Hogeschool van Amsterdam

2000WEF_19
2000DWEF19
Web Frameworks (vt, dt)

Assignment
Amsterdam Events

Version	: 20.3
Date	: 2 November 2020
Study tracks	: HBO-ICT, Software Engineering
Curriculum	: year 2, terms 1 & 2
Study guide	: vt: https://studiegids.hva.nl/co/hbo-ict-se-vt/100000026/049209 dt: https://studiegids.hva.nl/co/hbo-ict-se-dt/100000040/049531
Course materials	: vt: https://dlo.mijnhva.nl/d2l/home/197976 dt: https://dlo.mijnhva.nl/d2l/home/223020

Versions

Version	Date	Author	Description
19.26	2 maart 2020	John Somers	Backend assignment up to 4.5 Typo-s
20.1	3 Sep 2020	John Somers	Simplification, Content shuffle Upgrade to Angular 10
20.2	20 Sep	John Somers	Revisited 3.5 – 3.7, 4.1 – 4.5
20.3	2 Nov	John Somers	Siblings, PUT, references
21.1	13 Dec	Marcio Fuckner	4.5 JPA 3

Table of contents

1 Introduction

This document provides a casus description and a number of assignments for practical exercise along with the course Web Frameworks. The assignments are incremental, building some parts of the solution of the casus. 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/home/.

The frontend is well covered by Maximilian Schwarzmüller in:

O'Reilly-1. 'Angular 8 – The Complete Guide'

The backend is well covered by Ranga Karanam in:

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 'Amsterdam Events'. This application provides its users a platform at which they can publish events that are scheduled in the Amsterdam area and also allow visitors to register for participation. (Actually, the municipality of Amsterdam already provides such a kind of site at https://evenementen.amsterdam.nl/.)

2.1 Use cases

Visitors can search and navigate the site anonymously to find information about upcoming events. But, in order to actually register for participation or publish your own event, you need to register for an account and logon.

When you publish a new event, you must provide a title, some description, start and end date and time, a location and at least one category in which the event shall be classified. You have the option to add pictures to your event and provide a caption along with each picture.

You have the option to mark your event as 'ticketed', which opens up the event for registration by users who want to participate. In that case you must provide the maximum allowed number of participants and a participation fee (which can be zero if participation is free of charge).

Every event has a status of 'Draft', 'Published' or 'Cancelled'. Initially an event will have status 'Draft' while the organiser is still preparing information. Draft events are not visible to any other user than the organiser and admin users. When ready, the organiser changes the status to 'Published'. Only then visitors and other users can review the information of the event and register for participation.

New users can only register for an event if the number of registered users has not reached the maximum number of participants yet. When a user registers for an event that has a positive participation fee, the system will send the user an e-mail with a ('tikkie') payment

request. Once the user has paid the fee, the system will generate the (unique) ticket code, which will show as a QR code on the user's smart device and will grant access at the day of the event. If the participation fee is zero, then the ticket code will be generated immediately after registration by the user.

Users can cancel their registrations until one week before the start of the event. If you cancel a paid registration, your fee will be refunded, minus 10% administration cost. Free spaces that follow from cancellations are available for other registrations, but then new, unique ticket codes will be generated.

Events can be cancelled by the organiser or an admin user at any time, also after the event seems to have finished. When a paid event is cancelled, all registered users get the full fee of their payment back.

At any time, the organiser or an admin user can change the information of the event and cancel registrations. Change of participation fee only applies to new registrations thereafter. Start and end dates of ticketed events cannot be changed, and a ticketed event cannot be changed into an un-ticketed event, if registrations have been recorded.

The list of categories can be managed by admin users. A category can only be removed if it has no associated events.

2.2 Class diagram

Below you find a navigable class diagram of the functional model that has been designed for 'Amsterdam Events'. 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!

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 nFree attribute is a derived attribute which can be calculated from maxParticipants minus the number of registrations.

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 Events it has organised and every Event knows by which Account it has been organised.
- Uni-directional navigability: Every Account knows which Registrations it has made, but a Registration has no clue for which Account it has been made.

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.events realises the navigability of the 'organises' relation. It provides the list of all Events that have been organised by a given Account. Event.organiser realises the navigability the other way around. It provides the organiser Account of a given Event.

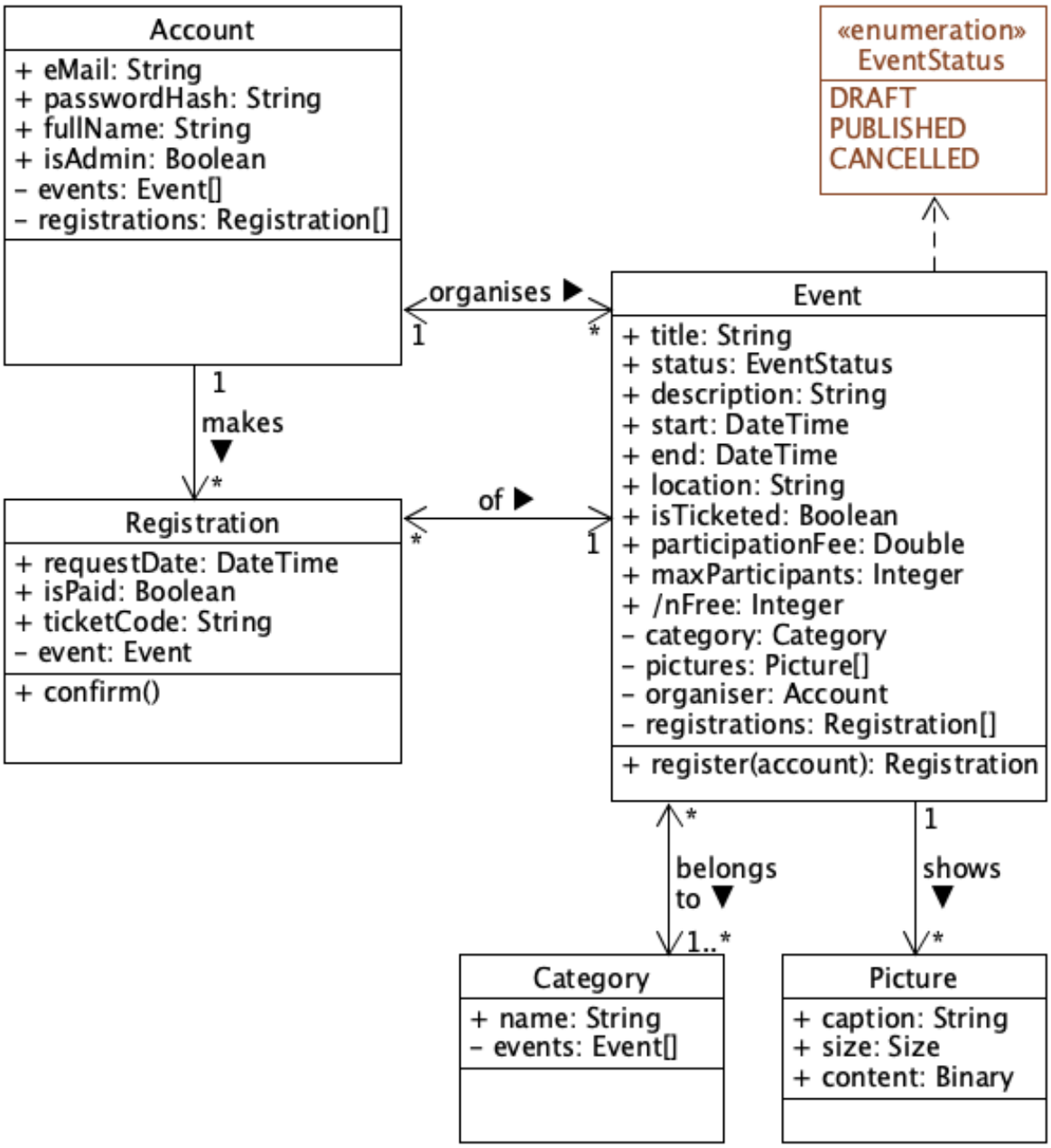


Figure 1: Navigable Class Diagram 'Amsterdam Events'

2.3 Layered Logical Architecture

Figure 2 depicts the designated full-stack, layered logical architecture of 'Amsterdam Events'. The frontend user interface layer shows the Model-View-Controller interdependencies of your Single-Page web application. The UI Service package provides the adaptors connecting the frontend with the backend RESTful web service.

The Functional Model is shared between the frontend and the backend, indicating that a single consistent model of the business entities shall be implemented. As you will be using different programming languages in for the frontend and the backend, you also will provide a dual implementation of this functional model.

The REST Service will integrate the Hibernate Object-to-Relational Mapper (ORM) by means of Dependency Injection of Repository Services. (One for each class in the Functional Model).

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 RDBMS.

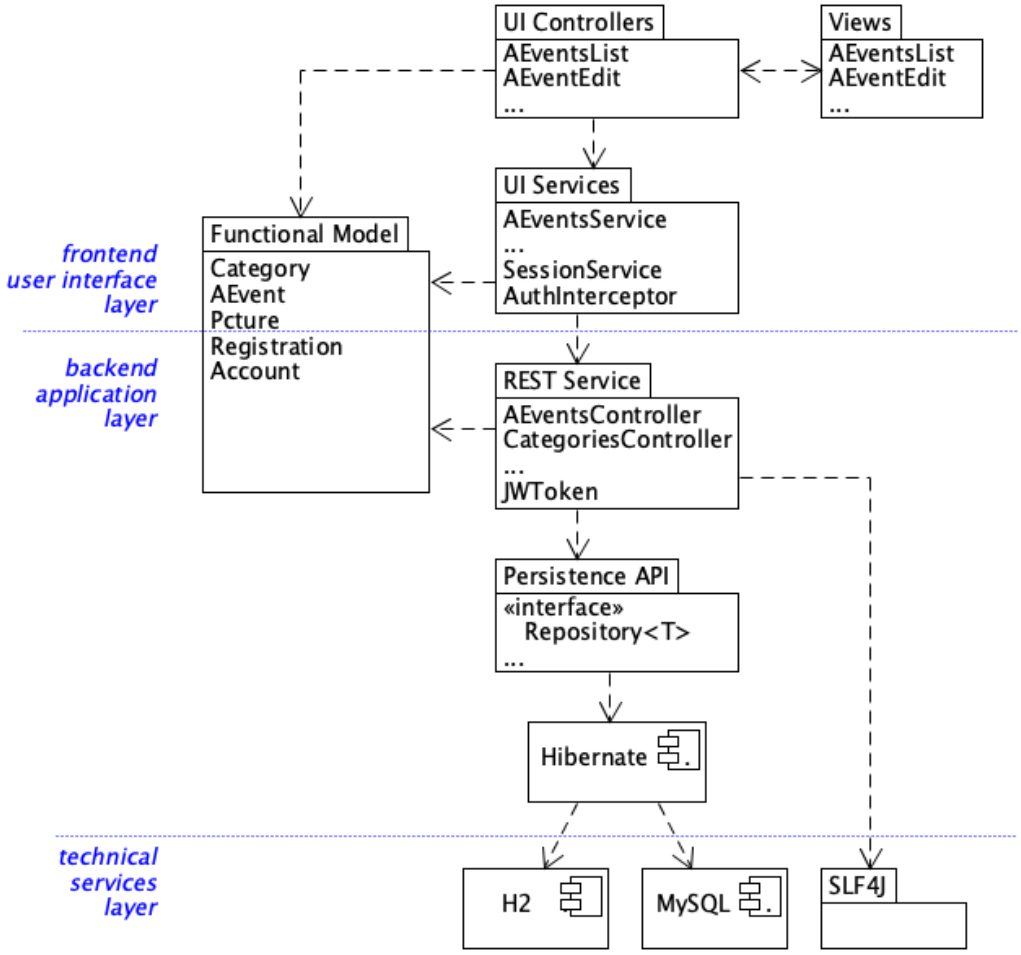


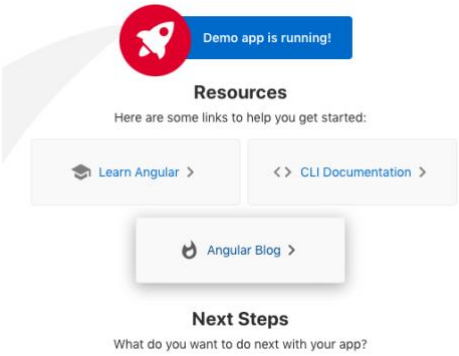
Figure 2: Full-stack Logical Architecture

3 First term assignments: Angular 10 and Spring Boot

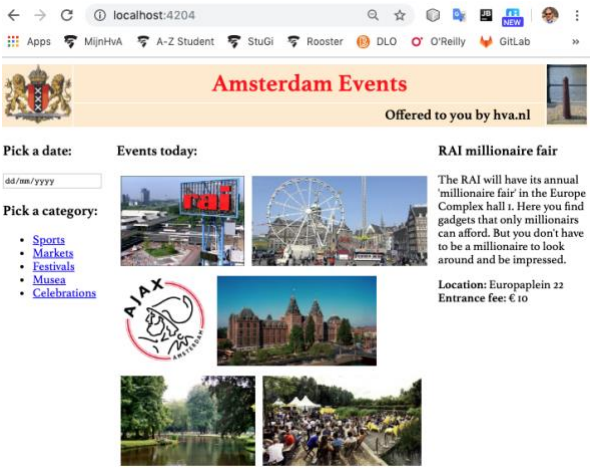
3.1 Amsterdam Events Home Page

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 an Angular 'Single Page Application' project for 'AmsterdamEvents':
   $ ng new AmsterdamEvents
   or New Project → JavaScript → Angular CLI
   Test your project by running 'start' from the package.json file.
   Open your browser on http://localhost:4200/
   Your application shall show in in the browser as in the image provided here.



B. Setup the src/app/components/mainpage source directory structure and define in there a header component and a home component for the page:
   $ ng g c components/mainpage/header
   $ ng g c components/mainpage/home
   or New → Angular Schematic → Component
   in the context menu of src/app/components/mainpage
   Replace the content of your app.component.html such that is shows the header at the top of the main page, and the home page below that. First practice some plain HTML/CSS to produce a view similar to the picture below by only changing header.component.html, header.component.css, home.component.html, home.component.css and possibly src/styles.css

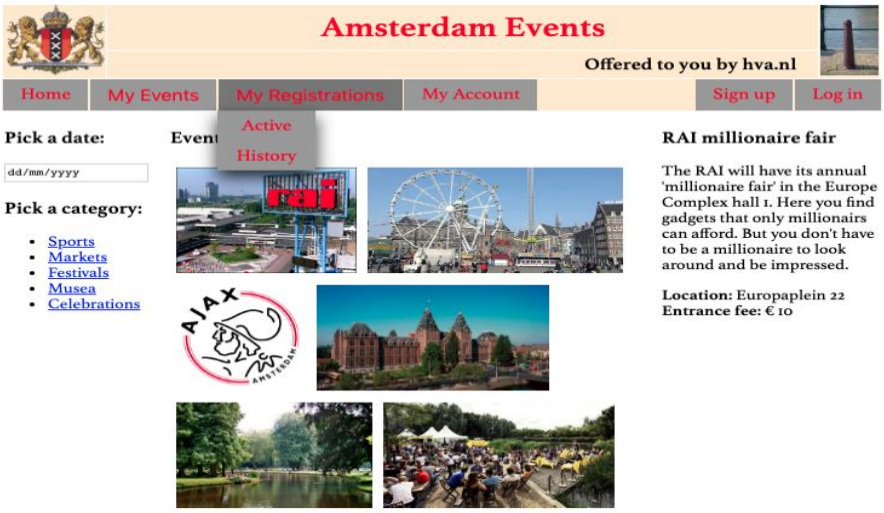


C. Design the header.component.html template to hold a title, sub-title and two (logo) pictures left and right. Store the pictures in src/assets/images.
Make sure the header is responsive such that its pictures use fixed size and the title space scales with the size of the window. The text of the subtitle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

D. Design the home.component.html 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.

E. Add another component ‘nav-bar’ to src/app/components/mainpage. Show this component just below the header in app.component.html. This navigation bar should be able to provide responsive menu items and sub-menus similar to the example picture above. (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 dynamic color highlighting and sub-menu expansion effects when navigating the menus.

3.2 Amsterdam Events overviews

In this assignment you explore the TypeScript controller code and data of an angular component. You define model classes to properly define and organise the functional entities of your application. You apply structural directives, all four methods of binding and the principal method of component interaction.

3.2.1 A list of Events

You apply the *ngFor directive, “String Interpolation binding’ and ‘Event binding’ to connect a simple html view to controller code and data.

A. Retrieve and format today’s date and time into the header.component.ts controller and use ‘interpolation binding’ to display the outcome at the left of the sub-title. (You may want to use the options parameter in the Typescript/Javascript function Date.toLocaleString for this.)

Id	Title	Start	End	Status	Entrance Fee	Max Participants
20001	The fantastic event-20001	Mon, 12 Oct 2020, 14:00	Mon, 12 Oct 2020, 22:30	PUBLISHED	€ 4.50	7000
20006	The fantastic event-20006	Tue, 24 Nov 2020, 13:00	Thu, 26 Nov 2020, 12:00	DRAFT	free	
20010	The fantastic event-20010	Wed, 4 Nov 2020, 13:30	Sat, 7 Nov 2020, 20:30	PUBLISHED	€ 15.00	200
20015	The fantastic event-20015	Fri, 30 Oct 2020, 13:00	Mon, 2 Nov 2020, 19:30	DRAFT	free	
20022	The fantastic event-20022	Mon, 16 Nov 2020, 13:30	Wed, 18 Nov 2020, 10:30	PUBLISHED	€ 3.00	3000
20027	The fantastic event-20027	Sun, 27 Sep 2020, 14:30	Thu, 1 Oct 2020, 15:30	DRAFT	€ 20.00	400
20031	The fantastic event-20031	Sat, 12 Dec 2020, 13:00	Tue, 15 Dec 2020, 12:30	PUBLISHED	free	
20034	The fantastic event-20034	Fri, 2 Oct 2020, 14:30	Sun, 4 Oct 2020, 05:30	PUBLISHED	€ 15.00	300
20035	The fantastic event-20035	Thu, 1 Oct 2020, 14:00	Fri, 2 Oct 2020, 07:30	PUBLISHED	€ 10.00	400

B. Setup the app/models source directory.
Define in there the a-event.ts model class. ($ ng g cl models/a-event). (We prepend the ‘a-‘ to the class filename to avoid possible conflict with the Event class in the JavaScript libraries). Specify the following class attributes:

title: string;	status: AEventStatus;
start: Date;	IsTicketed: Boolean;
end: Date;	participationFee: number;
description: string;	maxParticipants: number;

AEventStatus is a string-based Enum with values “DRAFT”, “PUBLISHED” and “CANCELED”.

Within the AEvent class, you implement a method:

```
public static createRandomAEvent():AEvent
```

which you will use to instantiate some a-events with random data for test purposes.
Make sure that each created a-event has a unique id starting with 20001
Use Math.random() to randomise the other content of each created a-event, i.e.:

- Provide a mix of statusses
- Provide a mix of future start dates and durations
- Some events are ticketed, some are not
- only ticketed events have a fee and a maximum number of participants

C. Setup the app/components/events source directory and define in there an ‘overview1’ component. Create a local list of nine events. Provide a method `addRandomAEvent()` which uses `AEvent.createRandomAEvent()` to add another event to the component’s list.

```
export class Overview1Component
                implements OnInit {

    public aEvents: AEvent[];

    constructor() {
    }

    ngOnInit() {
        this.aEvents = [];
        for (let i = 0; i < 9; i++) {
            this.addRandomAEvent()
        }
    }
```

D. Display the event data within a <table> in the HTML-template of the component.
Use the *ngFor directive on <tr>.
Use 'interpolation binding' on the event properties.
Use CSS to style the table.
If the event is not ticketed, the event fee shall report 'free' and no maximum number of participants shall be shown.
Replace the 'home' view in the app-component main view by your events list view.

E. Add a button at the bottom right of the view with text: 'Add Event'
use 'event-binding' on the button that binds its 'click'-event to the component function addRandomAEvent(). Test your application by adding some events.

3.2.2 Master / Detail component interaction

In this assignment you apply 'Property binding' and 'Two-way' binding between the html view and the controller data of a component. You explore inter-component interaction by means of event emitters with @Output decorators and @Input decorators on properties.

A. Add two more components 'overview2' and 'detail2' to app/src/components/events.
overview2 shall manage a list of events displaying only their titles, similar to the events list of assignment 3.2.
Implement the selection mechanism by binding the click event on every row in the titles overview at the left. The overview2 components tracks the 'selectedAEvent'. When selectedAEvent==null, nothing has been selected yet. Use CSS to highlight the AEvent that has been selected in the list.

B. Overview2 embeds the detail2 component in a right panel '<app-detail2>', which shall eventually provide all details of the selected event. While nothing is selected, the detail2 component shall display a simple message. (Use *ngif with an else alternative for this in the detail2.component.html template)

C. Otherwise detail2 component manages an edited AEvent. Include the id of the edited event in the header text of the detail2 component and the other properties below that.

Use appropriate <input type="…"> or <select> elements in its html template that bind to the properties of the edited AEvent. (Use two-way binding and don't forget to import the FormsModule.)
(You may omit the Date properties for now, because its two-way binding is more complicated).
Use property binding within the <app-detail2> element of the overview2

template to link the selected AEvent in overview2 with the edited AEvent in detail2. Use an appropriate @Input decorator in detail2 and test whether you can edit the events in the list.
When you edit the title of an AEvent in the detail, also the selected title in the events list may change instantly.
When you navigate among different events, any changes that you made to them will be remembered.
When a new AEvent is added via the 'Add Event' button at the left, that event should automatically get the selection focus (and be associated with the detail2 component).

Add a 'Delete' button to detail2 which the user can click to remove the AEvent altogether. Provide an EventEmitter with an @Output decorator rom the detail2 and use eventbinding within the <app-detail2> element of the overview2 template to catch the Angular event. Provide appropriate code in overview2 to find and delete the AEvent from the list (by id). After deleting an AEvent, it is also removed from the display at the left, and nothing is selected anymore.
(At https://levelup.gitconnected.com/angular-7-share-component-data-with-other-components-1b91d6f0b93f you find another basic tutorial about @Input and @Output decorators.)

3.3 Using a service and custom two-way binding.

In this assignment you will setup a service to manage events data that is to be shared and injected into multiple components. Then the lifecycle of the data has become decoupled from the lifetime of the UI components and you can just synchronise the AEvent id-s among components without carrying all the data. You will use a true copy of the service's object for the real time editing, such that you can implement a cancel operation on changes and warn for unsaved changes in the form when the selection is about to change.

```
@Injectable({
  providedIn: 'root'
})
export class AEventsService {

    public aEvents: AEvent[];

    constructor() {
        this.aEvents = [];
        for (let i = 0; i < 9; i++) {
            this.addRandomAEvent()
        };
    }

    /*
     findAll(): AEvent[] {
         // TODO return the list of all aEvents
     }
     findById(eId: number): AEvent {
         // TODO find and return the aEvent with the specified id
         // return null if none is found
     }

     save(aEvent: AEvent): AEvent {
         // TODO replace the aEvent with the same id with the provided
         // return the old, replaced aEvent
         // add the new aEvent if none existed and return null
     }

     deleteById(eId: number): AEvent {
         // TODO remove the identified aEvent from the collection
         // and return the removed instance
         // return null if none existed
     }
     */
```

A. Setup the src/app/services source directory. Create in there an a-events service, that will manage the collection (array) of available a-events.
($ ng g s services/a-events)
(A single instance of this AEventsService will be provided in 'root' and can be injected where needed in all other services or components.)
Implement four basic CRUD operations in your service:
findAll():AEvent[] retrieves the list of all a-events
findById(id): AEvent retrieves one a-event, identified by a given id
save(aEvent): AEvent saves an updated or new a-event and returns the previous instance with the same id, or null if no such aevent existed yet.
deleteById(id): AEvent deletes the a-event identified by the given id, and returns the a-event that was deleted, or null if none existed.
The choice of these signatures of CRUD operations aligns with the functionality of the RESTful web services, and the Java Persistence API which we will use later in the implementation of the backend.

B. Define an overview3 and detail3 component in app/components/events, which provide the same functionality as overview2 and detail2 of assignment 3.2.2, but now use the CRUD operations of the a-events collection as provided by aEventsService.
(Inject the aEventsService instance into both the overview3 and detail3 components via their constructors).
Overview3 shows a list of titles as provided by the injected aEventsService.findAll().
Detail3 retrieves the a-event to be edited from aEventsService.findById(id).

C. Maintain a selectedAEventId in overview3 and an editedAEventId in detail3 and implement two-way custom property binding between them in the <app-detail3> element in the overview3.component.html template:

```
<app-detail3 [(editedAEventId)]="selectedAEventId"></app-detail3>
```

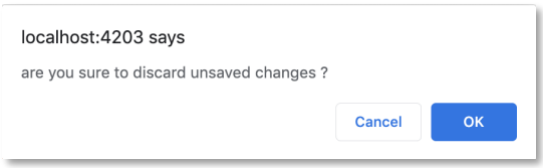
Now either side can change the selected id, and the other side should follow:
1.If the user selects another a-event in the list of overview3 at the left, detail3 should follow by loading the selected event for editing.
2.If the user deletes of saves the a-event being edited in the detail3 component at the right, that component will thereafter un-select that a-event (e.g. by setting editedAEventId = -1) and also physically removed from the service. As a consequence, it should also disappear from the overview (which has got injected the same a-events service).

This use of custom two-way binding requires a specific naming convention to be followed for the shared property and the change-event emitter.
(See: https://angular.io/guide/template-syntax#two-way-binding-)
(See: https://angular.io/guide/component-interaction#intercept-input-property-changes-with-a-setter)

Show the new overview3 component from your app-component. Thoroughly test correct behaviour of your solution.

D. Implement additional buttons ‘Save’, ‘Clear’, ‘Reset’ and ‘Cancel’ in detail3 with the following functionality:
‘Save’ will update the aEventsService with the currently edited a-event.
‘Clear’ will setup a new clean AEvent in the detail3 panel, but retain the a-event id.
‘Reset’ will discard all changes and reload the form with the original values from aEventsService.
‘Cancel’ will discard al changes and cancel the edit, whereafter no details shall be shown and no title shall be selected and highlighted anymore.
(Hint: make sure that detail3 operates on a true copy of the selected AEvent, and not directly on the data in the service)

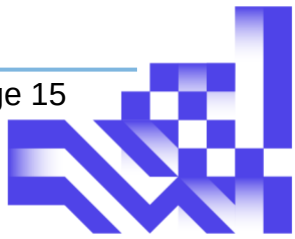
E. Implement unsaved changes detection in the detail3 component that will pop-up a confirmation box when the ‘Delete’, ‘Clear’, ‘Reset’ or ‘Cancel’ button is pressed while the form has unsaved changes.



If the user confirms with ‘OK’ the changes may be discarded. If the user presses Cancel the focus shall remain on the earlier selected AEvent, and the changes in the form shall be retained.
(Use the JavaScript confirm() function to popup this confirmation box before possible loss of changes.)
(Use object value comparison to check on differences between the edited object and the original object held by the service).
(This unsaved changes protection may not work yet, when another a-event is being

selected in the title list, or when the user navigates to a different URL altogether. That will be added in a later bonus assignment 3.4.3.)

F. 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 other buttons are always enabled.)

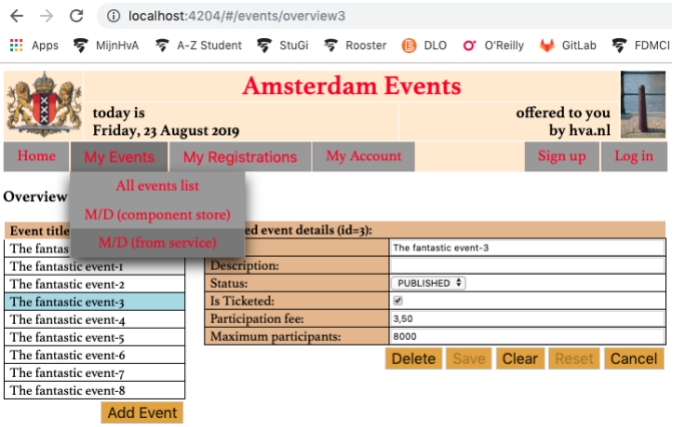


3.4 Page Routing

In these assignments we will provide navigation capabilities to our application, such that all of our pages can be found from a navigation menu bar, and users can bookmark the url-s of specific pages in your application

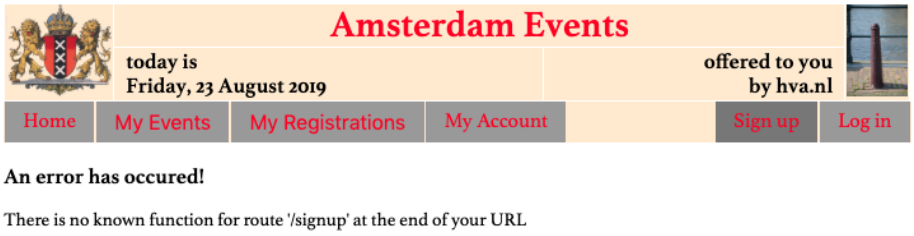
3.4.1 Basic Routing

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



A. Implement page routing from the app-module.ts and provide routes for the three AEvents components that you have created in previous assignments:
1. ‘home’ shows your welcome page of assignment 3.
2. ‘events/overview1’ shows the events list of assignment 3.2.1.
3. ‘events/overview2’ shows the master/detail of assignment 3.2.2 with the AEvents stored in the component.
4. ‘events/overview3’ shows the service-based AEvents master/detail of assignment 0.
Also provide a redirect from ‘/’ to the ‘home’ route.
Configure the ‘useHash’ option to separate the angular routes from the base in the browser’s url.
Configure the <router-outlet> in your app component.
Link these options to menu items in your navigation bar.

B. Connect the ‘Sign Up’ menu item to the ‘signup’ route and the ‘Log in’ menu item to the ‘login’ route, without providing these routes in the routes table.
Add a new ‘error’ component in src/app/components/mainpage with a simple error message, which indicates that a specified route is not available, just as in the example below. This component should be connected to any unknown route.



3.4.2 Parent / child routing with router parameters.

In order to be able to bookmark or share an URL for editing of a specific a-event, we will integrate selection of an a-event into the router. For that we use parent / child routing with router parameters.

A. Create two new components overview4 and detail4 in src/app/components/aevents which can be copies of overview3 and detail3 initially. Connect overview4 to a new route aevents/overview4 and also add the route to the 'My Events' sub-menu.

B. The two-way custom property binding between the Overview4Component.selectedAEventId and Detail4Component.editedAEventId can be removed. Instead, you create a child sub-route ':id' for the detail4 component under the aevents/overview4 parent route and also create the child <router-outlet> in the overview4.component.html.
If the user now selects a different *aevent* in the list of titles, you call the navigate method on the router towards the sub-route *aevent.id*
(For that you need to inject the router and the activatedRoute in the constructor of overview4).

```
onSelect(eId:number) {
    // activate the details page for the given a-event Id
    this.router.navigate([eId], { relativeTo: this.activatedRoute });
}
```

C. The detail4 component needs to extract the editedAEventId from the router parameter and retrieve (a copy of) the associated AEvent to be edited from the service. The value of the router parameter is available from the activatedRoute. The value of the router parameter may change in the future each time when the user selects different AEvent. The detail4 component is instantiated and initialised only once. Hence we must configure it to reinitialise its editedAEvent each time when the router parameter is touched. That we achieve by subscribing to the params observable in the activatedRoute.

```
constructor(public aEventsService: AEventsService,
            public router: Router,
            public activatedRoute: ActivatedRoute) {
}

private childParamsSubscription: Subscription = null;

ngOnInit() {
    // get the event id query parameter from the activated route
    this.childParamsSubscription =
        this.activatedRoute.params
            .subscribe((params: Params) => {
                console.log("detail setup id=" + params['id']);
                // retrieve the event to be edited from the service
                this.setEditedAEventId(params['id'] || -1);
            });
}

ngOnDestroy() {
    // unsubscribe from the router before disappearing
    this.childParamsSubscription &&
        this.childParamsSubscription.unsubscribe();
}
```

Here you find a snippet of sample code that could do that job. It injects the router and the activatedRoute in the constructor of detail4 and subscribes to the Observable to be notified about any change in the router parameters. In the method setEditedAEventId() you shall then obtain the associated AEvent object for editing from the aEventsService.
(Strictly it is not necessary to unsubscribe from activatedRoute observables, but we do it anyway as a good habit).
(This code does not use the value from activatedRoute.snapshot.params['id']. It appeared that the observable also provides an event immediately after initialisation of the component that is identical to value in the snapshot. You may want to test this to be sure…)

D. Test whether all navigation works fine, and also whether you can drive full navigation of overview4 and detail4 just by editing the url in the browser address bar. It is well possible that the highlighting of the selected aEvent title in the list will not always follow the URL or the details section. For that you also need to observe the 'id' router parameter in the overview4 component itself at activatedRoute.firstChild.params, and synchronise it with the selectedAEventId.
That is a bit more complicated though, because if no child id has been provided in the route, no detail4 will not have been instantiated yet and activatedRoute.firstChild will be 'undefined' so that you cannot subscribe to its param observable. (You can work around this issue by adding a convenient redirection in the children routes table from the empty root path to a dummy child id, e.g. '-1'.)

E. Fix the 'Save', 'Delete' and 'Cancel' operations in detail4 and unselect the edited aEvent by navigating to the parent of the activatedRoute without a router parameter. Before, in the detail3 component you would have emitted an event to notify the parent about un-selection of the aEvent.
If you have done well, the unsaved changes detection as explained in assignment 0.E, is still working. Also the '[disabled]' properties on the buttons still work fine as explained in 0.F. If these are not working anymore, you should repair them.

3.4.3 [BONUS] Unsaved changes protection with router guard 'canDeactivate'.

It was possible to protect against unintended loss of edited changes while the user was using the controls to navigate within the master / detail overview, but no protection was provided when the user changes the address in the browser manually or navigates to other parts of the application or even to another site. The Angular router provides for configuration of a 'canDeactivate' guard to protect that.

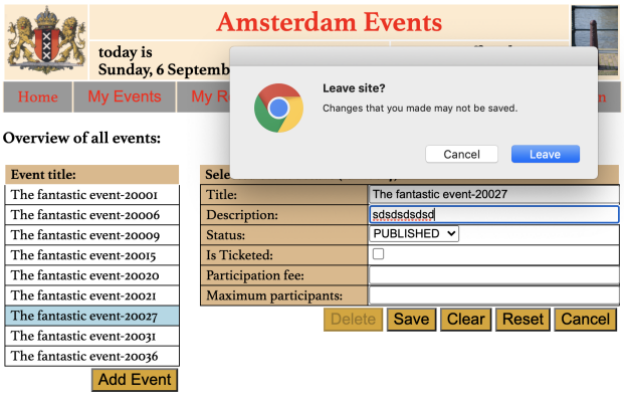
A. Create one new component detail41 in src/app/components/aevents which extends the component detail4 of assignment 3.4.2 and reuses in its @Component decorator the templateUrl and styleUrls of detail4. Initially there is no need for a constructor or other methods or attributes in detail41.
There will be no change in the interaction with overview4, so you can fully reuse that component.

Create a new route 'events/overview41' that opens overview4 and add a child route which opens the new component detail41. Provide the new route 'aevents/overview41' in the 'My Events' sub-menu.

B. Implement the CanDeactivateGuardService in src/app/services and the canDeactivate method in the detail41, leveraging your method to check dirtiness. 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 canDeactivate and one time by your own checks). Re-test all use of the buttons and also the '[disabled]' properties on them.

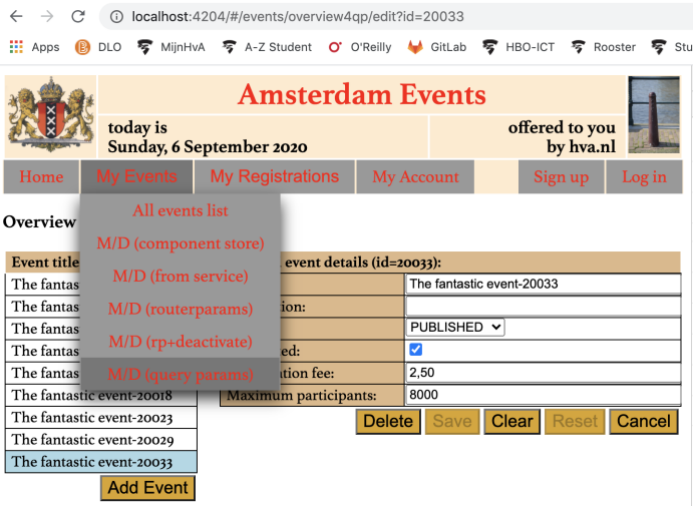
C. Unfortunately, canDeactivate only traps navigation within your application, not if the user navigates to a completely different site. For that, stewdebaker has proposed a nice solution at https://stackoverflow.com/questions/35922071/warn-user-of-unsaved-changes-before-leaving-page. Apply this solution to your implementation and test whether all navigation still works fine, all use of all buttons and also the '[disabled]' properties on the buttons.



3.4.4 [BONUS] Parent / child communication with query parameters

A alternative approach to maintain synchronisation of the selected a-event Id between the overview master and the details editor component is to use query parameters along with the router instead of router parameters.

A. Create one new component detail4qp in src/app/components/aevents which extends the component detail4 of assignment 3.4.2 or detail41 of assignment 3.4.3 and reuses in its @Component decorator the templateUrl and styleUrls of detail4. Initially there is no need for a constructor or other methods or attributes in detail4qp. There will be no change in the interaction with overview4, so you can fully reuse that component.
Create a new route 'aevents/overview4qp' that opens overview4 and add a child route with path = 'edit' which opens the new component detail4qp. Provide the new route 'aevents/overview4qp' in the 'aevents' sub-menu.



B. Now, the detail4qp component needs to extract the editedAEventId from the activatedRoute. For that, you can use similar code as provided with assignment 3.4.2.C, but now subscribe to the 'queryParams' observable instead of the 'params' observable. If you do well, you only need to override ngOnInit and ngOnDestroy in the extended Detail4qpComponent class and inherit all other functionality.
The same applies to the Overview4qpComponent, except that you also may need to

override the onSelect method, because child routes have been changed, and query parameters shall be used to pass the selected aevent id:

```
onSelect(eId: number) {
    // activate the details page
    this.router.navigate(['edit'], {
            relativeTo: this.activatedRoute,
            queryParams: {id: eId}
        }
    );
}
```

Test whether all navigation and unsaved changes detection still works fine, all use of all buttons and also the '[disabled]' properties on the buttons. (Specifically test the Cancel button both on clean and edited forms.)

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 'Amsterdam Events' application as depicted in the full-stack, layered logical architecture of section 2.3. You will use the Spring-Boot technology.
Relevant introduction and explanation about this technology can be found in the video O'Reilly-2 by Ranga Karanam at https://learning.oreilly.com/home/:

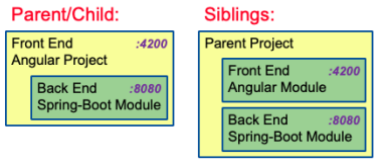
As of assignment 3.7 you will integrate the backend capabilities with your frontend solution of assignment 3.4.

First 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 a-events of your application. These a-events are actually managed in the Spring-Boot backend by an implementation of an AEventsRepository Interface. The AEventsRepository is injected into the REST Controller by setter dependency injection (O'Reilly-2, Chapter 3, step 7).

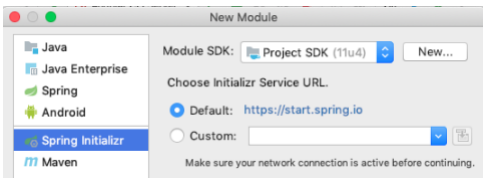
For now, you start with providing an AEventsRepositoryMock bean implementation that just tracks an internal array of aEvents, similar to how you did that before in an Angular service of your FrontEnd. In a later assignment you will provide a 'bean' implementation of this interface that links with H2 or MySql persistent storage via the Hibernate Object-To-Relational Mapper.

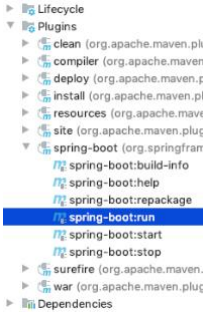
Below you find more detailed explanation of how you can implement the objectives of this assignment.

A. Basically, there are two approaches to combine a frontend and a backend module in an integrated development environment.
i) A backend module within a frontend project.
ii) Two separate projects as siblings in a parent folder.
Here we choose the first approach, because that is a straightforward extension of the work of earlier assignments.
(If you wish to configure CI/CD deployment in a real project the Siblings approach is a better option.)



Create a Spring-Boot application module 'aeserver' within the Single Page Application project.
Use the Spring Initializr plugin.
(You may need to install/activate the Spring plugins first in your Intellij settings/preferences).



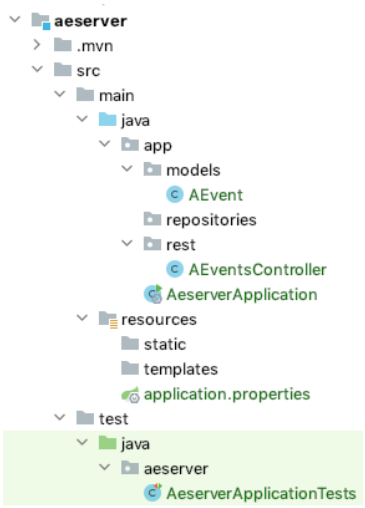


Choose the war format for your deployment package.
Activate the Spring Web dependency in your module.
Also check that Maven framework support is added.
Test your project setup by running the 'spring-boot:run' maven goal.

You may want to configure debug mode or adjust the tomcat portnumber in resources/application.properties:

```
server.port=8084
logging.level.org.springframework = debug
```

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 AeserverApplication 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'

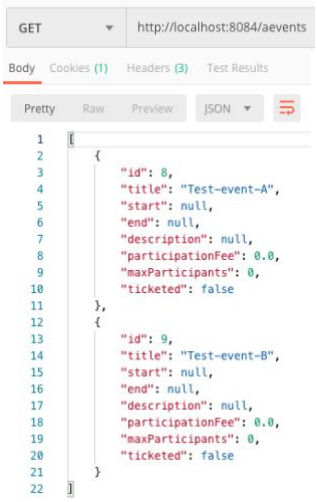


You may need to fix 'Sources root', 'Test sources root' and 'Resources root' designation of marked IntelliJ folders, if those were not picked up automatically.

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 with File->New->Module From Existing sources.

C. Implement the AEvent model class and the AEventsController rest controller class, as explained in O'Reilly-2.
Replicate your AEvent model from the frontend a-event.ts code.
Implement in the AEventsController a single method 'getAllAEvents()' that is mapped to the '/aevents' end-point of the Spring-Boot REST service. This method should return a list with two a-events like below.

```
public List<AEvent> getAllAEvents() {
        return List.of(
                new AEvent("Test-event-A"),
                new AEvent("Test-event-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 a-events like you expect.

D. Define an AEventsRepository interface and an AEventsRepositoryMock bean implementation class in the repositories package similar to the SortAlgorithm example of Ranga. Spring-Boot should be configured to inject an AEventsRepository bean into the AEventsController.

The AEventsRepositoryMock bean should manage an array of aEvents, similar to how your Angular AEventsService was managing the AEvent data. Let the

constructor of AEventsRepositoryMock setup an initial array with 7 a-events with some random data.

The static method to create some random aEvent can best be implemented in the AEvent class itself.

```
public static AEvent createRandomAEvent() {
    AEvent aEvent = new AEvent();

    // TODO put some random values in the aEvent attributes
```

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

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

```
public interface AEventsRepository {
    public List<AEvent> findAll();
}
```

```
public List<AEvent> getAllAEvents() {
        return repository.findAll();
}
```

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

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 /aevents REST-api with endpoints to create new a-events and get, update or delete specific a-events. 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.

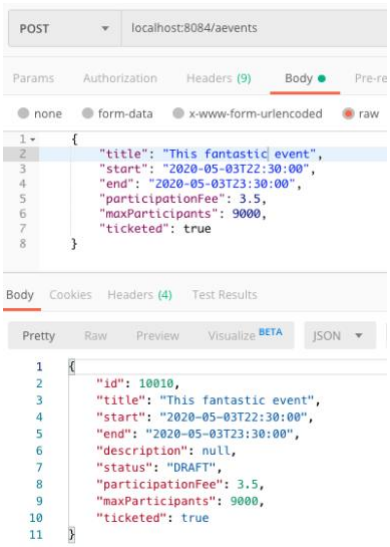
In this assignment you should practice hands-on experience with Spring-Boot annotations @RequestMapping, @GetMapping, @PostMapping, @DeleteMapping, @PathVariable, @RequestBody, @ResponseStatus, @JsonView and @Configuration and classes ResponseEntity, ServletUriComponentsBuilder.

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

```
public interface AEventsRepository {
    List<AEvent> findAll();          // finds all available a-events
    AEvent findById(long id);        // finds one a-event identified by id
                                     // returns null if the a-event does not exist
    AEvent save(AEvent aEvent);      // updates the a-event in the repository identified by aEvent.id
                                     // inserts a new a-event if aEvent.id==0L
                                     // returns the updated or inserted a-event with new aEvent.id
    boolean deleteById(long id);     // deletes the a-event from the repository, identified by id
                                     // returns whether an existing a-event has been deleted
}
```

A. Enhance the AEventsRepositoryMock class with actual implementations of all CRUD methods as listed in the AEventsRepository 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 aEvent-id with the available a-events in the private storage of the AEventsRepositoryMock instance.

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

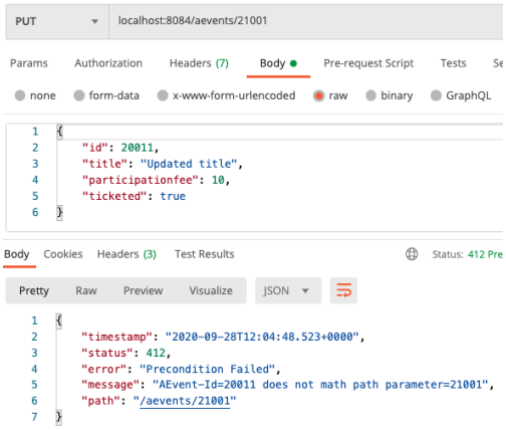


Test the new mappings with postman.

C. Use the ResponseEntity and ServletUriComponentsBuilder classes to return a response status=201 and Location header in the response of your aEvent creation request. Use the .body() method to actually create the

ResponseEntity such that it also includes the aEvent with its newly generated id for the client.

Test the mapping with postman.



D. Implement Custom Exception handling in your REST AEventsController:
- 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 aEvent in the request body.

Test the mapping with postman.

E. Implement a dynamic filter on a getAEventsSummary() mapping at '/aevents/summary', which only returns the id, title and status of every a-event. Dynamic filters can most easily be implemented with a @JsonView specification in your AEvent 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 with HTTPClient requests

In this assignment you will explore Angular HTTP requests to implement the interaction between your frontend application and the backend REST API. Basically, there are two approaches to this:

i. You refactor the implementation of your service such that it caches relevant a-events data from the backend and uses HTTP requests and responses to maintain cache consistency.

ii. You refactor your UI components and the implementation of your service such that the service provides an adaptor for accessing the backend REST API without caching any data. The UI-components will handle the a-synchronous responses from the backend.

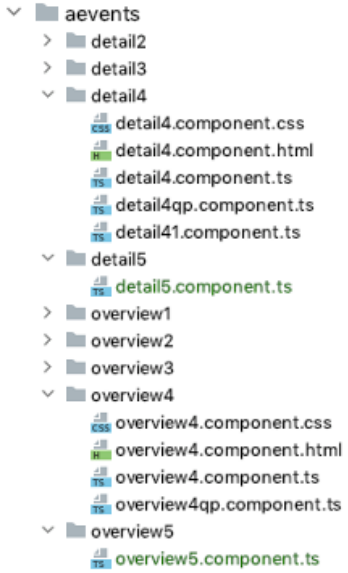
The advantage of option ii) is that your UI will always show up to data information, also if multiple users are accessing the backend concurrently.

The advantage of option i) is that we can address almost all of the interaction within the Angular service, and your UI components only require minimal change.

Below we will provide directives according to option i) but you may choose to implement option ii)

You will add @angular/common/http, rxjs to the frontend and WebMvcConfigurer and @Configuration in the backend.
You will configure the CORS across the full stack to facilitate use of multiple ports for different backend services.

A. Replicate ‘overview4’ and ‘detail4’ components of assignment 3.4 (or bonus versions thereof) into new overview5 and detail5 components.
(As we will apply only minor changes to these components, you only need to replicate the .ts files and may reuse the .html and .css templates).
Replicate the a-events.service into a new a-events-sb.service. We will be refactoring that service to connect to our backend REST service with http requests.
Change to overview5 and detail5 components to inject the new AEventsSbService in their constructors.



Create a new route aevents/overview5 in your router module, and configure that route in your navigation-bar.
First test whether your overview5 is still working in the same way as the original overview4

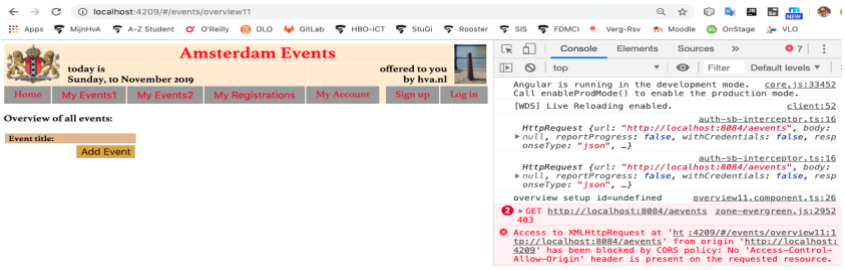
B. Provide HttpClient (from "@angular/common/http") in your app-module.ts and inject its instance into AEventsSbService.
Add a private method
restGetAEvents(): Observable<AEvent[]>
to the AEventsSbService.
This method issues an http get request to your backend endpoint for all a-events

(http://localhost:8080/aevents), and handles the response by caching all retrieved a-events in the local array in the service.
Call this method from the constructor of the AEventsSbService, such that upon instantiation of this singleton service object, it is immediately populated with data from the backend.
For now, make sure that any error conditions from the http requests are logged onto the browser console.

Notice, that the Json response of an http request gives you all a-event data, but does not deliver a true AEvent object as per your models/a-event.ts class definition. These Json data objects do not know their methods…
A systemic way to address this issue is to provide and use a static method 'trueCopy' in the AEvent class, that converts an AEvent object with data fields only into a true AEvent instance that has been created with a constructor, and got its data fields replicated with use of the Object.assign helper method.

```
static trueCopy(aEvent:AEvent): AEvent {
    return (aEvent == null ? null : Object.assign(new AEvent(), aEvent));
}
```

C. Launch both the Spring-boot backend and the Angular frontend and verify whether the backend-a-events appear in your new frontend overview5.
It is well possible that you run into a CORS issue (make sure you log the errors that you may get from an http request).



If your backend REST service is provided from a different port (registration) than your frontend UI site (4200), 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

```
@Override
public void addCorsMappings(CorsRegistry registry) {
    registry.addMapping("/**")
            .allowedOrigins("http://localhost:4203", "http://localhost:4200")
```

WebMvcConfigurer interface. In this class you need to implement addCorsMappings
Make sure that the class is found during the Spring Boot component scan and automatically instantiated.

D. Now, your AEventsSbService should initialise properly and your Overview5 component should be able to show an

a-events list that was retrieved from the backend.
if you select an a-event, you should see its details, and the aEvent-id that was generated in the backend.
Implement additional private methods in AEventsSbService:

restPostAEvent(aEvent): Observable<AEvent>
restPutAEvent(aEvent): Observable<AEvent>
restDeleteAEvent(aEventId): void

which use http requests and responses to interact with backend, and maintain a consistent state of the local a-events array within the service.
Call upon these methods from your service methods save(aEvent) and deleteById(aEventId) which are still being used behind the buttons of your UI components.

E. Depending on your implementation, you may hit three challenges:

1. The Add AEvent button in the Overview5Component is expected to create an a-event with some random content and then post that to the backend. Now the backend should assign it its unique id and return the updated and saved a-event in the response. Consequently, the Overview5Component can only select that new a-event after the response has been returned, because it needs the aEvent-Id for tracking the selection.
One way to address that is to let the AEventsSbService.addRandomAEvent() method return the observable of the http-request, and have the Overview5Component.onAddAEvent() method also subscribe to that observable and only select the new a-event after the response has been delivered.

2. In specific cases you may wish to subscribe multiple times to the same observable that is returned from an http request. (E.g. both in the service and in the UI component.) By default, that has a nasty side effect that the request itself will then also be issued multiple times, once for each subscriber. At https://blog.angulartraining.com/how-to-cache-the-result-of-an-http-request-with-angular-f9aebd33ab3 you can read how to prevent that with a .pipe(shareReplay(1)) operator suffix on the request.

3. If you reload the page, it may occur that your Detail5Component initialises before the service has loaded the a-events, and that your edit panel remains empty. Of course, the user can work around by reselecting the a-event in the overview, but that is not 'user friendly'. You can resolve the issue with some retry code in Detail5Component using setTimeout, but that is not elegant either. Here shows the disadvantage of the cacheing service approach. Yet, if you read all articles about proper use of rxjs in Angular, then also the adaptor approach seems to incur a lot of complexity…

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

4 Second term assignments: JPA and Authentication

In these assignments you will expand the backend part of the 'Amstedam Events' 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 in order to ensure data integrity across multiple updates.

By the end of this assignment you will have implemented the AEvent and Registration classes including its one-to-many relationship. Your REST API can add registrations to a-events. It will produce error responses on registrations for a-events that are not PUBLISHED and on registrations that would exceed the maximum number of participants of an a-event.

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 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

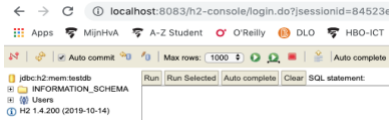
A. First update your pom.xml to include the additional dependencies for 'spring-boot-starter-data-jpa' and 'h2' similar to the demonstration of a project setup in O'Reilly-3.Ch3.
(Do not include the JDBC dependency).
Also enable the H2 console in application.properties, and make sure the logging level is at least 'info' so that Spring shows its configuration parameters when it starts.
Enable

spring.jpa.show-sql=true
logging.level.org.hibernate.type=trace

in application.properties such that you can trace the SQL queries being fired.

Relaunch the server app, and use the h2-console to check-out that H2 is running.

```
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'
```



(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)

B. Upgrade your AEvent 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 AEventsRepositoryJpa of your AEventsRepository 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 AEventsRepositoryJpa.

If you now run the backend you might get a NonUniqueBeanDefinitionException, because you may have two implementation classes of the AEventRepository interface: AEventsRepositoryMock and AEventsRepositoryJpa.
(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 a-event with postman doing a post at localhost:8080/aevents. Verify the associated SQL statements in the Spring Boot log and use the h2-console to verify whether the a-event ended up in H2.

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

```
Hibernate: call next value for aevent_ids
Hibernate: insert into aevent (description, end, is_ticketed, max_participants,
 participation_fee, start, status, title, id) values (?, ?, ?, ?, ?, ?, ?, ?, ?)
```

SELECT * FROM AEVENT;

ID	DESCRIPTION	END	IS_TICKETED	MAX_PARTICIPANTS	PARTICIPATION_FEE	START	STATUS	TITLE
30001	*null*	2021-06-18 05:00:00	TRUE	500	5.0	2021-06-17 04:00:00	PUBLISHED	Backend event 'a.27'

C. Also implement and test the other three methods of your AEventsRepository 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 AEventsRepositoryJpa.findAll()

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

Test your new repository with postman.

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

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

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

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.

```
private void createInitialAEvents() {
    // check whether the repo is empty
    List<AEvent> aEvents = this.aeventsRepo.findAll();
    if (aEvents.size() > 0 ) return;
    System.out.println("Configuring some initial aEvent data");

    for (int i = 0; i < 9; i++) {
        // create and add a new aEvent with random data
        AEvent aEvent = AEvent.createRandomAEvent();
        aEvent = this.aeventsRepo.save(aEvent);

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

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

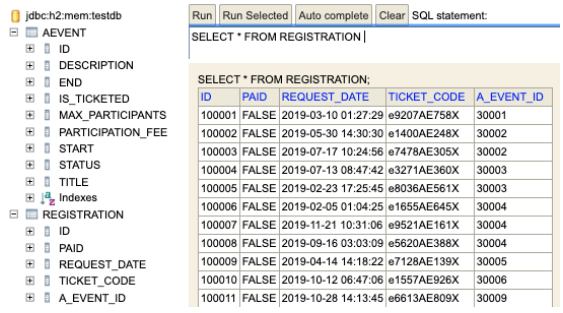
A. Now is the time to introduce a second entity. Make a new model class ‘Registration’ identified by an attribute named ‘id’ (long). Also record other relevant information about registrations:
   ticketCode (String)
   paid (boolean)
   submissionDate (LocalDateTime)

   Each Registration is associated with one AEvent.
   An AEvent is associated with many Registrations.
   Declare the corresponding association attributes in both classes and make sure they are initialised in their constructors.
   Also provide the JPA @ManyToOne and @OneToMany attributes as explained in RangaJPA.Ch8

B. Implement a RegistrationsRepositoryJpa similar to your AEventsRepositoryJpa. 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 needs extension…!!! That 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 AEventsRepository code….

C. Extend the initialisation in the command-line-runner of task 4.1.1-D to add a few registrations to every aEvent and save them in 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’) a-events only include ‘attached’ registrations.



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

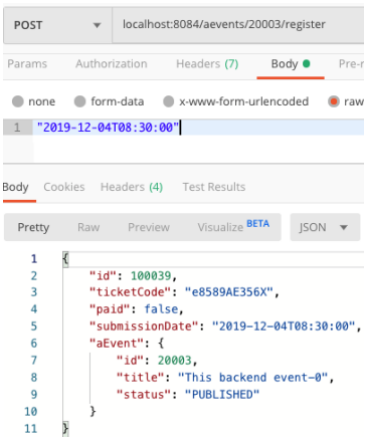
D. Re-test the REST API at localhost:8080/aevents with postman:
You may find a response like here with endless recursion in the JSON structure. For now, investigate the use of @JsonBackReference to fix that.
With (optional) bonus assignment 4.2.2 you can practice a better solution with custom Json serializers.

E. Implement a POST mapping at the /aevents/{aeventId}/register REST endpoint. This mapping should add a new registration to the a-event. An error response should be provided if
1) the a-event does not have status 'PUBLISHED' or
2) the maximum number of participants have been registered already.
In other cases the registration should be created and added to the aEvent, and a creation success status code should be returned.

Provide some additional functional methods in AEvent to support creation of new registrations, e.g.:

```
/**
 * @return number of registrations for this AEvent
 */
public int getNumberOfRegistrations() {
```

```
/**
 * Creates and adds new registration for this AEvent
 * Checks whether this.status == PUBLISHED and
 * maxParticipants has not been exceeded
 * @param submissionDateTime
 * @return the created registration or null
 */
public Registration createNewRegistration(LocalDateTime submissionDateTime) {
    // TODO check conditions and create and add registration
```



Test your new end-point with postman:

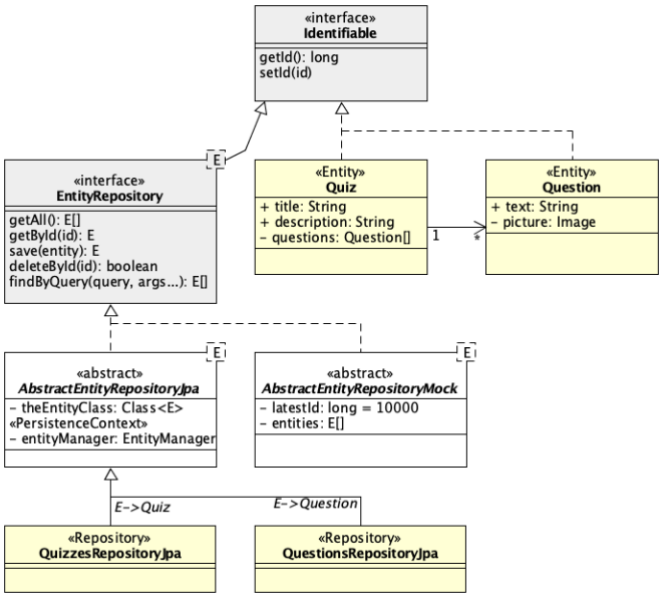
4.1.3 [BONUS] Generalized Repository

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 AEvent entity and the Question entity by your Registration 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 /aevents REST endpoint to optionally accept a request parameter '?title=XXX' or '?status=XXX' or '?minRegistrations=999', and then filter the list of a-events 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 a-events that actually 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, @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
```

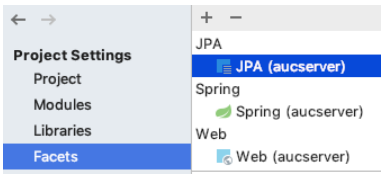
(Here we assume you use the generic repository interface)

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:
"AEvent_find_by_status": finds all a-events of a given status
"AEvent_find_by_title": finds all a-events that have a given sub-string in their title
"AEvent_find_by_minRegistrations": finds all a-events that have at least the given number of registrations.

Use an ordinal(positional) query parameter as a place-holder for the parameter value to be provided.
Use the @NamedQuery annotation to specify these named JPQL queries within your AEvent.java entity class.

(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)

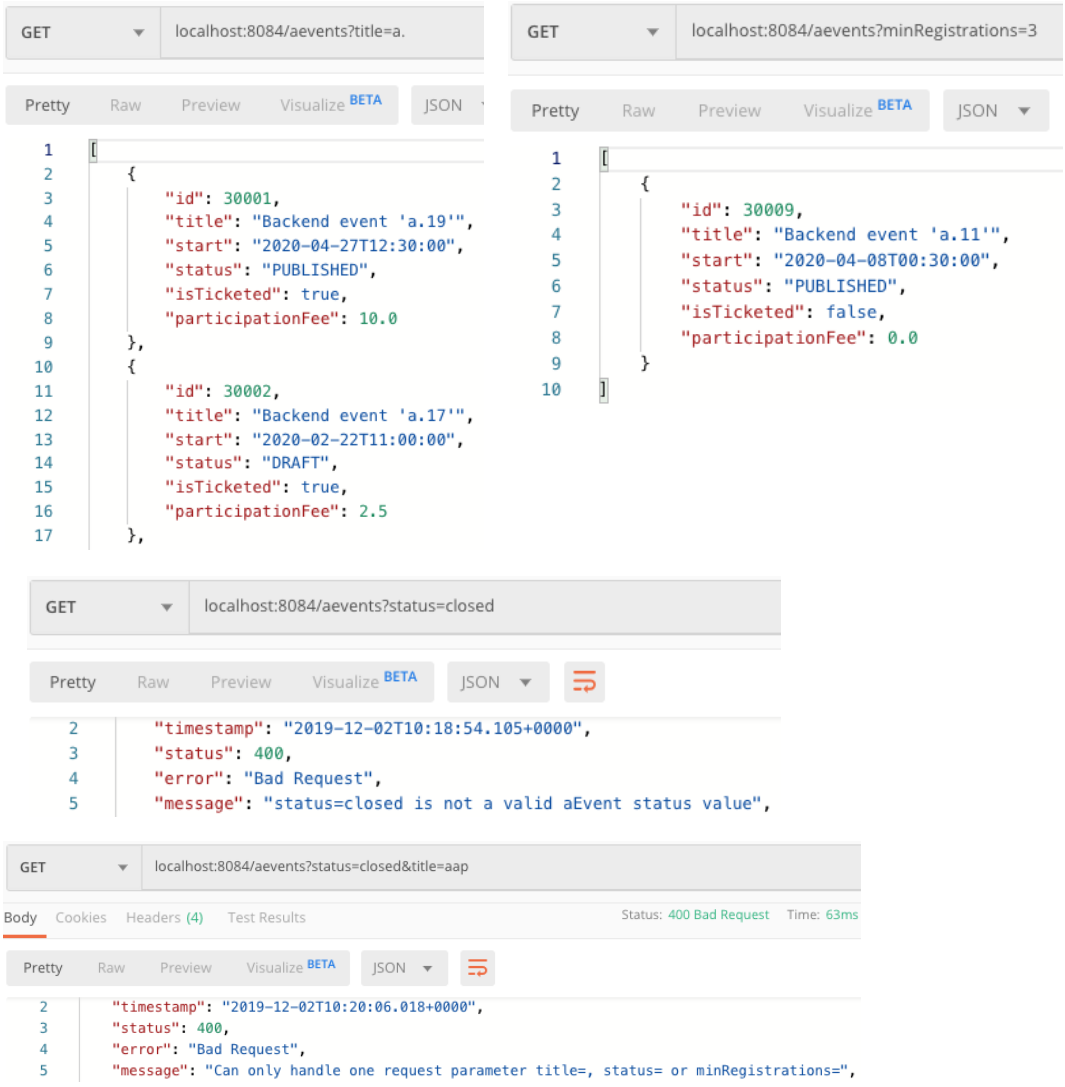


C. Extend your GetMapping on the “/aevents” REST end-point to optionally accept a ‘?title=XXX’ or ‘?status=XXX’ or ‘?minRegistrations=999’ request parameter.

If no request parameter is provided, the existing functionality of returning all a-events should be retained.
If more than one request parameter is specified, an error response should be returned, indicating that at most one parameter can be provided.
If the ‘?status=XXX’ parameter is provided, with a status string value that does not match the AEventStatus enumeration, an appropriate error response should be returned.

In the other cases the requested a-events 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 a-event.

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 @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 a-event, you want full information about the a-event but probably only shallow information about its registrations.
b) If you query a registration, you want full information about the registration, but only shallow information about the a-event.
c) It gets even more complicated with recursive relations.

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

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

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

```
public class CustomJson {

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

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

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

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

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

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

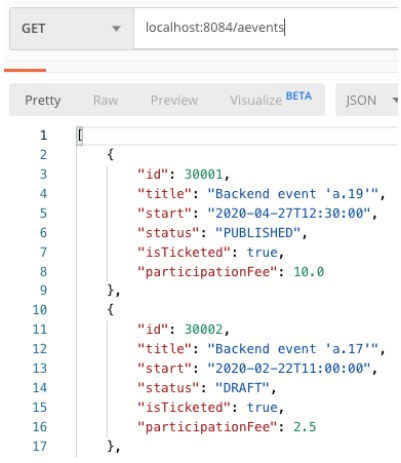
```
@ManyToOne()
@JsonView({CustomJson.Summary.class})
@JsonSerialize(using = CustomJson.ShallowSerializer.class)
private AEvent aEvent;
```

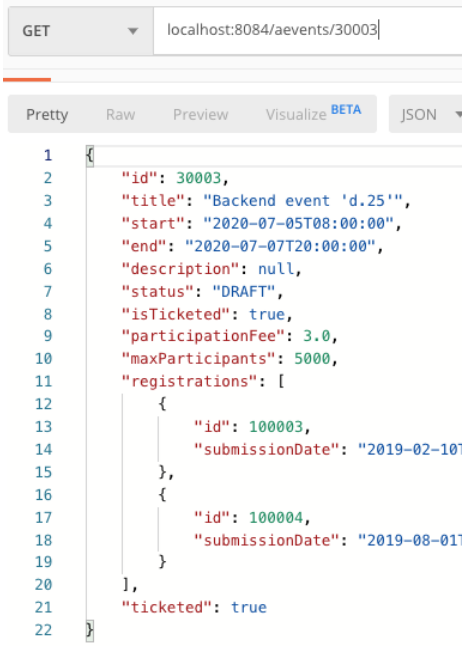

The consequence is:
1) the aEvent reference will only get serialized for unrestricted mappers and mappers that specify the CustomJson.Summary view.
2) when the a-event is serialized (as part of an registration serialization) its serialization will be shallow (and not recurse back into all its registrations…)

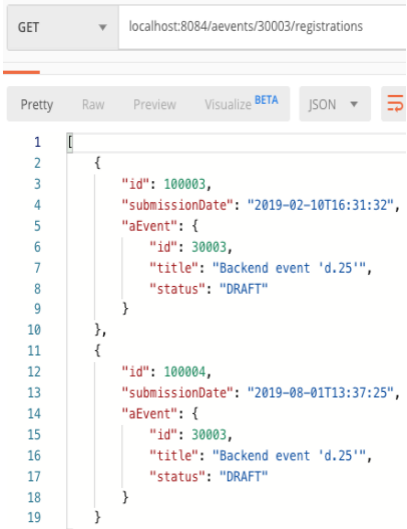
Extend this CustomJson class with a similar implementation of the custom SummarySerializer and the UnrestrictedSerializer internal classes which serialize to Summary view and default view respectively.

B. Apply these view classes and serializers to relevant attributes in the AEvent and Registration model classes.
Apply the Summary view class to the localhost:8080/aevents and localhost:8080/aevents/{aEventId}/registrations end-points.
Implement unrestricted end-points at localhost:8080/aevents/{aEventId} and localhost:8080/aevents/{aEventId}/registrations/{registrationId}.

Test your end-points with postman:







4.3 Backend security configuration, JSON Web Tokens (JWT)

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

The Spring framework includes an extensive security module. However, that module is rather difficult to understand and use 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 '/authenticate' which provides for user registration and user login.
This end-point 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.

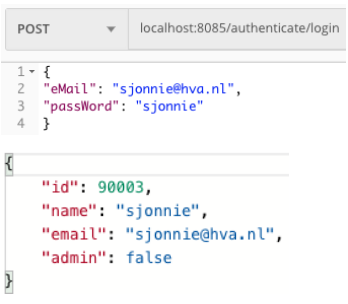
By the end of this assignment you will have further explored annotations @RequestBody, @RequestAttribute, @Value and classes ObjectNode, Jwts, Jws<Claims>, SignatureAlgorithm

4.3.1 The /authenticate controller.

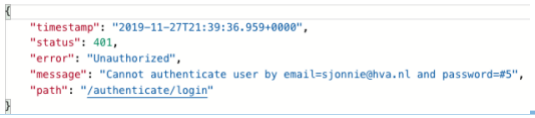
A. First create a new REST controller class 'AuthenticateController' in the 'rest' package.
Map the controller onto the '/authenticate' endpoint.
Provide a POST mapping at '/authenticate/login' which takes two parameters from the request body: eMail(String) and passWord(String)

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 4.6
For now we accept successful login if the provided password is the same as the user name before the @ character in the email address.
Throw a new 'UnAuthorizedException' if login fails.
Return a new User object with 'Accepted' status after successful login.
(Create a new entity 'User' in your models package. A User should have attributes 'id'(long), 'name'(String), 'eMail'(String), 'hashedPassWord'(String) and 'admin'(boolean). 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.

```
<!-- JWT jackson -->
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-api</artifactId>
    <version>0.10.7</version>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-impl</artifactId>
    <version>0.10.7</version>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-jackson</artifactId>
    <version>0.10.7</version>
    <scope>runtime</scope>
    <exclusions>
        <exclusion>
            <groupId>com.fasterxml.jackson.core</groupId>
            <artifactId>jackson-databind</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<!-- end JWT jackson -->
```

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 identification and his (admin) authorization.

```
public class JWToken {

    private static final String JWT_USERNAME_CLAIM = "sub";
    private static final String JWT_USERID_CLAIM = "id";
    private static final String JWT_ADMIN_CLAIM = "admin";

    private String userName = null;
    private Long userId = null;
    private boolean admin = false;

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

        String token = Jwts.builder()
                .claim(JWT_USERNAME_CLAIM, this.userName)
                .claim(JWT_USERID_CLAIM, this.userId)
                .claim(JWT_ADMIN_CLAIM, this.admin)
                .setIssuer(issuer)
                .setIssuedAt(new Date())
                .setExpiration(new Date(System.currentTimeMillis() + expiration * 1000))
                .signWith(key, SignatureAlgorithm.HS512)
                .compact();

        return token;
    }

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

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:

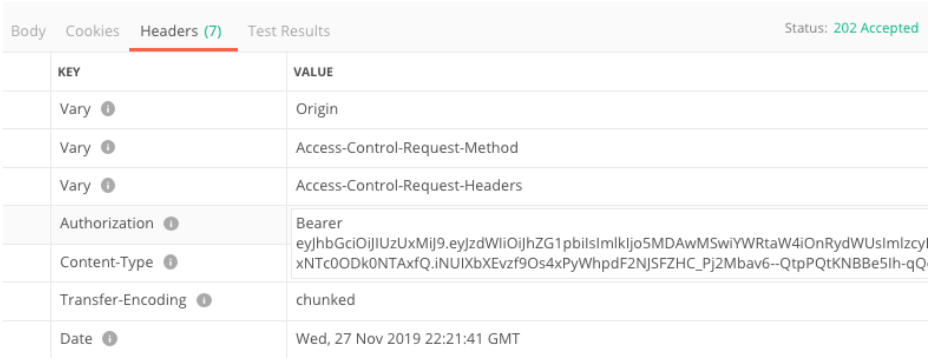
```
// JWT configuration that can be adjusted from application.properties
@Value("${jwt.issuer:private company}")
private String issuer;

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

@Value("${jwt.duration-of-validity: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

    // path prefixes that will be protected by the authentication filter
    private static final Set<String> SECURED_PATHS =
            Set.of("/aevents", "/registrations", "/users");
```

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, which does all the filter work:

```
@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()) ||
            SECURED_PATHS.stream().noneMatch(servletPath::startsWith)) {

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


It is important to let ‘pre-flight’ OPTIONS requests pass through without burden. Angular will 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 ‘/authenticate’, ‘/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 throw an UnAuthorizedException which will abort further processing of the request:

```
JWToken jwToken = null;

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

// block the request if no token was found
if (encryptedToken != null) {
    // remove the "Bearer " token prefix, if used
    encryptedToken = encryptedToken.replace("Bearer ", "");

    // decode the token
    jwToken = JWToken.decode(encryptedToken, this.passPhrase);
}

// Validate the token
if (jwToken == null) {
    throw new UnAuthorizedException("You need to logon first.");
}
```

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

```
public static JWToken decode(String token, String passPhrase) {
    try {
        // Validate the token
        Key key = getKey(passPhrase);
        Jws<Claims> jws = Jwts.parser().setSigningKey(key).parseClaimsJws(token);
        Claims claims = jws.getBody();

        JWToken jwToken = new JWToken(
                claims.get(JWT_USERNAME_CLAIM).toString(),
                Long.valueOf(claims.get(JWT_USERID_CLAIM).toString()),
                (boolean) claims.get(JWT_ADMIN_CLAIM)
        );

        return jwToken;

    } catch (ExpiredJwtException | MalformedJwtException |
            UnsupportedJwtException | IllegalArgumentException e) {
        return null;
    }
}
```

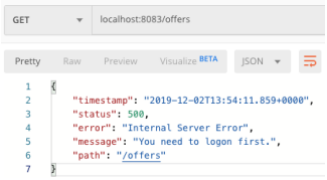
This decode method uses the same JWToken attributes and getKey method that were also shown earlier along with the encode method.

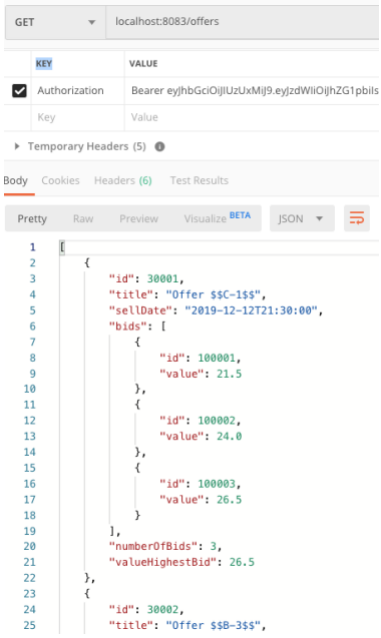
If all has gone well, we add the decoded token information into an attribute of the request and allow the request to progress further down the chain:

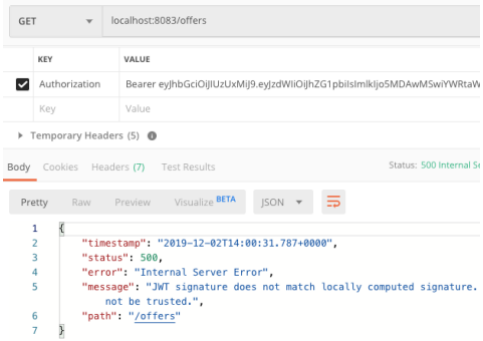
```
// pass-on the token info as an attribute for the request
request.setAttribute(JWToken.JWT_ATTRIBUTE_NAME, jwToken);

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 Angular 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("/**")
                .allowCredentials(true)
                .allowedHeaders(HttpHeaders.AUTHORIZATION, HttpHeaders.CONTENT_TYPE)
                .exposedHeaders(HttpHeaders.AUTHORIZATION, HttpHeaders.CONTENT_TYPE)
                .allowedMethods("GET", "POST", "PUT", "DELETE")
                .allowedOrigins("http://localhost:4805");
        }
}
```


4.4 Frontend authentication, and Session Management

With a backend that is capable of authenticating users and providing tokens for smooth and authorised access of secured REST end-points you now can integrate this security into your Angular user interface.
See also http://jasonwatmore.com/post/2018/10/29/angular-7-user-registration-and-login-example-tutorial for a mini-tutorial of a secured project.

4.4.1 Sign-on and session management.

A. Create a new SessionSbService that will provide an adapter (interface) to the backend for logon 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.

   In the subsequent steps you will implement the following methods in this service:
   signOn(eMail:string, password:string)
   logs on to the backend, and retrieves user details and the JWT authentication token from the backend.
   signOff()
   discards user details and the JWT authentication token from the session (and local browser storage).
   isAuthenticated(): boolean
   indicates whether a user has been logged on into the session or not.
   getTokenFromSessionStorage(): string
   retrieves the JWT authentication token and user details from the session (or local browser storage).
   saveTokenIntoSessionStorage(token: string, username: string)
   saves the JWT authentication token and user details into the session (and local browser storage).

   First implement the signOn method as indicated here. The http.post in the signon uses a special option {observe: 'response'} which gives you access to the complete response of the request, including the headers. Without this option, the observable would only expose the body of the response. You need access to the headers to be able to extract the authentication token from the Authorization header.
   As a consequence you also need to do a bit more work to extract the user name from the response body.

   At this stage, there is no need to decode the JWT token. Just save it in

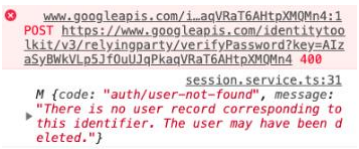
```
@Injectable({
  providedIn: 'root'
})
export class SessionSbService {
    public readonly BACKEND_AUTH_URL = "http://localhost:8084/authenticate";

    public currentUserName: string = null;

    constructor(private http: HttpClient) {
        this.getTokenFromSessionStorage();
    }

    signIn(email: string, password: string): Observable<any> {
        console.log("login " + email + "/" + password);
        let signInResponse =
            this.http.post<HttpResponse<User>>(this.BACKEND_AUTH_URL + "/login",
                {eMail: email, passWord: password},
                {observe: "response"}).pipe(shareReplay(1));
        signInResponse
            .subscribe(
                response => {
                    console.log(response);
                    this.saveTokenIntoSessionStorage(
                        response.headers.get('Authorization'),
                        (response.body as unknown as User).name
                    );
                },
                error => {
                    console.log(error);
                    this.saveTokenIntoSessionStorage(null, null);
                }
            )
        return signInResponse;
    }
```

a private instance variable of the service for use in future requests.

Also implement the signOff() method in this service.

B. Create a new component header-sb in src/app/components/mainpage which can be a copy of the original header component initially. inject the SessionService into your header-sb component
Welcome the currently logged-in user in the sub-title in this header at the right. If no user is logged-in, a 'Visitor' shall be welcomed.
Use this header in your app-component and retest your application.

C. Add a new component 'sign-on' to src/app/components/mainpage and connect it to a new 'login' route, which should be invoked from the 'Log in' menu item on the navigation bar.
Provide a form in which the user can enter e-mail and password and a Log in button to submit the request.



Use the signOn method of the service which should actually login the user and have the username displayed in the header. For now its is sufficient that any authentication errors are logged at the browser console.

D. Create a new component nav-bar-sb in src/app/components/mainpage, which can be a copy of nav-bar initially. Use this new navigation bar in your app-component.
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.
Connect the click-event of the 'Log out' option to the signOff() method in the session service.

4.4.2 HTTP requests with authentication tokens and use of browser storage.

Even if the user has been logged on, http requests towards the secured end-points of the backend will not be accepted yet, because sofar we did not add the JWT authentication token yet to any request. In this assignment you will configure the frontend to automatically add the JWT token to every outgoing HTTP request to inform the backend about the authentication and authorisation status of the user. For that you provide the HTTP_INTERCEPTORS service.

A. Create a new class auth-sb-interceptor in src/app/services which implements the HttpInterceptor interface from '@angular/common/http'.
Inject the SessionSbService into this interceptor such that it can use the token of the current user.
The 'intercept' method of the interceptor shall add the token into the Authorization header of the request.

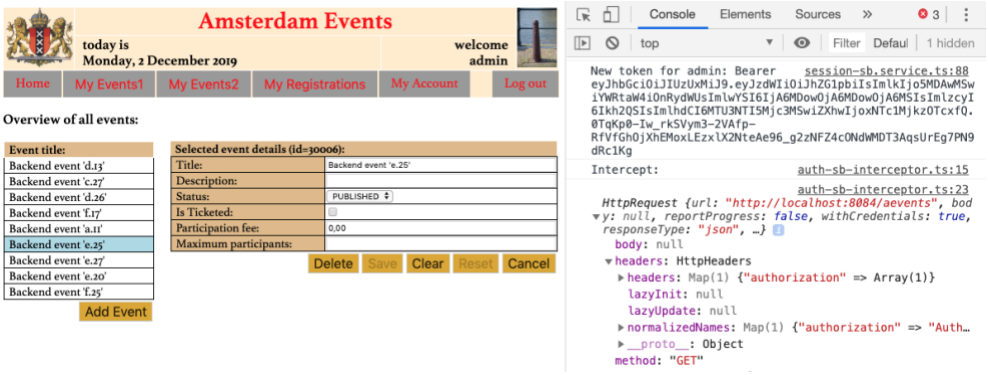
```
@Injectable()
export class AuthSbInterceptor implements HttpInterceptor {

    constructor(private session: SessionSbService) { }

    intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

        let token = this.session.getTokenFromSessionStorage();

        if (token != null) {
            // TODO clone the request, adding the token in the Authorization header
            // pass on the cloned request to the next handler

        } else {
            // just forward the request to the next handler
            return next.handle(req);
        }
    }
```

Requests are immutable, so you cannot change them. You add the token by cloning the request and then adding the token as part of that operation. Then you forward the cloned request. The code snippet provided here should get you going…

Make sure you provide the interceptor from app.module.ts:

```
{ provide: HTTP_INTERCEPTORS,
    useClass: AuthSbInterceptor, multi: true },
```

Re-test your application and verify that visitors cannot change data, while authenticated users can. Incluse console.log output in your interceptor and verify that the token is passed along with the request.



B. It would be nice if your token and username is preserved in the frontend also if you reload your page. That can be achieved by using browser sessionStorage and/or localStorage.
Below code snippet picks up your token from the sessionStorage and if that's gone, tries to pick up a token from localStorage.

```
private readonly BS_TOKEN_NAME = "AE_SB_AUTH_TOKEN";

// allow for different user sessions from the same computer
getTokenFromSessionStorage(): string {
    let token = sessionStorage.getItem(this.BS_TOKEN_NAME);
    if (token == null) {
        token = localStorage.getItem(this.BS_TOKEN_NAME);
        sessionStorage.setItem(this.BS_TOKEN_NAME, token);
    }
```

sessionStorage stores key value pairs for a single tab in the browser. If you reload the page, the values are preserved. If you close the tab or the browser, those values are gone.
localStorage stores key value pairs that are shared across all tabs. Even if you restart the browser, the values are preserved. (However, different brands of

browsers chrome, firefox, edge, etc. do not share their stores).

Also implement the saveTokenIntoSessionStorage() to save tokens in sessionStorage and/or localStorage and integrate that in your authentication handling. Also, find a way include the username into the session- and localStorage items,
Demonstrate that you can run two parallel sessions in different tabs of the same browser, which each are logged on with a different account.

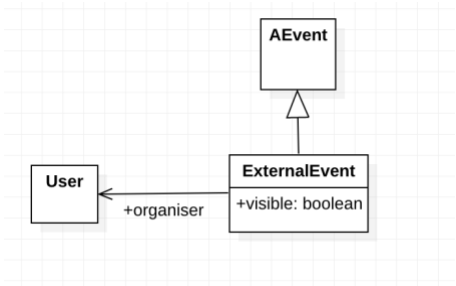
C. Having your tokens in localStorage opens your application to the 'Cross-Site Request Forgery (CSRF vulnerability 8A in OWASP-2013). Keeping tokens in sessionStorage only prevents that vulnerability if you also ensure that your application never opens an external page into its tab. For that you need a robust 'leaving the site'-guard in your Angular frontend. (See bonus assignment 3.4.3.) And first of all, you also need to implement use of the https protocol.
All that is beyond the scope of this assignment.

4.5 Expand the entity relationship model

This exercise is about object-relational mapping for inheritance. Here, you should refactor your code to attend to the requirements and at the same time produce minimum side-effects to the existing functionalities.

The Amsterdam Events business is a success in the city of Amsterdam. After a thorough investigation, the company decided to expand its business, allowing regular citizens, not only the city hall, organising events, contributing to the society well-being.

You should create a new entity called ExternalEvent, which inherits from the entity AEvent. There are different business rules that should be added to this new entity, but this is outside of the scope. From now, we are implementing only a back-end feature as can be seen in the class diagram below: a) The organiser can specify if the event is visible to the public or not. Thus, an attribute called “visible” should be added to the ExternalEvent entity. b) An external event should have an organiser. Make sure to specify the annotations, informing that this class is an entity. Also, you should change the parent class, telling the most suitable strategy. It is up to you to decide the most appropriate strategy for this scenario. Be prepared to explain your decision during the assessment of this exercise.



Add REST endpoints to add or modify external events. They should follow the following respective formats:

POST /users/:user-id/external-events (add an external event for the specified user)
PUT /external-events/:external-event-id (change an external event)

Note that the remaining endpoints should work without any problem for entities (e.g., getting and deleting a child or parent entities should work seamlessly)

To avoid the risks of having your exercise marked as insufficient, make sure that all other functionalities of your application are working with success.

4.6 [BONUS] Server notifications using WebSockets

This assignment is under revision – please wait for the updated document.

4.7 [BONUS] Full User Account Management

A.
Implement the User Repository in the backend. Also store the hashed passwords.
Implement the ‘authenticate/register’ endpoint to register a new user account.
Add an active (Boolean) attribute to the User entity; newly registered accounts are in-active initially.

B. Generate a unique activation code for each newly registered user account, and email a link that refers to /authenticate/activate/{activationCode} to the user that has registered the account. Record an expiration dateTime for the activation.
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.

C. Implement a ‘/users’ end-point where users can retrieve and update 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 admin attributes.
Use the userId in the Authorization token to identify the requesting user.