Week 08 - JSON, Designing the API, REST API, OpenAPI

Lukáš Grolig et al.

Outline

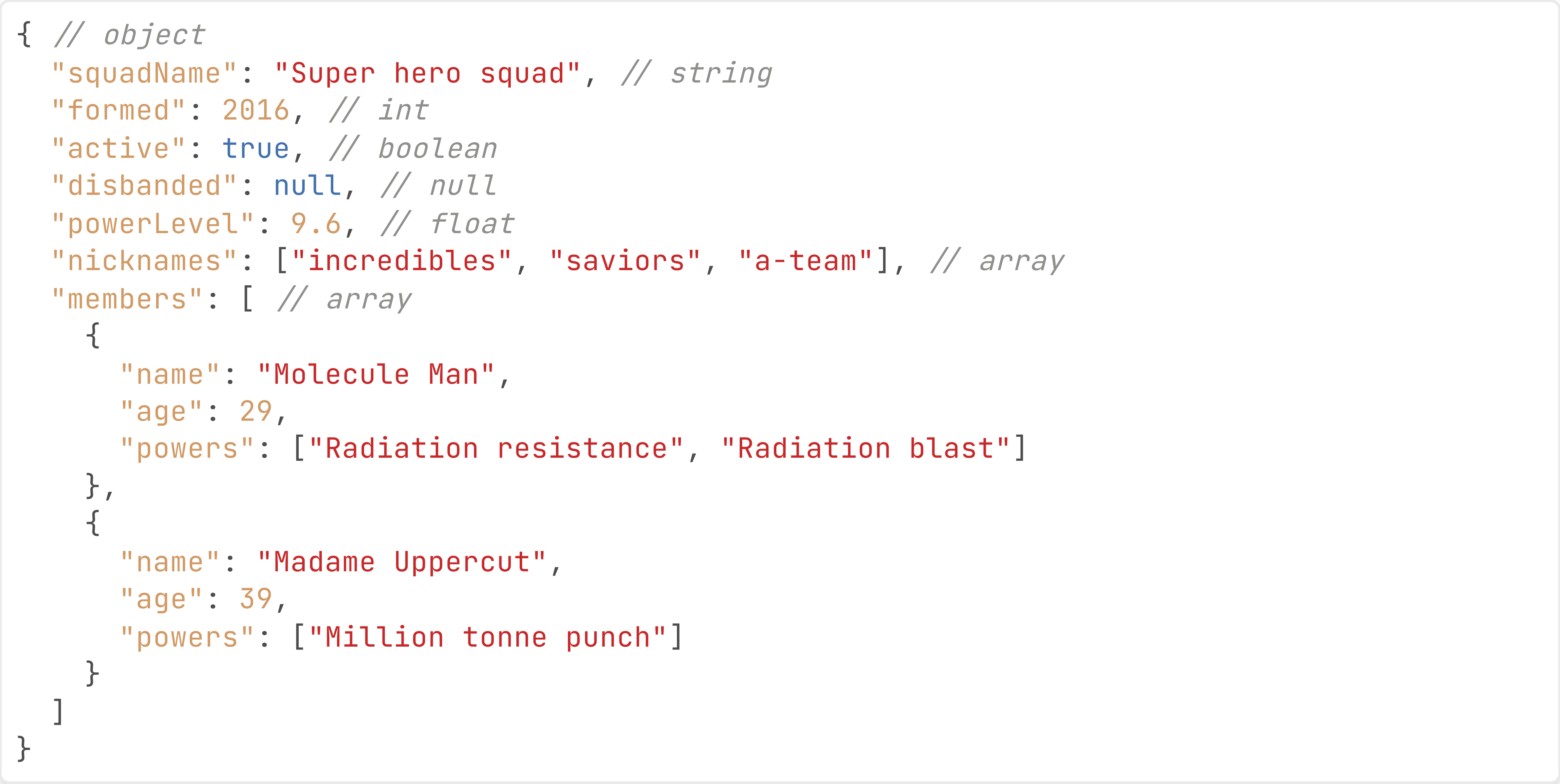
- JSON
- What is a Web API
- Restful API
- API Design (Resources, Actions)
- API Documentation (OpenAPI)
- Filtering, Sorting, Batching

JSON (JavaScript object notation)

- Not a markup language
- Used mainly for data exchange
- Can be compressed as the whitespaces are ignored
- Mimetype is `application/json`
- More compact than XML
- Based on subset of JavaScript (object notation)

JSON structure

- Two basic structures:
  1. key-value pairs `{"name": "John"}`
     - Values can be: booleans, ints, strings, objects, arrays and `null`
     - Usually implemented as: *object, dictionary, hash-table*
  2. Ordered lists of value `{"people": [ {"name": "John"}, {"name": "Jane"} ]`
     - Usually implemented as: *array, list, vector, sequence*
- Comments are **not** allowed



```
{ // object
  "squadName": "Super hero squad", // string
  "formed": 2016, // int
  "active": true, // boolean
  "disbanded": null, // null
  "powerLevel": 9.6, // float
  "nicknames": ["incredibles", "saviors", "a-team"], // array
  "members": [ // array
    {
      "name": "Molecule Man",
      "age": 29,
      "powers": ["Radiation resistance", "Radiation blast"]
    },
    {
      "name": "Madame Uppercut",
      "age": 39,
      "powers": ["Million tonne punch"]
    }
  ]
}
```

JSON Schema

It’s often necessary for applications to validate JSON objects, to ensure that required properties are present and that additional constraints (such as a price never being less than one dollar) are met. Validation is typically performed in the context of JSON Schema.

- JSON Schema is expressed by a schema, which is just a JSON object
- JSON Schema is maintained at http://json-schema.org
- It describes your existing data format
- It offers clear, human-readable, and machine-readable documentation
- It provides complete structural validation, which is useful for automated testing and validating client-submitted data

- It is recommended to use a generator
    - Results need to be further inspected

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "squadName": {
      "type": "string"
    },
    "powerLevel": {
      "type": "number"
    },
    "nicknames": {
      "type": "array",
      "items": [
        {
          "type": "string"
        }
      ]
...
```

What is Web API?

Service that provides an interface such as REST API to other services, which alters the service state or retrieves information from the service.

- API stands for the Application programming interface
- Two main types of API's: Private, Partner and Public
- Implemented in multiple ways:
    - RPC
    - SOAP
    - **REST**
    - gRPC
    - **GraphQL**

Restful API

HTTP methods, URN's and Responses in JSON form the REST API.
REST is used to perform **CRUD** operations on data.
We use URI to specify a resource: `/posts/1`
We use *HTTP verbs* to specify what should happen to data `VERB /posts/1`

- POST - **C**reate - non-idempotent - repeated calls will create new resources
- GET - **R**etrieve - safe, idempotent - repeated calls with same input produce the same result and don't modify DB
- PUT - **U**pdate - idempotent - repeated calls with same input produce same result
- DELETE - **D**elete - idempotent

Additional processing is specified by query parameters and handled by the server.
This includes sorting, filtering or limiting the number of results.

```
/posts/1?sort=asc&limit=10
```

After processing the request, the server returns a response code and (optionally) data - response body.

The following list contains examples of using response codes in REST API:

- 200 - OK - General request success code
- 201 - CREATED - Generally used as a response to POST request - indicates successful creation of resource
- 400 - BAD REQUEST - Validation of request failed, i.e. unparsable data format, invalid range of parameters
- 404 - NOT FOUND - Resource at URI does not exist
- 500 - SERVER ERROR - Any unexpected runtime error (Failed DB Connection, NullPointerException)

This should cover basic backend needs use-cases in this course.
For full list visit documentation.

200 (OK)

Response should include a response body. The information returned with the response is dependent on the method used in the request, for example:

- GET an entity corresponding to the requested resource is sent in the response;
- HEAD the entity-header fields corresponding to the requested resource are sent in the response without any message-body;
- POST an entity describing or containing the result of the action;
- TRACE an entity containing the request message as received by the end server.

201 (Created)

A REST API responds with the 201 status code whenever a resource is created inside a collection or as a result of a controller.

In response there is URI of the new resourse. Resourse MUST be created before sending response. If cannot be created immediately proper resonse is 202 (Accepted).

202 (Accepted)

A 202 response is typically used for actions that take a long while to process.

204 (No Content)

The 204 status code is usually sent out in response to a PUT, POST, or DELETE. It does not contain any body.

400 (Bad Request)

400 is the generic client-side error status, used when no other 4xx error code is appropriate. Errors can be like malformed request syntax, invalid request message parameters, or deceptive request routing etc.
The client SHOULD NOT repeat the request without modifications.

401 (Unauthorized)

A 401 error response indicates that the client tried to operate on a protected resource without providing the proper authorization.

The response must include a WWW-Authenticate header field containing a challenge applicable to the requested resource.

The client MAY repeat the request with a suitable Authorization header field. If the request already included Authorization credentials, then the 401 response indicates that authorization has been refused for those credentials.

403 (Forbidden)

A 403 error response indicates that the client’s request is formed correctly, but the REST API refuses to honor it, i.e. the user does not have the necessary permissions for the resource.

404 (Not Found)

The 404 error status code indicates that the REST API can’t map the client’s URI to a resource but may be available in the future. Subsequent requests by the client are permissible.

500 (Internal Server Error)

500 is the generic REST API error response. Most web frameworks automatically respond with this response status code whenever they execute some request handler code that raises an exception.

A 500 error is never the client’s fault, and therefore, it is reasonable for the client to retry the same request that triggered this response and hope to get a different response.

API response is the generic error message, given when an unexpected condition was encountered and no more specific message is suitable. It is not recommended to provide concrete information to the client. You can log an exception and show some kind of ID to the user. This ID can be used when communicating with app support.

Example

- `https://jsonplaceholder.typicode.com/posts/{id}/comments?sort=asc&limit=10`
- Let's dissect the example
  1. `https://jsonplaceholder.typicode.com` -- HOST
  2. `/users` -- A noun specifying the resource. **Always** use nouns, never verbs such as `getAllUsers` .
  3. `{id}` -- Identifier. A real request should be substituted by an identifier such as `1` . If ommitted the request should return all entities from the resource.
  4. `/comments` -- Nested entity. In this case, requesting comments made by user with `{id}` .
  5. `?sort=asc` -- Asking for results to be sorted in ascending fashion.
  6. `&limit=10` -- Additional parameters are separated by `&` . `limit=10` means to retrieve at most 10 results.

REST API Design

1. nouns, not verbs
2. statelessness

See: https://github.com/Microsoft/api-guidelines/blob/master/Guidelines.md

REST API Design

Here are some examples of DO's and DONT's.

```
# DO                    # DONT
GET /storage/1/files    GET /get_files/1
GET /phone-devices/1    GET /phone_devices or GET /phoneDevices
POST /subscription      POST /subscribe
GET /files/             GET /files
```

Communicating with REST API

- Notice a header of the request specifying how the server should interpret the body
- Some libraries (such as `got`) handle this automatically
- The code below is compatible with browser runtime, in node install `node-fetch` or `got`

```
fetch(
  'https://jsonplaceholder.typicode.com/posts',
  {
    method: 'POST',
    body: JSON.stringify({
      title: 'foo',
      body: 'bar',
      userId: 1,
    }),
    headers: {
      'Content-type': 'application/json; charset=UTF-8',
    },
  })
  .then((response) => response.json())
  .then((json) => console.log(json));
```

Communicating with REST API

- To avoid "callback hell"

```
try {
  const request = await fetch(
    'https://jsonplaceholder.typicode.com/posts',
    {
      method: 'POST',
      body: JSON.stringify({
        title: 'foo',
        body: 'bar',
        userId: 1,
      }),
      headers: {
        'Content-type': 'application/json; charset=UTF-8',
      },
    });
  const data = await request.json();
} catch (e) {}
```

REST Responses

There are many standards for the response structure of REST API.

- JSON:API
- JSend
- HAL
- rfc7807

GET /posts/2

```
{
  "status" : "success",
  "data" : { "post" : { "id" : 2, "title" : "Another blog post", "body" : "More content" }}
}
```

REST API using Express

```
app.get('/user/:id', function (req, res) {
  res.send('user ' + req.params.id)
})
```

REST API using Express

```
var express = require('express')

var app = express()

app.use(express.json()) // for parsing application/json
app.use(express.urlencoded({ extended: true })) // for parsing application/x-www-form-urlencod

app.post('/profile', function (req, res, next) {
  console.log(req.body)
  res.json(req.body)
})
```

Asynchronous REST API

- Does NOT provide data immediately.
- Returns 201 Created with Location header of a new resource
- Client performs pooling on the new resource
- It SHOULD containt a progress status
- Alternatively there can be temporary resourse returning 303 (See other) when finished

```
GET /api/v1/report/523
{
    "reportDate": "2021-01-01",
   "reportType": "COFFEE_SALES"
   "completed": true,
   "pdfUri": "https://s3.eu-central-1.amazonaws.com/.../report-123.pdf"
}
```

CORS

One more thing before writing your very own REST APIs

CORS was invented to facilitate serving REST API to different origins while preserving control and security.
The server can specify it in response headers, which origins are allowed to make requests.
If the browser detects that the server has not responded with the correct headers, it will cancel the request.

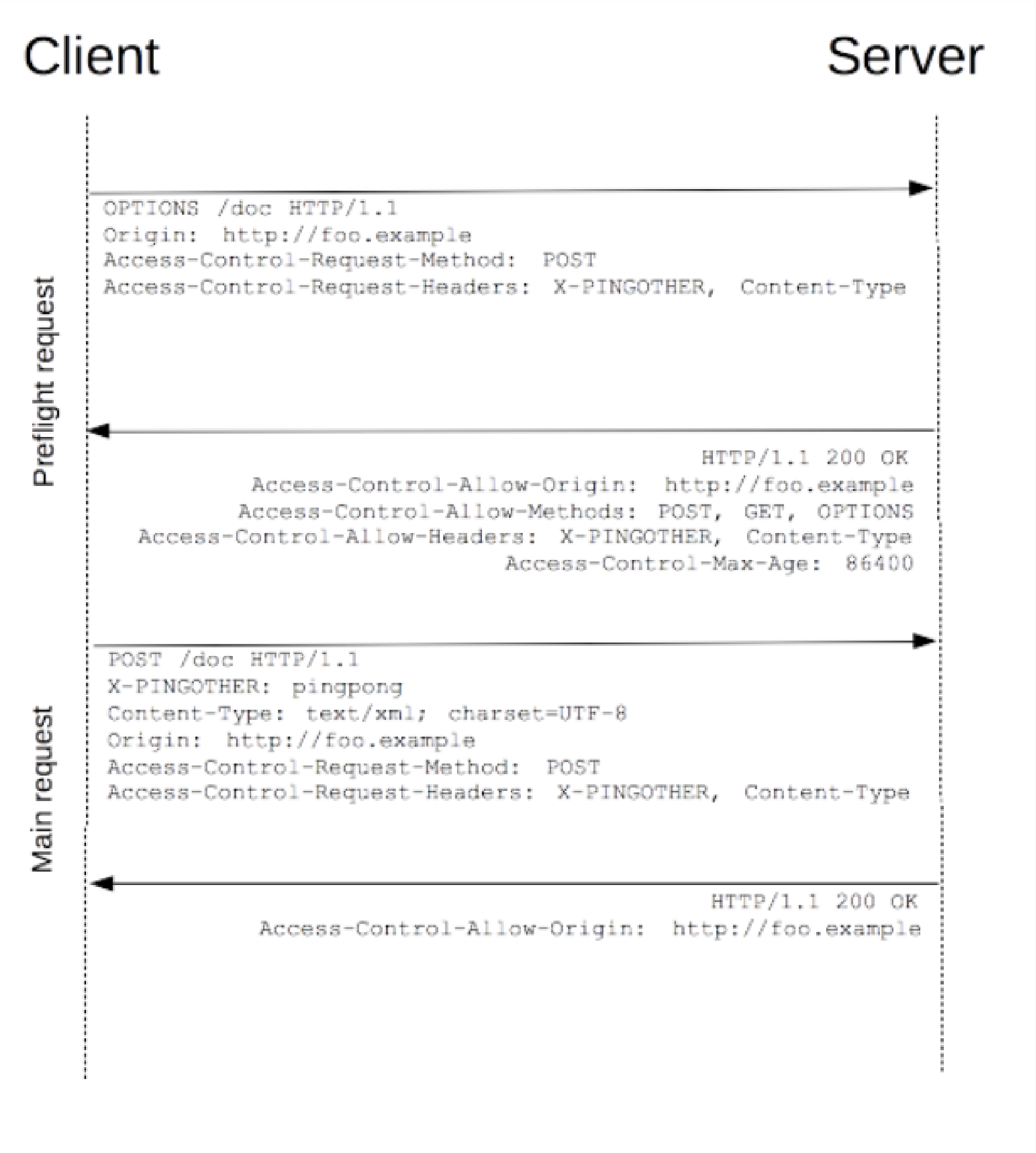
origin - The part of the URL from which the request originated e.g `app.example.com` (scheme, hostname, port)

*CORS is used as a substitution to **JSONP** (JSON wrapped with callback)*

CORS - how it works

1. The origin introduces itself with the `OPTIONS` method called **pre-flight** request.
    - Introduction includes what the client code intends to request (method and URI)
    - Browser does this automatically
    - Client is not even aware of this check (Until it stops working)
2. Server responds to `OPTIONS` request - informing the browser whether it will allow such request
3. Browser checks the response if method and origin are allowed
4. If the option checks pass, then the actual request is made
    - Checks are also repeated on the actual request
    - A well-behaved server will only respond yes if it allows the subsequent request
    - Many servers allow any `OPTIONS` request and then refuse the actual one

CORS - a schematic



```
OPTIONS /doc HTTP/1.1
Origin: http://foo.example
Access-Control-Request-Method: POST
Access-Control-Request-Headers: X-PINGOTHER, Content-Type
```

```
                                HTTP/1.1 200 OK
  Access-Control-Allow-Origin: http://foo.example
  Access-Control-Allow-Methods: POST, GET, OPTIONS
Access-Control-Allow-Headers: X-PINGOTHER, Content-Type
                     Access-Control-Max-Age: 86400
```

```
POST /doc HTTP/1.1
X-PINGOTHER: pingpong
Content-Type: text/xml; charset=UTF-8
Origin: http://foo.example
Access-Control-Request-Method: POST
Access-Control-Request-Headers: X-PINGOTHER, Content-Type
```

```
                               HTTP/1.1 200 OK
Access-Control-Allow-Origin: http://foo.example
```

CORS - Implementation

There are many nuances to CORS which are not listed here.
The following will configure the express to use the correct headers and everything should work smoothly.

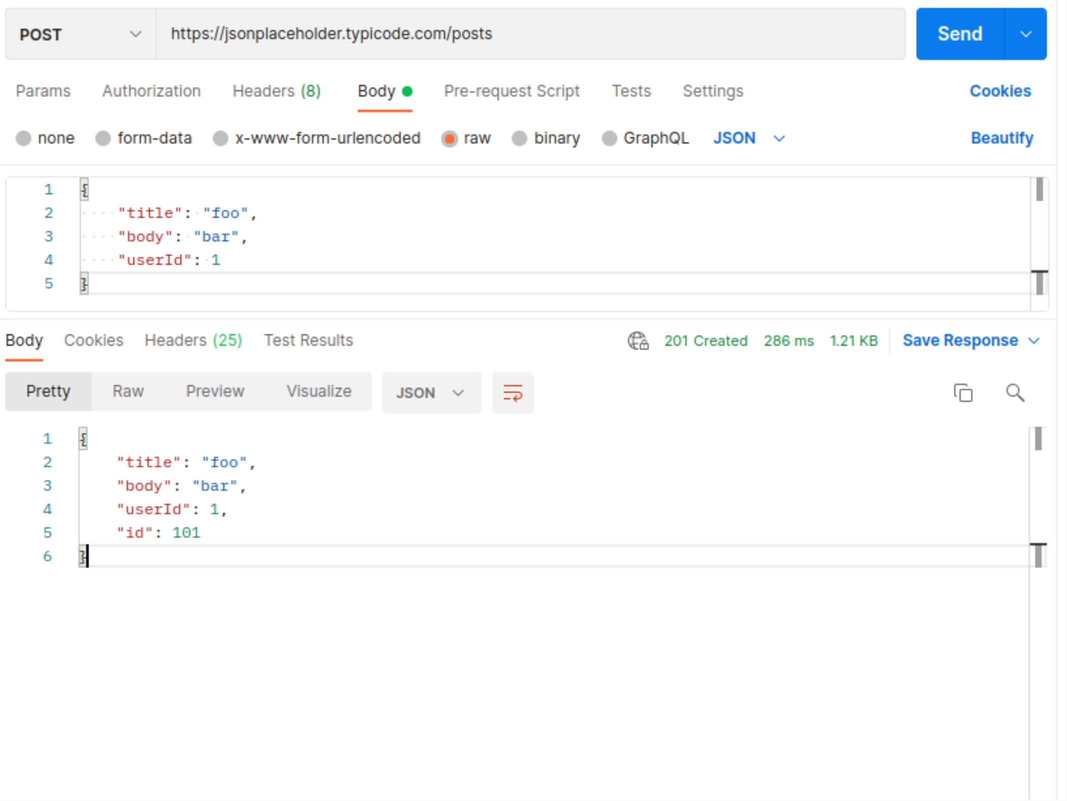
```
import express from 'express'
import cors from 'cors'
import app = express()

app.use(cors())
```

Body parser

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

const app = express();
const port = 3000;

// Where we will keep books
let books = [];

app.use(cors());

// Configuring body parser middleware
app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());

app.post('/book', (req, res) => {
    console.log(req.body.title);
});

app.listen(port, () => console.log(`Hello world app listening on port ${port}!`));
```

Testing

For API development, where quick testing and prototyping following clients are useful: Checkout Postman or Insomnia



POST https://jsonplaceholder.typicode.com/posts

```
{
    "title": "foo",
    "body": "bar",
    "userId": 1
}
```

201 Created 286 ms 1.21 KB

JSON

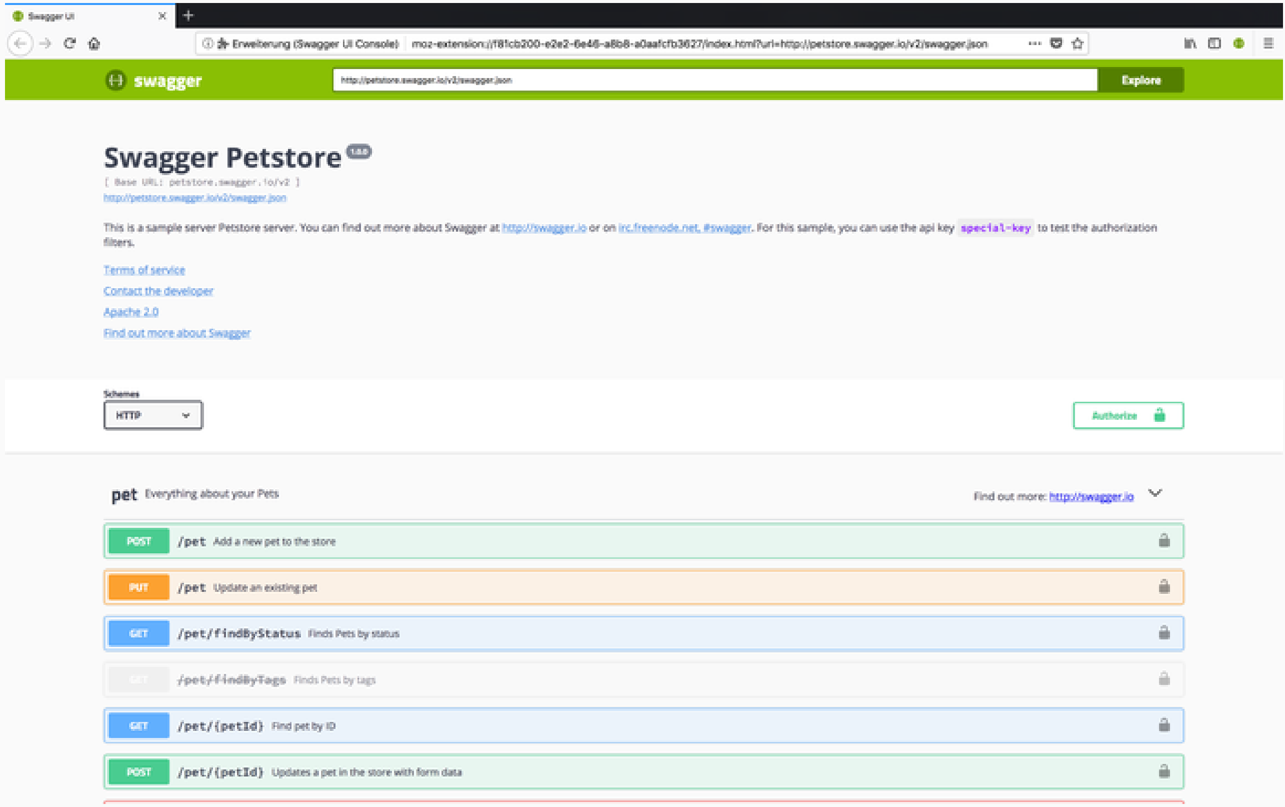
```
{
    "title": "foo",
    "body": "bar",
    "userId": 1,
    "id": 101
}
```

API Documentation

The OpenAPI Specification (OAS) defines a standard, language-agnostic interface to RESTful APIs which allows both humans and computers to discover and understand the capabilities of the service without access to source code, documentation, or through network traffic inspection.

Can be written as YAML or JSON.

Can be used to generate API Documentation in form of application Swagger.


Swagger UI
Erweiterung (Swagger UI Console)
moz-extension://f81cb200-e2e2-6e46-a8b8-a0aafcfb3627/index.html?url=http://petstore.swagger.io/v2/swagger.json
swagger
http://petstore.swagger.io/v2/swagger.json
Explore

Swagger Petstore

[Base URL: petstore.swagger.io/v2]

http://petstore.swagger.io/v2/swagger.json

This is a sample server Petstore server. You can find out more about Swagger at http://swagger.io or on irc.freenode.net, #swagger. For this sample, you can use the api key `special-key` to test the authorization filters.

Terms of service

Contact the developer

Apache 2.0

Find out more about Swagger

pet Everything about your Pets — Find out more: http://swagger.io

POST `/pet` Add a new pet to the store

PUT `/pet` Update an existing pet

GET `/pet/findByStatus` Finds Pets by status

GET ~~`/pet/findByTags`~~ Finds Pets by tags

GET `/pet/{petId}` Find pet by ID

POST `/pet/{petId}` Updates a pet in the store with form data

Info Object

The object provides metadata about the API. The metadata MAY be used by the clients if needed, and MAY be presented in editing or documentation generation tools for convenience.

```
title: Sample Pet Store App
description: This is a sample server for a pet store.
termsOfService: http://example.com/terms/
contact:
  name: API Support
  url: http://www.example.com/support
  email: support@example.com
license:
  name: Apache 2.0
  url: https://www.apache.org/licenses/LICENSE-2.0.html
version: 1.0.1
```

Component Object - Schema

Holds a set of reusable objects for different aspects of the OAS. All objects defined within the components object will have no effect on the API unless they are explicitly referenced from properties outside the components object.

```yaml
components:
  schemas:
    GeneralError:
      type: object
      properties:
        code:
          type: integer
          format: int32
        message:
          type: string
    Category:
      type: object
      properties:
        id:
          type: integer
          format: int64
        name:
          type: string
```

Server Object

```yaml
servers:
- url: https://development.gigantic-server.com/v1
  description: Development server
- url: https://staging.gigantic-server.com/v1
  description: Staging server
- url: https://api.gigantic-server.com/v1
  description: Production server
```

Path object

```
/pets:
  get:
    description: Returns all pets from the system that the user has access to
    responses:
      '200':
        description: A list of pets.
        content:
          application/json:
            schema:
              type: array
              items:
                $ref: '#/components/schemas/pet
```

Path item object

Describes the operations available on a single path.

```yaml
get:
  description: Returns pets based on ID
  summary: Find pets by ID
  operationId: getPetsById
  responses:
    '200':
      description: pet response
      content:
        '*/*' :
          schema:
            type: array
            items:
              $ref: '#/components/schemas/Pet'
    default:
      description: error payload
      content:
        'text/html':
          schema:
            $ref: '#/components/schemas/ErrorModel'
parameters:
```

That's it.

Resources

- https://en.wikipedia.org/wiki/URL
- https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods
- https://developer.mozilla.org/en-US/docs/Web/HTTP/Status
- https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS
- https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers
- https://www.w3.org/2001/sw/wiki/REST
- very recommended reading if you are serious about REST