



# DOSSIER PROFESSIONNEL (DP)

*Nom de naissance* ➤ Berbigier  
*Nom d'usage* ➤ Berbigier  
*Prénom* ➤ Thomas  
*Adresse* ➤ 983 avenue de l'amandier 84140 Avignon

## Titre professionnel visé

Développeur web et web mobile

### MODALITÉ D'ACCÈS :

■ Parcours de formation

## Présentation du dossier

Le dossier professionnel (DP) constitue un élément du système de validation du titre professionnel.

**Ce titre est délivré par le Ministère chargé de l'emploi.**

Le DP appartient au candidat. Il le conserve, l'actualise durant son parcours et le présente

**obligatoirement à chaque session d'examen.**

Pour rédiger le DP, le candidat peut être aidé par un formateur ou par un accompagnateur VAE.

Il est consulté par le jury au moment de la session d'examen.

### Pour prendre sa décision, le jury dispose :

1. des résultats de la mise en situation professionnelle complétés, éventuellement, du questionnaire professionnel ou de l'entretien professionnel ou de l'entretien technique ou du questionnement à partir de productions.
2. du **Dossier Professionnel** (DP) dans lequel le candidat a consigné les preuves de sa pratique professionnelle.
3. des résultats des évaluations passées en cours de formation lorsque le candidat évalué est issu d'un parcours de formation
4. de l'entretien final (dans le cadre de la session titre).

*[Arrêté du 22 décembre 2015, relatif aux conditions de délivrance des titres professionnels du ministère chargé de l'Emploi]*

### Ce dossier comporte :

- pour chaque activité-type du titre visé, un à trois exemples de pratique professionnelle ;
- un tableau à renseigner si le candidat souhaite porter à la connaissance du jury la détention d'un titre, d'un diplôme, d'un certificat de qualification professionnelle (CQP) ou des attestations de formation ;
- une déclaration sur l'honneur à compléter et à signer ;
- des documents illustrant la pratique professionnelle du candidat (facultatif)
- des annexes, si nécessaire.

*Pour compléter ce dossier, le candidat dispose d'un site web en accès libre sur le site.*



## Sommaire

### Exemples de pratique professionnelle

<b>Développer la partie front-end d'une application web ou web mobile sécurisée</b>	<b>p.</b>	<b>5</b>
- Jeu de dés	p. p.	5-11
- Gestion de stock	p. p.	12-22
<b>Développer la partie back-end d'une application web ou web mobile sécurisée</b>	<b>p.</b>	<b>22</b>
- Gestion de stock	p. p.	22-40
- Gestion Fournitures	p. p.	40-52
<b>Titres, diplômes, CQP, attestations de formation</b> <i>(facultatif)</i>	<b>p.</b>	<b>54</b>
<b>Déclaration sur l'honneur</b>	<b>p.</b>	<b>55</b>
<b>Documents illustrant la pratique professionnelle</b> <i>(facultatif)</i>	<b>p.</b>	<b>56</b>
<b>Annexes</b> <i>(Si le RC le prévoit)</i>	<b>p.</b>	<b>56-62</b>

# **EXEMPLES DE PRATIQUE PROFESSIONNELLE**

## Activité-type 1 Développer la partie front-end d'une application web ou web mobile sécurisée

### Exemple n°1 - Jeu de dés

---

#### 1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

##### Introduction au projet :

Dans le cadre de ma formation en développement web et mobile, j'ai conçu un jeu de dés interactif destiné à deux joueurs, jouable sur un seul et même écran. L'objectif principal de ce projet était de développer une interface utilisateur dynamique, mettre en pratique mes compétences en **JavaScript**, **HTML**, **CSS** et **Bootstrap** et créer un design responsive.

Concept du Jeu :

Le jeu repose sur un principe simple mais captivant : chaque joueur commence avec un score temporaire qui est initialisé à zéro à chaque tour. L'objectif est d'atteindre un score global de 100 points pour gagner. À chaque tour, les joueurs ont la possibilité de lancer un dé autant de fois qu'ils le souhaitent, en accumulant les points de leurs lancers dans leur score temporaire. Cependant, une décision stratégique est à prendre : cliquer sur "Hold" pour transférer les points temporaires vers le score global, ou tenter un lancer supplémentaire, au risque de perdre tous les points accumulés si le dé affiche un 1.

##### Installation de l'environnement de travail :

Tout projet débute par l'installation et la configuration de l'environnement de travail, une étape cruciale qui pose les bases du développement. Pour ce projet, j'ai choisi de travailler sur un environnement local, avec un poste de travail équipé de **Windows**.

J'ai utilisé **Visual Studio Code** comme éditeur de code, car il est gratuit et facile à prendre en main, ce qui est particulièrement bénéfique pour les débutants. Cet éditeur offre une multitude d'extensions qui

améliorent la productivité et la visibilité, telles que **Prettier** pour le formatage automatique et **Live Server** pour le rechargement en temps réel des modifications.

La gestion de version avec **Git** et **GitHub** a été indispensable pour suivre l'évolution de mon code, avec des commits réguliers pour chaque ajout ou correction significative, ce qui me permettait de revenir à des versions antérieures si nécessaire.

Dans ce projet, j'ai commencé par créer un dossier nommé **ProjetJeuStudi**, j'ai ensuite initialisé le dépôt **Git**,

```
mkdir ProjetJeuStudi
```

```
cd ProjetJeuStudi
```

```
git init
```

puis j'ai créé le dépôt distant sur **GitHub** depuis mon compte. J'ai lié le dépôt local au dépôt **GitHub** avec l'URL du dépôt.

```
git remote add origin https://github.com/ThomasBerbigier/ProjetJeuStudi.git
```

Le premier commit a été effectué puis poussé sur **GitHub**, et des commits réguliers sont faits pour garder une trace des modifications effectuées.

```
git commit -m "Initial commit"
```

```
git push -u origin main
```

J'ai créé un dossier **images** pour organiser le code et y placer les images de dés, la police d'écriture et les fichiers liés à **Bootstrap**. J'ai ouvert le dossier avec **VSCode**, créé le fichier **index.html** auquel j'ai intégré **Bootstrap** en local via les fichiers **bootstrap.bundle.min.js** et **bootstrap.min.css** que j'ai téléchargé depuis la documentation **Bootstrap**. Cela présente plusieurs avantages de performance et de disponibilité hors ligne.

```
<link rel="stylesheet" href="images/bootstrap.min.css">
```

```
<script src="images/bootstrap.bundle.min.js"></script>
```

L'utilisation de **Bootstrap** me permet de gagner un temps précieux dans la création d'une interface responsive et esthétiquement plaisante, car il fournit des classes préconçues qui facilitent la mise en page et le design sur différents appareils.

J'ai inclus une feuille de style **CSS** pour séparer la structure de la présentation.

```
<link rel="stylesheet" href="style.css">
```

Dans ce fichier, j'ai effectué un reset **CSS** afin de m'assurer que tous les navigateurs commencent avec des styles uniformes, et j'ai choisi une police personnalisée pour donner une identité visuelle à mon application.

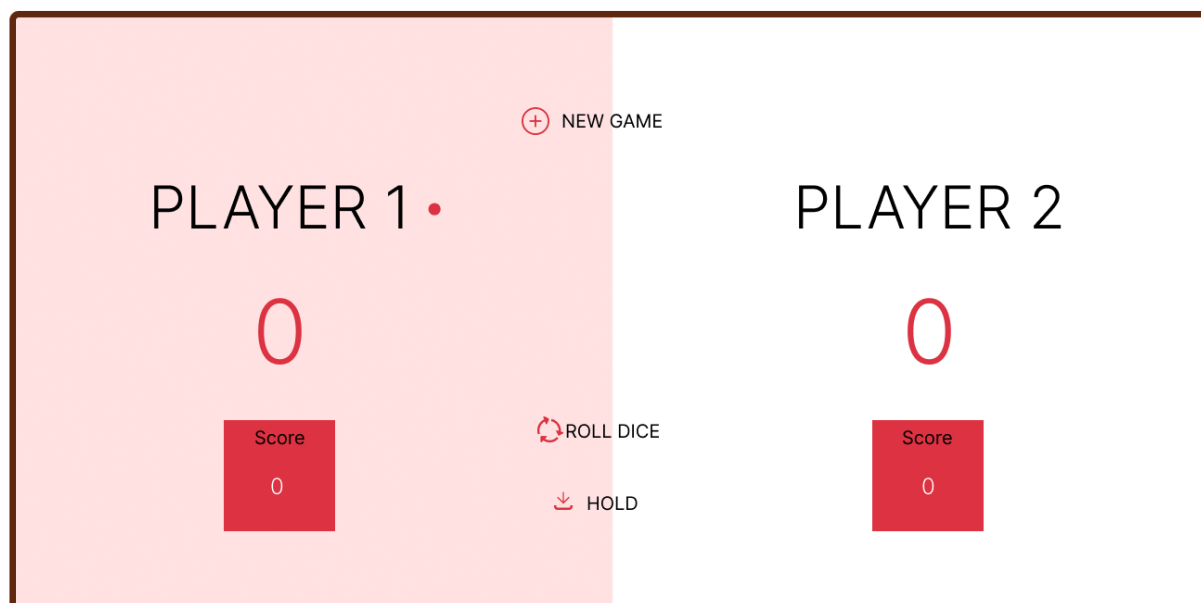
```
@font-face {  
    font-family: lato;  
    src: url(images/Lato/Lato-Black.ttf);  
}  
  
* {  
    margin: 0;  
    padding: 0;  
    box-sizing: border-box;
```

Le fichier **script.js**, qui contient la logique du jeu, est placé en bas de la page **HTML**, juste avant la balise **</body>** pour permettre au navigateur de charger et rendre tout le contenu **HTML** avant d'exécuter le script.

```
<script src="script.js"></script>
```

## Maquettage :

Pour le maquettage des interfaces utilisateurs statiques web et web mobile j'ai utilisé l'outil en ligne **FIGMA**. Ci-dessous la version bureautique, Figure 1 en Annexe pour la version mobile.



## Développement de l'interface statique :

L'intégration des maquettes dans le code **HTML** a été rapide car l'application ne comprend qu'une seule page. Les balises **HTML** sémantiques ont été utilisées pour structurer le contenu. L'ajout de commentaires permet de mieux séparer les parties de l'application.

```
<body>
  <main id="gradient">
    <!-- Player 1 -->
    <section class="content">
      <h1 class="player-1 display-3" id="player-1">PLAYER 1
      <svg xmlns="http://www.w3.org/2000/svg" width="50" height="50" fill="#DC3545" class="bi bi-dot" viewBox="0 0 16 16" id="dot-1">
        <path d="M8 9.5a1.5 1.5 0 1 0 0-3 1.5 1.5 0 0 0 0 3"/>
      </svg>
    </h1>
```



J'ai créé les éléments statiques de l'application comme les scores et les boutons en utilisant les classes **Bootstrap** et du **CSS** personnalisé pour harmoniser les couleurs et l'apparence des éléments. Chaque joueur a son propre affichage avec un score actuel et un score total.

```
main {  
  background: linear-gradient(to right, #FFE5E5 50%, rgb(255,255,255) 50%);  
  width: auto;  
  height: auto;  
  margin: 10% 7% 15% 7%;  
  padding: 5%;  
  border-radius: 10px;  
  border: #622A0F solid 5px;  
}
```

### Développement de la partie dynamique :

La logique du jeu repose sur **JavaScript**, où j'ai utilisé des événements DOM pour gérer les interactions utilisateurs, notamment lors des lancers de dés ou du passage de tour. Chaque clic sur le bouton "Lancer le dé" déclenche la génération d'un nombre aléatoire compris entre 1 et 6.

```
// Roll dice  
rollDice.addEventListener("click", () => {  
  // dice display  
  dice.style.display = "block"  
  
  // Random number between 1 and 6  
  randomDice = Math.floor(Math.random() * 6) + 1  
  // display result  
  dice.src = "images/Dé " + randomDice + ".png"  
  
  // Player 1 turn, else player 2  
  if(activePlayer) {  
    player1()  
  } else {  
    player2()  
  }  
});
```

Si le joueur obtient un 1, il perd les points accumulés lors de ce tour et cède la main à l'autre joueur. J'ai utilisé un changement de style visuel avec un linear-gradient pour indiquer le joueur actif, rendant l'expérience plus intuitive.

```
// Player 1
function player1() {
    // Display or hide dot when player's turn
    dot1.style.display = "block"
    dot2.style.display = "none"
    gradient.style.background = "linear-gradient(to right,#FFE5E5 50%, rgb(255,255,255) 50%)"
    // if dice = 1 change player's turn, else add to current
    if (randomDice !== 1) {
        currentScore1 = currentScore1 + randomDice
        currentScore1 = document.getElementById("current-score-1").textContent = currentScore1
    } else {
        currentScore1 = 0
        currentScore1 = document.getElementById("current-score-1").textContent = currentScore1
        activePlayer = false
    }
};
```

Lorsqu'un des deux joueurs atteint ou dépasse le score de 100 points, il est déclaré vainqueur de la partie.

```
// Winner
if (score1 >= 100){
    firstPlayer.innerHTML = "WINNER"
} else if (score2 >= 100) {
    secondPlayer.innerHTML = "WINNER"
}
});
```

L'application a finalement été mise en ligne grâce à GitHub Pages depuis la branche main.

### Conclusion :

Le développement du jeu de dés en **HTML**, **CSS**, et **JavaScript** m'a permis de consolider mes compétences en développement front-end en créant une interface ludique et interactive. Ce projet m'a appris à structurer efficacement un projet, en partant de la conception des maquettes via Figma jusqu'à

## DOSSIER PROFESSIONNEL <sup>(DP)</sup>

l'implémentation du jeu avec Bootstrap pour assurer une mise en page responsive.

La logique du jeu a renforcé mon apprentissage des concepts de programmation JavaScript, tels que la gestion des événements, la manipulation du DOM, et l'optimisation de l'expérience utilisateur grâce à des éléments interactifs.

### 2. Précisez les moyens utilisés :

Pour réaliser ce projet j'ai réalisé les maquettes avec Figma, utilisé l'IDE Visual Studio Code, HTML5, CSS, Bootstrap 5.3, JavaScript, Git, Github et mon OS Windows 11.

### 3. Avec qui avez-vous travaillé ?

Pour ce projet j'ai travaillé seul.

### 4. Contexte

Nom de l'entreprise, organisme ou association ▶ Studi

Chantier, atelier, service ▶

Période d'exercice ▶ Du : 10/01/2024 au : 25/01/2024

### 5. Informations complémentaires *(facultatif)*

Ce projet reprend les quatre compétences de la première activité type du référentiel, soit:

1. Installer et configurer son environnement de travail en fonction du projet web ou web mobile,
2. Maquetter des interfaces utilisateur web ou web mobile
3. Réaliser des interfaces utilisateur statiques web ou web mobile
4. Développer la partie dynamique des interfaces utilisateur web ou web mobile

## Activité-type 1

Développer la partie front-end d'une application web ou web mobile sécurisée

### Exemple n°2 - Gestion de stock

#### 1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

##### Introduction au projet :

Dans le cadre d'un projet personnel, j'ai développé une application de gestion de stock pour répondre aux besoins des entreprises. Cette application, entièrement réalisée en Angular, intègre diverses technologies pour assurer une expérience utilisateur optimale et une gestion efficace des données. Le projet visait à fournir une solution complète de gestion de stock tout en renforçant mes compétences sur le framework Angular pour la partie front-end.

Les fonctionnalités de l'application permettent aux utilisateurs de :

- Créer et gérer des profils d'entreprise.
- Gérer des catégories d'articles et suivre les mouvements de stock en temps réel.
- Créer des commandes clients et fournisseurs, tout en suivant les entrées et sorties de stock associées.
- Visualiser l'état du stock à tout moment et effectuer des ajustements si nécessaire.
- Gérer les utilisateurs, clients, et fournisseurs liés à une entreprise.

##### Installation de l'environnement de travail :

Pour ce projet, l'installation de mon environnement de travail a débuté par le téléchargement de **Node.js** pour gérer les dépendances du projet et utiliser npm, le gestionnaire de paquets nécessaires à Angular.

J'ai pu bénéficier de l'IDE **IntelliJ IDEA Ultimate** via le GitHub Student Pack, ce qui m'a permis de découvrir un éditeur de code professionnel.

**Angular CLI** a été installé pour générer les composants et services nécessaires via des commandes simples.

La commande `ng new gestionDeStock` m'a permis d'initialiser un projet **Angular** structuré avec les fichiers nécessaires à un bon démarrage (configuration, dossier src, etc.).

Les bibliothèques **Bootstrap** pour ses composants et sa grille responsive, et **FontAwesome** pour ses icônes ont été installées grâce aux commandes `ng add @ng-bootstrap/ng-bootstrap` et

```
npm install @fortawesome/angular-fontawesome
```

### Maquettage :

J'ai utilisé Figma pour concevoir les maquettes des différentes pages de l'application, visualisables en Annexe (Figures 2 à 5).

### Liaison de données et authentification :

Les maquettes effectuées, j'ai créé les interfaces statiques en suivant une approche modulaire, où chaque composant gère une partie spécifique de l'application. Par exemple, le composant **page-login** (Figure 2) gère une partie de l'application de manière indépendante. Cette interface est le point d'entrée pour accéder à l'application après authentification. J'ai conçu ce composant de manière responsive et sécurisée pour l'utilisateur. Le fichier **page-login.component.html**, créé avec la commande `ng g c page-login`, contient la structure du formulaire de connexion. L'utilisation de **Bootstrap** permet de profiter de composants préconstruits comme les cartes, ce qui facilite la mise en page tout en restant visuellement cohérent. Le formulaire de connexion est inclus dans une **card** avec une entête **card-header** et un corps **card-body** bien défini.

```
<div class="card justify-content-md-center">
  <div class="card-header">
    <h1 class="text-center fs-3">Se connecter</h1>
  </div>
  <div class="card-body">
```

Une couleur de fond discrète est appliquée dans le fichier **page-login.component.scss** pour une apparence sobre et moderne.

```
body {  
  background-color: #eff2f5 !important;  
}
```

Les champs d'entrée pour l'email et le mot de passe utilisent **ngModel**, une directive d'**Angular** qui crée une liaison bidirectionnelle entre le champ et l'objet **authenticationRequest**, permettant une mise à jour automatique des données.

```
<div class="mb-3">  
  <input type="email" class="form-control" placeholder="E-mail" [(ngModel)]="authenticationRequest.login">  
</div>  
<div class="mb-3">  
  <input type="password" class="form-control" placeholder="Mot de passe" [(ngModel)]="authenticationRequest.password">  
</div>
```

Si une erreur survient, une alerte **alert-danger** informe l'utilisateur que l'authentification a échoué. De plus, j'ai intégré des validations pour m'assurer que les informations soient correctes et sécurisées.

```
<div class="alert alert-danger" *ngIf="errorMessage">  
  {{errorMessage}}  
</div>
```

Dans le fichier **page-login.component.ts**, l'objet **authenticationRequest** est initialisé, et les propriétés **login** et **password** sont définies dans cet objet. Ainsi, **ngModel** connecte le formulaire à l'objet **authenticationRequest**.

```
authenticationRequest : AuthenticationRequest = {};
```

Le bouton S'inscrire redirige l'utilisateur vers la page d'inscription (figure 3 en Annexe) pour les nouveaux utilisateurs, et le bouton Se connecter déclenche la méthode `login()`, liée à la logique d'authentification.

```
<div class="mb-3 d-flex justify-content-around">
  <a [routerLink]="['/Inscription']" class="btn btn-link">S'inscrire</a>
  <button type="button" class="btn btn-primary" (click)="login()">
    <fa-icon [icon]="faCheckSquare"></fa-icon> Se connecter
  </button>
```

Grâce au **router Angular**, la navigation s'effectue facilement entre les sections de l'application. Lors du clic sur le bouton "S'inscrire", l'utilisateur est alors redirigé vers la page **Inscription** (Figure 3), permettant aux utilisateurs de créer un compte entreprise .

### Gestion des utilisateurs et inscription :

La page utilise une **card Bootstrap** pour contenir le formulaire d'inscription. La **card** est centrée à l'aide de la classe **Bootstrap justify-content-md-center** pour assurer une mise en page agréable sur différents types d'écrans. Le formulaire est surmonté d'un en-tête dans la **card-header** avec un titre de taille **h1** et une classe **fs-3** pour ajuster la taille du texte. Ce titre est centré pour capter l'attention de l'utilisateur.

```
<div class="card justify-content-md-center">
  <div class="card-header">
    <h1 class="text-center fs-3">S'inscrire</h1>
  </div>
  <div class="card-body">
```

Si des erreurs surviennent, elles sont affichées à travers une alerte `alert alert-danger`. La directive **Angular** `*ngIf` contrôle l'affichage de ce bloc, et la boucle `*ngFor` permet de parcourir et d'afficher chaque message d'erreur. Cette structure reste statique dans l'**HTML**, mais se met à jour dynamiquement selon les erreurs rencontrées.

```
<div class="alert alert-danger" *ngIf="errorMsg && errorMsg.length > 0">
  <div *ngFor="let msg of errorMsg">
    <span >{{msg}}</span>
  </div>
</div>
```

Le formulaire est composé de plusieurs champs d'entrée `input` et `textarea`, chacun associé à un modèle de données **Angular** `ngModel`. Ces champs incluent des placeholders qui guident l'utilisateur lors de la saisie des informations nécessaires telles que le nom de l'entreprise, le code fiscal, l'email, l'adresse, etc.

```
<div class="mb-3">
  <input type="text" class="form-control" placeholder="Nom" [(ngModel)] = "entrepriseDto.nom">
</div>
<div class="mb-3">
  <input type="text" class="form-control" placeholder="Code fiscal" [(ngModel)] = "entrepriseDto.codeFiscal">
</div>
```

Pour la partie TypeScript, les modèles de données `entrepriseDto` et `adresse` stockent les informations saisies par l'utilisateur dans le formulaire d'inscription.

```
entrepriseDto: EntrepriseDto = {};
adresse: AdresseDto = {};
```

Lorsque l'utilisateur clique sur le bouton "S'inscrire", la méthode `inscrire()` est déclenchée.

```
<div class="mb-3 d-flex justify-content-around">
  <button type="button" class="btn btn-primary" (click)="inscrire()">
    <fa-icon [icon]="faCheckSquare"></fa-icon> S'inscrire
  </button>
  <a [routerLink]="['/Login']" class="btn btn-link">Se connecter</a>
</div>
```



Cette méthode assigne l'objet `adresse` à la propriété `adresse` de `entrepriseDto` pour créer un modèle complet. Elle appelle ensuite le service `EntrepriseService` via la méthode `sinscrire()`, qui envoie les données au backend sous forme d'une requête HTTP POST.

```
inscrire(): void { Show usages  ⬆ Berbigier thomas
  this.entrepriseDto.adresse = this.adresse;
  this.entrepriseService.sinscrire(this.entrepriseDto)
    .subscribe({
      next: (entrepriseDto) => {
        this.connectEntreprise();
      },
      error: (error) => {
        this.errorMsg = error.error.errors;
      }
    });
}
```

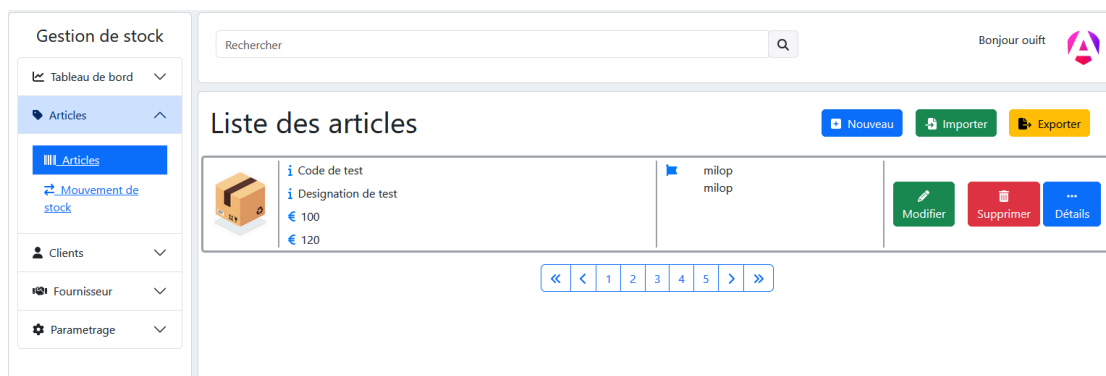
Une fois l'entreprise inscrite, la méthode `connectEntreprise()` est appelée. Cette méthode crée une requête d'authentification avec un mot de passe par défaut pour connecter automatiquement l'entreprise fraîchement créée. Le service `UserService` est utilisé pour envoyer la requête au backend.

Si la connexion est réussie, le token d'authentification est stocké dans le `localStorage` et l'utilisateur est redirigé vers une page de changement de mot de passe (Figure 4 en Annexe) grâce à `this.router.navigate('changerMotDePasse')`.

```
connectEntreprise(): void { Show usages  ⬆ Berbigier thomas
  const authenticationRequest: AuthenticationRequest = {
    login: this.entrepriseDto.email,
    password: 'som3R@nd0mP@$$word'
  };
  this.userService.login(authenticationRequest)
    .subscribe({
      next: (response) => {
        this.userService.setAccessToken(response);
        this.getUserByEmail(authenticationRequest.login);
        localStorage.setItem('origin', 'Inscription');
        this.router.navigate(['changerMotDePasse']);
      }
    });
}
```

## Gestion des articles et des commandes :

Une fois inscrit et connecté, l'utilisateur peut parcourir l'application et accéder à ses listes d'articles, de catégories, de clients ou de fournisseurs. Ces pages sont identiques au niveau de leur composition (Figure 5 en Annexe), mais pour l'exemple, dans la **page article** trois boutons sont présents dans l'en-tête, "Nouveau" pour créer un nouvel article, "Importer" et "Exporter".



La page utilise la grille **Bootstrap** pour une mise en page fluide, avec des classes telles que `col` et `row`. Cela permet une disposition bien structurée des éléments, assurant un bon alignement des composants même sur différentes tailles d'écrans.

Un bloc d'alerte `alert alert-danger` est présent dans le DOM pour afficher les messages d'erreurs en cas de problème avec la récupération des articles ou des actions utilisateur.

Chaque article est présenté grâce au composant `app-detail-article`, qui est répliqué pour chaque article disponible dans la base de données via une directive `*ngFor`. Lorsque la suppression d'un article est confirmée via l'événement (`suppressionResult`), la méthode `handleSuppression()` est appelée pour soit rafraîchir la liste des articles, soit afficher un message d'erreur en fonction de la réponse.

```
<div class="col m-3">
  <app-detail-article
    *ngFor="let article of listeArticle"
    [articleDto]="article"
    (suppressionResult)="handleSuppression($event)">
  </app-detail-article>
</div>
```

Lors de l'initialisation du composant avec la méthode `ngOnInit()`, une méthode `findListArticle()` est appelée pour récupérer les articles depuis le service `ArticleService`.

```
ngOnInit(): void { no usages  ⬆ Berbigier thomas
  this.findListArticle();
}

findListArticle(): void { Show usages  ⬆ Bert
  this.articleService.findAllArticles()
    .subscribe( articles => {
    this.listeArticle = articles;
  })
}

nouvelArticle(): void { Show usages  ⬆ Berbigi
  this.router.navigate(['NouvelArticle']);
}
```

Cette méthode utilise `subscribe()` pour traiter la réponse asynchrone et injecter la liste des articles dans le tableau `listeArticle`, qui est ensuite affiché dynamiquement à travers la directive `*ngFor` dans le template.

```
listeArticle: Array<ArticleDto> = [];
```

Le bouton pour créer un nouvel article utilise la méthode `nouvelArticle()` qui déclenche une navigation vers la page de création d'article à l'aide du service Router.

```
<div class="col-md-4 text-end">
  <app-boutton-action
    (clickEvent)="nouvelArticle()"
  ></app-boutton-action>
</div>
```

```
nouvelArticle(): void { Show usages  ⬆ Berbigi
  this.router.navigate(['NouvelArticle']);
}
```

La page **NouvelArticle** permet de recueillir les informations relatives à un article. La structure est organisée en utilisant la grille Bootstrap pour assurer une interface claire et responsive. Un bouton situé sur l'image présente pas défaut permet à l'utilisateur de télécharger une image pour l'article, avec un aperçu de l'image sélectionnée.



Un formulaire divisé en plusieurs champs de saisie permet de capturer des données comme le code de l'article, la désignation, le prix unitaire HT, le taux de TVA et la catégorie. Chaque champ utilise des composants de formulaire **HTML** standard (input et select) avec des placeholders pour guider l'utilisateur dans la saisie. Deux boutons sont placés en bas du formulaire, un bouton pour enregistrer l'article et un bouton pour annuler l'action. Ils sont stylisés avec des icônes **FontAwesome** pour améliorer l'ergonomie et l'interface utilisateur. Lorsque l'utilisateur soumet le formulaire pour enregistrer un article, la méthode `enregistrerArticle()` est appelée. Elle envoie les données du formulaire au backend via le service `ArticleService`.

```
enregistrerArticle(): void { Show usages  Berbigier thomas
  this.articleDto.categorie = this.categorieDto;
  this.articleService.enregistrerArticle(this.articleDto)
    .subscribe({
      next: (art) => {
        this.router.navigate(['Articles']);
      },
      error: (error) => {
        this.errorMsg = error.error.errors;
      },
      complete: () => {
      }
    });
}
```

## Conclusion :

Ce projet m'a permis de maîtriser l'approche modulaire d'**Angular**, qui encourage la réutilisation de composants dans toute l'application, assurant une cohérence visuelle et fonctionnelle. L'intégration de services pour la communication avec le backend, combinée à l'utilisation de directives **Angular** comme **ngIf** et **ngFor**, a renforcé ma capacité à créer des applications front-end dynamiques et évolutives. Enfin, l'adoption de **Bootstrap** et de **FontAwesome** a permis de garantir une interface à la fois intuitive et responsive, essentielle pour une bonne expérience utilisateur.

## 2. Précisez les moyens utilisés :

Dans ce projet, j'ai utilisé:

- **Windows 11** pour l'environnement de travail.
- **Angular 17.3** pour le développement de l'application frontend.
- **Bootstrap 5.3.2** pour le design et la mise en page responsive.
- **Fontawesome 6.6.0** pour les icônes.
- **OpenApi-Gen 0.51.0** pour la génération de services.
- **Node.js 20.10.0** et **npm 10.8.2** pour la gestion des packages et la compilation du projet.
- **Figma** pour la réalisation des maquettes.
- **IntelliJ IDEA** comme environnement de développement intégré.
- **Git** et **GitHub** pour le versioning et le dépôt distant.

## 3. Avec qui avez-vous travaillé ?

Pour ce projet j'ai travaillé seul

## 4. Contexte

# DOSSIER PROFESSIONNEL <sup>(DP)</sup>

Nom de l'entreprise, organisme ou association ▶      Projet personnel

Chantier, atelier, service ▶

Période d'exercice ▶ Du : 23/07/2024 au : 29/09/2024

## 5. Informations complémentaires *(facultatif)*

Ce projet reprend les quatre compétences de la première activité type du référentiel, soit:

1. Installer et configurer son environnement de travail en fonction du projet web ou web mobile,
2. Maquetter des interfaces utilisateur web ou web mobile
3. Réaliser des interfaces utilisateur statiques web ou web mobile
4. Développer la partie dynamique des interfaces utilisateur web ou web mobile

## Activité-type 2 Développer la partie back-end d'une application web ou web mobile sécurisée

### Exemple n° 1 - Gestion de stock

---

#### 1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

##### Introduction au projet :

Ce projet présente la partie back-end du projet précédent de gestion de stock. Il est basé sur le framework **Java Spring Boot** avec une API REST documentée via OpenAPI. Cette API sert de base pour générer les services et modèles nécessaires à l'application front-end. Le projet inclut une gestion sécurisée avec Spring Security et l'utilisation de JWT pour l'authentification.

J'ai créé le projet Spring Boot via start.spring.io en utilisant Maven avec Java 21 et Spring Boot 3.3.2.

##### Mise en place de la base de données relationnelle :

Dans ce projet, la conception de la base de données a commencé par la définition du modèle conceptuel de données, où j'ai identifié les principales entités et leurs relations.

Les entités incluent : Entreprise, Utilisateur, Rôles, Fournisseur, Client, Mouvement De Stock, Article et Catégorie. Ce modèle est essentiel pour représenter les données du système de manière cohérente et garantir une bonne gestion des relations entre les différentes tables.

Les cardinalités entre ces entités sont définies comme suit :

- Une entreprise peut avoir plusieurs utilisateurs, mais chaque utilisateur n'est associé qu'à une seule entreprise.
- Un utilisateur possède un rôle unique, tandis qu'un rôle peut être attribué à plusieurs utilisateurs.
- Un fournisseur et un client peuvent gérer plusieurs articles, mais chaque article est associé à un fournisseur ou un client unique.
- Une catégorie peut contenir plusieurs articles, mais un article n'appartient qu'à une seule catégorie.

Une fois ces relations établies, j'ai converti le modèle conceptuel en modèle logique de données. Chaque entité est devenue une table avec des clés primaires, et les relations sont implémentées via des clés étrangères :

- La table **Utilisateur** inclut une clé étrangère **identreprise**, faisant référence à l'entreprise associée.
- La table **Rôles** inclut une clé étrangère **idutilisateur**, faisant référence à la clé primaire de la table **Utilisateur**.
- La table **Fournisseur** récupère le champ **idarticle** clé étrangère qui référence la clé primaire de la table **Article**.
- La table **Client** récupère le champ **idarticle** clé étrangère qui référence la clé primaire de la table **Article**.
- La table **Article** a une clé étrangère **idcategorie** pour identifier la catégorie à laquelle appartient l'article.

Maintenant, je peux élaborer le diagramme de classe UML (Figure 6 en Annexe).

Pour le choix de la base de données, j'ai opté pour **MariaDB** car je l'ai déjà utilisée dans d'autres projets donc je la maîtrise relativement bien. De plus, sa proximité avec **MySQL** la rend facilement intégrable dans un environnement **Spring Boot** avec **Spring Data JPA**.

La configuration des accès à la base de données est effectuée via le fichier **application.yaml**, où j'ai défini un utilisateur dédié avec un mot de passe sécurisé.

```
spring:
  datasource:
    url: jdbc:mariadb://localhost:3306/gestionstock?createDatabaseIfNotExist=true
    username: administrateur_gs
    password: +Mb9UT!T]zbWU'+
    driver-class-name: org.mariadb.jdbc.Driver

  jpa:
    database-platform: org.hibernate.dialect.MariaDBDialect
    show-sql: false
    hibernate:
      ddl-auto: update
```



Bien que j'aie écrit une partie du code **SQL** pour l'exercice, j'utilise principalement **Spring Data JPA** couplé à **Hibernate** pour générer automatiquement les tables à partir des entités via les annotations JPA. **Hibernate** permet également de gérer les migrations de schéma grâce à la stratégie **update**, qui adapte la base de données à chaque déploiement, en fonction des changements d'entités.

```
CREATE TABLE `article` (  
  `id` int(11) NOT NULL,  
  `idcategorie` int(11) DEFAULT NULL,  
  `identreprise` int(11) DEFAULT NULL,  
  `prix_unitaire_ht` decimal(38,2) DEFAULT NULL,  
  `prix_unitaire_ttc` decimal(38,2) DEFAULT NULL,  
  `taux_tva` decimal(38,2) DEFAULT NULL,  
  `creation_date` datetime(6) NOT NULL,  
  `last_modified_date` datetime(6) DEFAULT NULL,  
  `code_article` varchar(255) DEFAULT NULL,  
  `designation` varchar(255) DEFAULT NULL,  
  `photo` varchar(255) DEFAULT NULL  
);
```

## Développer des composants d'accès aux données et de métier côté serveur :

Dans mon projet, j'ai utilisé une architecture à trois couches pour structurer l'accès et la manipulation des données dans la base de données, en séparant les responsabilités entre les entités **JPA** (modèles), les **DTO** (Data Transfer Objects), et les **services**. Cette approche permet une meilleure lisibilité et maintenabilité du code, tout en facilitant les tests et les évolutions.

J'utilise **JPA** (Java Persistence API) pour définir mes entités et mapper les objets **Java** aux tables de la base de données. Par exemple, l'entité **Categorie** est définie ainsi :

```
@Entity  
@Table(name = "categorie")  
public class Categorie extends AbstractEntity{  
  
    @Column(name = "code")  
    private String code;  
  
    @Column(name = "designation")  
    private String designation;  
  
    @Column(name = "identreprise")  
    private Integer idEntreprise;  
  
    @OneToMany(mappedBy = "categorie")  
    private List<Article> articles;  
}
```

L'annotation `@Entity` indique que cette classe est une entité **JPA**, et l'annotation `@Table` spécifie la table correspondante dans la base de données. Chaque champ est mappé à une colonne de la table `categorie`. Par exemple, `code` et `designation` représentent des colonnes dans la base de données. J'ai également une relation **OneToMany** avec la classe **Article**, indiquant qu'une catégorie peut avoir plusieurs articles.

Les **DTO** sont utilisés pour transférer les données entre les différentes couches de l'application sans exposer directement les entités. Cela permet de contrôler les informations envoyées via l'API et de respecter le principe d'encapsulation. J'ai défini des méthodes `fromEntity` et `toEntity` pour effectuer les conversions entre l'entité et le DTO.

```
// A partir d'une entité je construis un DTO
public static CategorieDto fromEntity(Categorie categorie) {
    if(categorie == null) {
        return null;
    }
    // fais un mapping de catégorie vers catégorieDTO
    return CategorieDto.builder()
        .id(categorie.getId())
        .code(categorie.getCode())
        .designation(categorie.getDesignation())
        .idEntreprise(categorie.getIdEntreprise())
        .build();
}

public static Categorie toEntity(CategorieDto categorieDto) {
    if(categorieDto == null) {
        return null;
    }
    Categorie categorie = new Categorie();
    categorie.setId(categorieDto.getId());
    categorie.setCode(categorieDto.getCode());
    categorie.setDesignation(categorieDto.getDesignation());
    categorie.setIdEntreprise(categorieDto.getIdEntreprise());
    return categorie;
}
```

La couche **service** gère la logique métier et fait le lien entre les contrôleurs et le dépôt (repository). Par exemple, la classe **CategorieServiceImpl** implémente l'interface **CategorieService** et utilise le **CategorieRepository** pour interagir avec la base de données :

```
@Service  ↳ Berbigier thomas
@Slf4j
public class CategorieServiceImpl implements CategorieService {

    private final CategorieRepository categorieRepository;  6 usages
    private final ArticleRepository articleRepository;  2 usages

    @Autowired  ↳ Berbigier thomas
    public CategorieServiceImpl(
        CategorieRepository categorieRepository, ArticleRepository articleRepository
    ) {...}

    @Override  ↳ Berbigier thomas
    public CategorieDto save(CategorieDto categorieDto) {

        List<String> errors = CategorieValidator.validate(categorieDto);
        if (!errors.isEmpty()) {
            log.error("Catégorie invalide {}", categorieDto);
            throw new InvalidEntityException("La catégorie n'est pas valide", ErrorCodes.CATEGORY_NOT_VALID, errors);
        }

        return CategorieDto.fromEntity(
            categorieRepository.save(
                CategorieDto.toEntity(categorieDto)
            )
        );
    }
}
```

Le service implémente plusieurs méthodes pour la gestion des catégories ([save](#), [findByCode](#), [findById](#), [findAll](#), [delete](#)). Avant chaque opération de sauvegarde, une validation est effectuée, et les erreurs sont gérées via des exceptions personnalisées comme **InvalidEntityException**.

```
public class CategorieValidator {  2 usages  ↳ Berbigier thomas *

    public static List<String> validate(CategorieDto categorieDto) {  ↳ Berbigier thomas *
        List<String> errors = new ArrayList<>();

        if (categorieDto == null || !StringUtils.hasLength(categorieDto.getCode())) {
            errors.add("Veuillez renseigner le code de la catégorie");
        }

        return errors;
    }
}
```

L'accès aux données se fait via le repository qui étend l'interface **JpaRepository**, offrant des méthodes **CRUD** sur l'entité **Categorie**. Cette interface permet d'effectuer des requêtes en base de données sans avoir à écrire de **SQL** explicite. Le framework **Spring Data JPA** se charge de générer les requêtes en fonction des méthodes définies.

```
public interface CategorieRepository extends JpaRepository<Categorie, Integer> {  
  
    Optional<Categorie> findCategorieByCode(String code); 1 usage  ⤴ Berbigier thomas  
  
}
```

Le contrôleur **CategorieController** expose les différentes opérations disponibles pour l'entité **Categorie** via des points d'accès HTTP. Ces opérations incluent la création, la mise à jour, la suppression, ainsi que la recherche de catégories par ID ou par code. Par exemple, la méthode **findById** permet de récupérer une catégorie en fonction de son identifiant, tandis que **delete** permet de supprimer une catégorie si celle-ci n'est pas associée à d'autres entités comme des articles.

```
@RestController ⤴ Berbigier thomas  
public class CategorieController implements CategorieApi {  
  
    @Autowired  
    private CategorieService categorieService;  
  
    public CategorieController(CategorieService categorieService) { this.categorieService = categorieService; }  
  
    @Override  ⤴ Berbigier thomas  
    public CategorieDto save(CategorieDto categorieDto) { return categorieService.save(categorieDto); }  
  
    @Override  ⤴ Berbigier thomas  
    public CategorieDto findByCode(String codeCategorie) {  
        return categorieService.findByCode(codeCategorie);  
    }  
  
    @Override  ⤴ Berbigier thomas  
    public CategorieDto findById(Integer idCategorie) {  
        return categorieService.findById(idCategorie);  
    }  
  
    @Override  ⤴ Berbigier thomas  
    public List<CategorieDto> findAll() { return categorieService.findAll(); }  
  
    @Override  ⤴ Berbigier thomas  
    public void delete(Integer id) { categorieService.delete(id); }  
}
```

## Documentation de l'API :

Afin de rendre l'API REST plus facile à consommer et à documenter, j'ai intégré la spécification **OpenAPI** dans mon projet à l'aide du plugin **springdoc-openapi**.

```
<dependency>
  <groupId>org.springdoc</groupId>
  <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
  <version>2.3.0</version>
</dependency>
```

Grâce aux annotations sur les méthodes du controller, chaque route, ses paramètres, et ses réponses sont documentés et accessibles via une interface Swagger, accessible en local.

The screenshot shows the Swagger UI for the 'Gestion de Stock API'. At the top, it displays 'v1' and 'OAS 3.0'. Below this, there's a 'Servers' section with a dropdown menu showing 'http://localhost:8081 - Generated server url' and an 'Authorize' button. The main content area lists several API endpoints under different categories: 'articles', 'authentication', 'categories', and 'clients'. The 'categories' section is expanded, showing five endpoints: a DELETE endpoint for deleting a category, and four GET endpoints for searching categories by ID, by code, and for a list of all categories, plus a POST endpoint for creating a new category. Each endpoint is color-coded (red for DELETE, blue for GET, green for POST) and includes a lock icon and a dropdown arrow.

**Gestion de Stock API** <sup>v1</sup> <sup>OAS 3.0</sup>  
/v3/api-docs

**Servers**  
http://localhost:8081 - Generated server url Authorize

**articles** API pour la gestion des articles ▼

**authentication** ▼

**categories** API pour la gestion des catégories ^

- DELETE** /gestiondestock/v1/categories/delete/{idCategorie} Supprimer une catégorie 🔒 ▼
- GET** /gestiondestock/v1/categories/id/{idCategorie} Rechercher une catégorie par ID 🔒 ▼
- GET** /gestiondestock/v1/categories/code/{codeCategorie} Rechercher une catégorie par CODE 🔒 ▼
- GET** /gestiondestock/v1/categories/all Renvoi la liste des catégories 🔒 ▼
- POST** /gestiondestock/v1/categories/create Enregistrer une catégorie 🔒 ▼

**clients** API pour la gestion des clients ▼

L'API est sécurisée avec **JWT** (JSON Web Tokens), et les spécifications OpenAPI tiennent compte de cette sécurité grâce aux annotations `@SecurityScheme` et `@SecurityRequirement`. Ces annotations permettent de configurer une authentification par "Bearer token", garantissant ainsi une protection adéquate des endpoints sensibles.

```
@Configuration
@OpenAPIDefinition(info = @Info(title = "Gestion de Stock API", version = "v1"),
    security = @SecurityRequirement(name = "bearerAuth"))
@SecurityScheme(
    name = "bearerAuth",
    type = SecuritySchemeType.HTTP,
    scheme = "bearer",
    bearerFormat = "JWT"
)
public class OpenApiConfiguration {

    @Bean
    public OpenAPI customOpenAPI() {
        return new OpenAPI()
            .components(new Components()
                .addSecuritySchemes(key: "bearerAuth",
                    new io.swagger.v3.oas.models.security.SecurityScheme()
                        .type(io.swagger.v3.oas.models.security.SecurityScheme.Type.HTTP)
                        .scheme("bearer")
                        .bearerFormat("JWT")))
                .addSecurityItem(new io.swagger.v3.oas.models.security.SecurityRequirement().addList(name: "bearerAuth")));
    }
}
```

De plus, j'ai utilisé le plugin Maven `openapi-generator-maven-plugin` pour générer automatiquement les **services** et **modèles** nécessaires à l'application front-end. En se basant sur le fichier OpenAPI (généré sous forme YAML), ce plugin permet d'obtenir un code client complet qui peut être utilisé directement dans mon application front-end. Cela assure une synchronisation parfaite entre les spécifications de l'API et les composants front-end. Le fichier de configuration Maven inclut les éléments suivants pour le

```
<plugin>
  <groupId>org.openapitools</groupId>
  <artifactId>openapi-generator-maven-plugin</artifactId>
  <!-- RELEASE_VERSION -->
  <version>7.7.0</version>
  <!-- /RELEASE_VERSION -->
  <executions>
    <execution>
      <goals>
        <goal>generate</goal>
      </goals>
      <configuration>
        <inputSpec>http://localhost:8081/v3/api-docs.yaml</inputSpec>
        <generatorName>java</generatorName>
        <configOptions>
          <sourceFolder>src/gen/java/main</sourceFolder>
        </configOptions>
        <output>${project.build.directory}</output>
      </configuration>
    </execution>
  </executions>
</plugin>
```

générateur OpenAPI :

L'exécution de ce plugin récupère la documentation OpenAPI à partir de l'endpoint généré par `springdoc-openapi /v3/api-docs.yaml` et génère automatiquement le code client pour l'application front-end.

## Sécurité :

Dans ce projet, la sécurité est un élément central, gérée à l'aide de Spring Security avec une approche basée sur des **JWT** (JSON Web Tokens). Cette configuration permet de sécuriser l'accès aux différentes ressources de l'API, tout en facilitant l'authentification des utilisateurs via des tokens.

Tout d'abord, l'API utilise une configuration de sécurité via une classe **SecurityConfiguration**. Les points d'accès non sécurisés, comme l'authentification, l'inscription, ou la documentation OpenAPI, sont explicitement autorisés via les annotations. Les autres points d'entrée nécessitent que l'utilisateur soit authentifié, c'est-à-dire qu'un JWT valide soit fourni dans l'en-tête de la requête.

```
@Configuration
@EnableWebSecurity
public class SecurityConfiguration {

    @Autowired
    private final ApplicationUserService applicationUserService;

    @Autowired
    private ApplicationRequestFilter applicationRequestFilter;

    public SecurityConfiguration(ApplicationUserService applicationUserService) {
        this.applicationUserService = applicationUserService;
    }

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http, ApplicationRequestFilter applicationRequestFilter) throws Exception {
        http
            .csrf(AbstractHttpConfigurer::disable)
            .sessionManagement(session ->
                session.sessionCreationPolicy(STATELESS)
            )
            .authorizeHttpRequests(authorize ->
                authorize
                    .requestMatchers(
                        "/gestiondestock/v1/auth/authenticate",
                        "/gestiondestock/v1/entreprises/create",
                    )
            )
    }
}
```

Les utilisateurs sont gérés via un service dédié appelé **ApplicationUserDetailsService**, qui charge les détails de l'utilisateur à partir de la base de données, en associant chaque utilisateur à ses rôles et à l'entreprise à laquelle il appartient. Cela permet de garantir que chaque utilisateur n'a accès qu'aux ressources de son entreprise.

```
@Service
public class ApplicationUserDetailsService implements UserDetailsService {

    @Autowired
    private UtilisateurService service;

    @Override
    public UserDetails loadUserByUsername(String email) throws UsernameNotFoundException {
        UtilisateurDto utilisateur = service.findByEmail(email);

        List<SimpleGrantedAuthority> authorities = new ArrayList<>();
        utilisateur.getRoles().forEach(role -> authorities.add(new SimpleGrantedAuthority(role.getRoleName())));

        return new ExtendedUser(utilisateur.getEmail(), utilisateur.getMotDePasse(), utilisateur.getEntreprise().getId(), authorities);
    }
}
```

La génération et la validation des JWT sont gérées par la classe **JwtUtil**. Ce composant est responsable de la création du token, qui inclut non seulement les informations de base de l'utilisateur (comme son nom d'utilisateur), mais aussi l'identifiant de l'entreprise à laquelle il est rattaché. Le JWT est ensuite utilisé pour authentifier les requêtes, en s'assurant que le token n'a pas expiré et correspond bien aux informations de l'utilisateur.

```
private String createToken(Map<String, Object> claims, ExtendedUser userDetails) {

    return Jwts.builder().setClaims(claims)
        .setSubject(userDetails.getUsername())
        .setIssuedAt(new Date(System.currentTimeMillis()))
        .setExpiration(new Date(System.currentTimeMillis() + 1000 * 60 * 60 * 10))
        .claim("idEntreprise", userDetails.getIdEntreprise().toString())
        .signWith(SignatureAlgorithm.HS256, SECRET_KEY).compact();
}
```



Enfin, une fois le token vérifié, un filtre personnalisé **ApplicationRequestFilter** s'assure que l'authentification est appliquée pour chaque requête, en injectant les informations de l'utilisateur et de l'entreprise dans le contexte de sécurité de Spring. Cela garantit que toutes les actions sont effectuées sous l'identité vérifiée de l'utilisateur, et que les contrôles d'accès sont strictement respectés.

```
@Override no usages Berbigier thomas
protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain chain)
    throws ServletException, IOException {

    final String authHeader = request.getHeader("Authorization");
    String userEmail = null;
    String jwt = null;
    String idEntreprise = null;

    if(authHeader != null && authHeader.startsWith("Bearer ")) {
        jwt = authHeader.substring(beginIndex: 7);
        userEmail = jwtUtil.extractUsername(jwt);
        idEntreprise = jwtUtil.extractIdEntreprise(jwt);
    }
}
```

## Documenter le déploiement d'une application dynamique web ou web mobile :

### • Déploiement en local

Pour permettre le déploiement de l'application en local, j'ai créé un fichier **Readme.md** contenant les démarches à suivre:

#### 1. Prérequis :

- a. **Java 21** : Vérifiez que Java est correctement installé en exécutant la commande suivante dans votre terminal : `java -version`
- b. **Maven** : Vérifiez l'installation de Maven avec la commande suivante : `mvn -version`
- c. **MariaDB** : Vous devez avoir une instance locale de MariaDB en cours d'exécution. Vous pouvez télécharger et installer MariaDB depuis leur site officiel.

#### 2. Cloner le dépôt Git :

- a. Clonez le dépôt GitHub de l'application sur votre machine locale en utilisant la commande suivante : `git clone https://github.com/ThomasBerbigier/Gestion_de_stock_backend.git`

### 3. Configurer la base de données MariaDB :

- a. Créez une base de données MariaDB pour l'application. Vous pouvez utiliser un outil comme **phpMyAdmin** ou la ligne de commande MariaDB pour ce faire :

```
1 CREATE DATABASE gestion_de_stock;
```

- b. configurez les informations d'accès à la base de données dans le fichier **application.yml** qui se trouve dans le répertoire **src/main/resources/**. Voici un exemple de configuration :

```
spring:
  datasource:
    url: jdbc:mariadb://localhost:3306/gestion_de_stock
    username: root
    password: mot_de_passe
  jpa:
    hibernate:
      ddl-auto: update
    show-sql: true
```

### 4. Construire le projet avec Maven :

- a. Accédez au répertoire du projet cloné, puis exécutez Maven pour construire l'application :

```
mvn clean install
```

### 5. Lancer l'application :

- a. Une fois le projet construit, vous pouvez démarrer l'application en utilisant la commande suivante :

```
mvn spring-boot:run
```

### 6. Accéder à l'API :

- a. Une fois l'application démarrée, vous pouvez accéder à l'API REST via votre navigateur ou un outil comme **Postman** à l'adresse suivante :

```
http://localhost:8080/api/v1
```

## 7. Documentation de l'API

- a. La documentation de l'API générée via **Springdoc-OpenAPI** peut être consultée en accédant à l'URL suivante :

```
http://localhost:8080/swagger-ui.html
```

### • Déploiement en ligne :

Pour le déploiement de l'application en ligne, j'ai décidé d'utiliser Heroku comme plateforme de déploiement pour mon application.

#### 1. Création d'un compte Heroku et installation de l'outil Heroku CLI :

- a. Premièrement, je me suis rendu sur [Heroku](#) et j'ai créé un compte.
- b. Ensuite, j'ai installé l'outil CLI de Heroku en suivant les instructions officielles.
- c. Après l'installation, j'ai utilisé la commande suivante pour me connecter à mon compte Heroku via la CLI :

```
PS C:\Users\berto\OneDrive\Bureau\Formation\gestionDeStock> heroku login
heroku: Press any key to open up the browser to login or q to exit:
Opening browser to https://cli-auth.heroku.com/auth/cli/browser/c1dcccce-
heroku: Waiting for login...
Error: spawn cmd ENOENT
Code: ENOENT
Logging in... done
Logged in as thomas.berbigier@gmail.com
```

Cette commande m'a permis de m'authentifier en ouvrant une fenêtre de navigateur pour saisir mes identifiants Heroku.

#### 2. Création d'une nouvelle application Heroku :

- a. J'ai créé une nouvelle application Heroku en ligne de commande avec :

```
PS C:\Users\berto\OneDrive\Bureau\Formation\gestionDeStock> heroku create api-rest-gestion-de-stock
» Warning: heroku update available from 8.11.5 to 9.3.0.
Creating • api-rest-gestion-de-stock... done
https://api-rest-gestion-de-stock-61bd1d441bd8.herokuapp.com/ | https://git.heroku.com/api-rest-gestion-de-stock.git
```

### 3. Ajout de l'add-on PostgreSQL sur Heroku :

- a. Puisque mon projet nécessite une base de données, j'ai ajouté l'add-on PostgreSQL avec la commande suivante :
- ```
heroku addons:create heroku-postgresql:hobby-dev
```
- Cela m'a fourni une base de données PostgreSQL sur Heroku avec les variables d'environnement automatiquement créées.

### 4. Récupération et configuration des variables d'environnement PostgreSQL :

- a. Pour voir les informations de connexion PostgreSQL, j'ai utilisé la commande suivante :

```
heroku config
```

### 5. Configuration des variables d'environnement spécifiques dans Heroku :

- a. J'ai configuré les variables d'environnement nécessaires à l'application comme la clé API de Flickr et les autres paramètres avec les commandes suivantes :

```
heroku config:set FLICKR_API_KEY=ma_cle_api  
heroku config:set FLICKR_API_SECRET=mon_secret_api  
heroku config:set MAVEN_CUSTOM_OPTS=-Pprod
```

### 6. Modification du fichier **application.yml** pour utiliser PostgreSQL :

- a. Dans mon fichier **application.yml**, j'ai configuré l'application pour qu'elle utilise PostgreSQL en production. J'ai remplacé les informations de connexion par les variables fournies par Heroku :

```
spring:  
  datasource:  
    url: ${JDBC_DATABASE_URL}  
    username: ${JDBC_DATABASE_USERNAME}  
    password: ${JDBC_DATABASE_PASSWORD}  
    driver-class-name: org.postgresql.Driver  
  
  jpa:  
    database-platform: org.hibernate.dialect.PostgreSQLDialect  
    hibernate:  
      ddl-auto: update  
      show-sql: false
```

## 7. Préparation du projet pour le déploiement sur Heroku :

- a. J'ai créé un fichier **Procfile** à la racine de mon projet avec le contenu suivant pour indiquer à Heroku comment démarrer mon application :

```
1 web: java -jar target/gestionDeStock-0.0.1-SNAPSHOT.jar
```

## 8. Déploiement du projet sur Heroku :

- a. Après avoir comité mes changements sur Git, j'ai déployé l'application sur Heroku en

```
git add .
git commit -m "modification application-prod"

git push heroku main
```

suivant ces étapes :

- b. Heroku a automatiquement construit et déployé l'application. J'ai pu suivre le processus de construction et de déploiement dans le terminal.

```
remote:      [INFO] BUILD SUCCESS
remote:      [INFO] -----
remote:      [INFO] Total time: 30.777 s
remote:      [INFO] Finished at: 2024-09-29T18:55:15Z
remote:      [INFO] -----
remote: ----> Discovering process types
remote:   Procfile declares types   -> (none)
remote:   Default types for buildpack -> web
remote:
remote: ----> Compressing...
remote:   Done: 132.1M
remote: ----> Launching...
remote:   Released v13
remote:   https://api-rest-gestion-de-stock-61bd1d441bd8.herokuapp.com/ deployed to Heroku
remote:
remote: This app is using the Heroku-22 stack, however a newer stack is available.
remote: To upgrade to Heroku-24, see:
remote: https://devcenter.heroku.com/articles/upgrading-to-the-latest-stack
remote:
remote: Verifying deploy... done.
To https://git.heroku.com/api-rest-gestion-de-stock.git
 * [new branch]      main -> main
```

## 9. Vérification du déploiement :

- a. Une fois le déploiement terminé, j'ai vérifié que mon application était en ligne en utilisant la commande suivante pour ouvrir l'application dans le navigateur : `heroku open`

## 10. Gestion et suivi de l'application en production :

- a. Pour surveiller les logs en direct et diagnostiquer les éventuels problèmes, j'ai utilisé :

```
PS C:\Users\berto\OneDrive\Bureau\Formation\gestionDeStock> heroku logs --tail
» Warning: heroku update available from 8.7.1 to 9.3.0.
2024-09-29T17:36:49.947978+00:00 app[api]: Initial release by user thomas.berbigier@gmail.com
2024-09-29T17:36:49.947978+00:00 app[api]: Release v1 created by user thomas.berbigier@gmail.com
2024-09-29T17:36:50.088165+00:00 app[api]: Release v2 created by user thomas.berbigier@gmail.com
2024-09-29T17:36:50.088165+00:00 app[api]: Enable Logplex by user thomas.berbigier@gmail.com
2024-09-29T17:44:05.553779+00:00 app[api]: Release v3 created by user heroku-postgresql@addons.heroku.com
```

## Conclusion :

Pour conclure, ce projet illustre la mise en place d'une architecture ordonnée et sécurisée pour une application de gestion de stock, en s'appuyant sur les technologies Spring Boot, Spring Security et JWT. Grâce à une API REST structurée, documentée avec OpenAPI, et une gestion de l'authentification et des autorisations, l'application garantit un accès sécurisé aux données sensibles présentes en base de données.

Les différentes entités, telles que les catégories, les utilisateurs, les entreprises, et les articles sont gérées à travers des couches de service bien définies, renforçant la modularité et la maintenabilité du code. La génération automatique des clients front-end à partir de la spécification OpenAPI simplifie l'intégration avec d'autres systèmes, en assurant une cohérence entre le back-end et le front-end.

Concernant le déploiement, j'ai documenté de manière complète les procédures nécessaires pour déployer l'application **en local**, que ce soit à partir de GitHub ou directement sur une machine locale, en fournissant toutes les étapes nécessaires pour configurer l'environnement. En outre, j'ai également couvert le **déploiement en ligne** via Heroku, où j'ai expliqué la création du compte, l'ajout de l'addon PostgreSQL, ainsi que la configuration des variables d'environnement et du fichier **Procfile** pour un démarrage fluide de l'application.

# DOSSIER PROFESSIONNEL <sup>(DP)</sup>

Cette double documentation, locale et en ligne, assure que l'application peut être facilement réutilisée ou adaptée dans divers environnements de développement et de production.

## 2. Précisez les moyens utilisés :

Dans ce projet, j'ai utilisé les outils et technologies suivants :

- **Windows 11** pour mon environnement de travail.
- **Java 21**, la dernière version stable de Java.
- **Spring Boot 3.3.2** pour faciliter le développement rapide d'applications Java, notamment avec son intégration Spring Security et Spring Data JPA.
- **Maven** pour la gestion des dépendances et la construction du projet.
- **Springdoc-OpenAPI 2.3.0**, pour documenter automatiquement l'API REST avec une interface Swagger.
- **OpenAPI Generator 7.7.0**, pour générer des clients et des services à partir de la spécification OpenAPI.
- **MariaDB 10.4.32**, pour la gestion de la base de données relationnelle de l'application.
- **IntelliJ IDEA**, mon environnement de développement intégré.
- **Git et GitHub**, pour la gestion du contrôle de version du code source et le dépôt distant.
- **Heroku**, pour le déploiement de l'application en ligne.

## 3. Avec qui avez-vous travaillé ?

J'ai travaillé seul dans ce projet.

## 4. Contexte

Nom de l'entreprise, organisme ou association ▶      **Projet personnel**

Chantier, atelier, service ▶

Période d'exercice ▶ Du : **23/07/2024**      au : **29/09/2024**

### 5. Informations complémentaires *(facultatif)*

Ce projet reprend les quatre compétences de la seconde activité type du référentiel, soit:

1. Mettre en place une base de données relationnelle
2. Développer des composants d'accès aux données SQL et NoSQL
3. Développer des composants métier côté serveur
4. Documenter le déploiement d'une application dynamique web ou web mobile



## Activité-type 2

Développer la partie back-end d'une application web ou web mobile sécurisée

Exemple n° 2 - Gestion Fournitures

### 1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

#### Introduction au projet:

Dans le cadre de ce projet, j'ai utilisé plusieurs technologies pour développer une application de gestion des fournitures avec une base de données NoSQL. J'ai choisi **Java** comme langage principal pour le développement de l'application, associé à **MongoDB** pour le stockage des données en NoSQL, et **Docker** pour assurer l'exécution de MongoDB de manière isolée et portable. Le projet est structuré de manière modulaire, avec **Maven** comme outil de gestion des dépendances.

Pour interagir avec MongoDB dans mon projet Java, j'ai ajouté la dépendance **MongoDB Java Driver** dans le fichier **pom.xml**. Cette dépendance permet d'établir une connexion entre l'application Java et MongoDB, tout en facilitant l'exécution des opérations CRUD (Create, Read, Update, Delete) sur la base de données.

```
<!-- MongoDB Java Driver -->
<dependency>
  <groupId>org.mongodb</groupId>
  <artifactId>mongodb-driver-sync</artifactId>
  <version>4.9.0</version>
</dependency>
```

Ce driver fournit une API Java permettant de manipuler directement les documents stockés dans MongoDB de manière efficace et sécurisée.

## Connexion à MongoDB :

Pour établir la connexion avec MongoDB, j'ai utilisé la classe **MongoClients** fournie par le driver. La connexion est configurée avec des informations d'authentification (nom d'utilisateur et mot de passe) et l'URL du serveur MongoDB, exécuté dans Docker. Cette approche me permet de lancer MongoDB dans un environnement isolé, sans conflit avec d'autres services sur ma machine locale.

```
// Collection MongoDB où sont stockées les fournitures
private final MongoClient<Document> collection; 5 usages

// Initialisation de la connexion à MongoDB et récupération de la collection "supplies"
public SupplyDao() { 1 usage
    var mongoClient = MongoClient.create("mongodb://utilisateur:motdepasse@localhost:8090");
    MongoDB database = mongoClient.getDatabase(s: "fournituresDB");
    collection = database.getCollection(s: "supplies");
}
```

Cette configuration me permet de me connecter à la base de données **fournituresDB** et d'accéder à la collection **supplies**, où sont stockées les différentes fournitures. Grâce à cette connexion, je peux effectuer des opérations CRUD sur les documents MongoDB.

Le fichier **docker-compose.yml** montre la configuration que j'ai utilisé pour démarrer MongoDB avec les bonnes options d'authentification et de port. Cela permet de facilement déployer MongoDB sur différentes machines sans installation directe.

```
1 version: '3.8'
2 services:
3   mongodb:
4     image: mongo:7.0.8
5     container_name: mongo_db
6     restart: always
7     ports:
8       - "27017:27017"
9     environment:
10      MONGO_INITDB_ROOT_USERNAME: utilisateur
11      MONGO_INITDB_ROOT_PASSWORD: motdepasse
12     volumes:
13       - mongo-data:/data/db
14
15 volumes:
16   mongo-data:
```

### Gestion des données :

Pour faciliter la gestion et la vérification des données dans MongoDB, j'ai utilisé **MongoDB Compass**, une interface graphique intuitive qui me permet de visualiser et d'interagir avec la base de données. MongoDB Compass offre une vue claire des collections et des documents, ce qui est particulièrement utile pour vérifier manuellement les résultats des opérations effectuées par l'application. Après chaque opération CRUD, je peux rapidement vérifier que les documents ont été correctement créés, modifiés ou supprimés.

Pour la gestion des entités de mon application, j'ai défini un modèle objet **Supply** (Figure 7 en Annexe), qui représente une fourniture dans la base de données. Ce modèle inclut des propriétés telles que le nom de la fourniture, la catégorie, la quantité en stock et son état.

### Composant d'accès aux données NoSQL et logique métier :

J'ai développé un composant d'accès aux données (DAO) appelé **SupplyDao**, qui encapsule toutes les opérations CRUD sur les fournitures dans MongoDB. Ce composant assure une interaction transparente avec la base de données NoSQL, en manipulant les documents MongoDB sans se soucier de la logique métier.

En complément, j'ai développé un service **SupplyService**, qui fait appel aux méthodes du DAO pour exécuter les opérations CRUD tout en appliquant des règles métiers spécifiques. Cette approche permet de séparer la logique métier (comme la vérification des stocks) de la gestion technique des données.

- **CREATE : Insertion d'une fourniture**

Le processus de création d'une fourniture commence dans le **DAO**, qui se charge de l'insertion proprement dite. La méthode `addSupply` prend un objet **Supply**, le convertit en document BSON, puis l'insère dans la collection **supplies** de la base de données MongoDB.

```
/**
 * Ajoute une nouvelle fourniture dans la collection MongoDB.
 *
 * @param supply L'objet fourniture à insérer.
 */
public void addSupply(Supply supply) { 1 usage
    Document document = new Document("name", supply.getName())
        .append("category", supply.getCategory())
        .append("quantity", supply.getQuantity())
        .append("inStock", supply.isInStock());
    collection.insertOne(document);
}
```

Le **service**, quant à lui, encapsule la logique métier liée à l'insertion. Il vérifie la quantité pour déterminer si la fourniture est en stock avant d'appeler le DAO pour effectuer l'insertion dans la base de données.

```
// DAO pour accéder aux données des fournitures dans MongoDB
private SupplyDao supplyDao = new SupplyDao(); 4 usages

/**
 * Ajoute une nouvelle fourniture en vérifiant si elle est en stock.
 *
 * @param name Le nom de la fourniture.
 * @param category La catégorie de la fourniture.
 * @param quantity La quantité initiale de la fourniture.
 */
public void addNewSupply(String name, String category, int quantity) {
    boolean inStock = quantity > 0;
    Supply supply = new Supply(name, category, quantity, inStock);
    supplyDao.addSupply(supply);
    System.out.println("Nouvelle fourniture ajoutée !");
}
```

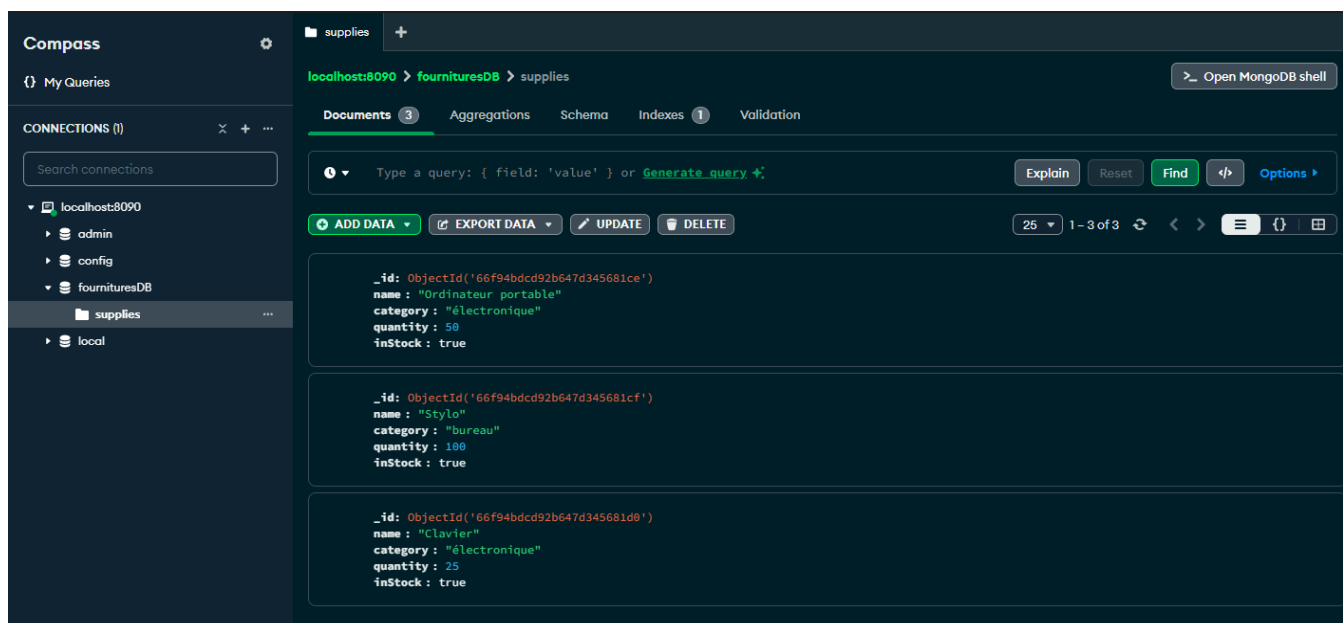
L'opération **Create** est déclenchée depuis la classe **SupplyApp**, qui sert de point d'entrée à l'application. C'est dans **SupplyApp** que l'appel à la méthode **addNewSupply** du service est effectué pour ajouter une nouvelle fourniture.

```
public class SupplyApp {  
    public static void main(String[] args) {  
        SupplyService service = new SupplyService();  
  
        // Ajout de trois nouvelles fournitures  
        service.addNewSupply(name: "Ordinateur portable", category: "électronique", quantity: 50);  
        service.addNewSupply(name: "Stylo", category: "bureau", quantity: 100);  
        service.addNewSupply(name: "Clavier", category: "électronique", quantity: 25);  
  
        // Affichage des fournitures ajoutées  
        System.out.println("Les fournitures ont été ajoutées avec succès !");  
    }  
}
```

Lors du démarrage de l'application, la fourniture est ajoutée dans MongoDB, et un message de confirmation s'affiche dans le terminal : **"Les fournitures ont été ajoutées avec succès !"**

```
Nouvelle fourniture ajoutée !  
Nouvelle fourniture ajoutée !  
Nouvelle fourniture ajoutée !  
Les fournitures ont été ajoutées avec succès !
```

Je peux ensuite vérifier dans **MongoDB Compass** que l'ajout a été correctement effectué en visualisant la collection **supplies**.



- **READ : Lecture d'une fourniture**

Le **DAO** effectue la recherche dans la collection MongoDB et retourne un objet **Supply**.

```
/**
 * Récupère une fourniture à partir de son nom.
 *
 * @param name Le nom de la fourniture à rechercher.
 * @return L'objet Supply correspondant ou null si non trouvé.
 */
public Supply getSupplyByName(String name) { 1 usage
    Document doc = collection.find(eq( fieldName: "name", name)).first();
    if (doc != null) {
        return new Supply(
            doc.getString( key: "name"),
            doc.getString( key: "category"),
            doc.getInteger( key: "quantity"),
            doc.getBoolean( key: "inStock")
        );
    }
    return null;
}
```

Le **service** vérifie si la fourniture existe avant d'afficher les détails. Cette vérification garantit que l'application ne tente pas d'afficher une fourniture inexistante.

```
/**
 * Affiche les détails d'une fourniture par son nom.
 *
 * @param name Le nom de la fourniture à afficher.
 */
public void displaySupply(String name) { 1 usage
    Supply supply = supplyDao.getSupplyByName(name);
    if (supply != null) {
        System.out.println("Fourniture : " + supply.getName() + ", Quantité : " + supply.getQuantity());
    } else {
        System.out.println("Fourniture introuvable.");
    }
}
```

La méthode `displaySupply` du service est appelée dans **SupplyApp**, déclenchant la lecture de la fourniture dans MongoDB via le DAO.

```
SupplyService service = new SupplyService();

// Lecture des fournitures ajoutées
service.displaySupply(name: "Ordinateur portable");
service.displaySupply(name: "Stylo");
service.displaySupply(name: "Clavier");
```

Au démarrage de l'application, les informations de la fourniture s'affichent correctement dans le terminal.

```
Fourniture : Ordinateur portable, Quantité : 50
Fourniture : Stylo, Quantité : 100
Fourniture : Clavier, Quantité : 25

Process finished with exit code 0
```

- **UPDATE : Mise à jour d'une fourniture**

Le **DAO** effectue une mise à jour partielle du document dans MongoDB.

```
/**
 * Met à jour la quantité d'une fourniture dans la base de données.
 *
 * @param name Le nom de la fourniture à mettre à jour.
 * @param newQuantity La nouvelle quantité de la fourniture.
 */
public void updateSupplyQuantity(String name, int newQuantity) { 1 usage
    collection.updateOne(eq(fieldName: "name", name), new Document("$set", new Document("quantity", newQuantity)));
}
```

Le **service** vérifie que la fourniture existe avant d'appeler le DAO et affiche un message de confirmation après la mise à jour. Cela garantit que seules les fournitures existantes sont modifiées.

```
/**
 * Met à jour la quantité d'une fourniture existante.
 *
 * @param name Le nom de la fourniture à mettre à jour.
 * @param newQuantity La nouvelle quantité de la fourniture.
 */
public void updateSupplyQuantity(String name, int newQuantity) {
    supplyDao.updateSupplyQuantity(name, newQuantity);
    System.out.println("Quantité mise à jour !");
}
```

Dans **SupplyApp**, l'opération de mise à jour est lancée pour modifier la quantité d'une fourniture existante.

```
public class SupplyApp {

    public static void main(String[] args) {
        SupplyService service = new SupplyService();

        // Mise à jour des quantités
        service.updateSupplyQuantity(name: "Ordinateur portable", newQuantity: 40); // Met à jour la quantité à 40
        service.updateSupplyQuantity(name: "Stylo", newQuantity: 150); // Met à jour la quantité à 150
        service.updateSupplyQuantity(name: "Clavier", newQuantity: 20); // Met à jour la quantité à 20

        // Affichage après mise à jour
        service.displaySupply(name: "Ordinateur portable");
        service.displaySupply(name: "Stylo");
        service.displaySupply(name: "Clavier");
    }
}
```

Au démarrage de l'application, les informations s'affichent correctement dans le terminal.

```
Quantité mise à jour !
Quantité mise à jour !
Quantité mise à jour !
Fourniture : Ordinateur portable, Quantité : 40
Fourniture : Stylo, Quantité : 150
Fourniture : Clavier, Quantité : 20
```



Je peux ensuite vérifier dans **MongoDB Compass** que la mise à jour a été effectuée en visualisant la collection **supplies**.

```
_id: ObjectId('66f94bcd92b647d345681ce')
name: "Ordinateur portable"
category: "électronique"
quantity: 40
inStock: true
```

```
_id: ObjectId('66f94bcd92b647d345681cf')
name: "Stylo"
category: "bureau"
quantity: 150
inStock: true
```

```
_id: ObjectId('66f94bcd92b647d345681d0')
name: "Clavier"
category: "électronique"
quantity: 20
inStock: true
```

- **DELETE : Suppression d'une fourniture**

Le **DAO** supprime le document correspondant dans MongoDB.

```
/**
 * Supprime une fourniture à partir de son nom.
 *
 * @param name Le nom de la fourniture à supprimer.
 */
public void deleteSupply(String name) { 1 usage
    collection.deleteOne(eq( fieldName: "name", name));
}
```

Le **service** appelle le DAO pour effectuer la suppression et affiche un message confirmant que la fourniture a bien été supprimée, assurant ainsi que l'utilisateur est informé du bon déroulement de l'opération.

```
/**
 * Supprime une fourniture par son nom.
 *
 * @param name Le nom de la fourniture à supprimer.
 */
public void deleteSupply(String name) { 1 usage
    supplyDao.deleteSupply(name);
    System.out.println("Fourniture supprimée !");
}
```

L'opération **Delete** est déclenchée dans **SupplyApp**, où le service est appelé pour supprimer une fourniture.

```
public static void main(String[] args) {
    SupplyService service = new SupplyService();

    // Suppression de la fourniture "Clavier"
    service.deleteSupply( name: "Clavier");

    // Vérification après suppression
    service.displaySupply( name: "Clavier"); // Ceci devrait afficher que la fourniture est introuvable
}
```

Au démarrage de l'application, les informations s'affichent correctement dans le terminal.

```
Fourniture supprimée !
Fourniture introuvable.
```

Je peux ensuite vérifier dans **MongoDB Compass** que la suppression a été effectuée en visualisant la collection **supplies**.

```
_id: ObjectId('66f94bdcd92b647d345681ce')
name: "Ordinateur portable"
category: "électronique"
quantity: 40
inStock: true
```

```
_id: ObjectId('66f94bdcd92b647d345681cf')
name: "Stylo"
category: "bureau"
quantity: 150
inStock: true
```

### Conclusion :

Ce projet m'a permis de mettre en pratique mes compétences dans le développement d'une application Java avec une base de données NoSQL, en utilisant MongoDB. Grâce à l'intégration de **MongoDB** et de son **driver Java**, j'ai pu concevoir un système de gestion des fournitures capable de manipuler efficacement les données à travers les opérations CRUD (Create, Read, Update, Delete).

L'utilisation de **Docker** a simplifié la gestion de l'environnement de base de données, tout en garantissant une isolation et une portabilité de l'application. Docker m'a permis de configurer rapidement MongoDB dans un environnement contrôlé, évitant les conflits de version ou de configuration.

Le projet a été structuré de manière modulaire, avec une séparation claire entre la couche d'accès aux données (**DAO**) et la couche métier (**Service**).

**MongoDB Compass** m'a offert une interface visuelle pratique pour vérifier et valider les opérations effectuées sur la base de données, ce qui m'a permis de confirmer l'exactitude de chaque étape du développement.

Ce projet m'a donc permis de développer des composants d'accès aux données NoSQL, tout en respectant les bonnes pratiques de développement logiciel. Il m'a également donné l'opportunité d'explorer les avantages d'une architecture modulaire et de renforcer mes compétences dans l'intégration d'outils modernes comme Docker et MongoDB dans un projet Java.

## 2. Précisez les moyens utilisés :

Dans ce projet, j'ai utilisé les outils et technologies suivants :

- **Java 21** : Langage principal pour le développement de l'application.
- **MongoDB 7.0.8** : Base de données NoSQL utilisée pour stocker les fournitures.
- **Maven 3.9.9** : Utilisé pour gérer les dépendances du projet, compiler le code, et exécuter l'application. Maven facilite également l'intégration du driver MongoDB.
- **MongoDB Java Driver 4.9.0** : Permet d'interagir avec MongoDB depuis l'application Java.
- **Docker**.
- **Git et GitHub** : Utilisés pour le suivi des versions du code source.
- **MongoDB Compass** : Utilisé pour visualiser et interagir avec la base de données MongoDB.
- **IntelliJ IDEA** : IDE utilisé pour écrire le code Java.

## 3. Avec qui avez-vous travaillé ?

Pour ce projet j'ai travaillé seul

## 4. Contexte

Nom de l'entreprise, organisme ou association ▶ Studi

Chantier, atelier, service ▶

Période d'exercice ▶ Du : 01/09/2024 au : 08/09/2024

## 5. Informations complémentaires (facultatif)

---

# DOSSIER PROFESSIONNEL <sup>(DP)</sup>

---

---

## Titres, diplômes, CQP, attestations de formation

*(facultatif)*

Intitulé	Autorité ou organisme	Date
Cliquez ici.	Cliquez ici pour taper du texte.	Cliquez ici pour sélectionner une date.

### Déclaration sur l'honneur

---

Je soussigné(e) [prénom et nom] **Berbigier Thomas** ,  
déclare sur l'honneur que les renseignements fournis dans ce dossier sont exacts et que je suis  
l'auteur(e) des réalisations jointes.

Fait à **Avignon** le **22/09/2024**  
pour faire valoir ce que de droit.

Signature :



## Documents illustrant la pratique professionnelle

*(facultatif)*

Intitulé
Cliquez ici pour taper du texte.



## ANNEXES

Figure 1

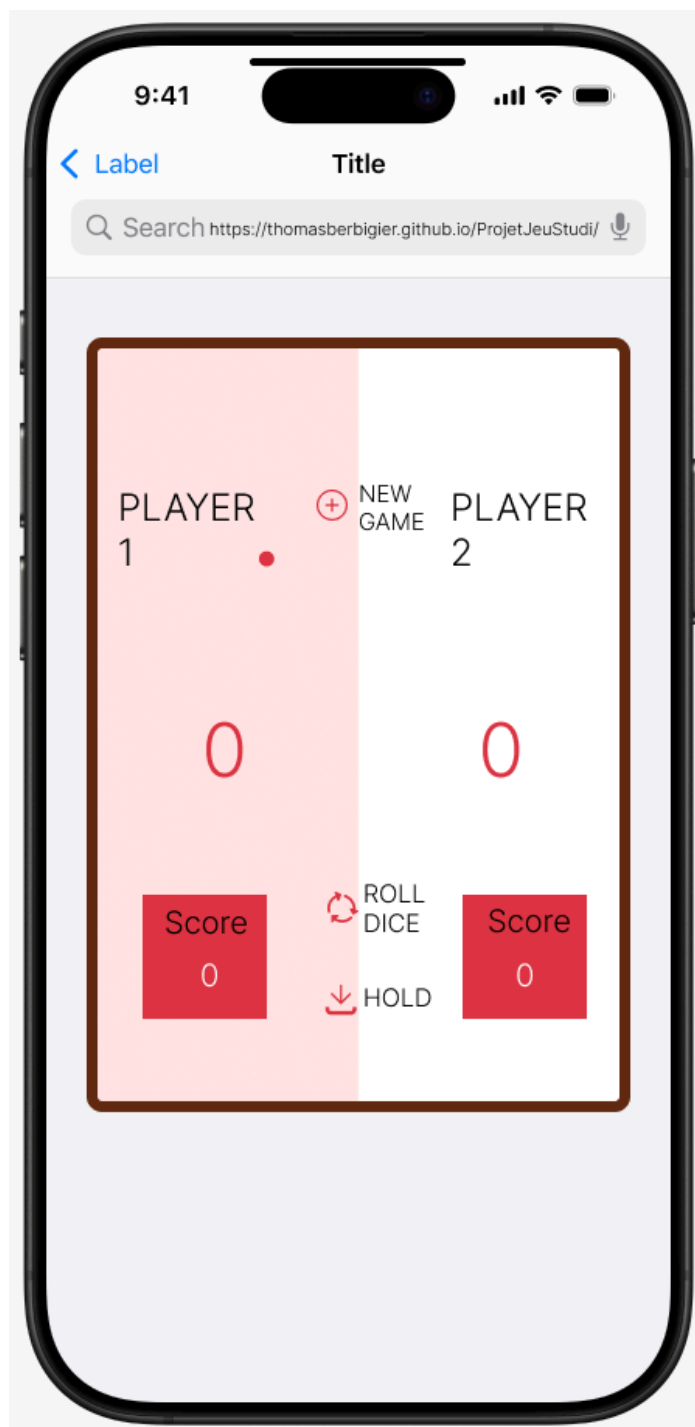


Figure 2

The image shows a login form titled "Se connecter" centered on a light blue background. The form has a white background and a thin grey border. It contains two input fields: "E-mail" and "Mot de passe". Below the "Mot de passe" field, there is a blue link "S'inscrire" and a blue button "Se connecter" with a white plus icon.

Se connecter	
<input type="text" value="E-mail"/>	
<input type="password" value="Mot de passe"/>	
<a href="#">S'inscrire</a>	<button>Se connecter</button>

**Figure 3**

The image shows a registration form titled "S'inscrire" (Register) centered on a light blue background. The form is a white rectangle with a grey header bar containing the title. Below the header, there are ten input fields stacked vertically: "Nom", "Code fiscal", "E-mail", "Adresse 1", "Adresse 2", "Ville", "Code Postal", "Pays", "Description", and "Numéro de téléphone". At the bottom left of the form is a blue button with a white plus icon and the text "S'inscrire". To the right of this button is a blue hyperlink labeled "Se connecter".


S'inscrire	
Nom	
Code fiscal	
E-mail	
Adresse 1	
Adresse 2	
Ville	
Code Postal	
Pays	
Description	
Numéro de téléphone	
 S'inscrire	<a href="#">Se connecter</a>

Figure 4

Gestion de stock

Tableau de bord

Articles

Clients

Fournisseurs

Parametrage

Recherche

Q

Bonjour Nom

Photo

Photo

Nom Prenom

Adresse, Ville, Code Postal,

Ancien mot de passe

Nouveau mot de passe

Confirmer mot de passe

Enregistrer

Annuler

© 2024 Gestion de stock - Tous droits réservés. Mentions légales | Politique de confidentialité | Conditions d'utilisation

Figure 5

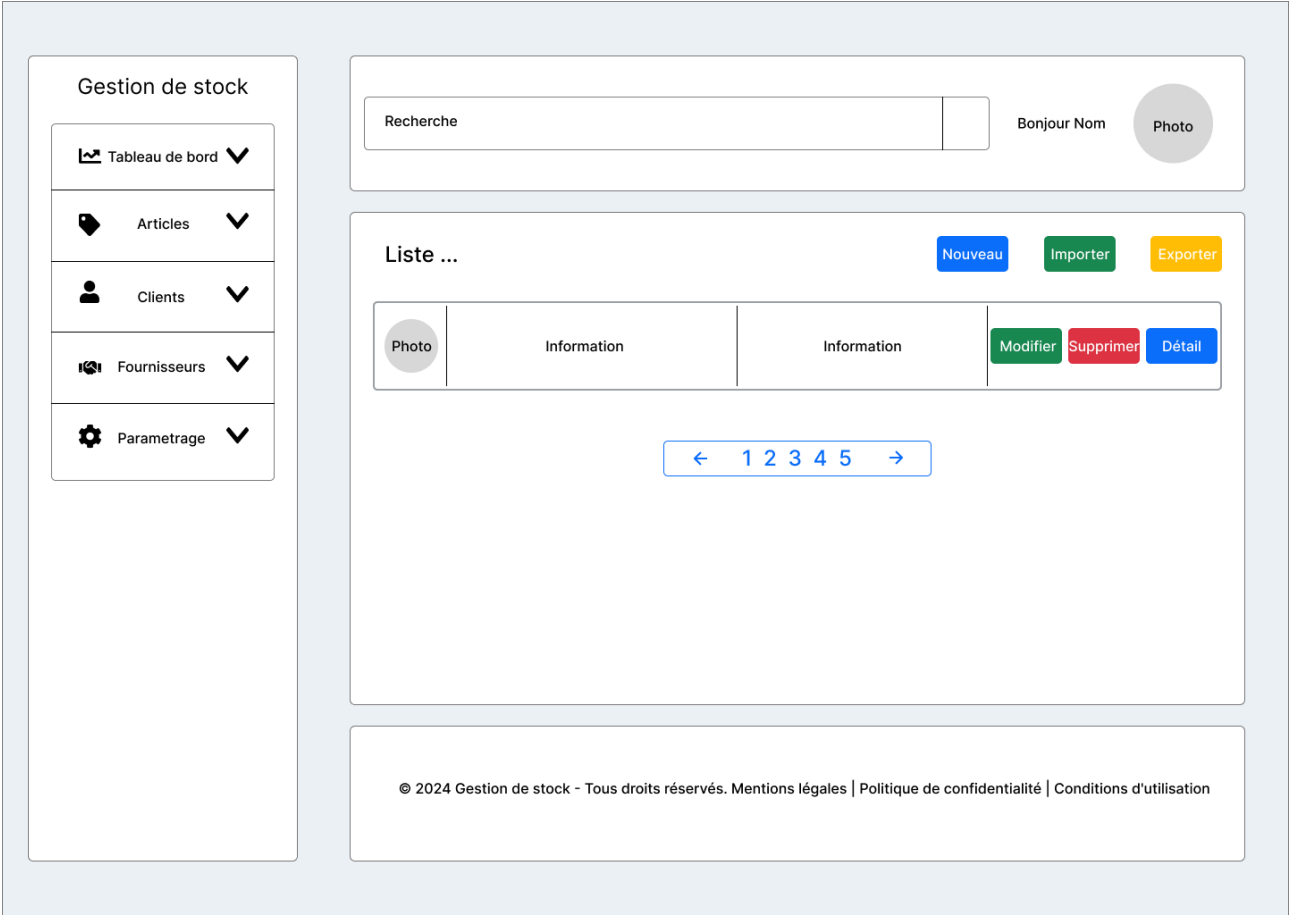


Figure 6

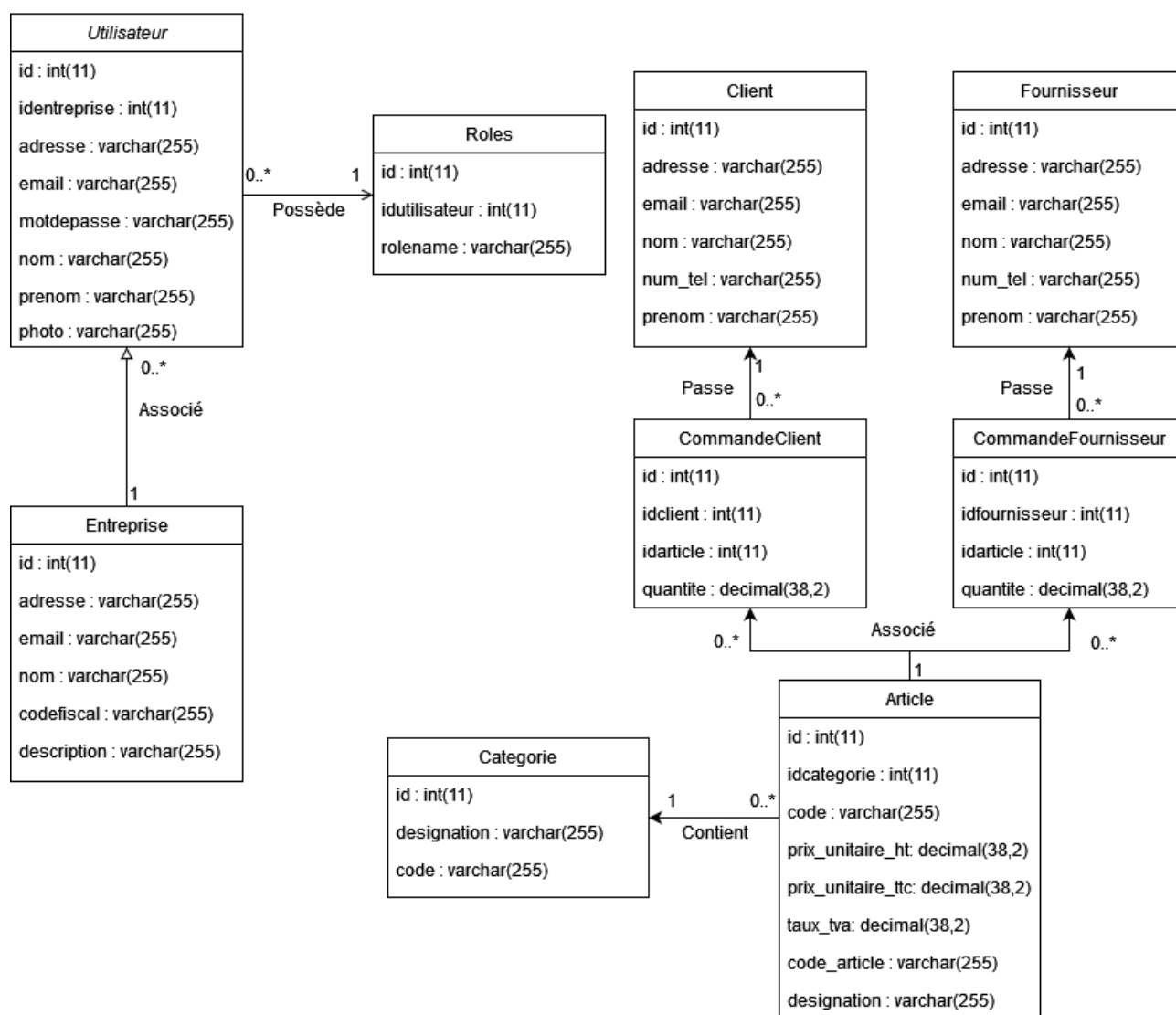


Figure 7

```
/**
 * Modèle de données représentant une fourniture.
 * Chaque instance de cette classe correspond à un document dans la collection "supplies" de MongoDB.
 */
public class Supply { 8 usages

    private String id; 2 usages
    private String name; 3 usages
    private String category; 3 usages
    private int quantity; 3 usages
    private boolean inStock; 3 usages

    // Constructeur avec les champs requis pour une nouvelle fourniture
    public Supply(String name, String category, int quantity, boolean inStock) { 2 usages
        this.name = name;
        this.category = category;
        this.quantity = quantity;
        this.inStock = inStock;
    }

    // Getters et Setters pour accéder et modifier les propriétés de la fourniture
    public String getId() { return id; } no usages
    public void setId(String id) { this.id = id; } no usages
    public String getName() { return name; }
    public void setName(String name) { this.name = name; } no usages
    public String getCategory() { return category; } 1 usage
    public void setCategory(String category) { this.category = category; } no usages
    public int getQuantity() { return quantity; } 2 usages
    public void setQuantity(int quantity) { this.quantity = quantity; } no usages
    public boolean isInStock() { return inStock; } 1 usage
    public void setInStock(boolean inStock) { this.inStock = inStock; } no usages
}
```