Hogeschool van Amsterdam

2000WEF 19
2000DWEF19
Web Frameworks (vt, dt)

Assignment
Amsterdam Events

Version	: 19.25
Date	: 1 December 2019
Study tracks	: HBO-ICT, Software Engineering
Curriculum	: year 2, terms 1 & 2
Study guide	: vt: https://studiegids.hva.nl/#/hbo-ict-se-vt/100000026/011501 dt: https://studiegids.hva.nl/#/hbo-ict-se-dt/100000040/011502
Course materials	: vt: https://dlo.mijnhva.nl/d2l/home/41386 dt: https://dlo.mijnhva.nl/d2l/home/41362

Versions

Version	Date	Author	Description
19.1	24 Aug 2019	John Somers	Initial version
19.12	1-8 Sep 2019	John Somers	Small corrections
19.25	24 Nov 2019	John Somers	Back-end assignment up to 4.5

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.

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.

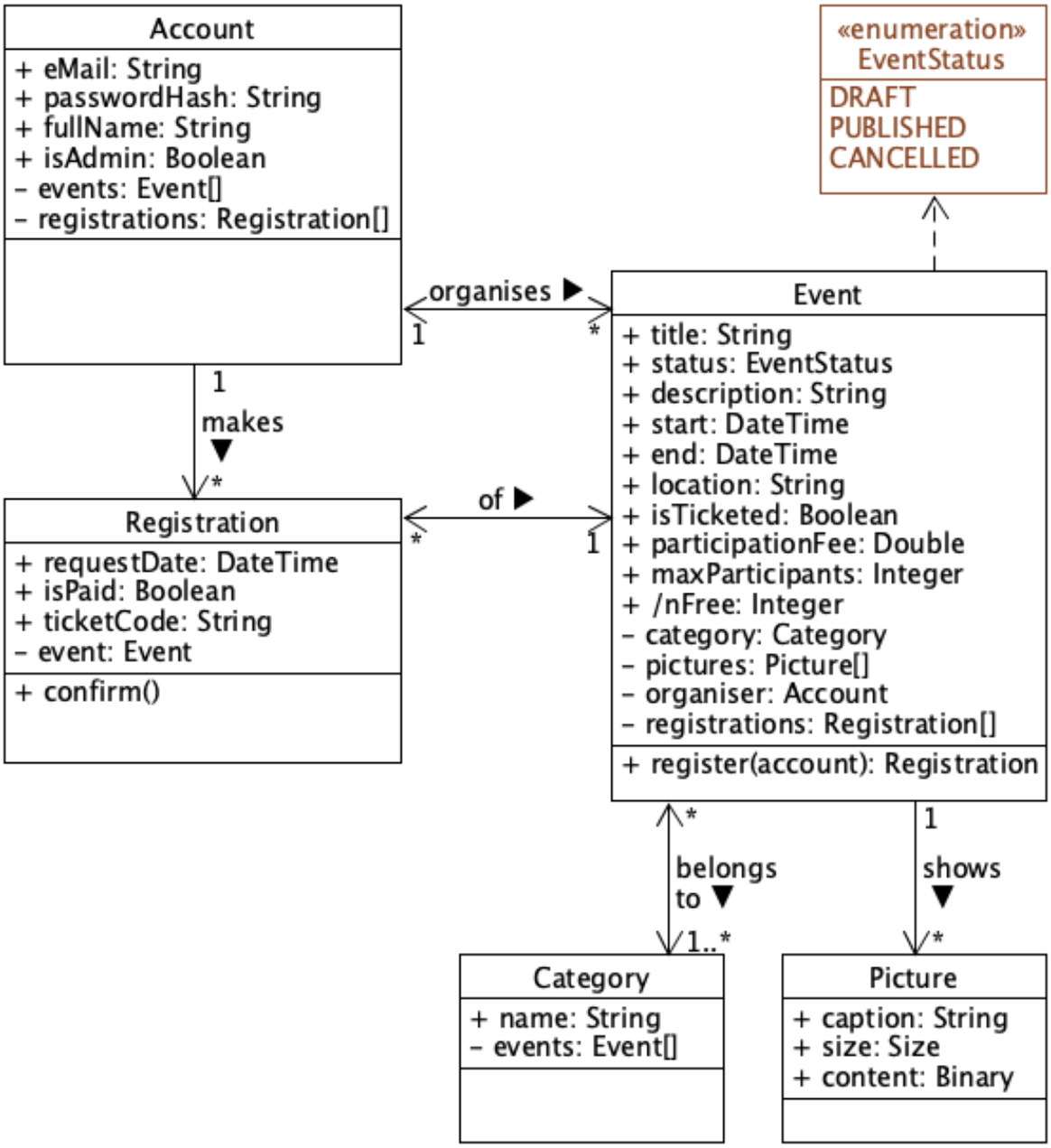


Figure 1: Navigable Class Diagram 'Amsterdam Events'

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 front-end with the back-end 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.

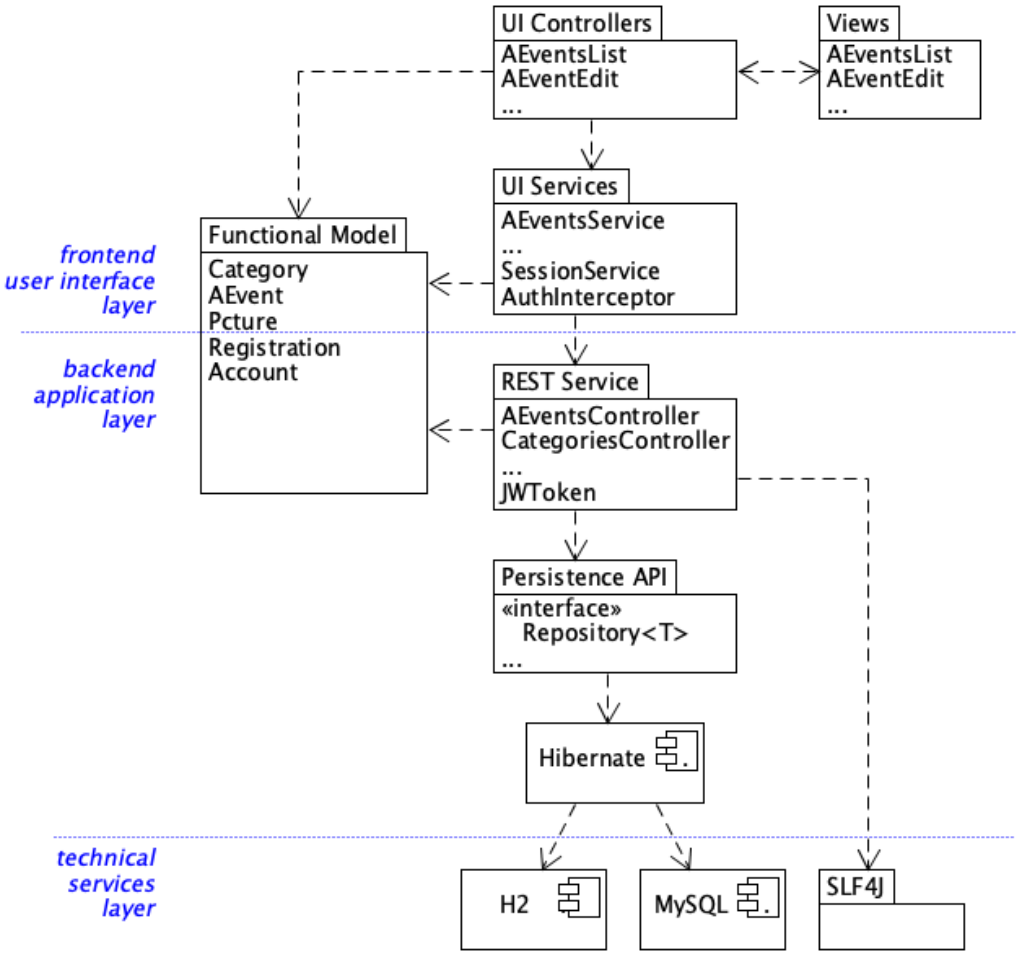
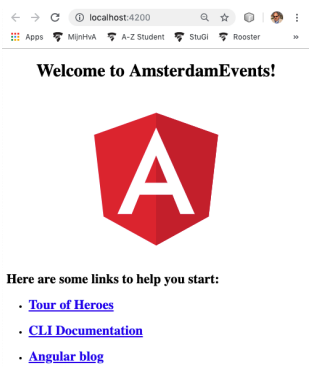


Figure 2: Full-stack Logical Architecture

3 Assignments frontend frameworks: Angular 8

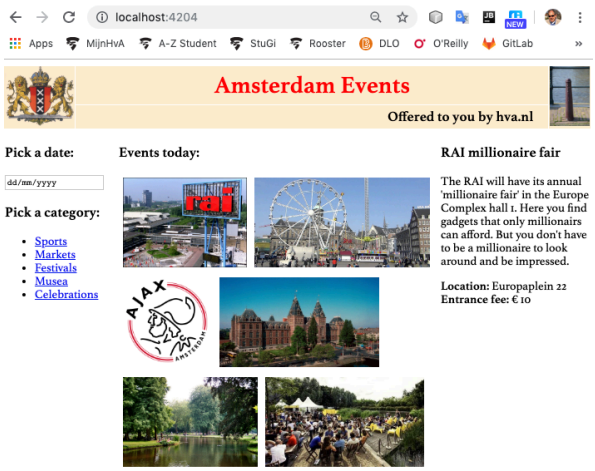
3.1 Amsterdam Events Home Page

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'. (New Project → Angular CLI.)
Test your project by running 'start' from the package.json file.
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).
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.)

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”.

C. Setup the app/components/events source directory and define in there an ‘overview1’ component.
Create a local list of nine events with random data in the component controller. (Use future event dates and a random mix of ticketed events with varying participation fees and maximum number of participants.)

```
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.
Replace the ‘home’ view in the app-component main view by your events list view.

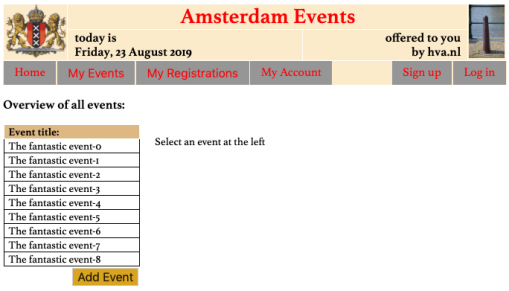
E. Add a button at the bottom right of the view with text: ‘Add Event’
use ‘event-binding’ on the button that binds its ‘click’-event to the component function

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. Use CSS to highlight the AEvent that has been selected in the list.
Overview2 embeds the detail2 component in a right panel, 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)



B. The detail2 component manages the edited AEvent. 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).
Add two buttons 'Save' and 'Delete' to detail2 which the user can click to confirm changes or to remove the AEvent altogether.

C. Use property binding and event binding within the <app-detail2> element of the overview2 template to link the selected AEvent in overview2 with the edited AEvent in detail2 and to pass Save and Delete actions in detail2 back to the overview2 component, such that those updates can be applied in the aEvents list managed by overview2.
(Provide EventEmitters from the detail2 component and apply appropriate @Input and @Output decorators in detail2.)
If the user changes selection at the left while changes have been made in the details form without clicking the Save button, those changes will be lost and not applied to the source list managed by overview2.

D. When a new AEvent is added via the 'Add Event' button at the left, that should automatically get the selection focus (and be associated with the detail2 component)

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.

A. Setup the src/app/services source directory. Create in there an a-events service, that will manage the collection of available AEvents. ($ 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 basic CRUD operations in your service to add, update or remove an AEvent from the managed collection. (For now, we use the list index as identification of an individual AEvent.)

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

    public aEvents: AEvent[];

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

    add(aEvent: AEvent): number {
        // TODO append the event at the end of the list
        // and return its index
    }

    update(eIdx: number, aEvent: AEvent) {
        // TODO replace the identified event with the provided
    }

    remove(eIdx: number): AEvent {
        // TODO remove the identified event from the collection
        // and return the removed instance
    }
```

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 aEvents 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.
Detail3 retrieves the aEvent to be edited from aEventsService, and also invokes update or remove operations at aEventsService after the user has clicked such button.

C. Maintain a selectedAEventIndex 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:

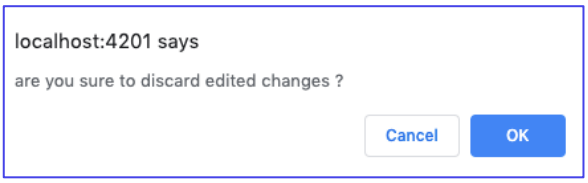
```
<app-detail3 [(editedAEventId)]="selectedAEventIndex"></app-detail3>
```

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 then overview3 should follow by also un-selecting that item.
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)
For visual attention, display the value of editedAEventId in the header of the detail3 form.
Show the new overview3 component from your app-component. Thoroughly test correct behaviour of your solution.

D. Implement additional buttons 'Clear', 'Reset' and 'Cancel' in detail3 with the following functionality:
'Clear' will clear all fields in the form as in a new AEvent.
'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. Also, when another AEvent is being selected in the title list while there are unsaved changes this confirmation box shall be presented.



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).

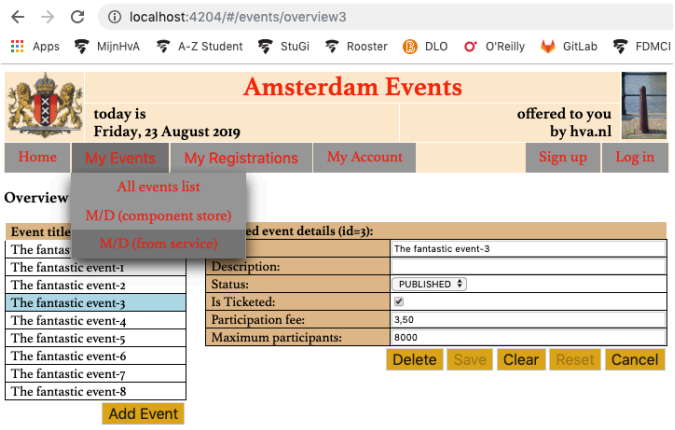
F. Implement '[disabled]' property binding on the 'Save', 'Reset' and 'Delete' buttons, such that 'Save' and 'Reset' are disabled if nothing has been changed yet, and 'Delete' is disabled if there are unsaved changes.
(The other buttons are always shown.)

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

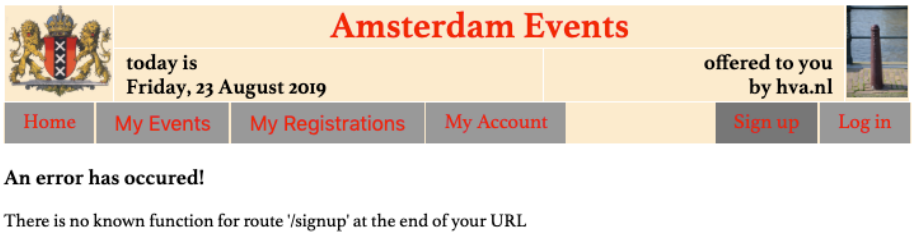
First you will configure the router module and connect the four components that you have built in the earlier assignments to your navigation bar.



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 3.1.
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.

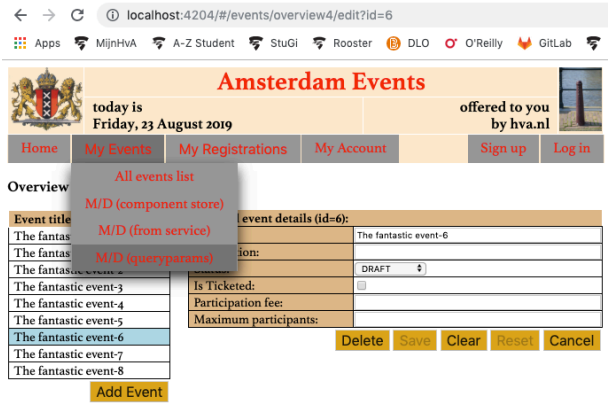
B. Connect the ‘Sign Up’ menu item to the ‘signup’ route and the ‘Log in’ menu item to the ‘login’ route, without providing these routes in the routes table.
Add a new ‘error’ component in src/app/components/mainpage with a simple error message, which indicates that a specified route is not available,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



3.4.2 Parent / Child communication with sub-routes and query parameters

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

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



B. The two-way custom property binding between the Overview4Component.selectedAEventIndex and Detail4Component.editedAEventId can be removed. Instead, you create a child sub-route ‘edit’ for the detail4 component under the events/overview4 parent route and also create the child <router-outlet> in the overview4.component.html.
If the user now selects a different event in the list of titles at row position=*index*, you call the navigate method on the router towards the sub-route ‘edit?id=*index*’.
(For that you need to inject the router and the activatedRoute in the constructor of overview4).

```
// activate the details page for event Id = index
this.router.navigate(['edit'], {
        relativeTo: this.activatedRoute,
        queryParams: {id: index}
    }
);
```

(Strictly it is not required to use a sub-route ‘edit’ here, but this approach gives better options of using guards later in assignment F.)

C. The detail4 component needs to extract the editedAEventId from the query parameter and retrieve (a copy of) the associated AEvent to be edited from the service. The value of the query parameter is available from the activatedRoute. The value of the query 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 itself each time when the query parameter is touched. That we achieve by subscribing to the queryParam observable in the activatedRoute.
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 every change in the query parameters. In the method getAEventToBeEdited() you shall then obtain the associated AEvent object from the aEventsService for editing.
(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.queryParams[‘id’]. It appeared that the observable also provides an event immediately after initialisation of the component that is identical to value in the snapshot. That you may want to test this to be sure…)

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

private queryParamsSubscription: Subscription = null;

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

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

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 event title in the list will not always follow the URL or the details section. For that you also need to observe the 'id' query parameter in the overview4 component itself, and synchronise it with the selectedAEventIndex.

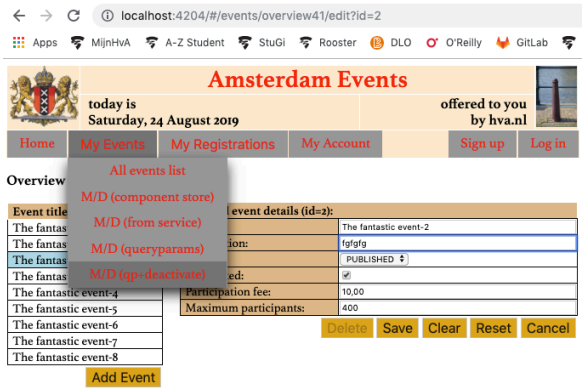
E. Fix the 'Save', 'Delete' and 'Cancel' operations in detail4 to navigate to the parent of the activatedRoute without a query parameter. Before, in the detail3 component you would have emitted an event to notify the parent about un-selection of the AEvent Id.

F. Repair the unsaved changes detection as explained in assignment 0.E, if needed. Also verify that the '[disabled]' properties on the buttons still work fine as explained in 0.F (or repair them).

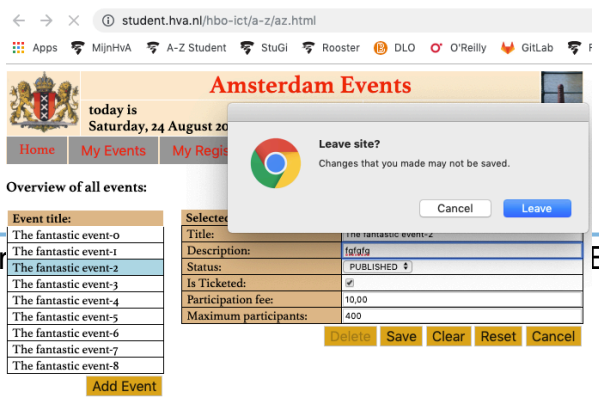
3.4.3 [BONUS] Unsaved changes protection with router guard 'canDeactivate'.

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/events which starts as a copy of detail4 of assignment 3.4.2 initially. There will be no change in the interaction with overview4, so you can reuse that component: create a new route 'events/overview41' that opens overview4 and add a child route 'edit' which opens the new component detail41. Provide the new route 'events/overview41' in the 'events' sub-menu.



B. Implement the CanDeactivateGuardService in src/app/services and the canDeactivate method in the detail4, leveraging your method to check dirtiness. The Angular router only traps changes in the path of the activatedRoute, not in the values of the query parameters. So, you still have to keep the code that traps the changes in the 'id' query parameter while the route stays at /events/overview4/edit.
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.



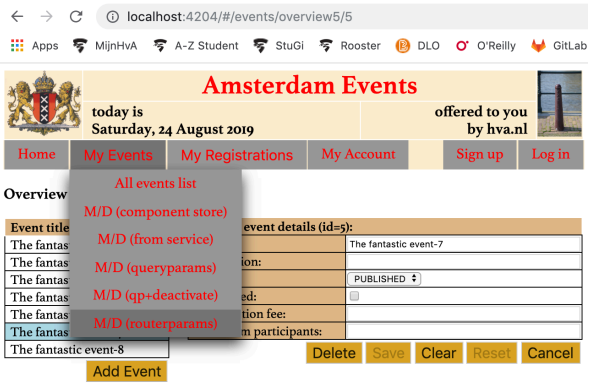
C. Unfortunately, canDeactivate only traps navigation within your application, not if the user navigates to a completely different

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 routing with router parameters.

There may be situations in which query parameters should be avoided. Angular provides a mechanism of router parameters that integrate seamlessly in the router path. Its use on the Master / Detail structure seems a bit more complicated though:

A. Create two new components overview5 and detail5 in src/app/components/events. These can be copies of overview4 and detail4 initially. Connect overview5 to a new route 'events/overview5' and also add the route to the My Events sub-menu. Provide a child route ':id' to the parent route 'events/overview5'. This dynamic child route should show the detail5 component.



Adjust the paths in the navigate methods of overview5 and detail5 accordingly. Query parameters should be removed and, instead, the AEvent id should now become part of the relative route path.

B. Now, the detail5 components 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 'param' observable instead of the 'queryParam' observable.
Overview5 should extract its selectedAEventIndex in a similar way from activatedRoute.children[0].params. That is a bit more complicated though, because if no child id has been provided in the route, no detail5 will not have been instantiated yet and activatedRoute.children[0] will be 'undefined' so that you cannot subscribe to its param observable. (You can workaround this issue by adding a convenient redirection in the children routes table.)

3.5 Template Driven Form validation.

With the FormsModule in Angular you can track the status of user input and implement input validation.

A. Create one new component detail42 in src/app/components/events which starts as a copy of detail4 (or detail41) of assignment 3.4.2 (or 3.4.3) initially. There will be no change in the interaction with overview4, so you can reuse that component: create a new route 'events/overview42' that opens overview4 and add a child route 'edit' which opens the new component detail42. Provide the new route 'events/overview42' in the 'events' sub-menu.

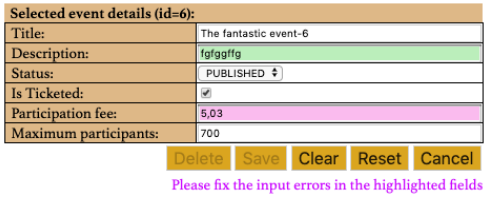
B. Redesign the editing form of detail42 to leverage the Angular FormsModule, i.e.
1. Use a <form> tag with a local reference to the NgForm to group your input controls.
2. Add the NgForm object attribute to the component class. Use the @ViewChild() decorator to bind this attribute to the local reference in the <form> tag.
(In Angular 8, @ViewChild has got an extra options parameter: provide {static:false})

```
@ViewChild('editForm', {static: false})
private detailForm: NgForm;
```

2. Register your <input> controls with the ngModel and a name attribute.
3. Provide an appropriate type on every <input> tag ("text", "number", etc.)

It is recommended to
1. Not provide the (ngSubmit) event binding in the <form> tag.
2. Not mark any <input> or <button> tags with type="submit" or type="reset"
3. Disable the default behaviour of the enter key by adding (keydown.enter)="$event.preventDefault()" to the <form>-tag.
(We have seen router guards generating submit events that act on your forms)

C. Add build-in validators to the <input> tags in the template. Use at least:
- required
- minlength
- pattern

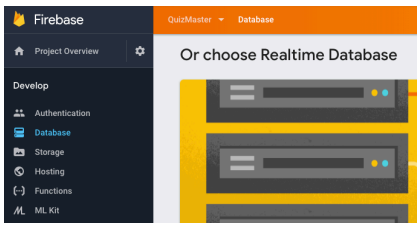


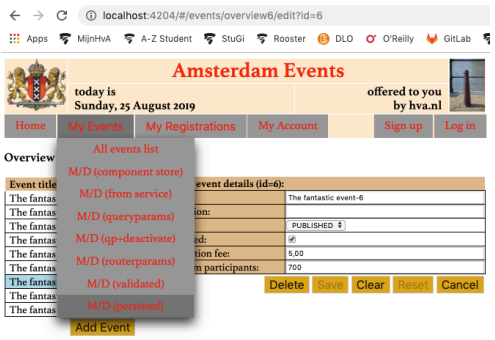
Specify CSS styling rules that highlight every invalid input field with some 'negative' color.
Specify CSS styling rules that highlight every dirty and valid input field with some 'positive' color (indicating what has been changed).
Make sure that dirty colors are reset after the 'Save' or 'Reset' button has been hit. (You may want to use the 'markAsPristine' method for that.)
Provide some error message if the form is invalid and also disable the Save button in that case.

3.6 Persistent storage with HTTPClient requests.

In this assignment you will explore HTTP requests to persist your data at a back-end service.

A. Create a new service a-events2 in src/app/services and copy all code of your AEventsService into AEvents2Service.
Create a firebase project (at firebase.google.com) with a realtime database in test mode. Copy the access url into a readonly property of your new AEvents2Service.

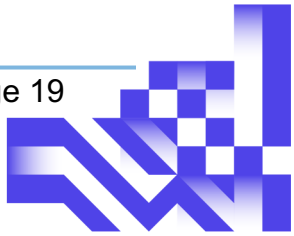




Create two new components overview6 and detail6 in src/app/components/events which initially start as copies of overview4 and detail42 of assignment 0.
Add a new route events/overview6 and add that to the My Events sub-menu in the menu bar.
Change the data type of the service injections into overview6 and detail6 to use the new AEvents2Service instead.
Test whether everything still works.

B. Provide HttpClient (from "@angular/common/http") in your app-module.ts and inject its instance into AEvents2Service. Add two new methods GetAllAEvents() and SaveAllAEvents() to AEvents2Service. These methods will use httpClient.get<AEvent[]>() and httpClient.put() methods to retrieve and store the events from and to firebase.
For now, make sure that any error conditions are logged onto the browser console.
Initialise the service by retrieving all known events upfront. If that list is empty, generate 5 random AEvents (and save those).
Invoke the SaveAllAEvents() method each time an event is added, updated or deleted (by a user).
Test the persistence of your operations.
Retest form validation, manual navigation in the address bar, and protection against loss of unsaved changes.

C. You may hit the challenge that either the overview6 or the detail6 component loads quicker than the service can complete retrieving the AEvents from the firebase store, and that you end-up with a partly empty master detail view. If that is the case, find a way to fix it. If needed, you may use the Javascript ‘setTimeout()’ function to retry initialisation of a component after waiting some time.



3.7 Authentication and authorisation.

In this assignment we will setup authentication in your frontend and firebase backend and configure different authorisation for authenticated users and visitors. You will explore the use of an HTTP_INTERCEPTOR which will automatically add the authentication token to each http request of the user's session.

3.7.1 Use of firebase authentication and authorisation

A. Prepare your firebase backend for authenticated access from your frontend web application, i.e.
1. Enable E-Mail/Password sign-in method within the authentication section of your firebase project.
2. Add two users with a password in the users list of your firebase project.
3. Configure a 'Web app' at the firebase Project Overview page, with standard access rules (including any localhost access).
4. Update and publish the database security rules, e.g. read access for all and write access for authenticated users only.

```
"rules": {
  ".read": true,
  ".write": "auth != null"
}
```

B. Now we will prepare a new root application component that integrates the firebase SDK:
1. Install the Firebase SDK into your Angular project from a terminal window:
'$ npm install --save firebase'
2. Create a new app-fb (root) component in src/app. This component should:

```
let firebaseConfig = {
    apiKey:
    authDomain: "amsterdamevents-d7770.firebaseapp.com",
    databaseURL: "https://amsterdamevents-d7770.firebaseio.com",
    projectId: "amsterdamevents-d7770",
    storageBucket: "",
    messagingSenderId: "474982691932",
    appId: "1:474982691932:web:2bcc79ca881df56b"
};

// Initialize Firebase
firebase.initializeApp(firebaseConfig);
```

a) import 'firebase/app'
b) initialise the firebase SDK with a copy of the firebaseConfig snippet that you can obtain from the WebApp settings in your backend.
c) use 'app-root' also as a component selector for AppFb, such that it can be loaded into the index.html root page of the application.

```
// bootstrap: [AppComponent]
bootstrap: [AppFbComponent]
```

Copy the html and css content from app.component into app-fb.component and change the bootstrap in app.module.ts to AppFbComponent.

Retest the persistent master detail of your application. You should still be able to navigate and edit all data. But when you try to save changes, an error should appear on the browser console, indicating you do not have write authentication.

```
▶ PUT https://amsterdamevents-d zone-evergreen.js:2952
7770.firebaseio.com/aevents.json 401 (Unauthorized)
                                  a-events3.service.ts:131
HttpErrorResponse {headers: HttpHeaders, status: 401
, statusText: "Unauthorized", url: "https://amsterda
mevents-d7770.firebaseio.com/aevents.json", ok: fals
e, …}
▶ error: {error: "Permission denied"}
▶ headers: HttpHeaders {normalizedNames: Map(0), la…
  message: "Http failure response for https://amste…
```

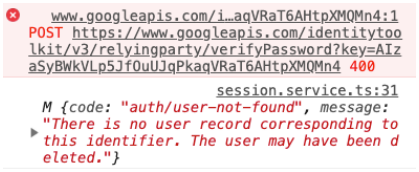
C. Create a new SessionService that will provide an adapter (interface) for the firebase authentication functions and track the currently logged on user for the session.
1. Import the firebase SDK and auth services.
2. Implement a method signOn(eMail:string, passWord: string) in this service which uses

```
import * as firebase from 'firebase/app';
import 'firebase/auth';
```


firebase SDK to sign on the user and track the name of the currently signed in user in a public member variable.
3. Implement a method signOff() in this service which uses firebase SDK to sign off the current user and reset the tracked name of the currently signed on user accordingly.

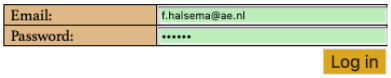
D. Create a new component header2 in src/app/components/mainpage which can be a copy of the original header component initially.
Welcome the currently logged-in user in the sub-title in this header at the right. If no user is logged-in, a 'Visitor' shall be welcomed.
Use this header in your app-fb component and retest your application.

E. 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.

F. Create a new component nav-bar2 in src/app/components/mainpage, which can be a copy of nav-bar initially. Use this new navigation bar in your app-fb root 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.

3.7.2 HTTP requests with authentication tokens.

Even if the user has been logged on, http put requests will not be accepted by the back-end because the back-end doesn't know any state of the frontend sending the request. In this assignment you will add secured tokens to the HTTP request to inform the backend about the authentication and authorisation status of the user. For that you apply an

HTTP_INTERCEPTOR service which automatically adds the token of the current user to every http-request that goes out.

A. Implement the getToken() and refreshToken() methods in the session service, which will track and give access to the user authentication token that is being provided and maintained by firebase. (Make sure that you cache a copy of the token immediately after signOn and that you discard the token after signoff).

B. Create a new class auth-interceptor in src/app which implements the HttpInterceptor interface from '@angular/common/http'.
Inject the session service into this interceptor such that it can use the token of the current user.
The 'intercept' method of the interceptor shall set the firebase 'auth' request parameter equal to the value of the token in (a clone of) every request, before forwarding it to the next handler.

```
export class AuthInterceptor implements HttpInterceptor {

    constructor(private session: SessionService) {
        console.log("New http interceptor");
    }

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

        let token = this.session.getToken();

        //console.log('intercept applies token: ' + token);
        if (token) {

            // TODO: add or override the auth query parameter in the request
            // and forward to the next handler

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

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

The code snippet provided here should get you going…
Make sure you provide the interceptor from app.module.ts
Re-test your application and verify that visitors cannot change data, while authenticated users can.

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.

3.8 TODO [BONUS] backend retrieval and updates of selected data.

TODO.

4 Assignments back-end frameworks: Spring-Boot

In these assignments you will build the back-end 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 two videos at https://learning.oreilly.com/home/:

1. Mastering Java Web Services and REST API with Spring Boot by Ranga Karanam
2. Master Hibernate and JPA with Spring Boot in 100 Steps by Ranga Karanam

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

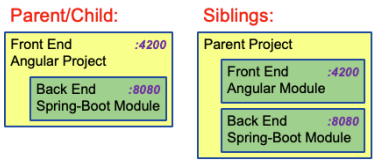
You will integrate these back-end assignments with your frontend solution of assignment 3.6 of the previous chapter. If you do not have a working version of assignment 3.6 also solutions of 3.5, 3.4 or 3.3 can be used to some extent.

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

In this assignment you create a basic Spring Boot back-end application and configure in there a simple REST Controller (Karanam, Chapter 4, step 4). The controller provides one resource end-point to access the AEvents of your application. These AEvents are actually managed in the Spring-Boot backend by an instance of an AEventRepository Interface. The AEventRepository is injected into the REST Controller by setter dependency injection (Karanam, Chapter 3, step 7). For now, you start with providing an AEventRepositoryMock bean implementation that just tracks an internal array of a-events, similar to how you did that before in an Angular service of your front end. 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.

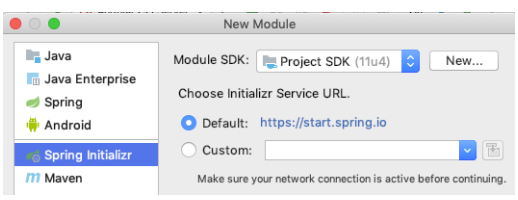
Below you find more detailed explanation of how you can implement the objectives of this assignment.

4.1.1 Configure the Spring-Boot back-end.

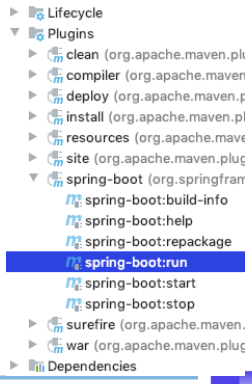


Basically, there are two approaches to combine a front end and a back-end module in an integrated development environment. Here we choose the Parent/Child approach, because that is straightforward proceeding from your solution of assignment **Error! Reference source not found.**:

A. 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).

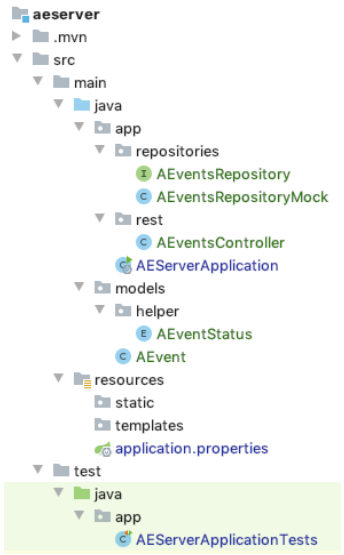


Choose the war format for your deployment package.
Activate the Spring Web dependency in your module.
Test your project setup by running the 'spring-boot:run' maven goal.

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

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

…



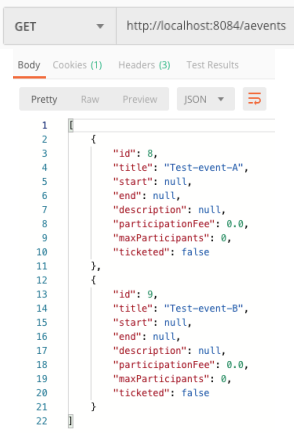
B. You may want to tidy-up the source tree of your module, similar to lay-out shown here. Always use the refactoring mode of IntelliJ to move files around such that the dependencies will be maintained. Basic rules for setup are:
   1. Your AEServerApplication class should reside within a non-default package (e.g. 'app'). (Otherwise, the autoconfig component scan may hit issues).
   2. Auto-configuration searches for beans only in your main application package and its sub-packages (e.g. '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.

C. Implement the AEvent model class and the AEventsController rest controller class, as explained by Ranga Karanam. Replicate your AEvent model from the front-end AEvent.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 just 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 two a-events as you expect:



D. Define an AEventsRepository interface and an AEventsRepositoryMock bean implementation class in the repositories package similar to the SortAlgorithm example by 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 AEvents with some random data.

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

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

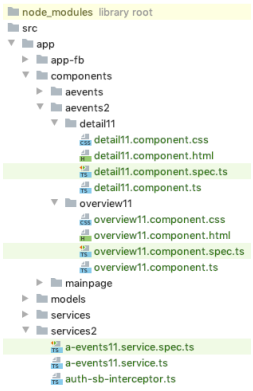
The AEventsRepository interface provides one method 'findAll()' that will be used by the endpoint in order to retrieve and return all AEvents:

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:

4.1.2 Hook-up with your Angular front end.

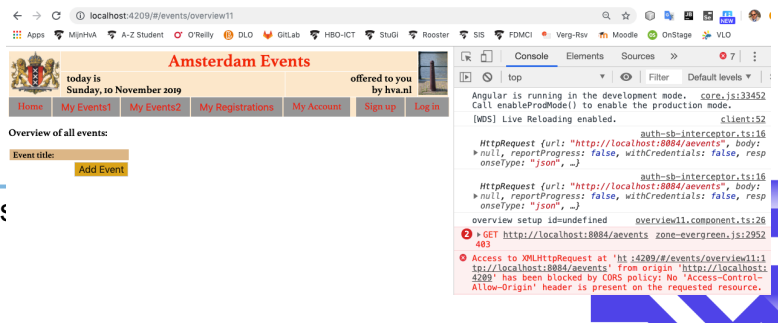
A. Replicate 'overview6' and 'detail6' components and 'a-events2' service of assignment **Error! Reference source not found.** to new 'overview11' and 'detail11' components and 'a-events11' service.
To keep your Angular source tree tidy and well organised, this is a good time to start a new components folder and services folder:
src/app/components/aevents2
src/app/services2
The only change to overview11 and detail11 is to inject AEvents11service.
The only change to AEvents11service is to use a different back-end URL: http://localhost:8083/aevents
(Check the port number with your application.properties configuration in the back-end).
The constructor of AEvents11service should start with retrieving all a-events from the back-end. (As implemented also in assignment **Error! Reference source not found.**).



Create a new route events/overview11 in your router module, and configure that route in your navigation-bar. (You may want to start a new main drop-down menu in your menu bar for all the work that will come hereafter).

If you had configured firebase in assignment 3.6, that shall be disabled for now.
If you had configured an HttpInterceptor for firebase authentication, you may replace that interceptor with another class, that just logs all http requests to your console without changing them.

Launch both the Angular front-end and the Spring-boot back-



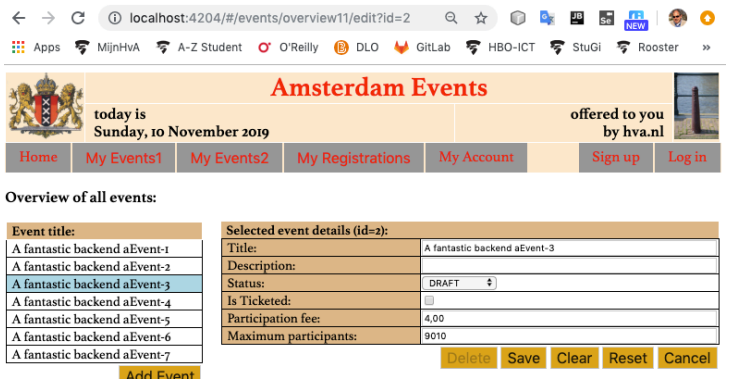
end and verify whether your AEvents maintained in the back-end repository show up: (Open the console while trying)

B. The CORS error indicates that the back-end does not authorise Cross Origin Resource Sharing (see https://en.wikipedia.org/wiki/Cross-origin_resource_sharing). The frontend, running at localhost:4204 is not authorised to share resources from the back-end running at localhost:8084.

   At https://www.onlinetutorialspoint.com/spring-boot/how-to-enable-spring-boot-cors-example-crossorigin.html is explained how to use the @CrossOrigin annotation to configure a Spring-Boot REST controller to enable such resource sharing.

   Implement this annotation and try again:

   Now you are able the retrieve and show the data from your own Spring-boot back-end service. In the next assignment you will address full CRUD.



4.2 Provide CRUD operations

In this assignment you will enhance the repository interface and the /aevents REST-api with end-points to create new AEvents and get, update or delete specific 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. You will configure the CORS globally, and also rework some of your frontend application to be able to add a single event to the back-end.

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, MappingJacksonValue.

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

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

4.2.1 Enhance your REST Controller

A. In order to be able to identify individual events we need to introduce some kind of key. When using JPA and an ORM it is common practice to use a long Id to identify

```
public class AEvent {

    private long id;        // identification of an AEvent
                            // unique id-s are generated by the back-end persistence layer
    private String title;
```

each entity uniquely and let the back-end persistence layer generate such unique id-s. Consequently, we cannot make further assumptions on the actual value ranges of these Id-s. We will let go of the approach that id-s match the range 0 .. nAEvents-1. From now on we represent the id of an aEvent explicitly in its functional class

B. Enhance the AEventsRepositoryMock class with actual implementations of all CRUD methods as listed in the AEventsRepository interface above.
For now, let the AEventsRepositoryMock class handle the generation of new unique ids. Later, that responsibility will be moved deeper into the back-end to the ORM.

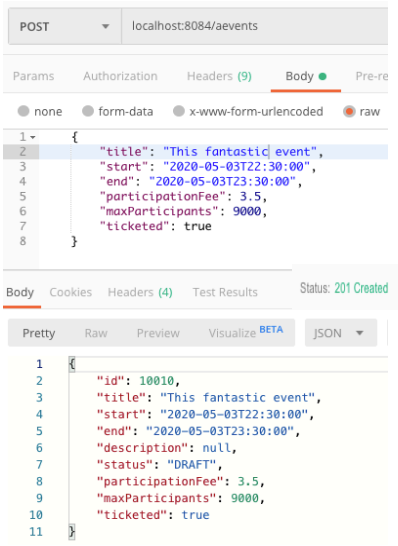
C. Enhance your REST AEventsController with the following endpoints:
- a GET mapping on '/aevents/{id}' which uses repo.findById(id) to deliver the event that is identified by the specified path variable.
- a POST mapping on '/aevents' which uses repo.save(aEvent) to add a new event to the repository and deliver that event with its newly generated id
- a PUT/POST mapping on '/aevents/{id}' which uses repo.save(aEvent) to

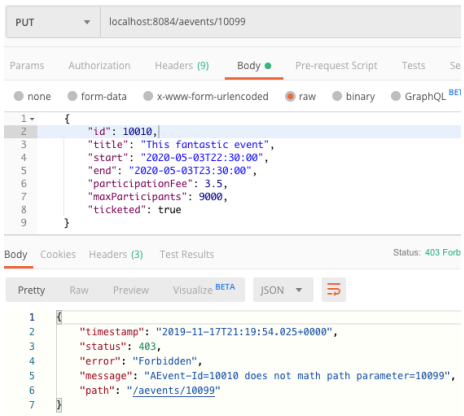
update/replace the stored event identified by id.
- a DELETE mapping on '/aevents/{id}' which used repo.deleteById(id) to remove the identified event from the repository.

Test the new mappings with postman.

D. 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.





E. Implement Custom Exception handling in your REST AEventsController:
- throw a ResourceNotFound exception on get and delete requests with a non-existing id.
- throw a Forregistrationden exception on a PUT/POST 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.

F. Implement a dynamic filter on the getAllAEvents() mapping at '/aevents', which only returns the id, title and status of every aEvent. Dynamic filters can most easily be implemented with a @JsonView specification in your AEvent model in combination with the use of the MappingJacksonValue.setSerializationView method in the mapping of your AEventsController.

Test the change with postman.

G. Replace your local CORS configuration in the AEventsController by a global configuration class which implements WebMvcConfigurer.

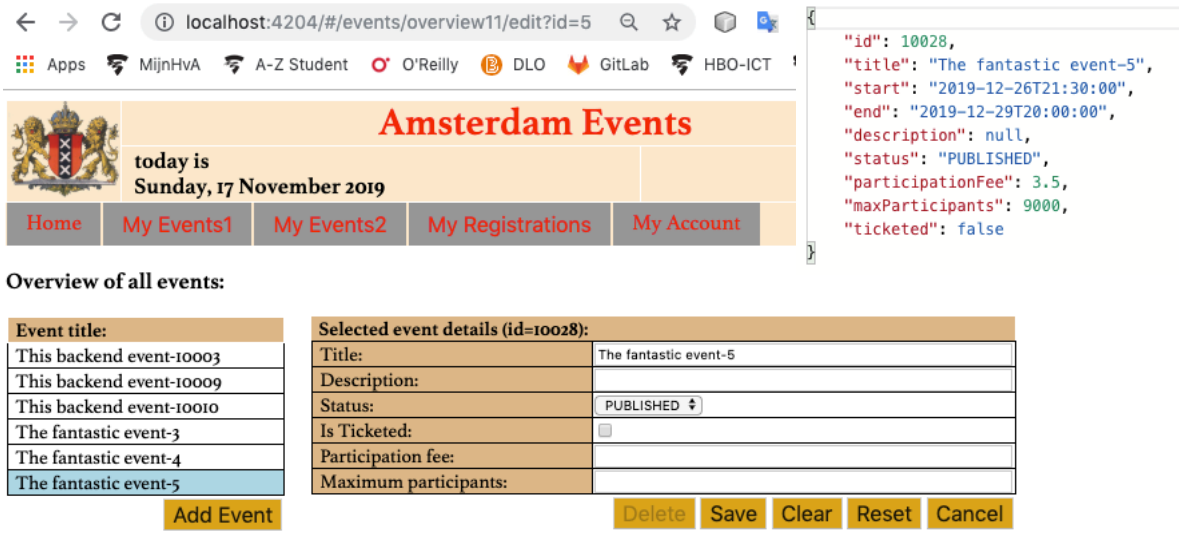
4.2.2 Reconnect your Angular front end.

A. Connect the ‘Add’ button in your Angular user interface to the POST mapping on ‘/aevents’ in your back-end.
Currently, the ‘Add’ button in your overview11 component probably connects to some ‘addRandomAEvent()’ in the AEvents11Service. Then only minor changes should be required:
- Update your AEvent class to include the new id attribute of type ‘bigint’.

```
export class AEvent {
    public id: bigint;
    public title: string;
    public start: Date;
```

- Post the newly created aEvent to the backend at ‘/aevents’ and receive it again in the response with its newly generated id. Retain this updated copy in your local aEvents list.
- Update your details11 view to show the true id in the header.

Test the ‘Add AEvent’ button in your user interface:
Verify the contents with postman
Notice that after a page reload in your frontend, values of number of registrations and highest registration have been lost…. How come? Postman still shows them in the backend… !



4.3 JPA and ORM configuration

In this assignment you will configure data persistence in the back-end 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. 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.

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 events. It will produce error responses on registrations for events that are not PUBLISHED and on registrations which would exceed the maximum participation of an event.

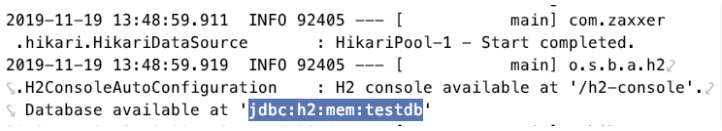
Relevant introductions into the topics you find in the O'Reilly course 'Master Hibernate and JPA with Spring Boot in 100 Steps' by Ranga Karanam chapters 3 and 5.

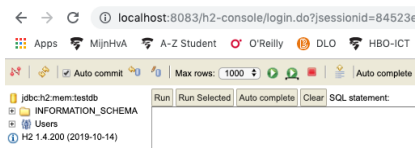
In this assignment you should practice hands-on experience with Spring-Boot annotations @Entity, @Id, @GeneratedValue, @Enumerated, @OneToMany, @ManyToOne, @Repository, @PersistenceContext, @Primary, @Transactional, @Qualified, @JsonIgnore,
and classes EntityManager and TypedQuery ….

4.3.1 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 RangaJPA.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 and logging.level.org.hibernate.type=trace in application.properties such that you can trace the SQL queries being fired.

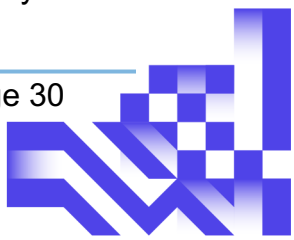
   Relaunch the server app, and use the h2-console to check-out that H2 is running.
(Retrieve your proper JDBC URL from the spring start-up log.)
(Spring Boot has auto-configured the H2 data source for you.)
(You do not need to create tables or load data into the database using plain SQL)





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. An entity manager should be injected into this new class providing 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 back-end 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 event with postman doing a post at localhost:8080/aevents.

```
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*	2019-12-01 15:30:00	FALSE	0	0.0	2019-11-30 19:30:00	1	This backend event-0

Verify the associated SQL statements in the Spring Boot log and use the h2-console to verify whether the 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.

C. Also implement and test the other three methods of your AEventsRepository interface (deleteById, findById and findAll).; Use a named query to implement the findAll method).
Implementation of the findAll method requires use of a JPQL query. The use of JPQL is explained in RangaJPA.Ch5.Step15, and RangaJPA.Ch10. In assignment **Error! Reference source not found.** 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 events repository into your main application class and implement the CommandLineRunner interface (as shown by RangaJPA.Ch3.Step6).
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 back-end, because the details of the generated H2 SQL schema will change as you progress your Java entities.
Also, this CommandLineRunner initialisation will also work with your Mock repository implementation.

```
@Transactional
protected 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
```


Make sure to configure transactional mode on the new initialisation method.

4.3.2 Configure a one-to-many relationship

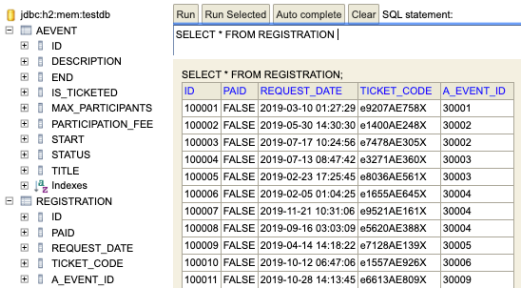
A. Now is the time to introduce a second entity. Make a new model class 'Registration' identified by an attribute named 'id' (long). Also record other relevant information about registrations:
   ticketCode (String)
   paid (boolean)
   submissionDate (LocalDateTime)

   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 or the Repository Interface needs extension…!!! That may motivate you to implement an approach of the(optional) bonus assignment 4.3.3 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 events repository code….

C. Extend the initialisation in the command-line-runner of task 4.3.1-D to add a few registrations to every event 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') 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 REGISTRATION table:

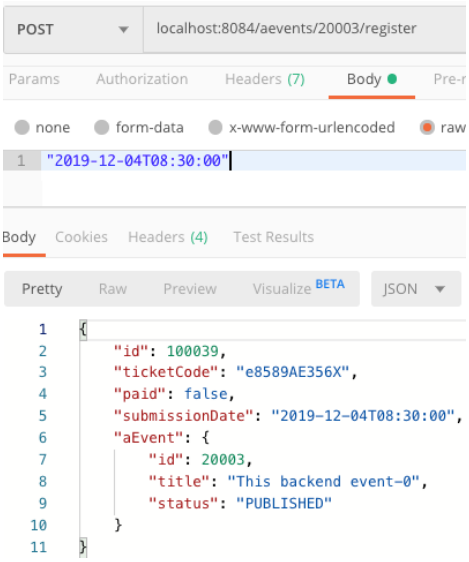


D. Re-test the REST API at localhost:8080/aevents with postman:
   You may find a response like here with endless recursion in the JSON structure. For now, you may fix that with an appropriate @JsonIgnore in one of

your model classes.
With (optional) bonus assignment 4.4.2 you can practice a better solution with custom Json serializers.

E. Implement a POST mapping on the /aevents/{id}/register REST endpoint. This mapping should add a new registration to the event.

The submission dateTime should be taken as an (optional) parameter from the body of the POST request. If the body does not provide a submission dateTime, then the local date time at the server should be used for the submission date of the new registration.



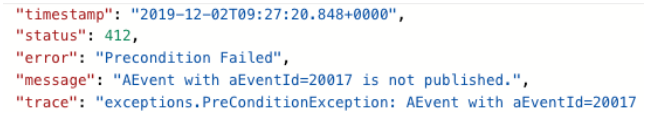
An error response should be provided if the event does not have status 'PUBLISHED'

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

Leverage appropriate methods in your AEvent class to implement the business logic for creating a new trip with a scooter, e.g.:

```
public Registration createNewRegistration(LocalDateTime startDateTime)

public boolean addRegistration(Registration registration)
```

The request should return the details of the new trip.



Test your new end-point with postman.

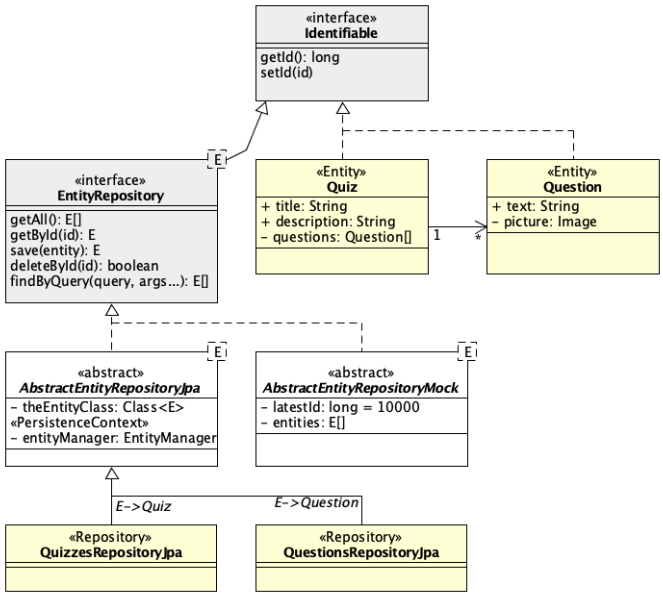
4.3.3 [BONUS] Generalized Repository

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;
```


4.4 JPQL queries and custom JSON serialization

In this assignment you will explore JPQL queries. You will extend the /events REST end-point to optionally accept a request parameter '?title=XXX' or '?status=XXX' or '?minRegistrations=999', and then filter the list of 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 events that actually meet the criteria will retrieved from the back-end.

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, @Enumerated, @JsonView, @JsonSerialize and class MappingJacksonValue….

4.4.1 JPQL queries

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

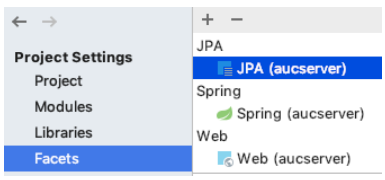
```
List<E> findByQuery(String jpqlName, Object... params);
                                    // finds all instances from a named jpql-query
```

(Here we assume you use the generic repository interface)

This method should accept the name of a predefined query which may include specification of 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 events of a given status
"AEvent_find_by_title": finds all events that have a given sub-string in their title
"AEvent_find_by_minRegistrations": finds all events that have at least the given number of registrations
Use an ordinal(positional) query parameter as a place-holder for the value to be given.
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)

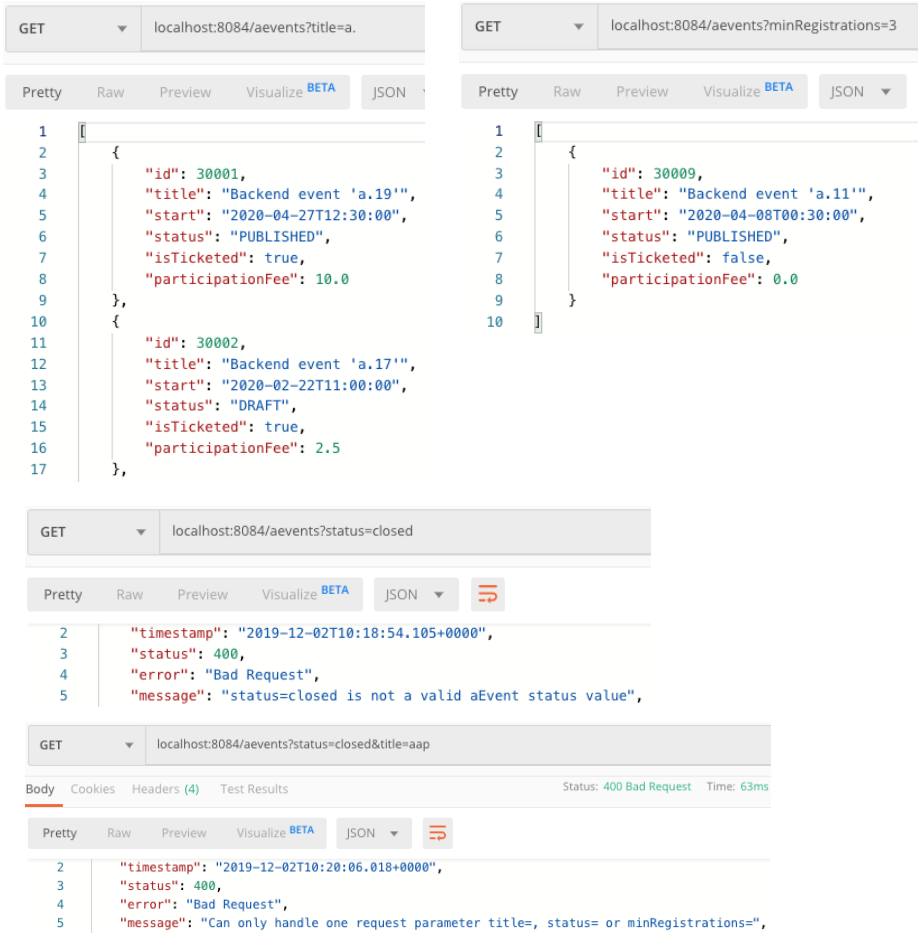


C. Extend your GetMapping on the "/aevents" REST end-point to optionally accept a '?title=XXX' or '?status=XXX' or '?minRegistrations=999' request parameter.

If no request parameter is provided, the existing functionality of returning all 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 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 a scooter.

Test the behaviour of your end-point with postman:



4.4.2 [BONUS] Custom JSON Serializers

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

These can be broken by placing @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 AEvent, you want full information about the 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 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 AEvent.registrations:

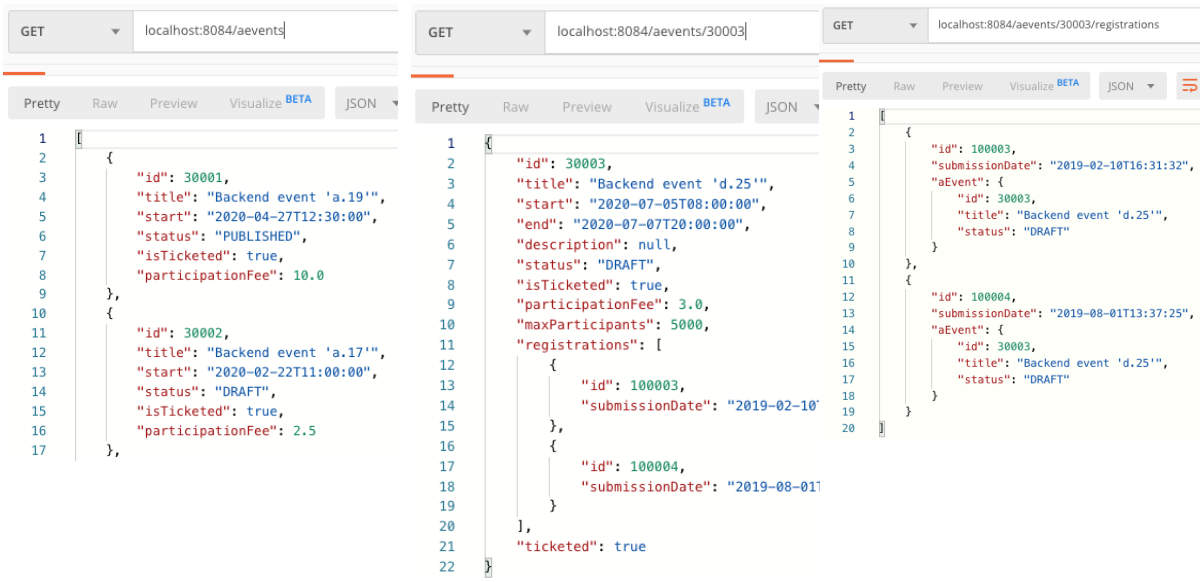
```
@JsonView({CustomJson.Summary.class})
@JsonSerialize(using = CustomJson.ShallowSerializer.class)
private List<Registration> registrations = null;
```


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

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/events and localhost:8080/events/{eventId}/registrations end-points.
Implement unrestricted end-points at localhost:8080/events/{eventId} and localhost:8080/events/{eventId}/registrations/{registrationId}.

Test your end-points with postman:



4.5 Authentication, JSON Web Tokens (JWT) and Session Management

In this assignment you will implement end-to-end secure authentication and authorisation.
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 back-end.
By the end of this assignment you will have further explored annotations @RequestBody, @RequestAttribute, @Value
and classes ObjectNode, Jwts, Jws<Claims>, SignatureAlgorithm

4.5.1 Back-end security configuration.

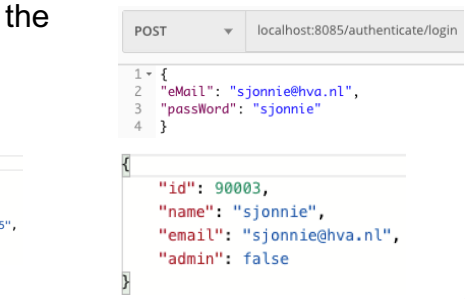
The back-end security configuration involves two components:
1. A REST controller at '/authenticate' which provides for user registration and user login.
This end-point will be 'in-secure', i.e. open to all clients: also to non-authenticated clients.
After successful login, a security token will be added 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.

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.
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.

```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 e.g. into your APIConfig bean using the @Value annotation:

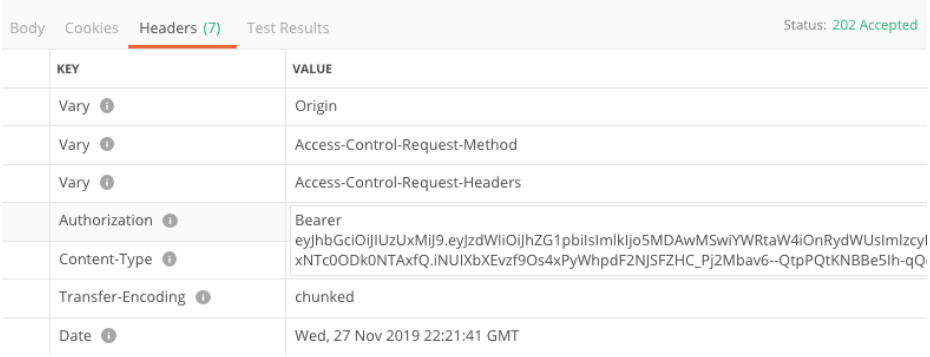
```
// JWT configuration that can be adjusted from application.properties
@Value("${jwt.issuer:private company}")
private String issuer;

@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:



C. 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.

D. 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;
    }
}
```

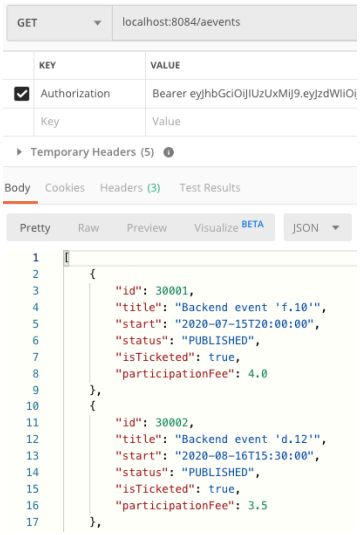

This decode method uses the same JWToken attributes and getKey method that were also shown earlier along with the encode method.

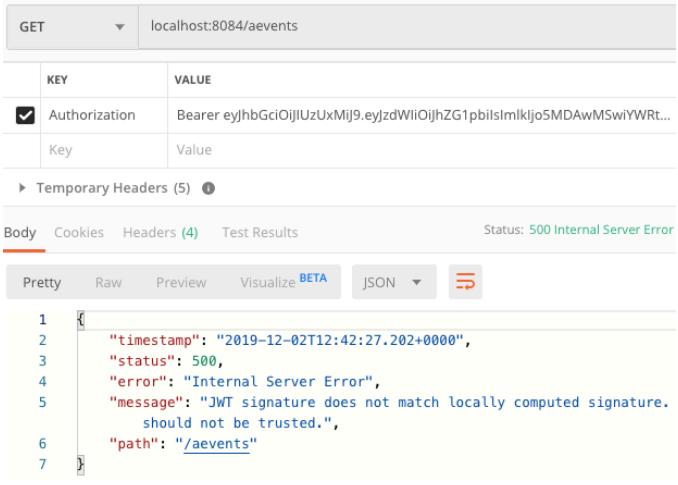
If all has gone well, we add the decoded token information into an attribute of the request and allow the request to progress further down the chain:

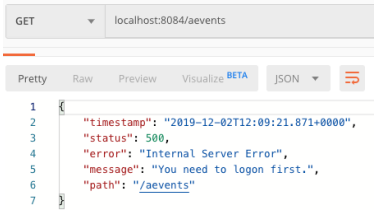
```
// pass-on the token info as an attribute for the request
request.setAttribute(JWToken.JWT_ATTRIBUTE_NAME, jwToken);

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…)







E. Now, all is 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");
        }
}
```

4.5.2 Authentication from the Angular frontend

With a back-end 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.

A. Adjust the signIn method of the session service such that it posts a request to your /authenticate/login endpoint with the user's eMail and passWord in the request body:

```
signIn(email: string, password: string, targetUrl?: string) {
  console.log("login " + email + "/" + password);
  let oObservable =
      this.http.post<HttpResponse<User>>(this.REST_BASE_URL + "/authenticate/login",
            { eMail: email, passWord: password },
            { observe: "response" });
  oObservable
      .subscribe(
          response => {
            console.log(response);
            this.setToken(
                response.headers.get('Authorization'),
                ((response.body as unknown) as User).name
            );
          },
          error => {
            console.log(error);
            this.setToken(null, null);
          }
      )
  return oObservable;
}
```

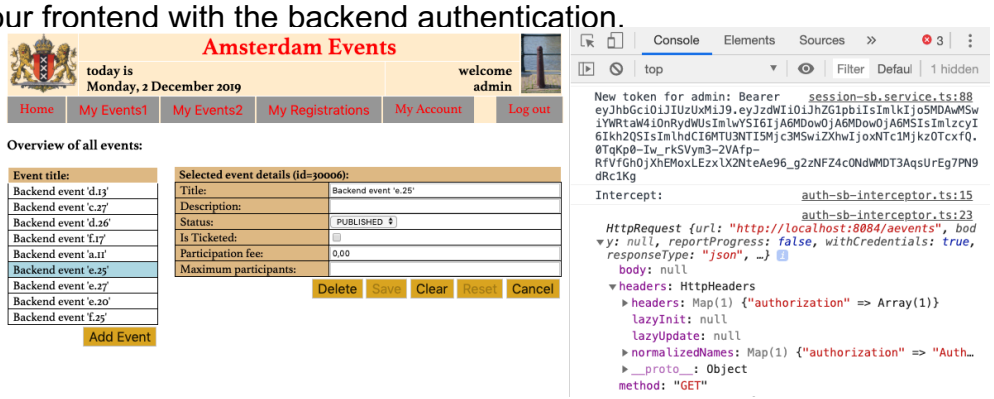
Make sure that the naming of attributes in your request body matches exactly the expectations of the back-end.

In the request options, you specify 'observe: "response"'.
That way, you get access to the complete response, including its headers.
You extract the token from the Authorization header in the response and keep it for later.
You also retrieve the user name from the response body, for display in the header component of your page.

B. Adjust your HttpInterceptor to include the JWT token with every (relevant) request similar to your solution with firebase authentication in assignment **Error! Reference source not found.**:

```
let token = this.session.getToken();
if (token == null) {
    return next.handle(req);
} else {
    const cloned =
        req.clone({ setHeaders: { Authorization: token } });
    console.log(cloned);
    return next.handle(cloned);
}
```

Test your frontend with the backend authentication.



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

```
// allow for different user sessions from the same computer
getToken():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);
    }
    return token;
}
```

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

D. 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.

4.5.3 [BONUS] Full User Account Management.

A. Implement the User Repository in the back-end. 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.

4.6 [TODO] Inheritance and performance optimization

Inheritance
Fetch types LAZY, EAGER, Deep Query
Cascade types ALL, PERSIST, DELETE
MySQL implementation
Performance tuning, first and second level cache

Bonus: XML file upload ???