JavaScript im Einsatz

REST

RESTful Services

REpresentational State Transfer

- Architektur zum Aufbau verteilter Systeme, die den Prinzipien aller webbasierten Systeme folgt

- REST ist ausschließlich eine Architektur
  > Web Services (SOAP-basiert) sind beides: Standard und Architektur

- Die Architektur beschreibt die Interaktion von Anwendungen gemäß dem webbasierten Interaktionsmodell
  > Verwendung von HTTP als Transportprotokoll
  > Feststehende Operationen
  > Interaktion mit „Ressourcen“ als Dreh- und Angelpunkt

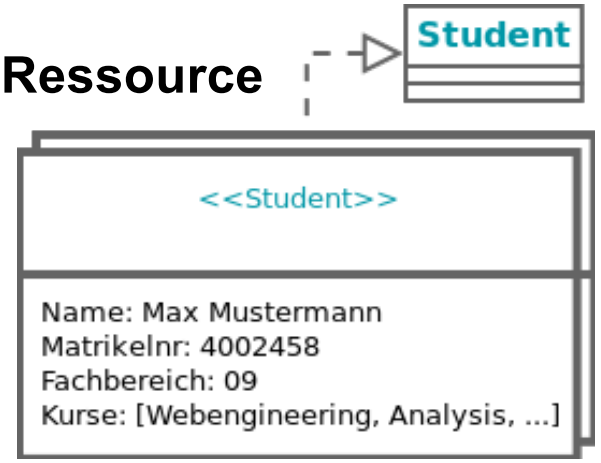
REST ist eine ressourcenorientierte Architektur (ROA)

RESTful Services
Konzept

URL

http://localhost/StudentAPI/studenten/4002458

Ressource



Repräsentation (JSON)


```
{
 „Name“: „Max Mustermann“,
 „Matrikelnr“: 4002458,
 „Fachbereich“: „09“,
 „Kurse“: [„Webengineering“,
             „Analysis“, …]
}
```

- Ressourcen (HTML-Seiten, Bilder, XML-Dokumente, **Objekte**) werden über URL angefordert
- Übertragen der Objekte in verschiedenen Repräsentationen möglich, z.B.
  GET StudentAPI/studenten/4002458
  als JSON-Objekt (andere Rückgabeformate möglich)

RESTful Services
Eigenschaften

Adressierbarkeit

- Jede Ressource ist über eine URL identifizierbar

Zustandslosigkeit

- Keine Anwendungszustände im Server (nur Ressourcenzustände)

Einheitliche Schnittstelle

- CRUD-Operationen (GET, POST, PUT, DELETE)

Entkopplung von Ressourcen und Repräsentation

- Anforderung einer Ressource im gewünschten Format (JSON, XML, HTML, …)

Hypermedia

- Verwendung von Hyperlinks zur Navigation

RESTful Services

Zustandslosigkeit

- Mit dem Ausliefern einer Repräsentation einer Ressource geht auch die **Verantwortung der Ressource** mit **an den Benutzer über**
- Der Server behält die Ressource zwar weiter vor, jedoch wird ein etwaiges Update konzeptionell von einem Benutzer initiiert
  - > Aufpassen: Andere Benutzer können durchaus die Ressourcen zwischenzeitlich ändern
  - > Zeitstempel/Versionierung der Ressource könnte hier sinnvoll sein
- Der Server behandelt jeden Aufruf als in sich abgeschlossenen Auftrag ohne impliziten Bezug auf vorhergehende Operationen
  - > Keine direkte Unterstützung von Transaktionen über Aufrufgrenzen hinweg
  - > Die Anwendung kann allerdings Attribute schaffen und Hypermedia verwenden, um Aufträge zu vernetzen
- Eigentlich sollte nur das Anlegen einer neuen Ressource nicht idempotent sein (Create ist nicht idempotent), PATCH (diff-Notation von JSON-Patch) ist aber nicht immer idempotent

Zustandslosigkeit und idempotente Operationen passen gut zusammen (At-least-Once-Semantik)

RESTful Services

Hypermedia

- Letztlich wollen wir nicht nur mit einzelnen Ressourcen arbeiten, wir wollen vernetzte Ressourcen und Prozesse abbilden
- Hypermedia liefert die Verlinkung von Ressourcen und kann so als wichtige Stütze für die Implementierung von Workflows dienen
- Ressourcen „pflegen“ also die Verlinkung mit anderen Ressourcen

Faktisch stützt die Verlinkung der Ressourcen den Aufbau von Arbeitsabläufen (Workflows)

Beispiel: Ein Warenkorb kann die Verlinkung auf eine „Bestellungen“-Ressource haben, die bei Create als Argument dann den zu bestellenden Warenkorb als JSON-Objekt erhält und so einen Bestellvorgang initiiert

RESTful Services
CRUD

Die Interaktion lässt sich mittels einer einfachen Abbildung auf die HTTP-Verben beschreiben

(CRUD-Operationen)

Action	Verb
Create	POST
Retrieve	GET
Update	PUT (PATCH)
Delete	DELETE

RESTful Services
CRUD

Create

- POST /resourceNames

Retrieve

- GET /resourceNames/resourceId

Update

- PUT /resourceNames/resourceId
- PATCH /resourceNames/resourceId

Delete

- DELETE /resourceNames/resourceId

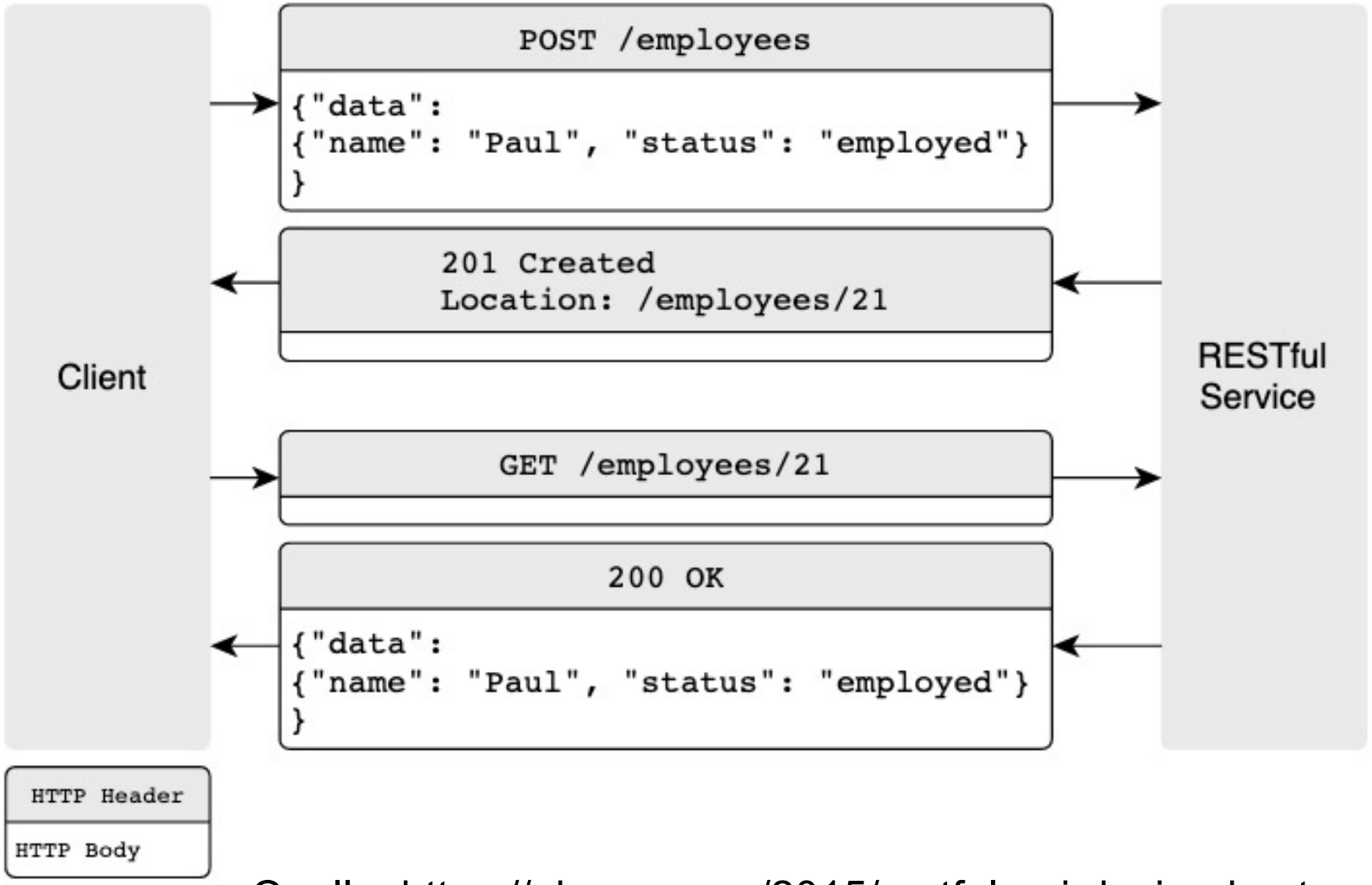
RESTful Services
Eigenschaften

- **HTTP als zentrales Protokoll zur Interaktion**
- **Im Mittelpunkt stehen die Ressourcen, die eindeutig adressierbar sind (URL / URI)**
  - > Ressourcen sind alles was man mit einem Namen bezeichnen kann
  - > Eine Naming authority weist den Ressourcen eine URI zu (oder Sie als Anwendungsentwickler)
- **Jede Ressource hat eine oder mehrere Repräsentationen (XML, Text, JSON, Bilder, mp3)**
  - > Repräsentationen stellen den Zustand einer Ressource dar, es handelt sich also quasi um eine flexible Objektserialisierung
  - > Besitzen ggf. Verweise auf andere Ressourcen (quasi Objektreferenzen)
  - > Ressourcen werden übertragen, manipuliert und wieder übertragen. Die Repräsentation ist dabei aushandelbar (**Content Negotiation**)
- **"ROA" (Resource-oriented Architecture)**

RESTful Services
Ressourcen

- Die genaue Struktur der verwendeten Ressourcen wird in REST nicht fest beschrieben
  - Wir müssen also nicht in explizit festgelegten Objektstrukturen denken, wie sie mittels XML-Schemata definiert werden, vielmehr eher wir in Script-Sprachen ohne feste Typisierung
  - Das eröffnet Freiheitsgrade, aber auch mögliche Probleme (wie genau ist die Struktur der Objekte?)
  - Prinzipiell kann man auch nur die wichtigen Teile eines Objektes erhalten (PATCH-Operation, Query-Parameter)
- Ressourcen sind oft denormalisiert
  - Man bekommt alles Wichtige mit einer einzelnen GET-Operation
  - Vermeidung externer Objektreferenzen
  - Passt zu der Vorstellung, dass alles was zum Ausführen der Operation notwendig ist auch mitgeliefert wird
  - Folgt der JSON-Idee der Embedded Documents
- Probleme mit der Nutzung der API: OpenAPI/Swagger

RESTful Services
Die Interaktion



Quelle: https://phauer.com/2015/restful-api-design-best-practices/

RESTful Services
Eigenschaften

- REST orientiert sich an **Nomen**. Damit sollte die URL auch nomenorientiert gebildet werden

  http://api.example.com/device-management/managed-devices
  http://api.example.com/device-management/managed-devices/{device-id}
  http://api.example.com/user-management/users/
  http://api.example.com/user-management/users/{id}

- Auf Basis von HTTP existieren 2 Möglichkeiten, die Anfragen zu parametrisieren. Die Parametrisierung ist auf Server-Seite bei der Programmierung wichtig zu berücksichtigen
  - **Query-Parameter** können als Argument der URL angehangen werden
    - ?Key1=Value1&Key2=Value2...
    - Vielfältige Parametrisierung der Operation auf Serverseite möglich
  - **Path-Parameter** fixieren dynamisch Ressourcen in der URL
    - Server verwaltet die Ressourcen, die unter einer URL verfügbar sind users**/{id}**
    - Create sollte daher die id nicht mitliefern, sondern dem Server überlassen
    - Ressourcen können auch verschachtelt sein

      http://api.example.com/device-management/managed-devices/{device-id}/parameter//{parm-id}

RESTful Services
URLs / Path-Params

- `GET /tickets/12/messages` - Retrieves list of messages for ticket #12
- `GET /tickets/12/messages/5` - Retrieves message #5 for ticket #12
- `POST /tickets/12/messages` - Creates a new message in ticket #12
- `PUT /tickets/12/messages/5` - Updates message #5 for ticket #12
- `PATCH /tickets/12/messages/5` - Partially updates message #5 for ticket #12
- `DELETE /tickets/12/messages/5` - Deletes message #5 for ticket #12

Quelle: https://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api#useful-post-responses

RESTful Services mit Fetch
Content Negotiation

- Der Client überliefert oder fordert eine Ressource in einer speziellen Repräsentation an
- Der Content-Type-HTTP-Header liefert dies
- Die Methode **`request.headers.get`** kann **auf Node.js-Seite** dazu genutzt werden, zu prüfen, in welchem Format die Ressource vorliegt/übertragen werden soll
  - Content-Type-Header (liefert hier ggf. Default aus)
  - Empfohlen: **Accept-Header**
- Auf **Browser-Seite** ist dies der **Content-Type mittels `response.headers.get`**
- Wenn der Server die angefragte Repräsentation nicht unterstützt, dann wird ein Fehler geworfen (Status 415)

RESTful Services mit Fetch

Content Negotiation mit dem http-Modul **im Browser**

```
async function getObj(url, requestparameter) {
    let response = await fetch( url, requestparameter, )
    const contentType = response.headers.get('content-type')
    if (response.status === 401)
         throw new Error('Request was not authorized.')
    if (contentType === null)
         return new Promise(() => null)
    else if (contentType.startsWith('application/json;'))
         return await response.json()
    else if (contentType.startsWith('text/plain;'))
         return awaitresponse.text()
    else throw
       new Error(`Unsupported response content-
         type:${contentType}‘)
}
async_getObj(https://abc.org/, { headers:{'Accept':'text/plain‘}}
    .then(receivedObject => console.log(receivedObject))
    .catch(err => console.log(err))
```

RESTful Services mit Fetch
Content Negotiation

Hinweis: Sobald wir mittels response.json auf das erhaltene Objekt zugegriffen haben ist es zu spät für eine Content-Negotiation:

```
fetch(url)
.then(response => response.json())
.catch(() => response.text())
        // TypeError: body stream already read
```

Hier hilft der Trick, auf die Objekte durch clone() zuzugreifen

```
fetch(url)
        .then(resp => resp.clone().json()
        .catch(() => resp.text()) )
        .then(data => {//data is now parsed JSON or raw text });
```

RESTful Services Server-Seite

Content Negotiation mit **express**

Bei express bietet sich die **res.format-Methode** an, die in Abhängigkeit vom **Accept-Header** des **Request**-Objekts arbeitet:

```
res.format({
        'text/plain': function () { res.send('hey') },
        'text/html': function () { res.send('<p>hey</p>') },
        'application/json': function (){ res.send({ message: 'hey'}) },
        'default': function () {
        // log the request and respond with 415
        res.status(415).send(,Unsupported Media Type') }
})
```

Verkürzte Notation **ohne vollständigen MIME-Namen** möglich

```
res.format({
        text: function () { res.send('hey') },
        html: function () { res.send('<p>hey</p>') },
        json: function () { res.send({ message: 'hey' }) } })
```

Abfrage des Content-Types auf Client-Seite mittels

```
res.get('Content-Type‘)     // => "text/plain"
```

RESTful Services Server-Seite

Content Negotiation mit der body-parser-Middleware

Zusammen mit express bietet sich gerade bei POST-Operationen die Nutzung des externen Moduls **body-parser** an. Diese Middleware kann die Verarbeitungsfunktionen über das req-Objekt vereinfachen und gleichzeitig auch Query-Parameter zugänglich machen

```
const express = require('express')
const bodyParser = require('body-parser')
let app = express()
// Verarbeitet application/json für alle Routen (POST, PUT, PATCH)!
app.use(bodyParser.json());

//Klassische Formulardaten application/x-www-form-urlencoded
app.use(bodyParser.urlencoded({ extended: true }));

// Annahme POST: name=foo&color=red <-- URL encoding der Query-Parms
// oder POST: {"name":"foo","color":"red"} <-- JSON encoding
app.post("/posturl",function(req,res,next){
        console.log(req.body);
        res.send("response");
})
```

RESTful Services
bodyParser

Never trust the user! Was für ein JSON-Objekt bekommen wir?

Note As `req.body` 's shape is based on user-controlled input, all properties and values in this object are untrusted and should be validated before trusting. For example, `req.body.foo.toString()` may fail in multiple ways, for example the `foo` property may not be there or may not be a string, and `toString` may not be a function and instead a string or other user input.

Quelle: https://www.npmjs.com/package/body-parser

`bodyParser.json` wird nur Content-Type json übernehmen. Das Middleware-Konzept des **bodyParsers erlaubt das Hinzufügen mehrerer Parser** (z.B. `urlencoded`) Hinweis:
Die Verwendung von **response.json** verarbeitet das response Objekt! Spätere Middleware kann dies also nicht noch einmal verarbeiten, `bodyParser.json` verarbeitet dieses nicht!

RESTful Services
Ein Server mit bodyParser und JSON-Verarbeitung

Wir arbeiten wie gewohnt mittels express

```
const express = require('express');
const app = express();
const bodyParser = require('body-parser');
let todos = [{id:1, title:'kaufe Milch'},
        {id:2, title:'mache Übungsaufgaben'},
        {id:3, title:'telefoniere mit Eltern'}];
app.use(bodyParser.json())
// gesendetes Todo: {"title": "Eis essen gehen"}
app.get('/todos', (req, res) => res.status(200).json(todos));
app.post('/todos', (req, res) => {
    let newTodo = {}
    newTodo.title = req.body.title;
    newTodo.id = todos.length + 1;
    todos.push(newTodo);
    res.setHeader('Location', '/todos' + newTodo.id);
    res.status(201).json();
});
```

RESTful Services
Ersatz für bodyParser

Seit Express 4.16 gibt es eine eigene Middleware in Express, die die Body-parser Funktionalität übernimmt:

Alt:

```
app.use(bodyParser.json())
```

Neu:

```
app.use(express.json()); // JSON-encoded bodies
```

Content-Type: application/json

Alt:

```
app.use(bodyParser.text())
```

Neu:

```
app.use(express.text()); // Text bodys
```

Content-Type: text/plain

Alt:

```
app.use(bodyParser.urlencoded({ extended: true }));
```

Neu:

```
app.use(express.urlencoded()); // Formulardaten parsen
```

Content-Type: application/x-www-form-urlencoded

RESTful Services
Parsing Body

Beispiel Text: `foo`

```
// Durch diese Middlewares kann diese Route einfachen Text verarbeiten
app.use(express.text());

app.post(,/post-url', function(req, res) {
        let name = req.body
        // ...
});
```

RESTful Services
Parsing Body

Beispiel URL-enkodierte Daten (Formulardaten): `name=foo&color=red`

Beispiel JSON-enkodierte Daten: `{"name":"foo","color":"red"}`

```
// Durch diese Middlewares kann diese Route sowohl JSON als auch URL enkodierte Daten verarbeiten
app.use(express.json());
app.use(express.urlencoded());

app.post(,/post-url', function(req, res) {
        let name = req.body.name;
        let color = req.body.color;
        // ...
});
```

RESTful Services
Never trust the User?

Problem: Wir können die Struktur des empfangenen JSON-Objektes nicht direkt mit instanceof prüfen, das liegt an den fehlenden Prototyp-Verbindungen. Wir sind immer vom Typ Object.

Mittels dem üblichen express-validator müssen alle Felder einzeln getestet werden.

Komfortabler sind die Ansätze, mittels JSON Schema und **express-json-validator-middleware**

RESTful Services
express-validation

```
const express = require('express')
const { validate, ValidationError, Joi } = require('express-validation')

const loginValidation = {
  body: Joi.object( schema: {
    email: Joi.string()
        .email()
        .required(),
    password: Joi.string()
        .regex( pattern: /[a-zA-Z0-9]{3,30}/)
        .required(),
  }),
}

const app = express();
app.use(express.json())

app.post( path: '/login', validate(loginValidation,  options: {},  joiRoot: {}), (req : ... , res : ... ) => {
  res.json( body: 200)
})

app.use(function(err, req, res, next) {
  if (err instanceof ValidationError) {
    return res.status(err.statusCode).json(err)
  }

  return res.status(500).json(err)
})

app.listen( port: 3000)
```

https://www.npmjs.com/package/express-validation

RESTful Services
Validierung über Schema

```
// Define a JSON Schema
var StreetSchema = {
    type: 'object',
    required: ['number', 'name', 'type'],
    properties: {
        number: {
            type: 'number'
        },
        name: {
            type: 'string'
        },
        type: {
            type: 'string',
            enum: ['Street', 'Avenue', 'Boulevard']
        }
    }
}
```

RESTful Services
Validierung über Schema

```
var express = require('express');
var bodyParser = require('body-parser');

var { Validator, ValidationError } = require('express-json-validator-middleware');


// Initialize a Validator instance first
var validator = new Validator({allErrors: true}); // pass in options to the Ajv instance

// Define a shortcut function
var validate = validator.validate;
```

```
var app = express();

app.use(bodyParser.json());

// This route validates req.body against the StreetSchema
app.post('/street/', validate({body: StreetSchema}), function(req, res) {
    // At this point req.body has been validated and you can
    // begin to execute your application code
    res.send('valid');
});

// Error handler for valication errors
app.use(function(err, req, res, next) {
    if (err instanceof ValidationError) {
        // At this point you can execute your error handling code
        res.status(400).send('invalid');
        next();
    }
    else next(err); // pass error on if not a validation error
});
```

https://www.npmjs.com/package/express-json-validator-middleware

RESTful Services
Die Server-Seite: Path-Parameter

```
app.use(express.json());
app.put('/cars/:id', (req, res) => {
  let id = parseInt(req.params.id);
  if (cars[id]){ // gibt es das Auto?
    let updatedCar = req.body;
    cars[id] = updatedCar;
    res.status(200).send();
  } else {
    res.status(404, 'The Car is not found').send();
  }
});

//app.get('/users/:userId/books/:bookId‘,function (req, res) {
// Access userId via: req.params.userId
// Access bookId via: req.params.bookId
```

RESTful Services
Die Server-Seite; Negotiation

Wir prüfen hier, in welcher Repräsenationsfrom die Ressource gewünscht wird. Hierzu greifen wir über res.format() auf den Accept-Header zu!

```
app.get('/', (req, res) =>  res.format({
    json: () =>{ res.status(200).json(cars);},
    html: () => {res.status(406).send('Not supported yet\n');}
  }));
});
```

RESTful Services
Die Server-Seite

Bei der Verarbeitung von Query-Parametern bei GET brauchen wir keinen body-Parser

```
app.get('/', (req, res) => { let id = req.query.id;
        res.status(200).json(cars[id]) } );
```

RESTful Services

Wie sollten Query-Parameter genutzt werden?

Filtering: Use a unique query parameter for each field that implements filtering. For example, when requesting a list of tickets from the `/tickets` endpoint, you may want to limit these to only those in the open state. This could be accomplished with a request like `GET /tickets?state=open`. Here, `state` is a query parameter that implements a filter.

Sorting: Similar to filtering, a generic parameter `sort` can be used to describe sorting rules. Accommodate complex sorting requirements by letting the sort parameter take in list of comma separated fields, each with a possible unary negative to imply descending sort order. Let's look at some examples:

- `GET /tickets?sort=-priority` - Retrieves a list of tickets in descending order of priority
- `GET /tickets?sort=-priority,created_at` - Retrieves a list of tickets in descending order of priority. Within a specific priority, older tickets are ordered first

Quelle: https://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api

RESTful Services

Wie sollten Query-Parameter genutzt werden?

Searching: Sometimes basic filters aren't enough and you need the power of full text search. Perhaps you're already using ElasticSearch or another Lucene based search technology. When full text search is used as a mechanism of retrieving resource instances for a specific type of resource, it can be exposed on the API as a query parameter on the resource's endpoint. Let's say `q`. Search queries should be passed straight to the search engine and API output should be in the same format as a normal list result.

Combining these together, we can build queries like:

- `GET /tickets?sort=-updated_at` - Retrieve recently updated tickets
- `GET /tickets?state=closed&sort=-updated_at` - Retrieve recently closed tickets
- `GET /tickets?q=return&state=open&sort=-priority,created_at` - Retrieve the highest priority open tickets mentioning the word 'return'

Quelle: https://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api

RESTful Services

Wie sollten Query-Parameter genutzt werden?

Limiting which fields are returned by the API

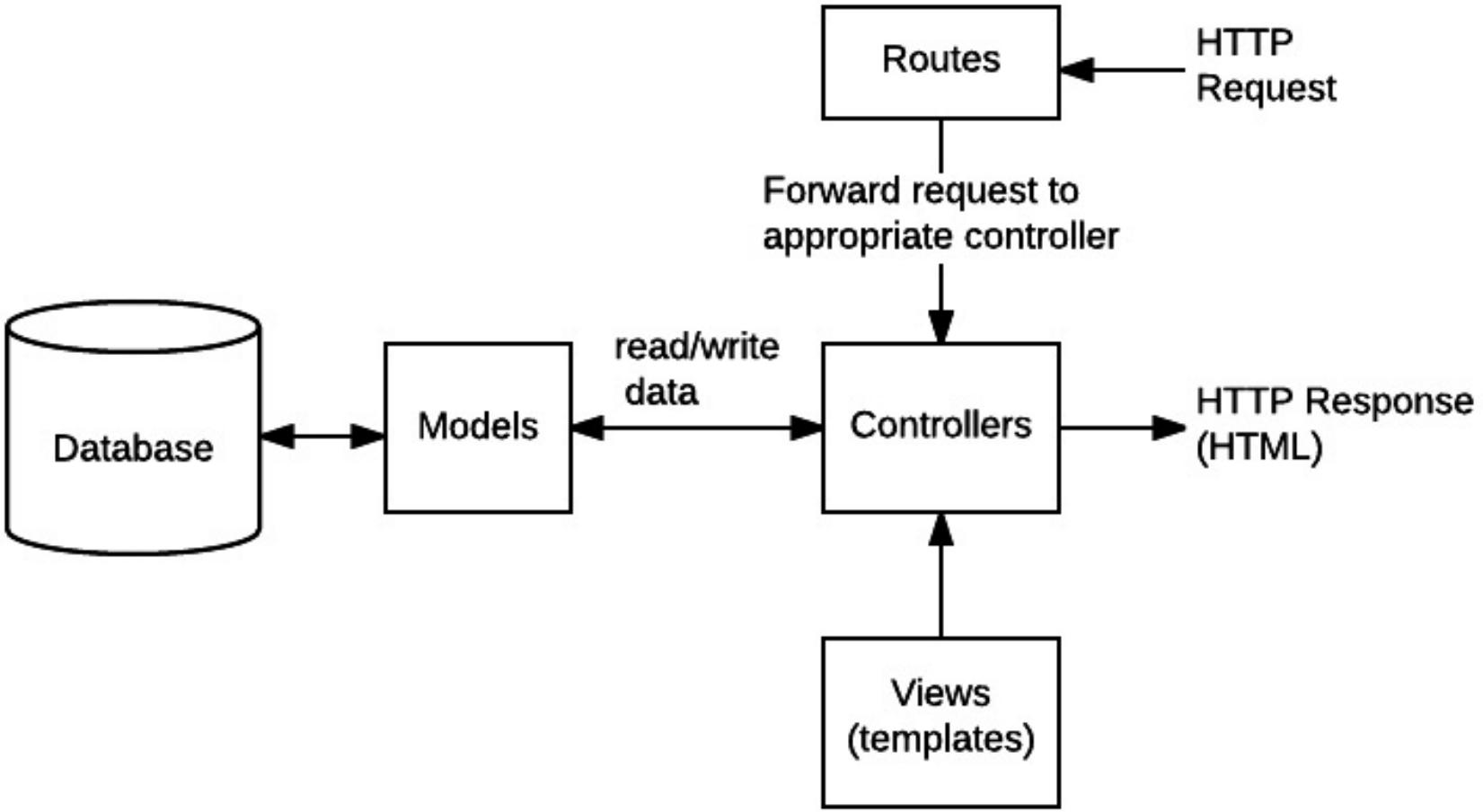
The API consumer doesn't always need the full representation of a resource. The ability select and chose returned fields goes a long way in letting the API consumer minimize network traffic and speed up their own usage of the API.

Use a `fields` query parameter that takes a comma separated list of fields to include. For example, the following request would retrieve just enough information to display a sorted listing of open tickets:

```
GET /tickets?fields=id,subject,customer_name,updated_at&state=open&sort=-updated_at
```

Quelle: https://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api

RESTful Services
Die Server-Seite im Großen



RESTful Services: Server

Erster Ansatz: Routes erstellen

```
// wiki.js
// Routes-Modul erstellen!
let express = require('express');
let router = express.Router();
// Home page route.
router.get('/', function (req, res) {
        res.send('Wiki home page'); })
// About page route.
router.get('/about', function (req, res) {
        res.send('About this wiki'); })
router.post('/about', function (req, res) {
        res.send('About this wiki'); })
module.exports = router;
```

Nutzung dann im Haupt-JS-File

```
var wiki = require('./wiki.js‘);
// ...
app.use('/wiki', wiki);
```

RESTful Services

Der bessere Ansatz: Die Server-Seite mit Routes

Elegante Zuordnung Routes und Controller

```
const express = require('express');
let router = express.Router()
const todoList = require('../controllers/todoListController');
// todoList Routes
router.route('/tasks')
        .get(todoList.list_all_tasks)
        .post(todoList.create_a_task);
router.route('/tasks/:taskId')
        .get(todoList.read_a_task)
        .put(todoList.update_a_task)
        .delete(todoList.delete_a_task);

module.exports router;
```

Nutzung dann im Haupt-JS-File über app.use

RESTful Services
Die Server-Seite mit Routes

Alternativ

```
module.exports = function(app) {
const todoList = require('../controllers/todoListController');
// todoList Routes
app.route('/tasks')
        .get(todoList.list_all_tasks)
        .post(todoList.create_a_task);
app.route('/tasks/:taskId')
        .get(todoList.read_a_task)
        .put(todoList.update_a_task)
        .delete(todoList.delete_a_task);
```

Aufruf mit

```
const express = require('express‘);
let app = express();
let routes = require('./routes/todoRoutes'); //importing routes
routes(app);
```

RESTful Services
Controller

```
let todo = require('../models/todo');

exports.list_all_tasks = (req,res) =>res.json(todo.allTodos());

exports.create_a_task = (req,res) =>
        { let newTodo = JSON.parse(request.body);
                Todo.add(newTodo);
                response.status(201).json();}


...
```

RESTful Services
Model Todo

```
Class Model { constructor() {
        this.todos = [{id:1, title:'kaufe Milch'},
        {id:2, title:'mache Übungsaufgaben'},
        {id:3, title:'telefoniere mit Eltern'}]
    }
addTodo(todoText) {
        const todo = {
            id: this.todos.length > 0 ?
                this.todos[this.todos.length - 1].id + 1 : 1,
            text: todoText
        }
        this.todos.push(todo)
}
deleteTodo(id) { this.todos = this.todos.filter(
                todo => todo.id !== id) }
allTodos() {return this.todos;}
}
```

RESTful Services
Der Client

Wir können hier beispielsweise die FETCH-API nutzen!

```
// Create a new ToDo
fetch('https://localhost:3000/users',
{ headers:
        { "Content-Type": "application/json; charset=utf-8" },
  method: 'POST‘,
  body: JSON.stringify({id:1, title:'buy the milk'})
}.then(...all Done, status code should be 201)
.catch(... Error);
```

RESTful Services
Frage beim Client

- Wer liefert beim Create die ID?
- Ids sollten nur beim Put vom client mitgeben werden, ein Put ist normalerweise ein update, so dass die Id schon bekannt ist
- Beim Post sollte keine Id mitgegeben werden, oder sie wird zumindest vom Server ignoriert
- Location Header bei der 201-Response sollte die URL mit id angeben!

Updates & creation should return a resource representation

A PUT, POST or PATCH call may make modifications to fields of the underlying resource that weren't part of the provided parameters (for example: created_at or updated_at timestamps). To prevent an API consumer from having to hit the API again for an updated representation, have the API return the updated (or created) representation as part of the response.

In case of a POST that resulted in a creation, use a HTTP 201 status code and include a Location header that points to the URL of the new resource.

Quelle: https://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api#useful-post-responses