2000WEF_19
2000DWEF19
Web Frameworks (vt, dt)

Assignment
'Play aan Zee'
(with Vue.JS and Spring Boot)

Version	: 22.1 - Vue
Date	: 24 August 2022
Study tracks	: HBO-ICT, Software Engineering
Curriculum	: year 2, terms 1 & 2
Study guide	: https://studiegids.hva.nl/#/ vt: 2000WEF19, dt: 2000DWEF19
Course materials	: register via https://courseselector.mijnhva.nl/ vt: https://dlo.mijnhva.nl/d2l/home/467567 dt: https://dlo.mijnhva.nl/d2l/home/457817

Versions

Version	Date	Author	Description
21.1	15 Aug 2021	John Somers	Initial version
22.1	24 Aug 2022	John Somers	Upgraded Vue version and to ES2021 Web sockets from Angular version

Table of contents

1 Introduction

This document provides a case study description and several assignments for practical exercise along with the course Web Frameworks. The assignments are incremental, building only some parts of the solution of the use case. Later assignments cannot be completed or tested without building part of the earlier assignments.

Relevant introduction and explanation about the technologies to be used in these assignments can be found in videos at https://learning-oreilly-com.rps.hva.nl/. You can find free access to these resources via https://databanken.bibliotheek.hva.nl/ and search for "O'Reilly online Learning".

The frontend is well covered by Maximilian Schwarzmüller (Academind):

O'Reilly-1. 'Vue – The Complete Guide (Including Vue Router, Vuex and Composition API)'

The backend is well covered by Ranga Karanam (In28minutes Official):

O'Reilly-2. Mastering Java Web Services and REST API with Spring Boot

O'Reilly-3. Master Hibernate and JPA with Spring Boot in 100 Steps

Consult the study guide and the study materials for specific directions and playlists that are relevant for each of the following assignments.

2 The Casus

The casus involves a web application that goes by the name '*Play aan Zee'*. Along the North-sea coastline of the Netherlands and Belgium there are small cabins available with services for daily recreation or overnight stay on the beach. This application provides a platform where owners and tourists meet for rental arrangements of these cabins.

Below you find requirements and design specifications for building this application. That should give you a big picture of how this application should work. In the assignments thereafter you will only address part of these requirements: whatever is sufficient to be able to explore relevant application of the web frameworks technologies.

2.1 Use cases

The following provides a list of features and use cases of the application:

1. There are four user roles for this application: visitors, registered users, owners, and managers.
2. Visitors can search and navigate the site anonymously to find out about availability of cabins, information about the location and terms of the rental service. There is no login required for this.
3. Visitors can self-register with just an e-mail address and a password. After registration an activation link will be sent to the specified e-mail address. Only after activation of the account, the services for registered users are available.
4. Registered users can apply for the owner role. That application will be approved by a manager. It requires full details of information in the account, such as address, VAT number, etc.
5. Managers can grant or revoke the roles of managers and owners to other registered users. Managers cannot demote themselves. Managers have all privileges in the application (also owner and registered user privileges)

6. Managers can update the list of locations and interesting information about the region of the location. Typically, a location is the area of a city or village near the coast. Managers will create new locations as and when new supply of cabins emerge at new places along the coastline.
7. Owners can create a manage the information of their cabins at various locations. This information includes the facilities that are available from each cabin, periods of availability, terms of rental and possibly additional information about the direct neighbourhood of the cabins in addition to the general information about the location that is being maintained by the managers.
8. Visitors and registered users can search the database of cabins by several criteria:
   a) by type of cabin (storage, lodge, …)
   b) by period of availability
   c) by location
   d) by owner
   e) by rental price
9. The workflow for setting up a rental arrangement of a cabin goes as follows:
   a) A registered user requests rental of a cabin for a given period.
   b) The owner approves or declines the request. When approving the owner also specifies specific payment terms (e.g. ultimate payment dates for partial amounts) and terms of cancellation.
   The platform generates and sends an approval or regret email that includes all relevant information, a contract for rental with cancellation terms and an invoice with the payment terms and instructions.
   d) The owner maintains the payment status in the system after each partial payment.
   e) The user can cancel the booking conform the cancellation terms, which the owner must approve or regret. The platform then generates and sends a cancellation approval or regret email with all relevant information and a credit invoice.
   f) The user can request an update of the period of the booking conform availability and cancellation terms, which the owner must approve or regret. The platform then generates and sends an update approval or regret email with all relevant information and an additional (credit) invoice.
   f) At any time the user can review the status of the booking and his complete history of earlier Rentals in the system.
10. The lifecycle evolution of a rental arrangement can be tracked with a status: Requested, Approved, Declined, Paid, Fulfilled, Cancelled. Owners can block availability of specific cabins for specific period. (This may be implemented with a special status ‘Blocked’ on a rental arrangement with the owner himself, without cost).
11. Owners can download an excel report of all bookings within a given period with all information per booking, including the status of each rental arrangement.
12. Owners can export a pdf-brochure with all information about their cabins.
13. Actual payments or integration with an external payment system is outside the scope of this casus.

2.2 Class diagram

Below you find a navigable class diagram of the functional model that has been designed for ‘Play aan Zee’. This diagram only includes the main entities, attributes and some operations. For a full implementation, additional classes, attributes, or operations may be required.

Every instance of the Cabin class represents a cluster of cabins of a given type of which multiple are available at a given location. Every instance of the Rental class represents a rental agreement of a single cabin for a given period. On any calendar day there can be no more approved rentals associated with a specific Cabin instance than indicated by numAvailable.

The duration attribute of a Rental is a derived attribute in days, which can be calculated from the start date up to but excluding the end date of a rental.

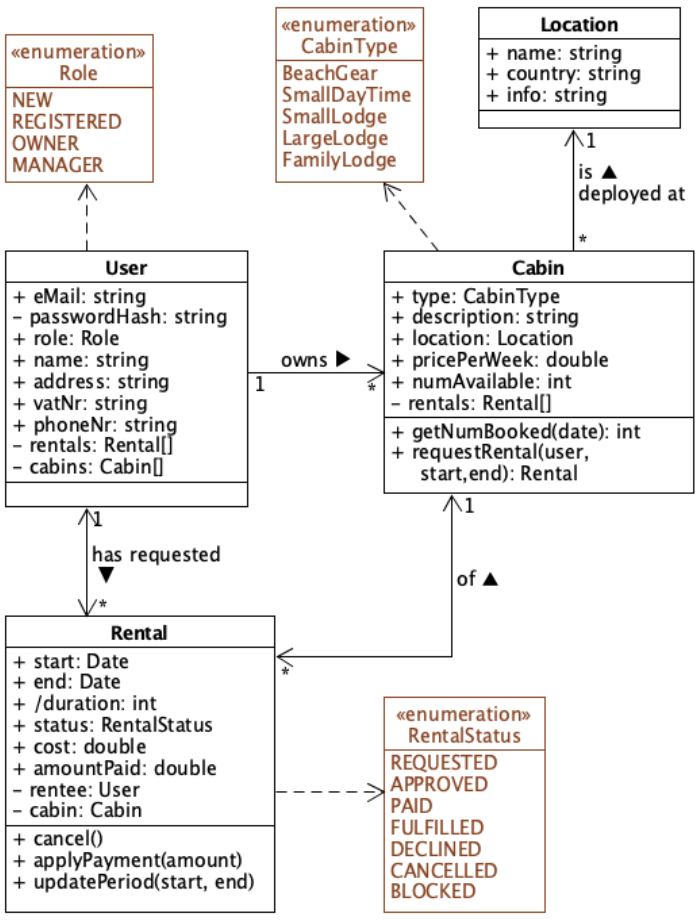
For the sake of readability, we have not included constructor and getter or setter methods in this diagram. Public property attributes should be implemented in Java with private member variables and public getters and setters.

The arrow tips of the associations indicate ‘navigable’ relations. Some are bi-directional, others are unidirectional. I.e.:

- Bi-directional navigability: Every User knows which Rentals it has requested and for every Rental we know the renter.
- Unidirectional navigability: Every Cabin knows in which Location it has been deployed, but a Location does not know any of its Cabins.

Navigability is implemented with (private) association attributes. E.g., User.rentals realises the navigability of the ‘has requested’ relation. It provides the list of all Rentals that have been requested by a User. Rental.renter realises the navigability the other way around. It provides the User of a given Rental.

This design of navigability is based on expected requirements from the use case scenario’s above. It may be that your specific implementation approach requires different relations or navigability. In any case: good software design aspires high cohesion and low coupling!



2.3 Layered Logical Architecture

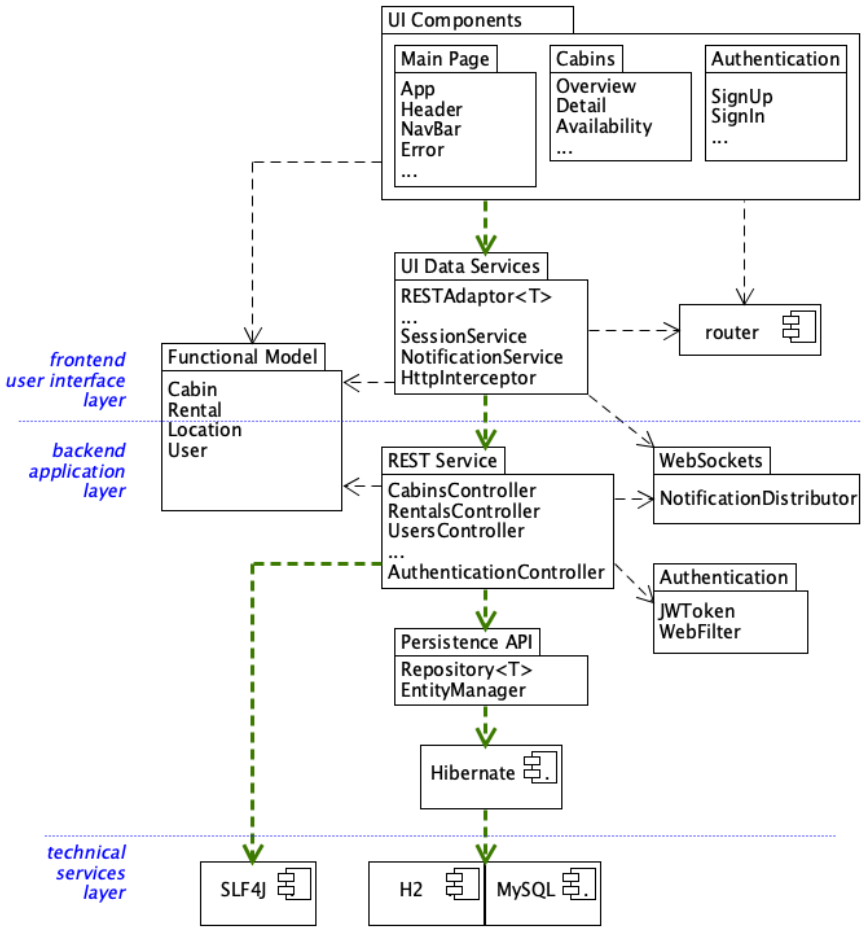
The diagram below depicts the designated full-stack, layered architecture of the *'Play aan Zee'* application. The frontend user interface layer follows the Model-View-Controller pattern of a Single-Page web application. The UI Components packages provides the View-Controller structures. The UI Data Services package provides the adaptors connecting the frontend data requirements with the backend RESTful web services.

The Functional Model is shared between the frontend and the backend, indicating that a single consistent model of the entities in the problem domain shall be implemented. This way there is no need for use of Data Transfer Objects (DTO's) by the REST services. As you will be using different programming languages in for the frontend and the backend, you will need a dual implementation of this functional model.

The green arrows in the diagram indicate dependencies on interfaces, which could be provided by multiple alternative implementation modules within the stack. Effectively, those dependencies can be inverted (the D of S.O.L.I.D.) to establish a 'plug-and-play', 'Clean Architecture' in which all dependencies are directed from the most detail (the views and the database) towards the most abstract (the functional model) (ref. Robert. C. Martin). We will leverage the 'Dependency Injection' capabilities of the Spring and Vue frameworks (at RESTAdaptor<T>, Repository<T> and EntityManager) to realize such plug-and-play architecture.

Hibernate will provide Object-to-Relational Mapping (ORM) between the Repository<T> service and the Relational Database Schema.

The H2 in-memory Database Management component will be used for testing purposes. The production configuration of your application would be expected to run with MySQL.



2.4 The technologies

To complete the assignments in this document, you need set-up of your development environment with proper technologies:

IntelliJ IDE:

1. Install IntelliJ Ultimate Edition from https://jetbrains.com/idea/.
   You can request a license code from their educational program using your HvA e-mail address. For continued use of this license, annual extension is required (and provided).
2. Enable IntelliJ Plugins (Preferences)
   Vue.JS, NodeJS
   HTML Tools, CSS Support, JavaScript Support
   Spring and Spring Boot, Hibernate
   GIT Integration, Maven Integration (+ Extension), Remote Hosts Access

FrontEnd:

1. Install Node + Node Package Manager (NPM) from https://nodejs.org/en/download/
   $ node --version
   $ npm --version
   $ sudo npm install -g npm
   $ sudo npm install -g npm@latest
   $ sudo npm install -g npm@6.14.13

2. Install Vue/CLI (Command Line Interface)
   https://cli.vuejs.org/guide/installation.html
   $ sudo npm install -g @vue/cli
   $ sudo npm update -g @vue/cli
   $ vue –version
3. If you need to upgrade an existing project to the latest version of vue:
   $ npm install vue@latest –save
   $ vue upgrade
4. If you need to add another package to an existing project:
   $ npm install postcss --save
   $ npm install vue-router –save
   $ npm install fetch-intercept whatwg-fetch –save
   $ npm install sockjs-client --save
5. If you need to upgrade packages to their latest patch/incremental release:
   $ npm outdated
   $ npm update --save
6. If you need to troubleshoot issues with dependencies:
   $ npm ls packageName

ECMAScript 2021

1. We follow the ES-2021 standard.
   Specific latest features of this version include:
   a) definition of static attributes within classes

b) the optional chaining operator '?.' Which simplifies guarding null-references
At https://kangax.github.io/compat-table/es2016plus/ you can find that these features are natively supported from the latest versions of browsers, and you may configure a vue.js project without use of the babel transpiler.
However, we have found that the native configuration is not that popular and well tested yet, so we recommend to add babel anyway (for stability and compatibility with older versions of browsers):
$ vue add babel
this adds core-js and @vue/cli-plugin-babel, and gives you a babel.config.js file.
In package.json you will find:

dependencies.core-js: "^3.8.3"
devDependencies.@babel/core: "^7.12.16"
devDependencies.@babel/eslint-parser: "^7.12.16"
devDependencies.@vue/cli-plugin-babel: "^5.0.0"
devDependencies.@vue/cli-plugin-eslint: "^5.0.0"
devDependencies.eslint: "^7.32.0"
devDependencies.eslint-plugin-vue: "^8.0.3"
eslintConfig.parserOptions.parser: "@babel/eslint-parser"

2. You may want to customize the ESlint parser rules for a better developer experience, e.g.:

eslintConfig.parserOptions.ecmaVersion: 2021
eslintConfig.rules.vue/multi-word-component-names: "off"
eslintConfig.rules.vue/no-mutating-props: "off"
eslintConfig.rules.no-unused-vars: "off"
eslintConfig.rules.no-unreachable: "off"

BackEnd:

A. Configure all dependencies in the pom.xml file.
Use a base reference to a parent version of spring:

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.6.3</version>
    <relativePath/>
</parent>
```


3 First term assignments: Vue.JS and Spring Boot

3.1 The 'Play aan Zee' Home Page

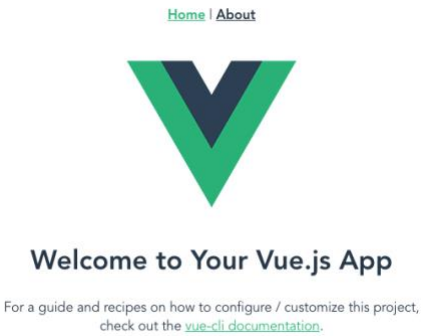
In this assignment you explore the setup of a Vue project, review its component structure and you (re-)practice some HMTL and CSS by populating the templates of a header and welcome page of your application.

A. Create a 'Single Page Application' Vue project within a parent project folder 'PlayAanZee':

$ vue create paz-vue

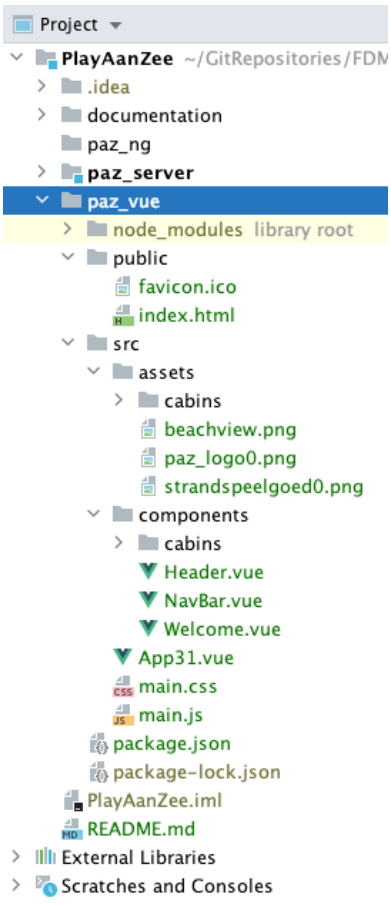
or File → New Module → JavaScript → Vue.js

Manually pick your features.
Select Vue 3.x with Babel and ESLinter.
(Router can be added later.)



Test your project by running 'serve' from the package.json file (or $ npm run serve).
Open your browser on http://localhost:8080/
Your application shall show in in the browser as in the image provided here.

Review public/index.html, src/main.js and src/App.vue and learn how your single page application is setup from the Vue foundation.

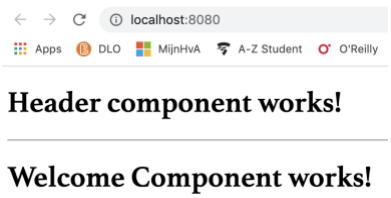


B. Let's create a simple application structure with a Header component and a Welcome component:
Clean-up/remove boiler plate sample components from the src and src/components folder structure.
Use 'src → New → Vue Component' to define three new components:

components/Header.vue
components/Welcome.vue
App31.vue

Reconfigure main.js to load the App31 component.
Reconfigure App31.vue to show the Welcome component below the Header component.
At the left you find an indication of the initial development folder structure.

Include one line of text in each of the template sections of the Header and the Welcome components. Test your project.



C. Design the Header component template to hold a background-image, two (logo) pictures left and right, a title and a sub-title. Store the pictures in src/assets.
Make sure the header is responsive such that its logos use fixed size and the title space scales with the size of the window. The subtitle should be justified to the right.

Design the Welcome component template to display its content in three columns. The left and right columns have a fixed width, the centre column scales with the window. The left-hand side of the page contains a date picker and a list of locations displaying how many cabins still are available per location at the given date. The right-hand side column has some interesting information about a special event.

All this content maybe coded statically in the template section of the components.

Import a global stylesheet main.css into main.js and use scoped styling as appropriate within each of the components.

D. Add another component 'NavBar.vue' to src/components. Show this component just below the header in App31.vue. This navigation bar should be able to provide responsive menu items and sub-menus similar to the example picture below. (See also https://www.w3schools.com/howto/howto_css_dropdown_navbar.asp or other examples in that section of w3c tutorials to find inspiration of how to build responsive navigation bars with HTML5/CSS).
Make sure that the Sign-up and Log-in entries stick to the right if you resize your window.
Also apply sub-menu expansion effects and dynamic color highlighting when navigating the menus.
Later we will associate actions with some of the menu items.

3.2 Cabins overviews

In this assignment, you further elaborate the dynamic behaviour of a component in the JavaScript controller code and data. You define model classes to identify and properly organise the functional entities of your application. You apply vue-directives to adapt the html structure dynamically and explore all four methods of binding data and events. You will configure interaction between interdependent components.

3.2.1 A list of Cabins

You apply iteration and conditional directives for the dynamic structure of your page, and use “string interpolation binding” and “event binding” to connect your HTML view to controller code and data.

A. Show today’s date in the Header from a dynamic calculation by the script code of the Header component.

(*Hint: Use the Date class and the .toLocaleString() method in the JavaScript code and string interpolation in the template of the Header.*)

B. Setup the src/models source directory.
Define in there cabin.js model class with the following instance attributes:

id: number;	image: string;
type: Cabin.Type;	pricePerWeek: number;
location: Location;	numAvailable: number;
description: string;	

Cabin.Type is an enumeration of string values “BeachGear”, “SmallDayTime”, “SmallLodge”, “LargeLodge” and “FamilyLodge”.

Besides a constructor, you implement in the Cabin class a static method:

```
static createSampleCabin(pId = 0)
```

which creates and returns a new cabin instance with realistic semi-random values for all other attributes besides the given id. (For realistic testing/demonstration purposes.)
Use Math.random() to randomise the sample content of each created cabin, i.e.:

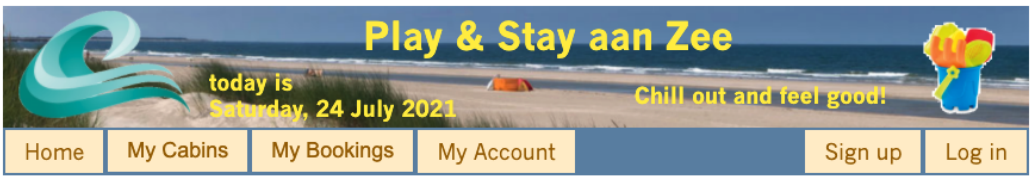
- Select at random a cabin type.
- Select a random location from a list of well known places at the coast
- Pick an image from a predefined list, which you may setup in Cabin.images
- Provide a useful description with some variation
- Pick a realistic price, also consistent with the type
- Pick a small number for the total availability.

C. Create a CabinsOverview31 component in src/components/cabins/Overview31.vue.
Instantiate a local list of at least 8 cabins from the created() hook in the controller code using `Cabin.createSampleCabin(pId)` to instantiate the

cabins. Let the component keep track of the next available cabin-id: start at 10000 and increase with a (random) increment of about 3 for each new cabin.

D. Display the cabin data within a <table> in the HTML-template of the component.
Use the iteration directives on <tr> to show all cabins from the local list.
Use 'string interpolation binding' on the cabin properties.
If the type of the cabin is 'BeachGear', no description shall be displayed.
Use CSS to style the table.
Replace the 'Welcome' view in the App31 component by your cabins list view.

E. Provide a method `onNewCabin()` which uses `Cabin.createSampleCabin(pId)` to add another cabin to the list (after generating a new unique id).
Add a button at the bottom right of the view with text: 'New Cabin'
use 'event binding' on the button that binds its 'click'-event to the component function onNewCabin().



All cabins overview:

Id:	Type:	Location:	Description:	Price p/wk:	Total available:
10003	BeachGear	Wijk aan Zee		€ 120	25
10006	SmallDayTime	Renesse	colourful SmallDayTime cabins, model-5	€ 300	8
10009	LargeLodge	Renesse	modern LargeLodge cabins, model-1	€ 900	2
10012	SmallLodge	Noordwijk	spacy SmallLodge cabins, model-4	€ 500	8
10015	LargeLodge	Noordwijk	comfortable LargeLodge cabins, model-2	€ 750	9
10018	SmallLodge	De Panne	spacy SmallLodge cabins, model-1	€ 750	12
10021	FamilyLodge	Egmond aan Zee	white FamilyLodge cabins, model-2	€ 1500	2
10024	LargeLodge	Oostende	spacy LargeLodge cabins, model-2	€ 1125	9
10027	FamilyLodge	Cadzand	cosy FamilyLodge cabins, model-2	€ 1500	6
10030	BeachGear	Oostende		€ 140	50
10033	BeachGear	Wijk aan Zee		€ 130	25
10036	SmallLodge	De Panne	characteristic SmallLodge cabins, model-1	€ 600	10

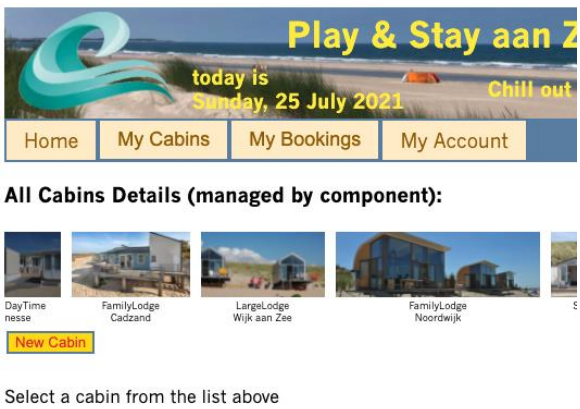
New Cabin

Test your application by adding some cabins.

3.2.2 Master / Detail component interaction

In this assignment you will develop a Master/Detail interaction component. The master component manages a selection list of cabins and embeds a detail child component which provides an edit form such that the user can change all properties of any specific cabin that is selected from the list. You apply 'property binding' and 'two-way' binding between the HTML view and the controller data of a Detail component. You explore inter-component interaction by means of 'property binding' and 'event-binding' between the Master and Detail components.

A. Add two more components 'cabins/Overview32.vue' and 'cabins/Detail32.vue' to src/components
Overview32 shall manage a list of cabins but only display their image, type and location name into a horizontal scroll container of 'cards'.
Implement the selection mechanism by binding the click event of each card in the container to a method that keeps track of the currently selected cabin in a property 'selectedCabin'. At any time, at most one cabin can be selected. When nothing has been selected yet then selectedCabin==null. When the same cabin is reselected, it will become unselected again.



Use CSS to highlight the card of the selected cabin.
The 'New Cabin' button adds a new cabin to the list and automatically selects it.

B. Overview32 embeds the Detail32 component in a sub-panel '<app-cabins-detail>'. This component shows all details of the selected cabin. While nothing is selected, a simple message shall be displayed prompting for a selection.

(*Hint: use v-if and v-else directives to configure alternative structures*)

C. Use property binding to share the selectedCabin of Overview32 with the Detail32 component and use two-way binding to connect the input elements in the template of Detail32 directly to the attributes of the selectedCabin.

```
47  import {Location} from "@/models/Location.js";
48  import {Cabin} from "@/models/Cabin.js";
49
50  export default {
51    name: "CabinsDetail32",
52    props: ['selectedCabin'],
53    emits: ['delete-cabin'],
54    data() {
55      return {
56        CabinType: Cabin.Type,
57        images: Cabin.images,
58        locations: Location.samples,
59      }
60    },
61    methods: {
62      onDelete() {...}
65    },
66    computed: {...}
80  }
```

A consequence of this approach will be that any change to selectedCabin by the Detail32 component will also apply directly to that cabin in the list that is maintained by Overview32. Both refer to the same Cabin object instance, and two-way binding is instantaneous.
Commonly that is undesired and ESLint will give you an error for this: 'vue/no-mutating-props'. For now, disable that error in eslintConfig.rules (in package.json or a dedicated ESLint config file). In assignment 3.4 you will implement a better approach.

D. Now populate the view template of Detail32:
Include the cabin-id of the 'selectedCabin' in the header text of the form.
Use appropriate <input type="…"> or <select> elements in the form with 'two-way' binding to the attributes of 'selectedCabin'.

```
computed: {
  locationUpdater: {
    // manages editing the reference to a location
    get() { return this.editedCabin.location.id; },
    set(id) {
      // TODO: find the location with given id from Location.samples
      this.editedCabin.location =
    }
  },
```

Two-way binding on primitive types may seem staightforward, but if the attribute is a reference to a complex data type, such as a Location instance, an intermediate computed property 'locationUpdater' with a custom getter and setter may be required. In this (incomplete) sample code we use the Location.id in the values of the html select options, and retrieve the associated location instance in the setter of the locationUpdater.

```
<tr>
  <td>Location:</td>
  <td>
    <select v-model="locationUpdater">
      <option v-for="(loc) of locations"...>
    </select>
    {{ editedCabin.location.country }}
  </td>
</tr>
```

All Cabins Details (managed by component):

New Cabin

Cabin details (id=30006)	
Type:	FamilyLodge
Location:	Cadzand Nederland
Description:	modern FamilyLodge cabins, model-5
Image:	FamilyLodge/roompot-family.jpg
Price per week:	1000
Total availability:	6

Delete

When a new Cabin is added via the 'New Cabin' button at the left, that cabin should automatically get the selection focus (and be associated with the Detail32 component).
When you navigate along different cabins, any changes that you make in the detail panel will be remembered, because all edits operate directly on the cabins in the list of the CabinOverview32 component.

E. Add a 'Delete' button to Detail32 which the user can click to remove the Cabin altogether:
1. Use event binding within the <app-cabins-detail> tag to pass responsibility for actual deletion of the cabin from Detail32 to Overview32 (which manages the list of cabins).
2. Provide appropriate code in Overview32 to find and delete the Cabin from the list. After deleting a Cabin, it is also removed from the display at the top, and nothing is selected anymore. (*Hint: use the .filter() method on a JavaScript array and compare cabins by their unique id value.)*
3. Unselect the deleted cabin.

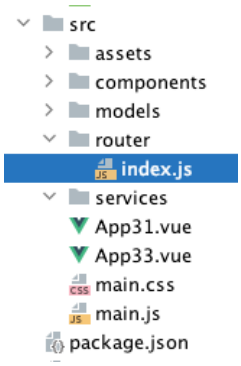
3.3 Page Routing

In these assignments we will introduce the Vue-router module to provide navigation capabilities to your application, such that all pages can be found from the navigation menu bar, and users can bookmark the url-s of specific pages in your application. The router also provides for programmatic navigation in response to computations.

3.3.1 Basic Routing

First configure the router module and connect the components that you have built in the earlier assignments to your navigation bar.

A. Check-out the default configuration of Vue-router from the boiler-plate set-up, or manually add the configuration to your existing project: (Details of this configuration have been seen to vary between versions of Vue and Vue-router.)
1. Include the vue-router module in your package.json dependencies:




```
14    "dependencies": {
15      "core-js": "^3.24.1",
16      "vue": "^3.2.13",
17      "vue-router": "^4.1.3",
```

2. Create a file src/router/index.js which instantiates the router and provides it with a routing table:
We prefer the 'hash' mode, by which the router inserts a hash-tag '#' in the URL just

```
14  import { createRouter, createWebHashHistory } from 'vue-router';
15  const routes = [...];
35
36  export const router = createRouter({
37    history: createWebHashHistory(),
38    routes
39  })
```

before the route path. That way, our webserver will ignore this internal path when resolving the page resource.

3. Configure a new src/App33.vue component, which shows the <router-view> outlet just below the <app-header> and <app-nav-bar> components. (Remove any of the other components that you had included in App31.vue.)

4. Update main.js to import and use the router, and mount your App33 component:

```
1  import { createApp } from 'vue'
2  import { router } from './router'
3  import App from './App33.vue'
4  import './main.css'
5
6  createApp(App).use(router).mount('#app');
```

B. Populate the routes table in src/router/index.js with paths to your components:
1. 'home' shows your welcome page of assignment 3.1.
2. 'cabins/overview31' shows the cabins list of assignment 3.2.1.

3. ‘cabins/overview32’ shows the master/detail of assignment 3.2.2.
Also provide a redirect from ‘/’ to the ‘home’ route.

Link these router paths to menu items in your navigation bar (using <router-link> elements with appropriate attributes.)
Apply a highlighted style to the currently selected menu-item.

Test whether routing works:
- test menubar navigation
- test redirection of the root path ‘/’
- test navigation from a bookmarked URL
- menubar highlighting should also follow the navigation from a bookmark

C. Connect the ‘Sign Up’ menu item to the ‘/sign-up’ route and the ‘Log in’ menu item to the ‘/sign-in’ route, without providing these routes in the routes table.
Add a new component ‘src/components/UnknownRoute.vue’ which provides a simple error message, indicating that a specified route is not available, just as in the example below. This component should be connected to any unknown route.

(Hint: Use the :pathMatch parameter to create a route with wildcards).

3.3.2 Parent / child routing with router parameters.

To be able to bookmark or share an URL for editing of specific cabin with the Master/Detail component, we will integrate the selection of a cabin into the router path. For that we use parent / child routing with router parameters.

A. Create a new component cabins/Overview33.vue in src/components which can be a copy of Overview32 initially. Connect Overview33 to a new route ‘cabins/overview33’ and add the route to the ‘Cabins’ sub-menu.

B. Add a child route ‘:id’ to the ‘cabins/overview33’ route in the routes table, invoking the cabins/Detail32 component as a subcomponent of Overview33. For that you also should replace the <app-cabins-detail> element in Overview33 by another <router-view> sub-component outlet.

Now the selection of a cabin to be edited must be connected to the router in two ways:

1. If the user clicks another cabin in the list view of Overview33, its code shall navigate the router programmatically to the route with designated id (or to the parent route without any selection).
2. If the router activates a new route e.g., at /cabins/overview33/12345 (initiated from the menu bar, from the browser url or programmatically) then Overview33 should select and highlight the cabin with id=12345, and also Detail32 should follow the selection in Overview33 and open that same cabin for editing.

The first task can be implemented in the 'onSelect' method of Overview33:

```
onSelect(cabin) {
  if (cabin != null && cabin !== this.selectedCabin) {
    this.$router.push(this.$route.matched[0].path + "/" + cabin.id);
  } else if (this.selectedCabin != null) {
    // TODO navigate to the parent path to unselect the selectedCabin
    {...}
  }
  return this.selectedCabin;
},
```

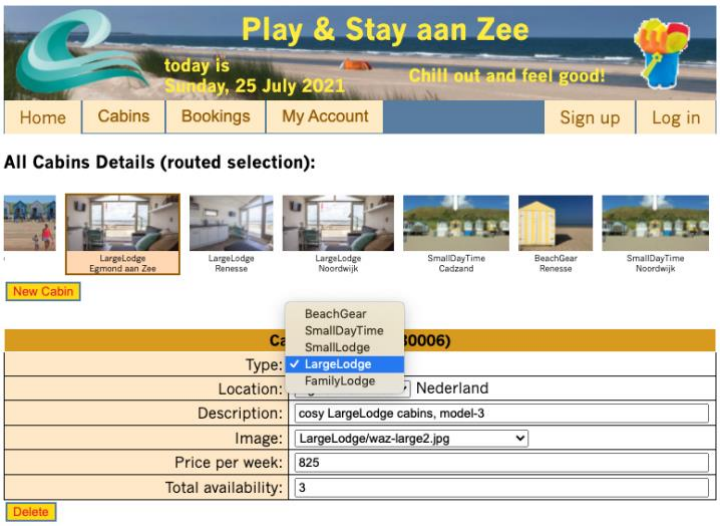
The second task can be achieved with a 'watch' declaration on '$route' in Overview33, which will fire at any time when the route has changed:

```
watch: {
  '$route'() {
    this.selectedCabin = this.findSelectedFromRouteParam(this.$route);
  }
```

Detaill32 can still bind to the 'selectedCabin' property from Overview33 and has no need to follow the router itself at this time.

Complete and update your code to fully integrate the Master/Detail components with the router.

Test your application:
1. Selection and un-selection of cabins.
2. Navigation by URL, with and without cabin-id
3. Deletion of cabins.
4. Navigation by URL to deleted cabins.



3.4 Master/Detail managed persistence

Direct editing of the selected cabin is not a best practice, because it does not allow the user to cancel any unintended changes. In this exercise you will amend the detail component to bind to a local copy of the selected cabin and manage persistence of changes to that copy. You will use navigation guards to protect loss of changes when the user decides to navigate a different route.

3.4.1 Recover from unintended changes.

A. Create a new component cabins/Detail34.vue which can be a copy of cabins/Detail32 initially. Create a new route /cabins/overview34 which invokes component Overview33 with child component Detail34. (The interface between the master and the new detail component can be compatible with the previous assignment. If you want to avoid that constraint, you may also copy and use a new master component cabins/Overview34).

B. Change your Detail34 component to clone a copy of the selected cabin (within its created() hook) and bind the input elements of the template to the attributes of that copy.
It is good practice to implement a static method copyConstructor(cabin) in the Cabin.js model class for this purpose in order to properly encapsulate deep cloning of a Cabin object within its class. You can use the utility Object.assign to automatically copy over all attributes from one instance to the other. Additional code should be added to deeply clone attributes of complex types (such as the reference to a Location in this case):

```
static copyConstructor(cabin) {
    if (cabin == null) return null;
    let copy = Object.assign(new Cabin(0), cabin);
    copy.location = Location.copyConstructor(cabin.location);
    return copy;
}
```

Don't forget to also make a fresh copy, when the selected cabin has changed.
(Hint: 'watch' the property that is bound to the selected cabin in the overview.)

C. Provide additional buttons in the form to manage persistence of changes:
The '**Save**'-button will update the selected cabin in the overview with the changes.
The '**Reset**'-button will revert any edits by reinitialising the cloned copy.
The '**Clear**'-button will clear all attributes of the edited cabin empty.
The '**Cancel'**-button will abandon the edit form and without persisting any changes.
The '**Delete**'-button will delete the selected cabin and remove it from the list regardless of any changes underway.

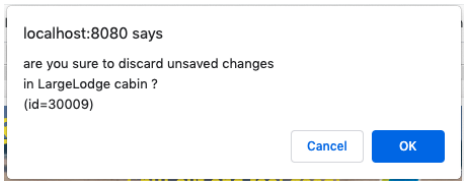
'Save', 'Cancel' and 'Delete' will also unselect the edited cabin (by programmatic interaction with the router).

D. Implement 'disabled' property binding on the 'Save', 'Reset' and 'Delete' buttons, such that 'Save' and 'Reset' are disabled if nothing has been changed yet, and 'Delete' is disabled if there are unsaved changes.
The 'Cancel' and 'Clear' buttons are always enabled.
(Hint: provide a computed property 'hasChanged' that determines whether there is any difference between the selected cabin and the cloned copy being edited. The actual (deep) comparison is best encapsulated into a Cabin.equals(other) instance method)

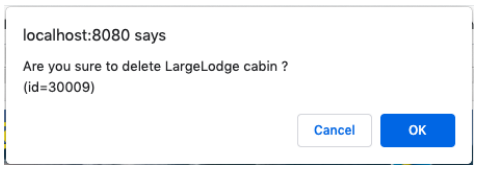
3.4.2 [BONUS] Guarded navigation

Quick use of the Reset, Clear, Cancel or Delete buttons of the previous assignment can easily lead to unintended loss of changes. Also, navigation of the application may cause loss of data that has not been saved yet.

A. Provide a pop-up confirmation box when the 'Clear'-, 'Reset'- or 'Cancel'-button is pressed while the form still has unsaved changes.
If the user confirms with 'OK' the changes may be discarded, and the original button action should be continued. If the user declines with 'Cancel' in the pop-up the focus shall remain on the earlier selected cabin, and the changes in the form shall be retained. If there are no changes in the form, no pop-up confirmation shall be prompted.



Also provide a pop-up a confirmation box when the 'Delete'-button is pressed, to prevent any unintentional deletion.



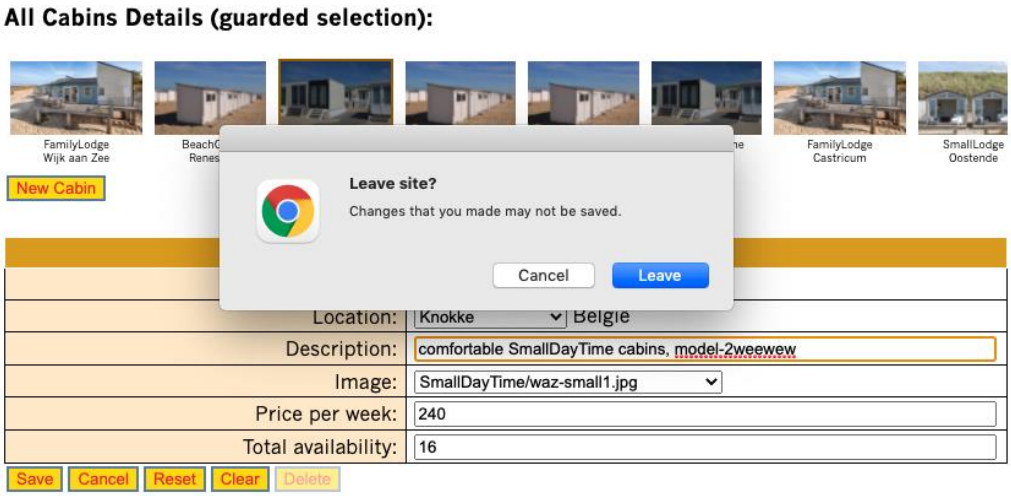
B. At https://router.vuejs.org/guide/advanced/navigation-guards.html it is explained how you can add 'Navigation Guards' to a component to assess whether it is safe to navigate away from the component or to a different selection of router parameters. If these guards evaluate false, then the undertaken navigation is blocked and reversed. This mechanism comes handy to protect unsaved changes in the form from unintentional loss by navigation.
Use 'beforeRouteUpdate' and 'beforeRouteLeave' to add two guards to Detail34 which check on unsaved changes and seek user confirmation to discard them (reusing the change detection and confirmation pop-up of the previous task).

Test whether all navigation within the application now traps changes in your form and make sure the confirmation box never pops up twice (e.g., one time by the router hooks and one time by the button handlers). Re-test all use of the buttons and the '[disabled]' properties on them.

C. These hooks only trap actions by the router. To also trap navigation to an external URL outside the scope of your application, you can add an event-listener for the ‘beforeunload’ event. (Seek the internet for advice how to use beforeunload within a Vue context)

```
mounted() {
  window.addEventListener('beforeunload', this.beforeWindowUnload);
},
beforeUnmount() {
  window.removeEventListener('beforeunload', this.beforeWindowUnload);
},
```

Your browser will pop-up a custom message when this event fires:



3.5 Setup of the Spring-boot application with a simple REST controller.

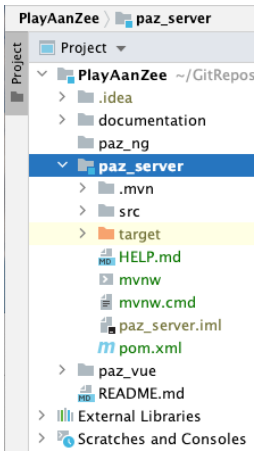
In the following assignments you will build the backend part of the 'Play aan Zee' application as depicted in the full stack, layered logical architecture of section 2.3. You will use the Spring-Boot technology.

You create a basic Spring Boot backend application and configure in there a simple REST Controller (O'Reilly-2, Chapter 4, step 4). The controller provides one resource endpoint to access the cabins of your application.

These cabins are managed in the Spring-Boot backend by an implementation of a CabinsRepository Interface. The CabinsRepository is injected into the REST Controller by setter dependency injection (O'Reilly-2, Chapter 3, step 7). First, you start with providing a CabinsRepositoryMock bean implementation that just tracks an internal array of cabins. In a later assignment you will provide a JPA implementation of this interface that links with H2 or MySql persistent storage via the Hibernate Object-To-Relational Mapper.

As of assignment 3.7 you will integrate the backend capabilities with your frontend solution of assignment 3.4

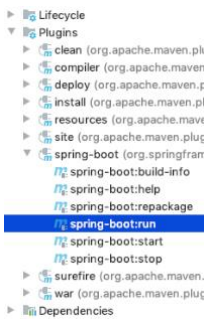
Frontend and backend modules are best configured as siblings of a full stack parent project. This approach supports:
i) use of different IDE-s for each of the modules
ii) separate configuration of automated deployment procedures for each module.
iii) multiple frontend solutions that all share the same backend api.
The git repository shall be configured at the level of the parent project or even above that (bundling multiple parent projects in a single repository)



If your frontend application was not configured in a sub-folder of your main project yet, now is the time to refactor that structure…

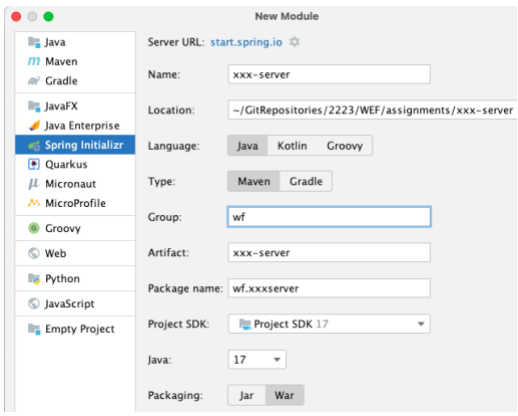
Below you find more detailed explanation of how you can implement the objectives of this assignment.

A. Create a Spring-Boot application module 'paz-server' within the parent project using the Spring Initializr plugin.
(You may need to install/activate the Spring plugins first in your Intellij settings/preferences).



Choose the war format for your deployment package.
Activate the Spring Web dependency in your module.
Also check that Maven framework support is added.

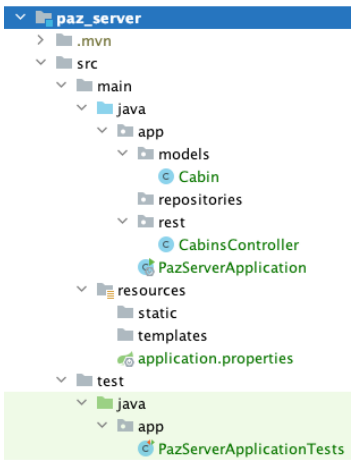
Test your project setup by running the 'spring-boot:run' maven goal.



You may want to configure the tomcat port number, the api root path, and the levels of verbosity in the server-log and error responses, all in resources/application.properties:

```
server.port=8086
server.servlet.context-path=/
logging.level.org.springframework = info
server.error.include-message=always
```

B. You may want to tidy-up the source tree of your module like shown here. Always use the refactoring mode of IntelliJ to move files around such that the dependencies will be sustained. Basic rules for setup are:



1. Your PazServerApplication class should reside within a non-default package (e.g. 'app'). (Otherwise, the autoconfig component scan may hit issues).
2. Autoconfiguration searches for beans only in your main application package and its sub-packages (e.g. 'models', 'repositories' and 'rest' in this example).
3. The package structure under 'test/java' should match the source structure under 'main/java'

If you have pulled the back-end source tree from Git, you may find that the Intellij module configuration file is not maintained by Git, and you need to configure the module File → New → Module from Existing Sources … → import from Maven pom.xml.

C. Implement the Cabin model class and the CabinsController rest controller class, as explained in O'Reilly-2.
Replicate/convert your Cabin.java model class from the frontend code.
Implement in the CabinsController a single method 'getTestCabins()' that is mapped to the '/cabins/test' end-point of the Spring-Boot REST service. This method should return a list with just two cabins like below.

```
public List<Cabin> getTestCabins() {
    return List.of( new Cabin(30001, Cabin.Type.SmallLodge),
                    new Cabin(30002, Cabin.Type.FamilyLodge) );
}
```

Run the backend and use Postman to test your endpoint. (Download and install Postman from https://www.getpostman.com/downloads/)
Your Postman test should deliver the cabins like you expect.

D. Define an CabinsRepository interface and an CabinsRepositoryMock bean implementation class in the repositories package like the SortAlgorithm example of Ranga. Spring-Boot should be configured to inject an CabinsRepository bean into the CabinsController.

The CabinsRepositoryMock bean should manage an array of cabins. Let the constructor of CabinsRepositoryMock setup an initial array with 7 cabins with some semi-random sample data.
The static method to create some sample cabin can best be implemented in the Cabin class itself:

```
public static Cabin createSampleCabin(long id) {
    Cabin cabin = new Cabin(id);
    // TODO put some sem-random values in the cabin attributes
```

The responsibility for generating and maintaining unique ids should be implemented by the CabinsRepositoryMock bean. Later, that responsibility will be moved deeper in the backend into the ORM.

The CabinsRepository interface provides one method 'findAll()' that will be used by the endpoint to retrieve and return all cabins:

```
public interface CabinsRepository {
    List<Cabin> findAll();
}
```

```
public List<Cabin> getAllCabins() {
    return this.cabinsRepo.findAll();
}
```

Make sure you provide the appropriate @RestController, @Component, @Autowired, @RequestMapping and @GetMapping annotations to configure the dependency injection of Spring Boot.

Test your endpoint again with Postman:

Pretty Raw Preview Visualize JSON


```
[
    {
        "id": 30003,
        "type": "BeachGear",
        "numAvailable": 50,
        "description": "cosy BeachGear cabin model d.2",
        "image": "BeachGear/de-panne-strandcabine.jpg",
        "pricePerWeek": 150.0
    },
    {
        "id": 30007,
        "type": "FamilyLodge",
        "numAvailable": 6,
        "description": "colourful FamilyLodge cabin model e.3",
        "image": "FamilyLodge/caz-6pers.jpg",
        "pricePerWeek": 1200.0
    },
    {
        "id": 30011,
        "type": "FamilyLodge",
        "numAvailable": 5,
        "description": "characteristic FamilyLodge cabin model a.2",
        "image": "FamilyLodge/caz-6pers.jpg",
        "pricePerWeek": 1500.0
    },
```

3.6 Enhance your REST controller with CRUD operations

In this assignment you will enhance the repository interface and the /cabins REST-api with endpoints to create new cabins and get, update, or delete specific cabins. You will enhance the api responses to include status codes, handle error exceptions and involve a dynamic filter on the response body to be able to restrict content from being disclosed to specific requests.

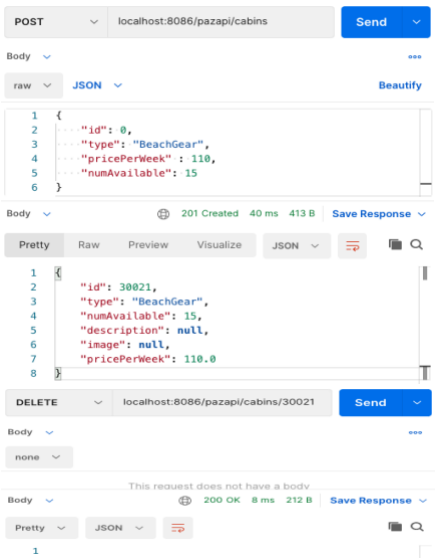
In this assignment you should practice hands-on experience with Spring-Boot annotations @RequestMapping, @GetMapping, @PostMapping, @DeleteMapping, @PathVariable, @RequestBody, @ResponseStatus, @JsonView and @Configuration and classes ResponseEntity, ServletUriComponentsBuilder.

By the end of this assignment, your CabinsRepository interface should have evolved to

```
public interface CabinsRepository {
    List<Cabin> findAll();          // finds all available cabins
    Cabin findById(long id);        // finds one cabin identified by id
                                    // returns null if the cabin does not exist
    Cabin save(Cabin cabin);        // updates the cabin in the repository identified by cabin.id
                                    // inserts a new cabin if cabin.id==0
                                    // returns the updated or inserted cabin with new cabin.id
    Cabin deleteById(long id);      // deletes the cabin from the repository, identified by id ;
                                    // returns the instance that has been deleted or null
}
```

A. Enhance the CabinsRepositoryMock class with actual implementations of all CRUD methods as listed in the CabinsRepository interface above.
The ids can be arbitrary integer numbers, so you may need to implement a linear search algorithm to find and match a given cabin-id with the available cabins in the private storage of the CabinsRepositoryMock instance.

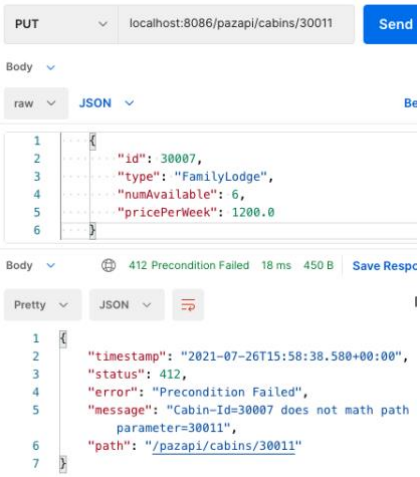
B. Enhance your REST CabinsController with the following endpoints:
- a GET mapping on '/cabins/{id}' which uses repo.findById(id) to deliver the cabin that is identified by the specified path variable.
- a POST mapping on '/cabins' which uses repo.save(cabin) to add a new cabin to the repository.
If a cabin with id == 0 is provided, the repository will generate a new unique id for the cabin. Otherwise, the given id will be used.
- a PUT mapping on '/cabins/{id}' which uses repo.save(cabin) to update/replace the stored cabin identified by id.
- a DELETE mapping on '/cabins/{id}' which uses repo.deleteById(id) to remove the identified cabin from the repository.



Use the ServletUriComponentsBuilder and ResponseEntity classes to return an appropriate response status(=201) and location header in your cabin creation response.
Use the .body() method to actually create the ResponseEntity such that it also includes the cabin with its newly generated id for the client.

Test the new mappings with postman.

C. Implement Custom Exception handling in your REST CabinsController:



- throw a ResourceNotFound exception on get and delete requests with a non-existing id.
- throw a PreConditionFailed exception on a PUT request at a path with an id parameter that is different from the id that is provided with the cabin in the request body.

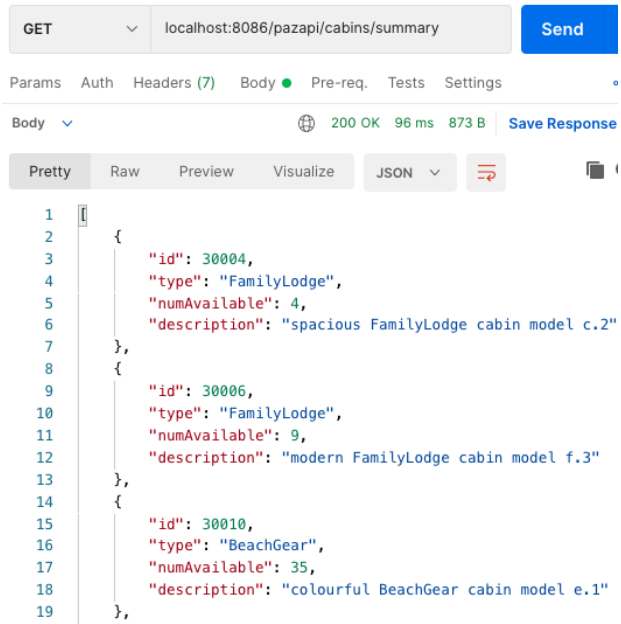
Test the mapping with postman.

If error messages do not show up in the response, you may need to update your configuration in application.properties:

```
server.error.include-message=always
```

D. Implement a dynamic filter at a getCabinsSummary() mapping at '/cabins/summary', which only returns the id, type, description and numAvailable of every cabin. Dynamic filters can most easily be implemented with a @JsonView specification in your Cabin class in combination with the same view class annotation at the request mapping in the rest controller.

Test the mapping with postman.



3.7 Connect the FrontEnd via the JavaScript Fetch-API

In this assignment you will explore the JavaScript Fetch-API and HTTP/AJAX requests to implement the interaction between your frontend application and the backend REST API.

A backend REST service is an a-synchronous service. Some delay may pass between the moment of the HTTP request and the event that a response is delivered. The ECMAScript language specification provides a powerful paradigm of 'Promises' and 'asynchronous functions' (async/await) by which you can maintain an intuitive sequential structure of your code which will execute responsively to a-synchronous events at run-time.

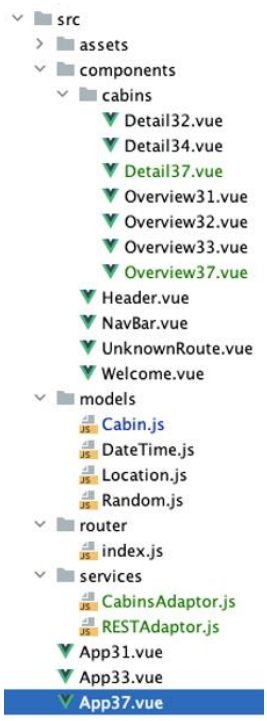
3.7.1 A REST Adaptor for Fetch-API requests.

You will implement a REST-adaptor frontend class in JavaScript, which will provide an a-synchronous interface to components in your user interface and uses 'async fetch' to handle all details of the interaction with the back end. You will instantiate multiple instances of this adaptor for different scopes and resource endpoints, and share adaptors among active components by means of 'Dependency Injection'

Additionally, you will configure the CORS in the backend to support use of multiple ports for different backend services

A. Replicate 'Overview33/Overview34' and 'Detail34' components of assignment 3.4 into new components 'Overview37' and 'Detail37'.
   Create a new route /cabins/overview37 which invokes component CabinsOverview37 with child component CabinsDetail37. Add an option for this route to your cabins menu.
   Replicate 'App33' into a new 'App37' component and don't forget to launch this App37 from main.js.

   Test whether your 'Overview37' still works as well as earlier.



B. Next, create a services folder and implement a cabins adaptor, which will provide connectivity to the /cabins endpoint mapping of the backend.
   Four basic CRUD operations shall be implemented by this adaptor:
   asyncFindAll() retrieves the list of all cabins
   asyncFindById(id) retrieves one cabin, identified by a given id
   asyncSave(cabin) saves an updated or new cabin and returns the saved instance.
   asyncDeleteById(id) deletes the cabin identified by the given id.

   New cabins shall be set up with id=0. The backend service should generate a new, unique id for such cabins that are being saved with id=0, and then return the saved instance with the proper id set.

Below is some code snippet to get started with the frontend implementation of the CabinsAdaptor:

```
export class CabinsAdaptor {
    resourcesUrl;
    constructor(resourcesUrl) {
        this.resourcesUrl = resourcesUrl;
        console.log("Created CabinsAdaptor for " + resourcesUrl);
    }

    async fetchJson(url, options : null  = null) {
        let response = await fetch(url, options)
        if (response.ok) {
            return await response.json();
        } else {
            // the response body provides the http-error information
            console.log(response, !response.bodyUsed ? await response.text() : "");
            return null;
        }
    }

    async asyncFindAll() /* :Promise<Cabin[]> */ {
        console.log('CabinsAdaptor.asyncFindAll()...');
        const cabins = await this.fetchJson(this.resourcesUrl);
        return cabins?.map(s => Cabin.copyConstructor(s));
    }

    async asyncFindById(id) /* :Promise<Cabin> */ {...}

    async asyncSave(cabin) /* :Promise<Cabin> */ {...}

    async asyncDeleteById(id) {...}
}
```

The resourcesUrl in the constructor specifies the endpoint of the backend API.

The (private) async method fetchJson is the workhorse of this adaptor, issuing all AJAX requests to the backend API. POST, PUT and DELETE requests can be configured by means of the options parameter.

Notice the use of the Cabin.copyConstructor static method to map all json object structures from the response into true instances of the frontend Cabin class. Without such mapping the cabins from the response would not have any methods defined, or complex attributes such as dates would not have been converted to proper classes. (See also assignment 3.4.1.)

C. Next, we provide a singleton instance of this CabinsAdaptor from App37 to be shared across all active components: Vue.js provides Dependency Injection for that (See provide/inject at https://v3.vuejs.org/guide/component-provide-inject.html).

The App37 component creates and provides the singleton instance:

```
import CONFIG from '../app-config.js'
export default {
  name: "App",
  components: {'app-header': Header...},
  provide() {
    return {
      // stateless data services adaptor singletons
      cabinsService: new CabinsAdaptor(CONFIG.BACKEND_URL+"/cabins"),
```

The Overview37 and Detail37 components both inject the instance:

```
export default {
  name: "CabinsOverview37",
  inject: ['cabinsService'],
```

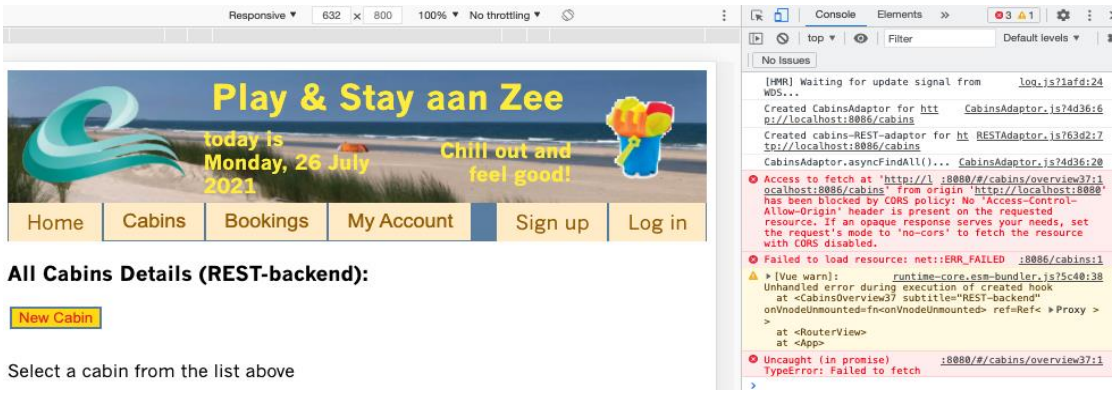
```
export default {
  name: "CabinsDetail37",
  inject: ['cabinsService',
```

With that, these components are both set up to use the service adaptor to interact with the backend. E.g. The CabinsOverview37 component can now dynamically initialise its list of cabins from the backend within its 'created()' lifecycle hook:

```
async created() {
  this.cabins = await this.cabinsService.asyncFindAll();
  this.selectedCabin = this.findSelectedFromRouteParam(this.$route);
},
```

Notice that we have changed the 'created()' hook into an a-synchronous function. Every function that waits for the result of another a-synchronous function must itself also be coded as an a-synchronous function (such that its caller also can wait for the result).

D. Launch both the Spring-boot backend and the Vue.JS frontend and verify whether the backend-cabins appear in your new frontend Overview37.
It is well possible that you run into a CORS issue:



If your backend REST service is provided from a different port (8086) than your frontend UI site (8080), you must configure your backend to provide Cross Origin Resource Sharing (see https://en.wikipedia.org/wiki/Cross-origin_resource_sharing). For that, add a global configuration class to your backend which implements the WebMvcConfigurer interface. In this class you need to implement addCorsMappings.

```
@Override
public void addCorsMappings(CorsRegistry registry) {
    registry.addMapping("/**")
            .allowedOriginPatterns("http://localhost:*", getHostIPAddressPattern())
```

Make sure that the configuration class is found during the Spring Boot component scan and automatically instantiated.

Alternatively you can explore configuration of a reverse proxy in Vue.JS

If CORS is resolved, you should be able to retrieve and view the backend data in your frontend UI:

E. For full functionality of your Master/Detail editor you should complete the implementation of the four methods in the CabinsAdaptor and use them appropriately in both the Overview37 and Detail37 components.

Some functionality involves special consideration:
1. If a New Cabin is added, the frontend shall generate a new, empty cabin with id=0, and first save it to the backend via cabinsService.asyncSave(newCabin). The backend shall generate and set a new unique id and return the updated cabin in the response. The frontend should have 'awaited' the response and then push the newly saved cabin into the list of cabins in CabinsOverview37 and then select it for display and editing.

2. You may find it a struggle to align date/time formats between HTML5 input fields, JavaScript Date objects and the JSON serialization of backend Java LocalDateTime objects. @JsonFormat(pattern = "yyyy-MM-dd'T'HH:mm:ss.SSS'Z'") can be used to serialize the Java LocalDateTime values into a format that is compatible with the ISOString format of the JavaScript Date class at GMT+00 time zone.

3. CabinsDetail37 no longer takes the selectedCabin as a property from CabinsOverview37 but uses the id-parameter from the route to retrieve itself an up-to-date version of the cabin from the backend. (Use cabinsService.asyncFindById()). CabinsDetail37 shall also directly save and delete cabins at the backend without seeking intermediary involvement of CabinsOverview37.

4. You may find that the list of Overview37 runs out-of-date after updates or deletes by the Detail37 and the user needs to issue a page refresh to update. That can be automated by implementing a refresh event from Detail37 to Overview37.

5. When multiple users are editing cabins from different client devices, their local info may get out of date quickly. For now, manual page refreshes are appropriate to catch up on changes by other users. In a later assignment you will explore the Web Sockets technology to automate server to client notifications of global changes.

Test your implementation, verifying new, save and delete operations, and verify that a page refresh (which reloads your frontend application) is able to retrieve again all data that is still held by the backend.

3.7.2 [BONUS] A generic caching REST Adaptor service

In the previous assignment you have developed a CabinsAdaptor that provides your components with an async/await API for retrieving and updating cabins from the backend.
That implementation misses two objectives:

I. If there are many entity classes involved in your application, you wish to avoid the code duplication across very similar service implementations for each entity class.
II. If two components (master and detail) inject the same instance of the (shared) service, then we would like Vue's change detection mechanism to automatically process the updates to the data by the other component without our need to emit events about that.

The first objective (I) you can solve by implementing a generic RestAdaptor<E> that takes the entity class E as a type parameter. The implementation of that generic adaptor would not need to depend on specific details of the entity, except for three aspects:

1) Every entity is provided by a different resource endpoint url at the backend.
2) Every entity comes with a specific copyConstructor that can be used to create proper object instances from a json representation.
3) Every instance of an entity should have an id. (This id also features as a parameter some of the CRUD methods.)

The first two aspects can be provided to the constructor of the generic adaptor. The third can be achieved by enforcing an Entity interface with the id attribute upon every entity class.

Now, JavaScript is weakly typed, so we do not need to parametrize generics explicitly. Also, ES 2021 does not specify use of interfaces yet, so below code snippet gives an out-line of our generic REST adaptor (ignoring the local helper methods):

```
export class RESTAdaptorWithFetch /* <E> */ {
    resourcesUrl;        // the full url of the backend resource endpoint
    copyConstructor;     // a reference to the copyConstructor of the entity: (E) => E

    constructor(resourcesUrl, copyConstructor) {
        this.resourcesUrl = resourcesUrl;
        this.copyConstructor = copyConstructor;
    }
    asyncFindAll() /* :Promise<E[]> */ { … }
    asyncFindById(id) /* :Promise<E> */ {
        return this.copyConstructor(fetch(`${this.resourcesUrl}/${id}`));
    }
    asyncSave(entity) /* :Promise<E> */  { ... }
    asyncDelete(id) /* :void */  { ... }
}
```

Multiple instances of this adaptor class, each for a different entity, can then be provided from the App component and injected into specific user interface components:

```
provide() {
  return {
    // stateless data services adaptor singletons
    cabinsService: new RestAdaptorWithFetch(BACKEND_URL+"/cabins", Cabin.copyConstructor),
    locationsService: new RestAdaptorWithFetch(BACKEND_URL+"/locations", Location.copyConstructor),
    usersService: new RestAdaptorWithFetch(BACKEND_URL+"/users", User.copyConstructor),
```


A. Generalize your implementation of the CabinsAdaptor into a generic implementation of RESTAdaptorWithFetch along the suggested out-line, which has no specific dependencies on the Cabin class in the implementation.

   Provide your components with this generic implementation.
   Retest your application.

So far you have met the second objective (II) by implementing a refresh event between components that share an adaptor service. (See assignment 3.7.1E.)
An alternative approach is to extend the functionality of the RESTAdaptor into a caching adaptor which maintains and provides a list of entities that have been processed by the CRUD operations:

- All entities retrieved from asyncFindAll are retained into a local cache copy of the adaptor.
- Each entity that is retrieved by asyncFindById is also updated in the local cache.
- Each entity that is saved by asyncSave is also updated in the local cache.
- Each entity that is removed by asyncDeleteById is also removed from the local cache.

The Overview components then can react to changes in the cache of the adaptor and automatically follow updates by any other component that is sharing use of the adaptor.

Below code snippet suggests the outline of a generic, caching REST adaptor:

```
export class CachedRESTAdaptorWithFetch /* <E> */
          extends RESTAdaptorWithFetch /* <E> */  {
    entities;        // the cache of the results of all CRUD operations

    constructor(resourcesUrl, copyConstructor) {
        super(resourcesUrl, copyConstructor);
        this.entities = [];
    }
    asyncFindAll() /* :Promise<E[]> */ {
        this.entities =  await super.asyncFindAll();
        return this.entities;
    }
    asyncFindById(id) /* :Promise<E> */ { ... }
    asyncSave(entity) /* :Promise<E> */  { ... }
    asyncDelete(id) /* :void */  { ... }
```

Now this adaptor is not stateless anymore, and components can only observe the changes in the state of the adaptor if we provide it in ‘reactive mode’. Vue provides a wrapper for that:

```
import { reactive, shallowReactive } from 'vue';
import {CachedRestAdaptorWithFetch} from "@/services/cached-rest-adaptor-with-fetch";

export default {
  name: "App",
  components: {'app-header': Header...},
  provide() {
    {...}
    return {
      // reactive data services adaptor singletons
      cachedCabinsService: reactive(
          new CachedRestAdaptorWithFetch(BACKEND_URL+"/cabins", Cabin.copyConstructor)),
```


In the overview we inject the cachedCabinsService and use a computed property to replace the local cabins array by a reference to the entities cache of the service:

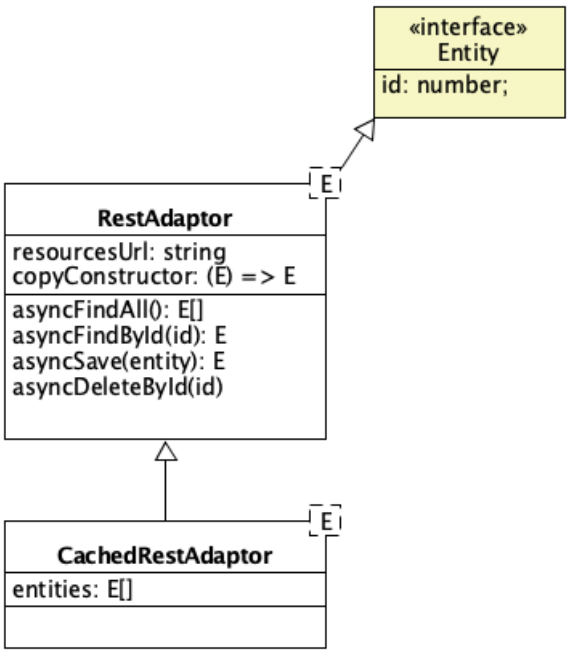
```
export default {
  name: "CabinsOverview37c",
  inject: {
    cabinsService: { from: 'cachedCabinsService' }
  },
  computed: {
    cabins() { return this.cabinsService.entities; }
  },
```

Notice the special inject syntax that allows us to rename the injected service, such that we can continue to reuse the existing code of the UI component.

B. Provide an implementation of CachedRESTAdaptorWithFetch and use it in Overview37c and Detail37c. Overview37c can be copied from Overview37 initially and Detail37c can be extended from Detail37 (to only replace its injected service). Double check, that you have no direct updates to the cabins array anymore in Overview37c, otherwise than by CRUD operations via the cabinsService. (E.g., verify the onNewCabin method and remove the refresh event handler.)

   Create a new route and menu item to test these components.

   Test your application.
   1. Verify that type/location and image updates to cabins also appear in the scroll container.
   2. Verify that deletions are removed from the list.
   3. Verify that new cabins are added to the list



4 Second term assignments: JPA, Authentication and WebSockets

In these assignments you will expand the backend part of the '*Play aan Zee'* application as depicted in the full stack, layered logical architecture of section 2.3. You will implement the Java Persistence API to connect the backend to a relational database. Also, full stack authentication and security will be addressed with JSON Web Tokens.

Relevant introduction and explanation about this technology can be found in O'Reilly-3 at https://learning.oreilly.com/home/

These assignments build upon your full stack solution as you have delivered at the end of assignment 3.7

4.1 JPA and ORM configuration

In this assignment you will configure data persistence in the backend using the Hibernate ORM and the H2 RDBMS. You will implement a repository that leverages the Hibernate EntityManager in transactional mode to ensure data integrity across multiple updates.

By the end of this assignment, you will have implemented the Cabin and Rental classes including its one-to-many relationship. Your REST API can add Rentals to cabins. It will produce error responses on Rentals for cabins that have no availability.

Relevant introductions into the topics you find in O'Reilly-3 chapters 3 and 5.

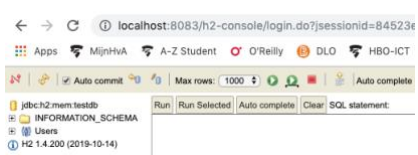
In this assignment you should practice hands-on experience with JPA and Spring-Boot annotations @Entity, @Id, @GeneratedValue @OneToMany @ManyToOne @Repository @PersistenceContext @Primary @Transactional @JsonManagedReference @JsonBackReference and classes EntityManager and TypedQuery.

4.1.1 Configure a JPA Repository

A. First update your pom.xml to include the additional dependencies for 'spring-boot-starter-data-jpa' and 'h2' similar to the demonstration of a project setup in O'Reilly-3.Ch3.
(Do not include the JDBC dependency).
Also enable the H2 console in application.properties, and make sure the logging level is at least 'info' so that Spring shows its configuration parameters when it starts.
Provide spring.jpa.show-sql=true such that you can trace the SQL queries being fired. (Detailed logging can be obtained from logging.level.org.hibernate.type=trace.)

Relaunch the server app, and use the h2-console to check-out that H2 is running.
(Retrieve your proper JDBC URL from the spring start-up log.)
(Spring Boot has auto-configured the H2 data source for you.)
(You do not need to create tables or load data into the database using plain SQL)

```
2019-11-19 13:48:59.911  INFO 92405 --- [           main] com.zaxxer
 .hikari.HikariDataSource       : HikariPool-1 - Start completed.
2019-11-19 13:48:59.919  INFO 92405 --- [           main] o.s.b.a.h2
.H2ConsoleAutoConfiguration    : H2 console available at '/h2-console'.
Database available at 'jdbc:h2:mem:testdb'
```



B. Upgrade your Cabin class to become a JPA entity, identified by its id attribute. Configure the annotations which will drive adequate auto-generation of unique ids by the persistence engine.

Create a new implementation class CabinsRepositoryJpa for your CabinsRepository interface. This new class should get injected an entity manager that provides you with access to the persistence context of the ORM. Use this entity manager to first implement the save method of your new repository. (Other methods will come later.)

Configure transactional mode for all methods of CabinsRepositoryJpa.

If you now run the backend, you might get a NonUniqueBeanDefinitionException, because you may have two implementation classes of the CabinRepository interface: CabinsRepositoryMock and CabinsRepositoryJpa.
(The tutorial on Spring Dependency Injection explains how to fix that with @Primary. Alternatively, you can explore the use of @Qualified.)

Test the creation of a cabin with postman doing a post at localhost:8086/cabins. Inspect the associated SQL statements in the Spring Boot log and use the h2-console to verify whether the cabin ended up in H2.

```
Hibernate: call next value for cabin_ids
Hibernate: insert into cabin (description, image, location_id, num_available,
 price_per_week, type, id) values (?, ?, ?, ?, ?, ?, ?)
```

SELECT * FROM CABIN;

ID	DESCRIPTION	IMAGE	NUM_AVAILABLE	PRICE_PER_WEEK	TYPE	LOCATION_ID
30001	modern SmallDayTime cabin model a.2	SmallDayTime/waz-small1.jpg	20	220.0	SmallDayTime	40001
30002	cosy SmallLodge cabin model e.3	SmallLodge/waz-small.jpg	16	700.0	SmallLodge	40004
30003	spacious SmallLodge cabin model e.1	SmallLodge/waz-small.jpg	10	600.0	SmallLodge	40001

You may want to explore the use of the @Enumerated annotation to drive the format of the registration of the cabin type in the database.

C. Also implement and test the other three methods of your CabinsRepository interface (deleteById, findById and findAll). Use a JPQL named query to implement the findAll method. The use of JPQL is explained in O'Reilly-3.Ch5.Step15, and -.Ch10. In assignment 4.2 you will explore JPQL in full depth. For now you can use the example given here to implement CabinsRepositoryJpa.findAll()

Test your new repository with postman.

```
@Override
public List<Cabin> findAll() {
    TypedQuery<Cabin> query =
            this.entityManager.createQuery(
                    "select c from Cabin c", Cabin.class);
    return query.getResultList();
}
```

D. Inject the cabins repository into your main application class and implement the CommandLineRunner interface (as shown byO'Reilly-3.Ch3.Step6).

From the run() method you can automate the loading of some initial test data during startup of the application.

```
@Override
@Transactional
public void run(String... args) {
    System.out.println("Running CommandLine Startup");
    //this.createInitialUsers();
    //this.createInitialLocations();
    this.createInitialCabins();
```

```
protected void createInitialCabins() {
    // check whether the repo is empty
    List<Cabin> cabins = this.cabinsRepo.findAll();
    if (cabins.size() > 0 ) return;
    System.out.println("Configuring some initial cabins");
    //List<Location> locations = this.locationsRepo.findAll();

    for (int i = 0; i < 6; i++) {
        // create and add a new cabin with semi-random sample data
        Cabin cabin = Cabin.createSampleCabin(0);
        Cabin savedCabin = this.cabinsRepo.save(cabin);

        // TODO maybe some more initial setup later
```

This approach is preferred above the use of SQL scripts in the H2 backend, because the details of the generated H2 SQL schema will change as you progress your Java entities.

This CommandLineRunner initialisation will also work with your Mock repository implementation.

Make sure to configure transactional mode on the command line runner.

4.1.2 Configure a one-to-many relationship

A. Now is the time to introduce a second entity. Make a new entity class ‘models/Rental’ identified by an ‘id’(long) attribute. Also record some of the other relevant information about Rentals:

start and end date of type LocalDate
status: an enumeration of values REQUESTED, APPROVED, DECLINED, PAID, FULFILLED, CANCELLED or BLOCKED.
cost of type double

Each Rental is associated with one Cabin.
A Cabin is associated with many Rentals.
Define the corresponding association attributes in both classes and provide methods to change the associations from either side. (Avoid endless cyclic recursion when automatically maintaining consistency in bi-directional navigability!):

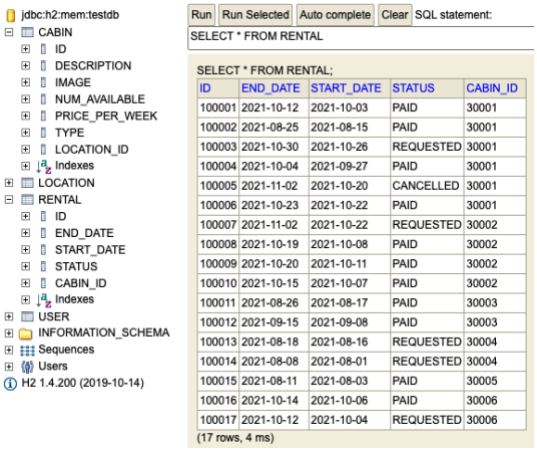
```
/**
 * Associates the given rental with this cabin, if not yet associated
 * @param rental
 * @return  whether a new association has been added
 */
public boolean associateRental(Rental rental) {
/**
 * dissociates the given rental from this cabin, if associated
 * @param rental
 * @return  whether an existing association has been removed
 */
public boolean dissociateRental(Rental rental) {
```

```
/**
 * Associates the given cabin with this rental, if not yet associated
 * @param cabin     provide null to dissociate
 *                  the currently associated cabin
 * @return          whether a new association has been added
 */
public boolean associateCabin(Cabin cabin) {
```

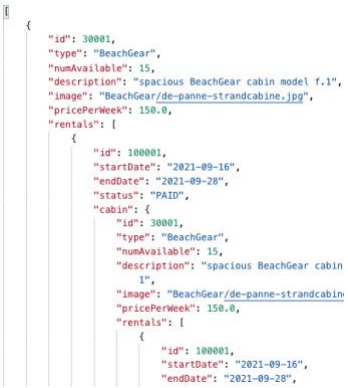
Also provide the JPA @ManyToOne and @OneToMany annotations as explained in O'Reilly-3.Ch8.

B. Implement a RentalsRepositoryJpa like your CabinsRepositoryJpa.
You should not enjoy this kind of code duplication and worry about all the work to come when 10+ more entities need implementation of the Repository Interface.
This may motivate you to implement an approach of the (optional) bonus assignment 4.1.3 and create one, single generalized repository for all your entities.
But, for this course you also may keep it simple and straightforward for now and just replicate the CabinsRepository code….

C. Extend the initialisation in the command-line-runner of task 4.1.1-D to add a few Rentals to every Cabin and save them into the repository.
Consider the JPA life cycle of managed objects within the transactional context in each of the steps of your code:
Make sure that at the end of the method (=transaction) all ('attached') cabins only include 'attached' Rentals.



Test your application and review the database schema in the H2 console. Check its foreign keys and review the contents of the RENTAL table.

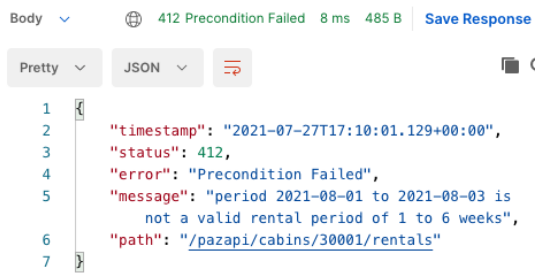


D. Re-test the REST API at localhost:8086/cabins with postman:
You may find a response like here with endless recursion in the JSON structure. For now, investigate the use of @JsonManagedReference and @JsonBackReference to fix that.
With (optional) bonus assignment 4.2.2 you can practice a better solution with custom Json serializers.

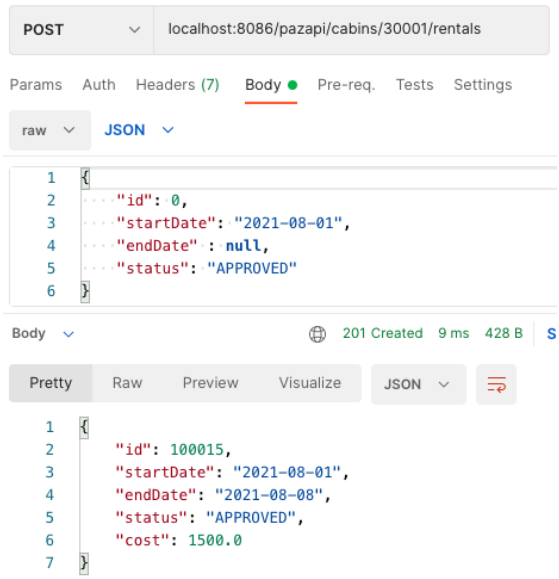
E. Implement a POST mapping on the /cabins/{cabinId}/rentals REST endpoint. This mapping should add a new Rental to the cabin.
The body of the POST request should provide a new rental object which will be further checked and amended by the rest controller:
If body does not provide a start or end date, then a start date of tomorrow or a duration of one week may be used.

A "PreCondition Failed" error response should be returned if:
1) the cabin Id does not match a valid cabin
2) the end date is not after the start date
3) the duration between start date and end date is not a multiple of 7 (whole weeks).

Otherwise, a new Rental shall be created and associated with the cabin.



Test your new endpoint with postman:
Also verify the transactional mode of JPA which will ensure that any saved Rental will be rolled back if an error is raised thereafter.



4.1.3 [BONUS] Generalized Repository

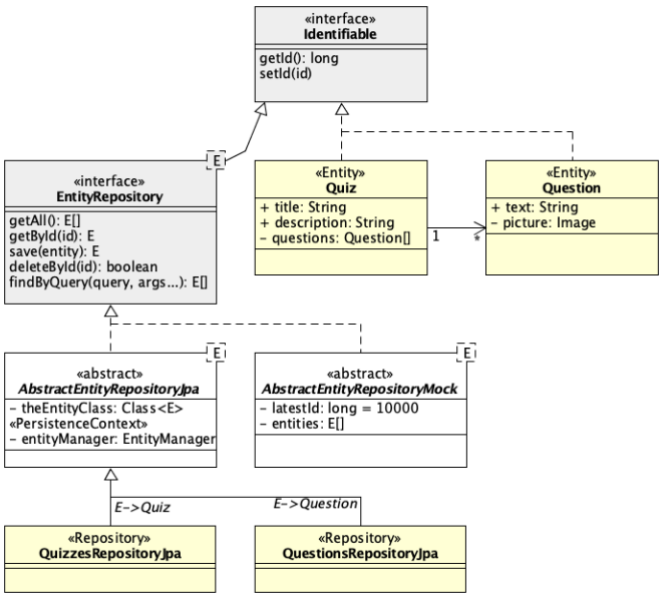
Implementing similar repositories for different entities calls for a generic approach. Actually, Spring already provides a (magical) generic interface JpaRepository<E, ID> and its implementation SimpleJpaRepository<E, ID>. The E generalises the Entity type ID the type of the Identification of the Entity. Such generalisation could provide all repositories that you need.

However, in this course, we require you to implement your own repository for the purpose of learning and developing true understanding. That is no excuse for code duplication though, so we challenge you to develop your own generic repository.

The SimpleJpaRepository<E, ID> class uses the JPA metadata of the annotations to figure out what are the identifying attributes of an entity. That is too complex for the scope of this course.

Also, the generalization of the identification type ID to non-integer datatypes would require custom implementation of unique id generation by the hibernate ORM. We don't want to go there either…

In the class diagram here, you find a specification of a simplified approach of implementing an EntityRepository<E> interface in a generic way, but assuming that all your entities are identified by the 'long' data type.
(Replace in this diagram and in the code snippets below the Quiz entity by your Cabin entity and the Question entity by your Rental entity).



This approach can be realised as follows:

A. Let every entity implement an interface 'Identifiable' providing getId() and setId() Unidentified instances of entities will have id == 0L

```
public interface Identifiable {
    long getId();
    void setId(long id);
```

```
@Entity
public class Quiz implements Identifiable

@Entity
public class Question implements Identifiable
```

B. Specify a generic EntityRepository interface:

```
public interface EntityRepository<E extends Identifiable> {
    List<E> findAll();          // finds all available instances
    E findById(long id);        // finds one instance identified by id
                                // returns null if the instance does not exist
    E save(E entity);           // updates or creates the instance matching entity.getId()
                                 // generates a new unique Id if entity.getId()==0
    boolean deleteById(long id); // deletes the instance identified by entity.getId()
                                 // returns whether an existing instance has been deleted
```


C. Implement once the abstract class AbstractEntityRepositoryJpa with all the repository functionality in a generic way:

```
@Transactional
public abstract class AbstractEntityRepositoryJpa<E extends Identifiable>
        implements EntityRepository<E> {

    @PersistenceContext
    protected EntityManager entityManager;

    private Class<E> theEntityClass;

    public AbstractEntityRepositoryJpa(Class<E> entityClass) {
        this.theEntityClass = entityClass;
        System.out.println("Created " + this.getClass().getName() +
                "<" + this.theEntityClass.getSimpleName() + ">");
    };
```

You will need 'theEntityClass' and its simple name to provide generic implementations of entity manager operations and JPQL queries.

D. Provide for every entity a concrete class for its repository:

```
@Repository("QUIZZES.JPA")
public class QuizzesRepositoryJpa
        extends AbstractEntityRepositoryJpa<Quiz> {

    public QuizzesRepositoryJpa() { super(Quiz.class); };
}
```

E. And inject each repository into the appropriate REST controllers by the type of the generic interface:

```
@Autowired // injects an implementation of QuizzesRepository here.
private EntityRepository<Quiz> quizzesRepo;
```


4.2 JPQL queries and custom JSON serialization

In this assignment you will explore JPQL queries. You will extend the get-mapping of the /cabins endpoint to optionally accept query parameters '?type=XXX' or '?location=XXX' and then filter the list of cabins being returned to meet the specified criteria.

You will also extend the get-mapping of the /cabins/{cabinId}/rentals endpoint to optionally accept two date parameters '?from=yyyy-mm-dd&to=yyyy-mm-dd' by which all rentals of the cabin within the given period are returned.

You will pass the filters as part of a JPQL query to the persistence context, such that only the cabins that meet the criteria will retrieved from the database.

In the bonus assignment you will customize the Json serializer with full and shallow serialisation modes to prevent endless recursion of the serialization on bi-directional navigability between classes.

In this assignment you should practice hands-on experience with Spring-Boot annotations @NamedQuery, @RequestParameter, @DateTimeFormat, @JsonView and @JsonSerialize.

4.2.1 JPQL queries

A. Extend your repository interface(s) and implementations with an additional method 'findByQuery()':

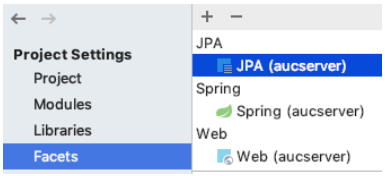
```
List<E> findByQuery(String jpqlName, Object... params);
                                // finds all instances from a named jpql-query
```

(Here we assume you use the generic repository interface)

This method should accept the name of a predefined query which may include specification of (multiple) ordinal (positional) query parameters. At https://www.objectdb.com/java/jpa/query/parameter you find a concise explanation how to go about ordinal query parameters. The implementation of findByQuery should assign each of the provided params[] values to the corresponding query parameter before submitting the query.

B. Design three (named) JPQL queries:
"Cabin_find_by_type": finds all cabins of a given status
"Cabin_find_by_locationName": finds all cabins at a given location
"Rental_find_by_cabinId_and_period": finds all Rentals of a given Cabin within a given period.

Use the @NamedQuery annotation to specify these named JPQL queries within your Cabin.java or Rental.java entity class.
Use an ordinal(positional) query parameters (?1, ?2, etc.) as a place-holder for the parameter values to be provided.



(If you are troubled by a mal-functioning inspection module

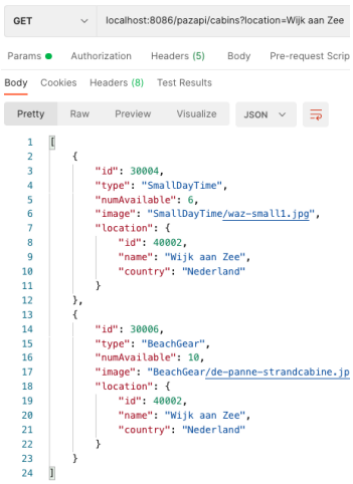
of IntelliJ on your JPQL language, verify whether you have a default JPA facet configuration in your project structure)

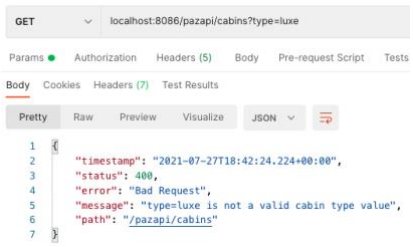
C. Extend your GetMapping on the “/cabins” end-point to optionally accept the ‘?type=XXX’ and ‘?location=XXX’ request parameters.

   If no request parameter is provided, the existing functionality of returning all cabins should be retained.
   If the ‘?type=XXX’ parameter is provided, with a type string value that does not match the Cabin.Type enumeration, an appropriate error response should be returned.
   If an unsupported combination of request parameters is provided, a “Bad Request” error response should be returned with a useful error message.

   In the other cases the requested cabins should be retrieved from the repository, using the appropriate named query and the specified parameter values.
   You may want to explore the impact of the @Enumerated annotation for the status attribute of an cabin.

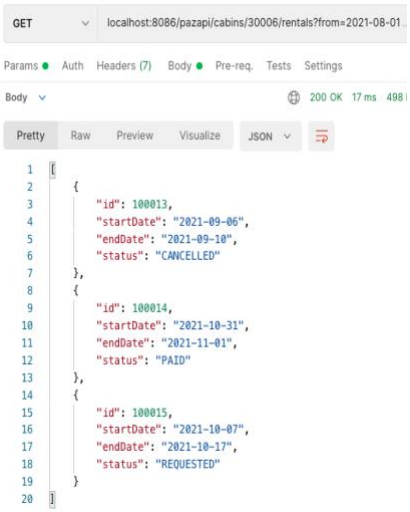
   Test the behaviour of your end-point with postman:





D. Extend the GetMapping at “/cabins/{cabinId}/rentals” with two request parameters ‘from’ and ‘to of type LocalDate.
   Use the @DateTimeFormat annotation to drive deserialization of the values
   Develop a named ‘JPQL’ query with three positional parameters that retrieves these Rentals from the Rentals repository (using findByQuery again).

   Test the behaviour of your endpoint with postman:



4.2.2 [BONUS] Custom JSON Serializers

Bidirectional navigation, and nested entities easily give rise to endless Json structures.

These can be broken by placing @JsonManagedReference, @JsonBackReference or @JsonIgnore annotations at association attributes that should be excluded from the Json. But this is not always acceptable, because depending on the REST resource being queried you may or may not need to include specific info.

Another option is to leverage @JsonView classes, but again these definitions are static and do not recognise the starting point of your query: I.e.:

a) if you query a Cabin, you want full information about the cabin but probably only shallow information about its Rentals.
b) If you query a Rental, you want full information about the Rental, but only shallow information about the Cabin.
c) It gets even more complicated with recursive relations.

At https://www.tutorialspoint.com/jackson_annotations/index.htm you find a tutorial about all Jackson Json annotations that can help you to drive the Json serializer and deserializer by annotations in your model classes.

At https://stackoverflow.com/questions/23260464/how-to-serialize-using-jsonview-with-nested-objects#23264755 you find a nice article about combining @JsonView classes with custom Json serializers that may solve all your challenges with a comprehensive, single generic approach:

A. Below you find a helper class that provides two Json view classes ‘Shallow’ and ‘Summary’ and a custom serializer ‘ShallowSerializer’.

```
public class CustomJson {

    public static class Shallow { }
    public static class Summary extends Shallow { }

    public static class ShallowSerializer extends JsonSerializer<Object> {
        @Override
        public void serialize(Object object, JsonGenerator jsonGenerator,
                              SerializerProvider serializerProvider)
                throws IOException, JsonProcessingException {
            ObjectMapper mapper = new ObjectMapper()
                    .configure(MapperFeature.DEFAULT_VIEW_INCLUSION, false)
                    .setSerializationInclusion(JsonInclude.Include.NON_NULL);

            // fix the serialization of LocalDateTime
            mapper.registerModule(new JavaTimeModule())
                    .configure(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS, false);

            // include the view-class restricted part of the serialization
            mapper.setConfig(mapper.getSerializationConfig()
                    .withView(CustomJson.Shallow.class));

            jsonGenerator.setCodec(mapper);
            jsonGenerator.writeObject(object);
        }
    }
```

The elements of this class are then used as follows to configure the serialization of Cabin.rentals:

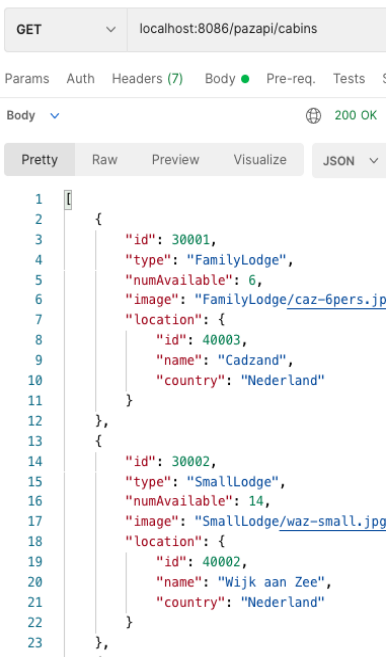
```
@JsonSerialize(using = CustomJson.ShallowSerializer.class)
private Set<Rental> rentals = new HashSet<>();
```

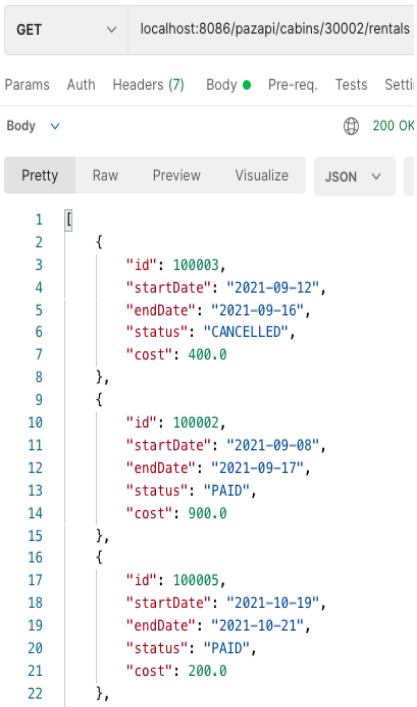

The consequence is:
1) the rentals will only be included in the full serialization of a cabin.
2) when the rentals are serialized (as part of a cabin serialization) its serialization will be shallow (and not recurse back into its own cabin…)

Extend this CustomJson class with a similar implementation of the custom SummarySerializer and the UnrestrictedSerializer internal classes which serialize to Summary view and default view respectively.

B. Apply these view classes and serializers to relevant attributes in the Cabin and Rental model classes.
Apply the Summary view class to the localhost:8086/cabins endpoint.
Implement unrestricted endpoints at localhost:8086/cabins/{cabinId} and localhost:8086/cabins/{cabinId}/rentals

Test your endpoints with postman:





4.3 Backend security configuration, JSON Web Tokens (JWT)

In this assignment you will secure the access to your backend.

The Spring framework includes an extensive security module. However, that module is rather difficult to understand and use at first encounter. For our purpose, we will explore the basic use of JSON Web Tokens (JWT) and implement a security interceptor filter at the backend.

Our backend security configuration involves two components:

1. A REST controller at '/authentication' which provides for user registration and user login.
This endpoint will be 'in-secure', i.e., open to all clients: also, to non-authenticated clients.
After successful login, a security token will be added into the response to the client.
2. A security filter that intercepts all incoming requests.
This filter will extract the security token, if included in the incoming request.
Only requests with a valid security token may pass thru to the secured paths of the REST service.

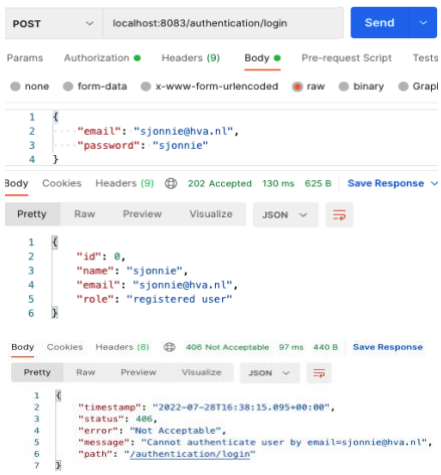
By the end of this assignment you will have further explored annotations @RequestBody, @RequestAttribute, @Value and classes ObjectNode, Jwts, Jws<Claims>, SignatureAlgorithm

4.3.1 The /authentication controller.

A. First create a new REST controller class 'AuthenticationController' in the 'rest' package.
Map the controller onto the '/authentication' endpoint.
Provide a POST mapping at '/authentication/login' which takes two parameters from the request body: email(String) and password(String)

Any request mapping can specify only one @RequestBody parameter.
You may want to import and use the class ObjectNode from com.fasterxml.jackson.databind.node.ObjectNode. It provides a container for holding and accessing any Json object that has been passed via the request body.

Full user account management will be addressed in the bonus assignment 4.7
For now we accept successful login if the provided password is the same as the username before the @ character in the email address.
Throw a new 'NotAcceptableException' if login fails.
Return a new User object with 'Accepted' status after successful login.
(Create a new User entity in your models package. A User should have attributes 'id'(long), 'name'(String), 'email'(String), 'hashedPassword'(String) and 'role' (String). Extract the name from the start of the email address and use a random id).



Test your endpoint with postman:

B. After successful login we want to provide a token to the client. Include the Jackson JWT dependencies into your pom.xml.

```
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-api</artifactId>
    <version>0.11.2</version>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-impl</artifactId>
    <version>0.11.2</version>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-jackson</artifactId>
    <version>0.11.2</version>
    <scope>runtime</scope>
    <exclusions>
        <exclusion>
            <groupId>com.fasterxml.jackson.core</groupId>
            <artifactId>jackson-databind</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

Create a utility class JWToken, which will store all attributes associated with the authentication and authorisation of the user (the 'payload') and implement the functionality to encrypt and decrypt this information into token strings.

Below is example code of how you can encode a JWToken string signing the user's identification and his role.

```
public class JWToken {

    private static final String JWT_CALLNAME_CLAIM = "sub";
    private static final String JWT_USERID_CLAIM = "id";
    private static final String JWT_ROLE_CLAIM = "role";

    public JWToken(String callName, Long userId, String role) {
        this.callName = callName;
        this.userId = userId;
        this.role = role;
    }

    public String encode(String issuer, String passphrase, int expiration) {
        Key key = getKey(passphrase);

        return Jwts.builder()
                .claim(JWT_CALLNAME_CLAIM, this.callName)
                .claim(JWT_USERID_CLAIM, this.userId)
                .claim(JWT_ROLE_CLAIM, this.role)
                .setIssuer(issuer)
                .setIssuedAt(new Date())
                .setExpiration(new Date(System.currentTimeMillis() + expiration * 1000L))
                .signWith(key, SignatureAlgorithm.HS512)
                .compact();
    }

    private static Key getKey(String passphrase) {
        byte[] hmacKey = passphrase.getBytes(StandardCharsets.UTF_8);
        return new SecretKeySpec(hmacKey, SignatureAlgorithm.HS512.getJcaName());
    }
```

The passphrase is the private key to be used for encryption and decryption. You can configure passphrase, issuer and expiration times in the application.properties file and then inject them into your APIConfig bean using the @Value annotation. You may want to amend the passphrase at run-time such that all tokens automatically invalidate once the backend service is restarted.

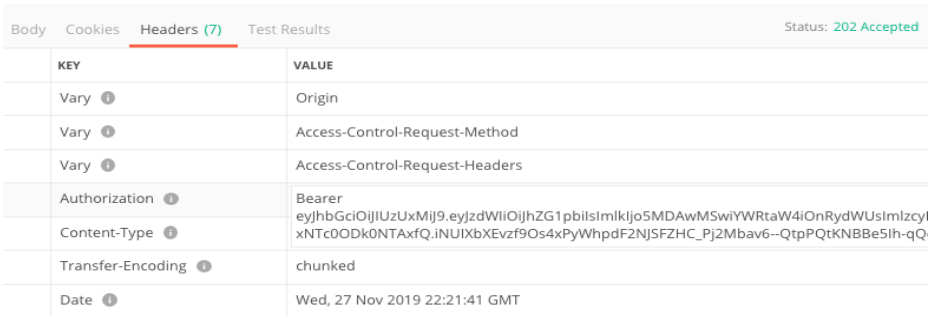
```
// JWT configuration that can be adjusted from application.properties
@Value("HvA")
public String issuer;

@Value("${jwt.pass-phrase:This is very secret information for my private encryption key.
private String passphrase;

@Value("1200") // default 20 minutes;
public int tokenDurationOfValidity;
```

At https://jwt.io/ you can verify your token strings after you have created them.

You add the token to the response of a successful login request with

```
return ResponseEntity.accepted()
        .header(HttpHeaders.AUTHORIZATION, "Bearer " + tokenString)
        .body(user);
```

This puts the token in a special ‘Authorization’ header.
Test with postman whether your authorization header is included in the response:



4.3.2 The request filter.

A. The next step is to implement the processing of the tokens from all incoming requests. If a request does not provide a valid token, then the request should be rejected.

For that you implement a request filter component:

```
@Component
public class JWTRequestFilter extends OncePerRequestFilter {

    @Autowired
    APIConfig apiConfig;

    @Override
    public void doFilterInternal(HttpServletRequest request, HttpServletResponse response,
                                 FilterChain chain) throws IOException, ServletException {

        String servletPath = request.getServletPath();

        // OPTIONS requests and non-secured area should pass through without check
        if (HttpMethod.OPTIONS.matches(request.getMethod()) ||
                this.apiConfig.SECURED_PATHS.stream().noneMatch(servletPath::startsWith)) {

            chain.doFilter(request, response);
            return;
        }
```

Because your filter class is a Spring Boot Component bean, it will be autoconfigured into the filter chain of the Spring Boot http request dispatcher, and hit every incoming http request.

This filter class requires implementation of one mandatory method ‘doFilterInternal’, which does all the filter work.
It is important to let ‘pre-flight’ OPTIONS requests pass through without burden. Your frontend framework may issue these requests without authorisation headers. The Spring framework will handle them.

Also you want to limit the security filtering to the mappings that matter to you. The paths '/authentication', '/h2-console', '/favicon.ico' should not be blocked by any security. In above code snippet we use the set 'SECURED_PATHS' to specify which mappings need to be secured.

Test with postman whether the filter is activated on your SECURED_PATHS and not affecting the other paths.

B. Thereafter we let the filter pick up the token from the 'Authorization' header and decrypt and check it. If the token is missing or not valid, you send an error response and abort further processing of the request:

```
// get the encrypted token string from the authorization request header
String encryptedToken = request.getHeader(HttpHeaders.AUTHORIZATION);

// block the request if no token was found
if (encryptedToken == null) {
    response.sendError(HttpServletResponse.SC_UNAUTHORIZED, "No token provided. You need to logon first.");
    return;
}

// decode the encoded and signed token, after removing optional Bearer prefix
JWToken jwToken = null;
try {
    jwToken = JWToken.decode(encryptedToken.replace("Bearer ", ""), this.apiConfig.getPassphrase());
} catch (RuntimeException e) {
    response.sendError(HttpServletResponse.SC_UNAUTHORIZED, e.getMessage() + " You need to logon first.");
    return;
}
```

Again, the magic is in the use of the Jackson libraries, decoding the token string:

```
public static JWToken decode(String token, String passphrase)
        throws ExpiredJwtException, MalformedJwtException {
    // Validate the token
    Key key = getKey(passphrase);
    Jws<Claims> jws = Jwts.parserBuilder().setSigningKey(key).build()
            .parseClaimsJws(token);
    Claims claims = jws.getBody();

    JWToken jwToken = new JWToken(
            claims.get(JWT_CALLNAME_CLAIM).toString(),
            Long.valueOf(claims.get(JWT_USERID_CLAIM).toString()),
            claims.get(JWT_ROLE_CLAIM).toString()
    );
    jwToken.setIpAddress((String) claims.get(JWT_IPADDRESS_CLAIM));
    return jwToken;
}
```

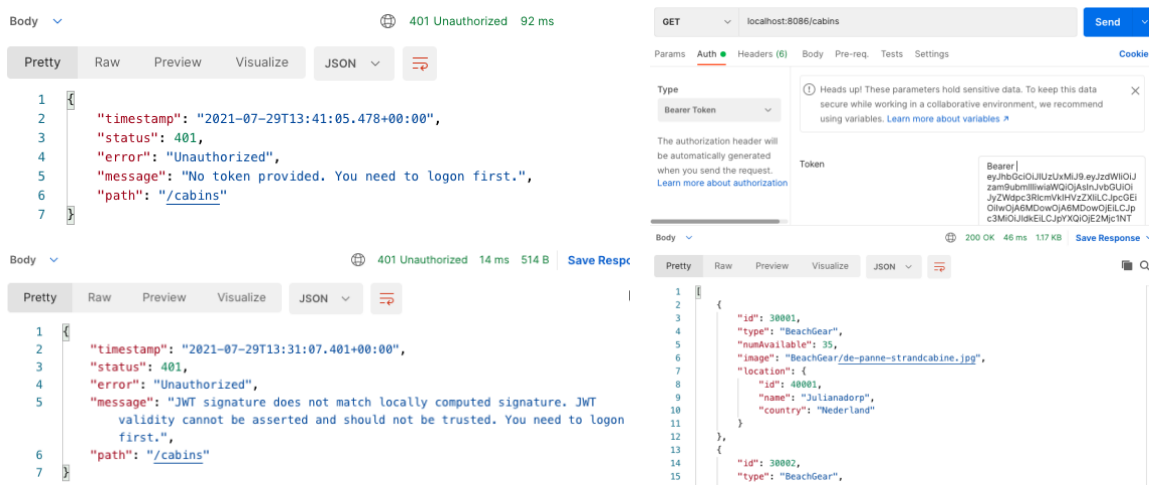
This decode method uses the same JWToken attributes and getKey method that were also shown earlier along with the encode method.

If all has gone well, we add the decoded token information into an attribute of the request and allow the request to progress further down the chain:

```
// pass-on the token info as an attribute for the request
request.setAttribute(JWToken.JWT_ATTRIBUTE_NAME, jwToken);

chain.doFilter(request, response);
```

Later, we can access the token information again from any REST controller by use of the @RequestAttribute annotation in front of a parameter of a mapping method. Then we can actually verify specific authorization requirements of the user that has issued the request.

Test your set-up with postman:
First try a get request without an Authorization Header.
Then try again with a correct header in the request.
Then try with a corrupt token (e.g. append XXX at the end of the token…)



C. Now, all may be working from postman, but it may not yet work cross-origin with your frontend client….

For that you need to further expand your global configuration of Spring Boot CORS to allow sharing of relevant headers and credentials:

```
@Configuration
public class APIConfig implements WebMvcConfigurer {

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/**")
                .allowedOriginPatterns("http://localhost:*")
                .allowedMethods("GET", "POST", "PUT", "DELETE")
                .allowedHeaders(HttpHeaders.AUTHORIZATION, HttpHeaders.CONTENT_TYPE,
                .exposedHeaders(HttpHeaders.AUTHORIZATION, HttpHeaders.CONTENT_TYPE,
                .allowCredentials(true);
    }
```

4.4 Frontend authentication, and Session Management

With a backend that can authenticate users and provide tokens for smooth and authorised access of secured REST endpoints you now can integrate this security into your frontend user interface. You will create a singleton session service in the frontend, that encapsulates the authentication API and tracks the token to be remembered for reuse on subsequent backend requests.

4.4.1 Sign-in and session management.

A. Create a new SessionSbService class that will provide an adapter (interface) to the backend for sign-in and sign-up. This SessionSbService will cache the information of the currently logged-on user, such as the username and the JWT authentication token that will be provided by the backend.

   Here follow some of the methods that you will implement in this service:
   asyncSignIn(email:string, password:string): Promise<User>
   logs on to the backend, and retrieves user details and the JWT authentication token from the backend.
   signOut()
   discards user details and the JWT authentication token from the service.
   saveTokenIntoBrowserStorage(token: string, user: User)
   saves the JWT authentication token and user details into the service and browser storage for automatic retrieval after application or page reload.
   getTokenFromBrowserStorage(): string
   retrieves the JWT authentication token and user details from the browser storage into the service after application or page reload.

   Explore the use of window.sessionStorage and window.browserStorage for retaining tokens and user information also after page reload or browser restart.
   In the constructor of the session service you can initialise the session token with anything left in browser storage by a previous (or parallel) session:

   Several components will require access to this session service, hence we'll instantiate a singleton of the service in a new App44 component and share it with provide/inject. SessionSbService will be stateful: it holds the token and current user information and some components injecting this service want to be able to react to changes in its state. We can achieve that by wrapping the service with a *reactive* proxy:

```
import {SessionSbService} from "@/services/session-sb-service";
import {reactive, shallowReactive} from "vue";
import CONFIG from '../app-config.js'
export default {
  name: "App",
  components: {'app-header': Header...},
  provide() {
    // create a singleton reactive service tracking the authorisation data of the session
    this.theSessionService = shallowReactive(
        new SessionSbService(CONFIG.BACKEND_URL+"/authentication", CONFIG.JWT_STORAGE_ITEM));

    return {
      // stateless data services adaptor singletons
      cabinsService: new CabinsAdaptor(CONFIG.BACKEND_URL+"/cabins"),
      // reactive, stateful services
      sessionService: this.theSessionService,
    }
  }
}
```

B. Next, implement the signIn(email,password) method in the service following the example as given here.
The fetch method needs the option *credentials: "include"* to ensure that access-control headers are included in cross-site requests.
Upon successful authentication by the backend, the JWT token can be extracted from the Authorization header in the response and stored for later reuse.
(At this stage, there is no need to decode the JWT token at the client side.)

```
async asyncSignIn(email, password) /* :Promise<User> */ {
    const body = JSON.stringify({ email: email, password: password });
    let response = await fetch(this.RESOURCES_URL + "/login",
        {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: body,
            credentials: 'include',
        })
    if (response.ok) {
        let user = await response.json();
        this.saveTokenIntoBrowserStorage(
            response.headers.get('Authorization'),
            user);
        return user;
    } else {
        console.log(response)
        return null;
    }
}
```

Also implement the signOut() method (removing the copy of the token from the service and browser storage).

C. Now we can display the name of the logged-on user in the header component just by retrieving that information from the SessionSbService. Create a new component HeaderSb, which can be a copy of the original Header component initially.
Inject the SessionSbService into HeaderSb and include the new header into your new App44 application component which also provides the session service.

Welcome the currently logged-in user in the sub-title in this header at the right. If no user is logged-in, your session service shall provide 'Visitor' as a default username.

D. Also copy your NavBar component to NavBarSb and include the new navigation bar into App44. Create a new component SignIn.vue and connect it to a new '/sign-in' route. This route shall be invoked from the 'Log in' menu item on the new navigation bar.
Provide a form in which the user can enter e-mail and password and add a 'Sign In' button to submit the request.
Use the SessionSbService.signIn method to sign-in the user at the backend and retrieve a token and the user's call name. This call name shall then automatically be displayed in the header as per previous assignment. For convenience you may also show the retrieved token in the SignIn component.:

If logon credentials did not match, the backend should have provided an error response without a valid token. The sign-in component should show a useful error message in that case and no token.

E. Connect the 'Log out' option to the '/sign-out' route. This route shall redirect to the /sign-in route with query parameter signOff=true in the routes table. Amend the SignIn component such that when it finds the signOff query parameter in the route path, it will first call the sessionService.signOff() and then show an empty logon screen.

Inject the session service into the NavBarSb component and adjust the visibility of the menu items in the navigation bar such that
a) Menu items 'Sign Up' and 'Log in' are visible only when no user is logged in yet (and user name Visitor is displayed).
b) Menu item 'Log out' is visible when a user has been logged in.
(Hint: add a method SessionSbService.isAuthenticated() for this.)

4.4.2 Using a Fetch-interceptor to add the token to every request.

Even if the user has been logged on, http requests towards the secured endpoints of the backend will not be accepted yet, because so far, we did not add the JWT authentication token to any request yet. In this assignment you will re-configure the Fetch API to automatically add the JWT token to every outgoing fetch request to inform the backend about the authentication and authorisation status of the user. We will use the fetch-intercept library for that.

A. Add the fetch-intercept library to your project with:

$ npm install fetch-intercept whatwg-fetch --save

At https://www.npmjs.com/package/fetch-intercept you can read the basics of how to use this package, but that doesn't quite show you how to integrate the package cleanly, the S.O.L.I.D. way…

Create a new class FetchInterceptor, which imports and builds on the library. The basic structure of this class should follow:

```
5   export class FetchInterceptor {
6       static theInstance; // the singleton instance that has been registered
7       session;
8       router;
9       unregister; // callback function to unregister this instance at shutdown
10      constructor(session, router) {
11
12          this.session = session;
13          this.router = router;
14          // fetchIntercept does not register the object closure, only the methods as functions
15          // so we need an additional static reference to the singleton's attributes
16          FetchInterceptor.theInstance = this;
17          this.unregister = fetchIntercept.register(this);
18          console.log("FetchInterceptor has been registered; current token = ",
19              FetchInterceptor.theInstance.session.getCurrentToken() );
20      }
21
22      request(url, options) {...}
41      requestError(error) {...}
45      response(response) {...}
53      responseError(error) {...}
```

Via the constructor of the interceptor we pass:
1. The session singleton by which the interceptor can retrieve the current token.
2. The router, because we want to use the interceptor to redirect to an error page whenever an error response is received.

Unfortunately, fetch-intercept registers only the methods as (static) function references within a JavaScript object, without access to the object instance variables from those functions (pre ES6 coding style...) Fortunately, we will only need one singleton instance of FetchInterceptor for the complete application, so we can access that instance explicitly via a static variable FetchInterceptor.theInstance and then access the instance variables from there. (A.k.a. the Singleton Design Pattern).

We will create and manage this singleton instance in the App44 component, just after creation of the SessionSbService (because the interceptor needs access to the token) but before the provision of the data service adaptors (because the interceptor needs to be in place before any component initiates a Fetch).

That goes as follows in the provide section of App44:

```
provide() {
  // create a singleton reactive service tracking the authorisation data of the session
  this.theSessionService = shallowReactive(
      new SessionSbService(CONFIG.BACKEND_URL+"/authentication", CONFIG.JWT_STORAGE_ITEM));
  this.theFetchInterceptor =
      new FetchInterceptor(this.theSessionService, this.$router);
  return {...}
},
unmounted() {
  console.log("App.unmounted() has been called.");
  this.theFetchInterceptor.unregister();
}
```

(Once the App component unmounts, we also unregister the interceptor.)

B. Now you can implement the request hook within the interceptor:

```
request(url, options) {
    let token = FetchInterceptor.theInstance.session.getCurrentToken();

    if (token == null) {
        // no change
        return [url, options];
    } else if (options == null) {
        // the authorisation header is the only custom option
        return [url, { headers: { Authorization: token }}]
    } else {
        // add authorization header to other options
        let newOptions = { ...options };
        newOptions.headers = {
            // TODO combine new Authorization header with existing headers
        }
        return [url, newOptions];
    }
}
```

Re-test your application and verify that visitors cannot change data, while authenticated users can.
Include console.log output in your interceptor and verify that the token is passed along with the request.
Verify that authorization errors are thrown again once the token has expired or after you have rebooted the server with a different passphrase.

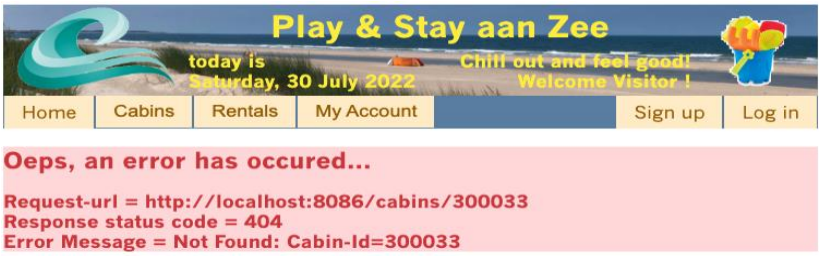
C. Make sure that the user will be redirected to the logon page automatically at any time that a backend request resolved with a 401-Unauthorized-error response.
This you now can achieve within the response hook of the FetchInterceptor.
If you find a response with status 401, just push the router to the '/sign-out' route.

D. [EXTRA] It would be nice if other error responses also were reported properly. For that create a new component RequestError.vue with a property 'message'
Create an '/error' route that will convert its parameters to properties.
(We do not want the error message itself to become part of the URL)

```
{ path: '/error', name: 'ERROR', component: RequestError, props: true },
```

In the response hook of the FetchInterceptor you now format a nice error message and push it as a router parameter into the router on route name 'ERROR'. It will then find its way into the RequestError component as a property.
(See https://router.vuejs.org/guide/essentials/passing-props.html#boolean-mode for further explanation.)
Use the v-html binding in the template to render a message that may contain html tags
 for lay-out purposes:

Test your automated error response handler by navigating to a cabin that does not exist:



4.5 Request rental of a Cabin

In this assignment you will build on all the preparatory work of the previous assignments to implement some of the workflow of renting a cabin:

a) You will create a rental request page, where a user can view the actual availability of cabins and submit a request for rental.

b) You will calculate and retrieve actual availabilities from all rental requests in the system.

The backend will track the and store the rentals, calculate the cost and availabilities, and provide the frontend with up-to-date information. The frontend visualises the availabilities and takes user input and posts http requests to the backend with new requests.

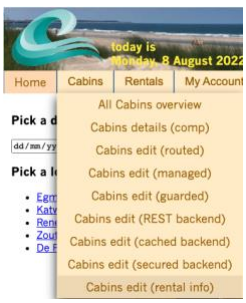
You will explore use of query parameters in the frontend, and the use of the @Transient JPA annotation that allows to add calculated/derived quantities to your entities that are not persisted by the ORM.

4.5.1 Show number of open and confirmed rentals.

First verify that you have a structurally complete backend implementation of the Cabin and Rental entities as per the class diagram of section 2.2. You may omit data attributes until they are required by your code but do verify that all JPA annotations of the relationships are present with proper Json serialization specifiers. Also provide both Repository<Cabin> and Repository<Rental> implementations.

A. Check the setup of your initial data by CommandLineRunner:
1) Initialise at least five Cabin entities each with a maximum availability between 3 and 20.
2) For every cabin: associate at least two Rental entities with (different) start dates in the near future and rental durations between one and five weeks. If all of your entity classes have got implemented the associate methods of section 4.1.2-A) you can robustly associate the rentals with the cabins.
3) Choose between the REQUESTED, APPROVED, PAID, DECLINED and CANCELLED status for each rental.
4) Verify that your JSON serialisation filter of the GetMapping of a specific cabin at /cabins/{id} includes the rentals (See also bonus assignment 4.2.2B.)

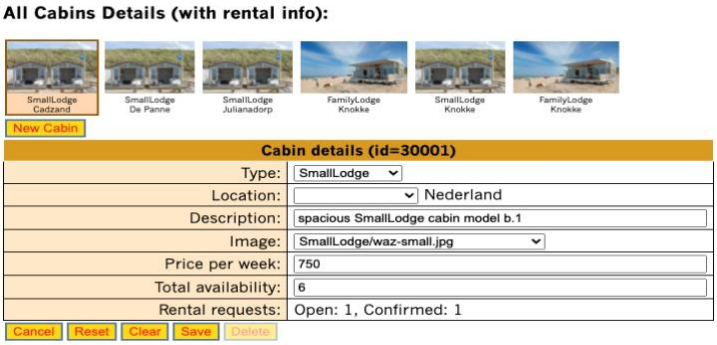
Restart your backend and verify your initial cabins and rentals with postman.



B. Add a new component Detail45, which can be a copy of your latest Detail37 component initially. Create e new route cabins/overview45, which combines Overview37 with your new Detail45 add an entry for this route in the cabin's menu.

Add one row to the details panel with label 'Rental requests:' This row should display the number of open rental requests on the cabin (with status=REQUESTED) and the number of confirmed

rental requests (with status=APPROVED or status=PAID) similar to below picture.



For this you may add two access functions to your frontend Cabin class:

```
getNumOpenRentalRequests() {...}
getNumConfirmedRentalRequests() {...}
```

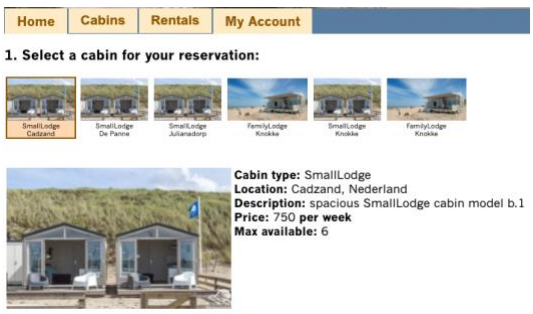
which calculate the numbers directly from the Cabin.rentals array attribute of the editedCabin. (For that it may be relevant that your Cabin.copyConstructor does a deep clone on the rentals as obtained from the json representation that was delivered by the backend.)

No additional AJAX requests should be required for this feature, as your Detail45 component should already have retrieved all info from the cabinsService.findById() request to setup the editedCabin.

4.5.2 Submit new rental requests.

A. Let's create a new page to make requests for rentals:
Add two new components Overview45 and Rental45 in a new folder src/components/rentals. Overview45 can be a copy of Overview37 initially, but its header text shall be amended, and the 'new cabin' functionality shall be removed. Rental45 has the same structure as Detail45. It will be connected to a child route of Overview45 with the selected cabin id as a router parameter. However, it will only show cabin details and not change cabin information.

Here you see two pictures of this setup, with and without a selected cabin.



Also add a route 'rentals/overview45' and a menu item for this functionality in the rentals menu.

B. Extend the Rental45 component to show weekly availabilities of the selected cabin as in the picture below:

The availabilities table shows weekly availability from a given start date onwards for a fixed number of (seven) weeks. (Cabins will be rented for complete weeks only, and all arrival and departures will be on Saturdays.) Add an availabilitiesStartDate instance variable to the Rental45 component which you initialise on the Saturday before the current date. Two arrow buttons increment or decrement this availabilitiesStartDate by one week.

C. The actual availability numbers will be retrieved from the backend.
A straightforward implementation involves a @Transient attribute and a calculation method in the back-end Cabin class as in below code snippet.
The transient attribute will not be persisted in the data layer but will be included in the full JSON serialisation of cabinsService.findById() as delivered to Rental45.

```
@Transient
public int[] availabilities;

/**
 * Calculates actual availability of this cabin for the next n weeks starting at startDate
 * includes all rentals except those with status CANCELLED or DECLINED
 * @param n             number of weeks ahead to calculate
 * @param startDate     the arrival date of the first week (availabilities[0])
 */
public void calculateAvailabilites(int n, LocalDate startDate) {
    if (startDate == null) return;
    this.availabilities = new int[n];
    // initialise all availabilities with the cabin's maximum available
    Arrays.fill(this.availabilities, this.numAvailable);
    // decrease availability of each week with an overlapping rental (request)
    for (Rental r: this.getRentals()) {...}
}
```

The GetMapping in the rest controller of /cabins/{id} shall retrieve the cabin from the repository, and then call calculateAvailabilities before returning the response. For that it needs a start date, which we pass from the frontend in a query parameter: /cabins/{id}?availabilitiesFrom=xxxx-xx-xx. (Use the @RequestParameter annotation in the mapping to include the optional query parameter.)

So, before you can complete the frontend part of this functionality you need to amend your frontend RESTAdaptorWithFetch implementation to support optional query parameters according to below interface/abstract class:

```
export class RESTAdaptorInterface /* <E extends Entity > */ {
    resourcesUrl;        // the full url of the backend resource endpoint
    copyConstructor;     // a reference to the copyConstructor of the entity: (E) => E
    asyncFindAll(queryParams : null = null)    /* :Promise<E[]> */ {...}
                 // returns all entities that match the optional queryParams
    asyncFindById(id, queryParams : null = null)   /* :Promise<E> */ {...}
                 // retrieves the entity with given id, and applies optional queryParams
    asyncSave(entity, queryParams : null = null)   /* :Promise<E> */ {...}
                 // saves the given entity and applies optional queryParams
    asyncDelete(id, queryParams : null = null) {...}
                 // deletes the entity with given id and applies optional queryParams
}
```

You can use the JavaScript class URLSearchParams to build a URL with multiple query parameters as in this code snippet:

```
// appends the query parameters into the url
private fullURL(target: string, queryParams?): string {
    let url = this.resourcesUrl + target;
    if (queryParams != null) {
        let newUrl = new URL(url);
        newUrl.search = new URLSearchParams(queryParams).toString();
        url = newUrl.toString();
    }
    return url;
}
```

Then the Rental45 component can specify the start date for the availabilities in the request for the details of the selected cabin.
And the Rental45 view can show the availabilities directly from the calculated attribute in the selected cabin.

Each time when the availabilitiesStartDate is changed, also a new AJAX request shall be posted to retrieve an updated availabilities array in the selected cabin.

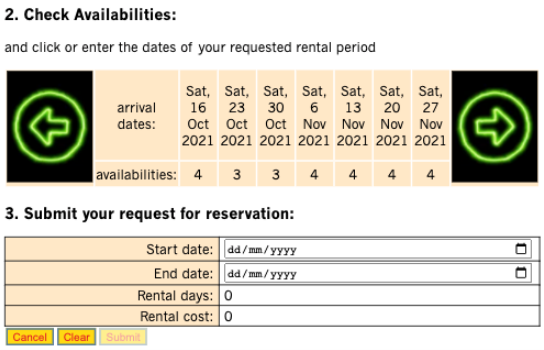
```
private async getSelectedCabin(cabinId, startDate) {
  return this.cabinsService.asyncFindById(
      cabinId,
      { availabilitiesFrom: startDate.toISOString().substr(0,10) }
  );
}
```

Test your implementation for different selected cabins and navigate the availabilities start date with the arrows.

D. Next, we want to submit a new reservation:
Extend the Rental45 component with a rental request panel as in the picture here:

Create an empty Rental instance that has its start and end dates bound to the form by two-way binding.

The date type of the html input element is not fully aligned with the JavaScript Date class, so you may need computed properties with a getter and a setter to bind the start and end date values in the form bi-directionally.



```
computed: {
  startDateUpdater: {
    get() { return this.editedRental.startDate?.toISOString().substring(0, 10); },
    set(localDate) { this.editedRental.startDate = new Date(localDate + "Z"); }
  },
  endDateUpdater: {...}
},
```

The form shall calculate the number of rental days and the total rental cost immediately after each update of a start or end date. (Encapsulate this calculation into two getters of the Rental class)
If a positive number of rental days is calculated, the submit button shall be enabled.
If the user presses submit, the new rental request shall be posted to the backend resource endpoint **/cabins/{cabinId}/rentals**.

Test your application whether you can submit rental requests, and whether subsequent availabilities are decreased accordingly.

Hint: instantiate a new local rentalsService RESTAdaptor within the Rental45 component by extending the cabinsService.resourcesUrl path to the correct endpoint for its Rentals. This rentalsService shall be redefined each time that the router selects another Cabin.

E. [EXTRA] It would be nice if the user could select the rental period just by clicking the availabilities table, and that the selected rental period would be highlighted. This you can achieve with extra event handling and calculation of a selected class in the Rental45 view:
a) If the user clicks a week while no start or end date have been configured in the rental, both the start date and the end date of that week shall be copied into the rental.
b) If the user clicks a week before the current start date of the rental, only the start date of the rental shall be updated (and a multi-week period will be selected).
c) If the user clicks a week after the current end date of the rental, only the end date of the rental shall be updated, extending the rental period uptil and including the clicked week.
d) If the user presses the clear button, both start and end date shall be cleared.

2. Check Availabilities:

and click or enter the dates of your requested rental period

	arrival dates:	Sat, 16 Oct 2021	Sat, 23 Oct 2021	Sat, 30 Oct 2021	Sat, 6 Nov 2021	Sat, 13 Nov 2021	Sat, 20 Nov 2021	Sat, 27 Nov 2021	
	availabilities:	4	3	3	4	4	4	4	

3. Submit your request for reservation:

Start date:	30/10/2021 Saturday, 30 October 2021
End date:	20/11/2021 Saturday, 20 November 2021
Rental days:	21
Rental cost:	4500

Cancel Clear Submit

F. [EXTRA] It would be nice if a confirmation message would be displayed after each submission of a rental request:

In the above solution, Rental45 awaits the response of the submission post to include the new rental ID into the message. Thereafter the message is emitted to the overview and the cabin is unselected. The message is shown by the Overview45 component just below the selection container such that the message will also be visible still when no Cabin is selected anymore.

G. [EXTRA] It would be nice if the availabilities table would automatically adjust its start state if the user entered a rental start date that is not within the window of availabilities. That way the user can use the date picker to choose a date in the far future or past and does not have to click many arrows to find a period of interest. It should not be too difficult to address this in the event handler of the input field.

4.6 [BONUS] Change notification with WebSockets

One shortcoming of the solution of the previous assignment is that users will not be made aware of any changes in the backend made by other users. You must frequently initiate a manual page refresh to get to the latest information about actual availabilities. That is not very convenient, and certainly not acceptable to enterprise applications that involve real-time collaboration of its users.

In this assignment you implement push notifications based on web sockets, to maintain consistency between the state of the user interface and the backend:

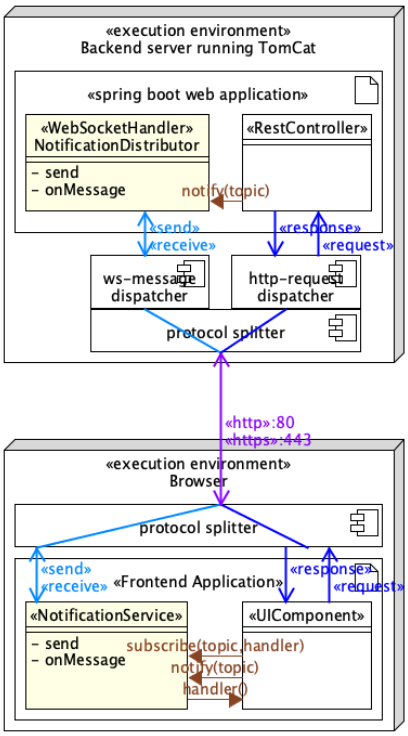
- You will ensure that the user interface of a user who is preparing a rental request will be updated automatically once the relevant availabilities have changed at the backend.

A pragmatic but robust approach to address these (and more similar) features is to implement a small notification framework that allows user interface components to subscribe to 'topics' which identify areas of change in the global state of the system. E.g.:

Topic **"cabin-rentals-12345"** identifies change in the rentals of cabin with id=12345.

There is no fixed list of such topics, and the kinds of change that they represent are not explicitly maintained. You can invent and consistently use as many topics as is required to implement purposeful notification.

The UML deployment diagram here shows the architecture of our solution approach: UI components can subscribe a call-back (handler) function for topics they want to be notified about (and unsubscribe afterwards). Both server RestControllers and client UIComponents can notify all subscribers of a topic. These notifications will not carry any data. They are just 'pings' indicating that something within the topic has changed. It is up to the handlers to follow up with appropriate http requests to obtain up-to-date information. (The backend has no need to subscribe itself, because it is on top of centralised processing of all changes.)



This diagram also shows how browsers and application servers multiplex the bi-directional WebSocket protocol with the request/response http protocol across one single TCP/IP communication port. They recognize WebSocket messages by a special header in a http communication packet.

You seek a fine balance between coarse grain and fine grain topics. Too big topics will be subscribed by many stakeholders and thereby incur more often redundant notification traffic. Too small topics incur more complex code with multiple subscriptions per component and thereby also more network traffic.

At https://www.devglan.com/spring-boot/spring-websocket-integration-example-without-stomp you find a basic tutorial about integrating WebSocket notification in a Spring boot

application without introducing (unnecessary, confusing) protocol layers.

4.6.1 Backend notification distributor.

At the server-side we only need one singleton instance of a 'NotificationDistributor' component, which tracks the active subscriptions of clients and distributes notifications from clients and servlets towards their subscribers.

A. Below you find an example outline code snippet of such distributor.

```
@Component
public class NotificationDistributor extends TextWebSocketHandler {
    // a map of subscribing client sessions per topic
    Map<String, Set<WebSocketSession>> sessionsMap = new HashMap<>();

    @Override
    public void afterConnectionEstablished(WebSocketSession session) {...}
    @Override
    public void handleTransportError(WebSocketSession session, Throwable throwable) {...}

    @Override
    public void handleTextMessage(WebSocketSession fromSession, TextMessage message) {
        String command = message.getPayload();
        if (command.startsWith("notify ")) {
            this.notify(command.split(" ")[1], fromSession);
        } else if (command.startsWith("subscribe ")) {
            {...}
        } else if (command.startsWith("unsubscribe ")) {
            {...}
        } else {
            SLF4J.LOGGER.error("Unsupported message '{}' from {}", command, fromSession);
        }
    }

    public void notify(String topic, WebSocketSession fromSession) {
        // get all sessions that have subscribed to the given topic
        Set<WebSocketSession> sessions = this.sessionsMap.get(topic);
        if (sessions == null) return;
        if (sessions.size() == 0) { // clean-up empty topic
            this.sessionsMap.remove(topic);
            return;
        }

        // TODO send a notification message to every subscribed session
        //  except the fromSession, if any, which has initiated the notification
        //  also clean-up from the map any sessions that throw errors (lost connection)
        {...}
    }

    public void notify(String topic) { this.notify(topic, null); }
}
```

Every session corresponds to an active WebSocket connection between a client application instance in a browser and the server. There will be one such connection per active user session (=browser tab, =application instance) on your client computer. The sessionsMap registers for each 'topic' to which user session instances the notifications for that topic should be re-distributed. The server does not know about individual user interface components in the client application instance. That further detail of distribution will be handled in the client NotificationService adaptor code later.

The notify method is the actual distributer of the topic notification. Notify can be invoked both from 'handleTextMessage' which receives incoming notifications from client components, and directly from servlet mappings which can inject the singleton

instance of NotificationDistributor.

Implement yourselves updates to sessionsMap from incoming ‘subscribe’ and ‘unsubscribe’ messages and complete the message distribution and sessionMap clean-up in the notify method.
(afterConnectionEstablished and handleTransportError should only log some info)

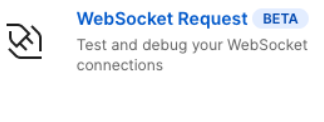
B. A specific maven dependency and spring configuration bean is required to set up your notification distributor as part of the component scan at Spring boot application startup:

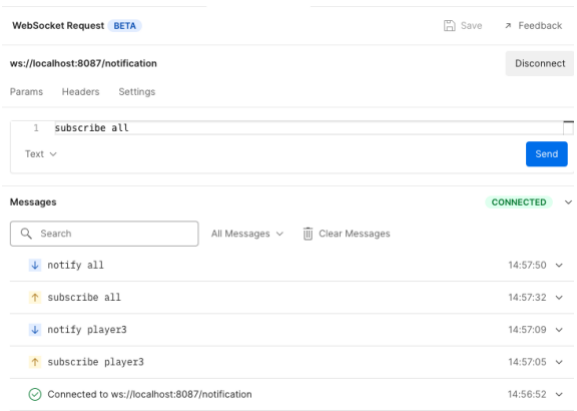
```
@Configuration
@EnableWebSocket
public class NotificationConfig implements WebSocketConfigurer {

    @Autowired
    private NotificationDistributor notificationDistributor;

    @Override
    public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
        registry.addHandler(this.notificationDistributor, "/notification")
                .setAllowedOriginPatterns("http://localhost:*", getHostIPAddressPattern(), "http://*.hva.nl:*")
                //.withSockJS()
        ;
    }
}
```

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-websocket</artifactId>
</dependency>
```

Don’t forget the @EnableWebSocket annotation above this configuration class.
WebSocket communication follows CORS rules like REST requests.
We use the native ‘ws:’ web-socket protocol when creating the sockets in the frontend client. Alternatively, you can use the (polyfill) SockJS class. Then you need to also install sockjs-client and @types/sockjs-client in the frontend and add the ‘.withSockJS()’ modifier to the server handler registration here.

C. Test your notification service with postman:
Open three WebSocket request panels
(New → WebSocket Request)
Connect each to http://localhost:8086/notification
Enter ‘subscribe all’ in each of them.
Enter ‘notify all’ in each panel, one at the time, and verify that this notification is distributed to every other pane.
Enter ‘notify x’ in a panel and verify that no redistribution happens because none have registered for x.
Enter ‘unsubscribe all’ in one panel and verify that ‘all’ notification is not redistributed to this panel anymore.





4.6.2 Frontend notification adaptor.

At the frontend we will use only one singleton instance of a 'NotificationAdaptor' service, which can track subscriptions to topics of client-side handlers and distribute notifications by local client components and from the backend notification distributor.

The implementation may appear a bit complicated, but the advantage is that only one Web socket per user tab in the browser will be created, and the UI developer does not need to worry about closing Web sockets that are not in use anymore. Also, the delay from the connection handshake of a socket will be once-off for the whole application session.
The seemingly simpler implementation of setting up a separate socket for each topic subscription in each UI-component may incur excessive leakage of memory resources at the server if the UI components are not properly closing their sockets at destruction. It also would slow down initial loading of a UI component, waiting for the handshake of setting up the Web Socket.

A. Below you find an outline code snippet of such an adaptor:

```
2   //import SockJS from "sockjs-client";
3   export class NotificationAdaptor {
4       socketUrl;          // socket-url
5       _newSocket = null;  // new socket pending handshake of connection
6       connection = null;  // hand shaken, open connection
7       handlersMap;        // <string, ((string) => void)[]>
8
9       constructor(socketUrl) {
10          this.socketUrl = socketUrl;
11          this.handlersMap = new Map();
12
13          this.socketUrl = socketUrl.replace("http://","ws://");
14          this._newSocket = new WebSocket(this.socketUrl);
15          //this.socketUrl = socketUrl.replace("ws://","http://");
16          //this._newSocket = new SockJS(this.socketUrl);
17
18          const theAdaptor = this;
19          theAdaptor._newSocket.onopen = function(event) {...}
24          theAdaptor._newSocket.onerror = function(error) {...}
27          theAdaptor._newSocket.onmessage = function(e) {
28              let msg = e.data.split(' ');
29              if (msg[0] === "notify") {
30                  console.log(`Received notification for target ${msg[1]}`)
31                  theAdaptor.distributeNotification(msg[1]);
32              }
33          }
34          theAdaptor._newSocket.onclose = function(reason) {...}
38
39          console.log(`Created notification adaptor for ${this.socketUrl}...`);
40      }
41
42      subscribe(topic, handler) {...}
60
61      unsubscribe(topic, handler) {...}
78
79      notify(topic) {...}
83
84      distributeNotification(topic) {
85          let handlers = this.handlersMap.get(topic.toString());
86          if (handlers != null) {
87              for (let handler of handlers) {
88                  handler(topic);
89              }
90          } else {
91              console.log(`WARNING obsolete notification for ${topic}`)
92          }
93      }
```

The handlersMap keeps a registry of all subscriptions from the local user interface components, grouped by topic. If a component subscribes to a new topic, we also

need to send a ‘subscribe topic’ message along the socket towards the backend, such that any remote notifications will also be passed to us. But additional subscriptions to the same topic need not to be registered at the server again, because the backend service only registers subscriptions against our socket (session) and does not know about different components in our user interface. Instead, we track those local handlers in our local handlersMap, and let the distributeNotification method replicate the incoming notifications locally.

The notify method can be invoked by any user interface component and shall pass on the notification both to the backend and to local distributeNotification, because the backend will not echo the notification back along the same socket. (Good for speed of local notification and reduced network traffic.)

Notice: theAdaptor local const is required to use the adaptor instance context within the callback function. The ‘this’ reference would get a different value within such function. (This is legacy semantics of the JavaScript language.)

Be aware: unfortunately, the WebSocket and SockJS constructors deliver a socket with a pending connection. Once the connection is established, the ‘onopen’ callback is invoked. If your code sends (subscribe) messages before ‘onopen’ has been called, you get an error. It is quite a challenge to cope with this robustly: You can try awaiting async creation during application startup, or queue-up any message send while the socket is pending, and flush that queue in the ‘onopen’ callback.
You can avoid the problem without special code, by ensuring that every user interface component awaits some data retrieval before subscribing to anything.

Finish the implementation of your NotificationAdaptor and make sure to add some console.log debug output…

B. Create a new App46.vue component that will instantiate and provide the notificationService: (don’t forget to invoke this component from main.js)

```
28  import CONFIG from '../app-config.js'
29  import {NotificationAdaptor} from "@/services/notification-adaptor";
30  export default {
31    name: "App",
32    components: {'app-header': Header...},
36    provide() {
37  ...
42      this.theNotificationService =
43          new NotificationAdaptor(CONFIG.BACKEND_URL+'/notification');
44      return {
45  ...
59        // non-reactive, state-full notification services
60        notificationService: this.theNotificationService,
61      }
62    },
63    unmounted() {
64      this.theNotificationService.disconnect();
65  ...
66    }
67  }
```


C. Create a new Rental46 component which can be an extension of Rental45.
Create a new route ‘rentals/overview46’ with parent component RentalOverview45 and child component RentalDetail46 and link the route to a new menu item.

If the user has the Rental46 UI component open in the browser, that user wants an instant update of the cabin’s availability information as soon as something changes in the backend concerning the selected cabin. So, each time that another cabin is selected in Rental46, the component shall subscribe for notification about change in the rentals of that cabin (and unsubscribe from the previously selected cabin). If you have properly structured your code of Rental45, you only need to override the reload(cabinId) method of 4.5.2 for that:

```
export default {
  name: "RentalsDetail46",
  extends: RentalsDetail45,
  inject: ['notificationService'],
  methods: {
    async reLoad(cabinId) {
...
      if (this.selectedCabinFromRoute?.id != cabinId) {
        // unsubscribe from notification for the previously selected cabin, if any
        this.notificationService.unsubscribeAll(this.refresh)
        // subscribe to notification for the newly selected cabin
        this.notificationService.subscribe("rentals-cabin-"+cabinId, this.refresh);
      }
      // get the selected cabin details including its availabilities info
      this.selectedCabinFromRoute =
          await this.getSelectedCabin(Number(cabinId), this.availabilitiesStartDate);
      // initialise an empty rental request
      this.editedRental = new Rental(0, this.selectedCabinFromRoute);
      // configure a new rentals REST adaptor at resource end point /cabins/{cabinId}/rentals
      this.rentalsService = new RESTAdaptorWithFetch(
          this.cabinsService.resourcesUrl + "/" + this.selectedCabinFromRoute?.id + "/rentals",
          Rental.copyConstructor);
    },

    async refresh() {
      // TODO reload cabin + availability info from the backend
...
    },

    beforeUnmount() {
      // TODO unsubscribe from notifications if the component is unmounted
...
    }
  }
}
```

Here Rental46 registers a local handler method ‘this.refresh’ that shall be called when a push notification on topic “rentals-cabin-NNN” is received (with NNN equal to the id of the selected cabin). This refresh method shall then retrieve an up-to-date copy of the cabin from the REST endpoint in backend, including updated availabilities.
Make sure you do not lose the local information of the new rental that your user may be editing when refreshing the cabin and availability information because of a notification.

The beforeUnmount lifecycle hook will be called just before the disposal of the component (when the user has pressed cancel or navigated away otherwise). In this hook you shall unsubscribe your notification handler, otherwise future notifications will cause your handler to throw an error because the data context of its component instance (this) will be gone…

D. Actual notifications can best be initiated by the backend, each time when rentals of a cabin are modified. The proper location for this is the REST controller. (Notification is workflow related.)
Inject the NotificationDistributor in the CabinsController, and then use something like the code snippet here to issue a notification each time when a new rental is added to a cabin. Similar code may be required when a rental of a cabin is updated.

```
rental.setId(0L);
Rental newRental = rentalsRepo.save(rental);
newRental.associateCabin(cabin);
newRental.recalculateCost();

URI location = ServletUriComponentsBuilder
        .fromCurrentRequest().path("/" + cabinId + "/rentals")
        .buildAndExpand(newRental.getId())
        .toUri();

this.notificationDistributor.notify("rentals-cabin-" + cabinId);

return ResponseEntity.created(location)
        .body(newRental);
```

Test your notification implementation by opening two browser windows to your application. Navigate both to rentals/overview46 and select the same cabin in both. Now submit a rental request in one tab and observe how the availabilities are immediately updated in the other tab without a manual page refresh.

	arrival dates:	Sat, 16 Oct 2021	Sat, 23 Oct 2021	Sat, 30 Oct 2021	Sat, 6 Nov 2021	Sat, 13 Nov 2021	Sat, 20 Nov 2021	Sat, 27 Nov 2021	
	availabilities:	3	3	1	1	1	3	3	

Submitted reservation id=100013 for rental of a LargeLodge cabin at Wijk aan Zee with arrival on Sat, 23 Oct 2021

	arrival dates:	Sat, 16 Oct 2021	Sat, 23 Oct 2021	Sat, 30 Oct 2021	Sat, 6 Nov 2021	Sat, 13 Nov 2021	Sat, 20 Nov 2021	Sat, 27 Nov 2021	
	availabilities:	3	3	1	1	1	3	3	

	arrival dates:	Sat, 16 Oct 2021	Sat, 23 Oct 2021	Sat, 30 Oct 2021	Sat, 6 Nov 2021	Sat, 13 Nov 2021	Sat, 20 Nov 2021	Sat, 27 Nov 2021	
	availabilities:	3	2	0	0	1	3	3	

Inspect the console for any log messages about the notifications.
Verify that no notifications arrive anymore when the user has navigated to a different page.

4.7 [BONUS] Full User Account Management.

A. Implement a SignUp page behind the corresponding menu item on the menu bar.
New users shall provide:
- email account
- call name (user name for on screen address)
- password

B. Implement the ‘authentication/signup’ endpoint to register a new user account.
Add an ‘active’ (boolean) attribute to the User entity; newly registered accounts are in-active initially.
Add an ‘activationCode’ (string) attribute to the User entity; new, inactive accounts get a unique activation code.
Add an ‘expiration’ (DateTime) attribute to the User entity; the activationCode of a new account has restricted validity (e.g. until 30 minutes after signup.)

Send an email to the new user that refers to
<serverpath>/authentication/activate/{activationCode}

Signup will be refused if the e-mail address is in use by another active user entity already. If signup is repeated on an inactive account, a new activation code is generated, with a new expiration time, and a new e-mail is sent.
(To prevent un-intended e-mail bursts, you may want to include a protection that no activation codes can be generated and send within the first five minutes after previous signup).

C. Implement the activate mapping:
1. Use a named query to retrieve the User entity by activation code from the repository.
2. Check the expiration date/Time and activate the account if ok.
3. Redirect the user to the /signin page of the application.

D. Implement a ‘/users’ end-point where users can retrieve and update their User profiles.
admin users can get and update all attributes of all accounts.
Regular users can only retrieve the info of their own account, and cannot update their active or role attributes.
Use the userId and role in the Authorization token to identify and authorize the requesting user.

E. Activate the MySQL data source in the backend.
Let the commandRunner only add sample data to the database, if no data is available yet in the users or games repositories.