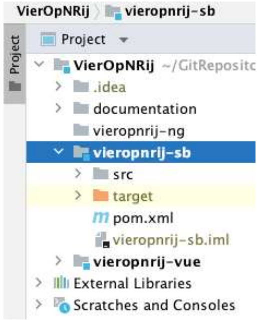
3.5 Setup of the Spring-boot application with a simple REST controller.

In the following assignments you will build the backend part of the ‘‘Vier Op N Rij” application as depicted in the full stack, layered logical architecture of section 2.3. You will use the Spring-Boot technology.
You create a basic Spring Boot backend application and configure in there a simple REST Controller (O'Reilly-2, Chapter 4, step 4). The controller provides one resource endpoint to access the games of your application.

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

As of assignment 3.7 you will integrate the backend capabilities with your frontend solution of assignment **Error! Reference source not found.**

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

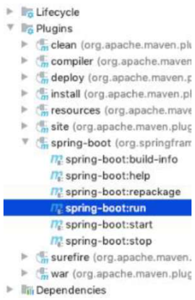


VierOpNRij 〉 vieropnrij-sb
Project ▾
VierOpNRij ~/GitReposito
.idea
documentation
vieropnrij-ng
vieropnrij-sb
src
target
pom.xml
vieropnrij-sb.iml
vieropnrij-vue
External Libraries
Scratches and Consoles

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

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

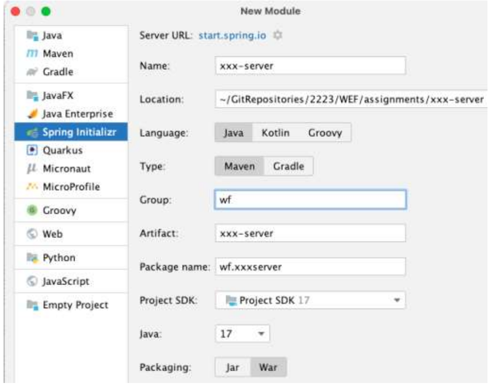
A. Create a Spring-Boot application module ‘vieropnrij-sb’ within the parent project using the Spring Initializr plugin.
(You may need to install/activate the Spring plugins first in your Intellij settings/preferences).



Lifecycle
Plugins
clean (org.apache.maven.plu
compiler (org.apache.maven
deploy (org.apache.maven.p
install (org.apache.maven.pl
resources (org.apache.mave
site (org.apache.maven.plug
spring-boot (org.springfram
spring-boot:build-info
spring-boot:help
spring-boot:repackage
spring-boot:run
spring-boot:start
spring-boot:stop
surefire (org.apache.maven.
war (org.apache.maven.plug
Dependencies

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

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



New Module

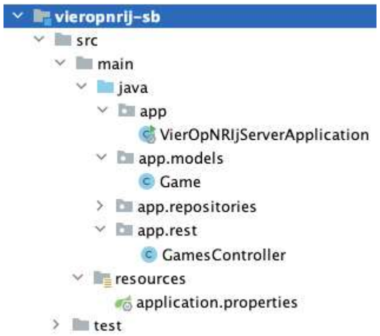
Java
Maven
Gradle
JavaFX
Java Enterprise
Spring Initializr
Quarkus
Micronaut
MicroProfile
Groovy
Web
Python
JavaScript
Empty Project

Server URL: start.spring.io
Name: xxx-server
Location: ~/GitRepositories/2223/WEF/assignments/xxx-server
Language: Java Kotlin Groovy
Type: Maven Gradle
Group: wf
Artifact: xxx-server
Package name: wf.xxxserver
Project SDK: Project SDK 17
Java: 17
Packaging: Jar War

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

```
server.port=8087
server.servlet.context-path=/
logging.level.org.springframework = info
server.error.include-message=always
```

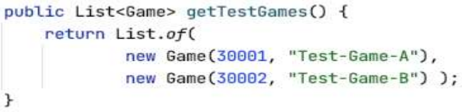
B. You may want to tidy-up the source tree of your module like shown here. Always use the refactoring mode of IntelliJ to move files around such that the dependencies will be sustained. Basic rules for setup are:
  1. Your VierOpNRijServerApplication class should reside within a non-default package (e.g. ‘app’). (Otherwise, the autoconfig component scan may hit issues).
  2. Autoconfiguration searches for beans only in your main application package and its sub-packages (e.g. ‘models’, ‘repositories’ and ‘rest’ in this example).
  3. The package structure under ‘test/java’ should match the source structure under ‘main/java’



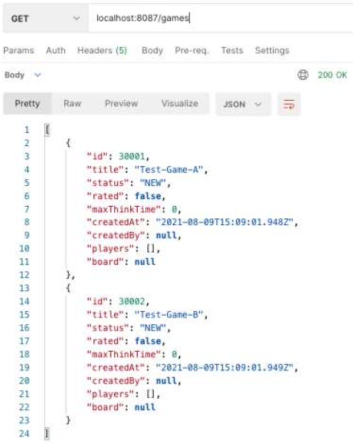
```
vieropnrij-sb
  src
    main
      java
        app
          VierOpNRIjServerApplication
        app.models
          Game
        app.repositories
        app.rest
          GamesController
      resources
        application.properties
    test
```

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

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



```
public List<Game> getTestGames() {
    return List.of(
            new Game(30001, "Test-Game-A"),
            new Game(30002, "Test-Game-B") );
}
```



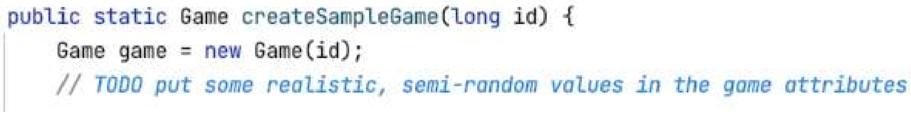
localhost:8087/games

```
[
    {
        "id": 30001,
        "title": "Test-Game-A",
        "status": "NEW",
        "rated": false,
        "maxThinkTime": 0,
        "createdAt": "2021-08-09T15:09:01.948Z",
        "createdBy": null,
        "players": [],
        "board": null
    },
    {
        "id": 30002,
        "title": "Test-Game-B",
        "status": "NEW",
        "rated": false,
        "maxThinkTime": 0,
        "createdAt": "2021-08-09T15:09:01.949Z",
        "createdBy": null,
        "players": [],
        "board": null
    }
]
```

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 games like you expect.

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

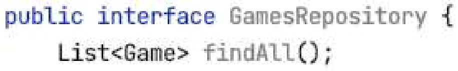
The GamesRepositoryMock bean should manage an array of games. Let the constructor of GamesRepositoryMock setup an initial array with 7 games with some semi-random sample data.
The static method to create some sample game can best be implemented in the Game class itself:



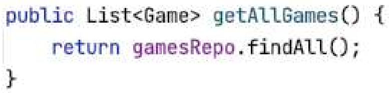
```
public static Game createSampleGame(long id) {
    Game game = new Game(id);
    // TODO put some realistic, semi-random values in the game attributes
```

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

The GamesRepository interface provides one method ‘findAll()’ that will be used by the endpoint in order to retrieve and return all games:



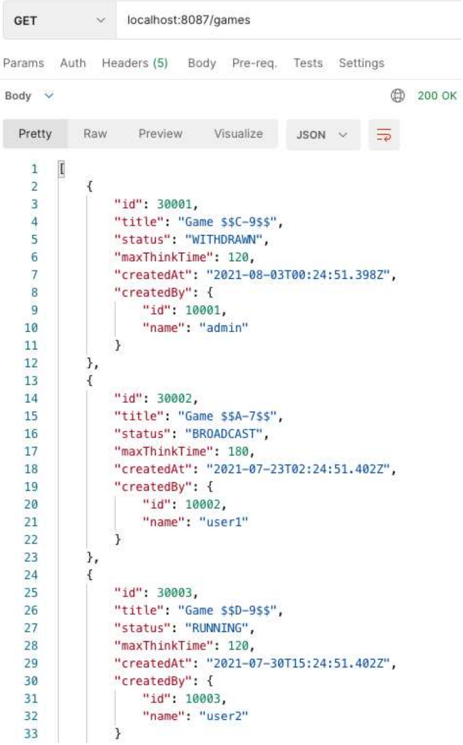
```
public interface GamesRepository {
    List<Game> findAll();
```



```
public List<Game> getAllGames() {
    return gamesRepo.findAll();
}
```

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

Test your endpoint again with Postman:



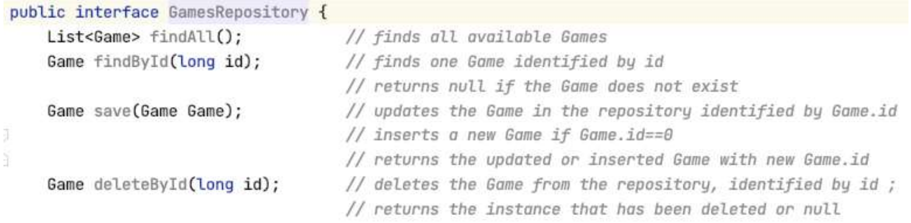
```
[
    {
        "id": 30001,
        "title": "Game $$C-9$$",
        "status": "WITHDRAWN",
        "maxThinkTime": 120,
        "createdAt": "2021-08-03T00:24:51.398Z",
        "createdBy": {
            "id": 10001,
            "name": "admin"
        }
    },
    {
        "id": 30002,
        "title": "Game $$A-7$$",
        "status": "BROADCAST",
        "maxThinkTime": 180,
        "createdAt": "2021-07-23T02:24:51.402Z",
        "createdBy": {
            "id": 10002,
            "name": "user1"
        }
    },
    {
        "id": 30003,
        "title": "Game $$D-9$$",
        "status": "RUNNING",
        "maxThinkTime": 120,
        "createdAt": "2021-07-30T15:24:51.402Z",
        "createdBy": {
            "id": 10003,
            "name": "user2"
        }
```

3.6 Enhance your REST controller with CRUD operations

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

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

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

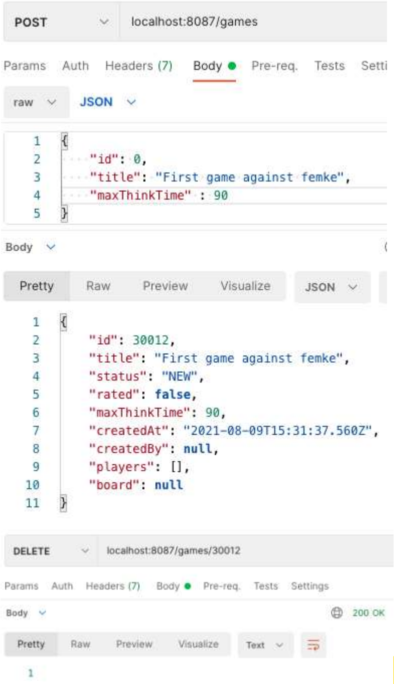


```
public interface GamesRepository {
    List<Game> findAll();              // finds all available Games
    Game findById(long id);            // finds one Game identified by id
                                       // returns null if the Game does not exist
    Game save(Game Game);            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

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

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

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



POST localhost:8087/games

```
{
    "id": 0,
    "title": "First game against femke",
    "maxThinkTime" : 90
}
```

```
{
    "id": 30012,
    "title": "First game against femke",
    "status": "NEW",
    "rated": false,
    "maxThinkTime": 90,
    "createdAt": "2021-08-09T15:31:37.560Z",
    "createdBy": null,
    "players": [],
    "board": null
}
```

DELETE localhost:8087/games/30012

200 OK

Test the new mappings with postman.

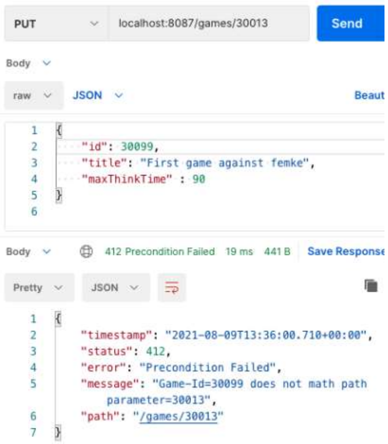
C. Implement Custom Exception handling in your REST GamesController:

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

Test the mapping with postman.

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

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



PUT localhost:8087/games/30013

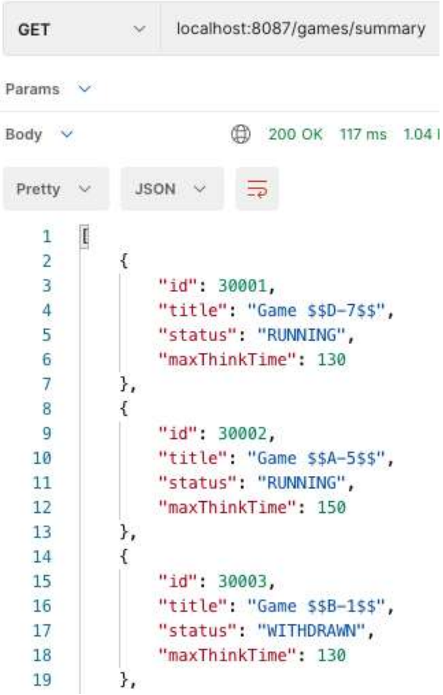
```
{
    "id": 30099,
    "title": "First game against femke",
    "maxThinkTime" : 90
}
```

412 Precondition Failed 19 ms 441 B

```
{
    "timestamp": "2021-08-09T13:36:00.710+00:00",
    "status": 412,
    "error": "Precondition Failed",
    "message": "Game-Id=30099 does not math path
        parameter=30013",
    "path": "/games/30013"
}
```

D. Implement a dynamic filter at a getGamesSummary() mapping at '/games/summary', which only returns the id, title, status and maxThinkTime of every game. Dynamic filters can most easily be implemented with a @JsonView specification in your Game class in combination with the same view class annotation at the request mapping in the rest controller.

Test the mapping with postman.



GET localhost:8087/games/summary

200 OK 117 ms 1.04 l

```
[
    {
        "id": 30001,
        "title": "Game $$D-7$$",
        "status": "RUNNING",
        "maxThinkTime": 130
    },
    {
        "id": 30002,
        "title": "Game $$A-5$$",
        "status": "RUNNING",
        "maxThinkTime": 150
    },
    {
        "id": 30003,
        "title": "Game $$B-1$$",
        "status": "WITHDRAWN",
        "maxThinkTime": 130
    },
```

3.7 Connect the FrontEnd via the JavaScript Fetch-API

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

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

3.7.1 A REST Adaptor for Fetch-API requests.

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

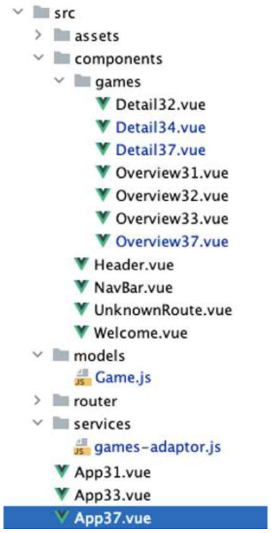
Additionally, you will configure the CORS in the backend to support use of multiple ports for different backend services

A. Replicate 'Overview33/Overview34' and 'Detail34' components of assignment 3.4 into new components 'Overview37' and 'Detail37'
Create a new route /games/overview37 which invokes component GamesOverview37 with child component GamesDetail37. Add an option for this route to your games menu.
Replicate 'App33' into a new 'App37' component and don't forget to launch this App37 from main.js.

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

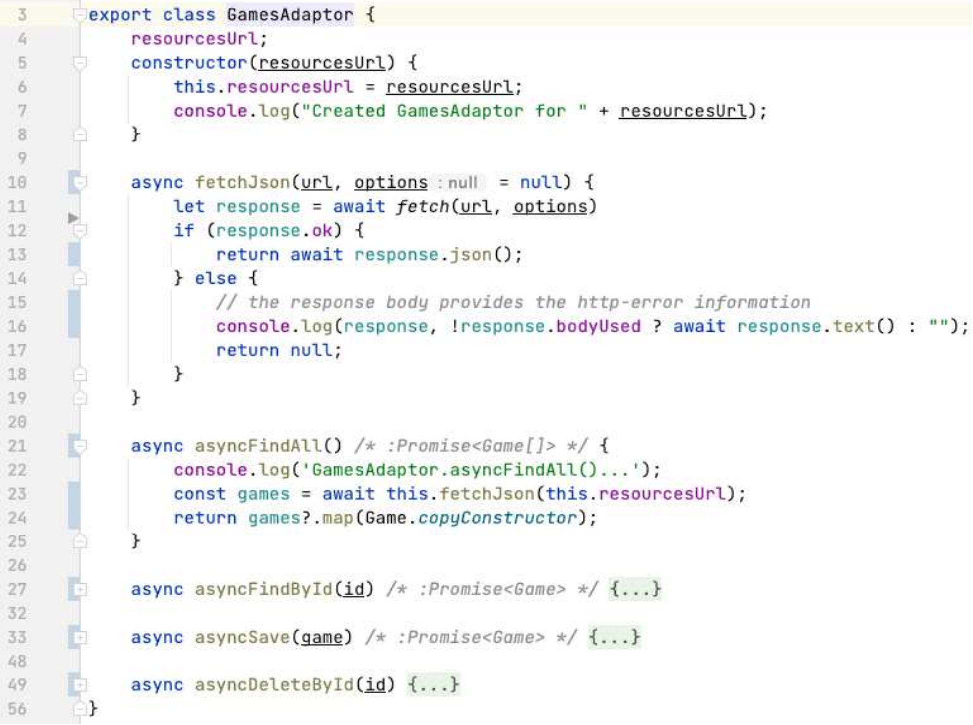
B. Next, create a services folder and implement a games adaptor, which will provide connectivity to the /games endpoint mapping of the backend.
Four basic CRUD operations shall be implemented by this adaptor:
asyncF**indAll()** retrieves the list of all games
asyncF**indById(id)** retrieves one game, identified by a given id
asyncS**ave(game)** saves an updated or new game
and returns the saved instance.
asyncD**eleteById(id)** deletes the game identified by the given id.

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



```
src
  assets
  components
    games
      Detail32.vue
      Detail34.vue
      Detail37.vue
      Overview31.vue
      Overview32.vue
      Overview33.vue
      Overview37.vue
    Header.vue
    NavBar.vue
    UnknownRoute.vue
    Welcome.vue
  models
    Game.js
  router
  services
    games-adaptor.js
  App31.vue
  App33.vue
  App37.vue
```

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



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

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

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

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

    async asyncSave(game) /* :Promise<Game> */ {...}

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

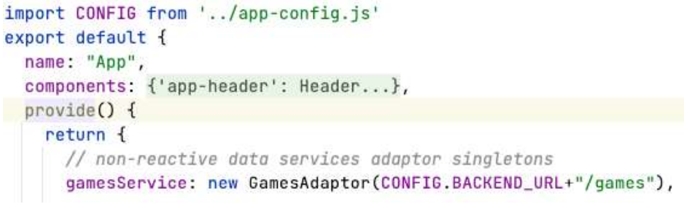
The resourcesUrl in the constructor specifies the endpoint of the backend API.

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

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

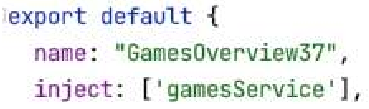
C. Next, we provide a singleton instance of this GamesAdaptor from App37 to be shared across all active components: Vue.js provides Dependency Injection for that (See provide/inject at https://v3.vuejs.org/guide/component-provide-inject.html).

The App37 component creates and provides the singleton instance:

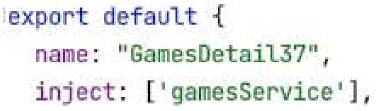


```
import CONFIG from '../app-config.js'
export default {
  name: "App",
  components: {'app-header': Header...},
  provide() {
    return {
      // non-reactive data services adaptor singletons
      gamesService: new GamesAdaptor(CONFIG.BACKEND_URL+"/games"),
```

The Overview37 and Detail37 components both inject the instance:

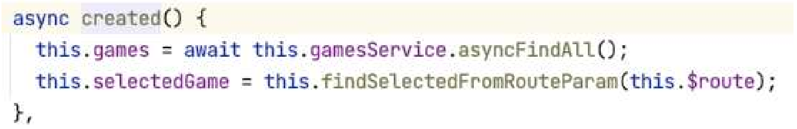


```
export default {
  name: "GamesOverview37",
  inject: ['gamesService'],
```



```
export default {
  name: "GamesDetail37",
  inject: ['gamesService'],
```

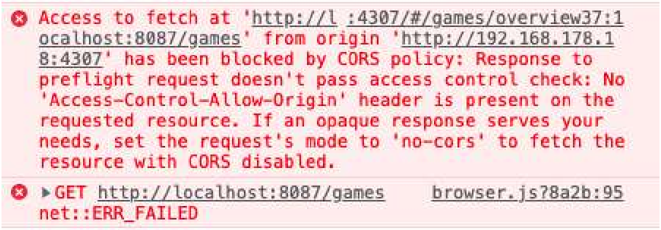
With that, these components are both set up to use the service adaptor to interact with the backend. E.g. The GamesOverview37 component can now dynamically initialise its list of games from the backend within its 'created()' lifecycle hook:



```
async created() {
  this.games = await this.gamesService.asyncFindAll();
  this.selectedGame = this.findSelectedFromRouteParam(this.$route);
},
```

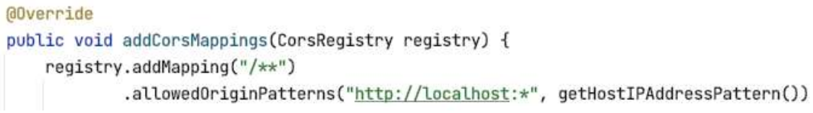
Notice that we have changed the 'created()' hook into an a-synchronous function. Every function that waits for the result of another a-synchronous function must itself also be coded as an a-synchronous function (such that its caller also can wait for the result).

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



```
Access to fetch at 'http://l :4307/#/games/overview37:1
ocalhost:8087/games' from origin 'http://192.168.178.1
8:4307' has been blocked by CORS policy: Response to
preflight request doesn't pass access control check: No
'Access-Control-Allow-Origin' header is present on the
requested resource. If an opaque response serves your
needs, set the request's mode to 'no-cors' to fetch the
resource with CORS disabled.
▶GET http://localhost:8087/games    browser.js?8a2b:95
net::ERR_FAILED
```

If your backend REST service is provided from a different port (8087) than your frontend UI site (4307), you must configure your backend to provide Cross Origin Resource Sharing (see https://en.wikipedia.org/wiki/Cross-origin_resource_sharing).
For that, add a global configuration class to your backend which implements the WebMvcConfigurer interface. In this class you need to implement addCorsMappings.



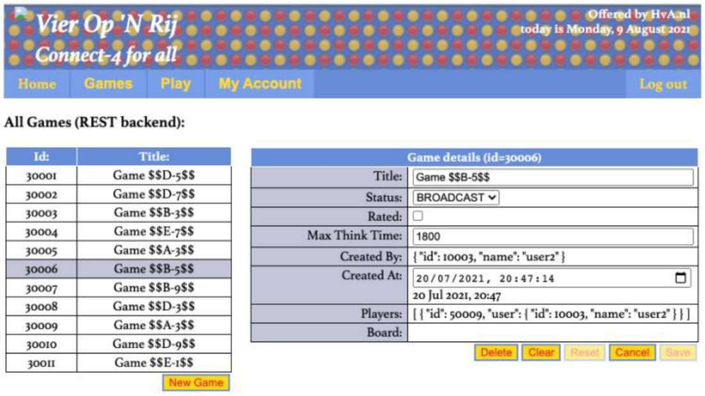
```
@Override
public void addCorsMappings(CorsRegistry registry) {
    registry.addMapping("/**")
            .allowedOriginPatterns("http://localhost:*", getHostIPAddressPattern())
```

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

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

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

(At this time you also may be able to view player info, if already instantiated by your backend)



All Games (REST backend):

Id:	Title:
30001	Game $$D-5$$
30002	Game $$D-7$$
30003	Game $$B-3$$
30004	Game $$E-7$$
30005	Game $$A-3$$
30006	Game $$B-5$$
30007	Game $$B-9$$
30008	Game $$D-3$$
30009	Game $$A-3$$
30010	Game $$D-9$$
30011	Game $$E-1$$

Game details (id=30006)	
Title:	Game $$B-5$$
Status:	BROADCAST
Rated:	□
Max Think Time:	1800
Created By:	{ "id": 10003, "name": "user2" }
Created At:	20/07/2021, 20:47:14 20 Jul 2021, 20:47
Players:	[{ "id": 50009, "user": { "id": 10003, "name": "user2" } }]
Board:	

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

Some functionality involves special consideration:
1. If a New Game is added, the frontend shall generate a new, empty game with id=0, and first save it to the backend via gamesService.asyncSave(newGame). The backend shall generate and set a new unique id and return the updated game in the response. The frontend should have 'awaited' the response and then push the newly saved game into the list of games in GamesOverview37 and then select it for display and editing.

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

3. GamesDetail37 no longer takes the selectedGame as a property from GamesOverview37 but uses the id-parameter from the route to retrieve itself an up-to-date version of the game from the backend. (Use gamesService.asyncFindById()). GamesDetail37 shall also directly save and delete games at the backend without seeking intermediary involvement of GamesOverview37.

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

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

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

3.7.2 [BONUS] A generic caching REST Adaptor service

In the previous assignment you have developed a GamesAdaptor that provides your components with an async/await API for retrieving and updating games at the backend.

That implementation misses two objectives:

I. If there are many entity classes involved in your application, you wish to avoid the code duplication across very similar adaptor implementations for each entity class.

II. If two components (master and detail) inject the same instance of the (shared) service, then we would like Vue's change detection mechanism to automatically process the updates to the data by the other component without our need to emit events about that.

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

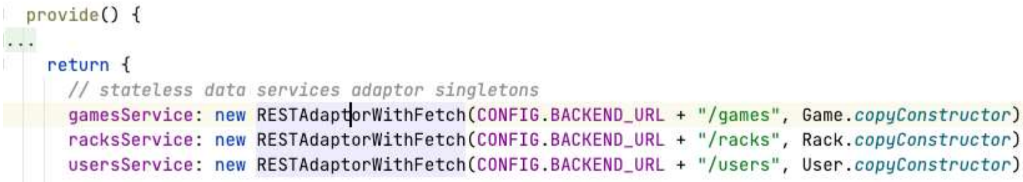
1) Every entity is provided by a different resource endpoint url at the backend.
2) Every entity comes with a specific copyConstructor that can be used to create proper object instances from a json representation.
3) Every instance of an entity should have an id. (This id also features as a parameter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

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

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

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

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

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



```
provide() {
...
  return {
    // stateless data services adaptor singletons
    gamesService: new RESTAdaptorWithFetch(CONFIG.BACKEND_URL + "/games", Game.copyConstructor),
    racksService: new RESTAdaptorWithFetch(CONFIG.BACKEND_URL + "/racks", Rack.copyConstructor),
    usersService: new RESTAdaptorWithFetch(CONFIG.BACKEND_URL + "/users", User.copyConstructor),
```

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

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

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

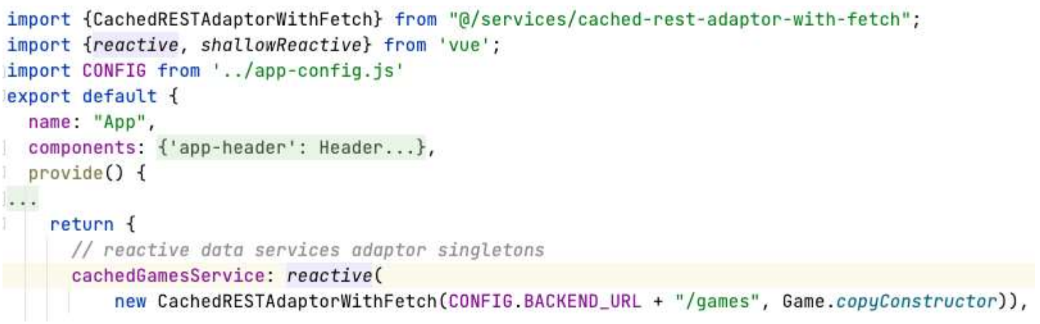
- Each entity that is retrieved by asyncFindById is also updated in the local cache.
- Each entity that is saved by asyncSave is also updated in the local cache.
- Each entity that is removed by asyncDeleteById is also removed from the local cache.

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

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

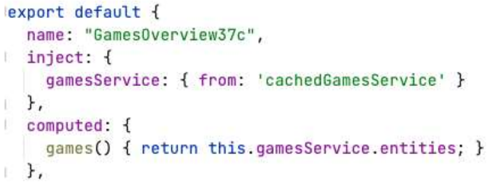
```
export class CachedRESTAdaptorWithFetch /* <E> */
         extends RESTAdaptorWithFetch /* <E> */  {
    entities;         // the cache of the results of all CRUD operations

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

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



```
import {CachedRESTAdaptorWithFetch} from "@/services/cached-rest-adaptor-with-fetch";
import {reactive, shallowReactive} from 'vue';
import CONFIG from '../app-config.js'
export default {
  name: "App",
  components: {'app-header': Header...},
  provide() {
...
    return {
      // reactive data services adaptor singletons
      cachedGamesService: reactive(
          new CachedRESTAdaptorWithFetch(CONFIG.BACKEND_URL + "/games", Game.copyConstructor)),
```

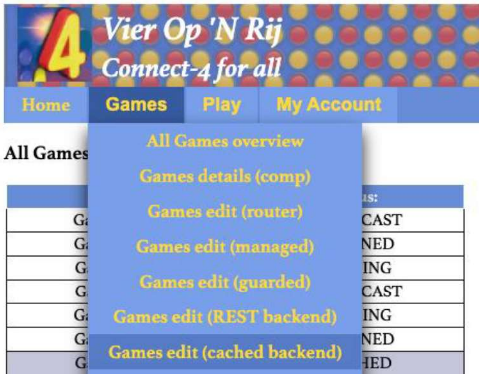
In the overview we inject the cachedGamesService and use a computed property to replace the local games array by a reference to the entities cache of the service:



```
export default {
  name: "GamesOverview37c",
  inject: {
    gamesService: { from: 'cachedGamesService' }
  },
  computed: {
    games() { return this.gamesService.entities; }
  },
```

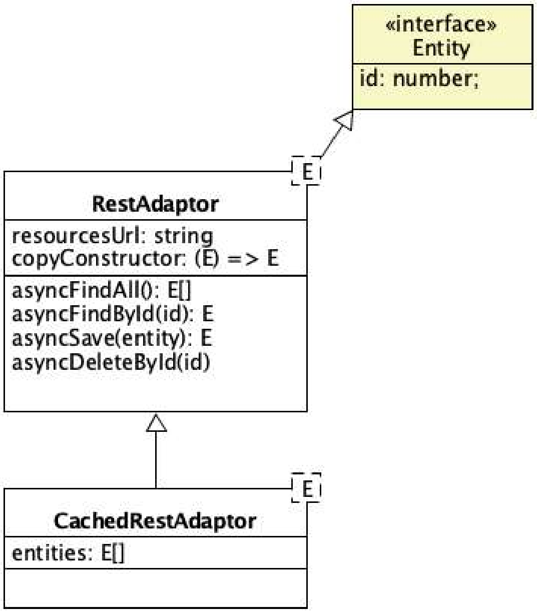
Notice the special inject syntax that allows us to rename the injected service, such that we can continue to reuse the existing code of the UI component.

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



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

Test your application.
1. Verify that status updates to games also appear in the selection list.
2. Verify that deletions are removed from the list.
3. Verify that new games are added to the list



4 Second term assignments: JPA, Authentication and WebSockets

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

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

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

4.1 JPA and ORM configuration

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

By the end of this assignment, you will have implemented the Game and Player classes including its one-to-many relationship. Your REST API can add Players to games. It will produce error responses on Players for games that are not open for joining.

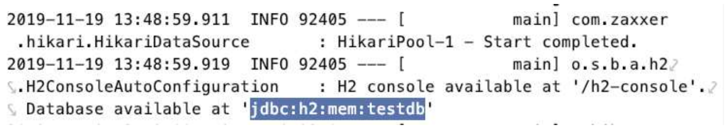
Relevant introductions into the topics you find in O'Reilly-3 chapters 3 and 5.

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

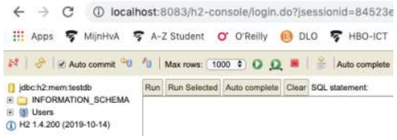
4.1.1 Configure a JPA Repository

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

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



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



localhost:8083/h2-console/login.do?jsessionid=84523e

jdbc:h2:mem:testdb
INFORMATION_SCHEMA
Users
H2 1.4.200 (2019-10-14)

B. Upgrade your Game 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.

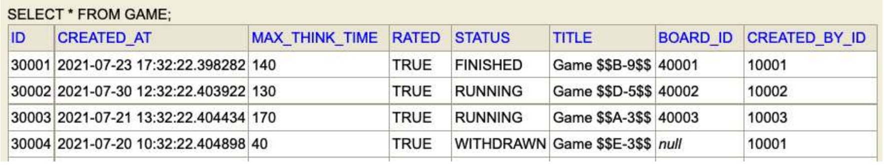
Create a new implementation class GamesRepositoryJpa for your GamesRepository interface. This new class should get injected an entity manager that provides you with access to the persistence context of the ORM. Use this entity manager to first implement the save method of your new repository. (Other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

Configure transactional mode for all methods of GamesRepositoryJpa.

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

Test the creation of a game with postman doing a post at localhost:8087/games. Inspect the associated SQL statements in the Spring Boot log and use the h2-console to verify whether the game ended up in H2.

```
        Hibernate: call next value for game_ids
        Hibernate: insert into game (rack_id, created_at, created_by_id, max_think_time,
rated,
                 status, title, id) values (?, ?, ?, ?, ?, ?, ?, ?, ?)
```



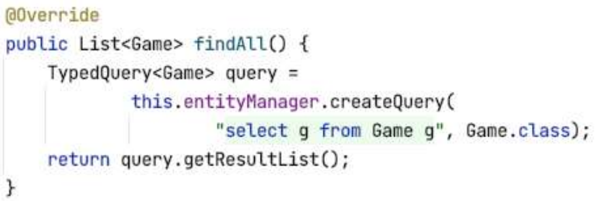
SELECT * FROM GAME;

ID	CREATED_AT	MAX_THINK_TIME	RATED	STATUS	TITLE	BOARD_ID	CREATED_BY_ID
30001	2021-07-23 17:32:22.398282	140	TRUE	FINISHED	Game $$B-9$$	40001	10001
30002	2021-07-30 12:32:22.403922	130	TRUE	RUNNING	Game $$D-5$$	40002	10002
30003	2021-07-21 13:32:22.404434	170	TRUE	RUNNING	Game $$A-3$$	40003	10003
30004	2021-07-20 10:32:22.404898	40	TRUE	WITHDRAWN	Game $$E-3$$	*null*	10001

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

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

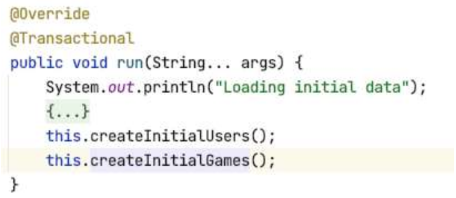
Test your new repository with postman.



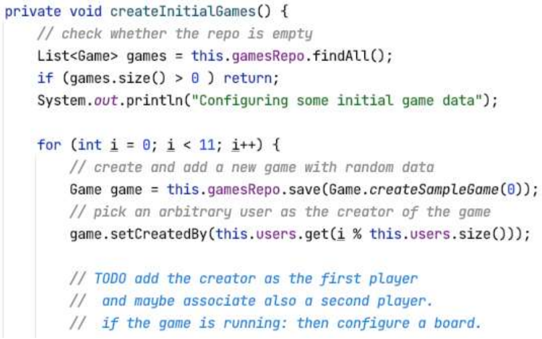
```
@Override
public List<Game> findAll() {
    TypedQuery<Game> query =
            this.entityManager.createQuery(
                    "select g from Game g", Game.class);
    return query.getResultList();
}
```

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

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



```
@Override
@Transactional
public void run(String... args) {
    System.out.println("Loading initial data");
    {...}
    this.createInitialUsers();
    this.createInitialGames();
}
```



```
private void createInitialGames() {
    // check whether the repo is empty
    List<Game> games = this.gamesRepo.findAll();
    if (games.size() > 0 ) return;
    System.out.println("Configuring some initial game data");

    for (int i = 0; i < 11; i++) {
        // create and add a new game with random data
        Game game = this.gamesRepo.save(Game.createSampleGame(0));
        // pick an arbitrary user as the creator of the game
        game.setCreatedBy(this.users.get(i % this.users.size()));

        // TODO add the creator as the first player
        //  and maybe associate also a second player.
        //  if the game is running: then configure a board.
```

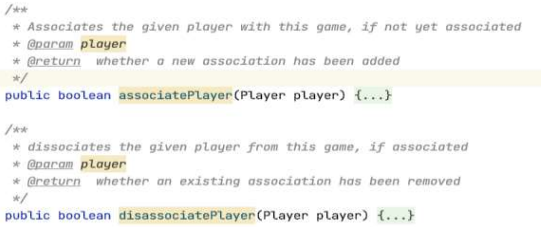
This approach is preferred above the use of SQL scripts in the H2 backend because the details of the generated H2 SQL schema will change as you progress your Java entities.

This CommandLineRunner initialisation will also work with your Mock repository implementation.
Make sure to configure transactional mode on the command line runner.

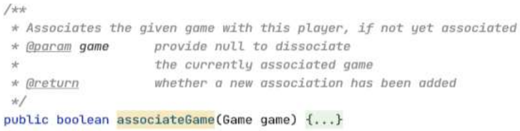
4.1.2 Configure a one-to-many relationship

A. Now is the time to introduce some more entities. Make a new entity class ‘models/User’ identified by an ‘id’(long) attribute. Also record the ‘name’(String) attribute of a User. (Later we also require ‘email’(String), ‘role’(String) and ‘hashedPassword’(String) for authentication of login and authorisation).

Also introduce a new entity class ‘models/Player’ identified by an ‘id’(long) attribute and also tracking a ‘color’(String) attribute which represents the color of the pieces that the player will play with. Player represents the (virtual) seats at a Game instance which will be operated by a real User:
Each Player is associated with one Game and a Game can be associated with many Players (mostly two in our case).
Each Player is associated with one User and a User may play multiple Player roles (each on a different Game).
Define the corresponding association attributes in the Game, Player and User classes and provide methods to change the associations from either side. (Avoid endless cyclic recursion when automatically maintaining consistency in bi-directional navigability!):



```
/**
 * Associates the given player with this game, if not yet associated
 * @param player
 * @return  whether a new association has been added
 */
public boolean associatePlayer(Player player) {...}

/**
 * dissociates the given player from this game, if associated
 * @param player
 * @return  whether an existing association has been removed
 */
public boolean disassociatePlayer(Player player) {...}
```

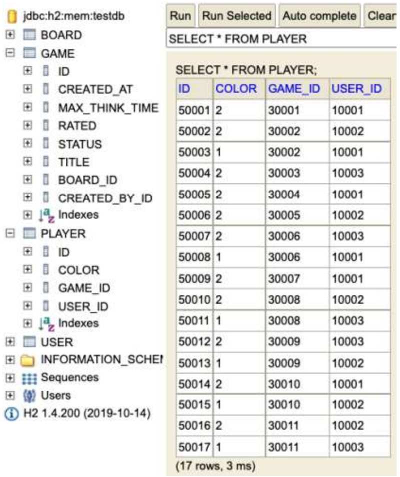


```
/**
 * Associates the given game with this player, if not yet associated
 * @param game      provide null to dissociate
 *                  the currently associated game
 * @return          whether a new association has been added
 */
public boolean associateGame(Game game) {...}
```

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

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

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



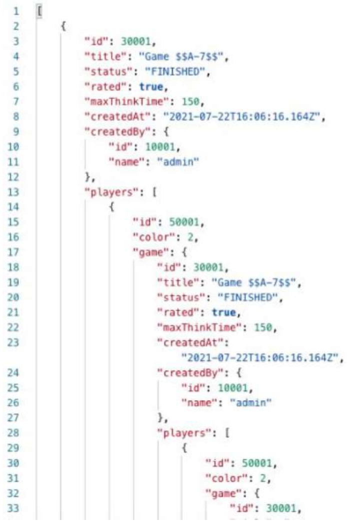
```
SELECT * FROM PLAYER
```

SELECT * FROM PLAYER;

ID	COLOR	GAME_ID	USER_ID
50001	2	30001	10001
50002	2	30002	10002
50003	1	30002	10001
50004	2	30003	10003
50005	2	30004	10001
50006	2	30005	10002
50007	2	30006	10003
50008	1	30006	10001
50009	2	30007	10001
50010	2	30008	10002
50011	1	30008	10003
50012	2	30009	10003
50013	1	30009	10002
50014	2	30010	10001
50015	1	30010	10002
50016	2	30011	10002
50017	1	30011	10003

(17 rows, 3 ms)

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



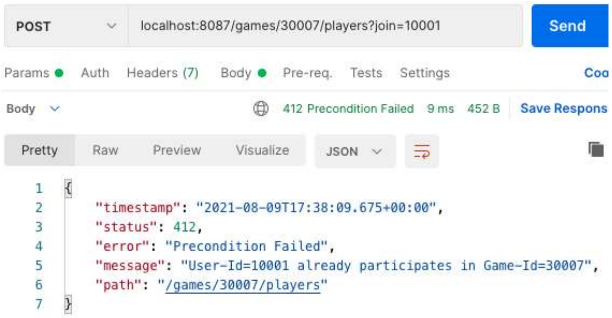
```
[
    {
        "id": 30001,
        "title": "Game $$A-7$$",
        "status": "FINISHED",
        "rated": true,
        "maxThinkTime": 150,
        "createdAt": "2021-07-22T16:06:16.164Z",
        "createdBy": {
            "id": 10001,
            "name": "admin"
        },
        "players": [
            {
                "id": 50001,
                "color": 2,
                "game": {
                    "id": 30001,
                    "title": "Game $$A-7$$",
                    "status": "FINISHED",
                    "rated": true,
                    "maxThinkTime": 150,
                    "createdAt":
                        "2021-07-22T16:06:16.164Z",
                    "createdBy": {
                        "id": 10001,
                        "name": "admin"
                    },
                    "players": [
                        {
                            "id": 50001,
                            "color": 2,
                            "game": {
                                "id": 30001,
```

D. Re-test the REST API at localhost:8087/games with postman:
You may find a response like here with endless recursion in the JSON structure. For now, investigate the use of @JsonManagedReference and @JsonBackReference to fix that.
With (optional) bonus assignment 4.2.2 you can practice a better solution with custom Json serializers.

E. Implement a POST mapping on the /games/{gameId}/players REST endpoint. This mapping should add a new Player to the game.

Instead of creating a Player instance client side it is more convenient to implement a 'join' request parameter for the post which passes the id of the User that wants to join the game. E.g. POST '/games/30001/players?join=10001' would add the User with id=10001 to the game with id=30001.

A “PreCondition Failed” error response should be returned if:
- The provided game Id does not match an existing game.
- The game is not open for joining (status is not broadcast or the maximum number of players has been associated already.)
- The provided user Id does not match an existing User
- The specified User already participates in the Game.



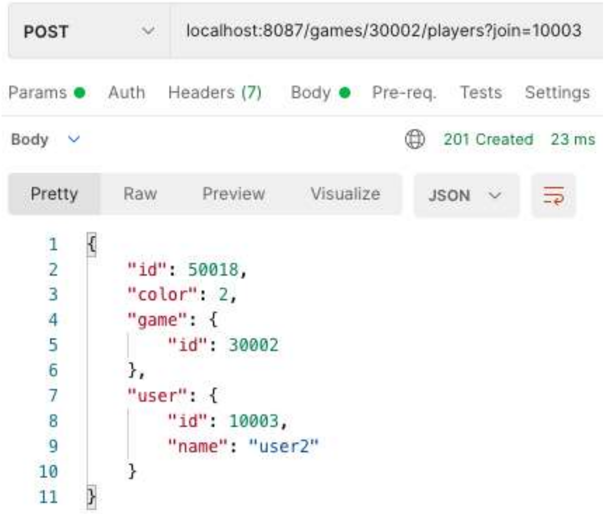
POST localhost:8087/games/30007/players?join=10001

412 Precondition Failed 9 ms 452 B

```
{
    "timestamp": "2021-08-09T17:38:09.675+00:00",
    "status": 412,
    "error": "Precondition Failed",
    "message": "User-Id=10001 already participates in Game-Id=30007",
    "path": "/games/30007/players"
}
```

Otherwise, a new Player instance shall be created in the backend, associated with the Game and the User and returned in the response.

Test your new endpoint with postman: (With @JSonBackReference you many not be able to show the game and the user in the Player anymore)



POST localhost:8087/games/30002/players?join=10003

201 Created 23 ms

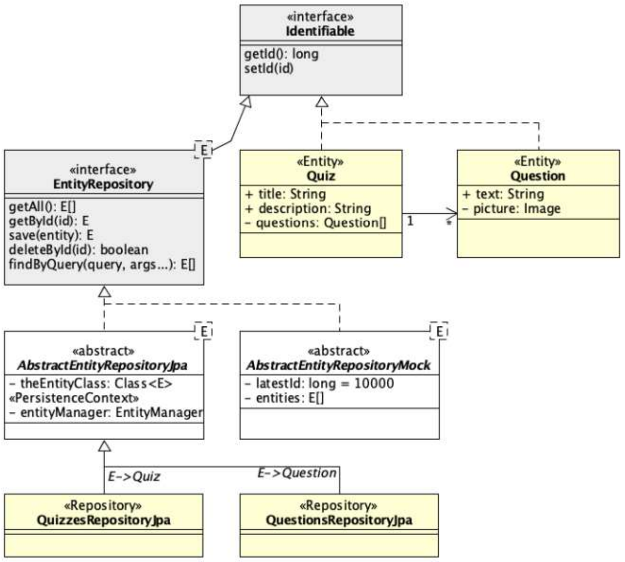
```
{
    "id": 50018,
    "color": 2,
    "game": {
        "id": 30002
    },
    "user": {
        "id": 10003,
        "name": "user2"
    }
}
```

Also verify the transactional mode of JPA which will ensure that any saved Player will be rolled back if an error is raised thereafter.

4.1.3 [BONUS] Generalized Repository

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

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



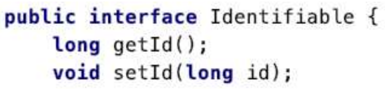
The SimpleJpaRepository<E, ID> class uses the JPA metadata of the annotations to figure out what are the identifying attributes of an entity. That is too complex for the scope of this course.

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

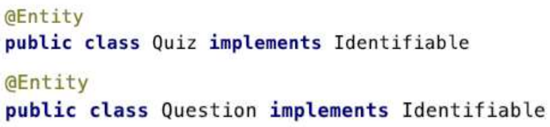
In the class diagram here, you find a specification of a simplified approach of implementing an EntityRepository<E> interface in a generic way, but assuming that all your entities are identified by the 'long' data type.
(Replace in this diagram and in the code snippets below the Quiz entity by your Game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

This approach can be realised as follows:

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

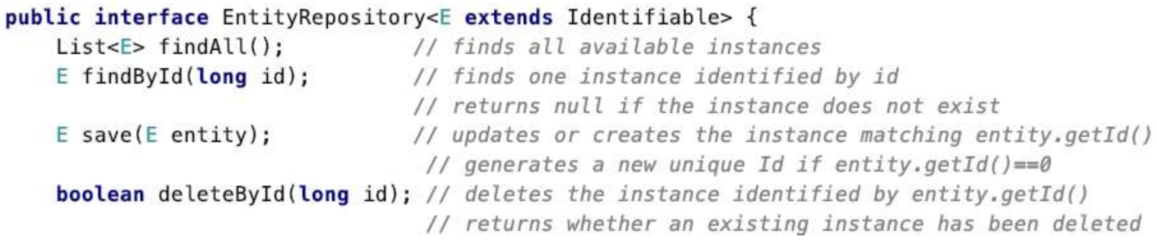


```
public interface Identifiable {
    long getId();
    void setId(long id);
```



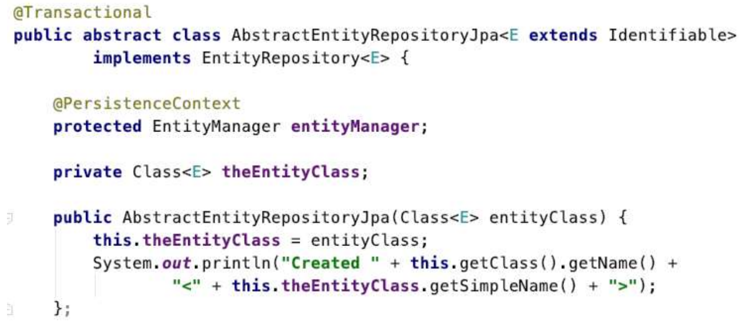
```
@Entity
public class Quiz implements Identifiable

@Entity
public class Question implements Identifiable
```

B. Specify a generic EntityRepository interface:



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

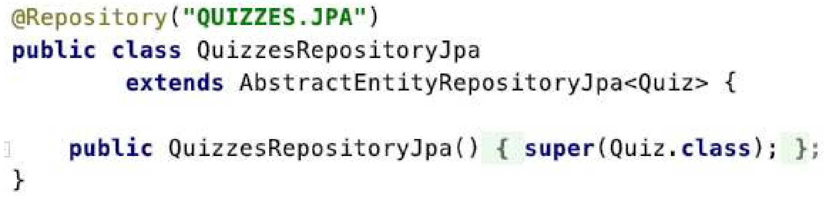
C. Implement once the abstract class AbstractEntityRepositoryJpa with all the repository functionality in a generic way:



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

    @PersistenceContext
    protected EntityManager entityManager;

    private Class<E> theEntityClass;

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

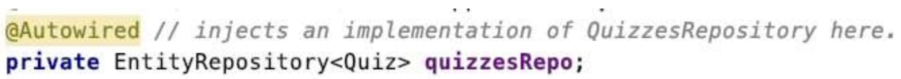
You will need ‘theEntityClass’ and its simple name to provide generic implementations of entity manager operations and JPQL queries.

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



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

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

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



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

4.2 JPQL queries and custom JSON serialization

In this assignment you will explore JPQL queries. You will extend the get-mapping of the /games endpoint to optionally accept query parameters '?title=XXX', '?status=XXX' and '?player=NNNNN' and then filter the list of games being returned to meet the specified criteria.

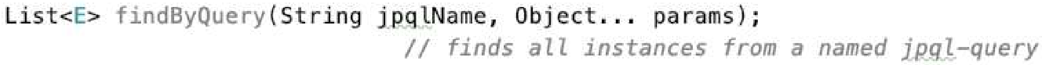
You will pass the filters as part of a JPQL query to the persistence context, such that only the games that meet the criteria will retrieved from the database.

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

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

4.2.1 JPQL queries

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

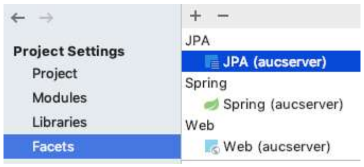


```
List<E> findByQuery(String jpqlName, Object... params);
                                            // finds all instances from a named jpql-query
```

(Here we assume you use the generic repository interface)

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

B. Design four (named) JPQL queries:
"Game_find_by_title": finds all games that have the given string in the title
"Game_find_by_status": finds all games of a given status
"Game_find_by_player": finds all games in which a given user is playing
"Game_find_by_status_and_player":
combines both the status and player criterium
Use the @NamedQuery annotation to specify these named JPQL queries within your Game.java entity class.
Use an ordinal(positional) query parameters (?1, ?2, etc.) as a place-holder for the parameter values to be provided.



(If you are troubled by a mal-functioning inspection module of IntelliJ on your JPQL language, verify whether you have a default JPA facet configuration in your project structure)

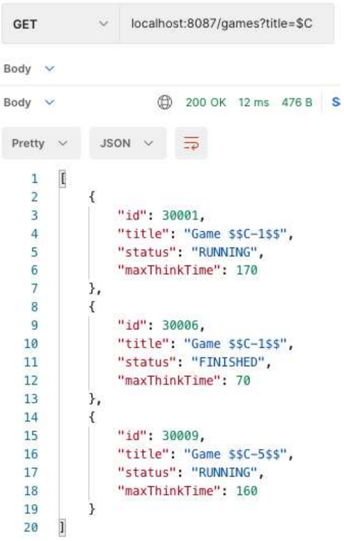
C. Extend your GetMapping on the “/games” end-point to optionally accept the ‘?title=XXX’, ‘?status=XXX’, ‘?player=NNNNN’ or ‘?status=XXX&player=NNNNN’ request parameters.

If no request parameter is provided, the existing functionality of returning all games should be retained.

If the ‘?status=XXX’ parameter is provided, with a status string value that does not match the Game.Status enumeration, an appropriate error response should be returned.
If an unsupported combination of request parameters is provided, a “Bad Request” error response should be returned with a useful error message.

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

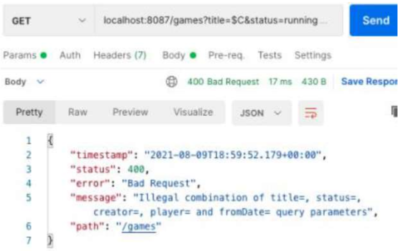
Test the behaviour of your end-point with postman:



GET localhost:8087/games?title=$C

200 OK 12 ms 476 B

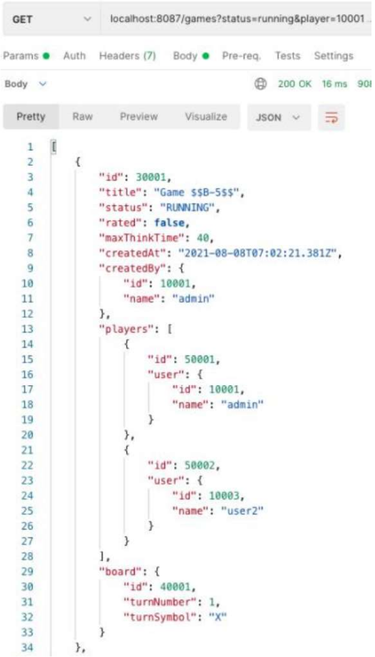
```
[
    {
        "id": 30001,
        "title": "Game $$C-1$$",
        "status": "RUNNING",
        "maxThinkTime": 170
    },
    {
        "id": 30006,
        "title": "Game $$C-1$$",
        "status": "FINISHED",
        "maxThinkTime": 70
    },
    {
        "id": 30009,
        "title": "Game $$C-5$$",
        "status": "RUNNING",
        "maxThinkTime": 160
    }
]
```



GET localhost:8087/games?title=$C&status=running ...

400 Bad Request 17 ms 430 B

```
{
    "timestamp": "2021-08-09T18:59:52.179+00:00",
    "status": 400,
    "error": "Bad Request",
    "message": "Illegal combination of title=, status=,
        creator=, player= and fromDate= query parameters",
    "path": "/games"
}
```



GET localhost:8087/games?status=running&player=10001 ...

200 OK 16 ms 90

```
[
    {
        "id": 30001,
        "title": "Game $$B-5$$",
        "status": "RUNNING",
        "rated": false,
        "maxThinkTime": 40,
        "createdAt": "2021-08-08T07:02:21.381Z",
        "createdBy": {
            "id": 10001,
            "name": "admin"
        },
        "players": [
            {
                "id": 50001,
                "user": {
                    "id": 10001,
                    "name": "admin"
                }
            },
            {
                "id": 50002,
                "user": {
                    "id": 10003,
                    "name": "user2"
                }
            }
        ],
        "board": {
            "id": 40001,
            "turnNumber": 1,
            "turnSymbol": "X"
        }
    },
```

4.2.2 [BONUS] Custom JSON Serializers

Bi-directional navigation, and nested entities easily give rise to endless Json structures.

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

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

a) if you query a Game, you want full information about the game but probably only shallow information about its Players.

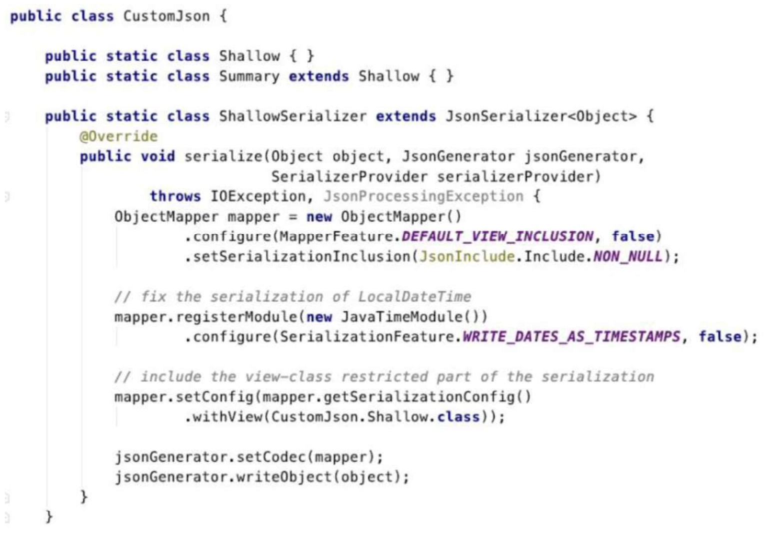
b) If you query a Player, you want full information about the Player, but only shallow information about the Game.

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.

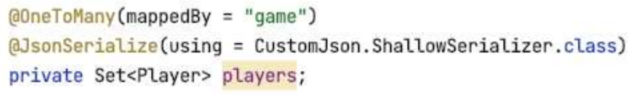
At https://stackoverflow.com/questions/23260464/how-to-serialize-using-jsonview-with-nested-objects#23264755 you find a nice article about combining @JsonView classes with custom Json serializers that may solve all your challenges with a comprehensive, single generic approach:

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 Game.players:



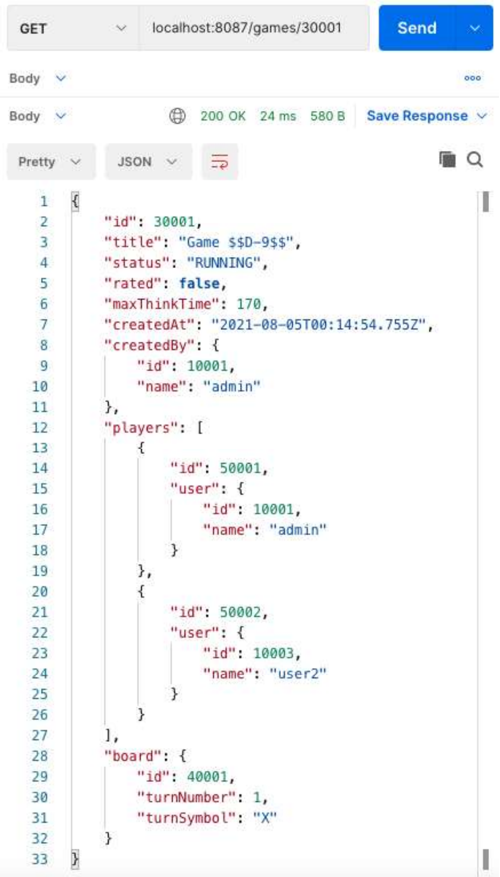
```
@OneToMany(mappedBy = "game")
@JsonSerialize(using = CustomJson.ShallowSerializer.class)
private Set<Player> players;
```

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

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 Game and Player model classes.
Apply the Summary view class to the localhost:8087/games endpoint.
Implement unrestricted endpoints at localhost:8087/games/{gameId} and localhost:8087/games/{gameId}/Players

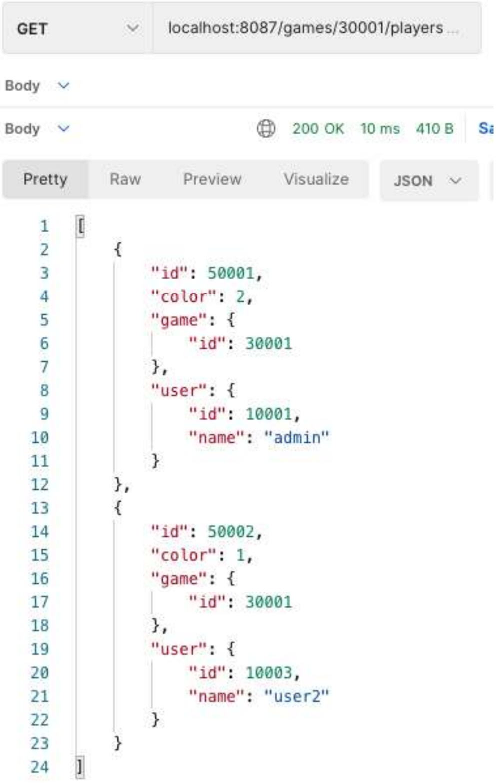
Test your endpoints with postman:



localhost:8087/games/30001

Body 200 OK 24 ms 580 B

```
{
    "id": 30001,
    "title": "Game $$D-9$$",
    "status": "RUNNING",
    "rated": false,
    "maxThinkTime": 170,
    "createdAt": "2021-08-05T00:14:54.755Z",
    "createdBy": {
        "id": 10001,
        "name": "admin"
    },
    "players": [
        {
            "id": 50001,
            "user": {
                "id": 10001,
                "name": "admin"
            }
        },
        {
            "id": 50002,
            "user": {
                "id": 10003,
                "name": "user2"
            }
        }
    ],
    "board": {
        "id": 40001,
        "turnNumber": 1,
        "turnSymbol": "X"
    }
}
```



localhost:8087/games/30001/players ...

Body 200 OK 10 ms 410 B

```
[
    {
        "id": 50001,
        "color": 2,
        "game": {
            "id": 30001
        },
        "user": {
            "id": 10001,
            "name": "admin"
        }
    },
    {
        "id": 50002,
        "color": 1,
        "game": {
            "id": 30001
        },
        "user": {
            "id": 10003,
            "name": "user2"
        }
    }
]
```

4.3 Backend security configuration, JSON Web Tokens (JWT)

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

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

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

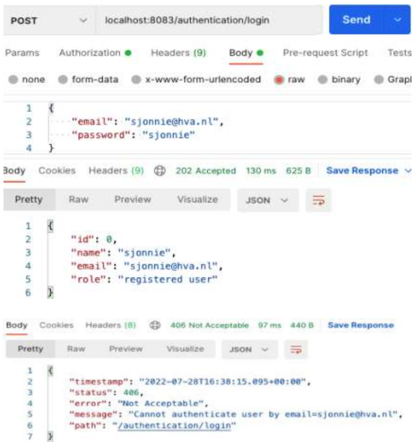
By the end of this assignment you will have further explored annotations @RequestBody, @RequestAttribute, @Value and classes ObjectNode, Jwts, Jws<Claims>, SignatureAlgorithm

4.3.1 The /authentication controller.

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

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

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



POST localhost:8083/authentication/login

Send
Params
Authorization ●
Headers (9)
Body ●
Pre-request Script
Tests
○ none ○ form-data ○ x-www-form-urlencoded ● raw ○ binary ○ Grapl

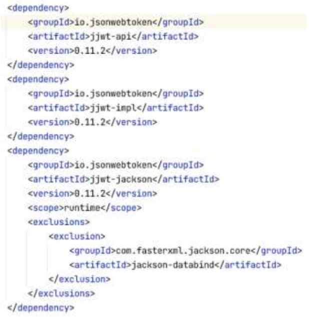
```
{
    "email": "sjonnie@hva.nl",
    "password": "sjonnie"
}
```

```
{
    "id": 0,
    "name": "sjonnie",
    "email": "sjonnie@hva.nl",
    "role": "registered user"
}
```

```
{
    "timestamp": "2022-07-28T16:38:15.095+00:00",
    "status": 406,
    "error": "Not Acceptable",
    "message": "Cannot authenticate user by email=sjonnie@hva.nl",
    "path": "/authentication/login"
}
```

Test your endpoint with postman.

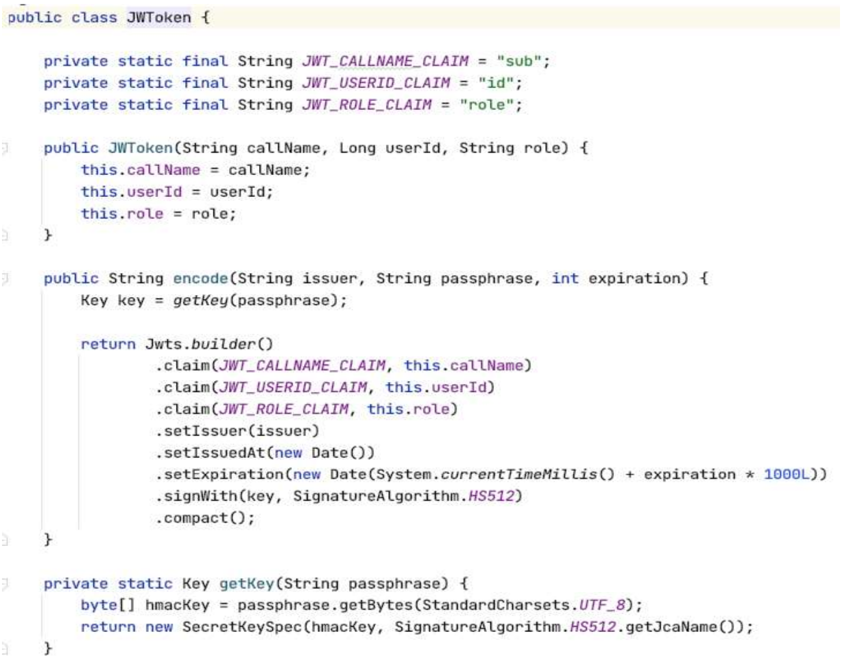
B. After successful login we want to provide a token to the client. Include the Jackson JWT dependencies into your pom.xml.



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

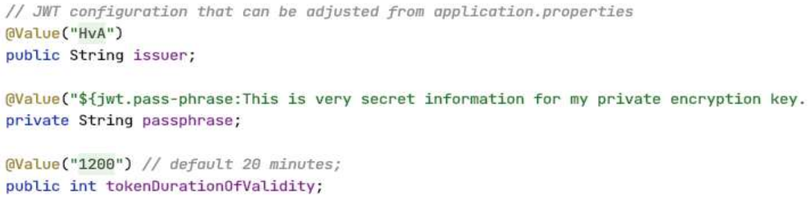
Create a utility class JWToken, which will store all attributes associated with the authentication and authorisation of the user (the ‘payload’) and implement the functionality to encrypt and decrypt this information into token strings.

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



```java
public class JWToken {

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

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

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

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

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

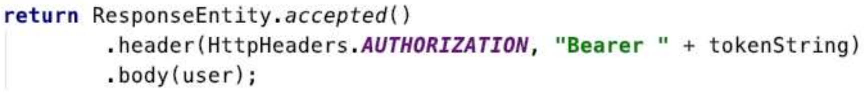
The passphrase is the private key to be used for encryption and decryption. You can configure passphrase, issuer and expiration times in the application.properties file and then inject them into your APIConfig bean using the @Value annotation. You may want to amend the passphrase at run-time such that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



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

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

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

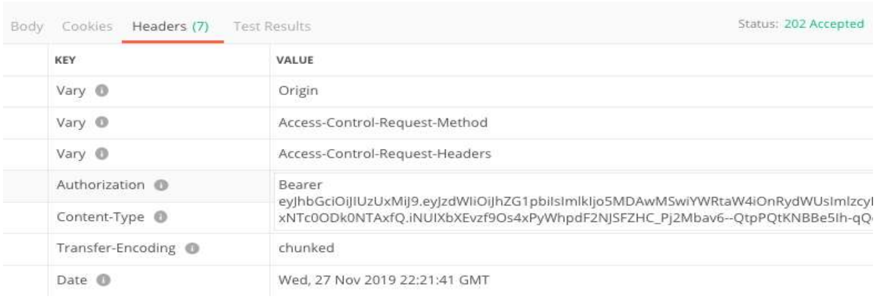
At https://jwt.io/ you can verify your token strings after you have created them.

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



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

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



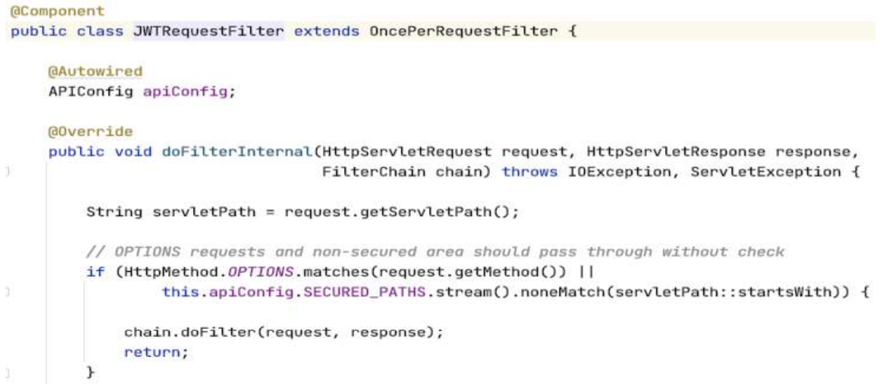
Status: 202 Accepted

KEY	VALUE
Vary	Origin
Vary	Access-Control-Request-Method
Vary	Access-Control-Request-Headers
Authorization	Bearer eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJhZG1pbiIsImlkIjo5MDAwMSwiYWRtaW4iOnRydWUsImlzcyI xNTc0ODk0NTAxfQ.iNUIXbXEvzf9Os4xPyWhpdF2NJSFZHC_Pj2Mbav6--QtpPQtKNBBe5Ih-qQ
Content-Type	
Transfer-Encoding	chunked
Date	Wed, 27 Nov 2019 22:21:41 GMT

4.3.2 *The request filter.*

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

For that you implement a request filter component:



```
@Component
public class JWTRequestFilter extends OncePerRequestFilter {

    @Autowired
    APIConfig apiConfig;

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

        String servletPath = request.getServletPath();

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

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

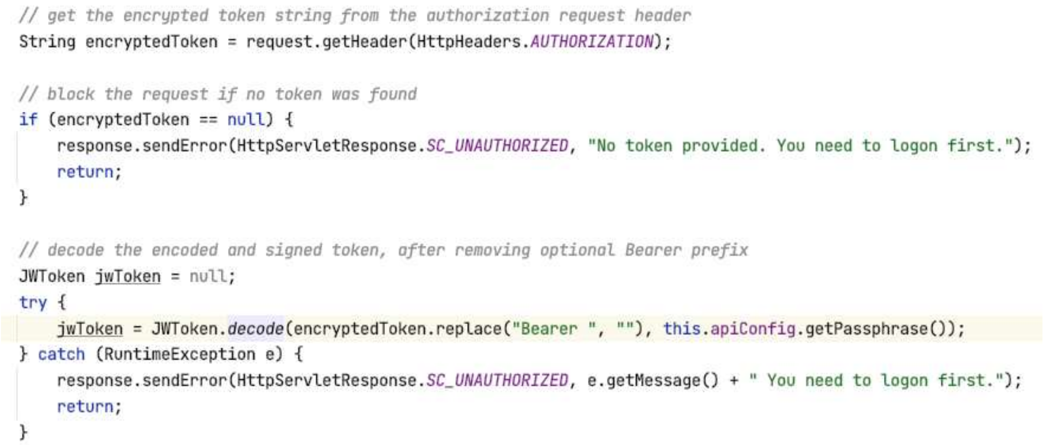
Because your filter class is a Spring Boot Component bean, it will be autoconfigured into the filter chain of the Spring Boot http request dispatcher, and hit every incoming http request.

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

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

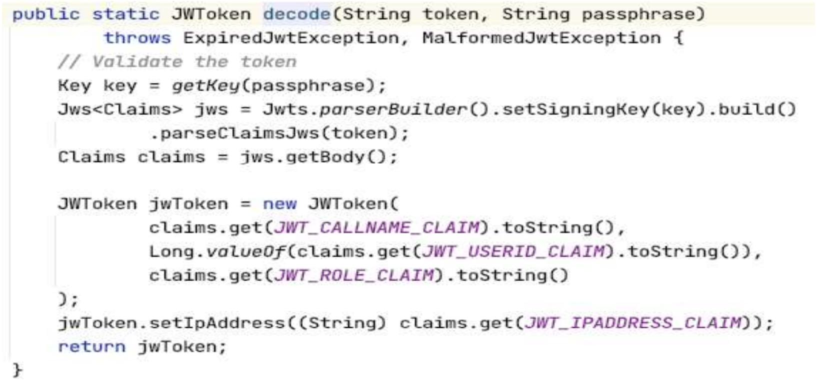
Test with postman whether the filter is activated on your SECURED_PATHS and not affecting the other paths.

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



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

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

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

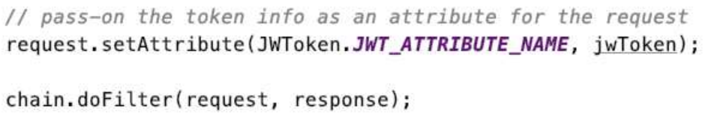
Again, the magic is in the use of the Jackson libraries, decoding the token string:



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

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

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

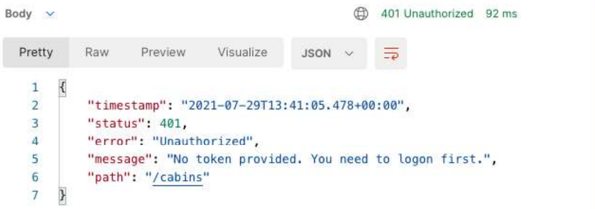
If all has gone well, we add the decoded token information into an attribute of the request and allow the request to progress further down the chain:



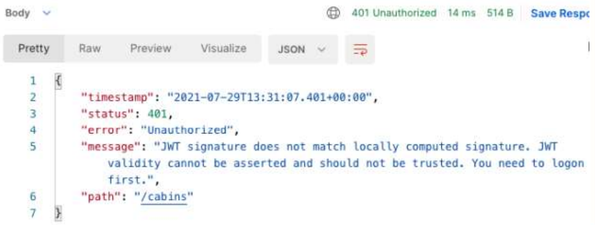
```
// pass-on the token info as an attribute for the request
request.setAttribute(JWToken.JWT_ATTRIBUTE_NAME, jwToken);

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

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

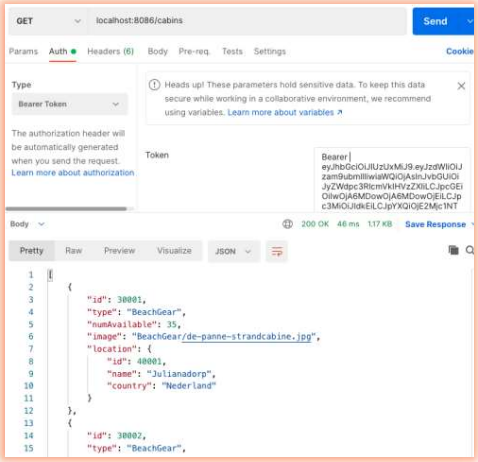
Test your set-up with postman:
First try a get request without an Authorization Header.
Then try again with a correct header in the request.
Then try with a corrupt token (e.g. append XXX at the end of the token…)



```
{
    "timestamp": "2021-07-29T13:41:05.478+00:00",
    "status": 401,
    "error": "Unauthorized",
    "message": "No token provided. You need to logon first.",
    "path": "/cabins"
}
```



```
{
    "timestamp": "2021-07-29T13:31:07.401+00:00",
    "status": 401,
    "error": "Unauthorized",
    "message": "JWT signature does not match locally computed signature. JWT
        validity cannot be asserted and should not be trusted. You need to logon
        first.",
    "path": "/cabins"
}
```



GET localhost:8086/cabins

Type: Bearer Token

The authorization header will be automatically generated when you send the request.

Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment, we recommend using variables.

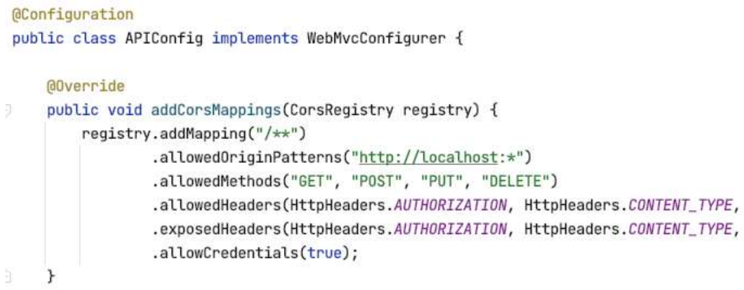
Token: Bearer eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJzam9ubmlliwiaWQiOjAsInJvbGUiOiJyZWdpc3RlcmVkIHVzZXIiLCJpcGEiOilwOjA6MDowOjA6MDowOjEiLCJpc3MiOiJIdkEiLCJpYXQiOjE2Mjc1NT

200 OK 46 ms 1.17 KB

```
[
    {
        "id": 30001,
        "type": "BeachGear",
        "numAvailable": 35,
        "image": "BeachGear/de-panne-strandcabine.jpg",
        "location": {
            "id": 40001,
            "name": "Julianadorp",
            "country": "Nederland"
        }
    },
    {
        "id": 30002,
        "type": "BeachGear",
```

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

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



```
@Configuration
public class APIConfig implements WebMvcConfigurer {

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