SpringBoot & JHipster development: masterclass

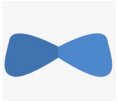
JHipster is a tech-stack generator that creates full applications stacks using modern components. In this session, we will get started editing the front end of our JHipster application.

You are encouraged to follow along and try the instructions with your own repository. This session will use the terminal, gitlab and IntelliJ Ultimate however the principles are the same as if using an different IDE. n.b. use the codepad links e.g. *try it on codepad!* to copy and paste code from as PDF can be a pain to get code listings from.

n.b. these changes have only been tested locally, other changes or further development may be need. e.g. tests the JUnit or .spec.ts files may fail.

This session is not exhaustive. Similar to git/CI, many of the commands are rarely used and it is completely fine to consult the documentation for Angular, SpringBoot, JDL or JHipster when needed.

Terminology



JHipster - an open-source development platform that generates high-quality Spring Boot web applications with integrated tools, libraries, and frameworks. JHipspter applies best practices of backend and frontend for rapid prototyping and efficient development.

JDL - JHipster Domain Language is a domain-specific language similar to UML, used for defining entities, relationships, and their attributes. JDL models the application domain and can be used by JHipster to generate code, configurations, and database schemas.

Springboot - an open-source Java-based framework for development of stand-alone, production-grade Spring web backend applications. It uses an opinionated, convention-over-configuration approach for rapid application development and deployment. A Springboot backend is created for us by JHipster, in Java.

Angular - Angular is a TypeScript-based open-source front end application framework, to streamline the development of web UIs by providing a comprehensive structure for building dynamic, modular and responsive applications. A basic Angular front-end is created for us by JHipster.

REST (Representational State Transfer) is an architectural style for designing networked API, emphasizing stateless communication, standard HTTP methods (GET, POST, PUT, DELETE), and resource-based interactions, enabling interoperability and scalability in web applications.

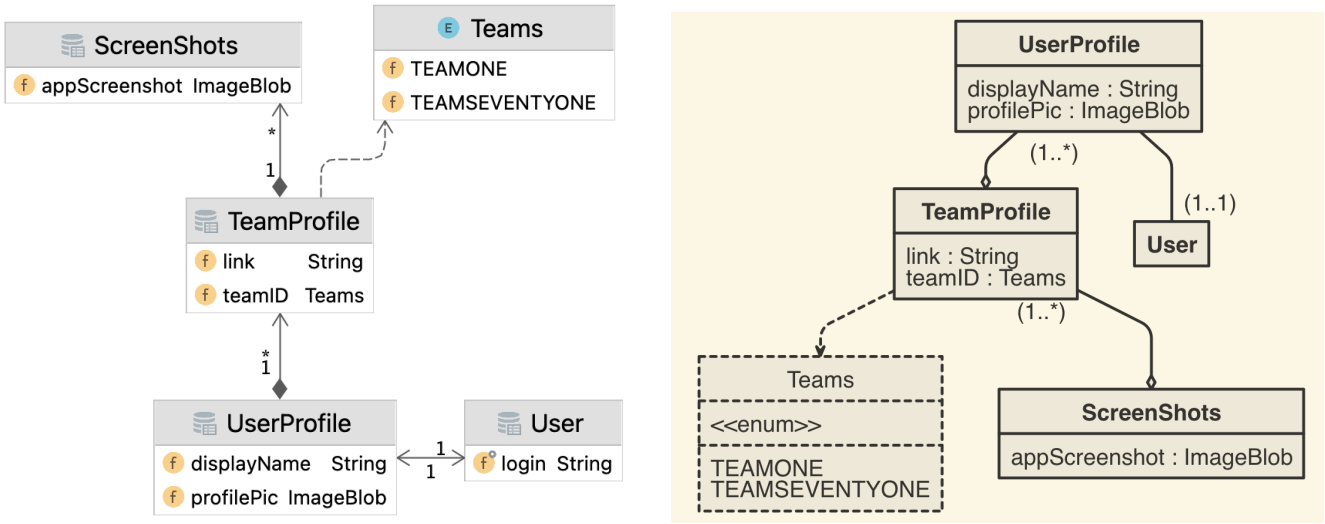


Figure 1: UML view of JDL created using JDL IntelliJ plugin (left) and JDL Studio (right)

Reminder: generating app from JDL

The JDL we developed last week is below

```
entity UserProfile{
    displayName String,
    profilePic ImageBlob
}

entity TeamProfile{
    link String,
    teamID Teams
}

enum Teams{
    TEAMONE(teamone),
    TEAMSEVENTYONE(teamseventyone)
}

entity ScreenShots{
    appScreenshot ImageBlob
}

relationship ManyToMany{
    TeamProfile to ScreenShots
}

relationship ManyToOne{
    UserProfile to TeamProfile
}

relationship OneToOne{
    UserProfile to User
}
```

try it on codepad!

Spring Boot - Back End

Change an API URI

Backend Changes

Our aim is to change the URI from `/api/team-profiles` to the more readable and REST *conventional* `/api/teams`.

First, in:

`src/main/java/team/bham/web/rest/TeamProfileResource.java`,

change instances of `@____Mapping("/team-profiles")` to `@____Mapping("/teams")` e.g.:

```
@GetMapping("/team-profiles")
```

```
@PostMapping("/team-profiles")
```

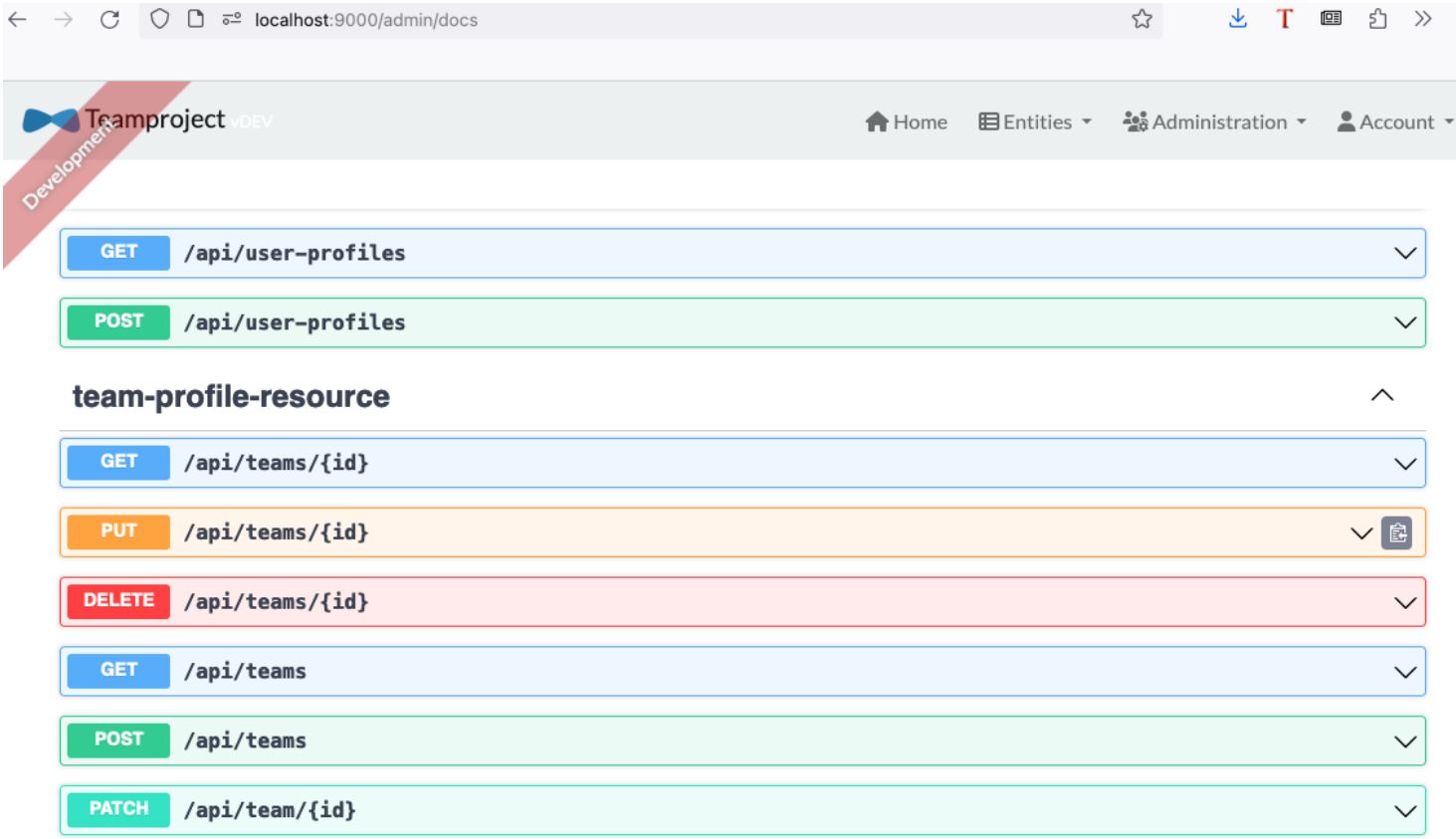
to

```
@GetMapping("/teams")
```

```
@PostMapping("/teams")
```

etc.

If we run `./mvnw` the app will start again and we can even see the swagger UI has updated:

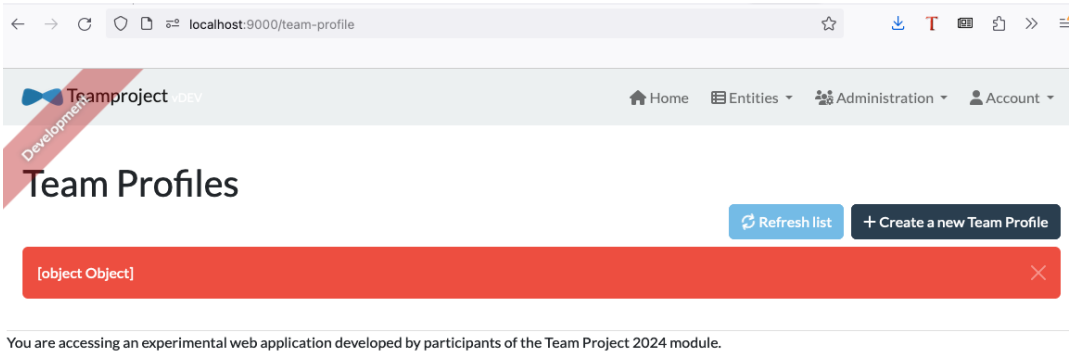


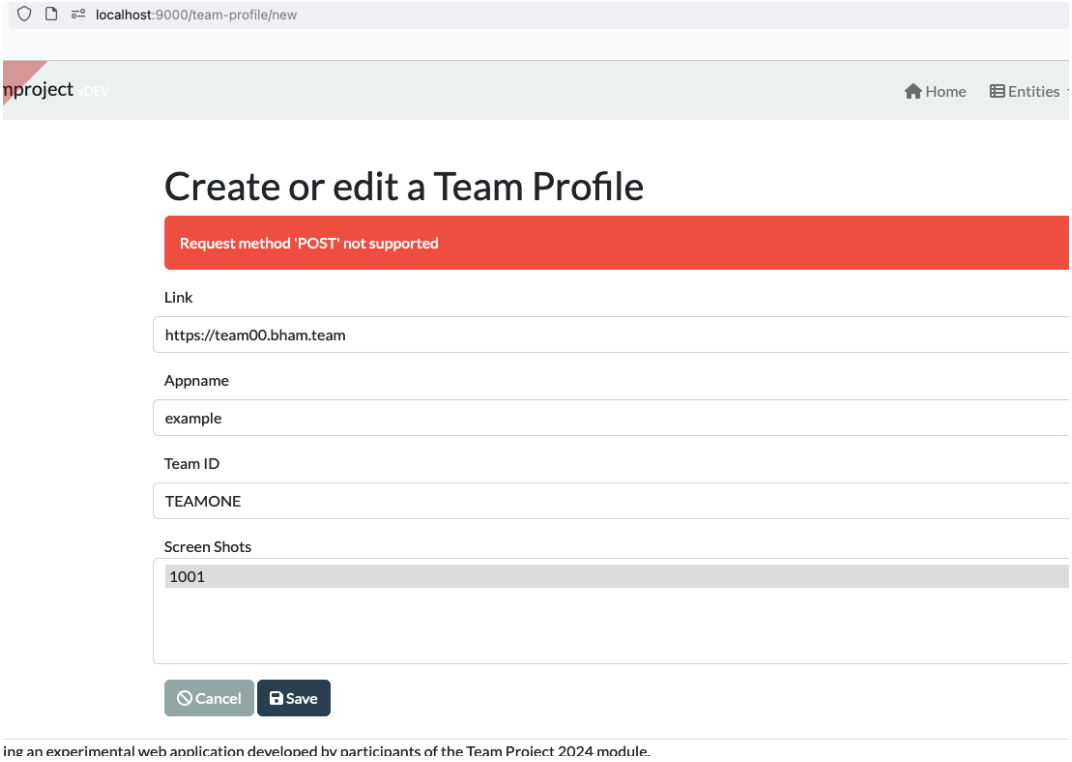
We can make calls against the new API to confirm it works.

However, there is a problem with this change if we navigate to:

`http:/localhost:8080/team-profile`

The issue is the front end still points to the old api endpoint:





The next change we need to make is to the front end, to point to the new API endpoint. In:

`src/main/webapp/app/entities/team-profile/service/team-profile.service.ts`

change:

```
protected resourceUrl = this.applicationConfigService.getEndpointFor('api/team-profiles');
```

to

```
protected resourceUrl = this.applicationConfigService.getEndpointFor('api/teams');
```

then

```
npm start
```

n.b. the changes may be incomplete so make sure to search for any instances of `team-profiles` and

update as needed.

Allow anonymous users to see profiles

Currently, our generated API only allows access to resources where the user is authenticated. For example, try:

`curl -X 'GET''http://localhost:8080/api/teams?eagerload=true'`

or

`curl -X 'GET''http://localhost:8080/api/teams?eagerload=false'`

Question: what is the difference between `eagerload=false` and `eagerload=true`?

Question: how does `eagerload` get communicated with and dealt with in the back end?

```
{
  "type" : "https://www.jhipster.tech/problem/problem-with-message",
  "title" : "Unauthorized",
  "status" : 401,
  "detail" : "Full authentication is required to access this resource",
  "path" : "/api/team-profiles",
  "message" : "error.http.401"
}
```

`curl -X 'GET''http://localhost:8080/api/teams?eagerload=false'-H 'Authorization: Bearer [AUTH_TOKEN]'`

`[AUTH_TOKEN]` comes from http://localhost:8080/admin/docs (need to execute a command).

In some cases, we would like to relax this constraint. To do this, we modify the security configuration in

`src/main/java/team/bham/config/SecurityConfiguration.java`

change:

`.antMatchers("/api/**").authenticated()`

to:

```
1 .antMatchers("/api/team-profiles").permitAll() // or "/teams" if we did part 1
2 .antMatchers("/api/**").authenticated()
```

n.b. the order matters, so the following will not work:

```
1 .antMatchers("/api/**").authenticated()
2 .antMatchers("/api/team-profiles").permitAll()
```

now try:

`curl -X 'GET''http://localhost:8080/api/team-profiles?eagerload=false'`

```
[ {
  "id" : 1,
  "link" : "Shirt Functionality",
  "appname" : "Awesome copying",
  "teamID" : "TEAMONE",
  "screenShots" : null,
  "userProfiles" : null
}, {
...
```

Perform a different action based on authentication status

In certain circumstances we would like to decide what happens depending on the user status, for example logged in or not.

To do this we can modify:

`src/main/java/team/bham/web/rest/TeamProfileResource.getAllTeamProfiles()`

to check the user `Authentication` status.

We could use:

```
Authentication authentication = SecurityContextHolder.getContext().getAuthentication();
```

or

`src/main/java/team/bham/security/SecurityUtils.isAuthenticated()`

we could write a conditional of the form:

```
1 if (SecurityUtils.isAuthenticated()) {
2     //do something
3 } else {
4     //do something different
5 }
```

or

```
1 Authentication authentication =
      SecurityContextHolder.getContext().getAuthentication();
2
3 if (authentication.isAuthenticated()) {
4     //do something
5 } else {
6     //do something different
7 }
```

Challenges:

Make it so that in the front end edit button only appears for team members who are members of that team. n.b. the back end contraint should also be there, why?

Right now no logged-out user can see a list of projects on the home page. Modify the home page to display a list of projects.