2000WEF_19
2000DWEF19
Web Frameworks (vt, dt)

Assignment
The Auctioneer
(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
19.25	2 Dec 2019	John Somers	Spring-Boot assignments up to 4.5
20.3	2 Nov 2020	John Somers	Simplification, Content shuffle Upgrade to Angular 10 Revisited 3.5 – 3.7, 4.1 – 4.5 Siblings, PUT, references
21.1	13 Dec 2021	Marcio Fuckner	Upgrade to Angular 12 Replaced 4.5
21.2	17 Dec 2021	Jur van Oerle	Included 4.6 Web Sockets
21.3	22 Jan 2022	John Somers	Revisited Clean Architecture Eliminated selectedOfferId in 3.3, 3.4
22.1	24 Aug 2022	John Somers	Migrated from Angular to Vue

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 some parts of the solution of the use case. Most 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 'The Auctioneer'. This application provides its users a platform at which they can buy goods and offer goods for sale by auction.

2.1 Use cases

Visitors can search and navigate the site anonymously to see whether there is anything of their interest. Only registered users can actually post a bid or offer something for sale. The auction of an item proceeds along the following workflow:

1. The seller posts an advertisement for an item, providing a description, possibly some pictures and a final date-time of the closure of the auction.
2. When the seller is happy about the advertisement, the item is published and visible for visitors.
3. A logged-on visitor, who is interested in the item posts a bid, which is higher than the previous bid on the item. When overbid, the same visitor can rebid. Bidders can also withdraw their bid. Sellers can also withdraw their offer. Sellers cannot bid on their own offers.
4. At the date-time of closure of the auction, the highest bidder is awarded the item, and a payment request goes out. The highest bidder can still withdraw his bid, at which case the next highest bidder is awarded the item and another payment request goes out. Sellers cannot withdraw anymore at this stage.
5. If the payment has been made, the delivery is initiated.
6. If the buyer has confirmed receipt of the delivery, the money is transferred to the seller and the auction workflow is closed.

Offered items are organised in Categories. E.g. ‘Antiques’, ‘Cars’, ‘Books’ could be categories available to sellers to register items. Categories can have sub-categories, sub-categories can have sub-sub-categories etc. Items on offer can be registered in multiple categories. Visitors can filter on categories when they search for items.

Normal accounts can only change offers and bids which they own. Sellers can also remove bids on their offers that appear not trustworthy. Some accounts have administrator privileges. Those accounts have update access to all data in the system. They can intervene with the standard auction workflow of all items and they also manage the Categories. They can set and withdraw administrator privileges on other accounts.

2.2 Class diagram

Below you find a navigable class diagram of the functional model that has been designed for ‘The Auctioneer’. This diagram only includes the main entities, attributes and some operations. For a full implementation additional classes, attributes or operations may be required. That is up to you to resolve!

The Offer class registers all items on offer. The auctionStatus attribute of an offer tracks the status of the workflow of an offer, as it is described above.

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

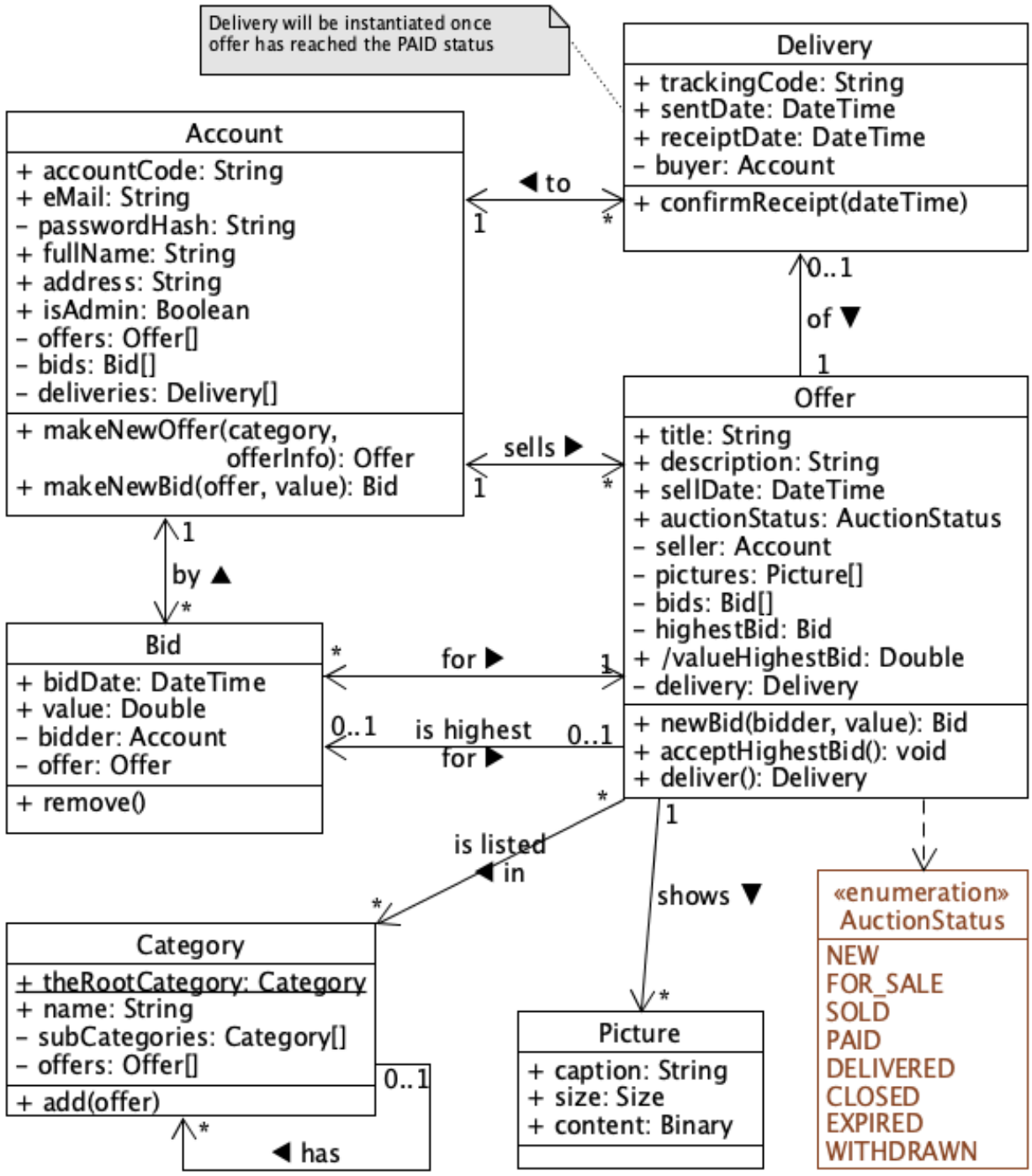
The valueHighestBid attribute is a derived attribute which can be calculated from all Bids made on an offer.

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

- Bi-directional navigability: Every Account knows which Offers it has made and for every Offer we know the seller Account.
- Uni-directional navigability: Every Offer knows in which Categories it is listed, but a Category does not have knowledge of all its Offers.

This design of navigability is based on expected requirements from the use case scenario’s above. From a data modelling perspective, you may find some redundancy in the navigability, but those will prove beneficial from an implementation performance perspective. It may be that your specific implementation approach requires additional or different relations or navigability. In any case: good software design aspires high cohesion and low coupling!

Navigability is implemented with (private) association attributes. E.g. Account.offers realises the navigability of the ‘sells’ relation. It provides the list of all Offers that have been proposed for sale by a given Account. Offer.seller realises the navigability the other way around. It provides the seller Account of a given Offer.


Delivery will be instantiated once offer has reached the PAID status
Delivery
+ trackingCode: String
+ sentDate: DateTime
+ receiptDate: DateTime
- buyer: Account
+ confirmReceipt(dateTime)
Account
+ accountCode: String
+ eMail: String
- passwordHash: String
+ fullName: String
+ address: String
+ isAdmin: Boolean
- offers: Offer[]
- bids: Bid[]
- deliveries: Delivery[]
+ makeNewOffer(category, offerInfo): Offer
+ makeNewBid(offer, value): Bid
to
1
*
0..1
of
1
Offer
+ title: String
+ description: String
+ sellDate: DateTime
+ auctionStatus: AuctionStatus
- seller: Account
- pictures: Picture[]
- bids: Bid[]
- highestBid: Bid
+ /valueHighestBid: Double
- delivery: Delivery
+ newBid(bidder, value): Bid
+ acceptHighestBid(): void
+ deliver(): Delivery
sells
1
*
1
by
*
Bid
+ bidDate: DateTime
+ value: Double
- bidder: Account
- offer: Offer
+ remove()
*
for
1
0..1
is highest for
0..1
*
is listed in
*
1
shows
*
Category
+ theRootCategory: Category
+ name: String
- subCategories: Category[]
- offers: Offer[]
+ add(offer)
0..1
*
has
Picture
+ caption: String
+ size: Size
+ content: Binary
«enumeration»
AuctionStatus
NEW
FOR_SALE
SOLD
PAID
DELIVERED
CLOSED
EXPIRED
WITHDRAWN

2.3 Layered Logical Architecture

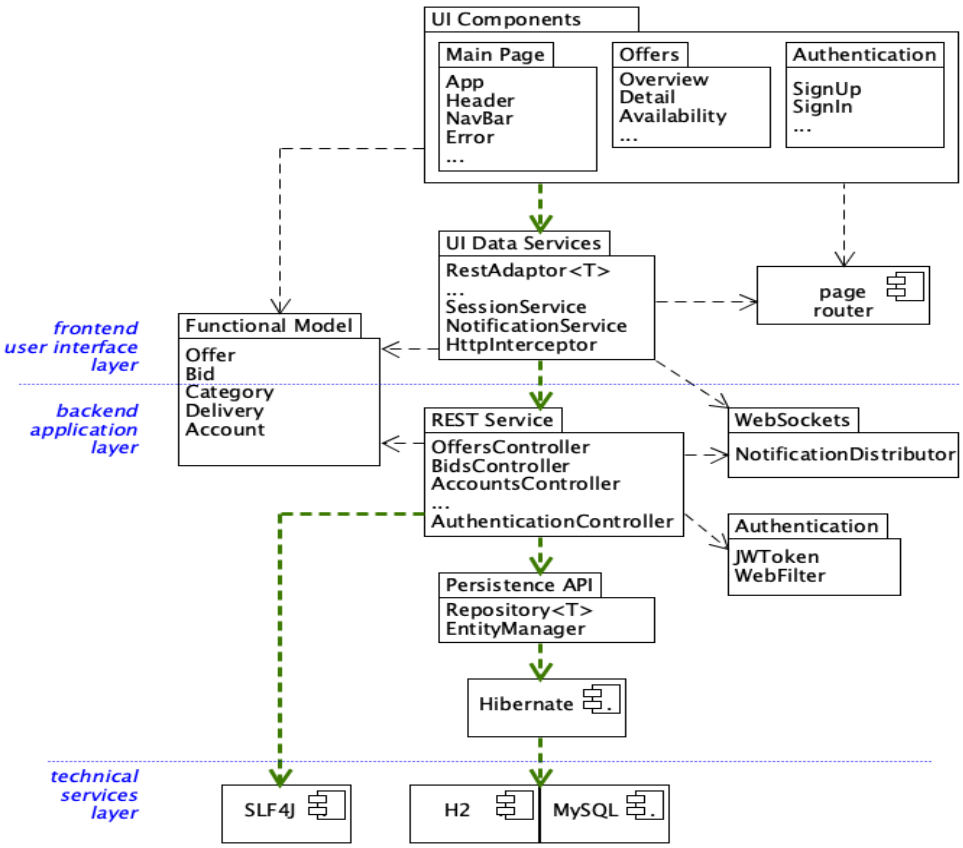
The diagram below depicts the designated full-stack, layered architecture of the 'the Auctioneer' application. The frontend user interface layer follows the Model-View-Controller pattern of a Single-Page web application. The UI Components packages provides the View-Controller structures. The UI Data 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 Angular 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), a.o.:
   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:

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

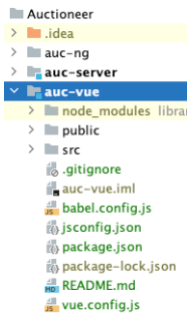
```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.6.3</version>
    <relativePath/>
</parent>
```


3 First term assignments: Vue.JS and Spring Boot

3.1 The Auctioneer Home Page

In this assignment you explore the setup of an Angular 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 Vue 'Single Page Application' project within a parent project folder 'Auctioneer':

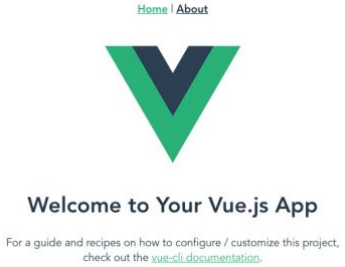


$ vue create auc-vue

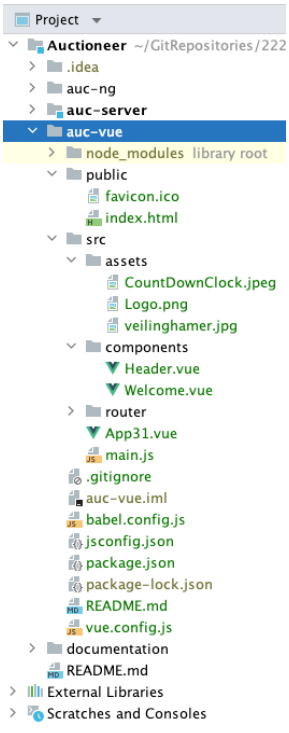
or File → New Module → JavaScript → Vue.js

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.

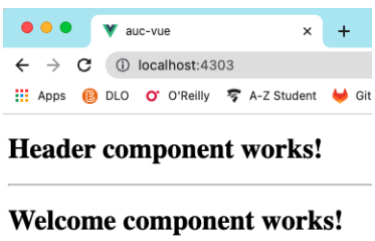


B. Let's create a simple application structure with a Header component and a Welcome component:
Clean-up/remove boiler plate sample components from the src and src/components folder structure.
Use 'File → New → Vue Component' to define three new components:

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

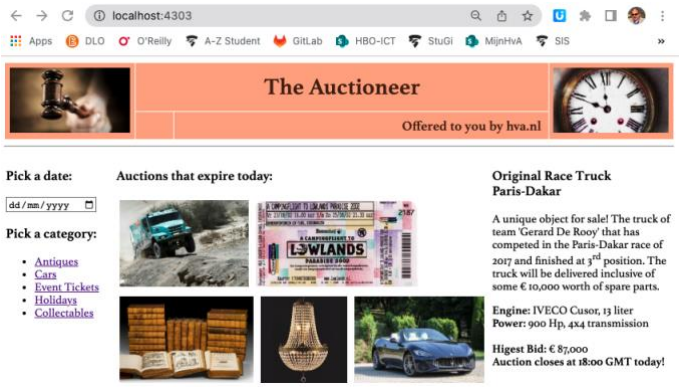
Reconfigure main.js to load the App31 component.
Reconfigure App31.vue to show the Welcome component below the Header component.
At the left you find an indication of the initial development folder structure.

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



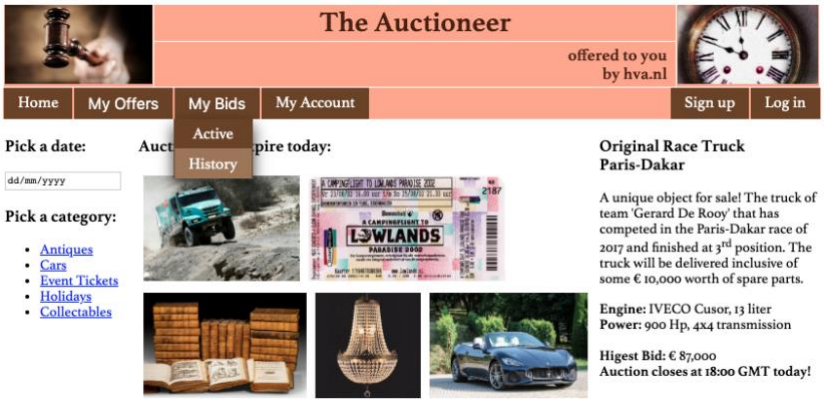
C. Design the Header component template to hold a title, sub-title and two (logo) pictures left and right. Store the pictures in src/assets. Make sure the header is responsive such that its pictures use fixed size and the title space scales with the size of the window. The text of the subtitle should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

Design the Welcome component to display its content in three columns. The left and right columns have a fixed width, the centre column scales with the window. Provide some extra margin between the images at the centre. When you click an image, a new tab should open and navigate to a related page on the internet.
The date input should provide a date picker, but no further action needs to be connected at this stage.
The categories are hyperlinks with dummy url-s for now.



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 Offered Articles 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 Offers

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 the offer.js model class with the following instance attributes:

id: number;	description: string;
title: string;	sellDate: Date;
status: Offer.Status;	valueHighestBid: number;

Offer.Status is an enumeration of string values “NEW”, “FOR_SALE”, “SOLD”, “PAID”, “DELIVERED”, “CLOSED”, “EXPIRED”, and “WITHDRAWN”.

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

```
public static createSampleOffer(pId = 0)
```

which creates and returns a new offer 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 content of each created offer, i.e.:

- Provide a mix of statusses
- Provide past and future sellDates
- Provide random values for valueHighestBid;
  but if status is NEW, valueHighestBid shall be zero.

C. Create an OffersOverview31 component in src/components/offers/Overview31.vue.
Instantiate a local list of at least 8 offers from the created() hook in the controller code using `Offer.createSampleOffer(pId)` to instantiate the offers. Let the component keep track of the next available offer-id: start at 30000 and increase with a

(random) increment of about 3 for each new offer.

D. Display the offer data within a <table> in the HTML-template of the component.
Use the iteration directive on <tr> to show all offers from the local list.
Use 'string interpolation binding' on the offer properties.
If the status of the offer is 'NEW', the value of the highest bid shall be left blank.
Use CSS to style the table.
Replace the 'Welcome' view in the App31 component by your offers list view.

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

Test your application by adding some offers.

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 offers and embeds a detail child component which provides an edit form such that the user can change all properties of any specific offer 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 'offers/Overview32.vue' and 'offers/Detail32.vue' to src/components
Overview32 shall manage a selection list of offers displaying only their titles, similar to the offers list of assignment 3.2.
Implement the selection mechanism by binding the click event of each offer in the selection list to a method that keeps track of the currently selected offer in a property 'selectedOffer'. At any time, at most one offer can be selected. When nothing has been selected yet then selectedOffer==null. When the same offer is reselected, it will become unselected again.
Use CSS to highlight the selected offer.
The 'New Offer' button adds a new offer to the list and automatically selects it.

B. Overview32 embeds the Detail32 component in a sub-panel '<app-offers-detail>'. This component shows all details of the selected offer. 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 selectedOffer 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 selectedOffer.

A consequence of this approach will be that any change to selectedOffer by the Detail32 component will also apply directly to that offer in the list that is maintained by Overview32. Both refer to the same Offer 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 different approach.

```
42  export default {
43    name: "OffersDetail32",
44    props: ['selected-offer-from-overview'],
45    emits: ['delete-selected'],
46    data() {
47      return {
48        OfferStatus: Offer.Status,
49      }
50    },
51    methods: {
52      onDelete() {...}
56    },
57    computed: {
58      sellDateUpdater: {
59        get() {...},
64        set(localDateTime) {...}
68      },
69  ...
72    }
73  }
```

The v-model does not offer two-way with complex data types such as Date. For that you shall provide a custom computed property interface with a getter and a setter.

D. Now populate the view template of Detail32:
Include the offer-id of the 'selectedOffer' 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 'selectedOffer'.

```
computed: {
  sellDateUpdater: {
    get() {
      // TODO return the ISOString of the date in format YYYY-MM-DDThh:mm
      {...}
    },
    set(localDateTime) {
      // TODO create a new Date from the localDateTime string; use time zone 'Z' (i.e. GMT+00)
      {...}
    }
  },
```

The v-model does not offer out of the box two-way binding with complex data types such as Date. For that you shall provide a custom computed property interface with a getter and a setter. Now the v-model will call the get method on the property if it wants to initialise the HTML-element from the property, and call the set method on the property if it wants to transfer the entered value to the property.

When a new Offer is added via the 'New Offer' button at the left, that offer should automatically get the selection focus (and be associated with the Detail32 component).
When you navigate along different offers, any changes that you make in the detail panel will be remembered, because all edits operate directly on the offers in the list of the OfferOverview32 component.

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

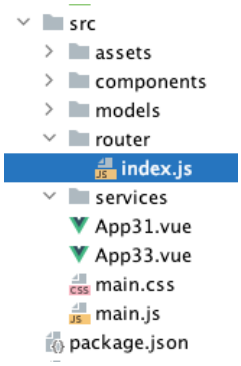
3.3 Page Routing

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:

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

We prefer the 'hash' mode, by which the router inserts a hash-tag '#' in the URL just 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. 'offers/overview31' shows the offers list of assignment 3.2.1.

3. ‘offers/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 offer with the Master/Detail component, we will integrate the selection of an offer into the router path. For that we use parent / child routing with router parameters.

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

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

Now the selection of an offer to be edited must be connected to the router in two ways:
1. If the user clicks another offer 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 /offers/overview33/12345 (initiated from the menu bar, from the browser url or programmatically) then Overview33 should select and highlight the offer with id=12345, and also Detail32 should follow the selection in Overview33 and open that same offer for editing.

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

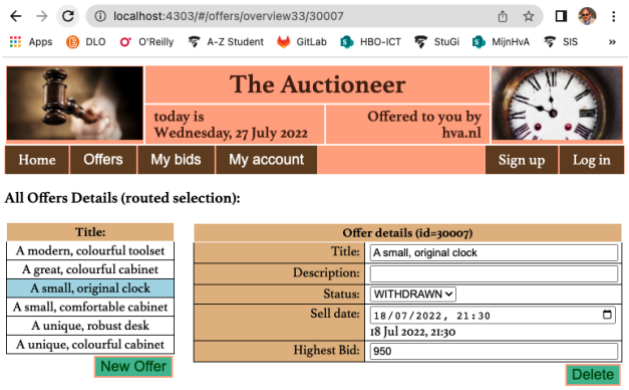
```
onSelect(offer) {
  if (offer != null && offer !== this.selectedOffer) {
    this.$router.push(this.$route.matched[0].path + "/" + offer.id);
  } else if (this.selectedOffer != null) {
    // TODO navigate to the parent path to unselect the selectedOffer
    {...}
  }
```

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'() {
    //console.log("watch $route", this.$route);
    this.selectedOffer = this.findSelectedFromRouteParam(this.$route);
  }
},
```

Detaill32 can still bind to the 'selectedOffer' property from Overview33 and has no need to follow the router itself yet 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 offers.
2. Navigation by URL, with and without offer-id
3. Deletion of offers.
4. Navigation by URL to deleted offers.

3.4 Master/Detail managed persistence

Direct editing of the selected offer 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 offer 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.

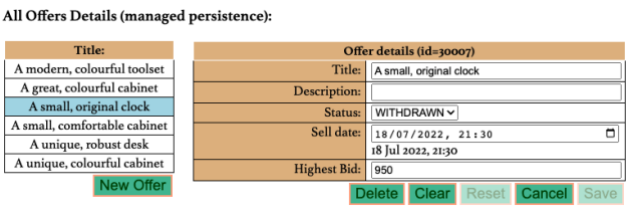
A. Create a new component offers/Detail34.vue which can be a copy of offers/Detail32 initially. Create a new route /offers/overview34 which invokes component Overview33 with child component Detail34. (The interface between the master and the new detail component can be compatible with the previous assignment.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

B. Change your Detail34 component to clone a copy of the selected offer (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(offer) in the Offer.js model class for this purpose in order to properly encapsulate deep cloning of an Offer 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 Date in this case):

```
static copyConstructor(offer) {
    if (offer == null) return null;
    let copy = Object.assign(new Offer(), offer);
    copy.sellDate = new Date(offer.sellDate);
    return copy;
}
```

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

C. Provide additional buttons in the form to manage persistence of changes:
The '**Save**'-button will update the selected offer 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 offer empty.
The '**Cancel'**-button will abandon the edit form and without persisting any changes.
The '**Delete**'-button will delete the selected offer and remove it from the list regardless of any changes underway.
'Save', 'Cancel' and 'Delete' will also unselect the edited offer (by programmatic interaction with the router).

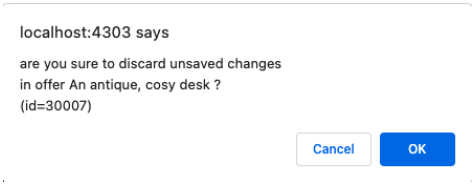


D. Implement ‘disabled’ property binding on the ‘Save’, ‘Reset’ and ‘Delete’ buttons, such that ‘Save’ and ‘Reset’ are disabled if nothing has been changed yet, and ‘Delete’ is disabled if there are unsaved changes.
The ‘Cancel’ and ‘Clear’ buttons are always enabled.
(Hint: provide a computed property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

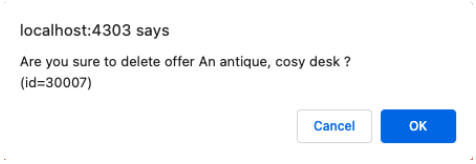
3.4.2 [BONUS] Guarded navigation

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 offer, 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.



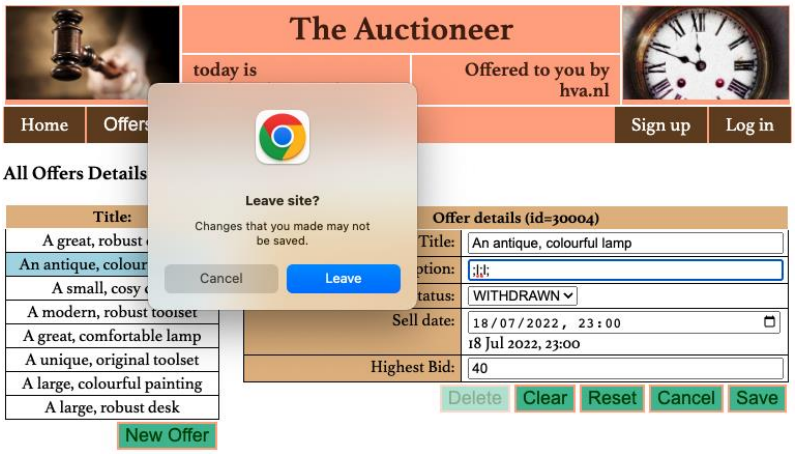
B. At https://router.vuejs.org/guide/advanced/navigation-guards.html it is explained how you can add ‘Navigation Guards’ to a component to assess whether it is safe to navigate away from the component or to a different selection of router parameters. If these guards evaluate false, then the undertaken 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.

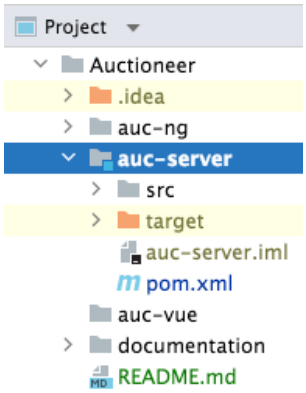
In the following assignments you will build the backend part of 'The Auctioneer' application as depicted in the full stack, layered logical architecture of section 2.3. You will use the Spring-Boot technology.

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 offers of your application.

These offers are managed in the Spring-Boot backend by an implementation of a OffersRepository Interface. The OffersRepository is injected into the REST Controller by setter dependency injection (O'Reilly-2, Chapter 3, step 7). First, you start with providing a OffersRepositoryMock bean implementation that just tracks an internal array of offers. 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

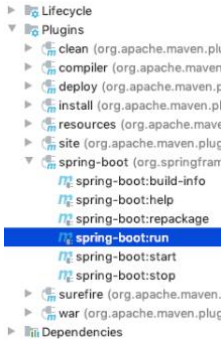
Frontend and backend modules are best configured as siblings of a full stack parent project. This approach supports:
i) use of different IDE-s for each of the modules
ii) separate configuration of automated deployment procedures for each module.
iii) multiple frontend solutions that all share the same 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…

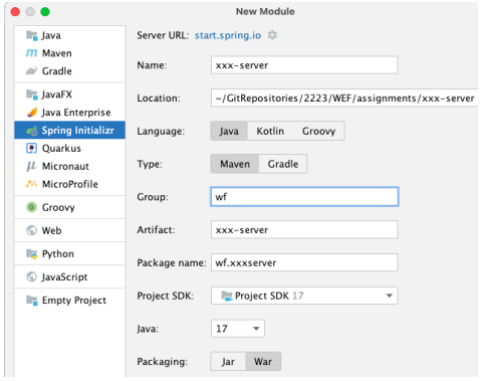
Below you find more detailed explanation of how you can implement the objectives of this assignment.

A. Create a Spring-Boot application module 'auc-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.

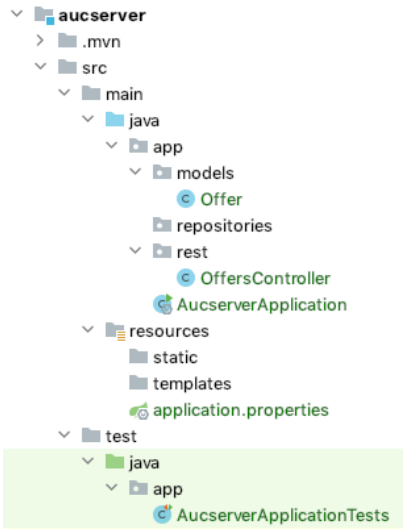
Test your project setup by running the 'spring-boot:run' maven goal.



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=8083
server.servlet.context-path=/
logging.level.org.springframework = info
server.error.include-message=always
```

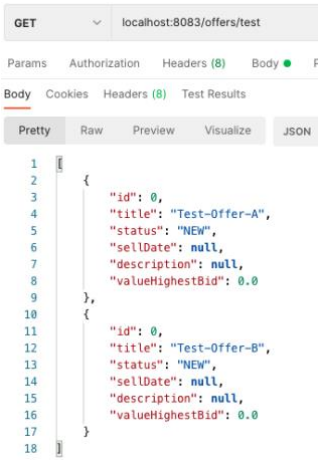
B. You may want to tidy-up the source tree of your module, similar to lay-out shown here. Always use the refactoring mode of IntelliJ to move files around such that the dependencies will be fixed. Basic rules for setup are:
   1. Your AucServerApplication class should reside within a non-default package (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 Offer model class and the OffersController rest controller class, as explained in O'Reilly-2.
   Replicate/convert your Offer.java model class from the frontend code.
   Implement in the OffersController a single method 'getTestOffers()' that is mapped to the '/offers/test' end-point of the Spring-Boot REST service.
   This method should return a list with two offers like below.

```
public List<Offer> getTestOffers() {
    return List.of(
            new Offer("Test-Offer-A"),
            new Offer("Test-Offer-B") );
}
```



   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 offers like you expect.

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

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

```
public static Offer createSampleOffer(long id) {
    Offer offer = new Offer(id);
    // TODO put some semi-random values in the offer attributes
```

The responsibility for generating and maintaining unique ids should be implemented in the OffersRepositoryMock class. Later, that responsibility will be moved deeper into the backend to the ORM.

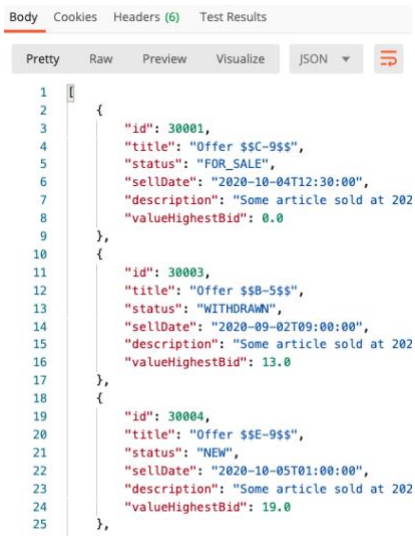
The OffersRepository interface provides one method 'findAll()' that will be used by the endpoint in order to retrieve and return all offers:

```
public interface OffersRepository {
    public List<Offer> findAll();
}
```

```
public List<Offer> getAllOffers() {
        return repository.findAll();
}
```

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

Test your end-point again with Postman:



3.6 Enhance your REST controller with CRUD operations

In this assignment you will enhance the repository interface and the /offers REST-api with endpoints to create new Offers and get, update or delete specific offers. 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 OffersRepository interface should have evolved to

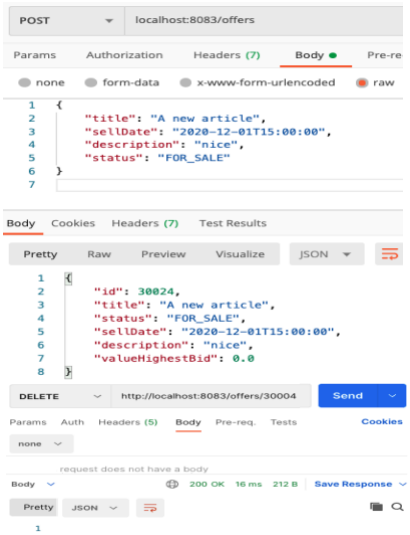
```
public interface OffersRepository {
    List<Offer> findAll();          // finds all available offers
    Offer findById(long id);        // finds one offer identified by id
                                    // returns null if the offer does not exist
    Offer save(Offer offer);        // updates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

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

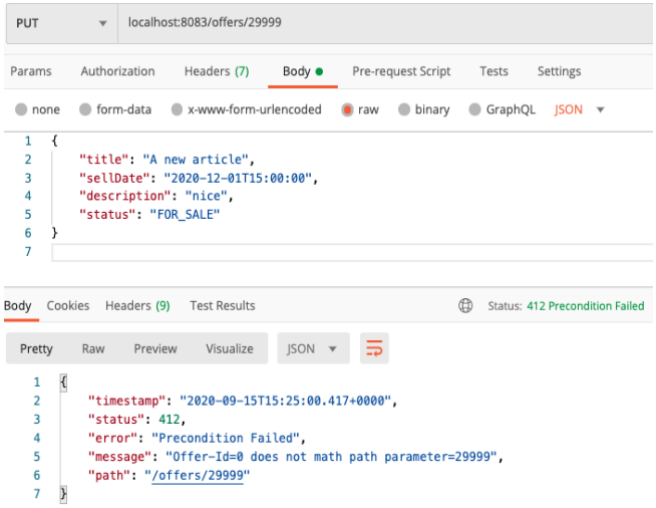
B. Enhance your REST OffersController with the following endpoints:
- a GET mapping on '/offers/{id}' which uses repo.findById(id) to deliver the offer that is identified by the specified path variable.
- a POST mapping on '/offers' which uses repo.save(offer) to add a new offer to the repository.
If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

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

Test the new mappings with postman.



C. Implement Custom Exception handling in your REST OffersController:



- 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 offer 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 on a getOffersSummary() mapping at '/offers/summary', which only returns the id, title and status of every offer. Dynamic filters can most easily be implemented with a @JsonView specification in your Offer 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.

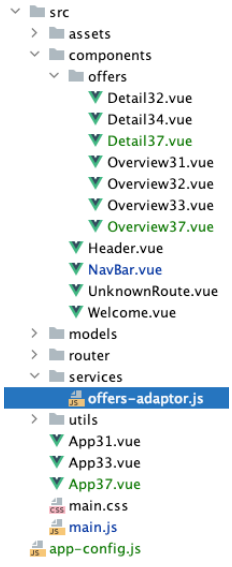
3.7.1 A REST Adaptor for Fetch-API requests.

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 /offers/overview37 which invokes component OffersOverview37 with child component OffersDetail37. Add an option for this route to your offers 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 an offers adaptor, which will provide connectivity to the /offers endpoint mapping of the backend.
   Four basic CRUD operations shall be implemented by this adaptor:
   asyncFindAll() retrieves the list of all offers
   asyncFindById(id) retrieves one offer, identified by a given id
   asyncSave(offer) saves an updated or new offer and returns the saved instance.
   asyncDeleteById(id) deletes the offer identified by the given id.

   New offers shall be set up with id=0. The backend service should generate a new, unique id for such offers 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 OffersAdaptor:

```
export class OffersAdaptor {
    resourcesUrl;
    constructor(resourcesUrl) {
        this.resourcesUrl = resourcesUrl;
        console.log("Created OffersAdaptor 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<Offer[]> */ {...}

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

    async asyncSave(offer) /* :Promise<Offer> */ {...}

    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 Offer.copyConstructor static method to map all json object structures from the response into true instances of the frontend Offer class. Without such mapping the offers 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 OffersAdaptor 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 {OffersAdaptor} from "@/services/offers-adaptor";
import CONFIG from '../app-config.js'
export default {
  name: "App",
  components: {'app-header': Header...},
  provide() {
    return {
      // stateless data services adaptor singletons
      offersService: new OffersAdaptor(CONFIG.BACKEND_URL+"/offers"),
    }
  }
}
```

The Overview37 and Detail37 components both inject the instance:

```
export default {
  name: "OffersOverview37",
  inject: ['offersService'],
```

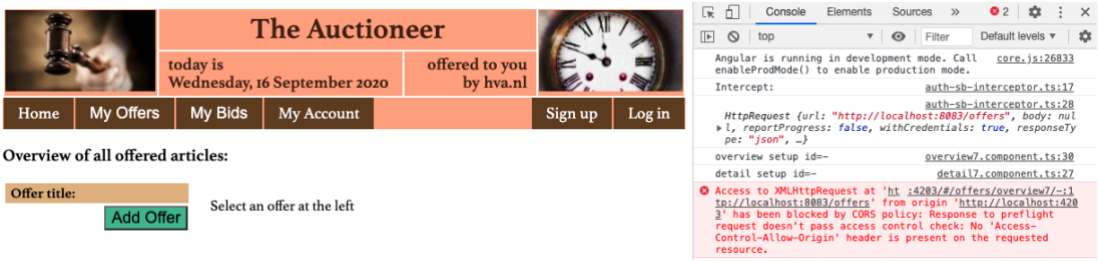
```
export default {
  name: "OffersDetail37",
  inject: ['offersService'],
```

With that, these components are both set up to use the service adaptor to interact with the backend. E.g. The OffersOverview37 component can now dynamically initialise its list of offers from the backend within its 'created()' lifecycle hook:

```
async created() {
  this.offers = await this.offersService.asyncFindAll();
  this.selectedOffer = 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-offers 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 (4303), 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 OffersAdaptor and use them appropriately in both the Overview37 and Detail37 components.

Some functionality involves special consideration:
1. If a New Offer is added, the frontend shall generate a new, empty offer with id=0, and first save it to the backend via offersService.asyncSave(newOffer). The backend shall generate and set a new unique id and return the updated offer in the response. The frontend should have 'awaited' the response and then push the newly saved offer into the list of offers in OffersOverview37 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. OffersDetail37 no longer takes the selectedOffer as a property from OffersOverview37 but uses the id-parameter from the route to retrieve itself an up-to-date version of the offer from the backend. (Use offersService.asyncFindById()). OffersDetail37 shall also directly save and delete offers at the backend without seeking intermediary involvement of OffersOverview37.

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 offers 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 an OffersAdaptor that provides your components with an async/await API for retrieving and updating offers 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
    //offersService: new OffersAdaptor(CONFIG.BACKEND_URL+"/offers"),
    offersService: new RESTAdaptorWithFetch(CONFIG.BACKEND_URL+"/offers", Offer.copyConstructor),
    usersService: new RESTAdaptorWithFetch(CONFIG.BACKEND_URL+"/users", User.copyConstructor),
  }
}
```

A. Generalize your implementation of the OffersAdaptor into a generic implementation of RESTAdaptorWithFetch along the suggested out-line, which has no specific dependencies on the Offer 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";
export default {
  name: "App",
  components: {'app-header': Header...},
  provide() {
    return {
      // statefull cached data services adaptor singletons
      cachedOffersService: reactive(
          new CachedRESTAdaptorWithFetch(CONFIG.BACKEND_URL+"/offers", Offer.copyConstructor)),
```

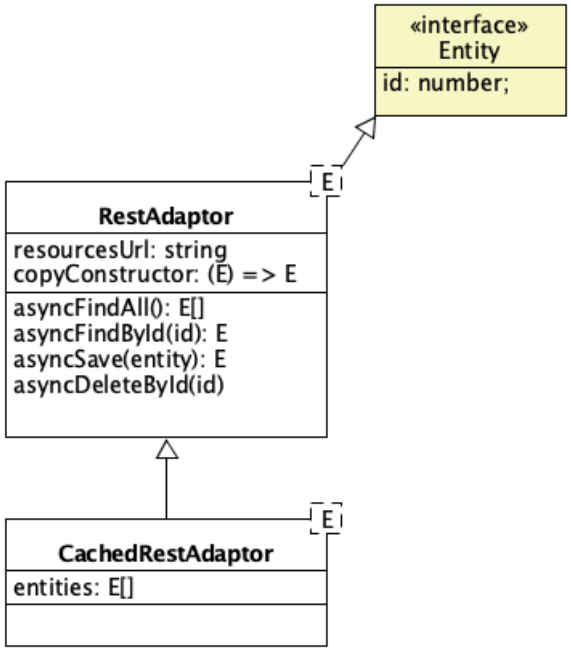

In the overview we inject the cachedOffersService and use a computed property to replace the local offers array by a reference to the entities cache of the service:

```
export default {
  name: "OffersOverview37c",
  inject: {
    offersService: { from: 'cachedOffersService'}
  },
  computed: {
    offers() { return this.offersService.entities; }
  },
```

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 offers array anymore in Overview37c, otherwise than by CRUD operations via the offersService. (I.e., verify the onNewOffer method and remove the refresh event handler.)

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

   Test your application.
   1. Verify that the title update to offers also appears in the selection list.
   2. Verify that deletions are removed from the list.
   3. Verify that new offers are added to the list



4 Second term assignments: JPA, Authentication and WebSockets

In these assignments you will expand the backend part of 'The Auctioneer' 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 EntityManger in transactional mode to ensure data integrity across multiple updates.

By the end of this assignment you will have implemented the Offer and Bid classes including its one-to-many relationship. Your REST API can add Bids to Offers. It will produce error responses on Bids on Offers that are not FOR_SALE and Bids that do not exceed the existing highest Bid on the Offer.

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

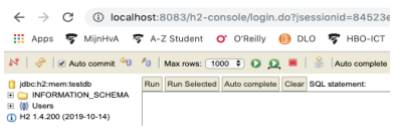
In this assignment you should practice hands-on experience with JPA and Spring-Boot annotations @Entity, @Id, @GeneratedValue @OneToMany @ManyToOne @Repository @PersistenceContext @Primary @Transactional @JsonManagedReference @JsonBackReference and classes EntityManager and TypedQuery.

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 Offer 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 OffersRepositoryJpa for your OffersRepository 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 OffersRepositoryJpa.

If you now run the backend you might get a NonUniqueBeanDefinitionException, because you may have two implementation classes of the OfferRepository interface: OffersRepositoryMock and OffersRepositoryJpa.
(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 an offer with postman doing a post at localhost:8083/offers. Verify the associated SQL statements in the Spring Boot log and use the h2-console to verify whether the offer ended up in H2.

```
Hibernate: call next value for offer_ids
Hibernate: insert into offer (description, sell_date, status, title, id)
 values (?, ?, ?, ?, ?)
```

SELECT * FROM OFFER;

ID	DESCRIPTION	SELL_DATE	STATUS	TITLE
30001	Some article sold at 2020-08-26T05:00:00	2020-08-26 05:00:00	WITHDRAWN	Offer $$B-7$$

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

C. Also implement and test the other three methods of your OffersRepository 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 OffersRepositoryJpa.findAll()

Test your new repository with postman.

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

D. Inject the offers 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.

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

```
private void createInitialOffers() {
    // check whether the repo is empty
    List<Offer> offers = this.offersRepo.findAll();
    if (offers.size() > 0 ) return;
    System.out.println("Configuring some initial offer data");

    for (int i = 0; i < 9; i++) {
        // create and add a new offer with random data
        Offer offer = Offer.createRandomOffer();
        //offer.setId(11*i);
        offer = this.offersRepo.save(offer);

        // 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 model class 'Bid' identified by an attribute named 'id' (long) and with one attribute named 'value'(double).
   A Bid is associated with one Offer.
   An Offer is associated with many Bids.

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 bid with this offer, if not yet associated
 * and only if the value of the bid exceeds the currently highest bid on this offer
 * @param bid
 * @return  whether a new association has been added
 */
public boolean associateBid(Bid bid) {...}

/**
 * Dissociates the given bid from this offer, if associated
 * @param bid
 * @return  whether an existing association has been removed
 */
public boolean dissociateBid(Bid bid) {...}
```

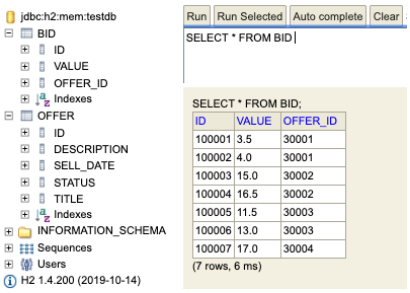
```
/**
 * Associates the given offer with this bid, if not yet associated
 * @param offer     provide null to dissociate
 *                  the currently associated offer
 * @return          whether a new association has been added
 */
public boolean associateOffer(Offer offer) {...}
```

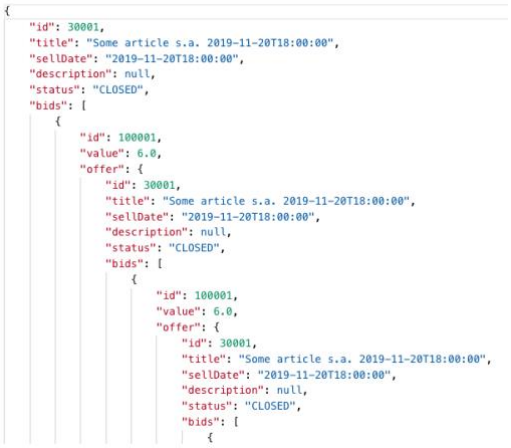
Also provide the JPA @ManyToOne and @OneToMany annotations as explained in O'Reilly-3.Ch8

B. Implement a BidsRepositoryJpa similar to your OffersRepositoryJpa.
You should not like 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 and just replicate

the OffersRepository code….

C. Extend the initialisation in the command-line-runner of task 4.1.1-D to add a few Bids to every Offer 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') Offers only include 'attached' Bids.



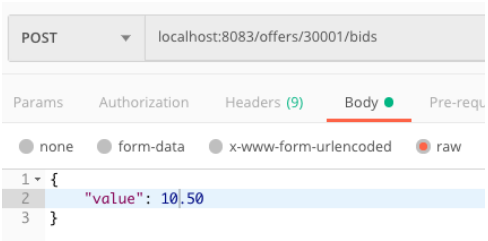


Test your application and review the database schema in the H2 console. Check its foreign keys and review the contents of the BID table:

Re-test the REST API at localhost:8083/offers 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.

D. Implement a POST mapping on the /offers/{offerId}/bids REST endpoint. This mapping should add a new Bid to the Offer.

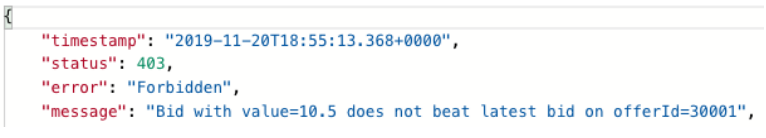
The body of the POST request should provide a new Bid object which will be further checked and amended by the rest controller.



A "PreCondition Failed" error response should be returned if:
1) the offer does not have status 'FOR_SALE' or
2) the bid value is not higher than the latest(=highest) bid on the offer.

In other cases the bid should be created and added to the offer, and a creation success status code should be returned.

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



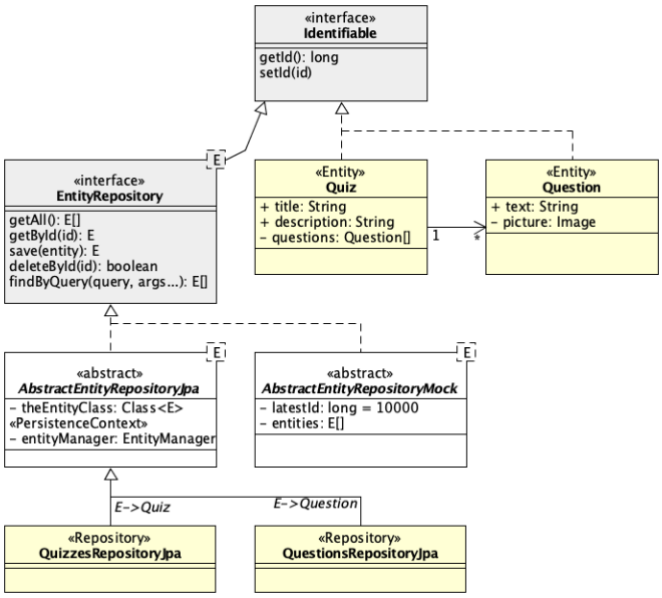
4.1.3 [BONUS] Generalized Repository

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 the code snippets below the Quiz entity by your Offer entity and the Question entity by your Bid 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 /offers REST endpoint to optionally accept a request parameter '?status=XXX' or '?title=XXX' or a combination of '?status=XXX' and '?minBidValue=999', and then filter the list of offers being returned to meet the specified criterium.

You will pass the filter as part of a JPQL query to the persistence context, such that only the offers that meet the criteria will retrieved from the backend.

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
```

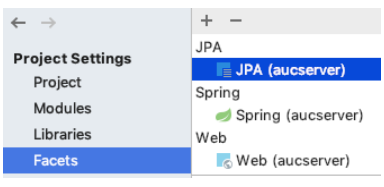
(Here we assume you use the generic repository interface)

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:
"Offer_find_by_status": finds all offers of a given status
"Offer_find_by_title": finds all offers that have a given sub-string in their title
"Offer_find_by_status_and_minBidValue": finds all offers that have a given status and a bid above a given minimum value

Use the @NamedQuery annotation to specify these named JPQL queries within your Offer.java entity class.
Use ordinal(positional) query parameters (?1, ?2, etc.) as place-holders 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)

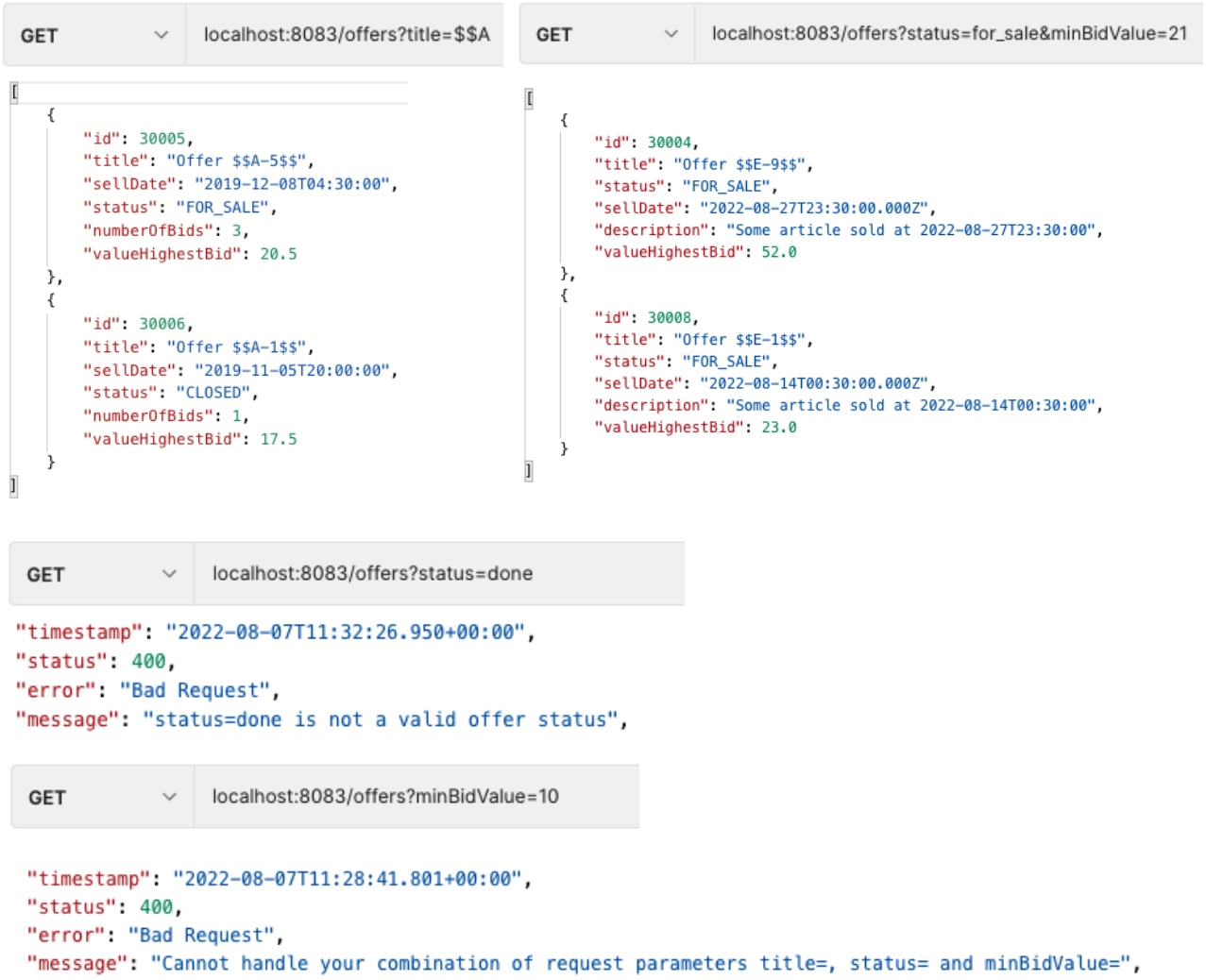


C. Extend your GetMapping on the "/offers" REST end-point to optionally accept a '?title=XXX' or '?status=XXX' and/or '?minBidValue=999' request parameter.

   If no request parameter is provided, the existing functionality of returning all offers should be retained.
   If the '?status=XXX' parameter is provided, with a status string value that does not match the Offer.Status 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 offers should be retrieved from the repository, using the appropriate named query and the specified parameter value.
   You may want to explore the impact of the @Enumerated annotation for the status attribute of an offer.

   Test the behaviour of your end-point 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 an Offer, you want full information about the offer but probably only shallow information about its bids.
b) If you query a Bid, you want full information about the bid, but only shallow information about the offer.
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);
        }
    }
```

```
@JsonView(CustomJson.Summary.class)
@JsonSerialize(using = CustomJson.ShallowSerializer.class)
private List<Bid> bids = null;
```


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

The consequence is:
1) the bids list will only get serialized for unrestricted offer mappers and mappers that specify the CustomJson.Summary view.
2) when bids are serialized (as part of an offer serialization) its serialization will be shallow (and not recurse back into its own offer…)

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 Offer and Bid model classes.
Apply the Summary view class to the localhost:8080/offers and localhost:8080/offers/{offerId}/bids end-points.
Implement unrestricted end-points at localhost:8080/offers/{offerId} and localhost:8080/offers/{offerId}/bids/{bidId}.

Test your end-points with postman:

localhost:8083/offers


```
[
    {
        "id": 30001,
        "title": "Some article s.a.
            2019-11-02T14:00:00",
        "sellDate": "2019-11-02T14:00:00",
        "status": "CLOSED",
        "numberOfBids": 1,
        "valueHighestBid": 23.5
    },
    {
        "id": 30002,
        "title": "Some article s.a.
            2019-11-30T23:00:00",
        "sellDate": "2019-11-30T23:00:00",
        "status": "NEW",
        "numberOfBids": 1,
        "valueHighestBid": 2.0
    },
```


localhost:8083/offers/30001


```
{
    "id": 30001,
    "title": "Some article s.a.
        2019-12-07T22:30:00",
    "sellDate": "2019-12-07T22:30:00",
    "description": null,
    "status": "FOR_SALE",
    "bids": [
        {
            "id": 100001,
            "value": 24.0
        },
        {
            "id": 100002,
            "value": 24.5
        },
        {
            "id": 100003,
            "value": 26.0
        }
    ],
    "numberOfBids": 3,
    "valueHighestBid": 26.0
}
```


localhost:8083/offers/30001/bids/100001


```
{
    "id": 100001,
    "value": 11.0,
    "offer": {
        "id": 30001,
        "title": "Some article s.a.
            2019-12-13T15:00:00",
        "status": "NEW"
    }
}
```


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 in 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 guards all incoming requests.
This filter will extract the security token from the incoming request, if included.
Only requests with a valid security token may pass thru to the secure parts of the REST service.

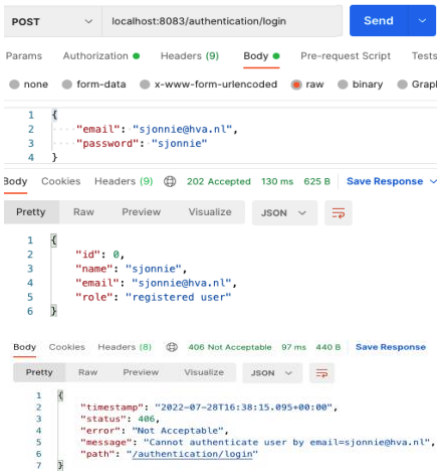
By the end of this assignment you will have further explored annotations @RequestBody, @RequestAttribute, @Value and classes ObjectNode, Jwts, Jws<Claims>, SignatureAlgorithm

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 encrypting 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.

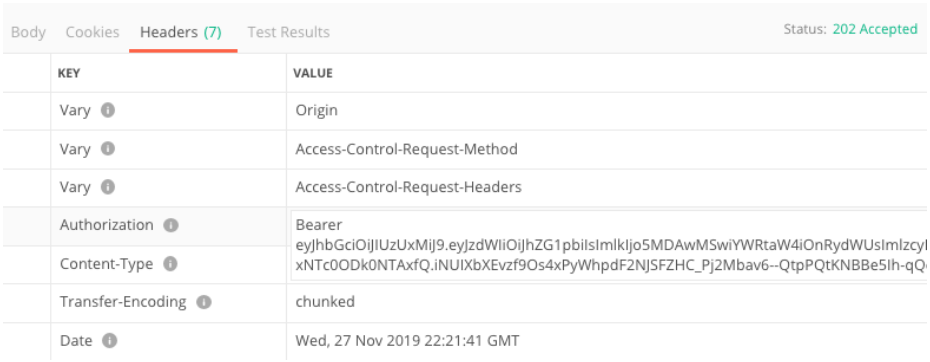
```
// JWT configuration that can be adjusted from application.properties
@Value("HvA")
public String issuer;

@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;
}
```

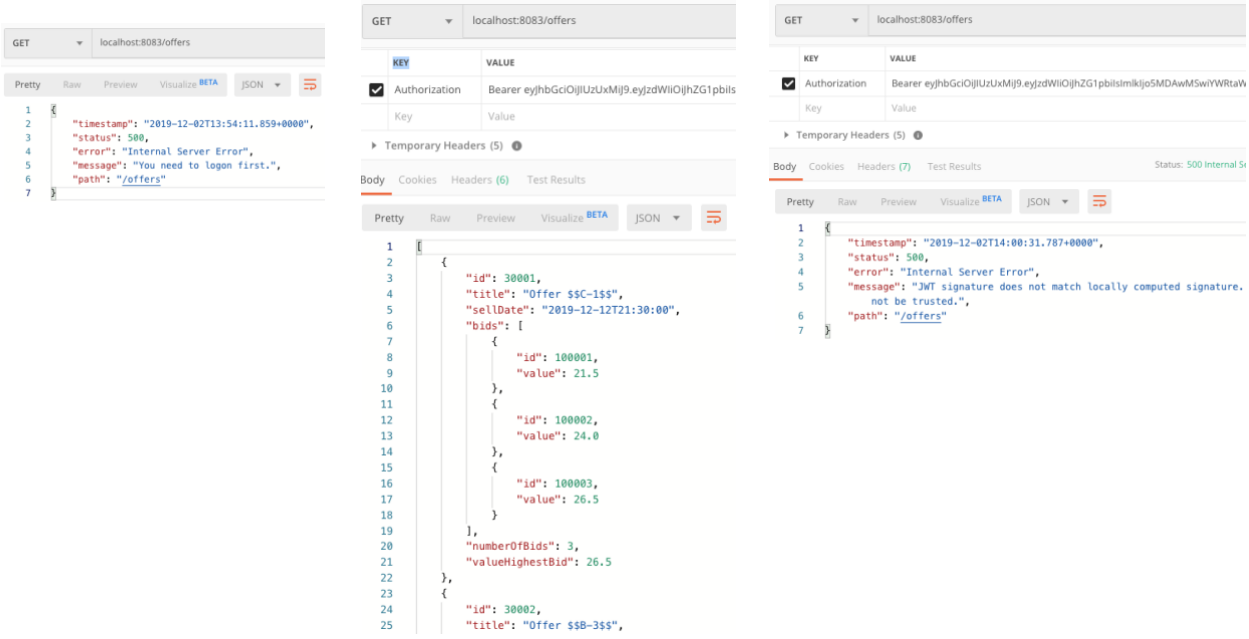
This decode method uses the same JWToken attributes and getKey method that were also shown earlier along with the encode method.

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 will 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 end-points 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, password)
   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, user)
   saves the JWT authentication token and user details into the service and browser storage for automatic retrieval after application or page reload.
   getTokenFromBrowserStorage()
   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:

```
session-sb-service.js
1  export class SessionSbService {
2      BROWSER_STORAGE_ITEM_NAME;
3      RESOURCES_URL;
4  ...
7      constructor(resourcesUrl, browserStorageItemName) {
8          this.BROWSER_STORAGE_ITEM_NAME = browserStorageItemName;
9          this.RESOURCES_URL = resourcesUrl;
10         this.getTokenFromBrowserStorage();
11         //console.log("SessionSbService recovered token: ", this._currentToken);
12     }
```

   Several components will require access to this session service, so we will 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 CONFIG from '../app-config.js'
import {reactive, shallowReactive} from "vue";

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
      usersService: new RESTAdaptorWithFetch(CONFIG.BACKEND_URL+"/users", User.copyConstructor),
      offersService: new RESTAdaptorWithFetch(CONFIG.BACKEND_URL+"/offers", Offer.copyConstructor),

      // reactive, state-full services
      cachedOffersService: reactive(
          new CachedRESTAdaptorWithFetch(CONFIG.BACKEND_URL+"/offers", Offer.copyConstructor)),
      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-in 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 login credentials did not match, the backend should have provided an error response without a valid token or User object. 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:

```
export class FetchInterceptor {
    static theInstance; // the singleton instance that has been registered
    session;
    router;
    unregister; // callback function to unregister this instance at shutdown
    constructor(session, router) {

        this.session = session;
        this.router = router;
        // fetchIntercept does not register the object closure, only the methods as functions
        // so we need an additional static reference to the singleton's attributes
        FetchInterceptor.theInstance = this;
        this.unregister = fetchIntercept.register(this);
        console.log("FetchInterceptor has been registered; current token = ",
            FetchInterceptor.theInstance.session.getCurrentToken() );
    }

    request(url, options) {...}
    requestError(error) {...}
    response(response) {...}
    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 offer that does not exist:

4.5 Make a bid

In this assignment you will build on all the preparatory work of the previous assignments to implement some of the workflow of making a bid. You will create a bid page, where a user can view all current offers still for sale and their currently highest bid. Then the user can select an offer and add an improved bid.

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

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.

A. First verify that you have a structurally complete backend implementation of the Offer, Bid and User entities as per the class diagram of section 2.2, and your work of sections 4.1.2 and 4.3.1. 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:

   We will use the relation between Offer and Bid: Every Offer can have many Bids; every Bid belongs to precisely one Offer.
   We will also use the relation between User and Bid: Every User can make many Bids; every Bid is made by exactly one User.
   We will not use the relation between User and Offer, so you may omit this relationship from your JPA annotations for now…
   Also provide all three Repository<Offer>, Repository<Bid> and Repository<User> implementations.

B. Check/update the setup of your initial data by CommandLineRunner:
   1) Initialise at least three User entities of which one should have an ADMIN role.
   2) Initialise about ten Offer entities, some of which should have status FOR_SALE.
   3) Associate between zero and five bids on each offer, except for the offers with status NEW. The values of the bids should be increasing. Associate each of the Bids at random with one of the Users.
   If all your entity classes have got implemented the associate methods of section **Error! Reference source not found.**-A) you can robustly make and change associations.
   4) Verify that your JSON serialisation filter of the GetMapping of a specific offer at /offers/{id} includes the bids without infinite recursion. (See also bonus assignment **Error! Reference source not found.**B.)


```
{
    "id": 30003,
    "title": "Offer $$D-5$$",
    "status": "FOR_SALE",
    "sellDate": "2022-08-12T21:00:
    "description": "Some article s
    "bids": [
        {
            "id": 100002,
            "value": 12.0,
            "madeBy": {
                "id": 500700003,
                "name": "user2"
            }
        },
        {
            "id": 100003,
            "value": 19.0,
            "madeBy": {
                "id": 500700001,
                "name": "admin"
            }
        }
    ],
```


   Restart your backend and verify your initial offers and bids with postman.

C. Add two new components Overview45 and Detail45 into a new folder src/components/bids. You may start with copies of Overview37 and Detail37 initially. Create e new route bids/overview45 to be invoked from the 'My Bids → Offers on sale' menu item.

The page should show something like below:

1. The selection list at the left only shows on sale offers with status FOR_SALE. (This filter shall be applied by the backend, to prevent the frontend being overloaded with irrelevant data…)
2. The selection list at the left also shows value and username of the highest(=latest) bid on the offer.
3. The detail at the right shows some but not all details of the latest Bid constrained by JSON serialization in the backend. (E.g. information about user credentials shall not be exposed to other users of the application!!!)

Below you find some suggestions for the design of your implementation:

Extend 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;
}
```

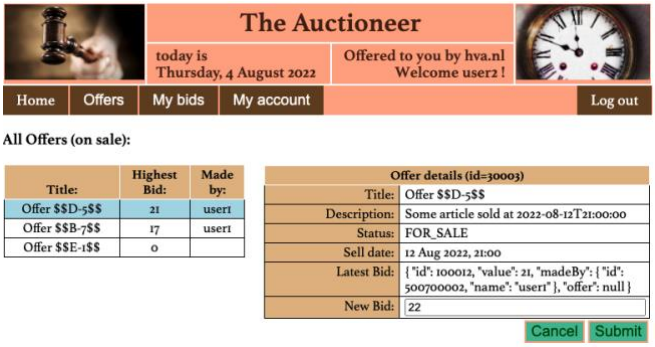
Then the Overview45 component can retrieve all ‘on sale’ offers providing the designated status as a query parameter on the generic asyncFindAll call (assuming that the backend /offers resource endpoint will accept and process the query parameter):

```
async reLoad() {
  this.offers = await this.offersService.asyncFindAll({ status: "FOR_SALE"});
  this.selectedOffer = this.findSelectedFromRouteParam(this.$route);
}
```

You can access the latestBid of an offer by defining a special getter of this property in your frontend Offer class assuming that offer will include the full list of associated Bids. You can also choose to define this getter in your backend code and use JSON serialization annotations to export this item from the API instead of the complete list of all bids:

```
@Transient
@JsonView({CustomJson.Summary.class})
@JsonSerialize(using = CustomJson.SummarySerializer.class)
public Bid getLatestBid() {
    if (this.bids == null || this.bids.size() == 0) {
        return null;
    }
    return this.bids.get(this.bids.size()-1);
}
```

The @Transient annotation indicates that this property shall not be mapped to the database. (This is a derived property, depending on the content of the bids list.)
The @JsonSerialize annotation will drive restricted content of the bid and the associated user who made the bid. (See also bonus assignment 4.2.2)



D. Amend the bids/Detail45 component adding a ‘New Bid’ numeric input field.
The initial value of this field should be set one Euro higher than the previously highest Bid on the offer.

Also add a ‘Submit’ and ‘Cancel’ button to the component.

When ‘Cancel’ is pressed, the currently selected Offer is unselected.

The Submit button shall be enabled only when the ‘New Bid’ value is truly above the latest Bid value.
When Submit is clicked, a backend POST request shall be issued to the ‘/offers/{id}/bids’ resource endpoint to add another Bid to the specified Offer.
The backend shall also associate that Bid with the currently logged on User (which can be found from the ‘userId’ that is part of the JWT authentication token).

When the POST has returned the newly created Bid, the frontend shall cancel the offer selection and show a message indicating that a new Bid has been created

successfully. (Including the id of the Bid that was allocated by hibernate, updated in the Bid instance by the JPA entity manager, returned by the repository to the REST controller, and then serialised into the body of the http response…)

Notice that the information in the Overview45 selection list has not been updated yet with the information of the new latest Bid! That will be handled in the next assignment by use of global notification which also observes bids by other users…

Hint: instantiate a new local bidsService RESTAdaptor within the Detail45 component by extending the offersService.resourcesUrl path to the correct endpoint for its Bids. This bidsService shall be redefined each time that the router selects another Offer.

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:

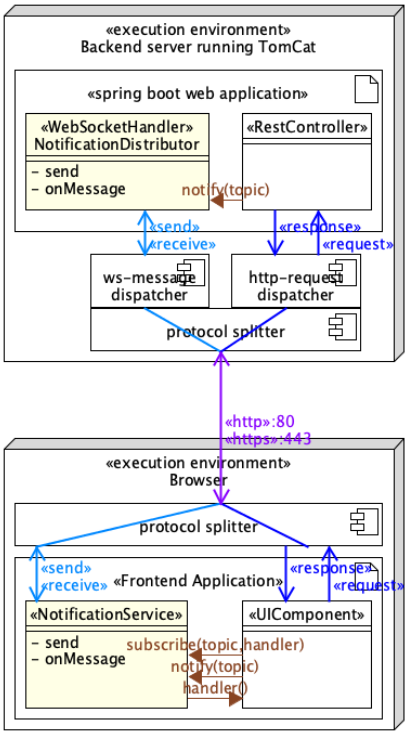
- You will ensure that the user interface of a user who is preparing a new bid will be updated automatically once more bids have been added in the backend.

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 **"offer-bids-12345"** identifies change in the bids on the offer 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. In our approach there will be one such connection per active user session (=browser tab, =application instance) on your client computer. The sessionsMap in the NotificationDistributor of the server 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 by SpringBoot dependency injection (enabled by the @Component annotation of the 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)

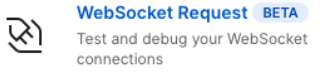
B. A Specific maven dependency and spring configuration bean is required to set up your notification distributor as part of the component scan at Spring boot application startup:

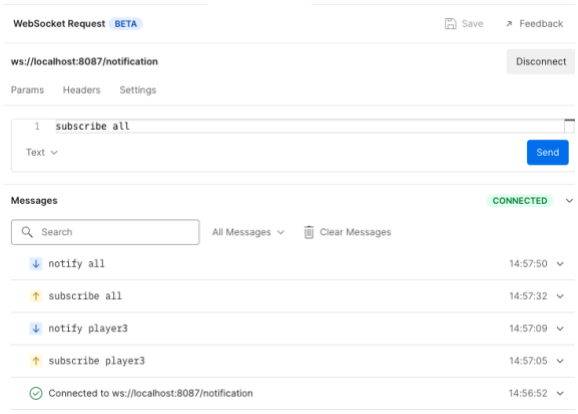
```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-websocket</artifactId>
</dependency>
```

```
@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()
        ;
    }
}
```

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:8083/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…

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  }
```


B. Create new bids/Overview46 and bids/Detail46 components which can be component extensions of bidsOverview45 and bids/DetailBid45 respectively. Create a new route ‘bids/overview46’ with parent component BidOverview46 and child component BidDetail46 and link the route to a new menu item.

If the user has the BidsOverview46 UI component open in the browser, that user wants an instant update of the latest bid information as soon as something changes in the backend. If you have properly structured your code of BidOverview45, you only need to add subscription to a notification for that:

```
4   export default {
5     name: "BidsOverview46",
6     extends: BidsOverview45,
7     inject: {
8       notificationService: { from: 'notificationService' }
9     },
10    created() {
11      //this.reLoad();
12      this.notificationService.subscribe("bids-on-offers", this.reLoad);
13    },
14    beforeUnmount() {
15      this.notificationService.unsubscribeAll(this.reLoad);
16    }
17  }
```

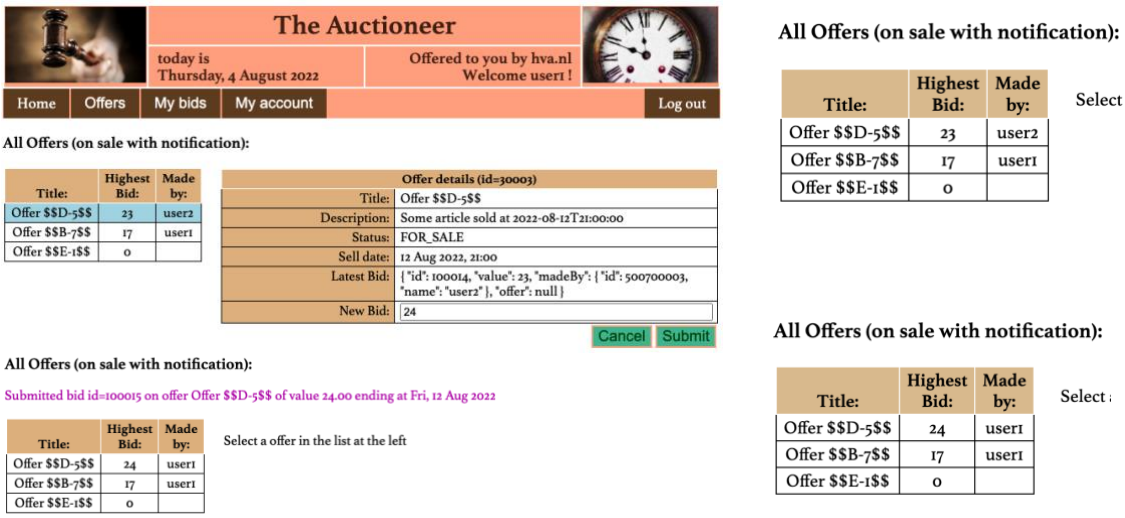
Vue life-cycle hooks are chained across extension, so if the BidsOverview45 invokes the reload method to retrieve all offers first time, here we then add subscription to a topic ‘bids-on-offers’ such that we reload all offers again when notified.

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…

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

```
@Transactional
@PostMapping(path = "{offerId}/bids", produces = "application/json")
public ResponseEntity<Bid> addBid(@PathVariable long offerId,
                                  @RequestBody Bid bid,
                                  @RequestAttribute(name = JWToken.JWT_ATTRIBUTE_NAME,
...
    Bid newBid = bidsRepo.save(bid);

    URI location = ServletUriComponentsBuilder.
            fromCurrentRequest().path("/{bidId}").
            buildAndExpand(newBid.getId()).toUri();

    this.notificationDistributor.notify("bids-on-offers");

    return ResponseEntity.created(location)
            .body(newBid);
}
```

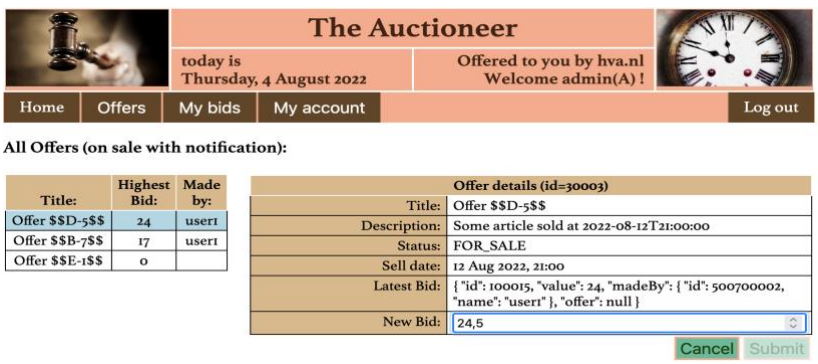
Test your notification implementation by opening two browser windows to your application. Navigate both to bids/overview46.



Now submit a bid request in one tab and observe how the bids information is immediately updated in the other tab without a manual page refresh.
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.

D. Now BidsOverview46 is instantly following all changes, but the Bid submission component BidsDetail46 will still be stuck at old information.
Override the reLoad(offerId) method in BidsDetail46 to subscribe to a “bids-on-offer-NNN” topic (with NNN equal to the id of the selected offer).
Add appropriate unsubscription.
Add the notification in the backend REST controller.
When BidsDetail46 is notified it should reload and update its specific Offer and Bids information, but not change user entered data.

Test your implementation:
Verify that the value of the new bid that the user may have entered is not reset by a reload of a notification.
Verify that the Submit button gets disabled if a reload of a notification sets a higher latest bid value.



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 repository.