

---

**Example Vue: Spotitube**

## Contents

<b>1</b>	<b>Preface</b>	<b>2</b>
<b>2</b>	<b>Prerequisites</b>	<b>2</b>
<b>3</b>	<b>How to create, run and lint a project</b>	<b>3</b>
<b>4</b>	<b>The start of a Vue application</b>	<b>6</b>
<b>5</b>	<b>The anatomy of a Vue Component</b>	<b>8</b>
5.1	Templates (View) . . . . .	8
5.2	Behaviour (Model: data and Controller: methods) . . . . .	10
<b>6</b>	<b>How to unit-test components</b>	<b>14</b>
<b>7</b>	<b>How to exchange data between components</b>	<b>17</b>
7.1	Between a parent and child component: properties . . . . .	17
7.2	Between non-related components: using a simple event-bus . . . . .	20
7.3	Between non-related components: using a shared store component . . . . .	21
7.4	How to communicate with REST services using Axios . . . . .	22
7.5	Axios supports all HTTP verbs by providing seperate functions (get, post, delete, put, etc.) but you can also use a generic request and tell Axios what verb to use. . . . .	22
7.6	Conclusion . . . . .	23

## 1 Preface

In this example we'll use a simple case study called "Spotitube" (a mashup between Spotify and YouTube to enable playlists with songs and videos) to demonstrate and explain some features of Vue.

## 2 Prerequisites

To use this example you need to install the following tools:

- Node.js ( $\geq 4.x$ , 6.x preferred)
- Npm (minimum version  $\geq 3.0$ )
- Git
- A JavaScript IDE (WebStorm is free to use with a student license from JetBrains)

- Vue.js devtools extension for Chrome, a great tool to inspect events and the structure and state of components.

### 3 How to create, run and lint a project

The easiest way to start a new Vue project is to use Vue CLI. When you've installed Vue CLI, you have the following options:

- `vue init webpack my-project`
- `vue init webpack-simple my-project`
- `vue init browserify my-project`
- `vue init browserify-simple my-project`
- `vue init pwa my-project`
- `vue init simple my-project`

---

For a serious Vue application we need at least unit-testing and the browserify template is the least complex option. Vue init starts a simple dialog, fill in the required fields and answer “Y” when asked for “Use ESLint to lint your code?” and “Setup unit tests with Karma + Jasmine?”.

---

You can use `npm run lint` to see if the JavaScript-code conforms to the ECMAScript-standard. Tools like WebStorm can also lint your code with ESLint or use SonarLint with the same profile so you can check if your code passed quality checks.

After creating the project enter the project-directory and run `npm install` followed by `npm run dev`.

---

The first command downloads all required libraries, including Karma and PhantomJS to run end-to-end and unit tests. Later we will replace Karma by Petrol because of its simplicity and because it works great with vue-inject and other vue extensions.

The second command actually runs the application in development mode. When you run the application in Chrome with the devtools extension installed and open the JavaScript console the screen looks like this:

---

The screen shows the demo application provided with the template. During this example we're going to rebuild the Spotitube application in Vue. This application is built in Angular by Meron Brouwer, for this example we use a design that looks like the Angular version:

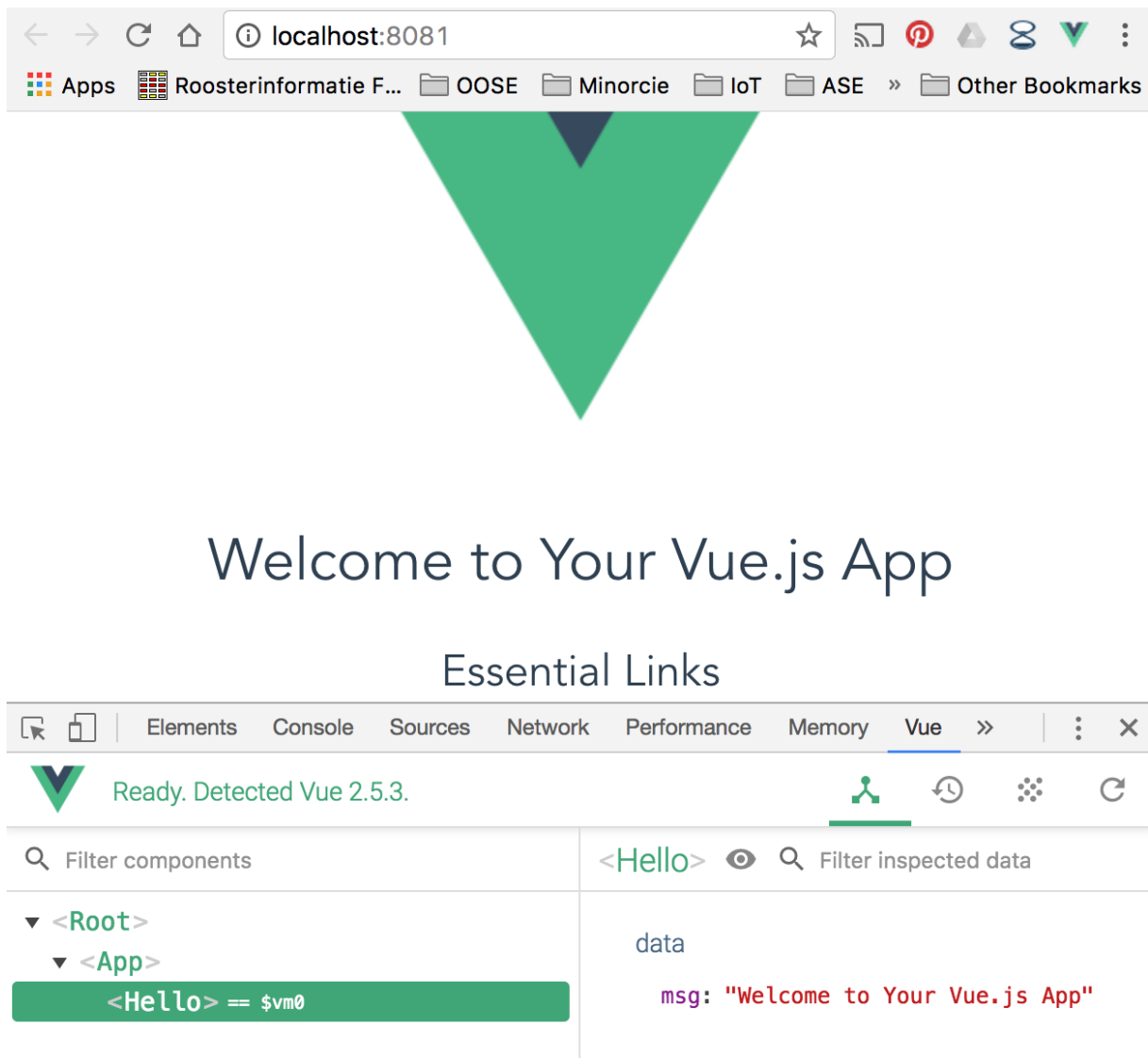
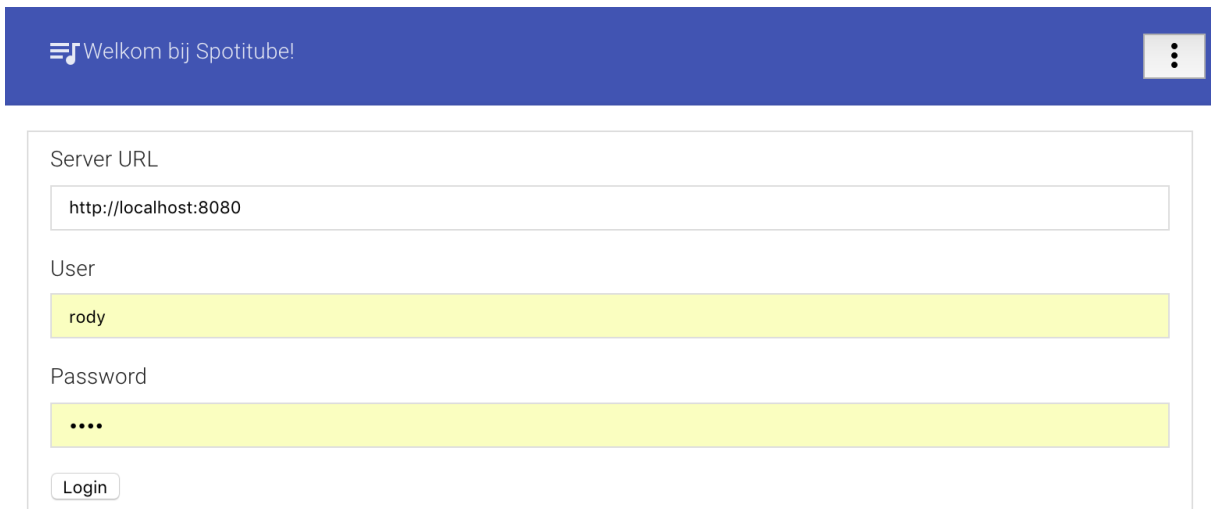


Figure 1: *Figure 1: Running the application with the devtools extension installed*

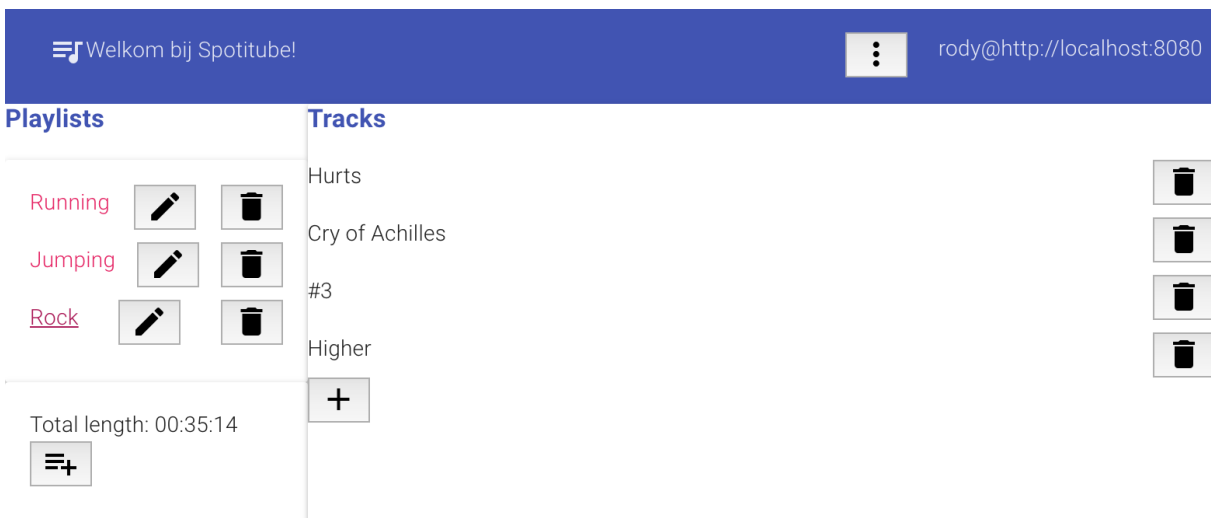
## Example Vue: Spotitube

---



The login screen features a dark blue header with the text "Welkom bij Spotitube!" and a hamburger menu icon on the left, and a three-dot menu icon on the right. Below the header is a white form with three input fields: "Server URL" containing "http://localhost:8080", "User" containing "rody", and "Password" containing four dots. A "Login" button is positioned at the bottom left of the form.

Figure 2: Figure 2: Login screen Spotitube



The Playlists screen has a dark blue header with "Welkom bij Spotitube!" and a hamburger menu icon on the left, and a three-dot menu icon and the user email "rody@http://localhost:8080" on the right. The main content is split into two panels. The "Playlists" panel on the left lists "Running", "Jumping", and "Rock" in pink text, each with edit and delete icons. It also shows "Total length: 00:35:14" and a playlist icon. The "Tracks" panel on the right lists "Hurts", "Cry of Achilles", "#3", and "Higher", each with a delete icon. A "+" button is at the bottom of the Tracks panel.

Figure 3: Figure 3: Playlists screen Spotitube

When you look closely to the design, you might notice that some parts of the screen are (with a bit of imagination) reusable in other applications, like the header and a logout button. For this example we chose to split the screen into the following parts:

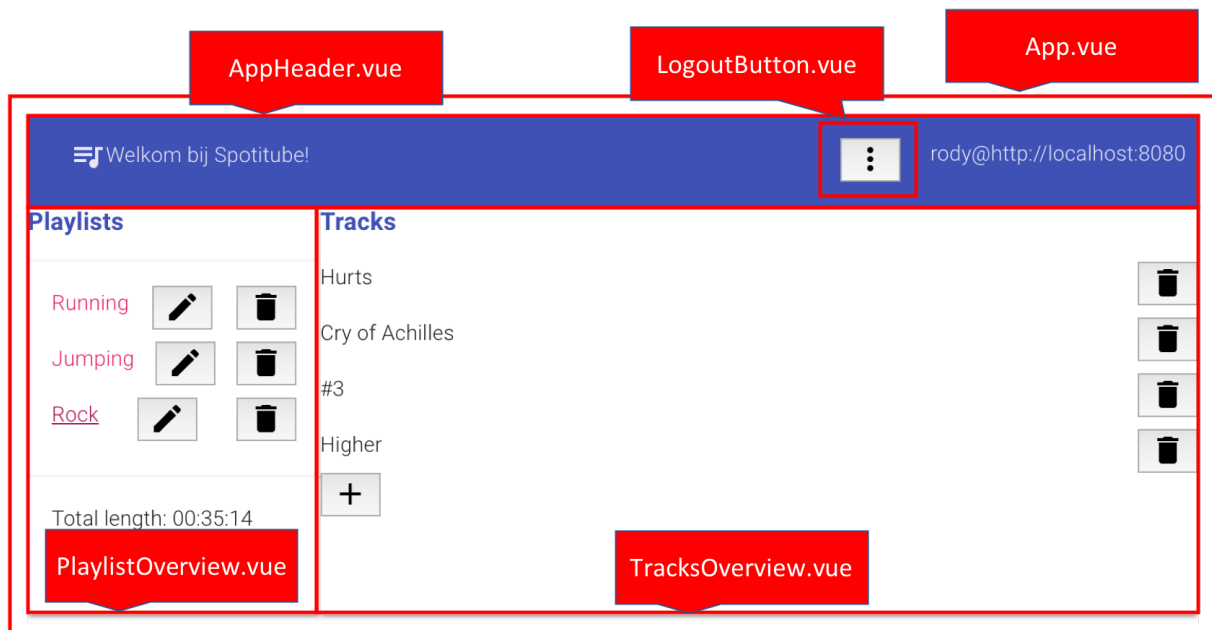


Figure 4: Figure 4: Playlists screen Spotitube in components

There are also a few components that are not always visible like:

- the Login component
- some modal dialog components: NewTrackModal, NewPlaylistModal and UpdatePlaylistModal

---

## 4 The start of a Vue application

A minimal Vue application has a html file and a javascript-file that get connected using a npm-script.

```
"build": "cross-env NODE_ENV=production browserify -g envify -p [ vueify/plu
```

---

When running `npm run build` the main.js and all its dependencies get compiled with a tool called browserify into a javascript file called build.js. This build.js is the source for the html file:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>spotitube-client</title>
    <link rel="stylesheet" href="dist/build.css">
```

```
<link rel="stylesheet" href="src/assets/css/modal.css">
<link rel="stylesheet" href="//fonts.googleapis.com/css?family=Roboto:300,400">
<link rel="stylesheet" href="//fonts.googleapis.com/icon?family=Material+Icons">
<link rel="stylesheet" href="node_modules/vue-material/dist/vue-material.css">
</head>
<body>
  <div id="app"></div>
  <script src="dist/build.js"></script>
</body>
</html>
```

---

In this HTML-file we refer to several stylesheets:

- build.css, a compile stylesheet by browserify
  - modal.css, a custom stylesheet for the application
  - two fonts and the css to use the Material style from Google (we'll get back later how to add Vue Material to a Vue application)
- 

The input for the `npm run build` steps is the `main.js` file:

```
import Vue from 'vue'
import App from './App.vue'
import VueLocalStorage from 'vue-localstorage'
import injector from 'vue-inject'
var VueMaterial = require('vue-material')

Vue.use(VueMaterial)
Vue.use(injector)
Vue.use(VueLocalStorage)

require('./api/apigateway')
require('./storage/localstorage')
require('./events/event-bus')

new Vue({ // eslint-disable-line no-new
  el: '#app',
  render: (h) => h(App)
})
```

---

This specific file is extended with extra libraries for Vue like vue-localstorage, vue-inject and vue-material. It also has a few custom classes that we're going to use later on. The minimal setup for a main.js is:

```
import Vue from 'vue'
import App from './App.vue'

new Vue({ // eslint-disable-line no-new
  el: '#app',
  render: (h) => h(App)
})
```

The el property refers to the element-id:

```
<div id="app"></div>
```

The render method renders a Vue component called App.vue, let's see what the anatomy of a Vue component looks like.

---

## 5 The anatomy of a Vue Component

Every Vue component has at least a template (the *view*) and most of the time also some “code behind” (a combination of a *model* and a *controller*). In the following paragraphs we'll use the App.vue component from the Spotitube application to show its parts.

---

### 5.1 Templates (View)

The first part of the Vue components is the template:

```
<template>
  <div id="app" class="flex-container">
    <app-header class="header flex-item"/>
    <div class="overview flex-container">
      <playlists-overview class="flex-item playlists" v-if="token != ''"/>
      <tracks-overview class="flex-item tracks" v-if="playlistId != ''/>
    </div>
  </div>
</template>
```



```
    </div>
    <login class="flex-item"/>
  </div>
</template>
```

---

Every template starts with a template tag and within the template tag some ordinary HTML-tags. Well, ordinary, you might notice tags like `app-header` and `login`. These tags refer to the names of other Vue components. Some components have some an extra Vue-attribute: `v-if`. This is called a *directive* and it can be used on a single tag or on a container tag like a `div` or even a template tag. This directive is used to do *conditional rendering* and can be used together with `v-else-if` and `v-else` but also has an alternative `v-show`.

---

Look at the Vue documentation for the exact difference. In the `App.vue` component the `playlists-overview` and `playlist-overview` components are only showed when a certain variable gets set.

---

One of the components that is placed on the `App.vue` components is the `Login` component. Before we take a look at the code behind the controller the `Login` components shows us two nice features of Vue called *data binding* and *event handling*:

```
<template>
  <div id="login" v-if="token == ''">
    <md-card>
      <md-card-content>
        <div class="form-control">
          <label class="form-label">
            Server URL
            <input id="urlInput" class="form-control" placeholder="Server URL"
              pattern="http://[a-zA-Z0-9:/.\-]+" required>
          </label>
          <label class="form-label">
            User
            <input id="userInput" placeholder="User" class="form-control">
          </label>
          <label class="form-label">
            Password
            <input id="passInput" type="password" class="form-control" p
```

```
                required>
            </label>
            <button id="loginButton" v-on:click="executeLogin">Login</button>
        </div>
    </md-card-content>
</md-card>
</div>
</template>
```

---

There are three HTML elements with a `v-model` attribute set. This attribute causes a two-way binding between the template and its data: when a user changes the input, the data is changed and vice versa (e.g. when the data is altered by a REST-call). The login template also has a button, when it gets clicked the method `executeLogin` is executed. But where is this method located? And where is the data located for the `v-if` and `v-model` directives?

---

When you need component specific CSS you can add a `style` tag at the bottom of the `.vue` file, these styles can be scoped so they can only be used by this single component:

```
<style scoped>
    .overview {
        height: calc(100% - 64px);
        width: 100%;
    }
</style>
```

---

## 5.2 Behaviour (Model: data and Controller: methods)

The behaviour of a Vue component is determined by its methods and its data. Let's first take a look at the behaviour of the `App.vue` component:

```
<script>
    import Login from './components/Login.vue'
    import AppHeader from './components/AppHeader.vue'
    import PlaylistsOverview from './components/PlaylistsOverview.vue'
    import TracksOverview from './components/TracksOverview.vue'
```

```
export default {
  name: 'app',
  components: {
    Login,
    AppHeader,
    TracksOverview,
    PlaylistsOverview
  },
  created () {
  },
  data () {
    return {
      token: '',
      playlistId: ''
    }
  }
}
```

</script>

---

It starts (and ends) with a `script` tag and a few imports to locate the inner components like the `Login.vue` component which are contained in a property called `components`. The name is set by a property called `name` and together with a `data` property it is contained in a simple JavaScript object (`{}`). This component also contains a property (a *lifecycle hook*) called `created`, the lines in this property are executed when the component gets created during the lifecycle.

---

Besides the `created` hook a Vue component can contain other hooks:

- `beforeCreate`
- `beforeMount`
- `mounted`
- `beforeUpdate`
- `updated`
- `beforeDestroy`
- `destroy`

In this example we only use the `created` hook.

---

The App.vue components has no methods, however the Login.vue component has:

```
<script>
  export default {
    name: 'login',
    dependencies: ['apiGateway', 'localStorage'],
    created () {
    },
    data () {
      return {
        user: '',
        pass: '',
        serverlocation: '',
        token: ''
      }
    },
    methods: {
      executeLogin: function () {
        this.apiGateway.login(this.user, this.pass, this.serverlocation, this.se
      },
    }
  }
</script>
```

---

The method `executeLogin` is executed when the button with id “loginButton” is clicked. This method delegates its behaviour to a helper object called `apiGateway` (implements Fowler’s Gateway pattern) from the “class” (between quotes for now, but you can actually use classes with TypeScript) `ApiGateway`.

---

This helper object is used in the list of dependencies, a non standard property of a Vue component. Non standard because it is used by a companion library called `vue-inject` that adds dependency injection capabilities to Vue. Instead of creating an instance (or reference) to the `ApiGateway` we rely on `vue-inject` to do that for us. This makes our code cleaner and improves testability.

---

The `vue-inject` library is already added to the `main.js` and it’s going to resolve all the dependencies of components by looking at its reference name (“`apiGateway`”) and looking it up in the injector.

For the ApiGateway to be found by vue-inject it must register itself with the injector (method contents of ApiGateway are left empty to improve readability):

```
import axios from 'axios'
import injector from 'vue-inject'

export class ApiGateway {
  login (user, pass, serverlocation, callback) {
  }

  getPlaylists (token, callback) {
  }

  deletePlaylist (token, id, callback) {
  }

  addPlaylist (token, name, callback) {
  }

  updatePlaylist (token, id, name, callback) {
  }

  getTracks (token, id, callback) {
  }

  addTrackToPlaylist (token, playlistid, track, callback) {
  }

  removeTrackFromPlaylist (token, playlistid, trackid, callback) {
  }

  getAllTracksNotInPlaylist (token, playlistid, callback) {
  }
}
```

```
injector.service('apiGateway', ApiGateway)
```

---

The injector is imported together with the Axios library which is used to create REST calls later in the example. The last line of the ApiGateway injects an instance to the application with the name “apiGateway” so it can be resolved by vue-inject when components have one or more dependencies.

---

## 6 How to unit-test components

Since we use dependency injection in our components we can stub or mock dependencies per component. But first we need a proper testcase and a test runner. When you use `vue-cli` it adds Karma (a test runner), Jasmine (a test dialect with asserts) and PhantomJS (a headless browser to run the code in/with).

---

During the development of the Spotitube example we stumbled upon Petrol which is very simple to use and it works great together with vue-inject (unlike Karma).

---

To switch from Karma to Petrol we need to change the package.json file on several places. Change the `scripts.test` property in the package.json file to the following value:

```
"test": "./node_modules/nyc/bin/nyc.js --reporter=lcov --e .vue ./node_modules"
```

Also add Petrol to the list of devDependencies:

```
"petrol": "^1.0.10",
```

Add Istanbul (nyc) to the list of devDependencies:

```
"nyc": "^11.3.0",
```

---

Let's build a test for our Login.vue component which has dependencies on the ApiGateway and the LocalStorage. When the test is not run in an actual browser the LocalStorage is not available and the build a unit test (instead of an integration test) we need to stub or mock the ApiGateway:

```
import VueTestCase from 'petrol/core/VueTestCase'
import Login from '../src/components/Login.vue'
import {EventBus} from '../src/events/event-bus.js'
```

---

```
import Vue from 'vue'
import injector from 'vue-inject'

Vue.use(injector)
require('./stubs/apigateway.js')
require('./stubs/localstorage.js')

export default class LoginTest extends VueTestCase {
  beforeEach () {
    this.SUT = this.mount(Login)
  }

  /** @test */
  correctLoginAndLogoutFlow () {
    this.fillField('#urlInput', 'http://localhost:8080')
    this.fillField('#userInput', 'test')
    this.fillField('#passInput', 'rody')
    this.click('#loginButton')
    this.assertEquals(this.SUT.token, 'tokentje')
    EventBus.$emit('logged-out')
    this.assertEquals(this.SUT.token, '')
  }

  /** @test */
  receivesNoTokenWhenLoginCredentialsAreIncorrect () {
    this.fillField('#urlInput', 'http://localhost:8080')
    this.fillField('#userInput', 'test')
    this.fillField('#passInput', 'rody_')
    this.click('#loginButton')
    this.assertEquals(this.SUT.token, '')
  }
}
```

---

A Petrol testcase for Vue Components extends the VueTestCase class so we can refer to the SUT (Subject Under Test) and the mount function to trigger the mount and created hooks on the component.

---

Petrol has a `beforeEach` (`@Before` in JUnit) that is called before every test function. A test function starts with `test` or gets annotated with `/** @test */`.

---

In a test function you can refer to the component with `this.SUT` and to several cool `VueTestCase` functions like:

- `this.assertEquals`, asserts that determine the pass/failure of the testcase
- `this.fillField`, fill a field like a user should. First parameter is a `querySelector`, second parameter is the value to fill the field with.
- `click`, forces a click event on a HTML-element given the `querySelector`

---

Properties (data-fields) can be read easily, e.g. `this.SUT.token`. For this testcase two test doubles (stubs) are used in combination with `vue-inject`:

```
Vue.use(injector)
require('./stubs/apigateway.js')
require('./stubs/localstorage.js')
```

---

This makes the `Login.vue` use the stubs instead of the real dependencies. The stubs are prepared to generate return values just to make the tests pass:

```
import injector from 'vue-inject'

export class ApiGateway {
  login (user, pass, serverlocation, callback) {
    if (pass === 'rody') callback('tokentje')
  }
}

injector.service('apiGateway', ApiGateway)
```

---

When you run `npm run test` all tests get executed but it also generates a coverage folder with LCOV-info that can be used by a code quality tool like SonarQube.

---



In this testcase we also see a call to a module called EventBus which is used to pass data between non-related (not in a parent-child hierarchy) components. More about the EventBus in the section “Between non-related components: using a simple event-bus”.

---

## 7 How to exchange data between components

Good components have a clear interface and follow good design guidelines like low coupling and high cohesion.

---

Vue components also have a clear boundary and besides their dependencies on helper classes in most cases these components don’t “know” about other components’ existence which is a good point from a design point of view but as a developer you need to exchange data between these components, what are your options?

---

### 7.1 Between a parent and child component: properties

When a component has child components there is a strong relation between these components but there is no object reference to access data or methods. To exchange data between parent and child components Vue has *properties* defined by the props attribute. Suppose we need to add tracks to a selected playlist, how are we going to pass the selected playlist from the TracksOverview.vue component to a modal dialog to select a new track for this playlist?

---

Given the following template of the TracksOverview.vue component:

```
<template>
  <md-card class="flex-item">

    <div id="tracks-overview" class="modal-header">
      <h3>Tracks</h3>

      <ul class="flex-container">
        <li v-for="track in tracks" class="flex-item flex-container">
          <span class="playlist-name">{{ track.title }}</span>
        </li>
      </ul>
    </div>
  </md-card>
</template>
```

```

        <button v-on:click="executeDelete(track.id)"><md-icon>delete</md-ico
    </li>
</ul>
    <button @click="openDialog('newTrack')" class="add-button" id="openNewTr
        <md-icon>add</md-icon>
    </button>
</div>
<new-track-modal :playlistid="playlistId"></new-track-modal>
</md-card>
</template>

<script>
import NewTrackModal from './NewTrackModal.vue'

export default {
  name: 'tracks-overview',
  dependencies: ['apiGateway', 'localStorage'],
  components: { NewTrackModal },
  created () {
  },
  data () {
    return {
      tracks: [],
      playlistId: ''
    }
  },
  methods: {
    openDialog (ref) {
      this.$bus.$emit('openNewTrack', ref)
    },
  }
}
</script>

```

---

A cool thing about this template is that it also shows a Vue directive called `v-for` that can be used to iterate over a collection (e.g. an array). The template of the `TracksOverview.vue` components refers to

a child component called `NewTrackModal`, a modal dialog for the user to select a track to add to the current playlist.

---

The `TracksOverview.vue` has a `playlistId` and passes it to the dialog:

```
<new-track-modal :playlistid="playlistId"></new-track-modal>
```

---

The `:playlistid` property refers to a defined property in the code behind of the `NewTrackModal.vue` component:

```
<script>
export default {
  template: '#new-playlist-modal-template',
  dependencies: ['apiGateway', 'localStorage'],
  props: ['playlistid'],
  data: function () {
    return {
      tracks: [],
      track: {},
      selected: ''
    }
  },
  created () {
    this.$bus.$on('openNewTrack', ref => {
      this.$refs[ref].open()
      this.apiGateway.getAllTracksNotInPlaylist(this.localStorage.get('token'))
    })
  },
}
</script>
```

---

This *binding* is one-way instead of the `v-model` directive that we used earlier in this example. That means that a child-component can *read* the property and not alter it, if you try to do so Vue will show an error in the JavaScript-console in the browser.

---

Notice that these code snippets also refer to `$bus`, a simple event bus to generate (`$emit`) and listen to (`$on`) developer-defined events. This eventbus is used to determine when the modal dialog has to be shown and hidden, normally a `v-if` or `v-show` directive would do, but this example depends on Material which needs a special kind of opening and closing dialogs.

---

## 7.2 Between non-related components: using a simple event-bus

In the LoginTest testcase we already saw the EventBus being used. The source code for this module is simple:

```
import Vue from 'vue'
export const EventBus = new Vue()
Object.defineProperty(Vue.prototype, {
  $bus: {
    get: function () {
      return EventBus
    }
  }
})
```

---

It creates an instance of Vue and passes it to every JavaScript object using the prototype and makes it accessible with the `$bus` property. Events can be generated using the `$emit` function:

```
openDialog (ref) {
  this.$bus.$emit('openNewTrack', ref)
},
```

---

Events can be received using the `$on` function, in most cases we start listening to incoming events in the created hook:

```
created () {
  this.$bus.$on('openNewTrack', ref => {
    this.$refs[ref].open()
    this.apiGateway.getAllTracksNotInPlaylist(this.localStorage.get('token'))
  })
},
```

This sequence diagram shows how the components work together using a property and the event-bus to open a dialog:

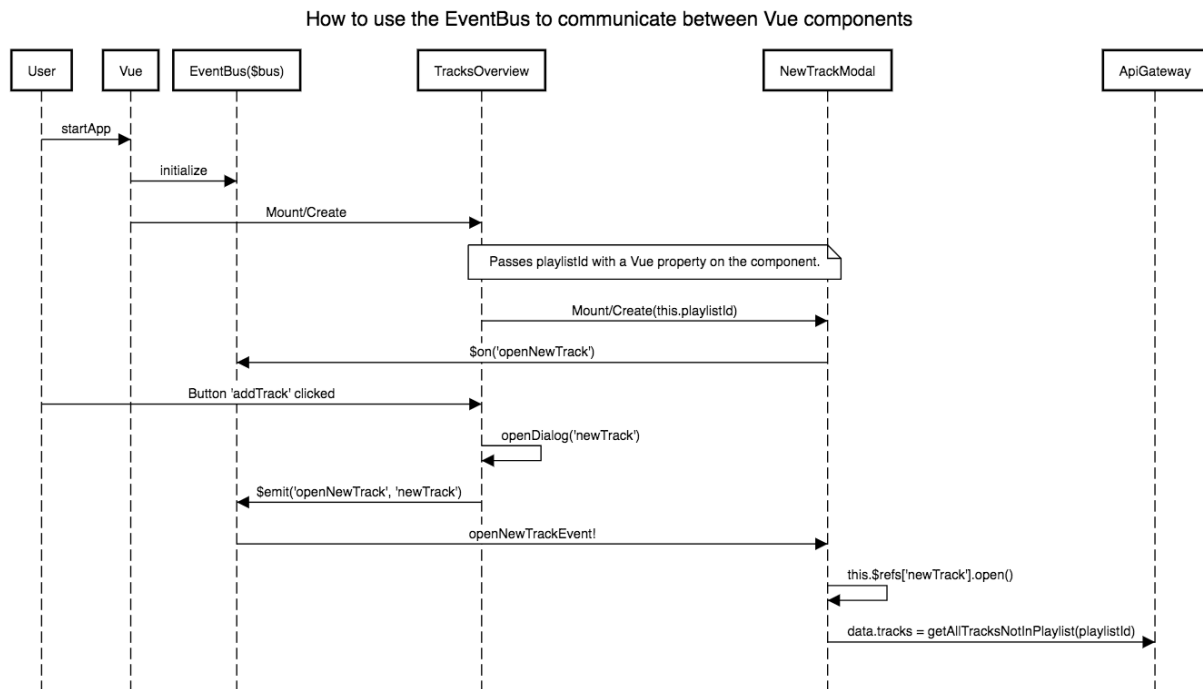


Figure 5: Figure 5: Sequence diagram to show how components can share data

---

### 7.3 Between non-related components: using a shared store component

The Vue documentation advises to only use the event-bus in simple cases and to use a shared store component in more complex cases. Vue offers several ways to handle state management:

- Vuex, a flux-like implementation
- Reacts' Redux with simple bindings to use in Vue
- Build a simple store component yourself

Look into one or more of these options yourself when the event-bus solutions gets to messy or untestable.

---

## 7.4 How to communicate with REST services using Axios

How does a Vue application communicate with a backend? Due to its simplicity Vue doesn't have features to communicate with REST-services but it's easy to extend Vue with extra libraries, for this example we use Axios, a promise based HTTP client.

---

## 7.5 Axios supports all HTTP verbs by providing separate functions (get, post, delete, put, etc.) but you can also use a generic request and tell Axios what verb to use.

Before you can use Axios, it needs to be added to the package.json file:

```
"dependencies": {  
  "axios": "^0.17.0",  
},
```

---

Libraries like Axios work asynchronous so you need to work with the Promise API or use a simple callback function, in this example we used a callback function, for instance in the Login.vue component:

```
methods: {  
  executeLogin: function () {  
    this.apiGateway.login(this.user, this.pass, this.serverlocation, this.se  
  },  
  setToken: function (token) {  
    this.localStorage.set('token', token)  
    this.token = token  
    this.$bus.$emit('token-set', {'token': this.token, 'user': this.user, 's  
  }  
}
```

---

The Login.vue component calls the ApiGateway to request a login and registers a callback function that can be executed by the ApiGateway when the REST-call receives a response:

```
import axios from 'axios'  
import injector from 'vue-inject'  
  
export class ApiGateway {
```

```
login (user, pass, serverlocation, callback) {  
  this.serverlocation = serverlocation  
  
  axios.post(this.serverlocation + '/login', {  
    'user': user,  
    'password': pass  
  })  
    .then(response => {  
      callback(response.data.token)  
    })  
    .catch(e => {  
      console.log(e)  
    })  
}  
  
injector.service('apiGateway', ApiGateway)
```

---

## 7.6 Conclusion

Vue is a simple but effective JavaScript library to build Single Page Applications. It has less features than Angular but it can be extended easily with libraries you choose yourself.