0 Contents

1 NodeJS and other WebServers

This is the interface between the operating system and the actual website itself. -> Apache

- Runs everywhere
- async with events
- easy to deploy
- Lots of APIs for different usecases
  HTTP/HTTPS, URL, FileSystem, Console, UDP, Cryptography, DNS
- modular
- not that much "magic"
  you need to write more yourself

negative: IT IS JS, FFS...

1.1 Async via events

The idea of events is *essentially a queue that doesn't block actual functionality*, this means that your UI still does what it should, as it will be prioritized compared to async functionaity.
Just like bevy, the events are handled in the next frame, or more precise *whenever there is nothing else to do.*
This means we will get the illusion of async/multi-threading, without js actually being able to do so.

1.2 Callback hell with async events

If you constantly need to check for finished events, then you will end up with unreadable code.
The solution is simple: Promises
The default in NodeJS is still callbacks, but there is work on this to change.
Example for callback

```
button.addEventListener('click', function (event) {
  console.log("1. subscription");
});
```

1.3 Example for NodeJS

```
const http = require('node:http'); // lua?

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200; // HTTP response
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello, World!\n');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

2 Modules

2.1 NPM

- npm init
  - -g for global installation
  - –save-dev for a module that will only be used during development
-
- item 3
- item 4

2.2 Exporting Modules

```
function add() { return ++counter; }
function get() { return counter; }
// default export
export default {count: add, get: get};
// named
export {add, get};
```

2.3 Importing Modules

```
// default
import counterA from './counter.mjs';
// Named
import {add, get} from './counter.mjs';
```

2.4 package.json

```
{
  "name": "my_package",
  "description": "",
  "version": "1.0.0",
  "main": "index.js",
  "type": "module",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "repository": {
    "type": "git",
    "url": "https://github.com/mgfeller/my_package.git"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "bugs": {
    "url": "https://github.com/mgfeller/my_package/issues"
  },
  "homepage": "https://github.com/mgfeller/my_package"
}
```

2.5 package-lock.json

This handles dependency problems, where one dependency will be updated but the first package still wants an older one.

```
{
  "name": "my_package",
  "version": "1.0.0",
  "lockfileVersion": 1,
  "requires": true,
  "dependencies": {
    "fancy-calc-demo": {
      "version": "5.0.2",
      "resolved": "https://registry.npmjs.org/fancy-calc-demo/-/fancy-
      calc-demo-5.0.2.tgz",
      "integrity": "sha512-
      93xBMjZMU6HfGLXlwi1uYtQKL6eiNqOsWqkuFmlWMGJqKVBMF3+jb/dSrsRHrvpplIIDxU
      klwLNUzeZ5BTMjwQ=="
    }
  }
}
```

This *must also be in the git repository!*

2.6 Native Modules

These are modules that are performance critical and are therefore written in other languages such as c++.
The problem with this is that you will now need OS specific versions again!

2.7 nvm/nodenv

This is a tool to handle different node versions, you prob don't need this crap on arch, btw...

2.8 API versions

typically you will have 2 versions of a specific API, one that is synchronous, and one that is async.
The async will not throw an exception, as it is either handled with callbacks or with promises, the synchronous variant however will throw an exception if it fails.

```
// async
let fs = require('fs');
let path = "test.txt";
fs.readFile(path, function(err, content) {
  if (err) return console.error(err);
  console.log('content of file: %s\n', path);
  console.log(content.toString());
});
// the synchronous variant
// readFileSync

// note this specific problem can be done better with streams, as you might have continous input/ouput
let server = http.createServer(function (req, res) {
  let stream = fs.createReadStream(__dirname + '/data.txt');
  stream.pipe(res);
});
```

2.9 EventEmitter

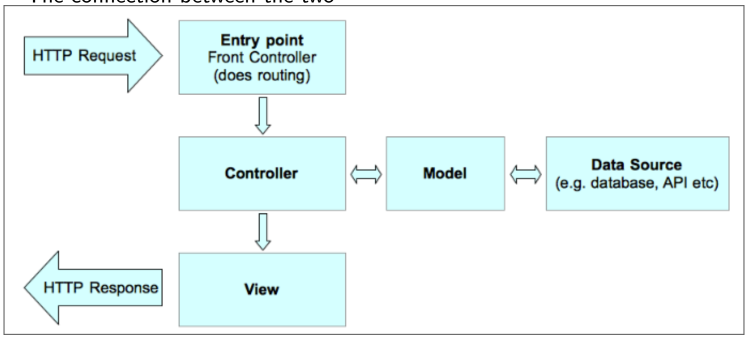
Similar to bevy, nodejs has an event emitter, which can then also be used in classes:

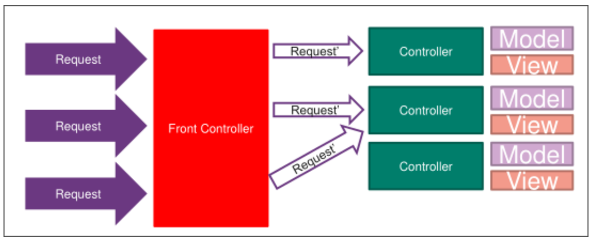
```
import {EventEmitter} from 'events'
export class Door extends EventEmitter {
  constructor() {
    super();
  }
  ring() {
    setTimeout(() => {
      this.emit('open');
    }, 1000);
  };
}
```

3 Web Framework Patterns

3.1 MVC

- Model
  Only responsible for the data
- View
  Only responsible for the presentation
- Controller
  The connection between the two





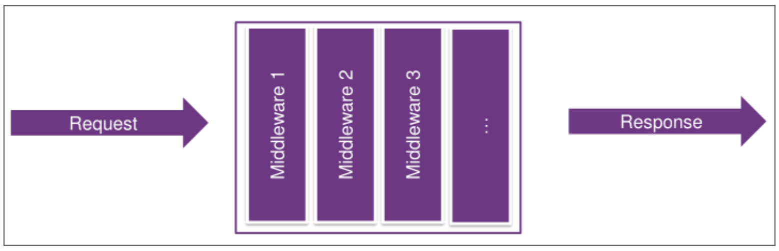
3.2 ExpressJS

- Most used framework
- Javascript
- rather old
- Integrated Middleware

Base Usage:

```
import http from 'http';
import express from 'express';

const app = express();
http.createServer(app).listen(3000);

// OR

import express from 'express';
const app = express();

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

3.3 Middleware



Modular components, that will handle individual tasks -> one for authentication, one for authorization.

3.3.1 Usage of Middlewares in ExpressJS

In order to register a middleware in expressJS, use the app.use(middleware) function.

```
import express from 'express';
import bodyParser from 'body-parser';

app.use(express.static(__dirname + '/public'));
app.use(bodyParser.urlencoded({ extended: false }));
app.use(router);

const app = express();
const router = express.Router();
```

3.3.2 Middleware archive "Connect"

- body-parser
- cookie-parser
- cookie-session
- errorhandler
- method-override
- serve-static

3.3.3 ExpressJS defaults

Middlewares:
- Router
- Static

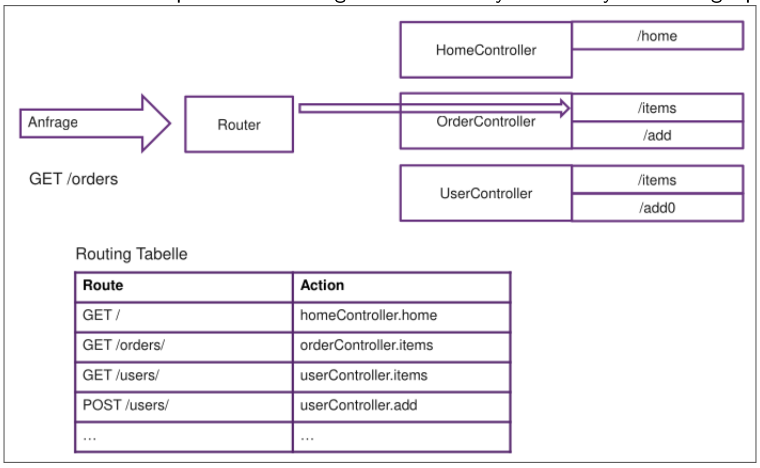
Request extensions:
- params
- is() -> checks if content-type is correct
- get() -> check header

Server-Side-Rendering

Response extensions:
- sendFile
- format
- json/jsonop

3.3.4 Routing

Routers handle requests with endings. This enables you to easily create single page application swith REST.



Route	Action
GET /	homeController.home
GET /orders/	orderController.items
GET /users/	userController.items
POST /users/	userController.add
...	...

Setup:

```
import express from 'express';
const router = express.Router();
```

Example for *all methods*:

```
router.all('/', function(req, res){
res.send('hello world');
}); // will be called without checking the method
```

Example with a specific method

```
router.get('/', function(req, res){
res.send('hello world');
}); // will only be called if it is a get method
```

Example with multiple defined methods

```
app.route('/book')
.get(function(req, res) {res.send('Get a random book');})
.post(function(req, res) {res.send('Add a book');})
// used to map multiple methods to one route
```

3.3.5 Route Specifics

Router uses pattern matching, this means that you can check for things such as:

- /ab*de -> c or something else implied
- /* map to anything

You can use variables to store requests from users:

```
router.get('/something/:id', function(req, res) ... // this set the variable id as req.params.id
// note that the req is passed to the function in question, meaning you can handle that variable in that function!
```

3.3.6 Satic-Middleware

In order to always deliver something like a png, you can use static.

```
app.use(express.static(__dirname + '/public'))
```

3.3.7 Custom Middleware

In order to create a middleware, you need to use 3 functions/parameters.
The next() has to be called, in order to end the request, otherwise the page will load indefinitely!
Reponse and Request are represented as the res and req in the function.

```
function myDummyLogger( options ) {
  options = options ? options : {};
  return function myInnerDummyLogger(req, res, next) {
    console.log(req.method +":"+ req.url);
    next();
  }
}
```

3.3.8 Error Middleware

This should always be registered as the last middleware -> makes sure it catches all errors.
You can however define multiple error middlewares, the last one just needs to end the request.
Will be called if an "error-object" is passed to the next() function inside another middleware.

```
app.use(function(err, req, res, next) {
  console.error(err.stack);
  res.status(500).send('Something broke!');
});
```

Note for forms, you can only use the GET and POST methods in forms, this means that something like delete won't work!
It will be converted to GET requests!
As a solution, *use the POST method as a workaround.*

3.4 Databases and Frameworks

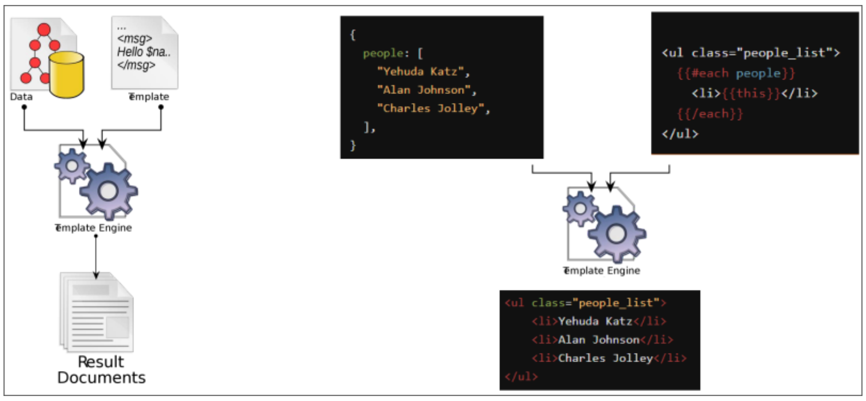
You can use any kind of databases like SQL -> Postgres, NoSQL, JSON, in memory databases etc.

3.4.1 Example with nedb (NoSQL)

```
import Datastore from '@seald-io/nedb';
const db = new Datastore({filename: './data/order.db', autoload: true});

// insert
db.insert(order, function(err, newDoc){
  if(callback){
    callback(err, newDoc);
  }
});

// find or findOne
db.findOne({ _id: id }, function (err, doc) {
  callback( err, doc);
});

// update
db.update({_id: id}, {$set: {"state": "DELETED"}}, {}, function (err, doc) {
  publicGet(id,callback);
});
```

3.4.2 Templating engine with DatabaseTemplating engine with Databasess

The question is, how do we generate html when we want to do it dynamically?
Meaning how do we interact with html and database? Templates!



```
{
  people: [
    "Yehuda Katz",
    "Alan Johnson",
    "Charles Jolley",
  ],
}
```

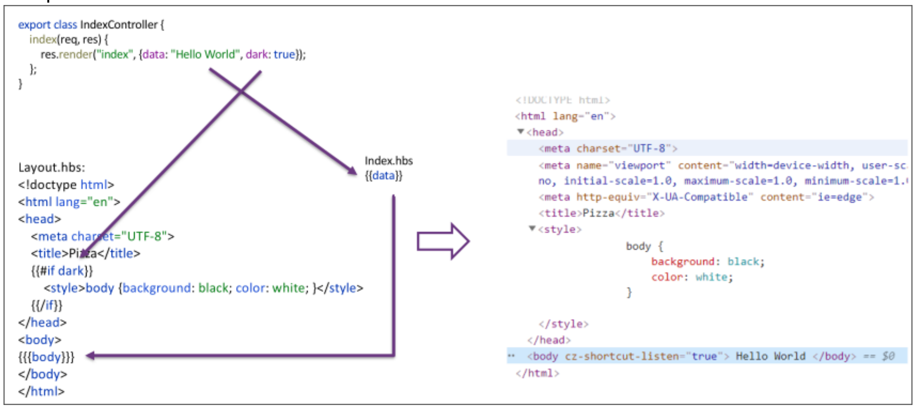
```
<ul class="people_list">
  {{#each people}}
    <li>{{this}}</li>
  {{/each}}
</ul>
```

```
<ul class="people_list">
    <li>Yehuda Katz</li>
    <li>Alan Johnson</li>
    <li>Charles Jolley</li>
</ul>
```

- definition of look -> handlebar framework language
- definition of tables -> json
- caching
- Partials
- converts html and json to the result document.

3.4.3 Handlebars template engine

Exmple for handlebars:



```
export class IndexController {
   index(req, res) {
      res.render("index", {data: "Hello World", dark: true});
   };
}
```

Layout.hbs:
```
<!doctype html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <title>Pizza</title>
   {{#if dark}}
      <style>body {background: black; color: white; }</style>
   {{/if}}
</head>
<body>
{{{body}}}
</body>
</html>
```

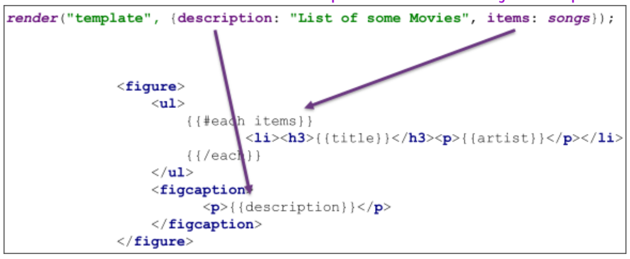
Index.hbs
```
{{data}}
```

```
<!DOCTYPE html>
<html lang="en">
▼<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, user-sc
    no, initial-scale=1.0, maximum-scale=1.0, minimum-scale=1.
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Pizza</title>
  ▼<style>
              body {
                  background: black;
                  color: white;
              }

    </style>
  </head>
··  <body cz-shortcut-listen="true"> Hello World </body> == $0
</html>
```

```
// 1. import express-handlebars
import exphbs from 'express-handlebars';

const app = express();

// 2. configure
const hbs = exphbs.create({
  extname: '.hbs',
  defaultLayout: "default",
  helpers: {
    ...helpers
  }
});

// 3. set engine and global values
app.engine('hbs', hbs.engine);
app.set('view engine', 'hbs');

// 4. path to views
app.set('views', path.resolve('views'));

// usage afterwards
// app.render(view, [locals], callback)
createPizza = async (req, res) => {
  res.render("succeeded", await orderStore.add(req.body.name, "unkown"));
};
```

```
<p>Order-Informationen</p>
{{#if pizzaName}}
  <p>Bestellte Pizza: {{pizzaName}}</p>
  {{#if_eq state "OK"}}
    <form action='/orders/{{_id}}' method='post'><input type='hidden' name='_method' value='delete'><input
      type='submit' value='Delete order'></form>
  {{/if_eq}}
{{/if}}

// render
async showOrder(req, res) {
  res.render("showorder", await orderStore.get(req.params.id));
};
```

You can also send data to the template from within javascript



```
render("template", {description: "List of some Movies", items: songs});

        <figure>
            <ul>
                {{#each items}}
                    <li><h3>{{title}}</h3><p>{{artist}}</p></li>
                {{/each}}
            </ul>
            <figcaption>
                   <p>{{description}}</p>
            </figcaption>
        </figure>
```

In loops, the current object is the current context:

```
<ul>
  {{#each items}}
  <li>
    <h3>{{title}}</h3>
    <p>{{this.artist}}</p>
  </li>
  {{/each}}
</ul>
// you can access the root with {{@root}}
```

Handlebars only defines a few helpers (functions), but you can define some yourself!

```
//Usage: {{? hasError 'error' 'ok'}}
Handlebars.registerHelper('?', function(exp, value1, value2, options) {
  if(exp) {
    return value1;
  }
    return value2;
  });
{{? hasError 'FEHLER' 'OK' }}
```