SEBA Master: Web Application Engineering

Building a REST-enabled back-end service

Florian Matthes

Chair of Software Engineering for Business Information Systems (sebis)
Faculty of Informatics
Technische Universität München
www.matthes.in.tum.de
Outline of the Lecture

6. Building a REST-enabled back-end service

- Target architecture and development environment
- Event-driven architecture and asynchronous I/O operations with node.js
- Creating REST interfaces using express.js
- Using document-oriented database storage: MongoDB example
- Enabling user authentication on the web service using JSON web token (JWT)
## Architecture of web applications

<table>
<thead>
<tr>
<th>Client (Web browser)</th>
<th>Client-side libraries and frameworks</th>
<th>HTTP protocol</th>
<th>Communication between client and server apps</th>
<th>Server (OS, language runtime environment)</th>
<th>Web application server-side framework</th>
<th>Persistency layer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Information presentation
- Information retrieval
- Information navigation
- [Information validation & editing/collaborative editing]
- Established common protocol
- Business logic
- Persistency management
### Technology Options

#### Client-side libraries and frameworks
- React
- Angular
- JQuery
- Vue.js
- Bootstrap
- ...

#### Communication between client and server apps
- REST
- Web Sockets
- ...

#### Web Application server-side framework
- Express.js
- Spring
- Django
- Play
- Ruby on Rails
- ...

#### Persistency layer
- MongoDB
- MySQL
- postgreSQL
- ...

---

**Continuously evolving technology options for web applications**

---

© sebis
### Example of a web technology stack

**MERN stack (MongoDB, express.js, React, node.js)**

<table>
<thead>
<tr>
<th>Client-side libraries and frameworks</th>
<th>React</th>
<th>Angular</th>
<th>JQuery</th>
<th>Vue.js</th>
<th>Bootstrap</th>
<th>…</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communication between client and server apps</td>
<td>REST</td>
<td>Web Sockets</td>
<td>…</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Web application server-side framework</td>
<td>Express.js</td>
<td>Spring</td>
<td>Django</td>
<td>Play</td>
<td>Ruby on Rails</td>
<td>…</td>
</tr>
<tr>
<td>Persistency layer</td>
<td>MongoDB</td>
<td>MySQL</td>
<td>postgreSQL</td>
<td>…</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Using MongoDB, Express.js (node.js) for the back-end project
Why using web development tools?

When implementing web applications, developers often face **similar issues and challenges:**

- Downloading, installing, and managing packages, frameworks, and tools for the web application server
- Downloading and managing libraries and frameworks for web clients, and ensuring the compatibility
- Preparing a basic project structure ("scaffolding")
- Transforming a component-based project structure to a deployable and executable structure
- Transcompiling a programming language (e.g., TypeScript) to an executable counterpart (e.g., JavaScript)
- Preparing and running different kinds of tests, e.g., unit-tests
- Minifying or uglifying source code
- …

In previous years, a **huge ecosystem of tools** emerged which address these issues and support developers in implementing web applications.
Node.js' package ecosystem, npm, is the largest ecosystem of open source libraries in the world.

- npm is the official package manager for NodeJS
- npm is an online registry for open-source NodeJS modules, resources, etc. [http://npmjs.org](http://npmjs.org)

### package.json

```
{
  "name": "VolunteerApp",
  "version": "1.2.0",
  "private": true,
  "scripts": {
    "start": "node ./bin/www"
  },
  "dependencies": {
    "body-parser": "~1.13.2",
    "cookie-parser": "~1.3.5",
    "debug": "~2.2.0",
    "express": ">4.13.1",
    "serve-favicon": "~2.3.0"
  }
}
```

### How to install a package:

```
npm install cookie-parser --save
```

Adds the dependency ‘cookie-parser’ to the package.json

**Source:** [https://docs.npmjs.com/files/package.json](https://docs.npmjs.com/files/package.json), [https://quickleft.com/blog/creating-and-publishing-a-node-js-module/](https://quickleft.com/blog/creating-and-publishing-a-node-js-module/)
NodeJS – node.js packages used in MovieApp

<table>
<thead>
<tr>
<th>Dependency Name</th>
<th>Short Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mongoose</td>
<td>Connect to MongoDB</td>
</tr>
<tr>
<td>Express</td>
<td>A rest enabling framework on top of node.js web application framework</td>
</tr>
<tr>
<td>bcrypt-nodejs</td>
<td>Useful to encrypt and decrypt salted passwords</td>
</tr>
<tr>
<td>jsonwebtoken</td>
<td>Web tokens identify users in a scalable manner and “replace” the old session id</td>
</tr>
<tr>
<td>helmet</td>
<td>Helpful to secure Express apps by setting various HTTP headers</td>
</tr>
<tr>
<td>body-parser</td>
<td>A node.js middleware to parse incoming request bodies before the handlers</td>
</tr>
</tbody>
</table>

More NodeJS Packages here: [https://www.npmjs.com](https://www.npmjs.com)
Outline of the lecture

6. Building a REST-enabled back-end service
- Target architecture and development environment
- Event-driven architecture and asynchronous I/O operations with node.js
- Creating REST interfaces using express.js
- Using document-oriented database storage: MongoDB example
- Enabling user authentication on the web service using JSON web token (JWT)
Event-driven architecture and asynchronous I/O operations with node.js

Client-side libraries and frameworks
- React based client

Communication
- Static Files (JavaScript, JSX, CSS files, images, etc.)

Web application server-side framework
- NodeJS (dev)
  - npm
  - babel
  - webpack

Persistency layer
- MongoDB

HTTP request
- REST specification

HTTP response
Black box view of a REST-enabled web service

A web application service receives a request via HTTP with:
- An URL
- Request header
- Request body

and sends back an HTTP response which contains the answer encoded in JSON

Client 1
(Web Application/iOS/Android Application)

Client 2
(Web Application/iOS/Android Application)

Client N
(Web Application/iOS/Android Application)

Web server
How to handle concurrent requests from multiple clients?

A) Each request is handled by a separate operating system (OS) process *(outdated)*

**Example:**
- cgi scripts written in Perl

**Pros:**
- Isolation
- Security
- Multi-language support
- Availability on every OS

**Cons:**
- High operating cost,
- High consumption of memory
How to handle concurrent requests from multiple clients?

B) Each request is handled by a separate thread in a single shared OS process

Examples:
Multithreading in JAVA (EJB), C# (.Net)

Pros:
- Use of multi-core CPU under your control
- Shared main memory data structure
- One language
- No limitations on blocking IO (easy sequential programming)
- Threads can be directly mapped to DB transactions

Cons:
- Difficulty of concurrent programming
- Requires thread-safe data structures
- Need for effective thread-pooling
How to handle concurrent requests from multiple clients?

C) Each request is handled by a single thread in a single OS process

**Examples:**
Node.js, Netty (Java, Scala)

**Pros:**
- Lower memory consumption
- Developer does not deal with concurrency issues on the level of web application server
- Better resistance to heavy loads by design

**Cons:**
- Long-running computations inside single thread block the server

**NodeJS example:**

![NodeJS - Asynchronous I/O](image)

*Event queue*

- Request 1
- Request 2
- Callback 1
- Callback 2

*Asynchronous call*

- Single event loop
  - Place callback method with result arguments in the event queue

*Workers (event handlers)*

- Read file
- Query database
- Query 3rd party API
NodeJS is an open-source, cross-platform JavaScript run-time environment for executing JavaScript code server-side.

Why developers chose NodeJS?
- Build on top V8 JavaScript engine (built and supported by Google, used inside Chrome Browser)
- Enable JavaScript on the server → no need to switch between different languages on a client and server
- Cross-platform
- Recent popularity of JSON as a standard for exchanging content
- Huge ecosystem of libraries with active developer communities

To effectively deal with concurrent requests in NodeJS developers are forced to use:
- Non-blocking asynchronous NodeJS API methods and callbacks
### Blocking synchronous method vs. Non-blocking asynchronous method

**Why callbacks?**

#### Blocking example

**Wrong:** use of synchronous method in the single event loop

```js
main.js
const fs = require('fs');
const data = fs.readFileSync('/file.md');
console.log(data);
console.log('Example with synchronous method');
```

**Console output:**

```
Example with synchronous method
```

#### Non-blocking example

**Correct:** Using asynchronous method with callback function

```js
main.js
const fs = require('fs');
fs.readFile('/file.md', (err, data) => {
  if (err) throw err;
  console.log(data);
});
console.log('Example with asynchronous method');
```

**Console output:**

```
Example with asynchronous method
```

**Non-blocking example**

**Correct:** Using asynchronous method with callback function

**Asynchronous method:**
- Performs I/O operations (file reads, db queries, etc.)
- Executed outside NodeJS event loop (without blocking it)
- Could use multiple threads (depends on OS and realization)
- Gets a callback function as the last parameter

**Callback Function:**
- Executed after an error occurs or the operations is successfully completed
- Executed inside event loop

---

6. Building a REST-enabled back-end service

- Target architecture and development environment
- Event-driven architecture and asynchronous I/O operations with node.js
- Creating REST interfaces using express.js
- Using document-oriented database storage: MongoDB example
- Enabling user authentication on the web service using JSON web token (JWT)
REST specification

Frontend Project

React based client

Communication

Static Files
(JavaScript, JSX, CSS files, images, etc.)

HTTP request

Backend Project

REST specification

NodeJS (dev)

npm  babel  webpack

NodeJS

JWT  Mongoose  Express

npm  nodemon

Persistency layer

MongoDB
Representational State Transfer

- Representational State Transfer (REST) is a software architectural style for distributed hypermedia systems like the world wide web.
- The term has been coined by Roy Fielding in his doctoral dissertation.
- REST provides a set of architectural constraints that, when applied as a whole, emphasizes:
  - scalability of component interactions,
  - generality of interfaces,
  - independent deployment of components, and
  - intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems.
- The REST architectural style has been used to guide the design and development of the architecture for the modern Web.
- The abstract discussion about architectural styles enables judgments over whether particular practices are consistent with the architecture of the Web.
The null style

- Is simply an empty set of constraints.
- Describes a system in which there are no distinguished boundaries between components.
- Is the starting point for the description of REST

*Example:* Bad mainframe application
Client-server

- Separates the user interface concerns from the data storage concerns.
- Improves the portability of the user interface across multiple platforms.
- Improves scalability by simplifying the server components.
- Allows the components to evolve independently.

Cacheable

- Require that the data within a response to a request be implicitly or explicitly labeled as cacheable or non-cacheable.
- Improves efficiency, scalability, and user-perceived performance by reducing the average latency of a series of interactions.
- Trade-off: a cache can decrease reliability if stale data within the cache differs significantly from the data that would have been obtained had the request been sent directly to the server.

Uniform interfaces

- This is the central feature that distinguishes the REST architectural style from other network-based styles.
- Emphasis on a **uniform interface** between components
  - Uniform identification scheme
  - Uniform representation of information exchanged
- Implementations are **decoupled** from the services they provide.

- **Trade-off**: information is transferred in a standardized form rather than one which is specific to an application’s needs.
Uniform representation of information exchanged

- The type of a representation is specified in the HTTP header `Content-Type` of the response.
- Content types are identified using MIME (Multipurpose Internet Mail Extension) types.
- A MIME type contains a type and a subtype.
  - Types are generic and predefined, e.g., text, image, audio, video, and application.
  - Subtypes provide more information about the content, e.g., text/plain, text/html, image/jpeg, application/json.
- The Web uses standardized (often human-readable) exchange content formats, in particular HTML and XML, JSON
- Contrast with “optimized” record layout & binary representations.
Uniform identification scheme

- URI: Uniform Resource Identifier (general term)
  - There are two mechanisms: naming and location
- URN: Uniform Resource Name
  - Identification of objects by name for the purpose of persistent labeling
  - E.g., ISBN
- URL: Uniform Resource Locator
  - Identification via the primary access mechanism

An example of an URI:

<table>
<thead>
<tr>
<th>scheme</th>
<th>authority</th>
<th>path</th>
<th>query</th>
<th>fragment</th>
</tr>
</thead>
<tbody>
<tr>
<td>http</td>
<td>example.org</td>
<td>/mysite/page</td>
<td>? name=cat</td>
<td># whiskers</td>
</tr>
</tbody>
</table>

A URI points to a hierarchical space reading from left to right; each block that follows is a branch from the previous block.
The URI dissected

- **Scheme** – Defines how the URI should be interpreted
- **Authority** – Has the structure `userinfo@host:port`
- **Path** – Looks like the path on a file system and is often used in a hierarchical fashion to address files on a system
  - Example: `http://del.iciou.us/danja/owl` refers to all items tagged by danja with owl
- **Query** – The spec describes the query part as being a non-hierarchical part of the URI
- **Fragment** – Is used to identify a secondary resource

- URIs are not being used uniformly on the Web, but: when a URI has been used to identify a resource, it should continue to be used to identify the same resource.
- See: “**Cool URIs don't change**” from Tim Berners Lee [http://www.w3.org/Provider/Style/URI]
The main problem domains identified in REST are the **nouns**, the **verbs**, and the **content-type** spaces.

- The *things that exist*, the *things you can do to them*, and the *information you can transfer* as part of any particular operation

- REST requires a **standardized** set of state transfer operations.
Mapping HTTP with REST methods

- Minimum methods: GET, PUT, POST, DELETE
  - **GET** is the HTTP equivalent of COPY
    - Transfers a representation from resource to client.
  - **PUT** is the HTTP equivalent of PASTE OVER
    - Transfers state from a client to a resource.
    - GET and PUT are fine for transferring state of existing resources.
  - **POST** is the PASTE AFTER verb
    - Don't overwrite what you currently have: Add to it
    - Create a resource.
    - Add to a resource.
  - **DELETE** is the HTTP equivalent of CUT
    - Requests the resource state being destroyed.
Web APIs do not provide presentation elements but services to be consumed by application code

- A Web API consists of a defined set of HTTP Request messages.
- For each request the Web API specifies the structure of response messages.
- A response message is typically expressed in JSON or XML.

**JSON** (JavaScript Object Notation) is lightweight text-based open standard designed for human-readable data interchange.
- It is derived from the object literal syntax of JavaScript programming language.

Two equivalent responses in different formats:

```
<lecture>
  <title>SEBA Master</title>
  <chapters>
    <chapter>
      <title>Web Site Genres</title>
      <number>1</number>
    </chapter>
    <chapter>
      <title>Web Site Design Process</title>
      <number>2</number>
    </chapter>
    <chapter>
      <title>Web Design Patterns</title>
      <number>3</number>
    </chapter>
  </chapters>
</lecture>
```

```
{
  "title": "SEBA Master",
  "chapters": [
    {
      "number": 1,
      "title": "Web Site Genres"
    },
    {
      "number": 2,
      "title": "Web Site Design Process"
    },
    {
      "number": 3,
      "title": "Web Design Patterns"
    }
  ]
}
```
# Overview of major open APIs

<table>
<thead>
<tr>
<th>API</th>
<th>Category</th>
<th>URL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>New York Times APIs</td>
<td>News</td>
<td><a href="http://Developer.nytimes.com">http://Developer.nytimes.com</a></td>
<td>Article search, best sellers, campaign finance, community, most popular, real estate, movie reviews, ...</td>
</tr>
<tr>
<td>Interworx API</td>
<td>Web hosting, control panel</td>
<td><a href="http://docs.interworx.com/interworx/api/">http://docs.interworx.com/interworx/api/</a></td>
<td>Documentation and full API for the Interworx Control Panel, including Nodeworx and Siteworx</td>
</tr>
<tr>
<td>The Guardian Open Platform</td>
<td>News</td>
<td><a href="http://www.guardian.co.uk/open-platform">http://www.guardian.co.uk/open-platform</a></td>
<td>Content API, politics API, data store</td>
</tr>
<tr>
<td>Google Code</td>
<td>Search and advertising</td>
<td><a href="https://developers.google.com/products?hl=de">https://developers.google.com/products?hl=de</a></td>
<td>AdSense, AdWords, feed, language, search, blog, news, ...</td>
</tr>
<tr>
<td>Facebook</td>
<td>Social networking</td>
<td><a href="https://developers.facebook.com/">https://developers.facebook.com/</a></td>
<td>Graph, authentication, social</td>
</tr>
<tr>
<td>Twitter</td>
<td>Microblogging</td>
<td><a href="https://dev.twitter.com/docs">https://dev.twitter.com/docs</a></td>
<td>Core Twitter data, search, trends data</td>
</tr>
<tr>
<td>Yahoo! Developer Network</td>
<td>Search and advertising</td>
<td><a href="http://developer.yahoo.com/">http://developer.yahoo.com/</a></td>
<td>PlaceFinder, YQL, OAuth, ...</td>
</tr>
</tbody>
</table>

Exemplary REST API: The Twitter API (1)

Some of the available methods:

**statuses**
- GET statuses/mentions_timeline
- GET statuses/home-timeline
- GET statuses/retweets_of_me
- GET statuses/retweets/:id
- GET statuses/show/:id
- GET statuses/destroy/:id
- GET statuses/update
- GET statuses/retweet:id
- GET statuses/update_with_media
- GET statuses/oembed
- GET statuses/retweeters/ids
- GET statuses/lookup

**friends**
- GET friendships/no_retweets/ids
- GET friendships/incoming
- GET friendships/outgoing
- POST friendships/create
- POST friendships/destroy
- POST friendships/update
- GET friendships/show
- GET friendships/lookup

**media**
- POST media/upload

**direct_messages**
- GET direct_messages
- POST direct_messages/destroy
- POST direct_messages/new

**followers**
- GET friends/ids
- GET friends/list

Fielding would not call that REST: Using POST for a deletion or GET for an update is bad practice!

... and many more

(https://dev.twitter.com/rest/public)
Example: Retrieving the ids of all retweeters of a certain status

Request:

GET
https://api.twitter.com/1.1/statuses/retweeters/ids.json?id=327473909412814850&count=100&stringify_ids=true

Answer:

```json
{
    "previous_cursor": 0,
    "ids": [ "1382021622", "931150754", "1364953914", "92313481", "1398853771" ],
    "previous_cursor_str": "0",
    "next_cursor": 0,
    "next_cursor_str": "0"
}
```

([https://dev.twitter.com/rest/public](https://dev.twitter.com/rest/public))
REST API in practice: API specification

- There are multiple frameworks and languages for specifying (REST) Web APIs
  - Swagger: http://swagger.io
  - RAML: http://raml.org
  - etc.

- REST API specification through JSON (or YAML) document
  - **General information** about the REST API
    - How to connect to the REST API?
    - What is the data format for the REST API (e.g., JSON, XML)?
  - **Definition of resource types**
    - Which resources are available through the REST API?
    - Which attributes of which types (e.g., integer) do the resources have?
  - **Definitions of operations**
    - Which operations are available?
    - Which REST verb to use for which URL?
    - Which parameters are allowed and/or required?
    - How does the response look like?
    - How does errors get reported?
```json
{
  "swagger": "2.0",
  "host": "petstore.swagger.wordnik.com",
  "basePath": "/api",
  "consumes": ["application/json"],
  "produces": ["application/json"],
  "definitions": {
    "pet": {
      "required": ["id", "name"],
      "properties": {
        "id": {
          "type": "integer",
          "format": "int64"
        },
        "name": {
          "type": "string"
        },
        "tag": {
          "type": "string"
        }
      }
    },
    "errorModel": {
      "required": ["code", "message"],
      "properties": {
        "code": {
          "type": "integer",
          "format": "int32"
        },
        "message": {
          "type": "string"
        }
      }
    }
  },
  "paths": {
    "/pets": {
      "get": {
        "description": "Returns all pets from the system",
        "operationId": "findPets",
        "produces": ["application/json"],
        "parameters": [
          {
            "name": "limit",
            "in": "query",
            "description": "maximum number of results",
            "required": false,
            "type": "integer",
            "format": "int32"
          }
        ],
        "responses": {
          "200": {
            "description": "pet response",
            "schema": {
              "type": "array",
              "items": {
                "$ref": "#/definitions/pet"
              }
            }
          },
          "default": {
            "description": "unexpected error",
            "schema": {
              "$ref": "#/definitions/errorModel"
            }
          }
        }
      }
    }
  }
}
```

http://swagger.io/
The REST API Maturity Model describes **four levels** of REST APIs

- A higher level indicates a **better adherence** to the REST **constraints**

<table>
<thead>
<tr>
<th>LEVEL 0: POX (Plain old XML)</th>
<th>LEVEL 1: Resources</th>
</tr>
</thead>
</table>
| - The API uses HTTP as transport protocol, i.e., to tunnel requests and responses | - The API distinguishes between resources
| - Essential attributes: | - Essential attributes:
|  - One URI |  - One URI for each resource
|  - One HTTP method (usually POST) |  - Still a single HTTP method |
|  - Examples: SOAP and XML-RPC | |

<table>
<thead>
<tr>
<th>LEVEL 2: HTTP Verbs</th>
<th>LEVEL 3: Hypermedia</th>
</tr>
</thead>
</table>
| - The API makes **use of HTTP verbs**, e.g., GET for requesting resources, or DELETE to delete them | - Use of **HATEOAS** (Hypermedia as the Engine of Application State)
| - Essential attributes: | - Clients are robust against changes of the REST API
|  - Multiple URIs | - Essential attributes
|  - Multiple HTTP methods |  - Resources explain themselves |

[http://restcookbook.com/Miscellaneous/richardsonmaturitymodel/](http://restcookbook.com/Miscellaneous/richardsonmaturitymodel/)
Postman allows to interactively test your REST API

Client-side libraries and frameworks

Communication

Web application server-side framework

Persistency layer

Backend Project

HTTP request

HTTP response

REST specification

NodeJS

JWT  Mongoose  Express

npm

todemon

MongoDB
<table>
<thead>
<tr>
<th>Http Verb (CRUD operation)</th>
<th>URL</th>
<th>Authentication</th>
<th>expected request body</th>
<th>expected response</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET (READ)</td>
<td>/movies</td>
<td>No</td>
<td>[empty]</td>
<td>serialized movie object (JSON)</td>
</tr>
<tr>
<td>GET (READ)</td>
<td>/movies/{movieId}</td>
<td>No</td>
<td>[empty]</td>
<td>array of serialized movie objects (JSON)</td>
</tr>
<tr>
<td>POST (CREATE)</td>
<td>/movies</td>
<td>Yes</td>
<td>serialized movie object (JSON)</td>
<td>serialized movie object (JSON)</td>
</tr>
<tr>
<td>PUT (UPDATE)</td>
<td>/movies/{movieId}</td>
<td>Yes</td>
<td>serialized movie object (JSON)</td>
<td>serialized movie object (JSON)</td>
</tr>
<tr>
<td>DELETE (DELETE)</td>
<td>/movies/{movieId}</td>
<td>Yes</td>
<td>[empty]</td>
<td>[empty]</td>
</tr>
<tr>
<td>GET</td>
<td>/auth/me</td>
<td>Yes</td>
<td>[empty]</td>
<td>serialized user object (JSON)</td>
</tr>
<tr>
<td>GET</td>
<td>/auth/logout</td>
<td>Yes</td>
<td>[empty]</td>
<td>JSON object with token as null</td>
</tr>
<tr>
<td>POST</td>
<td>/auth/login</td>
<td>No</td>
<td>serialized JSON object {&quot;username&quot;: &quot;johndoe&quot;, &quot;password&quot;: &quot;swordfish&quot;}</td>
<td>JSON object with JSON Web Token</td>
</tr>
<tr>
<td>POST</td>
<td>/auth/register</td>
<td>No</td>
<td>serialized JSON object {&quot;username&quot;: &quot;johndoe&quot;, &quot;password&quot;: &quot;swordfish&quot;}</td>
<td>JSON object with JSON Web Token</td>
</tr>
</tbody>
</table>
Example of HTTP request aligned to the provided REST API specification and JSON as content exchange format

PUT /api/movies/5730be244f2424d42c7e4110 HTTP/1.1
Host: sebis-movie.com
Authorization: JWT eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VyIjp7Il9pZCI6IjU3MzBiZTEwNGYyNDI0ZDQyYzd1NDEwZiIsInVzZXJuYW1lIjoiaGFucyJ9fQ.67mjkRUkkPqLLzTPBNyDjpP6OF9zNYO_v0JBRj4NuSw
Content-Type: application/json
Cache-Control: no-cache

Request body:
{
    "id": "5730be244f2424d42c7e4110",
    "title": "Interstellar",
    "synopsis": "In Earth's future, a global crop blight and second Dust Bowl are slowly rendering the planet uninhabitable. Professor Brand (Michael Caine), a brilliant NASA physicist, is working on plans to save mankind by transporting Earth's population to a new home via a wormhole. But first, Brand must send former NASA pilot Cooper (Matthew McConaughey) and a team of researchers through the wormhole and across the galaxy to find out which of three planets could be mankind's new home."
    "year": "2000",
    "user": "5730be104f2424d42c7e410f"
}
Creating REST interfaces with Express

Frontend Project

Client-side libraries and frameworks

React based client

Communication

Static Files
(JavaScript, JSX, CSS files, images, etc.)

HTTP request

Web application server-side framework

NodeJS (dev)

- npm
- babel
- webpack

HTTP response

Persistence layer

Persistency layer

NodeJS

- JWT
- Mongoose
- Express

- npm
- nodemon

Backend Project

REST specification

MongoDB
Server-side web frameworks (a.k.a. "web application frameworks") are software frameworks that make it easier to write, maintain and scale web applications. They provide tools and libraries that simplify common web development tasks, including routing URLs to appropriate handlers, interacting with databases, supporting sessions and user authorization, formatting output (e.g. HTML, JSON, XML), and improving security against web attacks.

Express is a web application framework on top of Node.js
Express provides a thin layer of fundamental web application features, without obfuscating Node.js features. In particular, it is a set of higher level asynchronous Node.js methods that were implemented by experienced developers taking into account:

- Security
- Error handling
- Routing
- Middleware modules (request parsers, cors, authentication module (passport), compressors) (more)
- Template engines (avoided in our example architecture, REST interfaces instead)

Express is “unopinionated” It’s up to the developer to use certain middleware modules or templates engines (developers rely on their own decisions and experiences of others)

Similar frameworks: Koa, Hapi, Restify
Frameworks based on top of Express (usually opinionated): LoopBack, Sails, Locomotive
Express: Typical project structure and app initialization and middleware

api.js example

```
"use strict";

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

const middlewares = require('./middlewares');
const auth = require('./routes/auth');
const movie = require('./routes/movie');

const api = express();

// Adding Basic Middlewares
api.use(helmet());
api.use(bodyParser.json());
api.use(bodyParser.urlencoded({ extended: false }));
api.use(middlewares.AllowCrossDomain);

// Basic route
api.get('/', (req, res) => {
  res.json({
    name: 'SEBA Master Movie Backend'
  });
});

// API routes
api.use('/auth', auth);
api.use('/movies', movie);

module.exports = api;
```

server.js example

```
"use strict";

const http = require('http');
const mongoose = require('mongoose');
const api = require('./src/api');
const config = require('./src/config');

const server = http.createServer(api);

// Set the port to the API
server.set('port', config.port);

// Create a http server based on Express
server = http.createServer(api);

// Connect to the MongoDB database; then start the server
mongoose.connect(config.mongodb).
  .then(() => server.listen(config.port))
  .catch(err => {
    console.log('Error connecting to the database', err.message);
    process.exit(err.statusCode);
  });

server.on('listening', () => {
  console.log('API is running in port ${config.port}');
});

server.on('error', (err) => {
  console.log('Error in the server', err.message);
  process.exit(err.statusCode);
});

```
Express: Defining REST API resources (routing)

movie.js example

```javascript
"use strict";

const express = require('express);
const router = express.Router();

const middlewares = require('./middlewares');
const MovieController = require('./controllers/movie');

router.get('/', MovieController.list); // List all movies
router.post('/', middlewares.checkAuthentication, MovieController.create); // Create a new movie
router.get('/:id', MovieController.read); // Read a movie by Id
router.put('/:id', middlewares.checkAuthentication, MovieController.update); // Update a movie by Id
router.delete('/:id', middlewares.checkAuthentication, MovieController.remove); // Delete a movie by Id

module.exports = router;
```
Express: Controllers

movie.js example

```
"use strict";

const MovieModel = require('./models/movie');

const create = (req, res) => {
  if (Object.keys(req.body).length === 0) return res.status(400).json({
    error: 'Bad Request',
    message: 'The request body is empty'
  });

  MovieModel.create(req.body)
  .then(movie => res.status(201).json(movie))
  .catch(error => res.status(500).json({
    error: 'Internal server error',
    message: error.message
  }));

const read = (req, res) => {
  MovieModel.findById(req.params.id).exec()
  .then(movie => {
    if (!movie) return res.status(404).json({
      error: 'Not Found',
      message: 'Movie not found'
    });
    res.status(200).json(movie)
  })
  .catch(error => res.status(500).json({
    error: 'Internal server Error',
    message: error.message
  }));

const update = (req, res) => {
  if (Object.keys(req.body).length === 0) return res.status(400).json({
    error: 'Bad Request',
    message: 'The request body is empty'
  });

  MovieModel.findByIdAndUpdate(req.params.id, req.body, { new: true, runValidators: true }).exec()
  .then(movie => res.status(200).json(movie))
  .catch(error => res.status(500).json({
    error: 'Internal server error',
    message: error.message
  }));

```
Outline of the lecture

6. Building a REST-enabled back-end service

- Target architecture and development environment
- Event-driven architecture and asynchronous I/O operations with node.js
- Creating REST interfaces using express.js
- Using document-oriented database storage: MongoDB example
- Enabling user authentication on the web service using JSON web token (JWT)
Using document-oriented database storage: MongoDB example

- **Client-side libraries and frameworks**: React based client
- **Communication**: Static Files (JavaScript, JSX, CSS files, images, etc.)
- **Web application server-side framework**: NodeJS (dev) with npm, babel, webpack
- **Persistency layer**: NodeJS with JWT, Mongoose, Express, npm, nodemon

**Frontend Project**

- **Backend Project**
  - HTTP request
  - HTTP response
  - REST specification
  - MongoDB
NoSQL databases in web application development

Reasons for popularity in web development:
• Flexible data model → database schema can evolve with business requirements
• Complex data types
• Scaling
• API easy to work with (no need for object relational mapping)

When relational databases are still better to use?
• In applications that require complex, multi-row transactions. E.g., travel reservation system or core banking application (both rely on complex transactions)
• In legacy applications, where all business logic is built around a relational data model and SQL.
Document-oriented databases

MongoDB - Terminology

<table>
<thead>
<tr>
<th>RDBMS</th>
<th>MongoDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database</td>
<td>Database</td>
</tr>
<tr>
<td>Table</td>
<td>Collection</td>
</tr>
<tr>
<td>Row</td>
<td>Document</td>
</tr>
<tr>
<td>Column</td>
<td>Field</td>
</tr>
<tr>
<td>Table Join</td>
<td>Embedded Documents</td>
</tr>
<tr>
<td>Primary Key</td>
<td>Primary Key (Default key: _id)</td>
</tr>
<tr>
<td>Partition</td>
<td>Shard</td>
</tr>
<tr>
<td>Partition Key</td>
<td>Shard Key</td>
</tr>
</tbody>
</table>

Database server and client

| mysqlld                      | mongod                       |
| mysql                        | mongo                         |

- Collections do not enforce a schema.
- Collections can be created on demand.
- A document is a set of key-value pairs and can have dynamic schema.
- Typically, all documents within a collection are of similar or related purpose.
Document-oriented databases

Sample document and collection

```
{
    name: "sue", field: value
    age: 26, field: value
    status: "A", field: value
    groups: [ "news", "sports" ] field: value
}
```

```
{ name: "al",
  age: 18,
  status: "D",
  groups: [ "politics", "news" ]
}
```

<table>
<thead>
<tr>
<th>String</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>Double</td>
</tr>
<tr>
<td>Arrays</td>
<td>Timestamp</td>
</tr>
<tr>
<td>Object</td>
<td>Null</td>
</tr>
<tr>
<td>Symbol</td>
<td>Date</td>
</tr>
<tr>
<td>Object ID</td>
<td>Binary Data</td>
</tr>
<tr>
<td>JS Code</td>
<td>Reg Expressions</td>
</tr>
</tbody>
</table>

(Expressions: MongoDB)
Document-oriented databases

MongoDB – inserting document into collection

```javascript
db.users.insert({
  name: "sue",
  age: 26,
  status: "A",
  groups: ["news", "sports"]
})
```

[Document databases, MongoDB]
Document-oriented databases

MongoDB – querying data

```javascript
db.users.find({ age: { $gt: 18 } }).sort({ age: 1 })
```
Mongoose object document mapper for MongoDB concepts to JS objects

- **Frontend Project**
  - React based client
- **Backend Project**
  - HTTP request
  - HTTP response
  - REST specification

**Client-side libraries and frameworks**
- React

**Communication**
- Static Files (JavaScript, JSX, CSS files, images, etc.)

**Web application server-side framework**
- NodeJS (dev)
  - npm
  - babel
  - webpack
- Express
- Mongoose
- JWT
- npm
- nodemon

**Persistency layer**
- MongoDB
Connecting to the database

There are two ways to connect MongoDB from NodeJS app:

- MongoDB Node.js native driver ([http://mongodb.github.io/node-mongodb-native/2.2/](http://mongodb.github.io/node-mongodb-native/2.2/))
  - Pros:
    - Preserves the schemaless paradigm, no modelling required from the beginning
    - Supported by MongoDB developers
  - Cons:
    - Type casting, data validation have to be implemented by the developers on their own inside NodeJS app (harder for the beginners)

- Mongoose ([http://mongoosejs.com/](http://mongoosejs.com/))
  - Pros:
    - Data validation, type casting, business logic boilerplates are provided to the developers
    - By design introduces the concept of data models in your NodeJS application providing separation of concerns
  - Cons:
    - Data model have to be introduced from the beginning (contradicts to the schemaless paradigm)
Defining a data schema with mongoose

**movie.js: schema definition**

```javascript
const mongoose = require('mongoose');

// Define the movie schema

const MovieSchema = new mongoose.Schema({
  title: {
    type: String,
    required: true
  },
  synopsis: String,
  runtime: Number,
  mpaa_rating: String,
  year: {
    type: Number,
    required: true
  },
  posters: {
    thumbnail: String,
    profile: String,
    detailed: String,
    original: String
  }
});

MovieSchema.set('versionKey', false);
MovieSchema.set('timestamps', true);

// Export the Movie model
module.exports = mongoose.model('Movie', MovieSchema);
```

**user.js: schema definition**

```javascript
const mongoose = require('mongoose');

// Define the user schema

const UserSchema = new mongoose.Schema({
  username: {
    type: String,
    required: true
  },
  password: {
    type: String,
    required: true,
    unique: true
  }
});

UserSchema.set('versionKey', false);

// Export the User model
module.exports = mongoose.model('User', UserSchema);
```

**mongoose documentation:** [http://mongoosejs.com/docs/guide.html](http://mongoosejs.com/docs/guide.html)
### Resulting JS objects

<table>
<thead>
<tr>
<th>Movie</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>_id: ObjectId</td>
<td>_id: ObjectId</td>
</tr>
<tr>
<td>title: String</td>
<td>username: String</td>
</tr>
<tr>
<td>synopsis: String</td>
<td>password: String</td>
</tr>
<tr>
<td>runtime: Number</td>
<td>year: Number</td>
</tr>
<tr>
<td>mpaa_rating: String</td>
<td>posters: { thumbnail: String, … }</td>
</tr>
<tr>
<td>year: Number</td>
<td>... [many more] ...</td>
</tr>
<tr>
<td>posters: { thumbnail: String, … }</td>
<td>... [many more] ...</td>
</tr>
</tbody>
</table>

**Movie**

- find (object conditions, function cb) : Query
- findById (number id, function cb) : Query
- update (object conditions, object updates) : Query
- ... [many more] ...

**User**

- find (object conditions, function cb) : Query
- ... [many more] ...
Utilizing resulting JS objects

Example: findByIdAndUpdate() – predefined asynchronous method defined for mongoose Schema object, that allows to update the document in the collection if the documents exists, otherwise error occurs.
Outline of the lecture

6. Building a REST-enabled back-end service
   - Target architecture and development environment
   - Event-driven architecture and asynchronous I/O operations with node.js
   - Creating REST interfaces using express.js
   - Using document-oriented database storage: MongoDB example
   - Enabling user authentication on the web service using JSON web token (JWT)
Access control: User/client authentication and authorization

Access controls give web service owners the ability to control, restrict, monitor, and protect resource availability, integrity and confidentiality.

Access Control Challenges

- Various types of users need different levels of access (content readers, editors, platform administrators, etc.)
- Resources have different classification levels (Confidential, internal use only, private, public, etc.)
- Diverse identity data must be kept on different types of users (Credentials, personal data, contact information, work-related data, digital certificates, cognitive passwords, etc.)
- The corporate environment is continually changing (Business environment needs, resource access needs, employee roles, actual employees, etc.)

Access Control Criteria

- Roles
- Groups
- Location
- Time
- Transaction Type

Authentication: Verification of the identity of a user or a client
Authorization: Verification of a user’s or client’s permissions to access a resource

Methods for user and client authentication

Manual implementation of authorization is time consuming and error prone.

The following has to be taken into account:

• Simple log in/sign up flow
• Encryption and secure data transfer
• Password reset
• Email verification
• Sustainability against different brute force attacks
• …

There are different types of HTTP/HTTPS-based methods for user and client authentication methods addressing above challenges:

• Basic Authentication
• Session-based Authentication
• JSON Web Tokens (JWT)
• OpenID
• oAuth

All of them supported by specifically tailored modules/packages/libraries in various languages (implemented and maintained by communities of developers)
Basic Authentication

+ Simple, improves interoperability
  • Any kind of client can use basic authentication
+ Stateless
  • Each request contains everything required to determine the user identity
  - Basically no security measures
    • Although Base64-encoded, the password is sent in clear-text. Therefore, it should only be used in conjunction with SSL / HTTPS
    • On the client, the password is cached or permanently stored in clear-text for future requests to the server
Session-based authentication (with cookies)

+ More secure than Basic Authentication
  - Apart from the initial exchange of credentials, the password is not included in the requests anymore
  - Password has not to be cached or stored on the client

- Stateful
  - Requests only contain the session ID, by which the server restores the corresponding session
JSON Web Tokens (JWT)

Illustration of process

+ Security
  - Apart from the initial exchange of credentials, the password is not sent in clear-text
  - Additional security concepts: Token expiration and token revocation

+ Stateless
  - The JWT contains everything required to determine and verify the user identity

- Still vulnerable to attacks
  - Even when using asynchronous encryption methods, JWT are usually still vulnerable to various threats (e.g., man-in-the-middle attacks)
  - Security standard depends on used hash methods and the token vendor / provider
JSON Web Tokens (JWT)

Structure of a Token

- Emphasis on API authentication, especially for JSON-based REST APIs
- JWTs are JSON objects consisting of 3 parts:
  - The **header** describes the type of the token as well as the hashing algorithm used for the signature
  - The **payload** contains multiple claims, e.g., user name, expiration date, etc.
  - The **signature** is generated by the hash of the header, payload, and a secret held by the server. It ensures the integrity of the token.

```
{  "alg": "HS256",  "typ": "JWT"}
{  "sub": "1234567890",  "name": "John Doe",  "admin": true}
HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), secret)
```

[https://jwt.io/](https://jwt.io/)
OpenID

Motivation and involved roles

- Allow users to use a login and password from one web-site to login to other web-sites.
  - Example: Using a Google account to login at Wordpress.org
  - The service provider never gets access to the login and password of the identity provider.

Involved Roles:

- **User** needs to authenticate against a Relying Party with his digital identity.
- The **Identifier** represents a user and is typically represented as a url. It points to a resource that holds information on this user (e.g., OpenID provider url, version of OpenID, ...).
- The **Relying Party** accepts an assertions from an OpenID Provider that represents a digital entity of a specific user.
- The **OpenID Provider** or **Identity Provider** is responsible for authenticating the User against a relying party.
- The **Identifier Host** manages the Identifier-describing resources.

OpenID

Typical protocol flow

1. Protocol is initiated by the User requesting the Relying Party's (RP) site.
2. RP responds with its login page.
3. Users enters his Identifier
4. RP performs discovery
5. Identifier Host sends metadata
6. RP requests a shared secret
7. Identity Provider responds with key
8. RP redirects to OpenID Provider
9. Login form is presented to User
10. Users fills out login form
11. OpenID Providers verifies the Users credentials
12. User is either authenticated against the RP or an adequate error message is presented.

Authentication with JWT

Frontend Project

Client-side libraries and frameworks

React based client

Communication

Static Files
(JavaScript, JSX, CSS files, images, etc.)

Web application server-side framework

NodeJS (dev)
npm  babel  webpack

Persistency layer

NodeJS

JWT  Mongoose  Express

npm

nodemon

Backend Project

HTTP request

HTTP response

REST specification

MongoDB

Web Application Engineering 2019 - Building a REST-enabled backend service

© sebis
Authentication with JWT

Implementation as middleware

→ Middleware can be added to certain routes and executes independently before calling actual route processing function.

Examples of existing middlewares:

- **json body-parser**: parses JSON-formatted request
- **cors**: Modifies headers to accept cross-server requests

Our middleware function parses the JWT token from the auth header.
If JWT is valid → injects user to the request payload → ready for e.g. further authorization.
Authentication with JWT

user.js example

```javascript
"use strict";

const express = require('express');
const router = express.Router();

const middlewares = require('./middlewares');
const AuthController = require('./controllers/auth');

router.post('/login', AuthController.login);
router.post('/register', AuthController.register);
router.get('/me', middlewares.checkAuthentication, AuthController.me);
router.get('/logout', middlewares.checkAuthentication, AuthController.logout);

module.exports = router;
```

middlewares.js: example of middleware for authentication

```javascript
"use strict";

const jwt = require('jsonwebtoken');
const config = require('./config');

const checkAuthentication = (req, res, next) => {
  // check header or url parameters or post parameters for token
  const token = req.headers['x-access-token'];

  if (!token)
    return res.status(401).send({
      error: 'Unauthorized',
      message: 'No token provided in the request'  
    });

  // verifies secret and checks exp
  jwt.verify(token, config.jwtSecret, (err, decoded) => {
    if (!err) return res.status(200).send({
      error: 'Unauthorized',
      message: 'Failed to authenticate token.'
    });

    if (err && err.statusCode === 401)
      return res.status(401).send({
        error: 'Unauthorized',
        message: 'Failed to authenticate token.'
      });

    // if everything is good, save to request for use in other routes
    req.userId = decoded.id;
    next();
  });
};
```