«Moin»

an event-driven microservice application server, written in Node.js

Torben Hartmann

Abstract

«In computing, microservices are small, independent processes that communicate with each other to form complex applications which utilize language-agnostic APIs.»(Fowler 2014) The goal of this thesis is to implement and document a microservice application server, written in Node.JS. The resulting software will be advertised and published on the online platform GitHub as well as on the Node.JS package repository NPM.

Contents

1 Introduction

1.1 Motivation

While developing a framework, some of the desired concepts to follow are a high modularity and an easy and "comfortable" interface for the end-user. Because of the low restrictions on coding style the programming language Javascript is well suited to achieve this goal. It allows loading code on-the-fly and offers a way to structure data in objects which are not based on a static model. Furthermore can be run on almost any device (Browser, Server[1], Desktop[2], Mobile[3] and even on Microcontrollers[4]). One of the limitations which slow down the development process is, that despite the language can load code dynamically, it has no functionality to unload or reload the code without restarting the application. Without this limitation and with a few other enhancements to the runtime environment, javascript could be a good choice for writing Microservices. "Moin" is an attempt to create such an environment.

1.2 The Name "Moin"

> Moin is a German greeting meaning "hello" and in some places "goodbye".

One aim of the thesis is to create a framework which is published on the Node Package Manager(NPM). A so called *package* on NPM has to have a unique name which is used to install the package from the command line.

With over 200,000 packages the chances, that another project has already taken the desired name is quit high. The following names could not be used, because they were

[1]with Node.js `https://nodejs.org/en/`
[2]with Electron `http://electron.atom.io/`
[3]with Ionic `http://ionicframework.com/`
[4]with Espruino `http://www.espruino.com/`

already registered:

- “service” or “microservice” (because of the Microservice architecture the project is based on)
- “events” (because of the extended EventEmitter)
- “say”, “shout” or “whisper” (as a metaphor for the communication between the services)

Since it was nearly impossible to find a name which is bound to the functionality of the project, the german word “Moin” was chosen, as it is easy to write for the english speaking community.

1.3 Objective

The Objective of this work is to create an application which:

- Is modular in a way that every functionality can be extended.
- Has an asynchronous event-system which has a good performance and the ability to filter events by more than just a name.
- Implements a Microservice architecture where the services can utilize the NPM repository in order to be installed with the NPM command line tools.
- Has the ability to reload the services “on the fly” without restarting the whole application.
- Should trigger the reloading, when the javascript file is changed, easing the way the developer can work with the system.

The resulting software will be published on the NPM Repository and Github.com. Furthermore, a simple website for the documentation will be created to serve as an API documentation.

1.4 Structure of the Thesis

Technical Background In this chapter some of the terms used in this thesis are being explained.

Architecture and Implementation In this chapter the architecture, design decisions and the considerations made in the process are shown.

Advertising & Publishing In this chapter the process of publishing and the generation of an online documentation is being presented.

Conclusion In the last chapter the project is reviewed and possible future development is considered.

Methods and Events

Methods and events are documented in the following format:

Method: **methodName**(*Type* **arg1**= *defaultValue*, *Type* **arg2**)

parameters ***Type*** **arg1** = ***defaultValue*** an argument

Type **arg2** another argument

return value returnType

description description of the method

Event: eventName⇒(*Type* **arg1**)

parameters ***Type*** **arg1** an argument

description description of the event

2 Technical Background

2.1 Node.js

Node.js is a runtime environment for writing server-side javascript applications. It enforces an event driven architecture with asynchronous I/O. This allows the process to be non-blocking while waiting for an I/O operation to finish. The javascript code is being interpreted by Google's V8 Engine[1].

2.2 Yeoman Generator

The *yeoman generator*[2] is a framework which simplifies the process of creating wizards. These so called generators can create static files and directories, download content from the web or generate files out of information, which they asked the user for. Until now, over 4,000 generators[3] exist for any kind of project. Despite the fact, that the generators are written on the Node.js platform, they do not have to generate code for Javascript projects. There are also generators which setup a whole PHP Blog-system[4].

2.3 Node Package Manager (NPM)

The Node Package Manager (or in short just NPM), is a package repository and management tool for Node.js. The tool is distributed alongside the Node.js package. It offers a command line app, which lets you download and publish packages on the

Figure 2.1: The NPM Logo

[1]`https://developers.google.com/v8/`
[2]`http://yeoman.io/`
[3]`http://yeoman.io/generators/`
[4]`https://github.com/wesleytodd/YeoPress`

online platform `http://npmjs.com`. Considering the easy installation of packages and the amount of over 320.000 published packages[5], one can say that NPM has a huge influence on the popularity of Node.js.

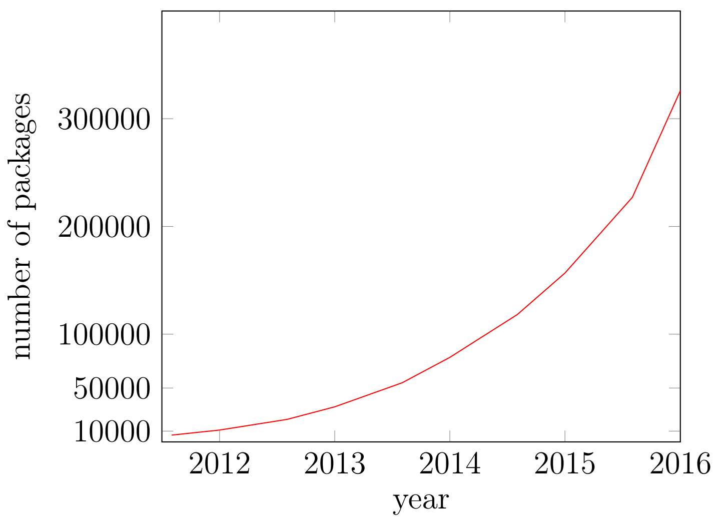


Figure 2.2: Number of NPM packages over time

package.json

In order to turn ones script into a valid package, it is necessary to create a *package.json*[6] file.

[5]`https://www.npmjs.com/` ⇒ "total packages"
[6]`https://docs.npmjs.com/files/package.json`

property	description
name	The name of the package. Must be lowercase and can only contain URL-save[7] characters
version	The version of the package. This is important in order to publish updated versions of the package. The version has to be given in the *Semantic Versioning*[8] format.
description	A short description which gets displayed right below the name and in the search results on `npmjs.com`
license	Defines the license[9] under which the package is distributed.
bin	Is used to link a script as a global command. The property has to be an object, where the key is the name of the command to be registered and the value the relative path to the Javascript file.
dependencies	An object that defines which packages the package depends on (including their version). They get automatically installed when the *npm install* command is executed at the root of the package.
author	The author of the package

Table 2.1: The most important properties of a package.json file

It is possible to extend the contents of the *package.json* file with custom properties.

2.4 Event-Driven Architecture

An Event Driven Architecture is a pattern, which handles the generation and reaction of so called events. An event is a "significant change in state"(Chandy 2006). The common event most UI programmers have encountered is a click on a button. When clicking it, the button's state changes from *not_ clicked* to *clicked.* Event Handlers which registered for the click event will be notified and can react to the event.

[7]Only alphanumerics [0-9a-zA-Z], the special characters `$-_.+!*'(),` and reserved characters used for their reserved purposes may be used unencoded within a URL.(T. Berners-Lee 1994)
[8]`http://semver.org/`
[9]`https://spdx.org/licenses/`

Event Emitter Produces or emits an event notification. It has no knowledge if a consumer exists or has received the notification.

Event Consumer Listens to or consumes an event.

Event Channel Sends the event notifications, which were emitted by the Event Emitter to the previously registered Event Consumers.

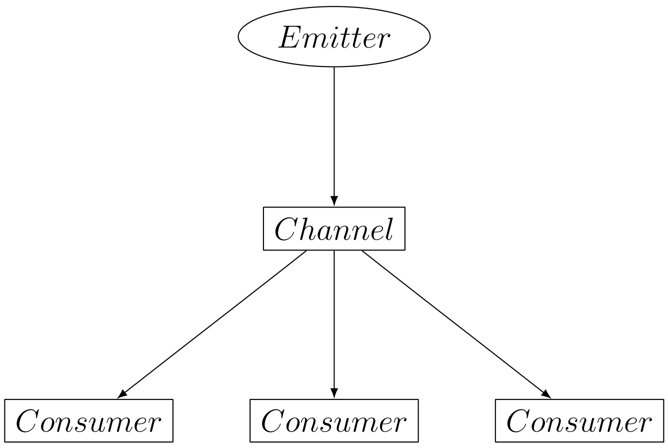


Figure 2.3: A simple event system

An event driven architecture encourages loose coupling of the components of the application. The emitter or producer of an event typically does not know wether there is a listener or consumer, which has registered for the specific event. In most cases, when a response is required by the emitter, the handler itself emits an event, which gets consumed by the original emitter. E.g. **A** emits an event. **B** consumes it and emits an event as a response, which gets consumed by **A**.

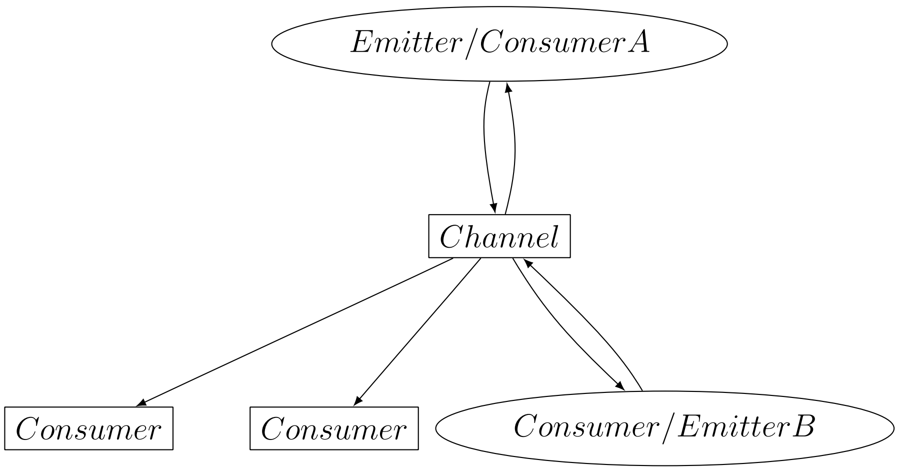


Figure 2.4: Responses in event driven systems

Implementation in Node.js

As of the asynchronous concept of Node.js, it makes heavy use of events. A basic form of an event emitter is part of the core modules. It is often used in network related or child_process modules in Node.

The `EventEmitter`:

- Runs events synchronously. The `emit` call is finished, after every handler has run.
- Runs events in the order they have been added. Multiple emits lead to the same outcome.
- Distinguishes events from one another by an event name. Examples for common event names are “connect”, “data” or “error”.

```javascript
const EventEmitter = require("events");
class TestEmitter extends EventEmitter {}

const testEmitter = new TestEmitter();
testEmitter.on("event", (A, B) => {
    console.log("an event occurred!");
    console.log("A:", A, "B:", B);
});
testEmitter.emit("event", 1, 2);
```

Code Listing 2.1: A simple example of the node event emitter

2.5 Hot Swapping

"Hot Swapping" is a term which stands for replacing a computer component without shutting down the system. The most known technology which includes "Hot Swapping" is USB. In the field of software development it describes the ability to change program code, without the need to restart the program.

When a system supports "Hot Swapping", it makes it possible to reload one part of the application, without interfering other parts. For example: Imagine a system with two components where the first one receives E-Mails and the second serves webpages. The Web-component can be reloaded independently without interfering the receival of E-Mails.

2.6 ECMA Script 6

ECMA Script 6 is the latest implemented standard of the Javascript programming language. It adds a few syntactic enhancements to the language which are also used in the code base of Moin.

Block Scoped Variables

The keyword *let* creates a new variable (like the keyword *var*) which is only accessible in the current block and its child-blocks.

```
//"Old" style
var tmp = "123";
if (true) {
    var tmp = "567";
}
console.log(tmp); //567
//----------No double declaration
let tmp = "123";
if (true) {
    let tmp = "567"; //ERROR tmp allready declared
}
//----------block-scoped
if (true) {
    let tmp = "567";
}
console.log(tmp); //ERROR tmp is not defined
```

Code Listing 2.2: Block scoped variables / the *let* keyword

Arrow Functions

Arrow functions are a shorthand syntax for anonymous functions.

```javascript
var isEven = function(num) {
    return num % 2 == 0;
};
var isEven = num => num % 2 == 0;

var add = function(a, b) {
    return a + b;
};
var add = (a, b) => a + b;

var log = function(val) {
    val = "Hello, " + val;
    console.log(val);
};
var log = val => {
    val = "Hello, " + val;
    console.log(val);
};
```

Code Listing 2.3: Arrow Functions

Additionally the *this* and *arguments* keywords are not redefined when using arrow functions. Therefore there is no need for the *var that = this;* anti-pattern.

Function Parameters

Default values and the ability to collect multiple parameters into one variable have been added.

```
//old
function list(name) {
    var entries = Array.prototype.slice.call(arguments, 1);
    return "The List " + name + " has " +
    entries.length + " entries";
}
//new
function list(name, ...entries) {
    return "The List " + name + " has " +
    entries.length + " entries";
}
list("Students", "Friedemann", "Moritz", "Torben");
//old
function say(text, name) {
    if (name == undefined) name = "Unknown";
    console.log(name + ": " + text);
}
//new
function say(text, name = "Unknown") {
    console.log(name + ": " + text);
}
```

Code Listing 2.4: Parameters

Enhanced Object Properties

A shorthand syntax for function-properties and for keys with the same name as local variables have been added.

```
let first_name = "John";
let last_name = "Doe";
//old
var obj = {
    first_name: first_name,
    last_name: last_name
};
//new
var obj = {
    first_name,
    last_name
};
//old
var foo = {
    bar: function() {
        return 5;
    }
};
//new
var foo = {
    bar() {
        return 5;
    }
};
```

Code Listing 2.5: Enhanced Object Properties

Destructuring

With destructuring one can assign multiple local variables with values from an array or an object.

```
var test = [1, 2, 3];
var [a, b, c] = test; //a=1; b=2; c=3;
[a, b] = [b, a]; //swap values

var test = {
    first_name: "John",
    last_name: "Doe"
};
var {first_name, last_name} = test;
var {
    first_name: fname,
    last_name: lname
} = test; //lname="John"
```

Code Listing 2.6: Destructuring

Classes

A new class-like syntax for the prototype based programming has been added.

```
class test {
    constructor(name) {
        this._name = name;
    }
    getName() {
        return this._name;
    }
}
class test2 extends test {
    constructor(name) {
        super(name);
    }
}
```

Code Listing 2.7: Classes & Inheritance

Template String

This new way of defining strings enables variables to be added in place instead of being concatenated.

```
var elements = [1, 2, 3, 4];

var text = "There are " + elements.length + " elements\n" +
    "(" + elements.join(",") + ")";
var text = `There are ${elements.length} elements
(${elements.join(",")})`;
```

Code Listing 2.8: Template String

2.7 Promises

In order to explain what a promise is and why it is needed, one has to look at the traditional way of writing asynchronous programs in Node.js.

```javascript
const fs = require("fs");
try {
    fs.readFile("input.csv", (err, result) => {
        if (err) {
            throw err;
        } else {
            doCalculations(result, (err, result) => {
                if (err) {
                    throw err;
                } else {
                    fs.writeFile("output.csv", result,
                    function(err) {
                        if (err) {
                            throw err;
                        } else {
                            console.log("Done!");
                        }
                    });
                }
            });
        }
    });
} catch (e) {
    console.error(e);
}
```

Code Listing 2.9: Example for nested callbacks

In this example a file is being read, some calculations are being made and the results are being written back to disk. Because the three called functions are all working asynchronous, we have to nest the calls in the previous callback function. This leads to code which soon becomes unreadable. Furthermore, if the order of the two calls should be switched, a big part of the code has to be rearranged.

```
const fs = require("fs-promise"); //Promise version of fs
fs.readFile("input.csv")
    .then(doCalculations)
    .then((result) => fs.writeFile("output.csv", result))
    .then(() => console.log("Done!"))
    .catch(e => console.error(e));
```

Code Listing 2.10: The same example, with promises

Promises, as a language feature, were introduced in ECMASript 6. A Promise represents an operation that has not been completed yet and gives access to its return value in a proxy-like way. Promises prevent the "habit" of nesting callbacks. This makes it possible to write asynchronous code in a more synchronous fashion. A Promise is in one of three states

pending The initial state, when the Promise is created.

fulfilled The operations has been performed successfully.

rejected The operation failed.

A pending Promise can either become fulfilled with a value (by calling the resolve function) or rejected with a reason (by calling the reject function or by throwing an error). After the Promise has switched to either state (which can only occur once per Promise), the *then* method of it is called.

2.8 Microservices

"In computing, microservices are small, independent processes that communicate with each other to form complex applications which utilize language-agnostic APIs." (Fowler 2014)

Microservices are a modern interpretation of the term "Service Oriented Architectures". The goal is to wrap small parts of the program, which do one particular job, into a so called service. These services communicate with each other over a (network-) protocol.

In a system which uses a Microservice architecture the following principles apply.

- The services can easily be replaced, even when the other services are still running. This reduces the downtime of the application.
- The code, which is composited in a service should only be responsible for one specific task.
- The system should easily be spread across multiple servers and provide the ability to use different programming languages for the service implementation.
- The system is symmetrically structured. The communication between the services are made through an event-like system.

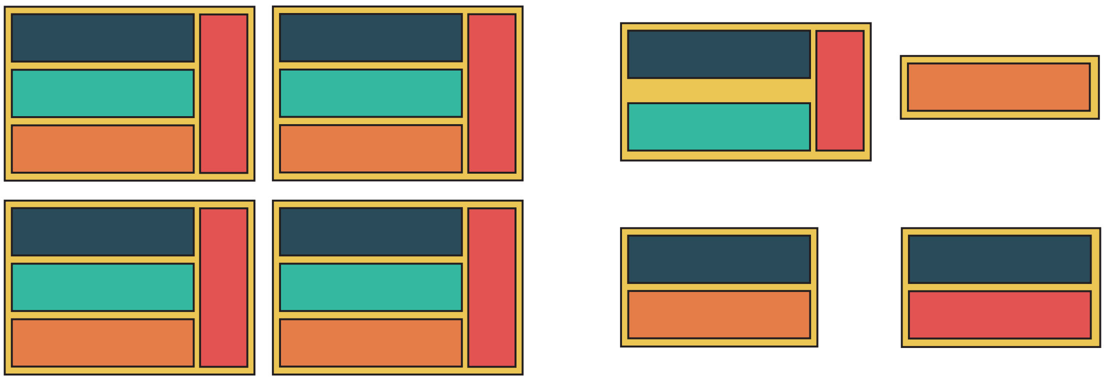

(a) Monolithic (b) Microservice

Figure 2.5: Monolithic system vs. Microservices

Each color in the figures 2.5 represents a function of a software system. The first figure shows a monolithic architecture. This term defines an architecture where the parts are interwoven with each other. In case of a high load of the system, the whole application is duplicated and run on another server. The microservice architecture, as shown in the second figure, consists of small individual services which do not depend stongly on each other. The single services can be moved or duplicated independently to another server without the need to do the same with the other services.

Current Implementation

Currently only one major implementation of the Microservice architecture seems to exist in the NPM repository: The package Seneca[10] is a project by Richard Rodger, which was first released in 2010. Seneca and Moin differ in the following cases:

Seneca	Moin
Event System	
Executes the best-matching[11]handler.	Has the ability to only execute the best-matching[7] handler or all handlers, which match the event.
Transportation	
Has 15 transportation modules.	Has one transportation module which works over socket.io.
Persistence	
Has 13 different persistence modules.	Currently only has the ability to read configuration, but not to save data back in any way.
Services	
Services are written as independent programs. The configuration is done inside the service. Changing the code needs a manual restart of the program.	Services do not have to follow any pattern. The configuration is done from within the Moin-node, and the API of Moin is exposed in a global Moin-object to the service. A change of the code leads to an automatic reload of the service.

Table 2.2: Comparison of Seneca and Moin

Due to the modularity of the Moin-system, it has the potential to be competing to the Seneca project.

[10]`http://senecajs.org/`

[11]The best-matching handler is the handler out of all matching handlers with the most filter properties

3 Architecture & Implementation

3.1 Application Structure

The Program is extendable in two ways:

Modules Modules are loaded at startup time. They add functionality, which are accessible by other modules and services. They can not be reloaded while the program is running.

Services Services are loaded after all modules are processed. They can dynamically be loaded, unloaded and run in a sandbox-like environment

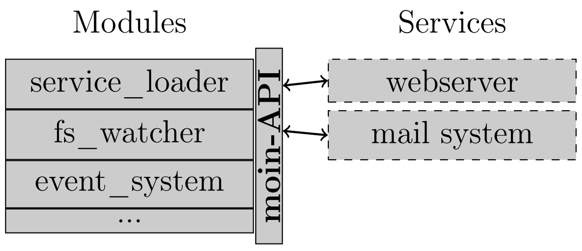


Figure 3.1: Modules, API and Services

3.1.1 Utilizing the package.json file

In order to tell the system that a folder holds a service or a module, the *package.json* file is extended with a new property called "moin".

To the time of this thesis it has 3 sub-properties:

property	description
moin.type	Can either be “service” or “module”. If this property is not present, the package is not recognized by the system and will be ignored.
moin.moduleDependencies	Only used for `MoinModules`. An array with the `MoinModule`-names, which have to be loaded before this module is loaded.
moin.settings	An object which holds default values for the conponent’s configuration. If the key *active* is not defined by the settings objects, it is set to the value *true*.

Table 3.1: Additional properties of the package.json file

The component folder also has to include an *index.js* file. This file is loaded by the system.

3.1.2 Settings

The settings defined in a module’s *moin.settings* are collected in a file called *config.json.* This file holds all configuration variables of the loaded modules. They are saved as an object, where the keys are the names of the modules and the value their corresponding settings from their *package.json* file.

For modules and components an *active* key is added. It defines if the component should be loaded or not. Three objects are merged into another to form the final settings.

$$\begin{tikzcd}
\begin{array}{c} Active-object \\ \{active : true\} \end{array} \arrow[dr] & \begin{array}{c} Default-settings \\ \{level : \text{debug}\} \end{array} \arrow[d] & \begin{array}{c} Config-object \\ \{level : \text{info}\} \end{array} \arrow[dl] \\
& \begin{array}{c} Results \\ \{active : true, level : \text{info}\} \end{array} &
\end{tikzcd}$$

Figure 3.2: Merging of settings

3.1.3 Moin command

The application can be run with the *moin* command. By default, it scans the *node_ module* folder for modules and services and loads them by default. The settings of every found Moin-module are written to the *config.json* file.

3.1.4 Modules

A module has to export a function with one parameter and return nothing. The parameter contains the Moin API.

```
module.exports = function(moin, settings){
  //Module code goes here
};
```

Code Listing 3.1: The minimal module code

3.1.5 Services

Contrary to the module a service does not need to export anything. The Moin-API object is accessible as a global variable inside the service code.

```
console.log("Hello, World!");
```

Code Listing 3.2: The minimal service code

3.2 Core

The core of Moin currently consists only of about 450 lines of code. Most of the functionality is added by modules, which makes the overall system very adaptable. The core exposes the following API:

Method: **joinPath**(*String* **. . . segments**)

parameters ***String*** **. . . segments** The path segments

return value String

description Concatenates the path the application was started in, with the given path segments.

Method: **load**(*String* **path**)

parameters ***String*** **path** Path of the folder of the service/module

return value null / `MoinComponent`

description Returns a Promise which gets fullfilled either with an instance of a `MoinComponent` or with null (in the case that the folder did not hold a valid module or service)

Method: **getLogger**(*String* **name**)

parameters ***String*** **name** The name for the logger (is printed in the logs)

return value Logger

description Returns a new instance of the internal Logger. The logger has an identical API like the console object (despite adding an additional method called "info") but adds a name, the current date and time and coloring to the output.

Method: **on**(*String* **event**, *Function* **function**)

parameters ***String*** **event** The event-name to filter

Function **function** The handler function

return value *nothing*

description ⇒see PromiseEventEmitter (subsection 3.2.1)

Method: **emit**(*String* **event**, *Any* **. . . arguments**)

parameters ***String*** **event** The event to emit

Any **. . . arguments** Arguments which are passed to the handlers

return value Promise

description ⇒see PromiseEventEmitter (subsection 3.2.1)

Method: **registerMethod**(*String* **methodName**, *Function* **fnc**, *Bool* **after**= *true*)

parameters ***String*** **methodName** The name of the method

Function **fnc** Handler of the method

Bool **after** = ***true*** If the variable is true, the handler is added at the end of the chain (which is the default case). If it is false, the handler is added at the start of the chain.

return value *nothing*

description ⇒see API extension (subsection 3.2.2)

3.2.1 PromiseEventEmitter

The PromiseEventEmitter has a similar API as the original EventEmitter but works asynchronous. Therefore, the *emit* function returns a promise which gets fulfilled as soon as all handlers have handled the event. The listener function can either return nothing or a Promise, if it works asynchronous. The listeners are chained, meaning that each handler gets called, when its predecessor finished its execution. If all handlers should be processed in parallel, the *emitParallel* function can be used. The *this* object inside the handler is an instance of the Moin API.

3.2.2 API extension

The core of Moin has not much functionality by itself. Therefore modules can add additional methods via the *registerMethod* function. The registered methods are called as if they are part of the *moin* api-object. When a method is registered more than once by different modules, the calls are getting chained. The *this* object inside the method is changed to the following object:

Method: **getLastValue**( )

parameters *none*

return value Any

description Returns the last value a function has returned in the chain. If the function is the first which is called, the value is *undefined*.

Method: **setArguments**(*Any* . . . **arguments**)

parameters ***Any* . . . arguments** The new Arguments

return value *nothing*

description Changes the Arguments for the following functions in the chain.

Method: **stopPropagation**( )

parameters *none*

return value *nothing*

description Breaks the chain at this point. No further function is called.

Method: **getAPI**( )

parameters *none*

return value *nothing*

description Returns the Moin API

This enables modules to override or decorate functionalities which were added by other modules.

```javascript
let order1 = {
    price: 80,
    country: "DE"
};
let order2 = {
    price: 30,
    country: "DE"
};
let order3 = {
    price: 30,
    country: "NL"
};

//default method. 4 euro delivery cost
moin.registerMethod("getDeliveryCost", (package) => 4);
//add 6 euro for orders outside of germany
moin.registerMethod("getDeliveryCost", function(package) {
    //get the previos delivery cost(4 euro)
    let last = this.getLastValue();
    if (package.country != "DE") last += 6;
    return last;
});
//is added before the 2 already defined functions.
//Free shipping for price>80 euro
moin.registerMethod("getDeliveryCost", function(package) {
    if (package.price > 80) {
        this.stopPropagation();
        return 0;
    }
}, true);
```

Code Listing 3.3: Example of the API extension mechanism

3.2.3 Events

The following events are emitted by the core.

Event: init⇒( )

parameters *none*

description Is emitted when all modules are loaded.

Event: exit⇒( )

parameters *none*

description Is emitted when the application has received a *SIGINT* signal.

3.2.4 Settings

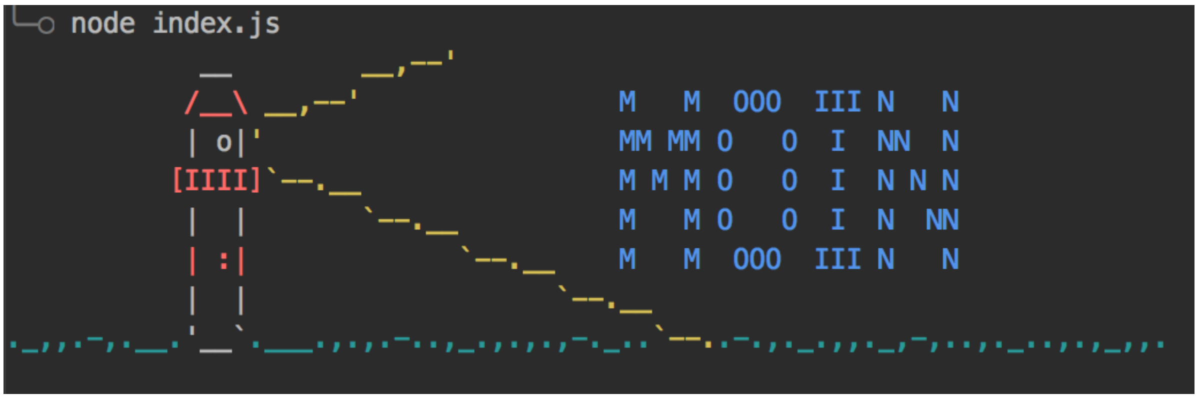
key	value
moin.modulePaths	Array that contains folders, where the system should search for `MoinModules`
logging.level	Minimum level of log-messages to display. A level includes every level which has a higher number: **0 | "debug"** *console.log* or *logger.debug* **1 | "info"** *console.info* or *logger.info* **2 | "warning"** *console.warning* or *logger.warning* **3 | "error"** *console.error* or *logger.error*
logging.disabled	Array with logger-names which should not be printed.

Table 3.2: Settings of the Moin-core

3.3 Modules

3.3.1 Logo

This Module adds a startup CLI graphic to Moin.



```
└o node index.js
                      __,--'
            __   __,--'
           /__\                           M   M  OOO  III N   N
           | o|'                          MM MM O   O  I  NN  N
          [IIII]`--.__                    M M M O   O  I  N N N
           |  |       `--.__              M   M O   O  I  N  NN
           | :|             `--.__        M   M  OOO  III N   N
           |  |                   `--.__
'_,,'‾,'__'`__`'___',','‾'',_',',',‾'_''  `--.'‾','_',,'_,‾,'','_'','',_,,'
```

Figure 3.3: The Moin startup graphic

3.3.2 Service Loader

The Service Loader adds the ability of loading and unloading services to the system. In the process of loading the service, the code is wrapped inside a template code and saved as a temporary file inside a *.moin* folder. This file is loaded instead of the original file.

```
/*
  The following argumentlist is not complete.
  It gets filled by the Moin-modules.
*/
module.exports = function({__errorHandler,moin,setInterval}){
  try {
    //Service code goes here
  } catch (e) {
    __errorHandler(e);
  }
};
```

Code Listing 3.4: The code which wraps the service code

Problems

Timer

In order to unload a Javascript module every active timer-callback has to be unloaded as well. Because it cannot be assumed, that the developer thinks of this circumstance, it is important that the system tries to unload all registered timers by itself. In order to do so, the *setInterval, setImmediate* and *setTimeout* functions as well as their *clear* counterparts get overwritten by the module. Every timer ID is saved per service and gets automatically cleared, when the service is unloaded.

Loading of Submodules

There are two ways services can be made available to the system:

They can be installed from the NPM repository. In this case, the dependencies defined in the *package.json* file have to be accessible by the service.

They can be implemented and put into a dedicated folder. In this case, the Node-modules which are installed at the root application should be accessible by the services.

To achieve this, the location for the temporary file differs for the two cases.

1. When the service is inside a *node_modules* folder, the *.moin* folder is created as a sub-folder of the services.
2. Otherwise the *.moin* folder is created inside the root of the node application.

Compilation- and other Errors

An error in a service should not lead to an error, which shuts down the whole system. Therefore, the whole service code and every timer-callback is wrapped inside a try-catch-block. With this method most errors are being caught.

API

Method: **loadService**(*String*|*`MoinService`* **service**)

parameters ***String*|*`MoinService`*** **service** A path to a service or a `MoinService`

return value Promise

description Prepares the temporary service file, decorates the service with an API and interchanged globals, saves it to disk and loads it. The service API is being build with an event, which can be listened to by other modules. The returned Promise is resolved with an unique serviceID (String) or rejected with an error message.

Method: **getTemp**(*String* **subFolder**)

parameters ***String*** **subFolder** Optional

return value String

description Returns the path to the temporary folder of the Moin-application. The optional subFolder argument is concatenated if present.

Event: beforeServiceLoad⇒(*MoinService* **service**, *Function* **cancel**)

parameters ***MoinService*** **service** The service to be loaded.

Function **cancel** Used to cancel the loading process.

description Is emitted before the service is loaded. If the `cancel` method is called, the service will not be loaded.

Event: loadService⇒(*Object* **handler**)

parameters ***Object*** **handler** See below.

description Is used to let modules decorate the API for a service. Globals and API methods can be added via the handler object.

Method: **handler.getTemp**(*String* **subFolder**)

parameters ***String*** **subFolder** Optional

return value String

description Returns the path to the temporary folder of the service. The optional subFolder argument is concatenated if present.

Method: **handler.getId**( )

parameters *none*

return value String

description Returns the unique id of the service.

Method: **handler.getService**( )

parameters *none*

return value `MoinService`

description Returns the service itself.

Method: **handler.registerGlobal**(*String* **name**, *Any* **data**)

parameters ***String*** **name** Name of the global variable to overwrite.

Any **data** The data which should be linked to the global.

return value *nothing*

description Registers a new global variable for the service. With this method it is possible to overwrite functions such as *setInterval*.

Method: **handler.addApi**(*String* **name**, *Any* **data**)

parameters ***String* name** Name of the global

***Any* data** The data, which should be linked to the global.

return value *nothing*

description Registers a new api variable for the service. The added variables are accessible via the *moin* global object (e.g.*moin.<name>*).

Method: **handler.getApi**( )

parameters *none*

return value Object

description Returns the Moin-service-API.

Method: **handler.registerErrorHandler**(*Function* **fnc**)

parameters ***Function* fnc** The handler function

return value *nothing*

description Adds an error handler. The handler is being called (with the error message as a parameter) when an error occurs in the service code.

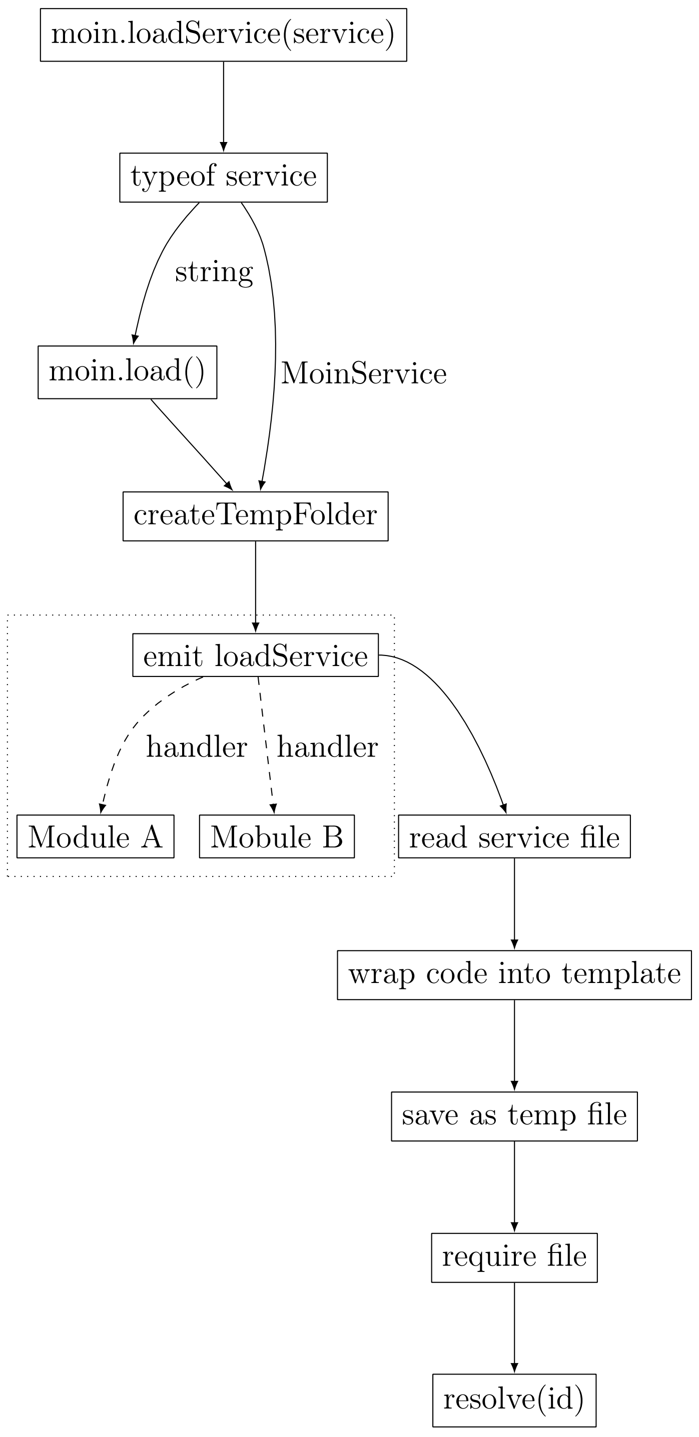


Figure 3.4: Loading of a service

Method: **unloadService**(*String* **serviceId**)

parameters ***String* serviceId** ID of the service, which should be unloaded.

return value Promise

description Unloads a service.

Event: unloadService⇒(*String* **serviceId**)

parameters ***String* serviceId** Id of the service, which should be unloaded.

description Is emitted when the *unloadService* method was called.

Method: **unloadAllServices**( )

parameters *none*

return value Promise

description Unloads all loaded service.

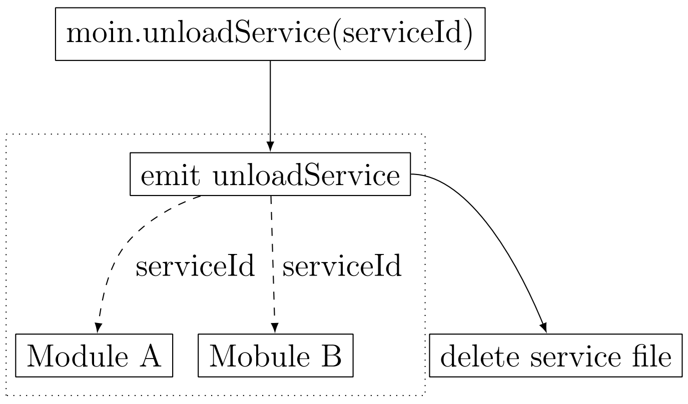


Figure 3.5: Unloading of a service

3.3.3 Event System

The event system in Moin differs by some aspects from the original Node `EventEmitter`:

Asynchronicity Contrary to the original EventEmitter, the emit call is non-blocking. Instead it returns a Promise, which is resolved as soon as every handler has run.

Timeouts The emit method can take a timeout (in ms) as an optional parameter. After the timeout is reached, the emit Promise is resolved with the results of the handlers which have finished processing in time. It is important to note that reaching a timeout does not terminate the handler function.

Return Values The *emit* call returns a Promise. The fulfillment-value contains stats on how many handlers were interested in the event, how many returned a value, how many threw an error and how many timed out. Additionally it holds two arrays with the thrown errors and the returned values.

Filterable The Moin event system can not only filter events by a name. Instead filtering by multiple properties as well as dynamic checks are supported.

Filter

The filter is a one dimensional Javascript object. The key defines the property which is checked. The corresponding value has to match the one defined in the event. The value is casted into a string for value comparison. If the value is a function, it is used as a dynamic filter. Dynamic filter functions receive the value of the field as a parameter and should return true, when their condition is fulfilled.

```
let event1 = {x: 1, y: 2};
let event2 = {x: 5, y: 6};

//matches event1
let test1 = {x: 1};
//matches event2
let test2 = {x: 5};
//matches event1 and event2
let test3 = {
    x: (x) => x >= 1 && x <= 5
};
//matches nothing
let test4 = {x: 1, y: 6};
//matches event2
let test5 = {
    y: (y) => y > 4
}
```

Code Listing 3.5: Example for filter rules and matches

Data Structure

Handlers are saved in a special structure to ensure fast retrieval of fitting handlers for an event.

```
let _fieldFilter = { //contains all filtered fields and handlers
  "x":{//a field called x
    //holds handler wich are filtered by this fields
    _handler:["test1","test2","test3","test4"],
    //static checks
    _static:{
      //test1 and test2 filter x = "1"
      "1":["test1","test4"],
      //test2 filters x = "5"
      "5":["test2"]
    },
    //dynamic checks
    _dynamic:{
      //test 3 filters 1 ≤ x ≤ 5
      "test3":(x) => x>=1 && x<=5
    },
    //count of dynamic checks
    _dynamicCount:0
  },
  "y":{//a field called y
    //holds handler wich are filtered by this fields
    _handler:["test4","test5"],
    //static checks
    _static:{
      //test4 filters y = "6"
      "6":["test4"]
    },
    //dynamic checks
    _dynamic:{
      //test5 filters y > 4
      "test5":(y)=>y > 4
    },
    //count of dynamic checks
    _dynamicCount:1
  }
};
```

Code Listing 3.6: Structure of event handlers

The main goal is to process the fields, which most handlers have set a filter rule on first. The more handlers are registered for a field, the more can potentially be eliminated as a candidate for the event.

Therefore, each field is given a score. It consists of the number of handlers and a penalty for dynamic values, which take a longer time to calculate. The fields are then sorted by their score in descending order. The field with the highest score is processed first.

$$\text{score} = \text{\_handler.length} - \text{\_dynamicCount} * 0.5 \tag{3.1}$$

Algorithm

The algorithm can either return a list of all handlers which match the event or the best matching handler. The best matching handler matches the event and has the most filter-rules defined. This adds the possibility to implement a handler which handles a general case and implement all special cases as individual handlers, which are filtering for their special case.

The two parameters for the function are: **event** which holds the event and **bestMatch** which indicates if multiple handlers or just one handler and its score should be returned. The following steps are performed:

1. Copy a list of all event handlers to *handlers*
2. If the bestMatch option is set, initialize a Map *count* with the *handlers* as keys and 0 as value.
3. For every property a handler is filtering on
    a) Initialize *toCheck* with every handler which is filtering the property and is in the *handlers* list.
    b) When *toCheck* is empty, go on with the next property
    c) Remove every dynamic check, which returns true from *toCheck*
        i. If the bestMatch option is set, increment the value of the passing dynamic checks by one
    d) Remove every value check, which equals the event value from *toCheck*

i. If the bestMatch option is set, increment the value of the passing static checks by one

e) *toCheck* holds all handlers, which do not meet the requirements.

f) Remove all handler from *handlers*, which are in *toCheck*

4. If the bestMatch option is set
   a) If handlers has no entries return *(id:=null, score:=-1)*
   b) Search the largest value from *count*
   c) Return an object *(id:=key, score:=value)*
5. Else
   a) Return *handlers* as an array

Example

1. The event *{x:1, y:2}* is being emitted
2. It is assumed that every handler is interested in the event. They are being copied into *handlers*.
3. The field x is being checked. The handlers *test1, test2, test3* and *test4* are filtering the x-field and have to be checked.
4. The dynamic filter *test3* $(x >= 1 \wedge x <= 5)$ matches and is removed from toCheck.
5. The static filter *test1* and *test4* (1) are matching and are being removed from toCheck.
6. *test2*'s filter does not match the x-filter and is being removed from handlers.
7. The field y is being checked. The handlers *test4* and *test5* are filtering the y-field and have to be checked.
8. No dynamic filter matches. *toCheck* is not modified.
9. No static filter matches. *toCheck* is not modified.

10. Both *test4* and *test5* do not match the *y*-filter and are being removed from handlers.

11. The two handlers *test1* and *test3* are being returned.

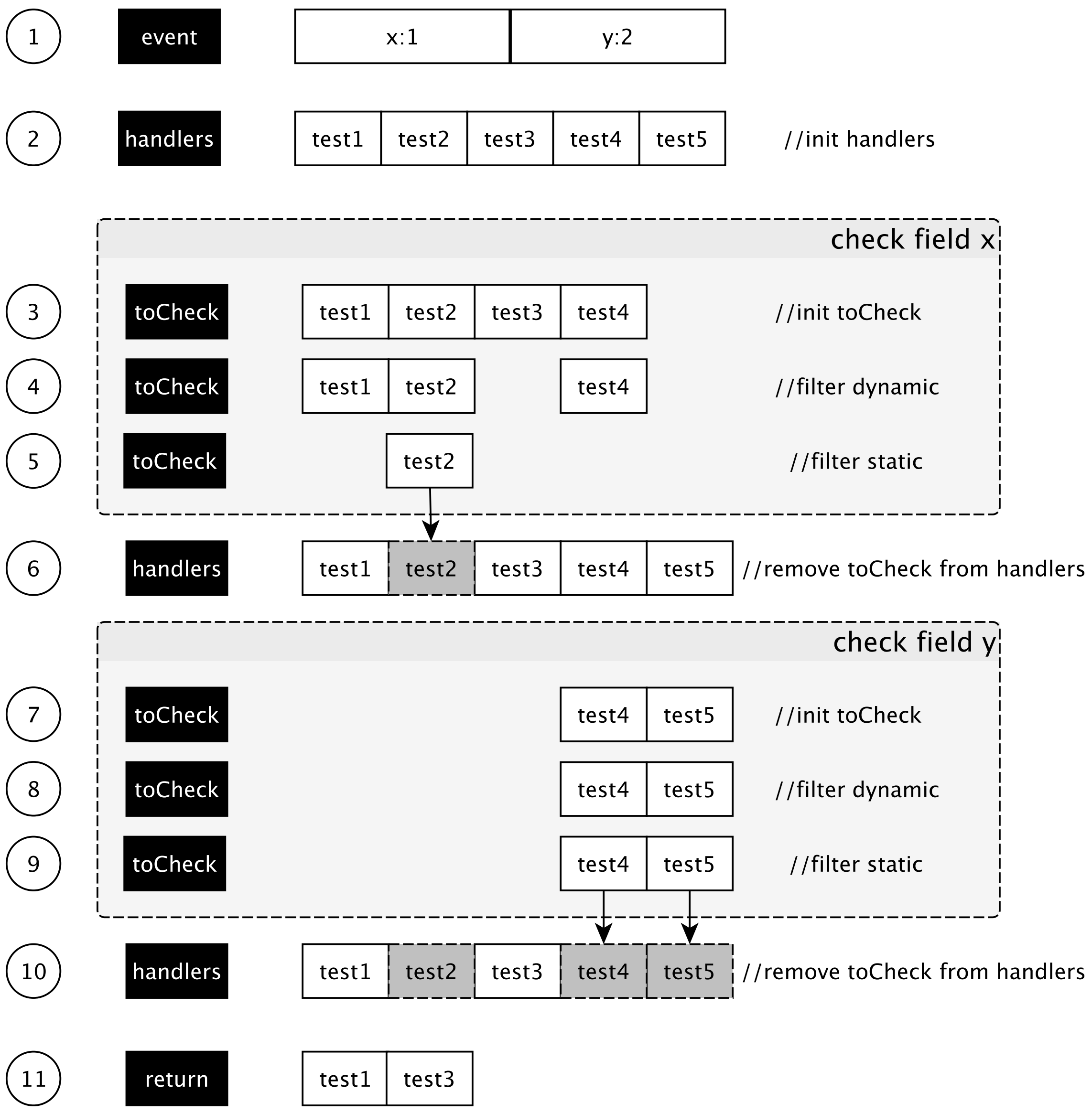


Figure 3.6: Event filter algorithm

Speed of Set Implementations

The bottleneck[1] in the algorithm are the two lists *toCheck* and *handlers*. They have to support inserts and deletes as well as checks, if a given value is present. To find the best implementation of a `Set` a speed comparison between three solution was done.

1. A simple unsorted `Array`.
2. An `Object` where the values are being added as keys (with a value of *true*)
3. the new `Set` implementation, which was introduced by ECMAScript 6

For every implementation the following operations were measured:

1. Insertion of an unique element (*add*)
2. Deletion of an element (*delete*)
3. Searching for an element which is not part of the set (*find(false)*)
4. Searching for an element which is part of the set (*find(true)*)

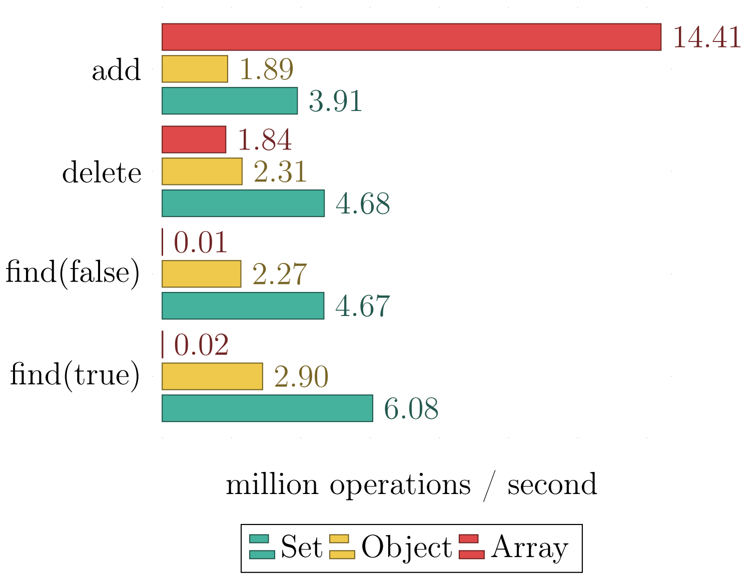


Figure 3.7: Benchmark test with 10000 inserts, lookups and deletions

The `Array` is the fastest when it comes to adding of elements. In every other aspect the `Set` solution surpasses the other methods.

[1]Bottleneck: component, which limits the performance of the whole system.

Collect & Execute

In the Moin system, the process of determining the fitting listeners and executing them is named "Collect and Execute". The collect function is the function defined in Section 3.3.3 wrapped in a promise. The execute function receives a handler id and should return *null* if the system has no knowledge of the handler or, in the other case, the corresponding callback function.

This enables a flexible way to extend the event handling, as other modules can add their own event system. This is utilized by the *Remote Event Dispatcher (Section 3.3.6)* which sends events over a network connection to other Moin instances.

Method: **registerEventHandler**(*Fnc(event,bestMatch)* **collect**, *Fnc(id)* **exec**)

parameters ***Fnc(event,bestMatch)*** **collect** Should return matching handlers (as of Section 3.3.3) as a value or Promise.

Fnc(id) **exec** Should return a function, when the id is a valid handler or *null* otherwise.

return value Integer

description Registers an additional event handler. Returns an id, which is used to unregister the handler once it is not needed any more.

Method: **removeEventHandler**(*Integer* **id**)

parameters ***Integer*** **id** The id of the handler.

return value *nothing*

description Removes a previously defined event handler.

Method: **collectEventHandlerIds**(*Object* **event**, *bool* **bestMatch**= *false*)

parameters ***Object* event** The event

bool* bestMatch = *false Return only the best match or all matching handlers.

return value Array|Object

description Runs the internal collect functions (see Section 3.3.3) and returns

1. The concatenated handlers from each collect function (bestMatch=false)
2. The handler with the highest score out of the returned candidates (bestMatch=true)

Method: **execEventHandlerById**(*String* **id**, *Object* **event**)

parameters ***String* id** The id of the handler

***Object* event** The event

return value Promise

description Executes a local event handler callback. Returns an object with a *state* and either a *value* or an *error*. *state* is an enum-integer with the following meaning:

-1 Timeout

0 Rejected (*error* is filled with the error-message)

1 Resolved (*value* is filled with the return value of the function)

Service API

For the added Service API functionality see Section 3.4.1.

3.3.4 Filesystem Watcher

To ease the process of writing an application, the dynamic loading and unloading of the services should be automatically executed, when the script is changed. The *Filesystem Watcher* module adds an API function, which monitors a folder for new or deleted subfolders. New folders (this includes every subfolder, which is in the monitored folder, when the module is loaded) are tested, if they hold a valid `MoinService`. If this is the case, the service is handed over to the *Service Loader*, which handles the initialization of the service. Additionally the *index.js* and *package.json* files are being monitored for changes. If a change in one of these two files is detected, the Service is reloaded. When the service folder is removed from disk or renamed, the service is unloaded.

Chokidar

As of its own Readme[2], chokidar is "A neat wrapper around node.js fs.watch / fs.watchFile / fsevents.". The module provides a cross-platform, unified way of watching for filesystem events. The documentation of the NPM-module can be found at `https://www.npmjs.com/package/chokidar`

API

Method: **addServiceFolder**(*String* **path**)

parameters ***String* path** The path to be watched.

return value *nothing*

description Starts watching a folder for services.

Settings

key	value
serviceFolders	Array with folders, which should automatically be watched.

Table 3.3: Settings of the *Filesystem Watcher*

[2] `https://www.npmjs.com/package/chokidar`

3.3.5 Configuration

The Configuration module adds settings for services to Moin. The default settings are defined in the same way as the settings for `MoinModules` (inside their *package.json* file under the key *moin.settings*, see ⇒ Section 3.1.1). Opposite to the module settings, the service settings are not collected in one file. A JSON file, with the name "<name>.json" (where name is the name defined in the *package.json* file) is created inside a config folder ("config.d" by default). In case of a change of the file's contents, an event is emitted. This allows the *Filesystem Watcher* to reload the module, if its settings were changed. The module also checks for the `active` flag inside the settings object. When the value is `false` the service will not be loaded.

Events

Event: serviceChanged⇒(*String* **id**)

parameters ***String*** **id** Id of the Service

description Is emitted, when a settings file was changed.

Settings

key	value
configFolder	Folder, where the settings for the services should be saved.

Table 3.4: Settings of the *Configuration* module

3.3.6 Remote Event Dispatcher

The Remote Event Dispatcher enables a connection between Moin instances. With this connection, event handlers and emitters do not have to be on the same host.

Communication

The communication is done by *Socket.io*[3] which is an event-driven client-server library on top of the Websocket technology. The library,

- adds automatic reconnection.
- handles the transfer of javascript objects.
- adds an event system, where a response can be send back to the emitter.

In order to connect multiple Moin systems, there has to be one server and an arbitrary number of clients, which all connect to the server.

Flowchart

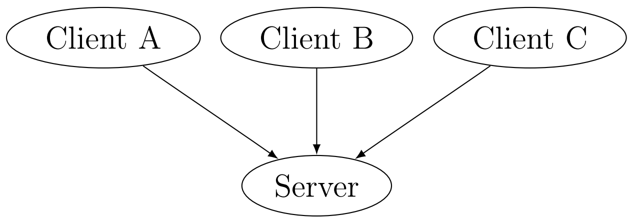


Figure 3.8: One server with 3 clients

In this scenario, 3 clients are connected to a server.

[3]`http://socket.io`

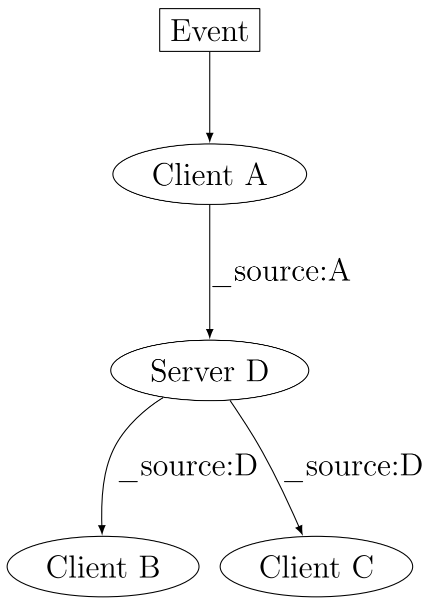


Figure 3.9: Collect phase

Client **A** sends an event to the server. The server sends the event to **B** and **C**. The *source* field is added to the event to indicate over which socket the event was send. This ensures that an event is not send back to the socket it came from.

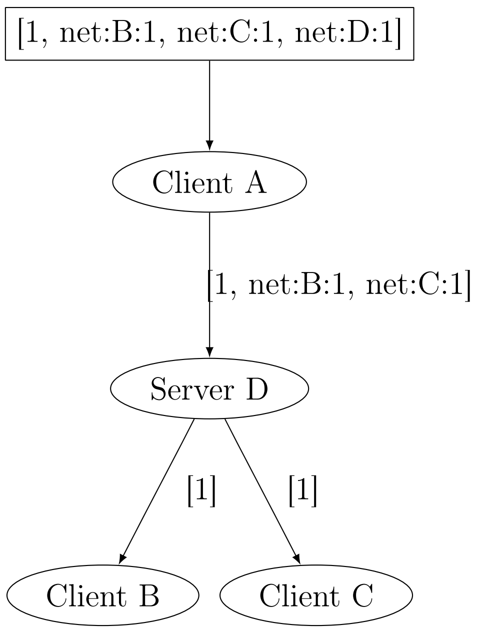


Figure 3.10: Execute phase

In this case, every client and the server itself have a registered listener for the event (in this case the local id of the handler is 1 to increase readability). This leads to following id list: *1 (**A**'s handler), net:B:1, net:C:1, net:D:1.* The ids which belong to a remote instance are preceded by "net:<InstanceId>:<handlerId>". **A** sends every request beginning with "net:" to the server **D**. The id "net:D:1" is unwrapped, because the id belongs to the server itself. The server then sends the unwrapped ids to **B** and **C**.

Settings

key	value
mode	Can be one of the following: **client** Connect to a server. **server** Listen on a port. **dynamic** Do nothing automatically.
host	Only needed when *mode=client*. Domain or IP the server is listening on.
port	Port to connect to or to listen on.
id	Id of the socket. Optional, if not set it is assigned to a new unique id each run.
token	A string, which is used for authentication. Has to be the same on all instances.

Table 3.5: Settings of the *Remote event Dispatcher*

3.4 Services

3.4.1 API

Service Loader

Method: **moin.registerUnloadHandler**(*Function* **fnc**)

parameters ***Function* fnc** Function which is executed.

return value *nothing*

description Registers a function which is called, when the service is unloaded. This is the desired way of closing open IO handles.

The following globals are defined by the Module:

__servicename Holds the name of the service, which was defined in its *package.json* file.

console Is overwritten by a MoinLogger instance. *console.log* is bound to the *Logger.debug* function.

(set|clear)(Timeout|Immediate|Interval) The timer functions are wrapped inside a try catch block and their ids are saved. Every active timer is automatically stopped, when the service is unloaded.

Event Emitter

Method: **moin.on**(*Object* **filter**, *Function (event)* **callback**)

parameters ***Object* filter** The filter for the handler.

***Function (event)* callback** Function, which is called when the event occurs.

return value *nothing*

description Listens for an event. The callback function can return a scalar value or a promise.

Method: **moin.emit**(*String* **eventName**, *Object* **data**, *Integer* **timeout**= *null*)

parameters ***String*** **eventName** Name of the event. Is saved inside the event-data object under the “event”-key.

Object **data** Additional data for the event.

Integer **timeout** = ***null*** Timeout in ms, after which the results should be returned, even when not every handler has finished its execution.

return value Promise $\rightarrow \{values, errors, stats\}$

description Emits an event. The return object consists of the following keys:

values Array with the return values of all resolved handlers.

error Array with the error messages of all rejected handlers.

stats.handler Total handlers which got executed by the event.

stats.rejected Number of rejected handlers.

stats.resolved Number of resolved handlers.

stats.timeout Number of handlers which timed out.

Method: **moin.act**(*String* **eventName**, *Object* **data**, *Integer* **timeout**= *null*)

parameters ***String*** **eventName** Name of the event. Is saved inside the event-data object under the “event”-key.

Object **data** Additional data for the event.

Integer **timeout** = ***null*** Timeout in ms. When the timeout is reached before the handler has finished its execution, a “Timeout” error is returned.

return value Promise $\rightarrow \{value\} \vee \{error\}$

description Emits an event, but only executes the fitting handler, which has the most filter-properties defined. The return object can either have one of the two keys:

value The return value if the handler was resolved.

error The error messages if the handler was rejected.

Config

Method: **moin.getSettings( )**

parameters *none*

return value Object

description Returns the merged config for the service.

3.4.2 Example

This example consists of two services:

httpServer Creates an HTTP-server and sends out an event as soon as a request is made. Listens on HTTP-events for the url */services* which should return a list of all services.

httpRoute Listens on HTTP-events and returns "Hello, World!" if the url */hello* is being requested. Also listens on the url */services*.

```
{
  "name": "httpRoute",
  "moin": {
    "type": "service"
  }
}
```

Code Listing 3.7: *package.json* file of the httpRoute service

```javascript
moin.on({
    event: "http",
    url:"/hello"
}, (event)=> {
    return "Hello, World!";
});
moin.on({event: "services"}, (event)=> __servicename);
```

Code Listing 3.8: *index.js* file of the httpRoute service

```json
{
  "name": "httpServer",
  "moin": {
    "type": "service",
    "settings":{
      "port":8080
    }
  }
}
```

Code Listing 3.9: *package.json* file of the httpServer service

```javascript
const http = require('http');
const PORT=moin.getSettings().port;
function handleRequest(request, response){
  let event={
    url:request.url,
    method:request.method
  };
  moin.act("http",event).then(({error,value})=>{
    if(error){
      console.error(`[${error.code}] URL: ${event.url} `+
          `METHOD: ${event.method}`);
      response.writeHead(error.code);
      response.end(error.message);
    }else{
      response.writeHead(200);
      console.info(`[200] URL: ${event.url} `+
        `METHOD: ${event.method}`);
      response.end(value);
    }
  });
}
moin.on({
    event: "http"
}, (event)=> {
    throw {"code":404,message:"Not found"};
});
moin.on({event: "services"}, (event)=> __servicename);
moin.on({
    event: "http",
    url:"/services"
}, (event)=> {
  console.log("collecting service names...")
  return moin.emit("services")
    .then(({values})=>values.join(", "));
});
var server = http.createServer(handleRequest);
server.listen(PORT, function(){
    console.log("Server listening on: http://localhost:"+PORT);
});
moin.registerUnloadHandler(()=>server.close());
```

Code Listing 3.10: *index.js* file of the httpServer service

httpServer/package.json:6 Defines the setting “port” with a default value of *8080*.

httpServer/index.js:2 Reads the “port” setting.

httpServer/index.js:8 On an incoming request an *http*-event is emitted. Since *moin.act* is used, only the best-matching handler is called.

httpServer/index.js:9-14 When an error has occurred, the error message is send to the browser and an error is logged.

httpServer/index.js:14-19 When a value was returned, it is send to the browser and an information message is logged.

httpServer/index.js:22-26 The default handler for the *http*-event. Throws a “Not Found” error. Is only called when there is no handler which could serve the request.

httpServer/index.js:27 Returns its name on an *service*-event.

httpServer/index.js:28-35 Handles the *http*-event with the url “/services”. Emits a *services* event and returns the joined list of values.

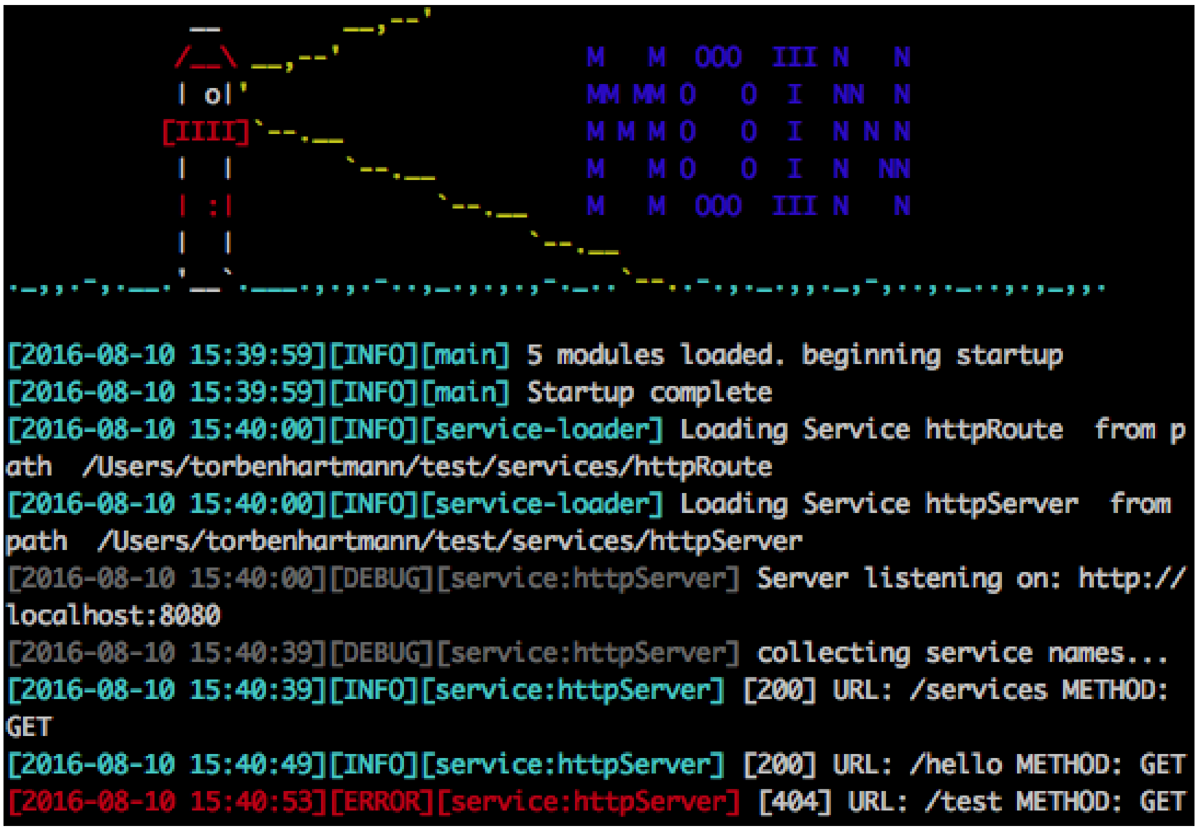
httpServer/index.js:36-39 Opens the http server.

httpServer/index.js:40 Registers an unload handler, which closes the http port, when the service is unloaded.

httpRoute/index.js:1-6 Handles the *http*-event with the url “/hello”. Returns the string “Hello, World!”.

httpRoute/index.js:7 Returns its name on an *service*-event.

To test this example, the Moin application was started and the urls */services* (returned “httpRoute, httpServer”), */hello* (returned “Hello, World!”) and */test* (returned “Not Found”) have been requested.



```
M   M  OOO  III N   N
MM MM O   O  I  NN  N
M M M O   O  I  N N N
M   M O   O  I  N  NN
M   M  OOO  III N   N

[2016-08-10 15:39:59][INFO][main] 5 modules loaded. beginning startup
[2016-08-10 15:39:59][INFO][main] Startup complete
[2016-08-10 15:40:00][INFO][service-loader] Loading Service httpRoute  from p
ath  /Users/torbenhartmann/test/services/httpRoute
[2016-08-10 15:40:00][INFO][service-loader] Loading Service httpServer  from
path  /Users/torbenhartmann/test/services/httpServer
[2016-08-10 15:40:00][DEBUG][service:httpServer] Server listening on: http://
localhost:8080
[2016-08-10 15:40:39][DEBUG][service:httpServer] collecting service names...
[2016-08-10 15:40:39][INFO][service:httpServer] [200] URL: /services METHOD:
GET
[2016-08-10 15:40:49][INFO][service:httpServer] [200] URL: /hello METHOD: GET
[2016-08-10 15:40:53][ERROR][service:httpServer] [404] URL: /test METHOD: GET
```

Figure 3.11: The example output of the application

4 Advertising & Publishing

4.1 Yeoman-Generator

To make the start for the programmer slightly easier a yeoman generator was build, to help with configuring the Moin-application and creating new services.

The generator can be installed using *npm install -g yo generator-moin*.

Usage

The configuration is started with the command *yo moin*. An interactive wizard is shown, which offers the following options:

- Should Moin look for services in other folders than *node_modules*? If yes it creates the directory.
- Should Moin look for modules in other folders than *node_modules*? If yes it creates the directory.
- Should Moin connect to or listen for other Moin instances? If yes, it asks for the host and port.

After finishing, the generator creates the necessary folders and generates a *config.json* file with the desired values.

To generate a new service the command *yo moin:service* can be used. A similar wizard as in the configuration process is shown:

- In which service folder should the service be created?
- How should the service be called?
- Which kind of bootstrap-code is desired?

Basic Example Just a *console.log* call and an *unloadHandler*

Event Example Some event calls as well as an *unloadHandler*

The generator creates a new subfolder in the desired location and puts an *package.json* file and an *index.js* file into it. If the Moin application is running when the generator is started, the newly created service is loaded automatically.

4.2 NPM

For every Module as well as for the core, a package was released in the NPM-repository.

Core `https://npmjs.com/packages/moin`

Service Loader `https://npmjs.com/packages/moin-service-loader`

Event System `https://npmjs.com/packages/moin-event-system`

Filesystem-Watcher `https://npmjs.com/packages/moin-fs-watcher`

Remote Event Dispatcher `https://npmjs.com/packages/moin-remote-dispatcher`

Configuration `https://npmjs.com/packages/moin-service-settings`

Every module which is documented in this thesis is a dependency of the *moin*-package. This means that they are being installed, when the *moin* package is installed.

To install Moin the command "*npm install -g moin*" can be used (assuming that Node.js is installed on the system).

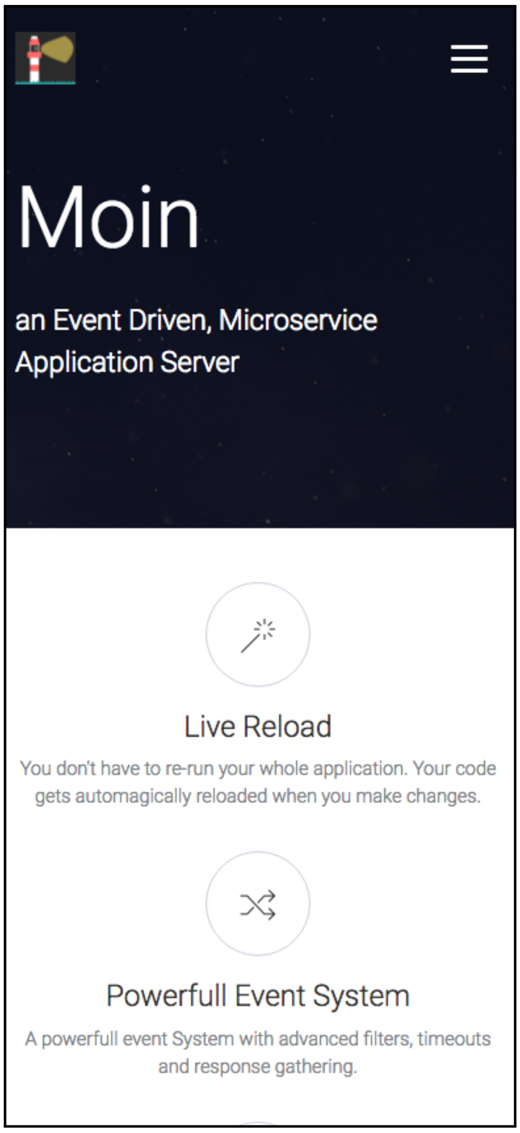
4.3 GitHub & GitHub-Pages

The source code of Moin was released at `http://github.com` under the MIT license. As a consequence of the number of packages and their repositories a GitHub organization was created, which holds the repositories as well as the github.io website.

The sources can be accessed at `https://github.com/moinjs`.

GitHub-Pages

In order to provide access to a documentation for the programmers, a website was created at `http://moinjs.github.io/`. The used template was bought and is not part of this work. The HTML template was rewritten, to work with Jekyll[1], the template engine used by GitHub.io.



Moin

an Event Driven, Microservice Application Server

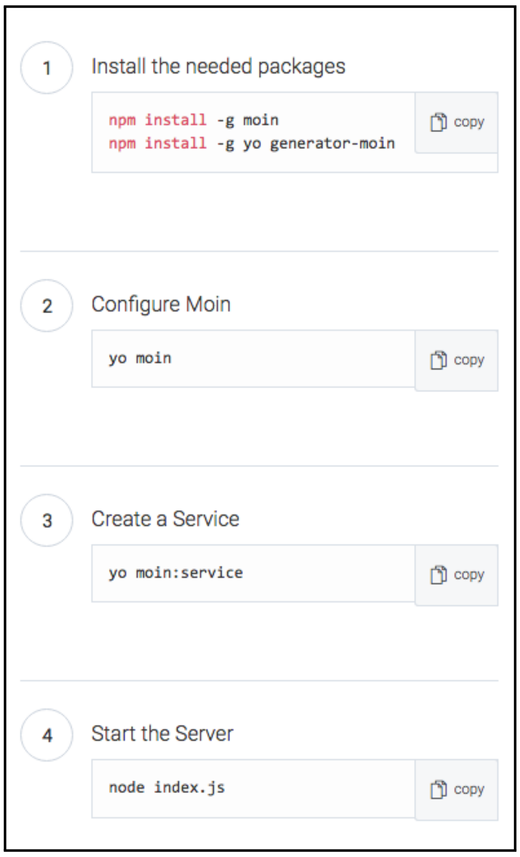
Live Reload

You don't have to re-run your whole application. Your code gets automagically reloaded when you make changes.

Powerfull Event System

A powerfull event System with advanced filters, timeouts and response gathering.

(a) Landingpage



1 Install the needed packages

```
npm install -g moin
npm install -g yo generator-moin
```

2 Configure Moin

```
yo moin
```

3 Create a Service

```
yo moin:service
```

4 Start the Server

```
node index.js
```

(b) Documentation

Figure 4.1: Screenshots of `http://moinjs.github.io/` (mobile version)

[1]https://jekyllrb.com/

5 Conclusions

The aim of this thesis was to write an application which serves as a runtime environment for Microservice systems, written in Javascript. A prototype has been developed, which has all the features mentioned in the objectives section. It was then published to the NPM repository and an API documentation was created in form of a website.

5.1 Problems

- The application works as expected, but was only tested with a few test cases. It is to assume that when it is used in a production environment, bugs could occur.
- Since the application is not as “old” as its rivaling project Seneca and misses Seneca’s number of additional modules, the success of the Moin project will be uncertain.
- The whole architecture is build in a very modular fashion. This leads to a high extendability but has the disadvantage to be hard to debug.

5.2 Planned Features

Event caching When an event is emitted in a regular interval, the list of handlers could be cached to accelerate the event process.

Homepage & Documentation The online documentation was made at an early state of development. It should be extended to fit the current API.

Module: Persistence The services can load config values, but do not have the ability to save data. Therefore a module which offers an interface to a local database could be created.

Remote Dispatcher Multi-server-systems often use a messaging queue (e.g. RabbitMQ) to distribute messages to each node. Adding another connection provider than Socket.io would ease the integration into existing architectures.

Services Some services could be published to NPM:

1. A mail sending and receiving service.
2. A web server with an authentication component.

List of Tables

List of Figures

Code Listings

References

T. Berners-Lee L. Masinter, M. McCahill (1994). *RFC1738: Uniform Resource Locators.* URL: `http://www.rfc-editor.org/rfc/rfc1738.txt`.

Chandy, K. Mani (2006). *Event-Driven Applications: Costs, Benefits and Design Approaches.* URL: `http://docplayer.net/15630715-Event-driven-applications-costs-benefits-and-design-approaches-gartner-application-integration-and-web-services-summit-2006.html`.

Fowler, Martin (2014). *Microservices: a definition of this new architectural term.* URL: `http://martinfowler.com/articles/microservices.html`.