganz weg (Server-Fehler)
  - Häufig gibt es gar nicht mehrere Versionen…

Versionierung: Beispiel

In URL wird v1, bzw. v2 eingefügt

```
app.get('/v1/employees', (req, res) => {
  const employees = [];
  // code to get employees
  res.json(employees);
});

app.get('/v2/employees', (req, res) => {
  const employees = [];
  // different code to get employees
  res.json(employees);
});
```

Versionierung: Wahl per Header-Parameter

- Version des Endpoints kann per Header-Parameter angegeben werden

```
GET https://api.kundendatenbank.com/customers HTTP/1.1
Custom-Header: api-version=1
```

- Vorteile
  - Ein Endpoint für alle Versionen (Zugriff für Clients einfacher)
  - Abbildung von Versionen häufig über Marshalling-Adapter umsetzbar → keine überflüssige Duplizierung von Code
  - Wenn Version nicht mehr unterstützt wird, kann trotzdem sinnvolle Antwort geschickt werden
- Nachteile
  - Umsetzung des Endpoints wird komplexer, weil unterschiedliche Versionen dort abgebildet werden müssen
- Weniger gebräuchlich als URL-Versionierung, weil in der Umsetzung etwas komplexer

HATEOAS

REST - HATEOAS

- Hypermedia as the Engine of Application State (HATEOAS)
- REST-Antworten sind angereichert mit weiteren Links und möglichen Aktionen

```
HTTP/1.1 200 OK
Content-Type: application/vnd.acme.account+json
Content-Length: ...

{
   "account": {
      "account_number": 12345,
                  "links": {
         "deposit": "/accounts/12345/deposit",
         "withdraw": "/accounts/12345/withdraw",
         "transfer": "/accounts/12345/transfer",
         "close": "/accounts/12345/close"
      }
   }
}
```

HATEOAS

- Der Ansatz ist, Dynamik in der Web-Anwendung über REST-Antworten zu steuern
- Mögliche weitergehende Informationen
  - Mögliche Aktionen für die betreffende Entität
  - Rechte für die Entität (z.B. Ändern, Löschen)
  - Nachgeordnete Daten (zur Vermeidung von weiteren REST-Calls)

REST: Umsetzung

- Die meisten REST-Services auf Level 2