Rapport projet JEE 2023-2024
Application CTF avec Quarkus et React.js

Par Emilien MENDES et Téo ZWIEBEL
Enseignant encadrant : Emmanuel CONCHON

Table des matières

1 Introduction

Le but de ce projet est de réaliser une application de CTF en utilisant les notions d'applications distribuées vues en classe. Étant en Master CRYPTIS, nous avons voulu expérimenter la sécurisation d'une telle application en gardant en tête l'approche micro-services. De plus, nous avons pu aller plus loin dans l'utilisation du framework React.js pour l'application cliente.

2 Sécurité

Le but de cette application est en fait d'être d'abord une API qui sera accessible par n'importe quelle personne souhaitant utiliser l'utiliser. C'est pour cela que nous nous sommes tout de suite concentrés sur la partie sécurité. En effet, rendre accessible l'API à n'importe quel utilisateur sans sécurité est une invitation aux débordements. L'authentification permet d'associer des rôles aux utilisateurs de l'API et d'ouvrir des routes qu'à un certain rôle.

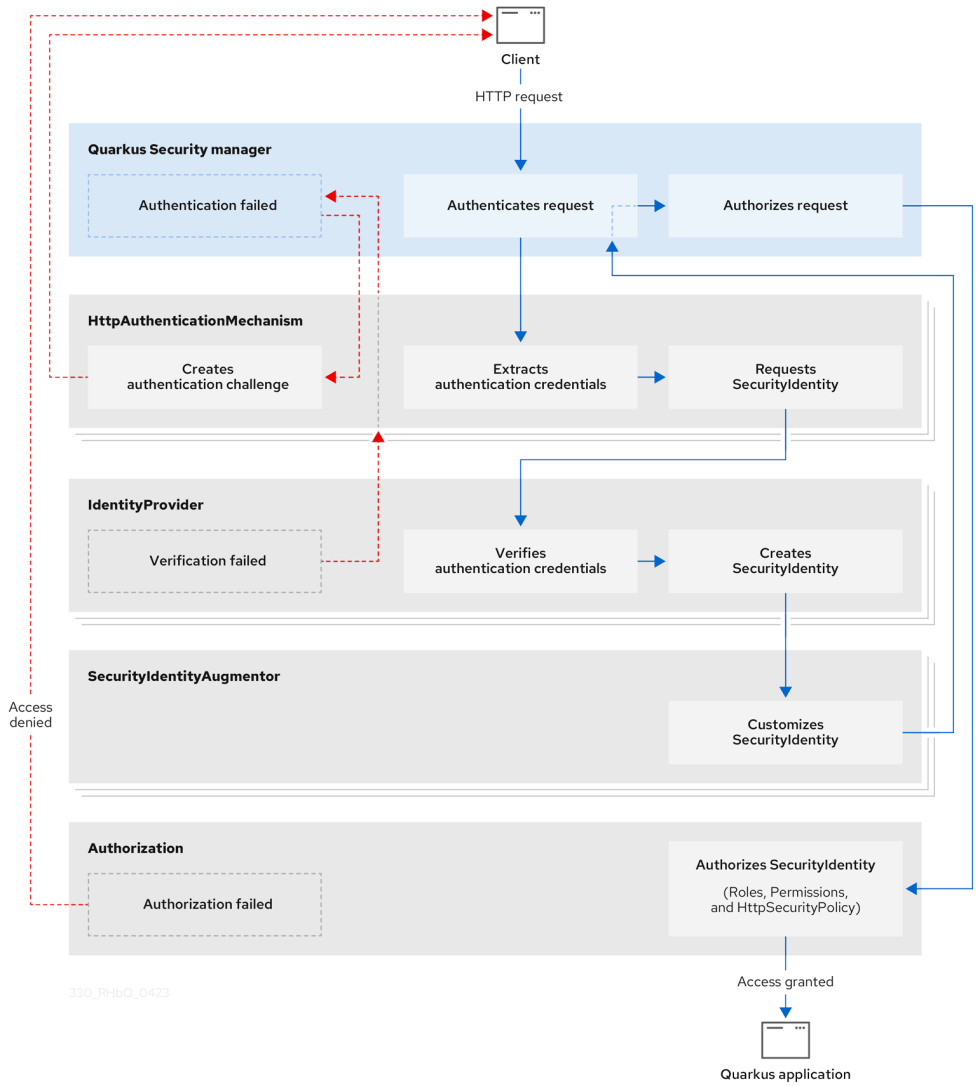

Figure 1 – Processus de sécurité dans Quarkus[1]

Le but ici était d'avoir un système d'authentification complètement isolé de l'application principale, de ce fait, l'authentification est possible même en cas de problème au niveau du coeur de l'application, ce qui permet d'utiliser les autres services tel que celui de discussion. Le processus décrit ici est donc censé se dérouler sur le serveur de sécurité.

2.1 Stockage des rôles

Notre système de stockage de rôle est pratiquement le même que celui que nous pouvons retrouver sur le tutoriel de Quarkus. [2]

```
@Entity
@Table(name = "compte", uniqueConstraints = @UniqueConstraint(columnNames = "pseudo")
    )
@UserDefinition
public class Compte extends PanacheEntity {

    @Username
    public String pseudo;
    @Password
    @JsonbTransient
    public String mdp;
    @Roles
    public String role;

    public static void add(String username, String password, String role) {
        User user = new User();
        user.pseudo = username;
        user.mdp = BcryptUtil.bcryptHash(password);
        user.role = role;
        user.persist();
    }
}
```

Code 1 – Classe Compte permettant de faire un système de comptes et d'authentification

Il est intéressant de noter que cette classe est une "anomalie" dans notre code. En fait, aucune autre entité n'est une PanacheEntity ou encore utilise des fonctions statiques, que cela soit pour ajouter un utilisateur ou recuperer l'*EntityManager* de cette classe. Notre réfléxion ici est vraiment portée encore une fois micro-service. C'est à dire que nous ne gagnerons pas forcement a comprendre ce qu'est une PanacheEntity ou comment cela fonctionne et se traduit en base. On constate que la classe nous permet de créer des utilisateurs avec des mots de passe chiffrés et des rôles en base et va servir à faire de la sécurité. De plus, étant lié à l'authentification, elle sera sur un serveur distant, donc peu importe si nous ne comprenons pas exactement comment elle fonctionne, on sait qu'elle fait quelque chose dont nous avons besoin et elle le fait bien. Cependant, stocker des comptes avec des rôles n'est pas suffisant pour faire de la sécurité, il faut maintenant un moyen d'authentifier l'utilisateur qui interagit avec l'application.

2.2 Authentification basée sur formulaire

Quarkus met à disposition plusieurs mécanismes pour s'authentifier. Le mécanisme sur lequel nous avons jeté notre dévolu est le mécanisme basé sur formulaire car il est assez simple pour que nous puissions l'utiliser et en même temps fournit la sécurité nécessaire à notre application.[3] Comme décrit dans la documentation, pour s'authentifier il faut réaliser une requête *POST* à l'url *j_security_check* de notre application. De la même manière, il nous suffit de reprendre le tutoriel dans la documentation de Quarkus. Ce qui donne en Javascript :

```
const handleLoginFormSubmit = (event) => {
        // evite le rechargement
        event.preventDefault();

        const formData = new URLSearchParams();
        formData.append("j_username", pseudo);
        formData.append("j_password", mdp);

    // Make an HTTP POST request using fetch against j_security_check endpoint
    fetch("http://localhost:8080/j_security_check", {
        method: "POST",
        body: formData,
        headers: {
            "Content-Type": "application/x-www-form-urlencoded",
        },
        credentials: 'include',
    })
    .then((response) => {
```

```
34          if (response.status === 200) {
35              setRedirigerAccueil(true);
36              console.log("Authentification reussie.");
37          } else {
38              setErreur("Pseudo et/ou mot de passe errone(s).")
39              console.error("Pseudo et/ou mot de passe errone(s).");
40          }
41      })
42      }
```

Code 2 – Requête POST sur j_security_check

Si l'authentification est réussie, un cookie est renvoyé, et est stocké dans le navigateur. Il suffira d'ajouter *include : credentials* à chaque requête nécessitant d'être authentifié sinon l'accès à la route sera refusé. On pourrait également changer *j_security_check* pour faire notre propre gestion, mais ce n'est pas le but ici.

3 Fonctionnement de l'application

L'application repose sur plusieurs groupes de trois classes liées. En effet, la première permet de définir le modèle de l'objet en base, la seconde est un bean qui permet de faire des opérations en base sur l'objet, enfin, la dernière est une servlet JAX-RS qui met à disposition une API pour communiquer avec le contrôleur effectuant les opérations.

3.1 Classe/Table/Modèle

Prenons l'exemple d'une Equipe.

```
9  @Entity
10 @Table(uniqueConstraints = @UniqueConstraint(columnNames = "nom"))
11 public class Equipe {
12     @Id
13     @GeneratedValue(strategy = GenerationType.IDENTITY)
14     private int id;
15
16     @Access(AccessType.PROPERTY)
17     private String nom;
18
19     @OneToMany
20     @JsonbTransient
21     private List<Participant> membres;
22
23     @ManyToOne
24     private Participant chef;
25
26     @Access(AccessType.PROPERTY)
27     private String description;
28
29     public Equipe(String nom, String description, Participant chef) {
30         this.nom = nom;
31         this.membres = new ArrayList<>();
32         this.chef = chef;
33         this.description = description;
34     }
35
36     public Equipe() {
37     }
38     // getters & setters
39 }
```

Code 3 – Classe Equipe

On reste sur une définition d'une table en base de données classique avec un id et des champs. Cependant, on notera quelques précisions. Nous avons à la ligne 10 une contrainte sur l'unicité du nom d'équipe. En effet, on se sert des noms des objets pour faire nos opérations en base, il est donc nécessaire qu'ils soient uniques. De plus, les relations sont directement traduites avec des annotations, on peut voir *@OneToMany* au dessus de la liste des membres de l'équipe (ligne 19) qui sont des *Participant*. Mais encore, en dessous on remarque l'annotation

@JsonbTransient. Cette annotation permet de ne pas sérialiser ce champ lorsqu'on voudra le faire avec Jsonb. Il est nécessaire d'utiliser cette annotation pour éviter une serialisation circulaire du type *Equipe* → *Membres* → *Equipe des membres* → *Membres* → ...

3.2 Bean Gestionnaire/Contrôleur

```
@Transactional
@RequestScoped
public class EquipeGestion {

    @PersistenceContext
    EntityManager em;

    @Inject
    ParticipantGestion participantGestionnaire;

    @Inject
    CTFGestion ctfGestionnaire;

    /**
     * Recherche d'une {@link Equipe} avec son nom.
     */
    public Equipe rechercherEquipe(String nom) {
        try {
            Query emQuery = em.createQuery("SELECT e FROM Equipe e WHERE e.nom = :nom", Equipe.class);
            emQuery.setParameter("nom", nom);
            return (Equipe) emQuery.getSingleResult();
        } catch (NoResultException e) {
            return null;
        }
    }
}
```

Code 4 – Classe EquipeGestion

Comme dit précédemment, le but du gestionnaire est d'effectuer des opérations en lien avec les *Equipe*. Sur tous les gestionnaires de notre application on retrouvera le même schéma d'injection. C'est à dire que le *scope* de ce dernier est lors d'une requête, on aura pas besoin de garder des données liées à ce bean dans une session ou autre. Ensuite, on aura besoin d'un *EntityManager* pour interagir avec la base de données via des requêtes. Enfin, pour éviter la redondance de code, on injecte d'autre gestionnaire qui nous permettrons de rechercher des *Participant* ou encore des *CTF* par exemple. Cependant, on limitera ce genre d'injection, un gestionnaire ne s'occupe vraiment que de la table le concernant. Nous pouvons maintenant définir les fonctions d'interaction avec la BDD.

3.3 Servlet JAX-RS

```
@Path("/equipe")
public class EquipeAPI {

    @Inject
    EquipeGestion equipeGestionnaire;

    @Inject
    DemandeGestion demandeGestionnaire;

    @Path("/creer")
    @POST
    @Consumes(MediaType.APPLICATION_FORM_URLENCODED)
    @RolesAllowed("participant")
    public Response creerEquipe(@FormParam("nom") String nom, @FormParam("description") String description, @Context SecurityContext securityContext) {
        equipeGestionnaire.creerEquipe(nom, description, securityContext.getUserPrincipal().getName());
        return Response.ok("Equipe creee avec succes").build();
```

```
36      }
37
38      @Path("/all")
39      @GET
40      @Produces(MediaType.APPLICATION_JSON)
41      @PermitAll
42      public Response tousEquipes() {
43          ArrayList<Equipe> equipes = equipeGestionnaire.toutesEquipe();
44          return Response.ok(equipes, MediaType.APPLICATION_JSON).build();
45      }
46  }
```

Code 5 – Servlet JAX-RS EquipeAPI

On peut voir l'injection des gestionnaire dans notre servlet. On reste sur une servlet JAX-RS classique, cependant on peut maintenant voir l'usage du service de sécurité à partir de la ligne 32 avec l'annotation *@RolesAllowed()*. Mais encore, on utilise *@Context SecurityContext* afin de récupérer l'utilisateur connecté, ce qui empêchera les utilisateurs malveillants d'effectuer des opérations sur des objets ne les concernant pas. Enfin, on peut voir à la ligne 44 que nous retournons des objets au format JSON.

3.4 ParticipantCTF

```
10  @Entity
11  public class ParticipantCTF {
12
13      @Id
14      @GeneratedValue(strategy = GenerationType.IDENTITY)
15      private int id;
16
17      @ManyToOne
18      private CTF ctf;
19
20      @ManyToOne
21      private Participant participant;
22
23      @Access(AccessType.PROPERTY)
24      private int score;
25
26      public ParticipantCTF(CTF ctf, Participant participant, int score) {
27          this.ctf = ctf;
28          this.participant = participant;
29          this.score = score;
30      }
31
32      public ParticipantCTF() {
33      }
34
35      // getters & setters
36  }
```

Code 6 – Classe ParticipantCTF

L'une des subtilité à laquelle nous avons dû faire face est celle de la relation entre les *Participant* et les *CTF* auxquels ils participent. Initialement, nous avons utilisé une simple annotation *@ManyToMany* mais nous ne pouvions pas manipuler la table résultante pour y ajouter le score. Nous avons donc crée une classe permettant de faire la relation.

4 FrontEnd React.js

Le problème de la sécurité géré sur le serveur nous permet une flexibilité au niveau du frontend. Il nous fallait juste trouver un moyen de faire une application multi-pages et cohérente avec nos routes sur l'API, c'est à dire utiliser des paramètres dans le *Path*. Notre solution est *react-router*. [4]

4.1 React-router

Avec React-router, on peut associer un composant à une route, ce qui nous permettra de bien cloisonner les usages des composants. De plus, on peut également avoir un composant de navigation permanent sur toutes les pages de notre application. La principale difficulté concernant l'import de react-router était de trouver une version disponible téléchargeable depuis le navigateur et qui disposait du *hook useParams()*. Merci à l'utilisateur *duhaime* sur stackoverflow pour sa réponse qui nous à été utile.

4.1.1 Utilisation du composant de navigation

```
106             {this.state.role == "[admin]" ? (
107               <ul>
108                 <li>
109                   <Link to="/adminPanel/CTF" className="link-style-specific-element">
    Gestion des ctf</Link>
110                 </li>
111                 <li>
112                   <Link to="/adminPanel/participants" className="link-style-specific-
    element">Gestion des participants</Link>
113                 </li>
114               </ul>) : ("")}
115             {this.state.role == "[organisateur]" ? (
116               <ul>
117                 <li>
118                   <Link to="/creerCTF" className="link-style-specific-element">
    Creation des CTF</Link>
119                 </li>
120                 <li>
121                   <Link to="/modifCTF" className="link-style-specific-element">
    Modification des CTF</Link>
122                 </li>
123               </ul>
124             ) : ("")
```

Code 7 – Composant de navigation avec conditions

L'une des utilisations que nous avons faite du composant de navigation est d'afficher des routes en fonction du rôle de l'utilisateur connecté. On utilise la propriété des composant de React suivante : le composant appelle sa fonction de rendu lorsque l'un de ses états est mis à jour. Donc, lorsque nous récupérons le rôle et l'utilisateur, le composant de navigation appelle de nouveau *render()* et peut afficher les routes cachées.
Cependant, un problème que nous n'avons pas réussi à régler est le fait de mettre à jour ces états au sein du composant. La solution actuelle, bien que peu optimale nous permet d'atteindre ce but :

```
58  componentDidMount() {
59    this.recupererUtilisateur();
60    this.recupererRole();
61    this.refreshInterval = setInterval(() => {
62      this.recupererUtilisateur();
63      this.recupererRole();
64    }, 3000);
65  }
66
67  componentWillUnmount() {
68    clearInterval(this.refreshInterval);
69  }
```

Code 8 – Intervalle de rafraichissment du rôle et de l'utilisateur

Ce type de fonctionnement peut quand même permettre de gérer une éventuelle déconnexion suite à l'expiration d'un cookie d'authentification.

4.1.2 Définition des routes

On définit les routes, et on les associe à un composant au niveau du rendu du composant principal, entre deux balises *Switch*.

```
142     <Switch>
143             <Route path="/ctfs" component={MenuCTFComposant} />
```

```
144             <Route path="/creerCTF" component={CreerCTFComposant} />
145             <Route path="/ctf/:titreCTF" component={ConsulterCTFComposant} />
146     </Switch>
```

Code 9 – Exemple de définition de route

Un paramètre de chemin est précédé de *:* nous verrons par la suite comment nous l'utiliserons.

4.2 Fonctionnement général

Maintenant que nous avons associé un composant à une route, il ne reste plus qu'à faire les composants. Regardons de plus près notre composant le plus complet, qui reprend les points cruciaux de notre conception. Ce document se trouve en annexe.

- Lignes 1 à 5 : Le composant n'est pas défini de la même manière que le composant principal. En effet, il est défini avec une fonction. Nous avons fait ce changement afin de pouvoir utiliser *useParams()* qui est un *hook* inutilisable dans un composant de type objet.
- Lignes 7 à 9 : *useEffect()* est également un *hook* permettant d'exécuter du code, on peut l'assimiler à *componentDidMount()* d'un composant objet. Ce qui va nous intéresser le plus est le fait de récupérer le paramètre de la route en fonction de la définition donnée dans le composant principal.
- Lignes 12 à 33 : Pour gérer la saisie des scores pour les participants, nous ne pouvions pas utiliser un simple formulaire. Nous avons donc du implémenter une fonction de gestion de la modification d'un élément d'un tableau qui est un état du composant.
- Lignes 35 à 53 : Voici l'une des fonctions classique de récupération de participants. C'est une requête GET sur notre API de CTF permettant de récupérer les participants d'un CTF donné en paramètre de la route, on utilise également les paramètres de la route de l'application cliente. Comme le format d'échange est JSON, on peut transformer la réponse en JSON et la traiter.
- Lignes 54 à 77 : La fonction de saisie des scores principale, elle va permettre de faire une requête POST sur l'API de CTF. Sur notre API, on précise avec *@Consumes* que nous allons recevoir un objet JSON, c'est au client de bien le formater et l'envoyer dans le body. Étant donnée que la saisie des scores est ouverte uniquement aux organisateurs et qu'il faille vérifier l'identité de ce dernier pour qu'aucun organisateur ne puisse saisir les score d'un CTF ne lui appartenant pas on rajoute *credentials : "include"* ce qui aura pour effet d'envoyer son identité à l'aide du cookie donné lors de la connexion.
- Lignes 79 à 100 : Ce que retourne un composant est ce qu'il sera affiché sur la page. *className=""* permet d'affecter un style venant d'un module CSS prédéfini.

5 Problèmes

5.1 CRUD

Initialement, les gestionnaires permettaient de faire des actions en base sur la table les concernant mais aussi de faire des actions plus éloignées. Par exemple, le gestionnaire de Participant permettait évidemment de créer un participant et de le modifier mais aussi de poster un commentaire. Cela rendait le code moins lisible et moins cohérent. De plus, lors de l'injection dans une servlet JAX-RS soit nous avions un gestionnaire avec trop d'interactions possibles, soit une servlet mettant à disposition trop de routes. De ce fait, la relation entre URI et méthode HTTP dans une architecture REST n'était pas respectée. En effet, le CRUD n'était donc pas respecté, les méthodes de consultations de commentaire se trouvaient dans l'API de commentaire mais la création était dans l'API de participant. Nous avons donc changé cela pour respecter l'architecture REST.

5.2 DTO

Lorsque l'on envoie des objets au format JSON, on envoie bien plus de données que nécessaire, malgré l'utilisation de l'annotation *@JsonbTransient*. L'une des améliorations possible est d'utiliser une système de DTO (Data Transfer Object) pour formater les données envoyées. Par exemple, la consultation d'un participant renvoi son équipe également. On pourrait faire en sorte d'ajouter des paramètres en requête pour que l'utilisateur choisisse si il veut également l'équipe ou non.

6 Conclusion

En résumé, même si nous n'avons pas réussi à mettre en place de multiples serveurs pour les services de notre application, l'architecture micro-service nous garanti une extensibilité future. En effet, les différentes parties de l'application sont le plus indépendantes possible, ce qui permet par exemple de facilement d'ajouter une route à l'API principale que ce soit via une nouvelle servlet ou une nouvelle interaction disponible dans un gestionnaire ou d'ajouter un système de mail sur un serveur distant. Enfin, nous avons vu à quel point les framework javascript peuvent être puissants et utiles dans le contexte de développement d'une application frontend.

7 Annexe

```
const SaisieScoresCTFComposant = () => {
    const [participants, setParticipants] = React.useState([]);
    const [scores, setScores] = React.useState([]);
    const [erreur, setErreur] = React.useState("");
    const { CTFtitre: routeCTFtitre } = useParams();

    React.useEffect(() => {
        fetchParticipants();
    }, [routeCTFtitre]);

    // Change la valeur du tableau de score en ne modifiant que la valeur de l'
    element modifie
    const handleScore = (event) => {
        setScores(prevScores => {
            // Copie du tableau initial
            var updateScore = [];
            for (let index = 0; index < prevScores.length; index++) {
                updateScore.push(prevScores[index]);

            }
            // Modification de la valeur correspondant au score modifie
            var _value = 0;
            try{
                _value =parseInt(event.target.value);
            }
            catch(error){
                console.log(error);
            }
            finally{
                updateScore[event.target.id] = _value;
            return updateScore;
            }
        });
    };

    const fetchParticipants = () => {
        fetch(`http://localhost:8080/ctf/${routeCTFtitre}/participants`, {
            method: 'GET',
            credentials: "include"
        })
            .then(resp => {
                if (!resp.ok) {
                    console.error("Erreur de recuperation des participants du CTF.");
                }
                return resp.json();
            })
            .then(data => {
                // Tableau  avec tous les pseudos des participants
                setParticipants(Object.keys(data).map((pseudo) => pseudo));
                // Tableau avec tous les scores des participants
                setScores(Object.keys(data).map((pseudo) => data[pseudo]))
            })
    }

    const saisieScore = (event) => {
        //evite le rechargement
        event.preventDefault();
        // Creation d'un tableau que l'on met sous format JSON
        const jsonData = JSON.stringify( Object.fromEntries(participants.map((key,
    index) => [key, scores[index]])));

        fetch(`http://localhost:8080/ctf/${routeCTFtitre}/saisirScore`, {
            method: "POST",
            body: jsonData,
```

```
            headers: {
                "Content-Type": "application/json",
            },
            credentials: 'include',
        })
            .then((response) => {
                if (response.status === 200) {
                    console.log("Score saisis avec succes.");
                    setErreur("Score saisis !")
                } else {
                    console.error("Erreur lors de la saisie.");
                    setErreur("Erreur lors de la saisie, verifiez les champs.")
                }
            })
    }

    return (
        <div>
            <div className="conteneurGenerique">
                <h1> Saisie des scores {routeCTFtitre}</h1>
            </div>
            <div className="conteneurGenerique">
                <h2>Liste des partipants du CTF</h2>
                <form onSubmit={saisieScore} className="formulaire">
                    {Object.keys(participants).map((index) => (
                        <ul key={index} style={{ listStyleType: "none" }}>
                        <li>{participants[index]} </li>
                        <label htmlFor="score"> Score </label>
                        <input type="text" id={index} name="score" value={scores[
    index]} onChange={(e) => handleScore(e)} />
                        </ul>
                    ))}
                    <br /><button type="submit">Appliquer</button>
                    <br />{erreur}
                </form>
            </div>
        </div>
    );
}
```

Code 10 – Composant SaisieScoresCTFComposant

Références

[1] *Quarkus Security Architecture*. URL : https://quarkus.io/guides/security-architecture.

[2] *Quarkus Security with Jakarta Persistence*. URL : https://quarkus.io/guides/security-jpa.

[3] *Authentication Mechanisms in Quarkus*. URL : https://quarkus.io/guides/security-authentication-mechanisms#form-auth.

[4] *React-router*. URL : https://reactrouter.com/en/main.