Online Multimedia

Winter Semester 2019/20

Tutorial 09 – REST APIs
REST APIs
REpresentational State Transfer – REST

• REST is a general architecture for distributed systems, prominently (mis)used on the Web

• Defined by a number of architectural constraints:
  • Client-Server – Separation of concerns; given on the Web
  • Stateless – Each request must contain all the context necessary to respond
  • Cache – Responses must be cacheable and indicate whether that is a good idea
  • Uniform Interface –
    • Layered System – Servers can be clients for other services → Machine-to-machine communication
  • Code-On-Demand – Clients can request code to extend their functionality. Rarely used to prevent XSS

• A REST API consists of resources and actions on those resources

REST – Architecture
"The key abstraction of information in REST is a resource."

- Roy T. Fielding

<table>
<thead>
<tr>
<th>Name</th>
<th>Examples</th>
<th>Where to find it in HTTP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resource</td>
<td>Application-specific, e.g. <em>user, artist, album</em></td>
<td><em>Abstract concept</em></td>
</tr>
<tr>
<td>Resource Identifier</td>
<td>URI/URL</td>
<td><em>URL</em></td>
</tr>
<tr>
<td>Resource Metadata</td>
<td>Resource-Media-Type</td>
<td><em>Header</em></td>
</tr>
<tr>
<td>Representation</td>
<td>JSON, XML, HTML, Images</td>
<td><em>Body</em></td>
</tr>
<tr>
<td>Representation Metadata</td>
<td>Representation-Media-type, last-modified, etc.</td>
<td><em>Header</em></td>
</tr>
<tr>
<td>Resource Methods</td>
<td>GET, POST, PUT, ...</td>
<td><em>HTTP Method</em></td>
</tr>
</tbody>
</table>
REST – Resource Identifiers

• Resources are identified by URI/URLs

• Since resources are "things", resource identifiers primarily use nouns
  Good: /users
  Bad: /select-users

• URIs usually follow a hierarchical scheme:
  /projects/536/members/4

• By convention (and Googles recommendation) use hyphens (-) in URIs:
  Good: /date-of-birth
  Bad: /date_of_birth or /dateofbirth

• Use Query-Parameters to filter/sort a collection of resources
  /users?born-before=1990&sort-by=name
Resource Representation & Metadata

• Resource representation must be hypermedia/hypertext
  • "Hypertext (means) the simultaneous presentation of information and controls" – Roy T. Fielding
  • E.g. in HTML the <a>-tag both presents data and can be used to GET another resource

• A *true* RESTful API should be self-descriptive. Hypertext-references therefore should be described with their media-type or relationship

• This hypertext & metadata should be the primary definition of what is possible with a resource (*HATEOAS* – Hypertext As The Engine Of Application State)

• This can mean new media-types in conjunction with common ones
  application/hal+json (*HAL = Hypermedia API Language*)

Resource representation – JSON

GET /projects/10

{
    "departmentId": 10,
    "departmentName": "Administration",
    "locationId": 1700,
    "managerId": 200,
    "links": [
        {
            "href": "/projects/10/employees",
            "rel": "employees",
            "type": "application/myapp.list+json"
        }
    ]
}
REST – Resource Methods

• Resource Methods describe what should be done with resources

• The table on the right describes the common convention on the web

<table>
<thead>
<tr>
<th>HTTP Method</th>
<th>CRUD</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST</td>
<td>Create</td>
</tr>
<tr>
<td>GET</td>
<td>Read</td>
</tr>
<tr>
<td>PUT</td>
<td>(Full) Update</td>
</tr>
<tr>
<td>DELETE</td>
<td>Delete</td>
</tr>
<tr>
<td>PATCH</td>
<td>(Partial) Update</td>
</tr>
</tbody>
</table>
Idempotent and Safe Methods

- Certain HTTP methods are expected to be idempotent and/or safe

  - Safe methods do not change data
  - Idempotence means you can perform an operation multiple times and the result will always be the same.
  - i.e. You can call the same URL multiple times in a row and the result is always the same – as long no other URLs are called in between

<table>
<thead>
<tr>
<th>Method</th>
<th>Idempotent</th>
<th>Safe</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>POST</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>PUT</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>DELETE</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>PATCH</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>HEAD</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>OPTIONS</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Status Codes

• To ensure self-descriptiveness, servers should respond with the appropriate HTTP status code

1xx – Informational
2xx – Success
3xx – Redirects
4xx – Client Error
5xx – Server Error
Example: Spotify API

Artists

Endpoints for retrieving information about one or more artists from the Spotify catalog.

Base URL: https://api.spotify.com/v1

<table>
<thead>
<tr>
<th>METHOD</th>
<th>ENDPOINT</th>
<th>USAGE</th>
<th>RETURNS</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>/v1/artists/{id}</td>
<td>Get an Artist</td>
<td>artist</td>
</tr>
<tr>
<td>GET</td>
<td>/v1/artists/{id}/albums</td>
<td>Get an Artist's Albums</td>
<td>albums</td>
</tr>
<tr>
<td>GET</td>
<td>/v1/artists/{id}/top-tracks</td>
<td>Get an Artist's Top Tracks</td>
<td>tracks</td>
</tr>
<tr>
<td>GET</td>
<td>/v1/artists/{id}/related-artists</td>
<td>Get an Artist's Related Artists</td>
<td>artists</td>
</tr>
<tr>
<td>GET</td>
<td>/v1/artists</td>
<td>Get Several Artists</td>
<td>artists</td>
</tr>
</tbody>
</table>

https://api.spotify.com/v1/artists/1vCWHaC5f2uS3yhpwWbIA6/albums?market=ES&include_groups=album&limit=2
Code Along: The iPod API
Code Along: The iPod API

Imagine a „super-thin-client“ iPod

• Nearly no local memory
• Requests data for each screen from a server
• Transfer only the data necessary at a point of time
The iPod API: Requirements

The API should be able to...

• Handle multiple playlists
• Search the iTunes API for tracks
• Add songs to a playlist
• Remove songs from a playlist
Brainstorming Session

What would be a suitable API design?

<table>
<thead>
<tr>
<th>METHOD</th>
<th>ENDPOINT</th>
<th>USAGE</th>
<th>RETURNS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# The iPod API: Our Suggestion

<table>
<thead>
<tr>
<th>METHOD</th>
<th>ENDPOINT</th>
<th>USAGE</th>
<th>RETURNS</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>/playlists</td>
<td>Get all playlists</td>
<td>List of ids</td>
</tr>
<tr>
<td>GET</td>
<td>/playlists/{id}</td>
<td>Get a playlist</td>
<td>Playlist metadata (name, id)</td>
</tr>
<tr>
<td>GET</td>
<td>/playlists/{id}/songs</td>
<td>Get all playlist‘s song</td>
<td>List of track ids</td>
</tr>
<tr>
<td>GET</td>
<td>/playlists/{id}/songs/{trackId}</td>
<td>Get a song</td>
<td>Redirect to /songs/{trackId}</td>
</tr>
<tr>
<td>POST</td>
<td>/playlists/{id}/songs/{trackId}</td>
<td>Add a song to the playlist</td>
<td>-</td>
</tr>
<tr>
<td>PUT</td>
<td>/playlists/{id}/songs/{trackId}</td>
<td>Add a song to the playlist</td>
<td>-</td>
</tr>
<tr>
<td>DELETE</td>
<td>/playlists/{id}/songs/{trackId}</td>
<td>Remove a song from the playlist</td>
<td>-</td>
</tr>
<tr>
<td>GET</td>
<td>/songs/{trackId}</td>
<td>Get a song</td>
<td>Song metadata (title, artist, id)</td>
</tr>
<tr>
<td>GET</td>
<td>/search?term={term}</td>
<td>Search for a song</td>
<td>Metadata for all found songs</td>
</tr>
</tbody>
</table>
Breakout #1

Use the skeleton *ipod-server* from GitHub

- „virtual iPod“ as webpage
- NodeJS Express server, that manages playlists data

Open [http://localhost:3000](http://localhost:3000) to view the iPod in your browser

Implement the *playlists* and *songs* endpoints of the API! No changes have to be done on the client side. Do not implement *search* yet.

- The data is managed by the *DataStorage* class. One endpoint is already implemented for you, to see how *DataStorage* can be accessed.
iTunes Search API

• Search in the iTunes database
• Songs, books, artists, podcasts, videos, ...
• Read only – no authentication required

Lets use it to feed our search endpoint

Search Endpoint

iPod

- Playlists Request
  - Playlists Response
- Search Request
  - Search Response

iTunes

- iTunes Search Request
  - iTunes Search Response

iPod API

RESTful API

WAIT... WHAT?

APIS THAT CALL APIS? WHO IS CLIENT AND WHO IS SERVER NOW?
Breakout #2

Implement the search endpoint

http://localhost:3000/search?term=Californication

Under the hood, the search should be performed by the iTunes API

Hint: To avoid complications later on, cache each found song in the DataStorage’s songs array immediately.
Round-up Quiz

1. What is special about an idempotent endpoint?
2. Which groups of HTTP status codes do you know?
3. What is the key abstraction of REST?
4. What are query parameters appropriate for?
Happy Holidays!
See you next year!