



# DOSSIER PROFESSIONNEL (DP)

*Nom de naissance*      - GUIOU  
*Nom d'usage*      - GUIOU  
*Prénom*      - FREDERIC  
*Adresse*      - 42 RUE GRANDE 13390 AURIOL

## Titre professionnel visé

Développeur Web et Web Mobile

### MODALITÉ D'ACCÈS :

- Parcours de formation
- Validation des Acquis de l'Expérience (VAE)

## 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.*

 <http://travail-emploi.gouv.fr/titres-professionnels>



MINISTÈRE CHARGÉ  
DE L'EMPLOI

# DOSSIER PROFESSIONNEL (DP)

## Sommaire

### Exemples de pratique professionnelle

<b>Activité-type 1 : Développer la partie front-end d'une application web ou web mobile en intégrant les recommandations de sécurité.</b>	<b>p.05</b>
- CP 1 Maquetter une application.	p.05
- CP 2 Réaliser une interface statique et adaptable.	p.09
- CP 3 Développer une interface utilisateur web dynamique.	p.15
- CP 4 Réaliser une interface utilisateur avec une solution de gestion de contenu ou e-commerce. <b>(Non concerné par cette compétence)</b>	<b>N/C</b>
<b>Intitulé de l'activité-type n° 2 : Développer la partie back-end d'une application web ou web mobile en intégrant les recommandations de sécurité.</b>	<b>p.23</b>
- CP 5 Créer une base de données.	p.23
- CP 6 Développer les composants d'accès aux données.	p.30
- CP 7 Développer la partie back-end d'une application web ou web mobile.	p.37
- CP 8 Élaborer et mettre en œuvre des composants dans une application de gestion de contenu ou e-commerce. <b>(Non concerné par cette compétence)</b>	<b>N/C</b>
<b>Titres, diplômes, CQP, attestations de formation (facultatif)</b>	<b>p.44</b>
<b>Déclaration sur l'honneur</b>	<b>p.45</b>
<b>Documents illustrant la pratique professionnelle (facultatif)</b>	<b>p.46</b>
<b>Annexes (Si le RC le prévoit)</b>	<b>p.47</b>

# **EXEMPLES DE PRATIQUE**

## **PROFESSIONNELLE**



# DOSSIER PROFESSIONNEL (DP)

## Activité-type 1

Développer la partie front-end d'une application web ou web mobile en intégrant les recommandations de sécurité.

CP 1 - Maquetter une application.

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

Dans le cadre de ma formation, j'ai pu en fin d'apprentissage du HTML et CSS, durant une journée de révision, commencer la réalisation d'une page portfolio/CV qui pourrait ainsi exposer l'ensemble de mes projets futurs. Il s'agirait d'un site exposant mes réalisations et permettant à l'utilisateur d'accéder à chaque application déployée. Le nom de ce projet est My-Digital-CV.

La première étape est de lister les différents scénarios d'interactions possibles entre le portfolio et l'utilisateur. En développement Web, on appelle ces scénarios des **user stories**.

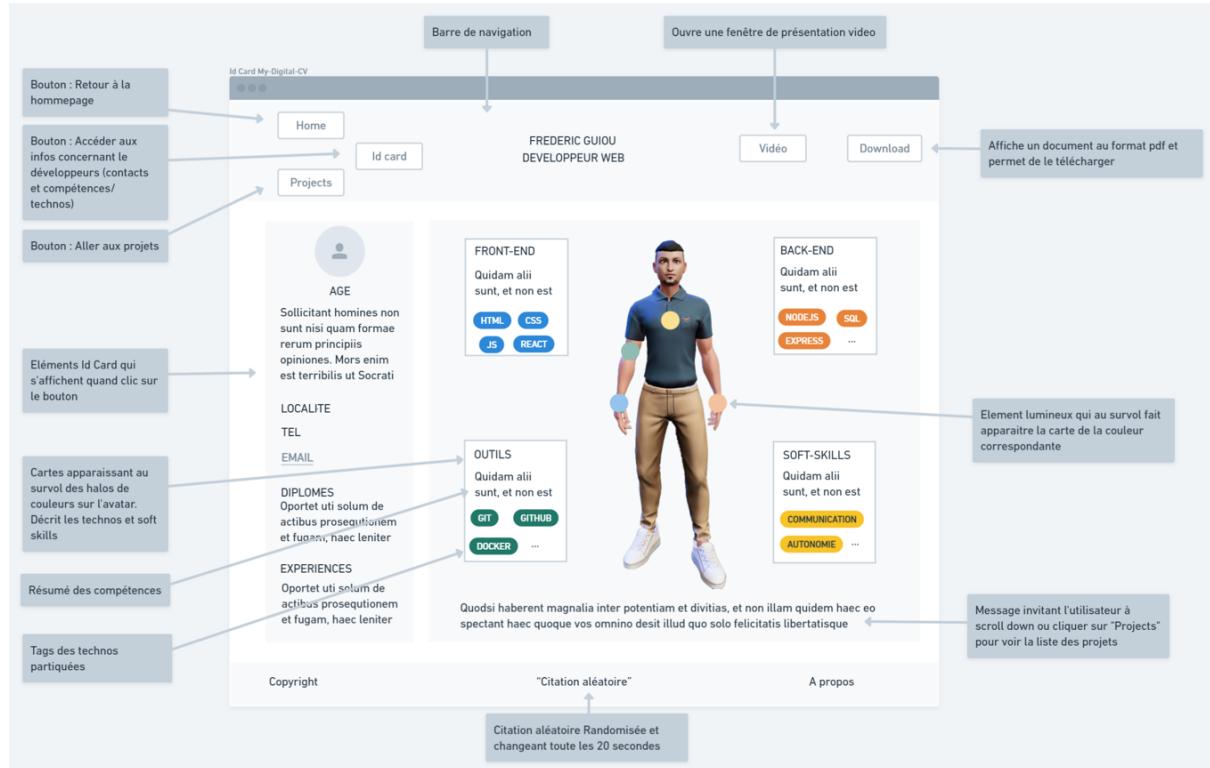
User Stories de My-Digital-CV :

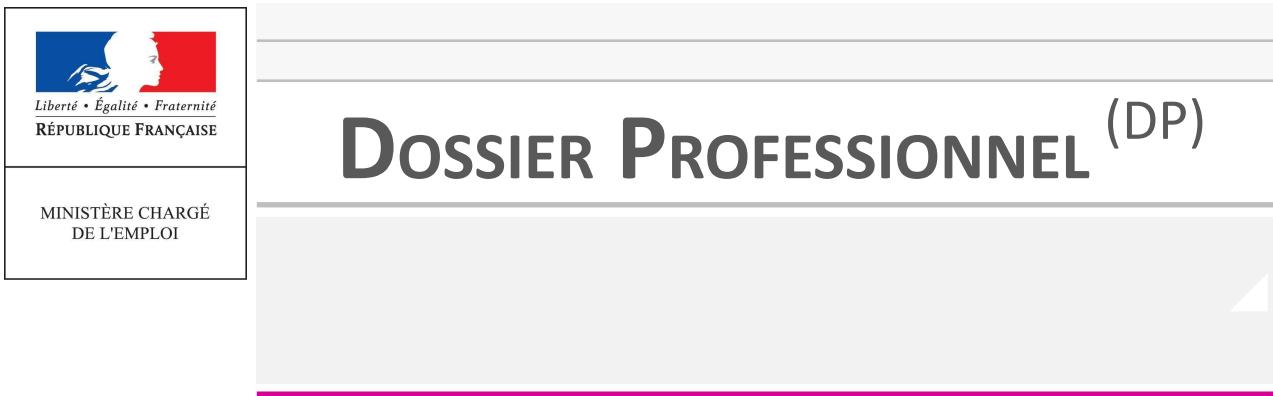
En tant que ...	Je veux ...	Afin de ...
Visiteur	accéder aux coordonnées/infos du prestataire développeur web	découvrir son profil et pouvoir le contacter de plusieurs façons.
Visiteur	retourner à la page d'accueil	démarrer à nouveau l'expérience utilisateur depuis le haut de page et à son état "zéro".
Visiteur	accéder à la liste des projets réalisés par le développeur	découvrir son savoir-faire et les projets qu'il a élaboré
Visiteur	Accéder/télécharger son CV au format pdf.	le conserver/l'imprimer.
Visiteur	consulter une vidéo de présentation du développeur (Lui, son activité, ses attentes pro...)	le découvrir sous un autre format de présentation.
Visiteur	consulter la liste des compétences du développeur	connaître l'ensemble des technos qu'il maîtrise
Visiteur	cliquer sur le lien ou consulter le code du repository.	tester les applications développées

J'ai par la suite élaboré, avec l'aide de l'outil Whimsical (<https://whimsical.com/>), des maquettes non fonctionnelles des principales pages de l'application. On appelle communément ces maquettes des **wireframes**. Elles ont pour but de présenter une ébauche de l'organisation de l'interface ainsi que les fonctionnalités prévues à chaque élément qui la compose. Dans le cas de My-Digital-CV, j'ai réalisé des wireframes au format desktop, tablette, et mobile afin de rendre l'application dès le départ responsive (adaptable en fonction du terminal utilisé pour accéder à l'application).

### Exemples de wireframes réalisés pour cette maquette d'application :

#### - Page Id\_Card (Desktop, Tablette, Mobile)

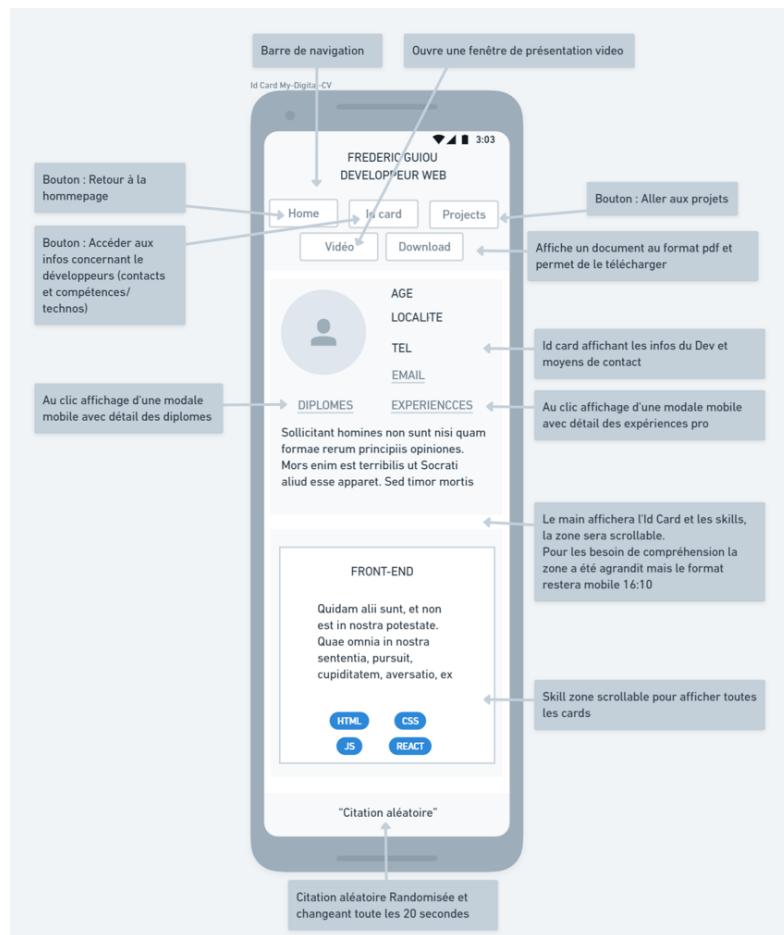




MINISTÈRE CHARGÉ  
DE L'EMPLOI

This diagram illustrates a digital professional dossier interface, likely a mobile application or web-based tool. It features a central profile card for a developer named Frédéric Guiou, showing details like age, location, diplomas, and experiences. The interface includes a navigation bar, video player controls, and various sections for skills and projects. A central figure (a man in a polo shirt) is surrounded by colored circles corresponding to skill categories. Callout boxes provide detailed descriptions of the interface elements.

- Bouton : Retour à la hommepage
- Bouton : Accéder aux infos concernant le développeurs (contacts et compétences/ technos)
- Bouton : Aller aux projets
- Eléments Id Card qui s'affichent quand clic sur le bouton
- Cartes apparaissant au survol des halos de couleurs sur l'avatar. Décrit les technos et soft skills
- Résumé des compétences
- Tags des technos partagées
- Barre de navigation
- Ouvre une fenêtre de présentation video
- Affiche un document au format pdf et permet de le télécharger
- Element lumineux qui au survol fait apparaître la carte de la couleur correspondante
- Message invitant l'utilisateur à scroll down ou cliquer sur "Projects" pour voir la liste des projets
- Citation aléatoire Randomisée et changeant toute les 20 secondes



## 2. Précisez les moyens utilisés :

Pour la réalisation des user stories, j'ai utilisé l'outil google docs après avoir répertorié manuscritement les différents scénarios possibles.

En ce qui concerne les wireframes, l'outil whimsical a été vraiment très agréable à utiliser. La palette de fonctionnalités est étendue. Le fait de disposer de frames aux différents formats (Mobile, Tablette, Desktop) est fort pratique).

## 3. Avec qui avez-vous travaillé ?

J'ai travaillé en totale autonomie sur ces documents durant une journée de révision prévue pendant la formation.



# DOSSIER PROFESSIONNEL (DP)

## 4. Contexte

Nom de l'entreprise, organisme ou association ➔ *O'clock*

Chantier, atelier, service ➔ *Travail personnel réalisé durant une journée de révision prévue dans le cadre de la formation.*

Période d'exercice ➔ Du : 22/02/2022 au : 09/08/2022

## 5. Informations complémentaires (facultatif)

Ce projet est toujours en cours d'élaboration depuis la fin de la formation et peut être sujet à modification.

### Activité-type 1

**Développer la partie front-end d'une application web ou web mobile en intégrant les recommandations de sécurité.**

**CP 2 ➔ Réaliser une interface web statique et adaptable.**

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

Une **application web** est par définition accessible depuis le World Wide Web. Aujourd'hui ce dernier peut être consulté depuis une **multitude de terminaux**, aux caractéristiques bien différentes. Il peut s'agir d'un **ordinateur** (Fixe ou laptop), mais également d'un **appareil mobile** (Smartphone ou tablette). Ces différents supports possèdent des résolutions d'affichage bien différentes de celles d'un écran d'ordinateur. Qui plus est, ils ont pour fonctionnalité la possibilité de passer d'un format paysage à portrait et inversement par simple rotation de l'appareil. Aussi il est important d'adapter l'affichage de l'application en fonction de la résolution et du format de l'écran utilisé. On appelle ces solutions web adaptables des **applications responsives**.

Le projet ci-dessous est une application de vente en ligne de Figurines. Il a été réalisé dans le cadre d'un exercice d'entraînement aux notions fondamentales en HTML et CSS. Nous avons eu un rendu attendu en image (format desktop) et une charte graphique nous a été communiquée.

Il était demandé de rendre cette page affichable sur la plupart des terminaux. L'intégralité du code HTML et CSS était à réaliser en autonomie.

S'agissant d'une application de vente en ligne, et celles-ci étant en général consultées depuis un ordinateur, j'ai d'abord construit la version desktop en me basant sur le final attendu. Puis j'ai rendu l'affichage responsive en utilisant les **medias queries** et définissant chacune d'elles en fonction de ces points d'arrêt (breakpoints).

L'autre option aurait été de penser l'application en **mobile first** puis de l'adapter au format desktop toujours à l'aide des média queries. (Tendance la plus souvent pratiquée).

### Voici tout d'abord la version Desktop de l'application :

The screenshot shows a desktop browser displaying the oFig website. At the top is a red header bar with the logo 'oFig Figurines et statuettes'. Below it is a grey navigation bar with the text 'Figurines sur le thème *Final Fantasy*'. The main content area displays three product cards for Final Fantasy VII figurines:

- Cloud Strife Remake**: Figurine de 8 pouces - livrée avec son packaging. Price: 74,90 € 59 €. Buttons: 'Ajouter au panier' and '74,90 € 59 €'.
- Tifa Lockhart**: Figurine de 7 pouces - livrée avec son packaging. Price: 69 €. Buttons: 'Ajouter au panier' and '69 €'.
- Vincent Valentine**: Figurine de 7 pouces - livrée avec son packaging. Price: 74,90 €. Buttons: 'Ajouter au panier' and '74,90 €'.

At the bottom of the page is a dark footer bar with the text 'oFig - toutes les images sont sous copyright SquareEnix'.

On retrouve une architecture HTML standard avec un header, un main, un footer.

Ci-dessous un extrait du code HTML de la page (détail du code pour une balise article):

```
<main class="productlist">
    <article class="product">
        
        <h3 class="producttheme">
            Final Fantasy VII
        </h3>
        <h2 class="productname">
            Cloud Strife Remake
        </h2>
        <p class="producttext">
            Figurine de 8 pouces - livrée avec son packaging
        </p>
        <a href="#" class="button">
            <span class="basket">Ajouter au panier</span>
            <span class="price">
                <span class="barredprice">74,90 €</span>
                <span class="promoprice">59 €</span>
            </span>
        </a>
    </article>
```



# DOSSIER PROFESSIONNEL (DP)

MINISTÈRE CHARGÉ  
DE L'EMPLOI

La balise main, à laquelle on attribue la **classe productlist** contient les éléments de la liste des articles organisés pour être présentés de façon lisible pour l'utilisateur. On applique du CSS à ce conteneur pour créer cette présentation à l'écran.

**Voici une partie du code CSS :**

```
...  
.productlist{  
    display: flex;  
    justify-content: space-evenly;  
    flex-wrap: wrap;  
    align-content: center;  
    padding: 2% 6% 2% 6%;  
  
    position: absolute;  
    top: 10%;  
    left: 0;  
    right: 0;  
    bottom: 6%;  
  
    overflow-y: auto;  
}
```

## Détails des principales propriété CSS utilisées :

**display : flex** permet à un élément flexible de redéfinir ses dimensions pour lui permettre de remplir l'espace disponible de son conteneur. Ainsi le fait que productlist soit affectée par cette propriété va impacter automatiquement ses éléments enfants (ici les articles).

**Justify-content : space-evenly** s'occupe de répartir l'espace autours des éléments en fonction de l'axe principal d'un conteneur flexible.

**Flex-wrap : wrap** a pour objectif de disposer les éléments flexibles sur une seule ligne ou de leur permettre d'occuper plusieurs lignes dans leur élément parent grâce à un retour à la ligne si l'espace maximum de la première ligne est atteint.

**Align-content : center** assigne l'espace disponible du conteneur autours des éléments enfants de ce dernier suivant l'axe horizontal. Chaque article est alors déposé au milieu de la productlist et l'espace est réparti de façon égale au-dessus et en dessous de ceux-ci.

Le **padding** permet d'ajouter des écarts de remplissage sur les côtés d'un élément. Ici afin de rendre les cartes articles moins imposantes dans la productlist, j'ai ajouté du padding pour réduire leur taille.

La **position : absolute** quant à elle définit la façon dont l'élément sera positionné dans la document.

Enfin le **overflow-y** permet de définir la possibilité de faire défiler le contenu de l'élément suivant l'axe vertical.

Ici les versions mobile et tablette de l'application :

Mobile

**oFig Figurines et statuettes**

Figurines sur le thème *Final Fantasy*



*Final Fantasy VII*

**Cloud Strife Remake**

Figurine de 8 pouces - livrée avec son packaging

Ajouter au panier      74,90€ 59 €



*Final Fantasy Advent Children*

oFig - toutes les images sont sous copyright SquareEnix

Tablette

**oFig Figurines et statuettes**

Figurines sur le thème *Final Fantasy*



*Final Fantasy VII*

**Cloud Strife Remake**

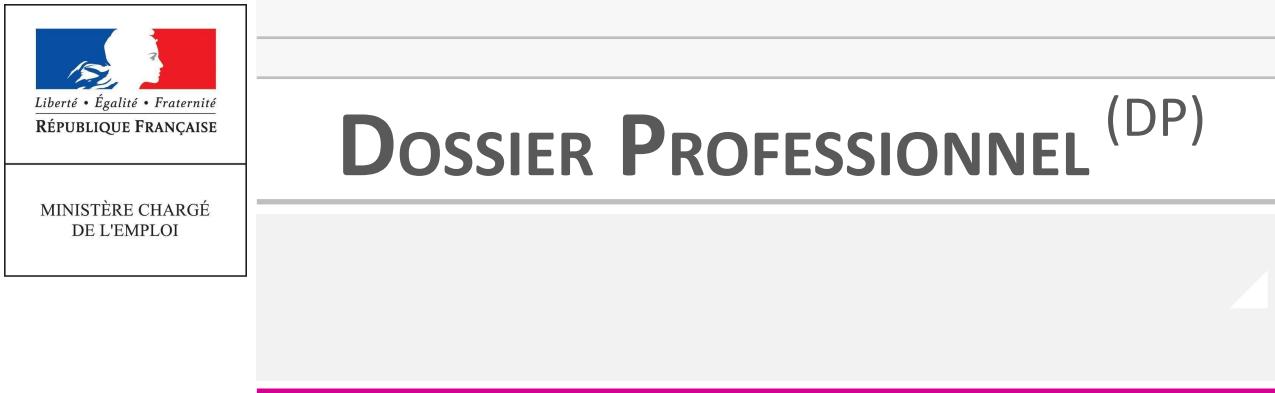
Figurine de 8 pouces - livrée avec son packaging

Ajouter au panier      74,90€ 59 €



oFig - toutes les images sont sous copyright SquareEnix

Pour rendre la disposition des éléments de la page cohérente par rapport à la définition et l'orientation de l'écran utilisé j'ai défini des **media queries** qui permettent de reconfigurer certaines propriété CSS.



-Voici un extrait du code CSS des media queries :

```
/*SMARTPHONE*/
/*PORTRAIT*/

@media screen and (min-device-width : 320px) and
(max-device-width : 767px) and (orientation :
portrait) {
    .productlist{
        flex-direction: column;
        flex-wrap: nowrap;

        position: inherit;

        margin-top: 20%;
        margin-bottom: 8%;

    }

    .headerup{
        justify-content: center;
        padding: 3.40%;

    }

    .headerdown{
        padding: 1.20%;

    }

    .product{
        width: auto;

    }

    .header{
        width: 100%;

    }

    .footer{
        align-content: center;
        font-size: small;
        top: 96%;

    }
}
```

Cette media query s'applique aux smartphones dont la **largeur d'affichage (width)** se situe entre 320 et 767 pixels en **orientation portrait**.

Le **display flex** étant déjà paramétré sur le code CSS de la version Desktop de la **classe productlist**, inutile ici de répéter cette propriété. Je ne modifie que celles permettant d'adapter l'affichage à l'appareil.

J'ajoute la propriété **flex-direction** en colonne afin de disposer les éléments enfants de la productlist à la verticale.

Je neutralise la propriété **flex-wrap** avec la valeur nowrap car dans le cas présent tous les articles seront affichés à la suite en colonne. Un retour à la ligne n'aurait aucune utilité si ce n'est de créer une colonne supplémentaire qui augmenterait le nombre d'éléments affichés à l'écran et ferait perdre en lisibilité.

J'ajuste plusieurs valeurs de **margin**, **padding**, **font-size** (**taille de police**) et **d'alignement d'éléments** sur les **classes header et footer** pour rendre le visuel plus homogène.

**2. Précisez les moyens utilisés :**

Tout comme expliqué de façon plus détaillée dans la section précédente, j'ai utilisé des media queries pour appliquer des propriétés CSS aux éléments d'une page web suivant des conditions de taille d'affichage et d'orientation de l'appareil.

**3. Avec qui avez-vous travaillé ?**

J'ai travaillé seul sur ce projet, il s'agissait de l'un de mes tous premiers challenges quotidiens de mise en application des notions du jour .

**4. Contexte**

Nom de l'entreprise, organisme ou association ➤ *O'clock*

Chantier, atelier, service ➤ *Exercice de formation (challenge quotidien)*

Période d'exercice ➤ Du : 22/02/2022 au : 09/08/2022

**5. Informations complémentaires (facultatif)**

(Aucune)



# DOSSIER PROFESSIONNEL (DP)

MINISTÈRE CHARGÉ  
DE L'EMPLOI

## Activité-type 1

Développer la partie front-end d'une application web ou web mobile en intégrant les recommandations de sécurité.

**CP 3** - Développer une interface utilisateur web dynamique.

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

Après avoir appris comment développer une application dans un langage statique, il nous a été expliqué comment rendre ses interfaces dynamiques et donner à l'utilisateur la possibilité d'interagir avec elle.

Comment ? Avec l'utilisation du langage **Javascript** qui le permet via l'exécution de **scripts**. Ces derniers définissent les interactions que l'utilisateur est susceptible d'effectuer et les réponses fournies par l'interface lors de ces événements.

Aussi ce qui est affiché à l'utilisateur n'est plus l'interface statique créé en HTML et CSS mais des **objets Javascript générés** par ce nouveau langage dans le **Document Object Model**.

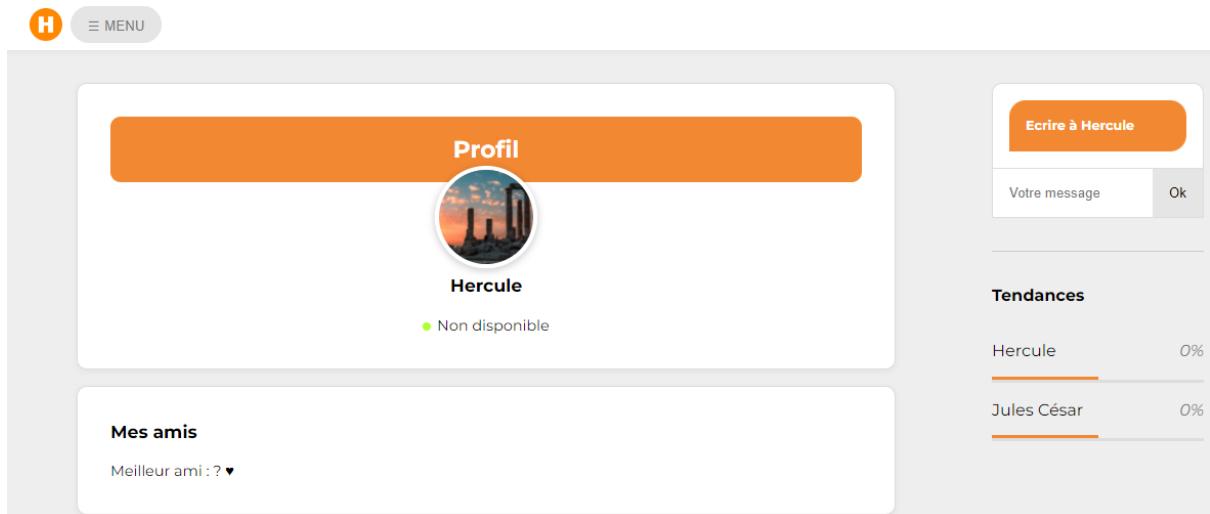
Le **DOM** est une représentation structurée des différents objets sous la forme d'un arbre manipulable en fonction des **événements** qui se produisent. Les objets du DOM possèdent des propriétés et des méthodes. Une **méthode** est une fonction qui a pour utilité de manipuler l'objet auquel elle est associée.

A la fin de chaque chapitre de la formation, j'ai eu une évaluation d'une demi-journée permettant de situer ma compréhension des concepts vus pendant le cours. Le parcours "Hercule - Le retour" m'a été proposé après m'avoir exposé les notions fondamentales de Javascript et de la dynamisation.

L'intégration HTML et CSS était fournie ainsi qu'un script base.js compte tenu de la durée de l'exercice limitée en temps de réalisation. L'ensemble du code à réaliser devait être inscrit dans un fichier exo.js vierge. Le fichier base contenait 4 méthodes qui gèrent l'affichage des objets du DOM. (setBestFriends, displayWork, printFriends, et fillProfil)

Au chargement, on découvrait la page de profil d'Hercule avec tout d'abord les informations le concernant puis la liste de ses 12 travaux. Malheureusement l'interface ne disposait pas des données de ce dernier.

**Voici l'état zéro de la page avant création du script Js :**



Aussi la première étape consistait à créer un objet dans le fichier .js nommé Hercule avec ses propriétés et valeurs, tout ceci dans une constante app.

```
● ● ●

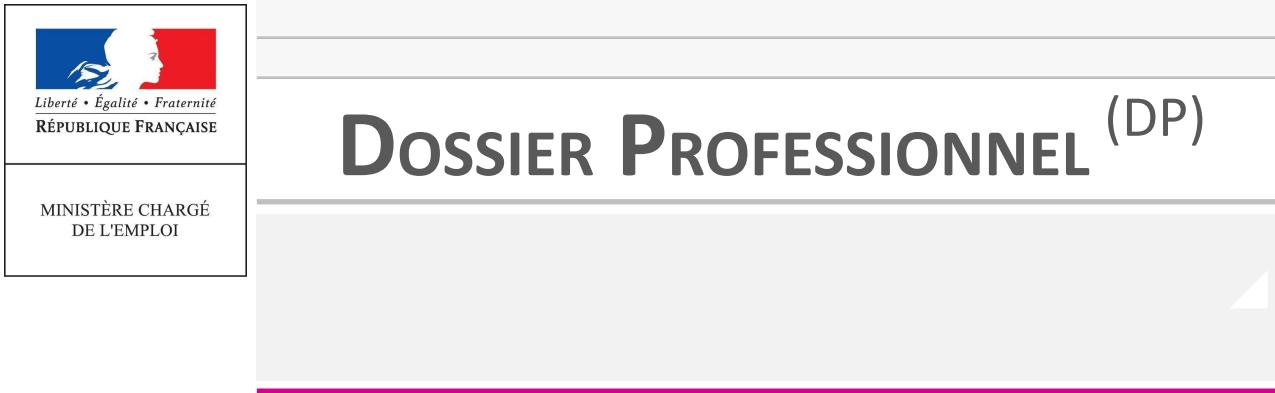
Hercule : {
    name: "Hercule",
    job: "Demi-dieu",
    age: 35,
    department: "75",
    arm: 60.5,
    inRelationship: true,
},
```

Cet objet afin d'être exploité devait être inséré en argument dans la méthode fillProfil de base.js.

```
● ● ●

function init () {
    base.fillProfil(app.Hercule);
};

init();
```



Une autre étape était d'**afficher les amis d'Hercule** et afficher qui est **son meilleur ami** (le premier nom du tableau).

Aussi j'ai créé un tableau dans une variable friends contenant les 4 amis demandés.

```
etape3 () {
    let friends = ["Jupiter", "Junon", "Alcmène", "Déjanire"];
    base.printFriends(friends);
    base.setBestFriend(friends[0]);
},
```

J'ai inséré ce tableau en argument de la méthode printFriends de la constante base du fichier base.js. Mais également dans la méthode setBestFriend. (Le meilleur ami étant le premier du tableau)

Le code HTML ne contenait pas balise <h1>, j'ai manipulé le DOM afin d'y créer un objet:

```
etape4 () {
    let title = document.createElement("h1");
    title.classList.add("banner_title");
    title.textContent = "Vous consultez le profil de Hercule";
    let header = document.querySelector("#header-banner");
    header.appendChild(title);
},
```

On crée un élément dans une variable (ici title) avec la méthode Javascript native **document.createElement**.

On ajoute la classe CSS prévue dans l'intégration fournie à l'objet créé avec **classList.add**.

La méthode **textContent** ajoute à title le contenu texte que l'on souhaite afficher avec cet objet.

Une fois cet élément créé, on le positionne dans l'interface. Pour cela on sélectionne l'élément parent par son ID #header-banner (celui qui va contenir l'objet dans le DOM) avec la méthode **document.querySelector** et on lui fait adopter notre objet title en lui appliquant **appendChild** avec pour argument title.

La grande force de Javascript est de modifier son DOM en fonction des actions de l'utilisateur sur l'interface. Pour cela il dispose d'une méthode native qui lui permet **d'écouter les événements** susceptibles d'intervenir.

Dans le challenge Hercule, il était demandé de programmer deux fonctionnalités à base d'écouteur d'événement : la première pour faire apparaître le menu déroulant de l'application, et la seconde pour envoyer un message au héros légendaire.

**Voici le code JS pour activer/désactiver le menu déroulant :**

```
etape8 () {
    let menuToggler = document.querySelector("#menu-toggler");
    let headerBanner = document.querySelector("#header-banner");
    menuToggler.addEventListener("click", function () {
        headerBanner.classList.toggle("banner--open")
    });
},
```

Dans un premier temps, je sélectionne l'Id pointant vers l'élément menu dans le HTML (#menu-toggler), ainsi que l'élément #header-banner car c'est par le biais de cet élément que le CSS affiche le menu. Ces Sélecteurs sont insérés dans des variables distinctes.

L'étape suivante consiste à placer un écouteur d'événement avec **addEventListener** sur l'action "click" qui déclenche l'exécution de la fonction qui lui est associée, ici l'événement lance l'activation via l'interrupteur **toggle** de la classe .banner- -open sur l'objet headerBanner.

**Voici le code JS pour envoyer un message :**

```
etape9 () {
    let contact = document.querySelector("#contact");
    contact.addEventListener('submit', function (event) {
        event.preventDefault();
        alert("Hercule ne souhaite pas être dérangé !");
    });
},
```

Dans une variable contact on sélectionne l'élément du DOM permettant de taper un message et l'envoyer (l'ID #contact).



# DOSSIER PROFESSIONNEL (DP)

MINISTÈRE CHARGÉ  
DE L'EMPLOI

On ajoute un écouteur d'événement avec **addEventListener** sur l'objet “contact”. Au moment de la soumission du formulaire ('submit') on empêche le rechargement de la page avec la méthode native Javascript **preventDefault** ce qui permet aux données entrées d'être envoyées.

Par souci de simplicité dans l'exercice, l'envoi du formulaire affichait une alerte à l'écran informant que le destinataire ne souhaitait pas être dérangé.

Chaque fonctionnalité est enfin appelée dans une méthode d'initialisation “init”, elle-même lancée à la fin du script.

```
function init () {
    base.fillProfil(app.Hercule);
    app.etape3();
    app.etape4();
    app.etape5();
    app.etape6();
    app.etape7();
    app.etape8();
    app.etape9();
    app.etape10();
    app.etape12();
};

init();
```

Au chargement de la page on obtient donc l'affichage final ci-dessous. (Le profil et les travaux en déroulant l'affichage)

The screenshot shows a user profile page for a character named 'Hercule-du-75'. The top navigation bar includes a logo with a large orange letter 'H', a 'MENU' button, and a status message 'Vous consultez le profil de Hercule'. The main profile area features a large orange header 'Profil' and a circular profile picture showing a sunset over industrial structures. Below the picture is the username 'Hercule-du-75'. A row of buttons displays basic information: 'Hercule' (highlighted in orange), 'Demi-dieu', '35 ans', 'Vit actuellement dans le 75', and '60.5cm de tour de bras'. A status indicator 'En couple' is shown above a green dot labeled 'Disponible'. To the right, there's a sidebar with an 'Ecrire à Hercule' button and a text input field 'Votre message' with an 'Ok' button. Another sidebar titled 'Tendances' shows two entries: 'Hercule' at 69% and 'Jules César' at 31%, each with a progress bar. The 'Activités' section lists three achievements: 'Le lion de Némée', 'Le sanglier d'Erymanthe', and 'L'hydre de Lerne'. The 'Mes amis' section lists four friends: 'Jupiter', 'Junon', 'Alcmène', and 'Déjanire', with 'Jupiter' as the 'Meilleur ami'. The overall layout is clean with a light gray background and rounded corners.



MINISTÈRE CHARGÉ  
DE L'EMPLOI

# DOSSIER PROFESSIONNEL (DP)

Le lion de Némée



Lorem ipsum dolor sit, amet consectetur adipisicing elit. Voluptatibus, maxime cumque quaerat eaque cupiditate consectetur eveniet dignissimos corporis laboriosam. Excepturi magnam itaque dolorem error laboriosam placeat similique amet culpa consequatur.

L'hydre de Lerne



## 2. Précisez les moyens utilisés :

Pour ce parcours de synthèse des notions, J'ai utilisé Javascript et ses méthodes natives **document.createElement**, **classList.add**, **textContent**, **document.querySelector**, **appendChild**, **addEventListener**, **toggle** (Liste non exhaustive). L'HTML et CSS étaient fournis pour les besoins de l'épreuve réalisée en 4h.

### 3. Avec qui avez-vous travaillé ?

J'ai travaillé seul.

### 4. Contexte

Nom de l'entreprise, organisme ou association ▶ *O'clock*

Chantier, atelier, service ▶ *Parcours synthétique de chapitre (4h)*

Période d'exercice ▶ Du : 22/02/2022 au : 09/08/2022

### 5. Informations complémentaires (facultatif)

L'exercice présenté détenait de la donnée par l'intermédiaire de tableaux mis à disposition dans base.js ou leur création lors de l'épreuve.

Cependant, lorsque la **donnée est détachée** du Frontend, **sur un serveur distant** il existe des méthodes **d'appels** Frontend permettant la dynamisation de l'interface.

Les plus employées sont **FETCH** et **AXIOS**.

La première est une API prise en charge nativement dans les navigateurs les plus courants par rétrocompatibilité, la seconde est une bibliothèque Javascript nécessitant une installation facile de package tier autonome.

Les deux méthodes ont cependant des modes de fonctionnements différents (type de propriété utilisée, méthode d'appel globale ou propre à l'objet JS) mais permettent à Javascript de rendre l'interface dynamique toujours en fonction des actions réalisées par l'utilisateur. L'intérêt de l'utilisation de ce type de méthode est un gain de performance dans le traitement Frontend de l'application mais aussi une diminution des ressources nécessaires à l'utilisation de l'interface car Javascript ne charge que la partie dynamique appelée par l'utilisateur et pas la totalité de la page. On parle ici de traitement asynchrone.

Ci-dessous un exemple d'appel via la méthode **FETCH** lors d'un parcours suivant :

```
...  
  
async fetchAndInsertTasksFromApi (event) {  
    // Récupère la liste des tâches à l'aide de la fonction fetch()  
    const urlToCall = taskManager.apiEndpoint + "/tasks";  
    const response = await fetch(urlToCall);  
    if (response.ok) {  
        const tasks = await response.json();  
        console.log(tasks);  
        for (const task of tasks) { taskManager.insertTaskInHtml(task);  
        }  
    }  
},
```



# DOSSIER PROFESSIONNEL (DP)

MINISTÈRE CHARGÉ  
DE L'EMPLOI

## Activité-type 2

Développer la partie back-end d'une application web ou web mobile en intégrant les recommandations de sécurité.

*CP 5 - Créer une base de données.*

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

Lors de la spécialisation data, j'ai eu à travailler sur un projet d'application de **générateur de phrase aléatoire**. Le nom de ce projet est **Cadex**. Abréviation de Cadavre exquis, il a été inventé par les auteurs du mouvement surréaliste en 1925. Les premiers joueurs furent entre autres Jacques Prévert et Frida Kahlo. Il consistait à écrire les 4 composants d'une phrase choisie par le joueur sur des papiers différents : nom, adjetif, verbe et complément. Puis on tirait un composant de chaque type au hasard et cela créait une phrase plus ou moins humoristique et compréhensible.

Pour développer toute la **gestion** de la donnée de façon **cohérente**, il est nécessaire dans ce type d'exercice de **respecter des étapes** dans la création de celle-ci.

On pense à l'organisation des entités qui vont composer notre base de données en commençant par chacun des attributs de l'une d'entre elles, et les relations entre chaque entité.

Pour ce faire, on va commencer par la **première étape** :

**La création d'un Modèle Conceptuel de Données (MCD).**

Il s'agit d'une **représentation graphique** des entités (**tables**) de la base de données (avec liste de leurs **propriétés**) permettant de comprendre leur lien grâce à l'utilisation de **relations** et de **cardinalités**.

Une **phrase** générée contient les éléments suivants : **nom, adjetif, verbe et complément**. Nous avons donc **cinq entités** en relation dans ce MCD.

Les cardinalités sont les quantités minimum et maximum d'occurrences concernées par ces relations pour chaque table. Il existe plusieurs types de combinaisons de cardinalités déterminées par les **maximums** de chaque combinaison : Le *One To Many*, le *One to One*, et le *Many to Many*.

Dans notre Cadex les cardinalités sont les suivantes :

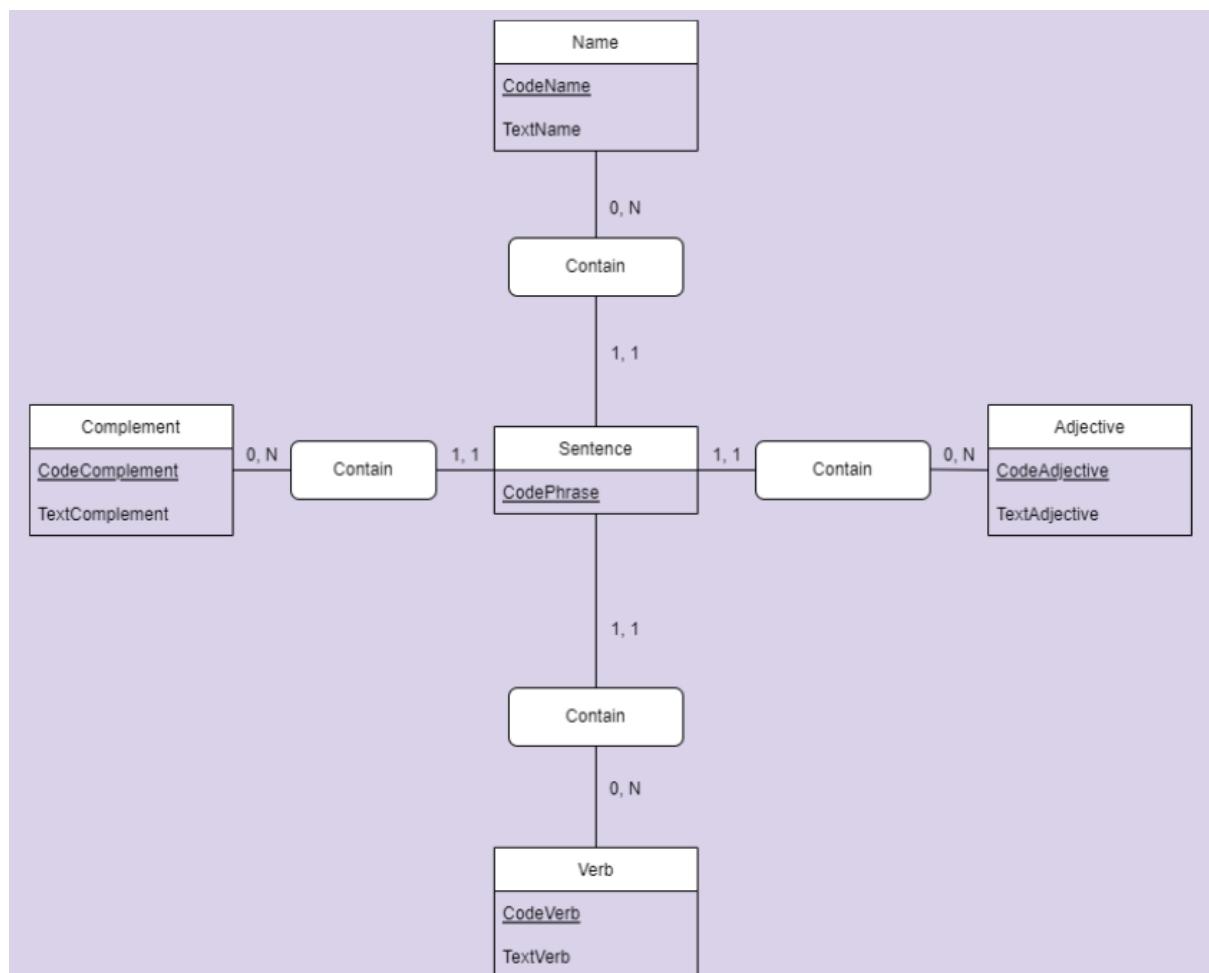
- Une phrase ne peut contenir qu'un et qu'un seul nom, adjectif, verbe ou complément. (1, 1)
- Un nom, un adjectif, un verbe ou un complément peuvent être contenus plusieurs phrases ou aucune. (0, N)

A noter que cette dernière cardinalité peut être en (1, N) si on décide qu'un nom, adjectif, verbe ou complément, lorsqu'il est inséré dans la base de données, doit obligatoirement servir à la création d'une phrase. On ajoute alors une contrainte.

On se retrouve donc avec une configuration *One to Many*.

NB : Une relation *Many to Many* nécessite de créer une table de liaison (en plus dans la base de données) , cette entité relationnelle est alors composée des clés primaires des deux tables auxquelles elle fait référence. Ici notre Projet n'est pas concerné par le cas Many to Many mais il est important d'avoir connaissance de cette particularité.

Voici le MCD une fois réalisé :





# DOSSIER PROFESSIONNEL (DP)

MINISTÈRE CHARGÉ  
DE L'EMPLOI

La seconde étape est d'établir un Modèle Logique des Données (MLD) à partir du MCD.

Les identifiants des entités du MCD deviennent des **clés primaires** représentées par un id unique à chaque entrée dans la table. La relation globale étant de type 1:N (**One to Many**) la table Sentence détient sa clé primaire et des **clés étrangères** qui sont les clés primaires de chaque élément qui compose une phrase (Name, Adjective, Verb, Complement).

```
# Modèle Logique de données ::

Name(id, TextName)
Adjective(id, TextAdjective)
Verb(id, TextVerb)
Complement(id, TextComplement)
Sentence(id, #Name(id), #Adjective(id), #Verb(id), #Complement(id))
```

La troisième étape est la création du **dictionnaire des données**. Il s'agit d'un recueil reprenant l'ensemble des tables, les champs qui les composent, leurs types et spécificités.

Voici le dictionnaire des données du projet Cadex :

**Table Name :**

Champ	Type	Spécificités	Description
id	INT	NOT NULL GENERATED ALWAYS AS IDENTITY	clé primaire d'un nom
TextName	TEXT	NOT NULL	nom

**Table Adjective :**

Champ	Type	Spécificités	Description
id	INT	NOT NULL GENERATED ALWAYS AS IDENTITY	clé primaire d'un adjetif
TextAdjective	TEXT	NOT NULL	adjectif

**Table Verb :**

Champ	Type	Spécificités	Description
id	INT	NOT NULL GENERATED ALWAYS AS IDENTITY	clé primaire d'un verbe
TextVerb	TEXT	NOT NULL	verbe

**Table Complement :**

Champ	Type	Spécificités	Description
id	INT	NOT NULL GENERATED ALWAYS AS IDENTITY	clé primaire d'un complément
TextComplement	TEXT	NOT NULL	complément

**Table Sentence :**

Champ	Type	Spécificités	Description
id	INT	NOT NULL GENERATED ALWAYS AS IDENTITY	clé primaire d'une phrase
Name_id	INT	NOT NULL	clé étrangère ref. id du nom
Adjective_id	INT	NOT NULL	clé étrangère ref. id de l'adjectif
Verb_id	INT	NOT NULL	clé étrangère ref. id du verbe
Complement_id	INT	NOT NULL	clé étrangère ref. id du complément



# DOSSIER PROFESSIONNEL (DP)

MINISTÈRE CHARGÉ  
DE L'EMPLOI

Enfin, Le **Modèle Physique de Données** (MPD) peut être comparé à la réunion du MLD et du dictionnaire des données. Il s'agit du fichier SQL que l'on importe dans la base de données pour créer les tables et les différentes propriétés qui les composent.

Dans le projet Cadex, j'ai ainsi créé un fichier `create_tables.sql` avec le script ci-dessous :

Le fichier de script commence par la commande **BEGIN** pour signifier l'initialisation de ce dernier.

```
•••••  
BEGIN;  
  
DROP TABLE IF EXISTS "name" CASCADE;  
DROP TABLE IF EXISTS "adjective" CASCADE;  
DROP TABLE IF EXISTS "verb" CASCADE;  
DROP TABLE IF EXISTS "complement" CASCADE;  
DROP TABLE IF EXISTS public.sentence CASCADE;  
  
CREATE TABLE IF NOT EXISTS "name"  
(  
    id integer NOT NULL GENERATED ALWAYS AS IDENTITY,  
    TextName text NOT NULL,  
    CONSTRAINT name_pk PRIMARY KEY (id)  
)  
  
CREATE TABLE IF NOT EXISTS "adjective"  
(  
    id integer NOT NULL GENERATED ALWAYS AS IDENTITY,  
    TextAdjective text NOT NULL,  
    CONSTRAINT adjective_pk PRIMARY KEY (id)  
)  
CREATE TABLE IF NOT EXISTS "verb"  
(  
    id integer NOT NULL GENERATED ALWAYS AS IDENTITY,  
    TextVerb text NOT NULL,  
    CONSTRAINT verb_pk PRIMARY KEY (id)  
)  
  
CREATE TABLE IF NOT EXISTS "complement"  
(  
    id integer NOT NULL GENERATED ALWAYS AS IDENTITY,  
    TextComplement text NOT NULL,  
    CONSTRAINT complement_pk PRIMARY KEY (id)  
)
```

Dans le cas d'une hypothétique réinitialisation de la base de données sans avoir la paramétriser de nouveau dans le sgbd, on prévoit une suppression des tables existantes (**DROP TABLE IF EXISTS + "nom\_de\_la\_table"**).

L'option **CASCADE** à la fin de la commande permet de supprimer toute référence à la table dans une autre (ex: *la clé étrangère name\_id sera supprimée de la table sentence si on supprime la table name*)

Je crée ensuite la structure de chaque nouvelle table si bien entendu elle n'existe pas avec la commande **CREATE TABLE IF NOT EXISTS**. Je définis chaque champ de propriété de la table en précisant son type et ses spécificités.  
(Reprise du dictionnaire des données)

**IF EXISTS** et **IF NOT EXISTS** permettent d'éviter de corrompre la BDD avec des tables créées plusieurs fois par le script. C'est une sécurité d'intégrité de la BDD définie en amont de la création dans le sgbd.

Le type **INTEGER** réclame un nombre entier. **TEXT** a été choisi car c'est une propriété sans limitation de longueur par rapport à **VARCHAR**.

La spécificité **NOT NULL** interdit que le champ soit **vide** lors de l'import des valeurs dans les tables.

**GENERATED ALWAYS AS IDENTITY** permet de définir une valeur incrémentée sur l'**Id**. Je complète à l'aide d'une contrainte (**CONSTRAINT**) définissant le champ **Id** comme clé primaire (**PRIMARY KEY**) afin de lui attribuer un caractère unique et l'impossibilité d'une valeur vide dans cette colonne.

```
CREATE TABLE IF NOT EXISTS public.sentence
(
    id integer NOT NULL GENERATED ALWAYS AS IDENTITY,
    name_id integer NOT NULL,
    verb_id integer NOT NULL,
    adjective_id integer NOT NULL,
    complement_id integer NOT NULL,
    CONSTRAINT sentence_pkey PRIMARY KEY (id),
    CONSTRAINT adjective_fk FOREIGN KEY (adjective_id)
        REFERENCES public.adjective (id) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE CASCADE,
    CONSTRAINT complement_fk FOREIGN KEY (complement_id)
        REFERENCES public.complement (id) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE CASCADE,
    CONSTRAINT name_fk FOREIGN KEY (name_id)
        REFERENCES public.name (id) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE CASCADE,
    CONSTRAINT verb_fk FOREIGN KEY (verb_id)
        REFERENCES public.verb (id) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE CASCADE
);
COMMIT;
```

**INTEGER NOT NULL**, puis on place la contrainte avec **CONSTRAINT adjective\_fk FOREIGN KEY (adjective\_id)**.

**MATCH SIMPLE** permet à n'importe quelle colonne de clé étrangère d'être nulle. Si l'une d'entre elle est nulle alors la ligne n'a pas besoin d'avoir de correspondance pour être référencée.

**ON UPDATE** et **ON DELETE** détermine les actions possibles lors de la mise à jour ou de la suppression.

Ici, à la suppression l'option **CASCADE** permet de supprimer toute référence à la clé dans la base de données en tant que clé primaire ou clé étrangère.

Toute table créée est automatiquement placée dans un **schéma** de la base de données du SGBD nommé "public". Aussi dans le cas présent on peut appeler la table par **public.nom\_de\_la\_table** ou simplement par son **nom**. Ici on utilise **public.sentence**.

Je définis chaque composant d'une phrase par référence à son Id. (**name\_id**, **verb\_id**, **adjective\_id**, **complement\_id**).

Chacun de ces champs est une **clé étrangère (FOREIGN KEY)** de la table **sentence** car il est la clé primaire de chacune des autres tables. On place une **contrainte de clé étrangère sur chaque champ en faisant référence à sa nature et place originelle**.

Exemple pour l'adjectif : on définit le champ de propriété **adjective\_id** en



# DOSSIER PROFESSIONNEL (DP)

## 2. Précisez les moyens utilisés :

Pour la création du MCD, j'ai utilisé l'outil diagrams.net (<https://app.diagrams.net>).

Le MLD et le dictionnaire des données ont été élaborés dans des fichiers .md sur VSCode.

La base de données a quant à elle été générée avec Postgresql.

## 3. Avec qui avez-vous travaillé ?

J'ai travaillé seul sur ce projet durant la spécialisation data au cours de challenges.

## 4. Contexte

Nom de l'entreprise, organisme ou association ➤ *O'clock*

Chantier, atelier, service ➤ *Challenge de formation durant la spécialisation Data.*

Période d'exercice ➤ Du : 22/02/2022 au : 09/08/2022

## 5. Informations complémentaires (facultatif)

(Aucune)

## Activité-type 2

Développer la partie back-end d'une application web ou web mobile en intégrant les recommandations de sécurité.

CP 6 - Développer les composants d'accès aux données.

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

Vers la fin du **socle** de formation que j'ai suivi, et dans l'optique de **synthétiser l'ensemble des notions vues**, il m'a été proposé comme **projet "fil rouge"** la **création d'un Kanban**. Il s'agit d'une *liste de tâches organisées* en fonction de leur *avancement* auxquelles on peut assigner des *catégories*.

Cette **application** était élaborée en **deux composants** : une *partie frontend* en HTML, CSS, Javascript Vanilla, et une *partie backend* sous la forme d'une **API** pour laquelle j'ai conçu la base de données mais également les **composants d'accès à cette dernière**.

Développée en **Programmation Orientée Objet (POO)**, j'ai utilisé l'ORM (Object-relational-Mapping) **Sequelize**. Le rôle de Sequelize est d'être **l'interface naturelle entre l'API et les données stockées dans la base de données** en effectuant des **requêtes** auprès de cette dernière. Cet ORM est **basé sur les promesses** pour Node.js. L'intérêt des promesses est de pouvoir **requêter** la base de données de façon **asynchrone**. La **source interrogée** étant **externe**, on ne sait pas **quand** et **si** elle va nous répondre.

Mise en oeuvre de Sequelize :

```
const { Sequelize } = require("sequelize");

const sequelize = new Sequelize(process.env.PG_URL, {
  define: {
    underscored: true,
    createdAt: "created_at",
    updatedAt: "updated_at",
  },
  logging: false
});

module.exports = sequelize;
```

L'utilisation de Sequelize se fait sous **installation préalable d'un package npm** en tant que **dépendances** dans node.js.

Par ailleurs, la **création d'un point d'entrée** vers la base de données est primordiale pour faire **communiquer le serveur backend avec le sgbd**. C'est la raison pour laquelle j'ai créé un **client de connexion** en javascript avec le code ci-contre. (dbclient.js)

J'importe dans une **constante objet** le module Sequelize via un **require**.

Puis je lance la connexion vers l'adresse URL de la base de données avec l'argument **process.env.PG\_URL** préalablement paramétrée dans un fichier d'environnement.

Le second argument est un objet qui définit les options communes de tous les modèles. **Underscored** permet de passer tous les formats camelCase en snake\_case. Sequelize utilise le format CamelCase dans Javascript mais il est parfois possible que la base de données soit pourvue de colonnes utilisant



# DOSSIER PROFESSIONNEL (DP)

MINISTÈRE CHARGÉ  
DE L'EMPLOI

le format underscore (snake\_case) aussi cette option permet de passer d'un format à l'autre automatiquement.

**CreatedAt** et **UpdatedAt** sont des propriété de Modèles utilisées pour dater une création ou une modification d'entrée dans la base de données.

Par défaut, Sequelize historise en console toutes les requêtes SQL qu'il effectue. La propriété **logging** en **false** permet d'empêcher ce processus.

Une fois la constante de connexion paramétrée, on l'exporte.

## Création de Modèles :

Les modèles utilisés par Sequelize peuvent être apparentés à des patrons permettant d'instancier les objets composées des données de la BDD. Notre Kanban est composé de trois types d'objets : Les listes, les cartes, et les tags qui catégorisent une carte.

Voici le code définissant un modèle de liste (à gauche) et un modèle de carte (à droite) :

```
const { DataTypes, Model } = require("sequelize");
const sequelize = require("../dbClient.js");
class List extends Model{};

List.init({
  name: {
    type: DataTypes.TEXT,
    allowNull: false
  },
  position: DataTypes.INTEGER
}, {
  sequelize,
  tableName: "list"
});

module.exports = List;
```

```
const { DataTypes, Model } = require("sequelize");
const sequelize = require("../dbClient.js");
class Card extends Model{};

Card.init({
  name: DataTypes.TEXT,
  position: DataTypes.INTEGER,
  color: DataTypes.TEXT
}, {
  sequelize,
  tableName: "card"
});

module.exports = Card;
```

Et ci-dessous celui d'un tag (catégorie) :

```
● ● ●

const { DataTypes, Model} = require("sequelize");
const sequelize = require("../dbClient.js");
class Tag extends Model{};

Tag.init({
  name:{
    type:DataTypes.TEXT,
    allowNull: false
  },
  color:DataTypes.TEXT
},{
  sequelize,
  tableName: "tag"
});

module.exports = Tag;
```

Chaque Modèle est conçu suivant la même structure. Pour commencer j'**importe les types de données et modèles** gérés par Sequelize dans un objet **{DataTypes, Model}**.

Puis je **connecte** le modèle à la Base de données en important le code du client de connexion précédemment détaillé (dbClient.js)

Enfin je nomme le modèle. Dans Sequelize cette opération s'effectue par l'utilisation de l'extension d'une classe.

Dans le cas du Tag ci-contre, **class Tag extends Model{};** .

Une fois ces opérations de base effectuées, je peux me lancer dans la **création des détails du Modèle**. Je liste toutes les **propriétés** d'un tag, d'une carte ou d'une liste grâce à la **méthode init** de Sequelize, en omettant pas de préciser les **types de chaque propriétés** grâce au DataTypes importés.

Prenons l'exemple d'une liste :

- Avec List.init je démarre la création de la liste des propriétés du modèle.
- Une liste est définie dans notre application par son nom et sa position. Le type de propriété de name est "text" **name:{type:DataTypes.TEXT, ...}**, Le booléen **false** sur la méthode de vérification **allowNull** oblige l'utilisateur à entrer un nom lors de sa création.
- Le rang de position d'une liste à l'écran est précisé par un nombre entier (INTEGER) avec **position:DataTypes.INTEGER**.
- On connecte le modèle avec la **const sequelize** et on nomme la table concernée (Ici, **tableName:"list"**)

Pour finir, chaque Modèle doit être exporté pour pouvoir être exploité par les autres fonctions de l'application.

**Les relations grâce aux cardinalités en créant des jointures :**

J'ai donné plus haut la définition d'un kanban : Une liste de tâches catégorisées.

On utilise les cardinalités du MCD pour mettre en forme les relations entre chaque table et créer un fichier "index.js" dans le répertoire "models" de l'architecture Sequelize.



# DOSSIER PROFESSIONNEL (DP)

MINISTÈRE CHARGÉ  
DE L'EMPLOI

Ci dessous le code du fichier de jointures index.js :

```
• • •

const List = require("./List");
const Card = require("./Card");
const Tag = require("./Tag");

/* ASSOCIATION 0,N */
ListhasMany(Card,{  
    as:"cards",  
    foreignKey:"list_id"  
});

/* ASSOCIATION 1,1 */  
Card.belongsTo(List,{  
    as:"list",  
    foreignKey:"list_id"  
});

/* ASSOCIATION N;N entre Card et Tag */  
Card.belongsToMany(Tag,{  
    as:"tags",  
    through:"card_has_tag",  
    foreignKey:"card_id",  
    otherKey:"tag_id",  
    timestamps: false  
});

Tag.belongsToMany(Card,{  
    as:"cards",  
    through:"card_has_tag",  
    foreignKey:"tag_id",  
    otherKey:"card_id",  
    timestamps: false  
});

module.exports = { List, Card, Tag};
```

On commence par l'import des modèles List, Card et Tag en faisant un **require** de chaque fichier.

Puis on définit les associations avec des méthodes :

- **Une liste peut avoir plusieurs cartes ou être vide (cardinalité 0, N).**

On utilise la méthode Sequelize **hasMany** pour créer la jointure entre les deux constantes.

**As** permet de définir un alias pour la propriété qui contiendra les cartes.

Enfin on **identifie la clé étrangère** concernée avec **foreignKey**.

- **Une carte appartient uniquement à une liste (cardinalité 1, 1).**

Ici c'est avec la méthode **belongsTo** qu'on va définir la jointure. On choisit toujours l'alias et la clé étrangère.

- **Une carte peut avoir plusieurs catégories et une catégorie peut être assignée à plusieurs cartes (cardinalité N;N).**

L'utilisation de la méthode **belongsToMany** sur les constantes Card et Tag est ici nécessaire pour définir la relation Many to Many. La propriété supplémentaire **through** sélectionne la table de relation au travers de laquelle la jointure est faite dans la base données. On précise les deux clés étrangères pour chaque association.

Je n'oublie pas d'exporter les classes List, Card et Tag du fichier index.js.

## Fonctionnement de Sequelize et ses Classes dans l'application :

Après cette démonstration sur la mise en œuvre et le paramétrage de l'ORM Sequelize et ses classes, je vais à présent vous exposer comment celles-ci manipulent la base de données et par quels moyens elles opèrent.

Les contrôleurs de l'architecture de cette application ont pour vocation avec leurs méthodes de donner l'ordre à Sequelize de Créer (Create), Lire (Read), Mettre à jour (Update), Supprimer (Delete) la donnée.

Ci dessous je prends pour exemple le contrôleur d'une liste :

```
const { List } = require("../models");

const listController = {
    async getAllLists(req, res) {
        try {
            const allLists = await List.findAll({
                include: [
                    {
                        association: "cards",
                        include: [
                            {
                                association: "tags"
                            }
                        ]
                    },
                    {
                        order:[
                            ["position","ASC"],
                            ["cards","position","ASC"]
                        ]
                    };
                ],
                res.json(allLists);
            }
        catch (err) {
            console.error(err);
            res.status(500).json({ error: err });
        }
    },
    async createList(req, res) {
        const list = req.body;
        try {
            if (!list.name) {
                throw "Le nom de la liste doit être précisé";
            }
            const listPosition = await List.count() + 1;
            let newList = List.build({
                name: list.name, position: listPosition
            });
            await newList.save();
            res.json(newList);
        }
        catch (error) {
            errorHandling.log(error);
        }
    },
}
```

On commence tout d'abord par appeler la classe **{List}** de Sequelize.

Le **listController** est composé de nombreuses méthodes du **CRUD**. Pour les besoins de l'explication je ne présente ici qu'une **méthode de lecture des données** de la bdd (**getAllLists**) et une méthode de création (**createList**).

Chacune de ces **méthodes** est développée, pour rappel, de manière **asynchrone** car la source interrogée est externe à l'API. On utilise les propriétés **async** et **await** pour donner à la méthode ce caractère asynchrone.

Par ailleurs, afin de gérer les éventuelles erreurs lors du lancement du code de la fonction, j'ai utilisé des gestionnaires d'exceptions **Try/Catch**. On teste ainsi la commande dans le try, si celle - ci renvoie une erreur elle est alors "attrapée" par le catch qui la signale à son tour par un statut 500 et json.

Pour **getAllLists**, on crée une **constante** qui interroge la base de données avec la **méthode native** **findAll** de Sequelize sur la classe List : **await List.findAll(...)**;

**Include** permet de faire une **jointure** avec les **cartes** et **tags**. **Order** quant à



# DOSSIER PROFESSIONNEL (DP)

MINISTÈRE CHARGÉ  
DE L'EMPLOI

elle effectue le **classement ascendant ("ASC")** dans le cas présent de chaque classe. Si la BDD renvoie un résultat, la constante est restituée au format json à l'application : `res.json(allLits)`, sinon le **catch** renvoie une erreur en console et un **statut 500** ainsi qu'un **json avec une erreur** qui pourra par exemple être utilisé pour avertir l'utilisateur : `res.status(500).json({error:err})`.

Pour **CreateList**, On commence par **récupérer le body** entré par l'utilisateur en front dans une constante. `const list = req.body;`

Si aucun nom pour la liste n'a été inscrit dans le formulaire : `if(!list.name)`, alors on renvoie à l'utilisateur un message l'invitant à le faire : `throw "...".`

On définit la position de la liste par rapport à celles existantes en incrémentant 1 dans une constante avec `List.count()+1`.

Une fois ces deux opérations effectuées, on **génère la nouvelle liste** avec `List.build` avec les propriétés précédemment déterminées `{name:list.name,position:listPosition}`.

On **sauvegarde** la nouvelle liste avec la méthode `.save`, et on restitue le résultat dans un objet json : `res.json(newList);`

## 2. Précisez les moyens utilisés :

Pour ce projet, j'ai utilisé l'ORM Sequelize et le pattern Active Record ainsi que le CRUD.

## 3. Avec qui avez-vous travaillé ?

J'ai travaillé seul sur ce projet fil rouge durant plusieurs challenges au cours de la formation.

#### **4. Contexte**

Nom de l'entreprise, organisme ou association ➤ *O'clock*

Chantier, atelier, service ➤ *Exercice de formation*

Période d'exercice ➤ Du : 22/02/2022 au : 09/08/2022

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

(Aucune)



MINISTÈRE CHARGÉ  
DE L'EMPLOI

# DOSSIER PROFESSIONNEL (DP)

## Activité-type 2

Développer la partie back-end d'une application web ou web mobile en intégrant les recommandations de sécurité.

**CP 7** - Développer la partie back-end d'une application web ou web mobile.

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

Durant ma formation, j'ai été initié au **concept d'architecture de projet** et notamment celle du **Model-View-Controller (MVC)**.

Le **modèle** est le composant qui **contient les données** ainsi que de la **logique** en rapport avec les données. Il **définit** grâce à ses **classes les règles universelles** de l'application. Il peut **manipuler les données** en réalisant les opérations du **CRUD**.

Le **contrôleur** est fait pour opérer sur la **gestion des comportements de l'utilisateur**, il va **modifier** les données de la **vue** et du **modèle**.

Enfin, la **vue** est la **partie visible de l'iceberg** pour l'utilisateur. Elle **contient** tous les **éléments** nécessaires pour permettre à ce dernier une **interaction avec l'application** dans sa globalité. Elle détient toute la **logique** pour **afficher les données du modèle**.

L'architecture MVC est **mise en œuvre** avec l'utilisation d'un **routeur**. Celui-ci recense l'ensemble des **appels réalisables** par le navigateur auprès des **contrôleurs et leurs méthodes** en fonction de l'**opération demandée par l'utilisateur**. (Création, lecture, mise à jour, suppression - **CRUD**)

On définit également au niveau de l'architecture les **outils de sécurité** pour éviter toute **entrée non autorisée** dans l'application et **contrer une éventuelle fuite des données** de l'application.

Voici le routeur élaboré pour le projet Kanban :

```
● ● ●

const express = require("express");
const cardController = require("./controllers/cardController");
const tagController = require("./controllers/tagController");
const listController = require("./controllers/listController");
const errorHandling = require("./middlewares/errorHandling");

const router = express.Router();

/* LISTES */
router.get("/lists", listController.getAllLists);
router.post("/lists", listController.createList);

router.get("/lists/:id", listController.getList);
router.patch("/lists/:id", listController.updateById);
router.delete("/lists/:id", listController.deleteList);

/* CARTES */
router.get("/cards", cardController.getAllCards);
router.get('/lists/:id/cards', cardController.getCardsInList);
router.get('/cards/:id', cardController.getOneCard);
router.post('/cards', cardController.createCard);
router.patch('/cards/:id', cardController.modifyCard);
/*
 * Route pour créer ou modifier une carte
 */
router.put('/cards/:id?', cardController.createOrModify);
router.delete('/cards/:id', cardController.deleteCard);

/* TAGS */
router.get('/tags', tagController.getAllTags);
router.post('/tags', tagController.createTag);
router.patch('/tags/:id', tagController.modifyTag);
router.put('/tags/:id?', tagController.createOrModify);
router.delete('/tags/:id', tagController.deleteTag);
router.post('/cards/:id/tags', tagController.associateTagToCard);
router.delete('/cards/:cardId/tags/:tagId', tagController.removeTagFromCard);

// gestion des 404
router.use(errorHandling.notFound);

module.exports = router;
```

L'architecture de ce routeur est fondée sur le **framework express** de nodeJS.

Un **framework** est un ensemble de **composants logiciels** permettant de **simplifier et uniformiser le développement** d'une application.

Je commence donc par **importer express** dans une constante via la commande **require**.

Le **rôle du routeur** étant de diriger les demandes de l'utilisateur vers les **méthodes des contrôleurs** adéquats, j'importe également chacun d'entre eux ainsi que quelques **middlewares**. Les middlewares sont des **codes intermédiaires fournissant des services** dont l'**application** ne **dispose pas** nativement. (exemple dans le cas présent un service de gestion des erreurs)

Je **lance** ensuite le **routeur** avec la commande **express.router()**;

Enfin je **définis** l'ensemble des **routes potentiellement utilisables**.

Celles-ci sont **accessibles** suivant des **méthodes de requêtes** définies par des **verbes HTTP** en fonction du **type d'opération** que l'on souhaite opérer.

Voici les plus couramment utilisées dans le **CRUD** :



# DOSSIER PROFESSIONNEL (DP)

MINISTÈRE CHARGÉ  
DE L'EMPLOI

- **get** : utilisée pour la récupération de données
- **post** : pour l'envoi de données et générer une entrée supplémentaire.
- **patch** : sert pour la mise à jour partielle d'informations d'un élément existant.
- **delete** : cette méthode est faite pour la suppression de la ressource indiquée.

Lors de l'appel d'une méthode d'un contrôleur, celui - ci prend le relais et effectue l'opération demandée avant de renvoyer le résultat en vue de l'afficher à l'utilisateur.

Je vais vous présenter ici deux méthodes du listController, updateById et deleteOneList appelées via des routes patch et delete :

```
● ● ●

const { List } = require("../models");

const listController = {
    async updateById(req, res) {
        try {
            const listID = req.params.id;
            const list = await List.findByPk(listID);
            if (!list) {
                res.status(404).json("Impossible to retreive the list with this id");
            }
            else {
                if (req.body.name) {
                    list.name = req.body.name;
                }
                if (req.body.position) {
                    list.position = req.body.position;
                }
                await list.save();
                res.json(list);
            }
        }
        catch (error) {
            errorHandling.log(error);
        }
    },
    async deleteOneList(req, res) {
        try {
            const listID = req.params.id;
            const list = await List.findByPk(listID);
            await list.destroy();
            res.json("OK");
        }
        catch (error) {
            errorHandling.log(error);
        }
    }
};

module.exports = listController;
```

Comme on peut le voir sur l'extrait de code du routeur, la **route /lists/:id** fait appel à la **méthode patch** nommée **updateById** du **listController**.

Mais on remarque que la **même route** appelée en **méthode delete** lance la fonction **deleteOneList** toujours du **listController**.

Le **verbe HTTP** est un donc un moyen d'appeler des méthodes différentes d'un même contrôleur pour une route identique. Il définit le type d'opération qui va être effectuée au niveau du CRUD.

Pour updateById on récupère l'Id de la liste grâce à **req.params.id**. On interroge la BDD grâce à Sequelize et sa classe List avec la méthode **findByPk** pour rechercher l'id.

Si elle n'existe pas alors je renvoie une erreur, sinon je mets à jour les infos du nom et de la position de la liste puis je la sauvegarde avec `await list.save();` et je **renvoie les infos mises à jours en json** pour les **actualiser** au niveau de la **vue** affichée à l'utilisateur. Toutes ces **opérations** sont effectuées de façon **asynchrone** bien entendu.

Dans l'introduction je faisais référence au fait que l'architecture devait être définie avec des **règles de sécurité** pour notamment **contrer à tout vol de données** par un utilisateur **tiers**.

La **première méthode d'intrusion** la plus courante est l'**injection SQL**. Elle consiste à **ajouter** dans la requête SQL en cours **un morceau de code non prévu** pouvant **compromettre l'intégrité** de **l'application** et permettre la **prise de contrôle** de celle-ci **par un tiers** qui aurait alors accès à la **base de données**.

Pour éviter cela j'ai appris quelques bonnes pratiques à mettre en œuvre :

- Préparer ses requêtes SQL dans une variable séparément à la fonction
- Utiliser des jetons dans la requête permet de définir l'élément en le détachant du reste du code.

exemple tiré d'un exercice quotidien TrombinOclock :

```
const studentDataMapper = {
  findByPromo: async (id) => {
    const result = await client.query(`SELECT * FROM "${TABLE_NAME}" WHERE "promo" = ${id}`);
    return result.rows;
  },
  add: async (studentData) => {
    const { firstName, lastName, githubUsername, promo } = studentData;
    const sql = {
      text: `
        INSERT INTO student (
          first_name, last_name, github_username, profile_picture_url, promo
        )
        VALUES ($1,$2,$3,$4,$5)
      `,
      values: [
        firstName,
        lastName,
        githubUsername,
        `https://github.com/${githubUsername}.png`,
        promo,
      ],
    };
    const result = await client.query(sql);
    return result.rowCount;
  },
};

module.exports = studentDataMapper;
```

La **première méthode** `findByPromo` de la constante `student DataMapper` contient une **requête SQL “classique”**, les **arguments** ne sont **pas “détachés”** pour la sécuriser dans la constante `result`. Toutes les lignes du résultat sont retournées à l'application.  
`return result.rows;`

Dans la **seconde méthode add** qui permet à l'utilisateur **d'ajouter un étudiant** dans une **constante** via destructuring de ses propriétés `const {firstName, lastName, githubUsername, promo} = studentData` contient une bonne pratique de développement à savoir la

préparation de la requête séparément de la commande d'interrogation à la bdd en JS, mais également l'utilisation des jetons (`$1,$2,$3`, etc...) permettant d'isoler les arguments de la requête



# DOSSIER PROFESSIONNEL (DP)

MINISTÈRE CHARGÉ  
DE L'EMPLOI

pour contrer une tentative d'injection. A savoir que lors de l'utilisation de Sequelize la gestion des requêtes préparées est nativement gérée par l'ORM.

## Contrôle des interactions avec une autre ressource chargée depuis une autre origine :

Une **application web** peut être **développée** aux moyens d'**appels à des applications "externes"** à son architecture propre. Par exemple dans l'application **Okanban**, j'ai développé une **API** gérant toute la **données dans un premier temps**, puis dans un **second temps** j'ai développée l'**application frontend** qui faisait des appel à l'API via la méthode **fetch** pour obtenir les **données au format json** à afficher à l'utilisateur. Ce **type d'appel externe** étant un **risque** pour la **sécurité** de l'application, il existe une **bibliothèque** permettant de **garantir l'origine de l'information reçue** en définissant dans l'architecture du serveur la/les adress(e)s externes d'API pouvant être consultée. La librairie **CORS** gère ces autorisations.

## Comment met-on en œuvre la librairie CORS ?

```
require('dotenv').config();
const express = require("express");
const app = express();
const cors = require('cors');
const multer = require('multer');
const sanitizer = require("./app/middlewares/bodySanitizer");
const router = require("./app/router");

app.use(express.urlencoded({ extended: false }));

const upload = multer();
app.use(upload.none());

app.use(sanitizer);

app.use(express.json());

app.use(cors({origin:"http://localhost:5001"}));

app.use(router);

const PORT = process.env.PORT ?? 3000;

app.listen(PORT, ()=>{
  console.log(`Serveur démarré sur le port http://localhost:${PORT}`);
});
```

Ici le code du serveur de l'application O'kanban. C'est le point d'entrée à l'application.

**CORS** étant une librairie node.js on **l'installe** en tant que **dépendance** par **package npm** dans le terminal. Puis on fait **appel à la librairie** dans la **constante cors** (ci-contre dans notre code exemple) avec un **require**.

On **lance** ensuite le **processus** et on **définit l'adresse autorisée** avec avec **app.use(cors({origin:"http://adresse\_de\_l'api"}));**

Le **contrôle de la same-origin-policy** est géré par CORS.

## Les failles de sécurité de type Cross-Site Scripting (Faille XSS) :

Une **application mal sécurisée** peut permettre à un **utilisateur malveillant d'insérer du code** lui permettant d'**obtenir le contrôle** et avoir une **porte d'accès aux données**. Comment ? Par l'**insertion de code** dans le **formulaire** pour déposer un message dans un forum, ou encore dans la page profil d'un compte en "description". Ces deux exemples énoncés illustrent l'insertion d'un script Javascript malveillant, celui-ci peut également être inséré dans l'url.

On parle alors de **faille de type Cross site scripting - XSS**.

Afin de prévenir ce type de faille de sécurité majeure, Il existe des librairies qui permettent d'assainir les chaînes de caractères envoyées par l'utilisateur. C'est le cas de Sanitizer que j'ai utilisée dans l'application de kanban.

Dans l'exemple de code présenté plus haut, on peut voir les lignes suivantes :

```
...  
const sanitizer = require("./app/middlewares/bodySanitizer");  
  
app.use(sanitizer);
```

J'appelle le **middleware** exécutant **sanitizer** dans une **constante** par l'utilisation de **require**, puis je **lance l'application** avec la commande **app.use(sanitizer)**;

## De quoi se compose le code du middleware bodySanitizer.js ?

```
...  
  
const sanitizer = require("sanitizer");  
  
const bodySanitizer = (req,res,next)=>{  
    if(req.body){  
        for(const property in req.body){  
            req.body[property] = sanitizer.escape(req.body[property]);  
        }  
    }  
    next();  
};  
  
module.exports = bodySanitizer;
```

Je commence par **l'import** de la **librairie** avec un **require**.

Puis je **crée la constante** **bodySanitizer** (celle que **j'importe** dans le fichier **index.js** plus haut)

Si l'utilisateur envoie un formulaire **if(req.body)**, alors **on boucle sur chaque propriété** du body reçu et **sanitizer nettoie** pour chaque **valeur** de chaque **clé**. Une fois le processus terminé on sort du middleware **next()**; pour que le reste du code de l'application se joue.

Je n'oublie pas d'exporter bodySanitizer pour permettre l'import en index.js



MINISTÈRE CHARGÉ  
DE L'EMPLOI

# DOSSIER PROFESSIONNEL (DP)

## 2. Précisez les moyens utilisés :

J'ai utilisé le framework Express et Node.js pour l'architecture Serveur de ce projet.

Pour la prévention des failles de sécurité SQL, j'ai utilisé les bonnes pratiques enseignées de requêtes préparées et jetons SQL. Pour les failles de type XSS j'ai utilisé la librairie Sanitizer et CORS pour garantir l'origine d'un appel à un API externe comme c'était le cas dans Okanban.

## 3. Avec qui avez-vous travaillé ?

J'ai travaillé seul sur ce projet.

## 4. Contexte

Nom de l'entreprise, organisme ou association ➤ *O'clock*

Chantier, atelier, service ➤ *Exercice de formation*

Période d'exercice ➤ Du : 22/02/2022 au : 09/08/2022

## 5. Informations complémentaires (facultatif)

(aucune)

## **Titres, diplômes, CQP, attestations de formation**

*(facultatif)*

<b>Intitulé</b>	<b>Autorité ou organisme</b>	<b>Date</b>
BTS comptabilité et gestion (BAC+2)	ISM La Cadenelle (13)	Juillet 2001



# DOSSIER PROFESSIONNEL (DP)

## Déclaration sur l'honneur

Je soussigné(e) Frédéric GUIOU,

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 Auriol

le 19/09/2022

pour faire valoir ce que de droit.

Signature :

Frédéric GUIOU

## Documents illustrant la pratique professionnelle

(*facultatif*)

Intitulé
Cliquez ici pour taper du texte.



# DOSSIER PROFESSIONNEL (DP)

## ANNEXES

(Si le RC le prévoit)

### CP01 - Maquetter une application

#### -Page Projects (Desktop, Talette, Mobile)

