Mémoire

Formation développeur web et applications mobiles

25/09/2023

TACITE Damien

TABLE DES MATIÈRES

REMERCIEMENTS

Je tiens à porter des remerciements particulièrement à mon tuteur, Robillard Anthony, pour l'aide qu'il m'a apporté autour du domaine du nautisme et de la mécanique, l'écoute et la patience dont il a fait preuve, du temps qu'il a pris à m'expliquer les choses à savoir de son métier, de son atelier et chaque procédures, avec qui nous avons partagé de grand moments de rire et de bonne humeur, mais qui a également parlé de moi à son employeur, Di Gregorio François.

Je tiens également à remercier ma famille ainsi que ma conjointe pour le soutien autour de ce projet professionnel.

Un clin d'œil également à Lemonnier Ludovic, qui est toujours présent avec la bonne humeur et sans qui on ne peut pas passer un moment sans rire.

INTRODUCTION

Le stage se déroule dans une concession & atelier de réparation de bateaux nommé ***Le Havre Nautic***. Il s'y trouve différents hangars comprenant un atelier de réparation de bateaux et moteurs ainsi qu'un Show Room d'expositions de bateaux et moteurs destinés à la vente.

L'atelier est organisé et tenu par le chef d'atelier qui est également mon tuteur, ayant plusieurs employés sous ses ordres. Il leur donne des instructions pour la journée en fonction des réparations à faire. Ils sont également dans l'obligation de se rendre sur le port afin de ramener les bateaux ou les remettre à l'eau, mais également d'analyser les besoins directement sur place.

Lors de l'entretien je suis arrivé avec un projet que j'ai proposé directement au chef d'atelier, car il allait principalement concerner sa partie de travail. Je m'étais préparé avec différentes idées de projet en fonction de ce que répondait le chef d'atelier sur ma première idée de manière à montrer que j'apportais une solution à l'entreprise, et non une demande d'être pris sans rien à proposer.

Ce projet porte sur les entretiens de moteurs, savoir où ils en sont dans les étapes, pouvoir faire en sorte d'ajouter des commentaires si besoin et également avoir la possibilité qu'un collègue prenne la relève et puisse savoir ce qu'il reste à faire sous forme de site web applicatif.

Cette idée était de soulager le temps de travail afin de mieux se répartir les tâches, ne pas perdre de temps à devoir lister à chaque fois ce qu'il reste et ainsi être facilement plus productif.

CAHIER DES CHARGES

Résumé »

Le produit final sera un site web applicatif qui permettra de référencer les informations clients, dont les moyens de contact, les dates de passages dans la concession, entrée et sortie, les informations bateau et moteur. Mais également les informations nécessaires à la vente, car la concession met également en vente bateaux et moteurs. Cela permettra de garder en données les informations pour de futurs entretiens.

Cet outil allégera la charge de travail autant pour la partie des employés dans l'atelier, mais également la partie vente de produits afin de pouvoir répondre facilement au potentiel acheteur. Ils pourront vendre des bateaux ou moteurs à des clients, et ainsi avoir déjà des informations stockées en base de données sur ces fameux produits ou sur les clients lorsque ceux-ci reviendront à la concession pour différents entretiens. Ainsi, tout sera regroupé et plus simple à retrouver. Ils pourront également accéder aux entretiens ajoutés au préalable et faciliter leur charge et temps de travail et mieux se répartir les tâches et se passer le flambeau sur les entretiens qui s'étaleront sur plusieurs jours.

Afin de travailler les données & n'ayant aucun accès au serveur ni base de données car un logiciel pour créer les devis existait déjà, je me suis rapproché de cet outil qui garde des informations clients, bateaux et autres. Ainsi donc j'ai regardé l'interface, lu les différents noms de tableaux, dont le tableau client, bateau, moteurs, ainsi j'ai pu reprendre les données importantes comme le nom et prénom client et j'ai rajouté une partie conformité, prise en charge, entretiens moteurs qui n'étaient pas dans onglets de l'interface. Les employés doivent aller sur les sites des fournisseurs, rentrer les numéros de série afin d'avoir les entretiens moteurs qui y sont liés. J'ai ainsi crée dans la partie administration la possibilité d'ajouter les étapes dans la base de données afin d'éviter de toujours aller sur le site. Elles seront automatiquement dupliquées à chaque nouveau moteurs ayant le même nom de modèle afin de les rendre propre au moteur du client.

Point de vue du client »

- Système de gain de temps,
- Fidélisation clientèle,
- Se retrouver lors d'une demande d'entretien/maintenance,
- Base de données retrouvant les clients, les immatriculations bateau, numéro de série de moteur ou remorques,

- Base de données retrouvant les comptes administrateurs ou employés,
- Contexte où le mécanicien n'est pas à la concession mais a sa tablette ou son téléphone et a besoin d'une information, n'ayant pas accès à son logiciel il y a les données dans l'application

Contrainte(s) »

- Aucun accès à une base de données déjà existante ainsi qu'un serveur déjà existant,
- Si manque d'accès, rendre similaire l'approche des données, ce qui est déjà utilisé,
- Assurer la sécurité des données clients,
- Autonomie dans l'entreprise, ne comporte pas de développeur web,
- Difficultés pour communiquer avec le patron

Maquette »

(Toutes informations correspondant à des données sont totalement fictives, elles ne servent qu'à illustrer)

CONNEXION

Ici, le formulaire de connexion est simple. Il n'y aura pas de formulaire d'inscription car c'est un site intranet. La gestion des employés, de leur identifiant et mot de passe, création de contenu ou de compte, revient totalement au patron qui veut garder le contrôle sur l'application, ses données et son effectif. Il n'y a pas de mot de passe oublié car le patron veut avoir la main mise sur les données.

La partie menu administrateur ne sera visible que par le patron mais également à ceux à qui il attribuera le rôle d'administrateur, comme par exemple la Directrice des Ressources Humaines, ou un futur adjoint. Le menu général sera visible par les employés qui pourront justement faire les recherches importantes concernant les bateaux, moteurs ou informations clients.

RECHERCHE DE CLIENT

Le tableau général de recherche client permet de regrouper tous les clients. Il sera visible par tous les employés, & ainsi lorsqu'un client

viendra à l'atelier ou bien dès lors qu'ils seront au port et devront chercher les informations concernant ce client autant pour emmener son bateau, le contacter, connaitre ses dates de passages ou toutes informations matérielles, tout sera regroupé sur cette page.

La barre de recherche permet justement de trouver le client plus facilement. Cela cache tous les clients qui n'ont pas les informations, caractères ou autres notés pendant la recherche et ainsi pouvoir facilement le retrouver.

FICHE DE CONTROLES D'ENSEMBLE

La partie employée concerne en grande partie le chef d'atelier, et son équipe. Ils pourront ainsi déterminer le client, qui aura le moteur lié à son bateau et choisir les entretiens qui seront à faire en fonction de la période et du moteur qui sera stocké dans la base de données. Ainsi, les données seront transmises dans la partie « étapes restantes » qui est lié à la « prise en charge ».
Les commentaires serviront à savoir des choses importantes, autant que les impacts sur moteur ou bateau, ou bien savoir où sont les clefs ou bien d'autres, afin que tout le monde soit mis au courant et améliorer la communication interne à l'équipe.

PRISE EN CHARGE

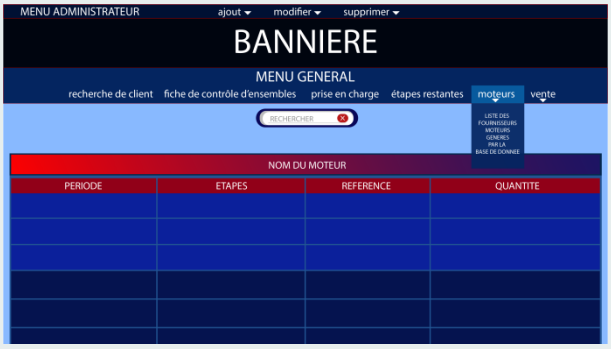
Dans la page de prise en charge nous retrouvons tous les clients et leur matériel, informations importantes dont une barre de progression qui sera calculée sous forme d'algorithme en fonction des étapes validées et des étapes restantes afin de mieux s'organiser et pouvoir prévenir le client de l'avancée des réparations sur son moteur.

ETAPES RESTANTES

Ici nous retrouvons les étapes liées au moteur que nous rechercherons grâce à sa référence, nous validerons les étapes et avec le bouton enverrons la validation.

MOTEURS

Le menu moteur sera généré par le biais de la base de données en récupérant les fournisseurs moteur de l'entreprise. Cela évitera de noter en « brut » et de devoir modifier à même le code les informations.
Les fournisseurs pourront être ajoutés, modifiés ou supprimés par l'administrateur dans l'interface dédié à cela (menu administrateur).



VENTES (Bateaux, Moteurs, Remorques)

Ces pages regroupent les informations importantes sous forme de tableau pour les produits mis en vente. Elles seront facilement trouvables grâce à la barre de recherche et permettra de répondre aux questions clients lors des ventes.

AJOUTS / MODIFICATIONS

Ci-joint, la maquette générale de l'interface que l'on pourrait retrouver sur les formulaires d'ajout ou de modification. (*cf* Annexe *page 34* – Lexique pour (*input, select, bouton radio*).)

Ci-dessous je vais lister ce qui sera attendu en fonction des différentes pages :

AJOUT CLIENT

- *CLIENT* **: (INPUT :** Nom, Prénom, Mail, Téléphone, Emplacement portuaire**)** ;
- *BATEAU CLIENT* **: (INPUT :** Nom, Gamme, Immatriculation, Année, Numéro de série, Longueur, Largeur, Tirant d'air, Tirant d'eau, Nombre de cabines, Nombre de couchettes**)** ;
- *MOTEUR CLIENT* **: (SELECT :** Marque du moteur**), (INPUT** : Nom, Numéro de série, Référence, Puissance, Année de fabrication, Poids, Bruit, Utilisation en année**)**

AJOUT BATEAU

- *BATEAU* **: (INPUT** : Nom, Gamme, Immatriculation, Année, Numéro de série, Longueur, Largeur, Tirant d'air, Tirant d'eau, Nombre de cabines, Nombre de couchettes, Prix**),** **(SELECT :** Marque du bateau, Etat (occasion, neuf), Moteur à rattacher)

AJOUT REMORQUE

- *REMORQUE* **: (INPUT :** Marque, Nom, Référence, PTAC, Charge, Longueur, Largeur, Résistance, Tete, Chassis, Roues, Prix**), (SELECT :** Etat**)**

AJOUT MOTEUR

- *MOTEUR* **: (INPUT :** Nom, Numéro de série, Référence, Puissance, Année de fabrication, Poids, Bruit, Utilisation en année, Prix**), (SELECT :** Marque du moteur, Gamme, Etat**)**

AJOUT D'ENTRETIEN MOTEUR

- *ENTRETIEN* : **(SELECT :** Moteur dont les entretiens seront à ajouter**), (INPUT :** Période (en année), Etape, Référence, Quantité**)**

AJOUT EMPLOYE

- *EMPLOYE* **: (INPUT :** Nom, Prénom, Mail, Téléphone, Mot de passe**), (BOUTON RADIO :** Administrateur ou Employé**)**

AJOUT FOURNISSEUR

- *FOURNISSEUR* : **(INPUT :** Nom du fournisseur**), (SELECT :** Type du fournisseur (bateau ou moteur)**)**

MODIFICATION CLIENT

- *CLIENT* **: (SELECT** : Sélection du client à modifier**), (INPUT :** Nom, Prénom, Mail, Téléphone, Emplacement portuaire**)** ;
- *BATEAU CLIENT* **: (SELECT :** Bateau déjà existant dans la base de données**)** ;
- *MOTEUR CLIENT* **: (SELECT :** Moteur déjà existant dans la base de données**)** ;

MODIFICATION BATEAU

- *BATEAU* **: (INPUT** : Immatriculation, Prix**), (SELECT :** Bateau sélectionné, Etat, Moteur à rattacher)

MODIFICATION REMORQUE

- *REMORQUE* **: (INPUT :** Référence, PTAC**), (SELECT :** Remorque sélectionnée, Etat**)**

MODIFICATION MOTEUR

- *MOTEUR* **: (INPUT :** Référence, Utilisation en année, Prix**), (SELECT :** Moteur choisi, Marque du moteur, Etat**)**

MODIFICATION D'ENTRETIEN MOTEUR

- *ENTRETIEN* : **(SELECT :** Entretien sélectionné**), (INPUT :** Période, Etape, Référence, Quantité**)**

MODIFICATION EMPLOYE

- *EMPLOYE* **: (SELECT :** Employé sélectionné**), (INPUT :** Nom, Prénom, Mail, Téléphone, Mot de passe**), (BOUTON RADIO :** Administrateur ou Employé**)**

MODIFICATION FOURNISSEUR

- *FOURNISSEUR* : **(INPUT :** Nom du fournisseur**), (SELECT :** Fournisseur sélectionné, Type du fournisseur (bateau ou moteur)**)**

SUPPRESSION

Voici un exemple **d'interface de suppression**, on récupère les données attendues afin de ne pas se tromper dans les éléments, nous avons le bouton suppression qui s'affiche et une vérification de si oui ou non on veut supprimer cet élément afin d'éviter les mauvaises manipulations qui sont fréquentes.

Lors de la suppression des employés, on utilise un filtre afin que le profil du patron ne soit pas visible à la suppression.

La partie administration n'apparaîtra que sur le support ordinateur afin d'éviter des soucis possibles de navigation ou autres, avec les écrans tactiles.

Cahier de réalisation »

- Application fonctionnelle afin d'être principalement sur tablette,
- Interface qui permettra de gérer la base de données,
- Site statique qui sera testé en dur,
- Création d'un serveur factice qui permettra le lien entre la base de données factice et le site,
- Adapter les entrées et sorties d'informations,
- Liste d'étapes de réparation en fonction du moteur à entretenir/réparer,
- Intégration des dates de début et fin de réparation et d'entretien,
- Commentaires concernant le moteur s'il y a des choses à savoir en plus,
- Base de données regroupant les clients avec immatriculations bateau, n° de série moteurs ou bateaux, moyen de contact,
- Intégration des informations appareils (bateaux, moteurs, remorques),
- Gestion administrateur en base de donnée à travers l'interface utilisateur par l'employé sans avoir besoin d'être dans le code mais juste en ayant l'accès administrateur de l'application

Le tout a été échangé avec le patron de la concession ainsi que mon tuteur, chef d'atelier, au tout début de stage en prenant compte de leur besoin à partir du projet que j'ai apporté. La priorité est l'application concernant les clients et la répartition des tâches ainsi que la base de données regroupant toutes les informations importantes concernant les clients, leur bateau et également le ou les moteurs.

Contenu du projet »

Le site web applicatif comportera un menu général ainsi qu'un menu administrateur. L'accès à tout contenu se fera obligatoirement par connexion, rien ne sera visible si la connexion ne se fait pas. Celui-ci comportera :

- **U**ne page de recherche client nommée « recherche client » – cette page permettra de facilement retrouver un client et d'y trouver les données le concernant tel que la marque de bateau, l'immatriculation, le moteur, sa marque, ses dates de passage

dans la concession, ses informations personnelles afin de pouvoir le contacter si besoin, son emplacement portuaire.

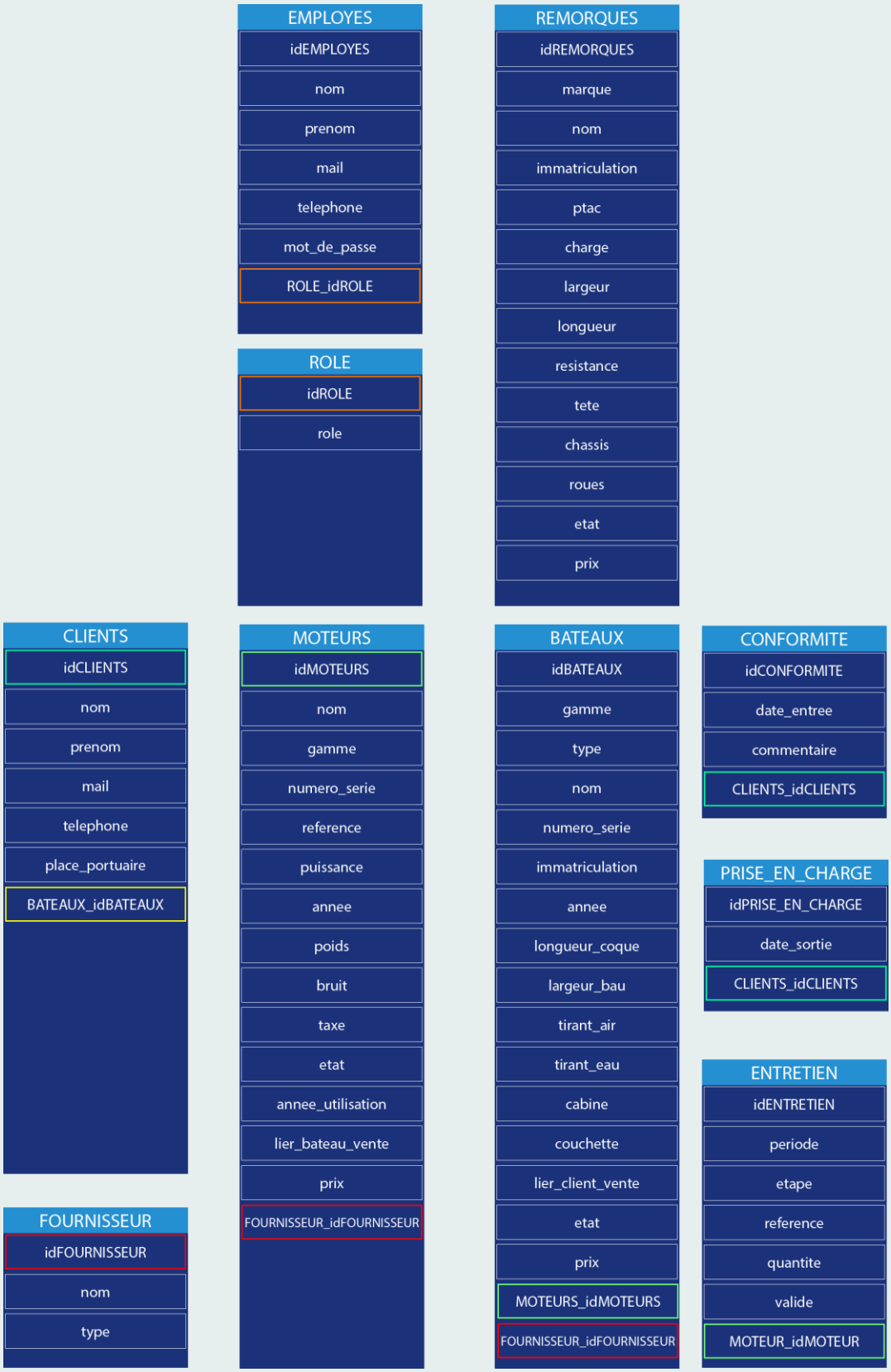
- **U**ne page concernant l'envoi d'un appareil vers la prise en charge nommée « fiche de contrôle d'ensemble » – Cette page permettra de sélectionner un client, de choisir quelle période d'entretien moteur l'atelier devra faire (car il existe des périodes d'entretien moteurs tel que 300 heures (ou 3 ans), 500 heures (ou 5 ans) qui sont obligatoires), afin que celles-ci s'ajoutent dans la prise en charge et qu'on puisse trouver les informations nécessaires.

- **U**ne page concernant les appareils pris en charge nommée « appareils pris en charge » - Cette page permettra d'avoir une liste des appareils pris en charge avec les informations clients, moteurs, bateaux, et une barre montrant la progression des réparations.

- **U**ne page concernant les étapes restantes en fonction du moteur client en récupérant son nom et numéro de série nommée « étapes restantes ». Cette page permettra de voir les étapes restantes et également de les valider vers la base de données afin de savoir où les employés en sont dans les réparations et améliorer l'organisation des choses à faire, répartition de tâches et autres.

- **U**ne page concernant les moteurs nommée « moteurs ». Cette page servira à retrouver les différents fournisseurs moteurs de la concession. Il y sera également listé tous les moteurs et les entretiens appartenant à chaque modèle en fonction des périodes, ainsi que les détails importants tels que quantité, référence. Ils sont utilisés dans la prise en charge afin de ne devoir les noter à chaque fois.

Base de données »



EXPLICATIONS BASE DE DONNEES

1. **Tableau client :** Ce tableau comporte les informations clients tel que son nom prénom, deux moyens de le contacter et également son emplacement portuaire lorsque les employés devront se déplacer là-bas ils auront directement l'information. Il est lié au tableau bateau car il possède un bateau.

2. **Tableau employés :** De même que le client, les informations sur l'employé tel que le nom prénom, un téléphone pour le contacter mais un mail qui permettra également à la connexion ainsi qu'un mot de passe. Il est lié au tableau des rôles.

3. **Tableau rôle :** Permet de donner un rôle à un utilisateur, par exemple administrateur ou employé.

4. **Tableau fournisseur :** Permet de répertorier les fournisseurs liés à l'entreprise, on leur attribue également un type pour savoir si c'est bateau ou moteur. Ainsi, nous avons qu'à les écrire qu'une seule fois et ainsi mieux les gérer.

5. **Tableau remorques :** Ce tableau est utilisé pour la vente, nous n'y avons pas attribué la liaison au fournisseur par choix de la hiérarchie. Il permet de stocker les informations des remorques qui seront mises en vente afin de mieux informer le client et de répondre à de potentielles questions.

6. **Tableau moteurs :** Ce tableau est utilisé pour la vente afin de répondre rapidement aux questions potentielles du client, mais également pour être lié au bateau des clients afin de connaitre ses appareils, mais aussi de pouvoir ajouter des entretiens à ces moteurs afin que l'atelier puisse s'y retrouver dans ces tâches. Il existe une colonne « *lier_bateau_vente* » qui permet d'avoir, sans à ajouter de prix au moteur une possibilité lors de modifications de données ou d'ajout de bateau, la possibilité de lier un moteur à un bateau.

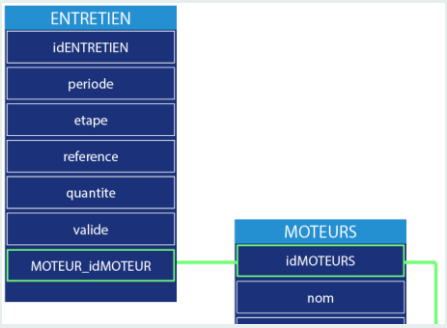
7. **Tableau bateaux :** Ce tableau est utilisé pour la vente afin de répondre aux questions potentielles du client ou d'y répondre rapidement, mais également pour être lié aux clients afin de connaitre ses appareils, son immatriculation ou autre. Il existe une colonne « *lier_client_vente* » qui permettra d'ajouter un bateau, sans y attribuer de prix et pouvoir malgré tout, et le lier à un client lors de modifications de données.

8. **Tableau entretien :** Ce tableau regroupe les entretiens qui auront une liaison avec les moteurs. On y retrouve également la colonne « *valide*» qui permet à l'interaction avec la prise en charge de l'atelier. Sachant qu'ils seront liés à un moteur, ils sont donc liés à un client. Ainsi les entretiens seront affichés lorsqu'ils ne seront pas valides mais une fois effectués alors cette colonne servira à mettre à jour cette partie grâce à 1 pour « non valide » et 0 pour « valide ».

9. **Tableau prise_en_charge :** Ce tableau permet de donner la date de sortie de l'appareil pris en charge par l'atelier et ainsi avoir une trace du passage d'un client car il y a une liaison avec le tableau clients.

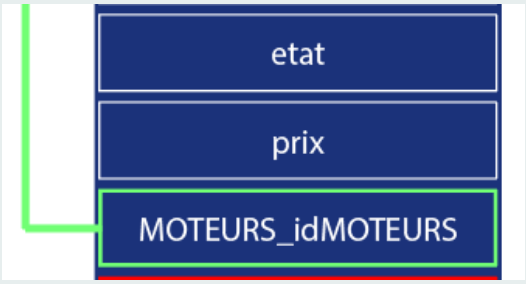
10. **Tableau conformité :** Ce tableau permet d'ajouter un appareil dans l'atelier, pouvoir avoir la date d'entrée mais également de mettre des commentaires si besoin. Sachant qu'il y a une limite de caractères pour stocker les données, cette colonne a été modifiée pour pouvoir stocker plus de caractères que les autres.

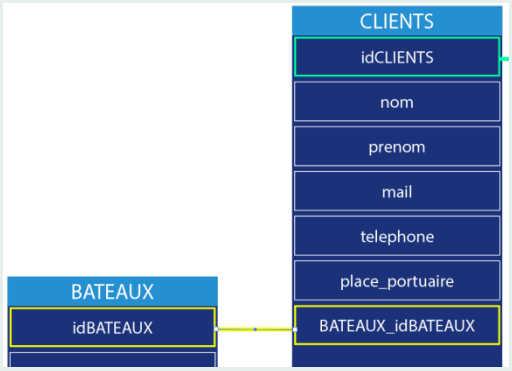
INTERACTION POUR LA PRISE EN CHARGE ET ENTRETIENS MOTEURS



Ici nous avons un tableau comportant les entretiens qui sont liés à un moteur en particulier.

Ce même moteur est lié à un bateau dans la table bateaux afin d'avoir une liaison.



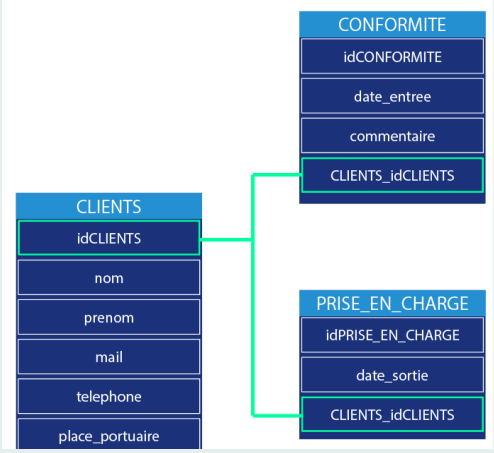


Ce même tableau bateau a une liaison avec le tableau clients afin de savoir quel bateau appartient à quel client.

Tout ceci permet d'abord d'avoir les informations sur un client, son bateau, son moteur et les entretiens qui sont liés au moteur.

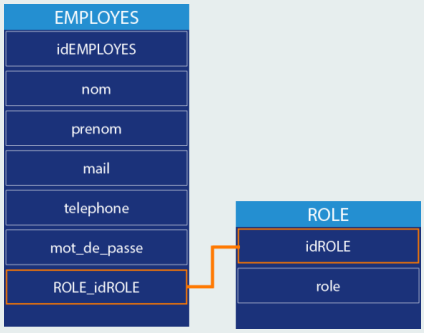
Ensuite en récupérant le client nous l'envoyons vers la conformité en fonction de la période d'entretien qui doit être faite dans le tableau entretien, on y ajoute la date d'entrée et les commentaires nécessaires s'il y a des choses en plus à savoir ou entretenir.

Ensuite une fois terminé on l'envoie dans le tableau prise en charge afin d'avoir sa date de sortie.



Voir l'interaction entière (*cf Annexe page 34 – Interaction base de données*)

INTERACTION POUR LES ROLES UTILISATEURS

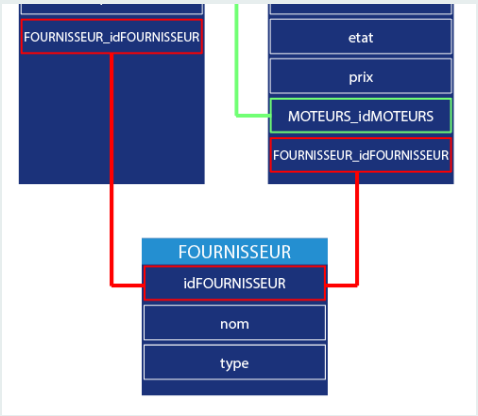


Ici, il suffit d'intégrer les rôles qu'on veut attribuer aux différents utilisateurs, par exemple dans cette entreprise un administrateur et des employés. Cela permet de donner différents accès à notre site en fonction du rôle.

INTERACTION POUR LES FOURNISSEURS

Ici, la liaison permet d'intégrer différents fournisseurs et de leur donner le type. Dans notre entreprise ce sera bateau ou moteur.

Grâce à cela dans les données nous pouvons directement retrouver le fournisseur du bateau ou du moteur et générer soit par exemple un menu dynamique où nous répartirons les différentes marques moteurs, leur gamme et leurs entretiens afin de mieux s'y retrouver.



Voir l'interaction entière (*cf Annexe page 34 – Interaction base de données*)

COMPARATIF

Explications des changements

(*cf Annexe page 34 – Lexique*)

Lors de la conception du site web applicatif, je suis parti sur le langage **HTML CSS** très basique. Dès lors, j'ai tenté de créer ce que j'avais proposé plus haut dans le cahier des charges. Par la suite, lors de l'apprentissage de l'utilisation du **Framework ReactJS**, j'ai décidé de reprendre le projet de zéro afin de comparer la différence entre les deux et quelle manière de travailler me plaisait le plus et était la plus rentable en terme d'ergonomie.

Je suis resté sur le **Framework ReactJS**, l'importation et l'exportation de composants, leur réutilisation, la facilité d'utilisation de paramètres qui pourront récupérer des données et les transmettre à mes composants m'étaient d'une plus grande facilité pour le site applicatif que je créais.

Plus bas un comparatif visuel du prototype et de la version finale, qui montrera principalement mon amélioration à l'utilisation du langage **CSS** mais également ce qu'a pu m'apporter comme idées de conception l'utilisation de **ReactJS**.

Les facteurs ont été le test utilisateurs finaux. Le prototype fut tester par les utilisateurs finaux et j'ai relevé plusieurs difficultés, manque d'ergonomie, difficulté d'utilisation, mal organisé, mais aussi le fait que je pouvais regrouper plusieurs utilisations en un pour le menu et ainsi que cela reste intuitif.

Voici quelques parties essentielles qui ont changé et ainsi permettre une meilleure prise en main :

1. Les changements entre la liste des étapes de l'appareil, la validation, la progression et la liste des appareils pris en charge.
2. La création du menu dynamique, généré par les informations en base de données, qui permet à l'affichage des marques de moteur qui seront gérés par l'administrateur.

ETAPES D'APPAREILS & PRISE EN CHARGE

Ici, le fait de devoir jongler entre deux pages, devoir faire la recherche d'un moteur pour connaître ses étapes restantes, les valider. J'ai décidé de m'inspirer de ce test pour améliorer et finir par tout regrouper en un seul format qui sera plus ergonomique pour les utilisateurs finaux.

Même chose pour les pages regroupant la recherche client ou faire sa conformité, j'ai préféré regrouper. Ce qui fait que le menu se retrouve réduit, moins compact et fourni et plus facile d'entrée à retrouver ce qu'on recherche pour les utilisateurs.

MENU MOTEUR

Lors du prototype, nous devions d'abord cliquer sur « moteurs » pour générer la liste des moteurs. Ensuite, une fois qu'on avait la liste le tableau apparaissait mais vide, et il fallait choisir la marque pour faire apparaître les moteurs. Ce qui était une perte de temps et d'ergonomie.

Tandis que dans la version actuelle, le menu est généré automatiquement peu importe où l'on se situe sur le site, et le tableau se génère avec ses données lorsque l'on choisit la marque moteur.

CONCLUSION COMPARATIF

Dans ce comparatif, j'ai cherché à montrer la différence entre un prototype avec lequel j'ai mis en place les idées premières, un design premier et une version plus aboutie et plus améliorée autant dans l'ergonomie que dans l'aspect.

Sachant que dans l'utilisation, elle, se fait de base sur tablette et par demande de dernière minute, s'est vu être rapidement sur téléphone, il sera plus intuitif de s'y retrouver.

EXEMPLES DE CODES

Note : Les noms de variables sont en franglais, cela m'a permis de mieux comprendre ce que je faisais en avançant. Mes futurs projets, tel que celui de l'examen blanc fourni par l'organisme de formation est intégralement en anglais, sont et seront tous dans une seule et même langue.

Merci pour votre compréhension.

Fonction d'affichage

Je compte vous présenter ci-dessous différentes parties de code qui me semblent intéressantes et qui permettent une meilleure gestion du site web de part le fait qu'ils soient dits génériques, c'est-à-dire qu'elles ne seront écrites qu'une fois, ensuite on les exporte, on leur attribue des paramètres, et on pourra les appeler pour les utiliser, éviter les copier coller pour seulement deux changements. Ce sera plus facile à comprendre avec des exemples et des explications au fur et à mesure :

```
export const fetchAffichage = async (Detail, id = null) => {

const ipAddress = window.location.hostname;
const url = id ?
`http://${ipAddress}:5005/api/${Detail}/${id}`
              :
`http://${ipAddress}:5005/api/${Detail}` ;

const response = await fetch(url, {
  method:'GET',
  credentials: 'include'
});

const result = await response.json()
return result;
};
```

Voici une **fonction** (*cf Annexe page 34 – Lexique*) et ses deux paramètres : ***Detail*** & ***id***. On les réutilisera plus bas dans le code. Elle est exportée afin d'être réutilisée sans copier coller.

Detail est une chaîne de caractères qui représente le type de données à afficher, et ***id*** **est un identifiant optionnel** qui permet de rajouter au chemin d'accès un élément spécifique à afficher.

Ici nous vérifions sur quelle adresse le site est lancé dans la **variable ipAddress**, et plutôt que de l'écrire en code **0.0.0.0** ou **localhost** à chaque fois il se fait automatiquement.

La variable **url** est définie en fonction de la valeur de ***id***. Si ***id*** est spécifié, la variable **url** contiendra un URL avec ***l'adresse IP***, le ***Detail*** et l'***id***, sinon elle contiendra un URL avec seulement l'adresse IP et le ***Detail***.

Ce **chemin** côté client permet de faire comprendre au code à quel **chemin** côté serveur il est lié. (*cf Annexe page 34 – Support de code*)

Cette partie permet d'attendre le retour du serveur, lui dit qu'on attend une méthode d'affichage (**GET**), le ***credentials*** **:*include*** attend une personne connectée, ce qui permet de donner la visibilité qu'aux personnes qui ont accès au site et ont un site dans la base de données. On crée un accès dans le serveur et il est ainsi récupéré ici et on retourne (**return**) les données qui sont attendues.

.json() est le nom du format envoyé par le serveur.

Voici des exemples de l'importation de la fonction **fetchAffichage** juste au-dessus, un format sans le paramètre id et un format avec.

Le premier sert à afficher les clients tandis que le second sert à afficher des étapes d'un moteur qui est lié au client.

(*cf Annexe page 34 – Ancien code* pour voir à quoi cela ressemblerait si la fonction n'était pas générique, réutilisable.)

```
import React, { useEffect, useState } from "react";

import { fetchAffichage } from
"./fonctionsAPI/AffichageAPI";
import { ValiderEtapes } from
"./fonctionsAPI/ValidationEtapes";

function EtapeMoteurs({ id }) {


  const [donnees, setDonnees] = useState([]);

  useEffect(() => {

    const getDonneesMoteurs = async () => {
      const moteursData = await

fetchAffichage('priseEnCharge', id);

      setDonnees(moteursData);
    };
    getDonneesMoteurs();
  }, []);


  const etapesNonValides = donnees.filter((item) =>
item.valide !== 1);

  return (
    <div className="EntretiensMoteurs">

      <h4>Etapes moteur</h4>

      {etapesNonValides.map((item, index) => (

        <div key={index}

className={`EntretiensMoteurs ${item.idENTRETIEN}`}>

          <p>{item.etape}</p>
          <p>{item.reference}</p>
          <p>Quantité : {item.quantite}</p>
```

Ces lignes de code utilisent la bibliothèque ReactJS (*cf Annexe page 34 – Lexique*)**. Elles définissent un composant fonctionnel qui sera utilisé dans l'application.**
La première ligne importe deux fonctions, **useEffect** et **useState**, de la bibliothèque **ReactJS**. Ces fonctions sont utilisées pour gérer l'état et les effets attendus dans les composants.
Ensuite nous importons "*fetchAffichage*" et "*ValiderEtapes*". La première est celle plus haut, tandis que l'autre sert à valider certaines étapes des entretiens moteurs.

Cette ligne de code définit une fonction appelée "*EtapeMoteurs*". Cette fonction prend le paramètre **id**, de manière déstructurée *(fait référence à une syntaxe qui permet d'extraire des valeurs spécifiques).*

Cette ligne déclare une variable (*cf Annexe page 34 – Lexique)* d'état appelée "*donnees*" et une fonction appelée "*setDonnees*", ce qui permettra de mettre à jour cette **variable** d'état. La **valeur initiale** de "*donnees*" **est un tableau vide => ([])** afin de pouvoir, par la suite, y ajouter les données attendues.

Cette fonction effectue une requête asynchrone *(cf Annexe page 34 – Lexique)* à une **API** à l'aide de la fonction "*fetchAffichage*". Elle passe **deux paramètres** à cette fonction : la chaîne de caractères *'priseEnCharge'* et la valeur de l'identifiant "*id*".
Une fois que la **réponse** de **l'API** est obtenue, la **fonction** "*setDonnees*" est utilisée pour mettre à jour la variable d'état "*donnees*" avec les **données renvoyées** par **l'API**.

Cette ligne permet de filtrer les étapes. Ainsi, toutes les étapes qui sont déjà validées ne seront plus affichées. Cette *variable* se nommera **etapesNonValides**.

Nous parcourons les données filtrées afin de pouvoir récupérer les éléments contenus dans la base de données que nous **pourrons afficher en-dessous**.

En utilisant ***item***, qui fonctionne comme une *clef*, on accède aux données grâce à **des noms de tableaux** dans la **base de données** (*comme ici etape, reference, quantite*) afin d'afficher les éléments attendues du côté navigateur.

```
        <button onClick={() => {
          if (window.confirm('Êtes-vous sûr de
vouloir valider cette étape ?')) {
            ValiderEtapes(item.idENTRETIEN);
          }
        }}

className="ButtonEntretien">Valider</button>
        </div>
      ))}
    </div>
  );
}
```

Bouton de validation qui permet de valider une étape **lorsqu'elle sera effectuée. On récupère l'identifiant de l'étape** que l'on veut valider afin d'être sûr de bien choisir celle cliquable.

Une fenêtre s'ouvre afin de demander à l'utilisateur s'il est sûr de vouloir valider l'étape ou non. Si oui, alors la fonction de validation se met en place sinon la page s'affiche de nouveau et l'étape n'a pas bougée.

Pour voir le code initial avant qu'il ne soit générique (*cf Annexe page 34 – Ancien code*)

Afin de revenir sur l'idée de la sécurité, lors de l'accès à ces données et montrer comment elles sont protégées, voici les différentes explications. Voici côté serveur comment sont vérifiés les utilisateurs lors de la connexion :

```
    async function loginData(req, res,
userId, password) {

      const { mail, mot_de_passe } =
req.body;

      const [rows] = await
connexion.query(
          `SELECT * FROM employe WHERE
${userId}=? AND ${password}=?`,
          [mail, mot_de_passe]
      );

      if (rows.length === 1) {
          const { prenom } = rows[0];
          const token =
generateAccessToken(mail);
          req.session.token = token;
```

Création de la **fonction** de **LoginData**, qui permettra de vérifier si l'utilisateur est bien stocké en base de données.

Récupération de l'email et du mot de passe envoyé depuis le formulaire de connexion.

Nous envoyons notre instruction vers la base de données et la stockons dans une *variable* nommée **rows**. Celle-ci est **entre deux crochets car la réponse sera sous forme d'un tableau**.

Nous avons **ici une condition**, **SI la taille du tableau renvoyé** (expliqué plus haut) **comporte une ligne, c'est que l'utilisateur est bien trouvé et existe**. Ainsi, nous récupérons son prenom, et nous créons ce que nous appelons un **token**.

Un **token** est une sorte de clé d'accès, ce sera comme **un mot de passe d'accès généré aléatoirement et qui sera propre à l'utilisateur lors de sa connexion**. Ainsi, le navigateur vérifiera, grâce à **include** vu dans le code précédent, si cette clé est similaire à celle crée.

Afin d'être **correctement vérifiée** à chaque fois, on la **stocke** dans une *session*.

```
            res.json({ retour: "OK",
message: `Bienvenue à vous, ${prenom}` });
        } else {
            res.json({ retour: "PAS OK" });
        }
    }
```

Nous renvoyons le ***retour*** avec comme réponse **« OK »** afin de donner l'accès côté navigateur. Ensuite nous récupérons le prénom pour souhaiter la bienvenue à l'utilisateur.

Si l'utilisateur n'est pas bon, alors il renvoie « PAS OK » ce qui **ne donnera aucun accès au reste du site**.

CONNEXION ET SECURITE

```
async function Login() {

    const mail =
document.getElementById('emailUser').value
    const mot_de_passe =
document.getElementById('mdpUser').value
```

Récupération des valeurs écrites **dans les champs de connexion**, l'email ainsi que le mot de passe. On les stocke pour la suite dans des variables, **mail** et **mot_de_passe**.

```
    let userData = {
      mail: mail,
      mot_de_passe: mot_de_passe,
    };
```

Stockage des données récupérées afin de les envoyer vers le serveur plus bas.

```
    const ipAddress =
window.location.hostname;

    const url =
`http://${ipAddress}:5005/api/connexionUtilis
ateur`;
```

Chemin qui renvoie **vers le serveur** afin de lier ce **formulaire** à la fonction de connexion qui est dans le serveur montrée précédemment.

```
    try {

      let response = await fetch(url, {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json',
        },
        credentials: 'include',
        body: JSON.stringify(userData)
      });
```

Voici les **instructions** à notre **navigateur** afin **d'envoyer les données récupérées**. Nous ajoutons le format attendu dans le serveur (*JSON*) et nous renvoyons les données récupérées dans le **body**.

Afin de récupérer l'information du serveur si l'utilisateur est dans la base de données, nous stockons ces **instructions** dans une variable nommée **response.**

```
    const result = await response.json();
```

Nous attendons la réponse du serveur et stockons ça dans une variable nommée **result**.

```
      if (result.retour === 'OK') {

localStorage.setItem('MessageConnexion',resul
t.message)


localStorage.setItem('isAuthenticated',
'true');

        window.location.href =
"/recherche_client";
```

Ici, nous ajoutons une condition pour la connexion. Si le serveur renvoie bien le fameux **OK** attendu, alors en plus de la clé **token** créée dans le serveur, nous créons :

- Un stockage dit local dans le navigateur afin de renvoyer le message de Bienvenue avec le prénom,
- Un stockage local d'authentification en retournant qu'il est vrai.
- Nous envoyons l'utilisateur sur la page de recherche de client.

Ainsi en créant ces stockages locaux côté navigateur mais également ceux du serveur, nous avons une double sécurité.

```
      } else {

        const messageElement =
document.getElementById("message");


messageElement.classList.add("error");

        messageElement.innerHTML = "Veuillez
vérifier vos identifiants";
      }
    } catch (error) {
      console.error(error);
    }
  }
```

A l'inverse, si l'utilisateur n'est pas dans la base de données ou s'est trompé dans ses identifiants alors nous renvoyons un message pour dire de vérifier les identifiants et il n'a accès à aucunes données et n'est pas redirigé vers une autre page.

Voyons maintenant côté serveur comment sont utilisés les vérifications d'utilisateur afin de donner les accès aux données, et nous verrons ensuite côté navigateur avec le stockage en local comment nous pouvons l'utiliser afin de bloquer les accès au site.

```
  function authenticateToken(req, res, next)
{
```

Nous créons cette fonction d'authentification qui servira à vérifier l'utilisateur pour l'affichage des données. Cette fonction est appelée couramment **middleware**.

```
      const token = req.session.token
```

Nous récupérons **la clé token** crée lors de la connexion et la stockons dans une **variable** afin de faire nos vérifications.

```
      if (token == null) {
        return res.sendStatus(401);
      }
```

Ici, si nous n'avons pas de **clé token** créée alors on renvoie un **status 401** qui est celui pour dire que **l'utilisateur n'est pas autorisé.**

```
 jwt.verify(token,
process.env.ACCESS_TOKEN_SECRET, (err,
token_decode) => {

          if (err) {
              return res.sendStatus(401);
          }

          if (token_decode.exp < (Date.now()
/ 1000)) {
              return res.sendStatus(401);
          }

          if (!token_decode.mail) {
              return res.sendStatus(401);
          }

        next();
      });
    }
```

Grâce à un module installé du nom de **jwt**, nous vérifions que la **clé token** générée dans le serveur correspond bien à celle générée précédemment. Ensuite, nous allons créer les conditions ci-dessous.

S'il y a une **erreur** lors de la **vérification**, alors **l'utilisateur** est de nouveau envoyé en « **non autorisé** ».

Si la **clé token** est expiré, car il y a un temps d'utilisation, alors de même **il ne sera plus autorisé à accéder aux données et devra se reconnecter.**

Si **l'adresse mail** n'est **pas similaire** à celle que nous avons **utilisée lors de la connexion**, alors l'utilisateur ne sera **pas autorisé à accéder aux données.**

Si toutes ces conditions sont passées, alors ça veut dire que l'utilisateur a une **clé token** et pourra accéder aux données, alors nous passerons à la suite.

Voyons maintenant **comment est utilisé ce fameux middleware** afin de faire la vérification dans nos affichages de données, par exemple pour celui des étapes que nous avons montrées plus haut.

```
app.get('/api/priseEnCharge',

authenticateToken,

 async function affichagePriseEnCharge(req,
res)
```

Ici nous écrivons que ce sera une méthode dite « **get** » qui permet à l'affichage des données, et nous donnons **le chemin qui communiquera entre le serveur et le navigateur.**

Ici notre fameux **middleware** qui sert à bloquer l'accès si l'utilisateur n'est pas autorisé.

Fonction qui donnera les instructions d'affichage vers la base de données.

Voyons maintenant de quelle manière et ce que nous faisons du **stockage local** de la connexion.

Comme montré précédemment, nous avons crée des **localStorage** ou **stockage locaux** avec des noms et des valeurs. Voici leurs utilisations, par exemple la première sert à afficher ou non le menu mais seulement si l'utilisateur est connecté :

```
const [isConnected] =
useState(localStorage.getItem('isAuthenticated')
=== 'true');
```

Nous récupérons le **stockage local** montré précédemment dans la partie de connexion côté navigateur nommé « *isAuthenticated* » sur la valeur enregistrée « *true* ».

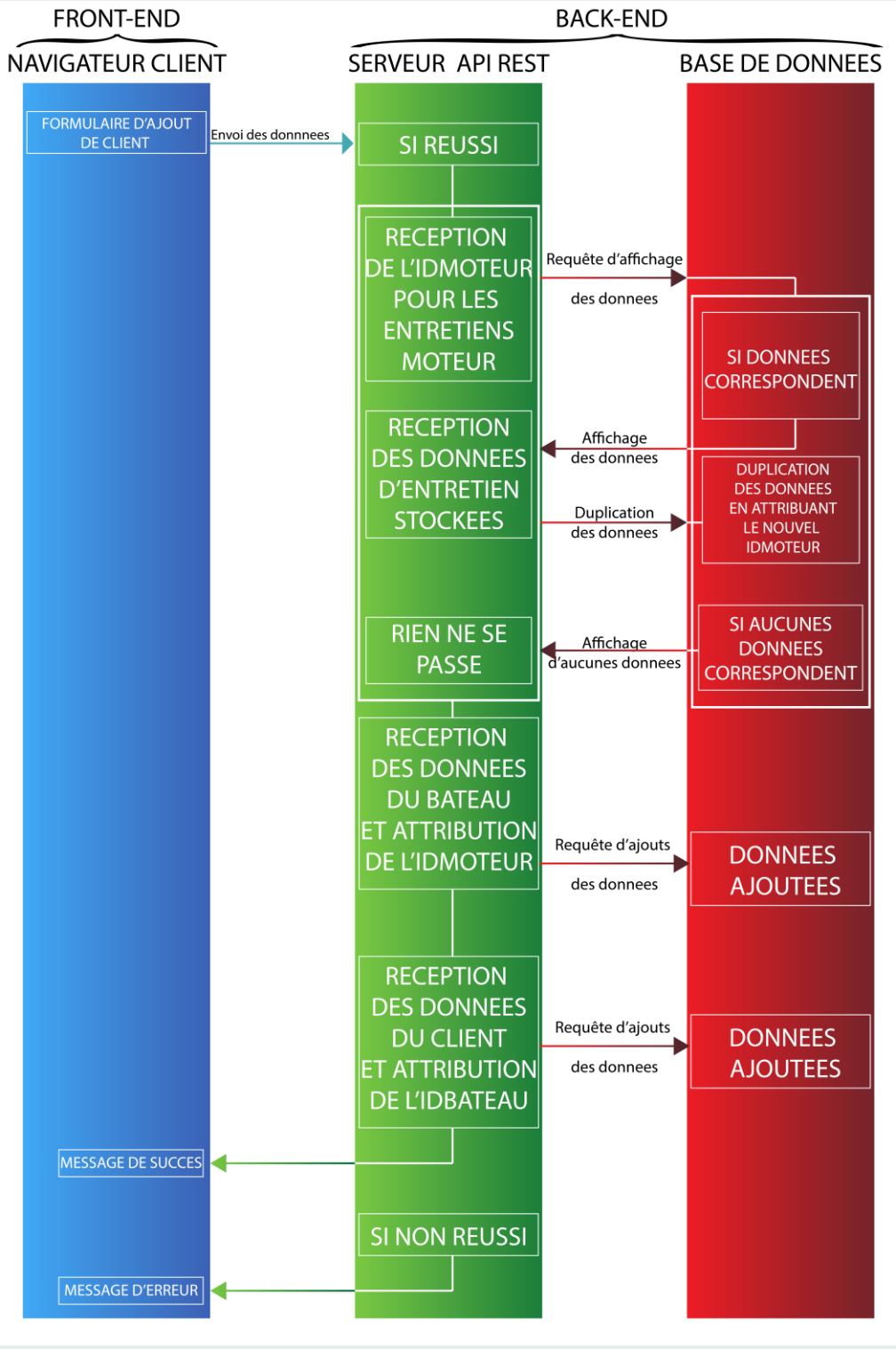
Si cette valeur est vraie, donc que l'utilisateur est connecté et a donc passé toutes les conditions du côté serveur, alors on stocke sa connexion dans cette variable nommée **isConnected** grâce au module, déjà utilisé dans les autres exemples de code, **useState**.

```
{isConnected && <Menu />}
```

Ainsi, grâce à cette **variable** qui **stocke l'information** du fait que **l'utilisateur est connecté** et donc a **passé toutes les conditions**, nous pouvons dire que si l'utilisateur est connecté **nous affichons le composant qui contient notre menu.**

INTERACTION DES DONNEES

AJOUT D'UN CLIENT



SITUATION DE RECHERCHE

Lors de différentes situations je voulais utiliser le système de condition directement dans mon rendu de code. L'une concernant directement le bouton de connexion ainsi que celui de déconnexion. L'autre concerne l'affichage des données. Si aucune donnée n'était disponible, plutôt que cela ne retourne rien du tout je voulais afficher au moins le message « Aucune données ».

Pour cela, j'ai recherché sur internet comment faire en sorte de pouvoir donner des conditions directement dans mon rendu visible mais qui elles, ne soient pas visibles par les utilisateurs que ce soit de simples fonctionnalités. Je suis alors tombé sur ce site (source) :

http://www.hackingwithreact.com/read/1/7/how-to-write-if-else-conditional-statements-in-jsx

TRADUCTION DE LA RECHERCHE

```
return <p>Hello, {chance.first()}!</p>;
```

Everything inside those braces is ES6 code and will be executed then have its results put back into the render. In this case, it means you'll see random names every time you refresh your page.

However, that code is actually a very specific kind of code called an *expression*, which very roughly means that it can be translated directly into a value. This is in comparison to another kind of code called a *statement*, which is where you can for example create variables or perform some other kind of action.

I realize this distinction might seem insignificant, but trust me: it's important. The reason it's important is because you can only use expressions inside JSX braces, not full statements. For example, `{this.props.message}` and `{chance.first()}` are both valid, but something like this is not:

```
{if (chance.first() === 'John') { console.log('Got John');
} else { console.log('Got someone else'); } }
```

return Bonjour, {chance.first()} ! ;

Tout ce qui se trouve entre ces accolades est du code qui sera exécuté, puis ses résultats seront insérés dans le rendu. Dans ce cas, ça signifie que vous verrez des noms aléatoires à chaque fois que vous actualisez votre page.

Cependant, **ce code est en réalité un type très spécifique de code appelé une *expression***, ce qui signifie approximativement qu'il **peut être directement traduit en une valeur**. Cela contraste avec un autre type de code appelé une instruction, où vous pouvez par exemple créer des variables ou effectuer une autre action.

Je comprends que cette distinction puisse sembler insignifiante, mais croyez-moi : elle est importante. La raison pour laquelle elle est importante est que vous ne pouvez utiliser que des expressions à l'intérieur des accolades **JSX** (*cf Annexe page 34 – Lexique*), pas des instructions complètes. Par exemple, **{this.props.message}** et **{chance.first()}** sont tous deux valides, mais quelque chose comme ceci ne l'est pas :

{if (chance.first() === 'John') { console.log('Got John');
} else { console.log('Got someone else'); } }

(If you were wondering, `===` is the recommended way of comparing values in JavaScript; if you use `==` right now you should probably switch, because there's a big difference between "truth" and "truthy".)

Now, you might very well think "I'm glad that kind of code isn't allowed, because it's so hard to read!" And you'd be right: you can't write if/else statements inside JSX. However, JavaScript is (very loosely) a C-like language, which means it has inherited conditional syntax that lets you do if/else as an expression.

Translated, that means there's a way to rewrite that above statement as an expression, which in turn means you can use it inside JSX. However, you should steel yourself for some pretty grim syntax. Here it is:

src/pages/Detail.js

```
{chance.first() === 'John' ? console.log('Got John')
: console.log('Got someone else') }
```

(Si vous vous posiez la question, === est la méthode recommandée pour comparer des valeurs en JavaScript ; si vous utilisez actuellement ==, vous devriez probablement passer à ===, car il existe une grande différence entre la "égal" et "strictement égal".)

Maintenant, vous pourriez très bien penser : "Je suis content que ce type de code ne soit pas autorisé, car il est si difficile à lire !" Et vous auriez raison : vous ne pouvez pas écrire de condition if/else à l'intérieur de JSX. Cependant, JavaScript est un langage de type C, ce qui signifie qu'il a hérité d'une syntaxe conditionnelle qui vous permet d'utiliser if/else en tant qu'expression.

En traduction, cela **signifie qu'il existe un moyen de réécrire cette déclaration ci-dessus en tant qu'expression, ce qui signifie à son tour que vous pouvez l'utiliser à l'intérieur de JSX.** Cependant, préparez-vous à une syntaxe plutôt sombre. La voici :

src/pages/Detail.js

```
{chance.first() === 'John'
? console.log('Got John')
: console.log('Got someone else') }
```

If I write it out on multiple lines it will probably help:

src/pages/Detail.js

```
{
    chance.first() == 'John'
    ? console.log('Got John')
    : console.log('Got someone else')
}
```

You can put that into your component if you want to, but it's just an example for demonstration – we'll be removing it shortly.

The opening and closing braces are old, but the bit in the middle is new and is called a *ternary expression* because it's made up of three parts: the condition (`chance.first() == 'John'`), what to use if the condition is true (`console.log('Got John')`) and what to use if the condition is false (`console.log('Got someone else')`).

The important part is the question mark and the colon: the "true" section comes after the question mark, and the false part comes after the colon. It's not at all obvious, and it really does make for ugly code, but it is absolutely allowable inside JSX and so, like it or not, you'll see ternary expressions all over the place in React code.

src/pages/Detail.js

```
{
        chance.first() == 'John' ?
                console.log('Got John')
        : console.log('Got someone else')
}
```

Vous pouvez mettre cela dans votre composant si vous le souhaitez, mais c'est juste un exemple à des fins de démonstration - nous le supprimerons bientôt.

Les accolades d'ouverture et de fermeture sont anciennes, mais la partie du milieu est nouvelle et s'appelle une expression ternaire car elle est composée de trois parties : la condition **(chance.first() == 'John')**, ce qu'il faut utiliser si la condition est vraie **(console.log('Got John'))** et ce qu'il faut utiliser si la condition est fausse **(console.log('Got someone else')).**

La partie importante est le point d'interrogation et les deux-points : la section "**true**" vient après le point d'interrogation, et la partie "false" vient après les deux-points. Ce n'est pas du tout évident, et cela donne vraiment un code moche, mais c'est absolument autorisé dans **JSX** et donc, que vous l'aimiez ou non, vous verrez des expressions ternaires partout dans le code **React**.

Worse, you'll often see double or even triple ternary expressions where the question marks and colons stack up higher and higher to form a true/false tree. These are also allowed by JSX, but I'm pretty sure they are disallowed by the Geneva Convention or something.

One of the few nice things about ternary expressions in JSX is that their result gets put straight into your output. For example:

src/pages/Detail.js

```
render() {
    return <p>
    {
        chance.first() == 'John'
        ? 'Hello, John!'
        : 'Hello, world!'
    }
    </p>;
}
```

In that example, the true/false blocks of the ternary expression just contains a string, but that's OK because the string gets passed back into the JSX and will be displayed inside the `<p>` element.

So: be prepared to use these ternary expressions sometimes, often written entirely on one line, but please remember they are easily abused!

Pire encore, vous verrez souvent des expressions ternaires doubles, voire triples, où les points d'interrogation et les deux-points s'empilent de plus en plus haut pour former un arbre de **vrai/faux**. Celles-ci sont également autorisées par **JSX**.
L'un des rares avantages des expressions ternaires dans **JSX** est que leur résultat est directement intégré dans votre sortie. Par exemple :

src/pages/Detail.js

```
render() {
        return <p>
                {
                chance.first() == 'John'
                ? 'Bonjour, John !'
                : 'Bonjour, le monde !'
                }
        </p>;
}
```

Dans cet exemple, les blocs **vrai/faux** de l'expression ternaire contiennent simplement une chaîne de caractères, mais c'est OK car la chaîne de caractères est renvoyée dans le JSX et sera affichée à l'intérieur de l'élément **<p>.**

CONCLUSION

Pour conclure, lors de la conception de mon site applicatif, j'ai appris à rebondir assez facilement en prenant en compte les besoins et utilisations client.

En partant sur une base test maquette, rendre cela rapidement utilisable mais statique afin de voir les besoins utilisateurs, les soucis rencontrés, simplifier les choses, je me suis rendu compte que le site serait amené à être en constante évolution.

En apprenant également à utiliser la librairie ReactJS, je me suis mis dans l'optique de reprendre mes notes d'améliorations et de repartir de zéro afin de parfaire à l'expérience utilisateur que j'avais pu avoir lors de la première conception, dites prototype, de l'application et faire en sorte d'améliorer aussitôt les points que j'avais relevé.

Ne pouvant avoir accès à la base de donnée déjà existante, qui était liée à un logiciel qui regroupait des informations clients et appareils afin de créer les devis, je me suis penché sur ce fameux logiciel et ai tiré un maximum d'informations qui pourraient se coupler au projet proposé de base. Ainsi, j'ai pu mettre en place ma base de donnée fictive et me rapprocher au plus de ce que le client attendait en retour.

Lors des tests, il fallait que je trouve le moyen d'utiliser un appareil tactile afin de pouvoir vérifier mes utilisations de menu, par exemple. Ce qui m'a permis également d'apporter des modifications qui font en sorte que le site web applicatif puisse être utilisable sur différents supports.

Non loin de la fin du stage, le patron me demanda de faire en sorte que le site puisse être également sur téléphone, ce qui m'a permis de rebondir une nouvelle fois car j'avais une marge de temps positive devant moi, et le rendre responsive, de cette manière si la concession souhaitait faire évoluer les supports pour les employés alors ils n'auront pas besoin de demander une maintenance.

Egalement, n'ayant pas l'autorisation d'avoir l'accès à la base de données ou le serveur du patron, j'ai commenté tout mon code et ai préparé une documentation pour que l'entreprise de développeur qui est en collaboration avec **Le Havre Nautic** puisse s'y retrouver (*cf Annexe page 34 – Documentation*).

ANNEXE

Glossaire

LEXIQUE

HTML : **Langage de structuration sémantique de site web**, permettant de mettre en page et de structurer son contenu.

CSS : Langage utilisé avec le **HTML permettant au design**, à l'aspect visuel du site web.

JavaScript : **Langage permettant de rendre un site web interactif**, avec ce langage on peut ajouter des fonctionnalités telles que des animations ou des formulaires dynamiques.

Framework : Un **Framework est un ensemble d'outils, de bibliothèques et de conventions de programmation qui facilitent le développement d'applications**. Il fournit une structure prédéfinie pour aider les développeurs à créer des logiciels plus rapidement et plus efficacement.

En utilisant un Framework, **les développeurs peuvent bénéficier d'un ensemble de fonctionnalités et d'abstractions** prêtes à l'emploi, **ce qui leur évite de devoir réinventer la roue à chaque fois**. Le Framework fournit des solutions courantes pour des problèmes spécifiques, tels que la gestion des bases de données, le routage des requêtes (cf ***API REST***), la manipulation des interfaces utilisateur, etc.

ReactJS : **ReactJS** est une **bibliothèque JavaScript utilisée pour construire des interfaces utilisateur interactives et réactives**. Elle permet, par exemple, de diviser l'interface utilisateur en composants réutilisables, ce qui facilite la gestion et la mise à jour de l'application.

API REST : **Une API REST est une interface de programmation qui permet de faire une liaison entre l'interface navigateur et la base de données**. Ainsi, dans un serveur API REST on peut donner des instructions d'affichage, d'ajout, de modification ou suppression de données.

On crée des dites **routes, chemin relatifs que nous réutilisons dans notre langage HTML**, JavaScript ou ReactJS pour pouvoir justement créer cette liaison. (Exemple : *'/api/Exemple-De-Route/'*).

L'utilisation d'une API REST permet une communication flexible et indépendante de la plateforme entre les différents systèmes. Cela évite d'avoir des informations dites brut dans notre code et de devoir le modifier à chaque fois à la main.

NodeJS : **NodeJS** est un environnement d'exécution **JavaScript côté serveur**. Il vous permet d'exécuter du code JavaScript en dehors d'un navigateur web, ce qui ouvre la porte à la création de serveurs web, d'applications en ligne de commande et d'autres applications réseau.

ExpressJS : **ExpressJS** est un **Framework** web minimaliste pour **NodeJS**. Il facilite la création d'applications web en fournissant une abstraction simple et expressive pour gérer les requêtes (ajout, modification, affichage, suppression) et les routes. **ExpressJS** est extrêmement flexible et permet aux développeurs de créer des applications web de toutes tailles et complexités. Il est souvent utilisé pour créer des **API RESTful**, des sites web dynamiques et d'autres applications web côté serveur.

Token d'accès à la connexion : Les **Token** d'accès sont utilisés pour **authentifier** et **autoriser l'accès d'un utilisateur à une connexion**. Lorsqu'un utilisateur se connecte avec succès, un **Token** d'accès unique est **généré** par le **serveur** et **renvoyé au client** (par exemple, au navigateur) sous forme de réponse une fois que la vérification si, par exemple l'email et le mot de passe utilisateurs correspondent. Ce Token est généralement crypté et signé pour garantir son intégrité. Afin de sécuriser les accès, on demande le Token à chaque page dans ce qu'on appelle un en-tête qui permet de donner l'autorisation ou non à l'affichage, ou l'accès à telle ou telle donnée. C'est une manière de rendre sécurisée la connexion à un site.

Cookies pour la connexion : Les cookies sont de petits fichiers de données stockés sur le navigateur de l'utilisateur par le serveur web. Dans le contexte de la connexion, les cookies sont souvent utilisés pour maintenir l'état de connexion d'un utilisateur. Lorsqu'un utilisateur se connecte à un site web, un **cookie** contenant un **Token** peut être créé et envoyé au navigateur. Ce cookie est ensuite renvoyé au serveur avec chaque requête ultérieure, permettant au serveur de reconnaître l'utilisateur connecté.

Input : **Elément utilisé dans la structure du langage HTML**, on peut lui demander différents types comme du texte, un mot de passe etc. Cet élément est utilisé lors de la gestion de base de données par l'interface utilisateur. Par exemple les formulaires de connexion, les formulaires qui peuvent ajouter des données dans la base de données.

Pour la partie menu administrateur ajouts/modifications, **les Input** seront de type texte afin de pouvoir écrire facilement les données que l'on souhaite envoyer.

Select : **Elément utilisé dans la structure du langage HTML**. Il prend la forme d'une liste déroulante, ce qui permet de choisir une information parmi plusieurs. Cela est utilisé dans la gestion d'ajout / modification lorsque les données attendues sont soit pré-écrites, soit dans la base de données.

Boutons radio : **Elément utilisé dans la structure du langage HTML**. Il prend la forme de boutons, lorsque nous avons peu d'informations et qu'on veut en choisir une ou plusieurs, on utilise les boutons afin d'y récupérer la valeur. De même que le select, cet élément est utilisé pour des informations pré-écrites qui seront attendues en base de données.

Fonction : C'est un **bloc de code réutilisable qui effectue une tâche spécifique**. Elle est utilisée pour exécuter une séquence d'instructions lorsque cela est nécessaire. Pensez-y comme une sorte de "recette" ou de "boîte à outils" qui peut être utilisée pour accomplir une action spécifique.

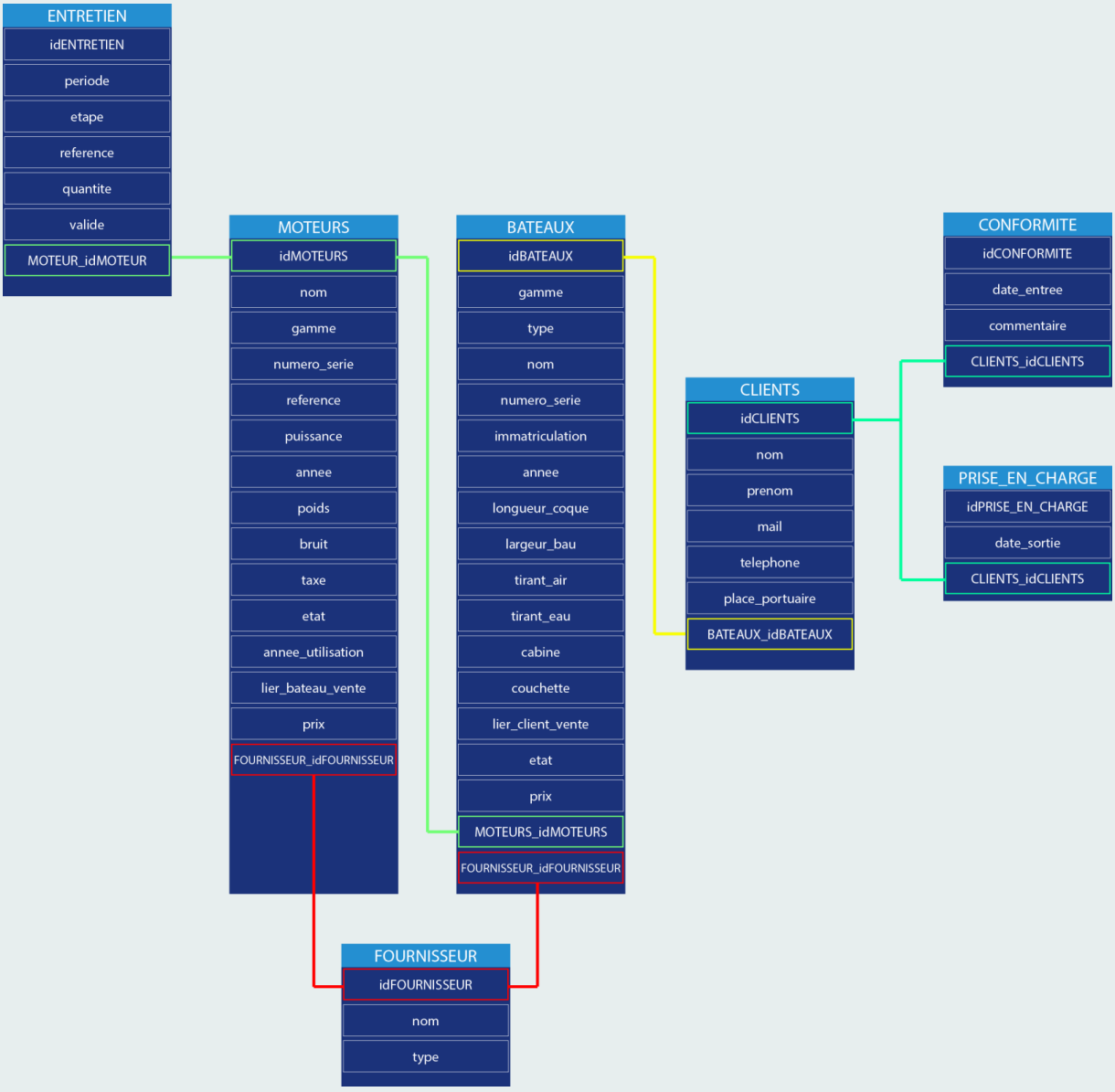
Asynchrone : L'asynchronisme fait référence à la **capacité d'exécuter des tâches indépendantes simultanément**, sans attendre que chaque tâche se termine avant de passer à la suivante. Cela permet d'améliorer l'efficacité et la réactivité des programmes.

En termes simples, lorsque quelque chose est asynchrone, **cela signifie qu'il peut être exécuté en arrière-plan tout en permettant au reste du programme de continuer à fonctionner sans être bloqué ou ralenti par cette tâche**. Cela permet d'effectuer plusieurs actions en parallèle, ce qui est utile lorsque l'on travaille avec des opérations qui prennent du temps, comme les appels réseau ou les accès à des bases de données.

JSX : (*JavaScript Extension*) est une extension de **JavaScript** que l'on utilise avec le **Framework ReactJS**. Elle ressemble beaucoup au **HTML** et permet de créer une structure sémantique pour le rendu visuel vers le navigateur.

INTERACTION BASE DE DONNEES

INTERACTION D'ENTRETIENS MOTEURS



INTERACTION FOURNISSEURS ET APPAREILS EN VENTE

SUPPORT DE CODES

Fonction d'affichage

SERVEUR

Pour l'exemple d'affichage des données, voici celui qui envoie les données concernant les employés.

```
app.get('/api/employes', authenticateToken,
async function affichageEmployes  (req, res) {
  await affichageGlobal(req, res, `SELECT *
                                   FROM employe`)
})
```

Ce code définit une route de méthode **GET** avec comme URL : */api/employes* afin d'afficher les employés stockés dans la base de données.

Ici on utilise ce qu'on appelle un **Middleware**, il donne accès à ces données seulement si une personne est connectée et autorisée. La vérification se fait également côté client avec les lignes *credentials :include.*

Requête qui est envoyée à la base de données, cela permet de sélectionner ce que l'on veut afficher ou sélectionner, comme ici les employés.

ANCIEN CODE

Fonction d'affichage sans qu'elle soit générique

```
mport React, { useEffect, useState } from "react";

import { ValiderEtapes } from
"./fonctionsAPI/ValidationEtapes";

function EtapeMoteurs({ id }) {

  const [donnees, setDonnees] = useState([]);
```

```
const fetchAffichage = async (id) => {

const ipAddress = window.location.hostname;


const url =
`http://${ipAddress}:5005/api/priseEnCharge
/${id}`;

const response = await fetch(url, {
  method:'GET',
  credentials: 'include'
});

const result = await response.json();
setDonnees(result) ;
};

useEffect(() => {
  fetchAffichage();
}, []);
```

```
  const etapesNonValides = donnees.filter((item) =>
item.valide !== 1);

  return (
    <div className="EntretiensMoteurs">

      <h4>Etapes moteur</h4>

      {etapesNonValides.map((item, index) => (
```

Ici, on recopie **entièrement** la **fonction d'affichage API** dans le code initial en intégrant le lien comme dans le code générique.

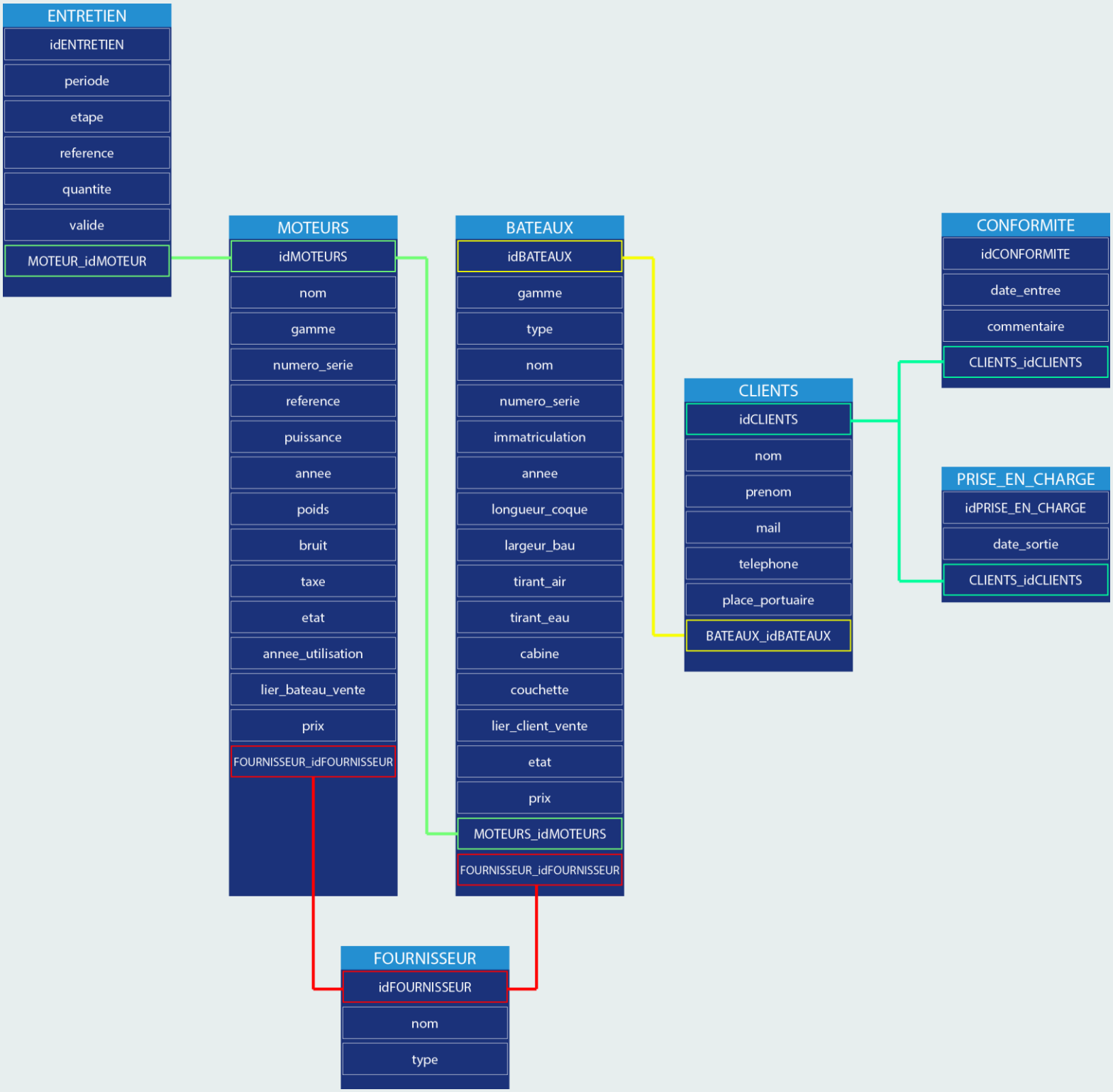
Le **problème** avec cette technique, c'est qu'on devra, **pour toutes les pages** qui attendent un affichage de données, **recopier** ces lignes et **juste changer une ou deux informations** dans **chaque partie de code** où l'on **attend un affichage de données.**

D'où **l'utilité** de rendre **générique** cette fonction en ne **donnant les clefs/paramètres** qu'au partie qui attendent un changement. Cela **économise beaucoup de place** et **permet** également, **en cas de problème dans le code, de ne pas devoir changer l'erreur dans chaque page mais qu'une seule fois.**

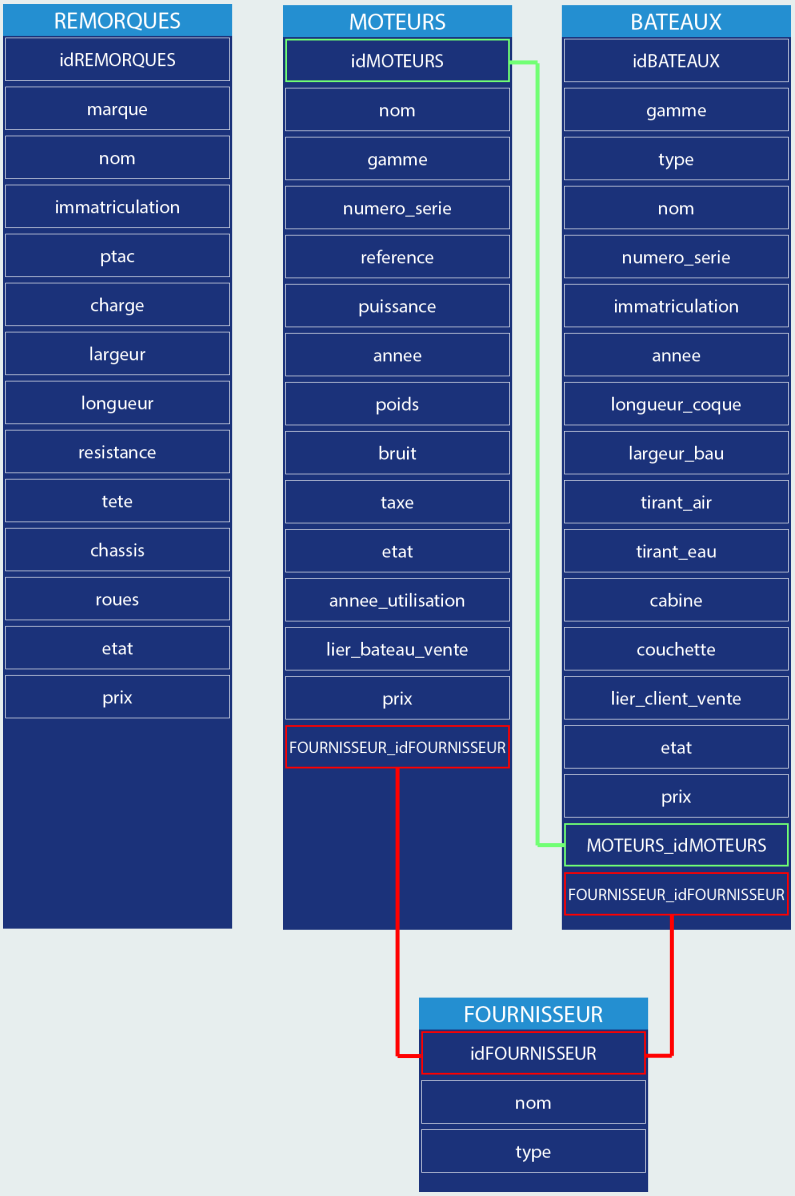
```
        <div key={index}

className={`EntretiensMoteurs ${item.idENTRETIEN}`}>

          <p>{item.etape}</p>
          <p>{item.reference}</p>
          <p>Quantité : {item.quantite}</p>




          <button onClick={() => {
            if (window.confirm('Êtes-vous sûr de
vouloir valider cette étape ?')) {
              ValiderEtapes(item.idENTRETIEN);
            }
          }}

className="ButtonEntretien">Valider</button>
        </div>
      ))}
    </div>
  );
```

DOCUMENTATION
–
LE HAVRE NAUTIC SITE APPLICATIF

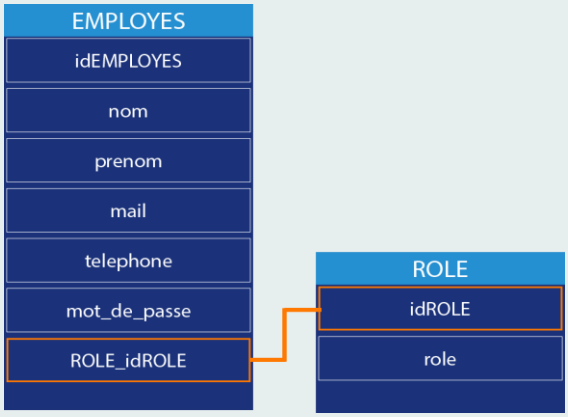
LIAISONS POUR LES AFFICHAGES D'ENTRETIEN, ENVOI DES CONFORMITE VERS LA PRISE EN CHARGE ET FIN DE LA PRISE EN CHARGE



LIAISONS POUR AFFICHER LES NOMS DE FOURNISSEURS



LIAISONS POUR AFFICHER LES ROLES



FRAMEWORKS & LANGAGES

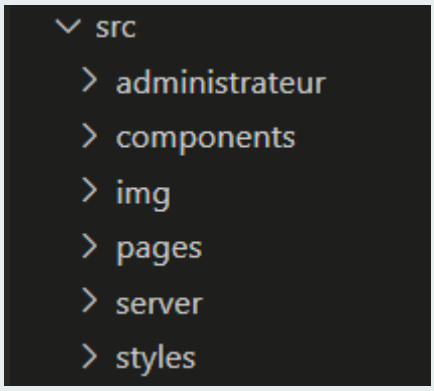
Front-End

- REACTJS
- JAVASCRIPT
- HTML
- CSS

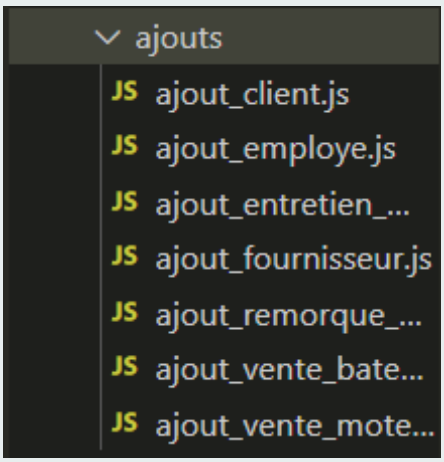
Back-End

- NODEJS (API REST)
- EXPRESSJS
- MYSQL

DOSSIERS DE LA STRUCTURE

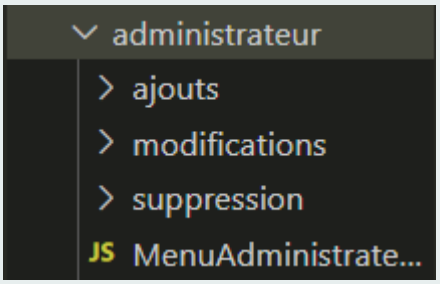


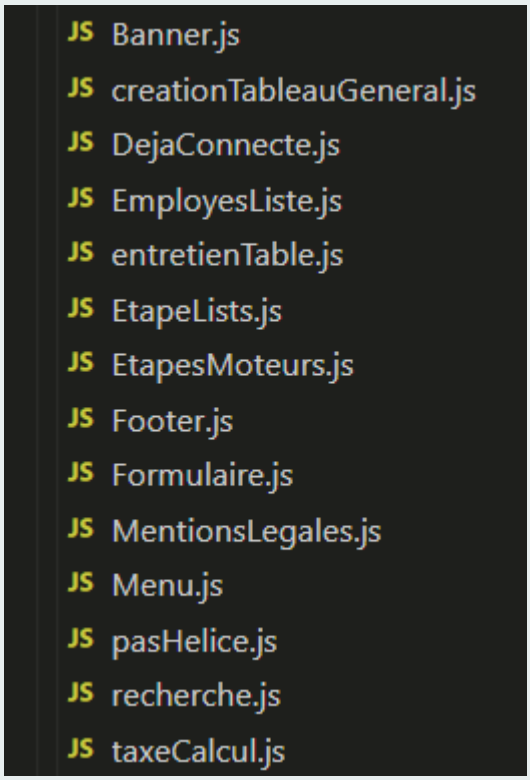
Le dossier **src**, qui avec le **framework ReactJS** permet de stocker les composants, les feuilles **JavaScript**, les différentes pages, les feuilles de styles ainsi que le dossier du **serveur API**.



Ici, le dossier **administrateur** comporte le dossier concernant la partie d'ajout, de modification et de suppression vers la base de données.

Il y a également le composant qui permet de générer ce **menu administrateur**. Chaque dossier comporte le même nombre de feuilles **JS**.



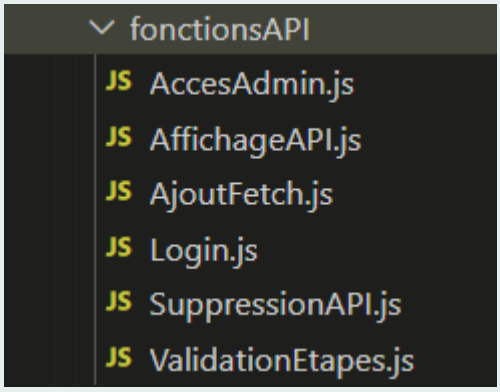


Dans le dossier **components** concerne les composants. On y retrouve le dossier **fonctionsAPI** qui retrouvera les fonctions génériques réutilisables dans les différentes feuilles afin d'éviter la réécriture. **JSONInformationsElements** concerne les informations de header tableau ou d'informations de formulaire afin d'éviter également la réécriture.

Ces composants sont réutilisés à travers les pages qui sont dans un autre dossier, ce qui facilite l'organisation. Voici leurs différentes utilités :

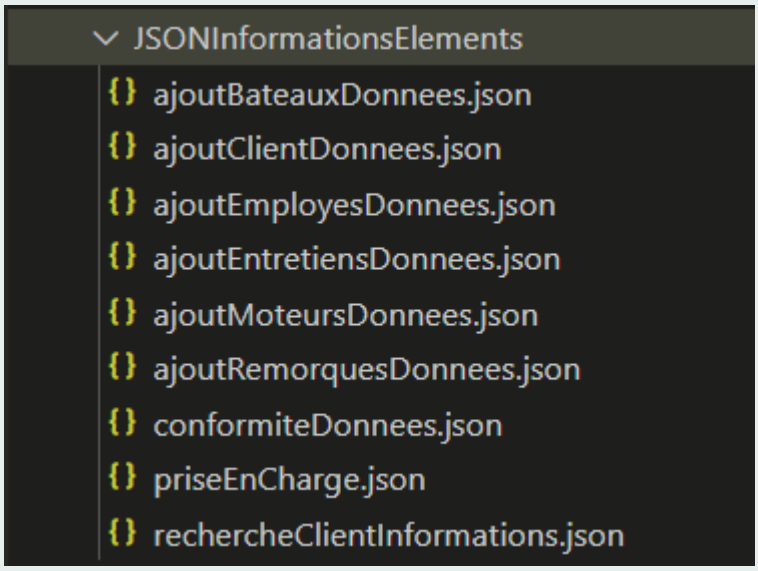
- **Banner.js :** Permet d'intégrer la bannière en entier. Il y a un lien avec la connexion ou déconnexion en fonction de si l'utilisateur est connecté ou non.
- **CreationTableauGeneral.js :** Composant générique qui permet de créer un tableau en réunissant des paramètres et permettant de donner un certain ordre dans le tableau, intégrer des données de base de données et ainsi le rendre générique plutôt que de devoir copier coller à chaque fois.
- **DejaConnecte.js** : Permet d'afficher un message à l'utilisateur sur le formulaire de connexion alors qu'il est déjà connecté.
- **EmployesListe**.js : Permet de lister les employés et le moyen de les contacter, sera réutilisé dans le footer pour les employés qui seront connectés.
- **EntretienTable.js :** Permet de mettre en place les tableaux regroupant tous les entretiens des moteurs qui seront affichés grâce à un menu cliquable généré par la base de données. Ainsi, nous pouvons le styliser en fonction des périodes et récupérer tous les entretiens importants.
- **EtapeLists.js :** Permet de lister toutes les étapes, faire afficher, lorsque les étapes sont toutes finies, un bouton qui permet de dire que les réparations sont terminées, un bouton qui permet d'afficher ou cacher les étapes également.

- **EtapesMoteurs.js** : Permet de lister les étapes liées à un moteur spécifique lui-même lié à un client.
- **Footer**.js : Composant qui permet de créer le footer.
- **Formulaire.js :** Composant générique de création de formulaire afin d'éviter les copier coller.
- **MentionsLegales**.js : Composant regroupant les mentions légales et autres informations nécessaires.
- **Menu.js :** Composant qui regroupe le menu général composé de :
    - **Client**
        - Recherche de client
        - Vérification client vers atelier
    - **Prise en charge**
    - **Vente**
        - Bateaux
        - Moteurs
        - Remorques
    - **Moteurs**
        - Menu généré à partir des fournisseurs moteurs dans la base de données
- **PasHelice.js :** Feuille qui regroupe un algorithme qui permet à l'employé de calculer le pas de l'hélice et ainsi retourner des inputs lui permettant d'intégrer les valeurs et d'y voir le résultat, ainsi qu'un bouton pour cacher ou afficher la « calculatrice ».
- **Recherche.js** : Algorithme et composant permettant à l'utilisation d'une barre de recherche qui sera réutilisée dans tout le site et dans tous les sens.
- **TaxeCalcul.js :** Algorithme qui servira à calculer la taxe d'un moteur afin d'avoir le résultat lors de l'ajout d'un moteur dans la base de données.



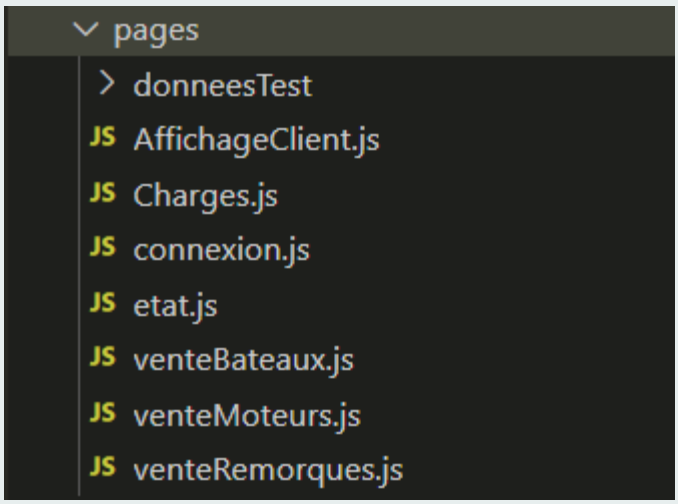
- **AccesAdmin.js :** Permet de récupérer l'accès de l'utilisateur s'il a le rôle d'administrateur afin d'afficher le menu administrateur montré plus haut.

- **AffichageAPI.js** : Fonction générique qui permet d'éviter la réécriture pour afficher les données, fonctionne avec l'id ou sans.
- **AjoutFetch.js :** Fonction générique d'ajout ou de modification de données.
- **Login.js** : Fonction de connexion.
- **SuppressionAPI.js** : Fonction générique de suppression.
- **ValidationEtapes.js :** Fonction de validation d'étapes pour la prise en charge.



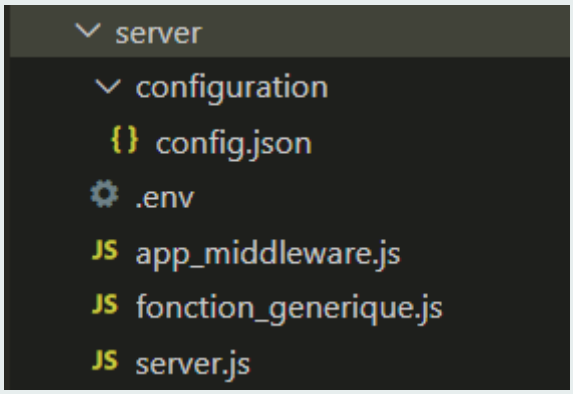
Feuilles JSON qui permet de regrouper des données afin d'éviter de les réécrire. On peut récupérer ainsi un label, un id ou autres. Exemple avec les informations des employés

```
[  {    "label": "Nom",
        "id": "nom_employe",
        "name": "nom_employe",
        "type": "text"  },

    {   "label": "Prénom",
        "id": "prenom_employe",
        "name": "prenom_employe",
        "type": "text"  },

    {   "label": "Mail",
        "id": "mail_employe",
        "name": "mail_employe",
        "type": "text"  },
```



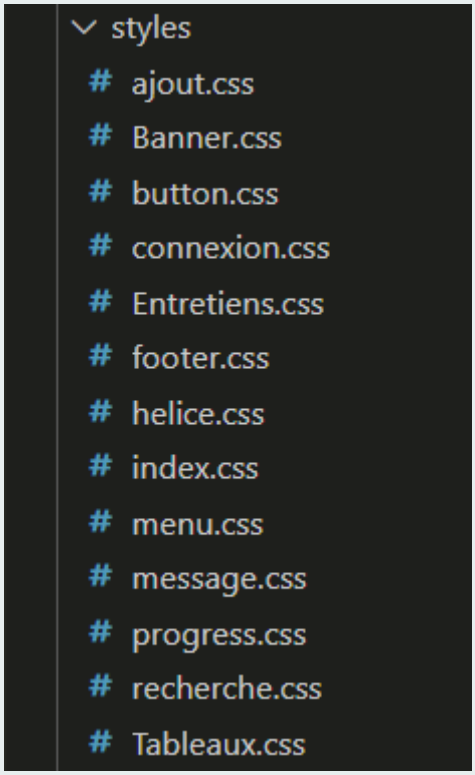
Dans le dossier pages nous pouvons retrouver les composants qui nous permettront à l'affichage de nos pages du site. On importe à l'intérieur les composants du dossier components en fonction de leurs utilités, et nous faisons l'affichage et la structure dans ces feuilles js.

- **AffichageClient.js :** Permet d'afficher le tableau des données client.
- **Charges.js :** Permet d'afficher le tableau de prise en charge
- **Connexion.js :** Permet d'afficher le formulaire de connexion.
- **Etat.js :** Permet d'afficher le tableau de conformité afin d'envoyer l'appareil vers la prise en charge, on y détermine quelle période d'entretien du moteur on doit faire et on y ajoute les commentaires supplémentaires au cas où.
- **VenteBateaux.js, VenteMoteurs.js, VenteRemorques.js :** Les pages qui permettent à la vente d'appareils sous forme de tableaux en regroupant leurs informations.



- **Config.json :** Feuille JSON qui regroupe les informations nécessaires à la liaison du serveur API et la base de données.
- **.env :** Permet de générer le token.

- **App_middleware.js :** Feuille qui regroupe les fonctions middleware de manière exportée au serveur.
- **Fonction_generique.js** : Fonctions générique nécessaires au serveur pour les ajouts, modifications, affichages et suppressions.
- **Server.js :** Serveur API REST qui regroupe toutes les requêtes SQL.



Voici les différents feuilles de CSS, les titres sont assez indicateurs sur le composant ou la page qu'elles touchent.

Ces pages sont toutes commentées afin de mieux comprendre les lignes qui sont écrites.

Respecter l'écriture du nom / prenom dans la table des employés en attribuant le rôle administrateur.

nom	prenom
Di Gregorio	François

idROLE	role
1	Administrateur
2	Utilisateur

Respecter l'id et les écritures de ces lignes :

Table bateaux respecter la gamme, l'idBATEAUX et le lier_client_vente :

idBATEAUX	gamme	type	nom	numero_serie	immatriculation	annee	longueur_coque	largeur_bau	tirant_air	tirant_eau	cabine	couchette	lier_client_vente
19	aucun bateau	NULL	NULL	none	none	NULL	NULL	NULL	NULL	NULL	NULL	NULL	0

Table moteurs respecter le nom, l'idMOTEURS ainsi que le lier_bateau_vente :

idMOTEURS	nom	gamme	numero_serie	reference	puissance	annee	poids	bruit	taxe	etat	annee_utilisation	lier_bateau_vente
16	aucun moteur	NULL	NULL	NULL	NULL	0	NULL	NULL	NULL	NULL	0	1

Table fournisseur :

idFOURNISSEUR	nom	type
13	none	moteur

```
{
    "connectionLimit" : 10,
    "host": "localhost",
    "user": "root",
    "password": "",
    "database": "lhnautic"
}
```

Mettre à jour ces configurations qui se trouvent dans le dossier src – server – configuration, en fonction des informations de la base de données réelle.