Node.js

Course Material • Hamburg Coding School • 20.01. - 01.02.2021

—

Outline

- Node.js
    - What is Node.js?
    - Node.js Properties and Terms
    - Call stack, Callback Queue and Task Queue (optional knowledge)
- NPM
    - Modules and require()
    - Package.json
- REST APIs
    - The HTTP Protocol
    - RESTful Web Services
    - Request and Response
    - User Agent
    - CRUD Operations
    - HTTP Methods
    - Parts of a URL
    - Form Input
    - Express.js
- Asynchronous Programming
    - Callbacks
    - Promises
    - Async-Await
- Connect Node to a Database
    - NoSQL Databases
    - Mongoose
    - Connect to the Database
    - Error Handling
    - Mongoose Schema
    - Model
    - Get All
    - Post New Data
    - Middleware
    - Validation
- Best Practices

- Documentation
- Endpoints
- Validation and Security

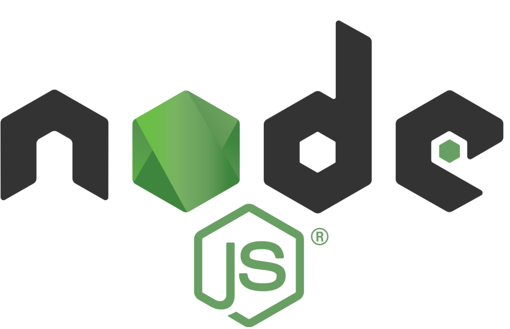
Node.js

What is Node.js?

Until 2009, JavaScript could only run inside a browser. In 2009, **Ryan Dahl** took the V8 JavaScript engine of the Chrome browser and embedded it in a C++ program to be able to run it outside of a browser and called it **Node**.

Node.js is not a programming language or framework, but a **runtime environment** for executing JavaScript code.

https://nodejs.org



Node.js Properties and Terms

Runtime environment: Node is a program to run other programs in. In this case, we want to run JavaScript programs, but not as usual in the browser, but on a server machine.
Node is a runtime environment where we can run JavaScript code on a server.

Single-threaded: Runs on only 1 thread.
A thread is a “space” for one program to run.
More than one thread can run in a process.
A process is what runs on an operating system (you can see them if you open your Activity Monitor (Mac) / Task Manager (Windows).
Node is single-threaded, which means it uses only one thread.

Callback-based: A callback is a function that is passed to another function.
It is called if a certain state is reached, e.g. the calculation in the function has a result.
Node.js uses callbacks: The programmer can pass a callback function into the built-in functions to be notified about certain events (see code examples below).

Event Loop: Node.js has an event loop. This is an infinite loop that is run over and over and over, and what is added to it (events), is executed in the next iteration of the loop.

Call Stack, Callback Queue and Task Queue (optional knowledge)

Call Stack: Node.js maintains a call stack.
This is a place where it stores all the functions that need to be run next.

Callback Queue: A place where node.js puts all callbacks it encounters.

Task-Queue: A queue where node.js stores all things that need to be executed before they can be put on the call stack.

This is optional knowledge. You can very well build node applications without it. If you want to dive in deep and understand the event queue better, we recommend this video:

- https://www.youtube.com/watch?v=8aGhZQkoFbQ

NPM

NPM - Node Package Manager

The node package manager (npm) is a tool that can install node packages.
Node packages are something like JavaScript libraries.

Npm can be used for managing (installing, removing, updating...) node packages.

It uses a semantic versioning scheme. This means, the versions of the packages look like this: **`1.12.7`**

https://www.npmjs.com/

Modules and require()

A Module is a JavaScript library: a set of functions you want to load into your project.

Node brings a bunch of predefined modules. These you can use without installation.

If you want to use other modules, you need to use the built-in **`require()`** function.

```
var http = require('http');
```

This loads the module “http”.

The difference between module and package:

Module: a JavaScript library, or simply a set of JavaScript functions. Modules are libraries for Node.js.

Package: one or more modules packaged together. Node.js uses a package manager (npm) to find and install these packages.

Package.json

If you need a certain package for your project, you put it into a file called **package.json**.

When you created your project with express, it created the file for you. It looks something like this:

```
{
  "name": "travelblog",
  "version": "0.0.0",
  "private": true,
  "scripts": {
    "start": "node ./bin/www"
  },
  "dependencies": {
    "cookie-parser": "~1.4.4",
    "debug": "~2.6.9",
    "express": "~4.16.1",
    "http-errors": "~1.6.3",
    "morgan": "~1.9.1",
    "pug": "2.0.0-beta11"
  }
}
```

In the section **`dependencies`** you see all the packages that are included for your project. They have the format **`"package-name": "version"`**.

You manage all your project's dependencies (meaning: packages that your project depends on) in this package.json file.

On your command line, navigate into your project's folder where the package.json file is, and type:

```
npm install
```

This installs all packages that are specified in your package.json.

Vice versa, you can also install a package directly on the command line, and it will be added to package.json. For example:

```
npm install body-parser
```

This installs the body-parser package for your project and automatically adds it to your package.json file.

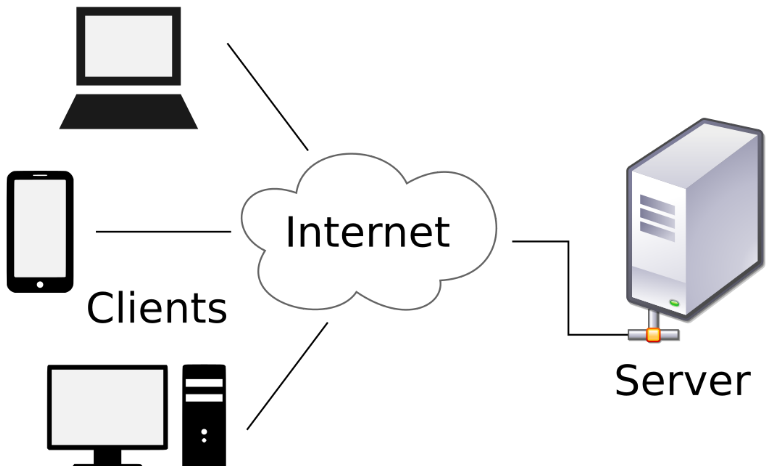
When you install your dependencies, node creates the file **package-lock.json** automatically. This is a file that makes sure that node knows exactly which packages of which versions to use. You don't need to understand its content. Just know that it is important for node, especially if multiple people work with the same code base.

> 👍 **Best practice:** If you use git, always add and commit your ***package.json*** and ***package-lock.json*** file.

REST APIs

Our goal in this course is to build a ***REST API***.

An ***API (Application Programming Interface)*** is an interface where an application (i.e. a ***client application***) can get data or send data to another application (i.e. an ***application server***).



For this, APIs usually follow certain standards.

The HTTP Protocol

HTTP (HyperText Transfer Protocol) is a protocol that is used for the web. It defines how client and server communicate with each other for websites and web services. It defines for example that the client does a ***request***, and the server replies with a ***response***. The protocol defines what these requests and responses look like.

RESTful Web Services

REST (REpresentational State Transfer) is an architectural style for creating APIs with HTTP. It uses the HTTP protocol so that the client can request data or send data to a server application. This data is in JSON format.

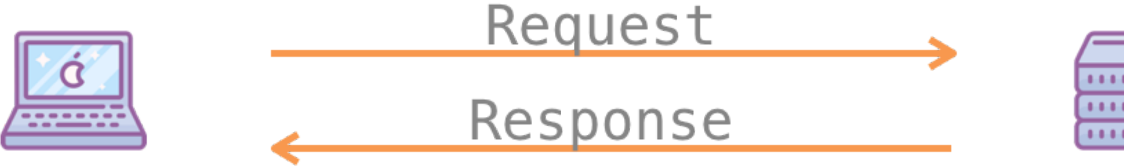
A server application that uses REST is called a ***RESTful web service***.

Example REST APIs:

Star Wars people: https://swapi.co/api/people/
Beers: https://api.punkapi.com/v2/beers

Request and Response

If a client application wants to get data from a RESTful web service, it sends an HTTP request. The server then sends the data via an HTTP response.

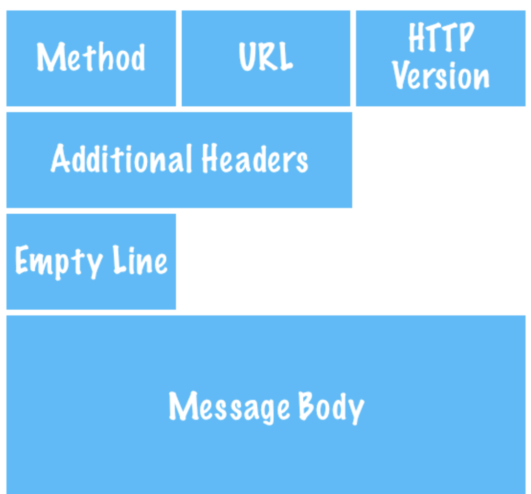


The same happens if the client wants to send data to the server. The data is stored in the body of the request. The server then sends a response to acknowledge whether or not the storing of the data was successful.

In the HTTP protocol, request and response are standardized. It is defined how many text fields exist, in which sequence, and how much text is allowed in that field.

Request

The fields of the request follow this order:



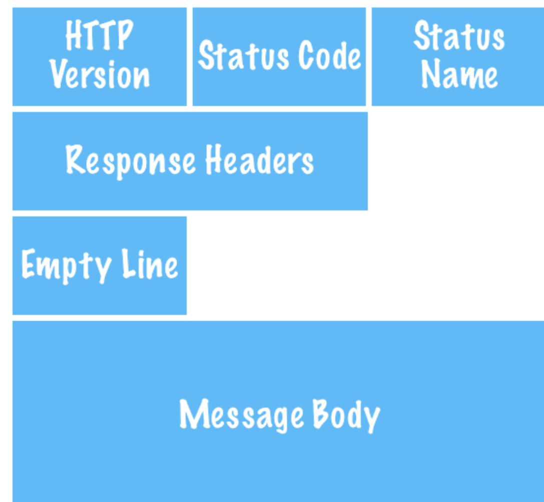
Under the hood, HTTP uses only text. In the message body, there can be binary data as well, for example if an image file is requested. The header, however, only contains text.

The request head looks something like this:

```
GET / HTTP/1.1
Host: www.spiegel.de
Connection: keep-alive
Purpose: prefetch
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_0) AppleWebKit/537.36 (KHTML…
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/
Accept-Encoding: gzip, deflate
Accept-Language: de
Cookie: __gads=ID=b983721bda83d7ae:T=1487252257:S=ALNI_MYlth5Tx71QzpXnN3I0AoPWAu4Uog;…
```

Response

The fields of the response follow this order:



As text, a response will look something like this:

```
HTTP/1.1 200 OK
Date: Thu, 25 Oct 2018 08:54:53 GMT
Cache-Control: no-transform
Expires: Thu, 25 Oct 2018 08:55:38 GMT
X-SP-TE: 5001893
X-Robots-Tag: index, follow, noarchive, noodp
Content-Type: text/html;charset=UTF-8
Content-Encoding: gzip
X-SP-AP: 5001887
Vary: Accept-Encoding, isssl
Age: 17
X-SP-PR: 5001887
Accept-Ranges: bytes
Content-Length: 68939
Connection: keep-alive

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd"><html lang="de">
<head>
<title>SPIEGEL ONLINE - Aktuelle Nachrichten</title>
<meta http-equiv="X-UA-Compatible" content="IE=edge" />
…
```

You can see the empty line separating the header from the body.

In this example, the body contains HTML, because a website was requested here. For REST APIs, the body would contain data in JSON format instead.

💻 **Try out REST APIs with Postman:**
To see the header and the body of a request or response, you can use this tool:

https://www.getpostman.com/

User Agent

A ***user agent*** is a software that acts on behalf of the user, i.e. your browser or mobile app. In HTTP, the user agent needs to be specified in a header field, so that the server can use this information to prepare the data.

```
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_0) AppleWebKit/537.36 (KHTML…
```

The user agent usually contains information about

- Device
- Operating system
- Web kit (the browser software)
- JavaScript being enabled or not
- Cookies being enabled or not
- Local time
- Screen resolution
- Screen size

CRUD Operations

REST APIs implement certain methods: ***CRUD***. This stands for:

C - Create
R - Read
U - Update
D - Delete

HTTP Methods

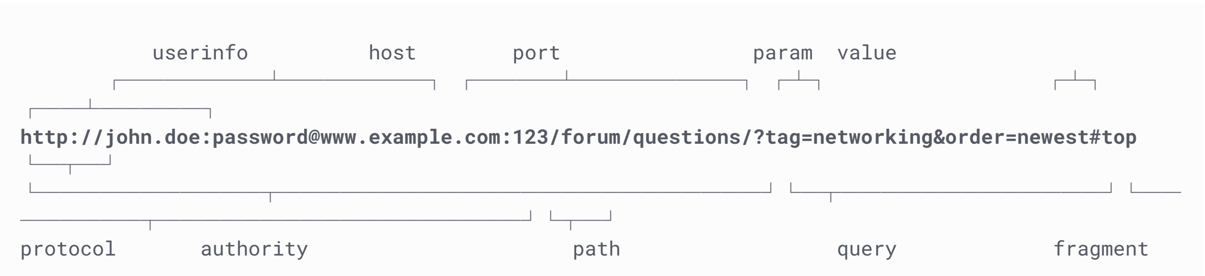
HTTP also has methods that follow the CRUD principle.

These are the HTTP methods we can use:

- **GET**
- **POST**
- **PUT**
- **PATCH**
- **UPDATE**
- **DELETE**

Parts of a URL

A ***URL (Uniform Resource Locator)*** or ***URI (Uniform Resource Identifier)*** is the web address that a client uses to request a document (or JSON data in the case of REST APIs).



Form Input

You can use form input for passing data from the browser to the server.

For that, you create an HTML page with a form, and declare that the form uses the POST method to submit the data that the user typed in.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset=”UTF-8”>
  <title>Locations</title>
</head>
<body>
  <form action=”http://localhost:3000/locations” method=”post”>
    <input type=”text” name=”name” id=”name”>
    <input type=”text” name=”lat” id=”lat”>
    <input type=”text” name=”lng” id=”lng”>
    <input type=”submit”>
  </form>
</body>
</html>
```

This is a website with a form that will call the localhost locations url as a POST request if

the button is pressed. It will submit all form inputs as a string in the body of the request.

Express.js

In this course, we build our own REST API with Node.js. For that, we use a package called ***express***.

Getting started: http://expressjs.com/en/starter/installing.html

Express is an API framework that we can use in a node application to create a REST API.

```
const express = require('express');
const app = express();
const port = 3000

app.use(express.static('public'));

app.get('/', (req, res) => res.send('Hello World!'))

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

`'/'` is the path to the root or home.

Asynchronous Programming

Callbacks

Node.js can handle callbacks. But with humans, it's another story:

When your application grows more complex, you can have callbacks in callbacks in callbacks...

This is hard to read and hard to understand for humans. That's why developers call it the ***callback hell***.

What do we do to solve it? We use ***Promises*** or ***async*** functions.

Promises

Promises are a better alternative to callbacks. They come with methods for the case that everything went well (**resolve**), and for the error case (**reject**).

```
var promise = new Promise(function(resolve, reject) {
  // do what your function is supposed to do

  if (/* everything turned out fine */) {
    resolve("It worked!");
  }
  else {
    reject(Error("This didn't work, because..."));
  }
});
```

You can then use your Promise like this:

```
promise.then(function(result) {
  console.log(result); // "Stuff worked!"
}).catch((error) => {
  console.log('Something went wrong: ' + error);
});
```

You use the function **then** to execute code that should run in the case when everything went well, and the function **catch** to run code in the error case.

Async-Await

There's a special way to work with Promises, called ***async-await***.

You can define a function as **async**, meaning that the result is returned at some later time.

```
async function calculateResult() {
  var result = // stuff your function does
  return result;
}
```

You can then use the function just like a promise:

```
calculateResult()
  .then(/* do something with it */);
```

Async functions will execute in the background. It is not clear, for example, which of

these will return first:

```
calculateResult(param1)
  .then(/* do something with the result of param1 */);

calculateResult(param2)
  .then(/* do something with the result of param2 */);
  // this might finish before the first async function with param1

val random = Math.random();
// this will execute immediately, even before the two async functions above
```

To handle the response in a synchronous way, you can use the keyword **`await`**.

```
val result1 = await calculateResult(param1); // executed first
val result2 = await calculateResult(param2); // executed second
val random = Math.random();                  // executed third
```

Connect Node to a Database

NoSQL Databases

NoSQL databases are databases that don't use SQL. One type of NoSQL databases is document-based databases.

In this course, we use ***MongoDB***, which is a document-based NoSQL database. It saves data in JSON format, which makes it easy to handle with JavaScript.

The schema of MongoDB is very flexible. This can be an advantage, but has its downsides too: if you work with many people on a project, it can be that data is saved that deviates from previous data.

A ***schema*** is a description of how the data looks like. For example, a person has a first name, a last name, an age etc.

In this example, the "person" is called the ***model***.

Mongoose

In Node.js, we use ***mongoose.js*** to access the MongoDB database. It acts like a bridge to the database and creates JSON objects from the documents in the database.

This is also called ***ODM: Object-Document Mapper***. Mongoose is an ODM that connects to MongoDB.

Mongoose is an npm package: https://mongoosejs.com/

Connect to the Database

Create a connection to the database:

```
mongoose.connect('mongodb://localhost/travelBlog', {useNewUrlParser: true });
```

What this does is: it connects to a certain database:

`mongodb`	the protocol (in this case: mongodb)
`localhost`	the host name
`travelBlog`	the database
`{ useNewUrlParser: true }`	use the new url parser, not the old one (don't worry about it: if you don't do that, you just get a deprecation warning)

💡 This is called an **object literal**.

`const myObject = { useNewUrlParser: true }`

This means an object is created here, with a property (`useNewUrlParser`) and a value (`true`).

Connect to the database and do something on the '***open***' event:

```
const mongooseDb = mongoose.connection;
mongooseDb.once('open', function() {
  const app = express();
  resolve(app);
});
```

Here we start an app with express once we have opened the connection to our Mongo database.

Error Handling

With mongoose, you can handle errors with a callback like this:

```
mongooseDb.on('error', (error) => {

});
```

With mongoose, we can use Promises. With Promises, we can do error handling with **`resolve`** and **`reject`** like this:

```
module.exports.initialize = function() {
  return new Promise(function(resolve, reject) {
    mongooseDb.once('open', function() {
      const app = express();
      resolve(app);
    });
    mongooseDb.on('error', reject);
  });
};
```

In the line **`mongooseDb.on('error', reject);`** we can use the `reject` function of the promise.

Mongoose Schema

In mongoose you can create schemas like this:

```
const mongoose = require('mongoose');
let Schema = mongoose.Schema;
let locationSchema = new Schema({
   name:  String,
   lat: Number,
   lng: Number
});
```

Read more in the official documentation: https://mongoosejs.com/docs/guide.html

Model

We need to export our model once before we can use it:

```
let locationModel = mongoose.model('Location', locationSchema);
module.exports.model = locationModel;
```

Once we exported our model, we can require it:

```
const Location = require('./models/location').model;
```

Get All

Now we can use our created data access object.
To get all items in the collection, we can do:

```
const locations = await Location.find({});
```

Make an application that is returning all locations if you type in **`/locations`** as path, like this:

```
async function run() {
  let app = await require('./bootstrap').initialize();

  app.get('/locations', async function (req, res) {
    const locations = await Location.find({});

    res.setHeader('Content-Type', "application/json");
    res.send(locations);
  });

  app.listen(3000, function () {
    console.log('Example app listening on port 3000!');
  });
}

run();
```

This mapping to the **'/location'** URL is also called an ***endpoint***.

Post New Data

To save new data into the database do:

```
const locations = await Location.find({});
```

To take the data from the POST request body and put them into the database, do:

```
async function run() {
  // …
  app.post('/locations', async function (req, res) {
    const name = req.body.name;
    const lat = req.body.lat;
    const lng = req.body.lng;

    Location.create({
      name: name,
      lat: lat,
      lng: lng
  });
  // ...
}

run();
```

Middleware

A middleware is a part of your node.js software that gets executed for every incoming request.

You usually use third-party packages for your middleware.
In our example, we use the body-parser package.

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

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

In this example, we create two middlewares: the first is parsing the body in a url-encoded way, and the second is parsing the body as json content.

You can add as many middlewares as you like. They are put on the **middleware stack**.

💡 **Note:** Because a middleware is called for every incoming request, you need to make sure that your middleware does not do any long or expensive operations.

Validation

If we get text from input fields of a form, we have to check that the user has put in text that is useful and not malicious. For that, we need to do ***validation*** in our code.

For that, we use the package validate.js.

Best Practices

Documentation

Best practices for documentation is a much discussed topic. Some people say, you need to document everything, some people claim that their code is documenting itself.

To find the middle way, ask yourself this question:
Do I need to explain ***why*** I wrote that part of the code? If there is something you need to explain, write documentation.

What? - This is the part that the code already explains on its own.

Why? - This is the part that you need to explain in your documentation.

Endpoints

You should create only one endpoint per model.

For get, put etc. you need to use the same path, but different methods.

GET: Only give out information, don't change anything on the database.

POST: Send data by putting data into the body.

Validation and Security

Studies show that open-source software (e.g. of validation packages) is more secure than proprietary software.

Experience shows that existing software is usually more secure than code you write yourself. (Of course there are exceptions to the rule.)

Glossary

`Node`	A runtime environment for JavaScript outside the browser. Used for building server applications.
`Runtime environment`	A program to run other programs in. Node is a runtime environment to run JavaScript programs in.
`single-threaded`	Running on only one thread, a single “space” for a program to run in.
`callback-based`	A framework or library that relies on callbacks being used by the developer is called callback-based.
`Event loop`	A loop that runs over and over again to execute programs (events) that wait in a queue.
`NPM`	Node Package Manager
`module`	In the node environment, a module is a JavaScript library.
`package`	In the node environment, a package is a module or multiple modules packaged in a way that you can install and manage with NPM.
`package.json`	A file that specifies the configuration for a node application, such as the packages it depends on.
`dependencies`	The set of packages your application depends on, i.e. the libraries that your project uses.
`API`	Application Programming Interface
`HTTP`	HyperText Transfer Protocol
`REST`	REpresentational State Transfer, a certain architectural style for implementing HTTP APIs

`user agent`	A program that acts on behalf of the user, e.g. the browser. Delivers information about the operation system, browser engine, screen size and many more to the server.
`CRUD`	Create Read Update Delete
`URL`	Uniform Resource Locator
`URI`	Uniform Resource Identifier
`Express`	A node package that contains a framework for building REST APIs in a node application.
`Schema`	A definition of what a model in a database should look like.
`model`	The items in a database: tables for an SQL database, or collections for a document-based database. A model is the definition or class representation of a database table or collection.
`ORM`	Object-relational mapper
`ODM`	Object-document mapper
`Mongoose`	A node package for connecting to MongoDB, an ODM for MongoDB.
`Endpoint`	A mapping from a URL or path to a server function, e.g. “`/locations`”
`Middleware`	A part of your node.js software that gets executed for every incoming request.

Useful Links

Node.js: https://nodejs.org/

NPM: https://www.npmjs.com/

Express: http://expressjs.com/

Postman: https://www.getpostman.com/

Mongoose: https://mongoosejs.com/

Mongoose Schema: https://mongoosejs.com/docs/guide.html

Body Parser: https://github.com/expressjs/body-parser

Validate.js: https://validatejs.org/

Compact summary (in German): https://www.techlounge.io/node#episodes