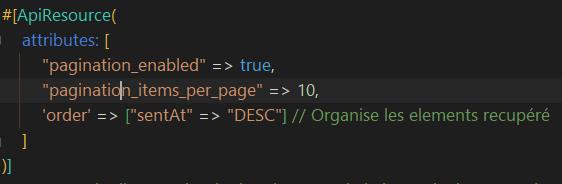
# Api-Platform

On ajoute Api Ressource () sur une entité pour spécifier que celle-ci peut être visible dans l’API

On peut lui ajouter des attributs pour spécifier les options a apporter comme la pagination

Ex :



Il existe une classe spécifique au filtre nommé ApiFilter

Sur celle-ci on a deux types de recherche, par ordre , OrderFilter et par search, SearchFilter

Pour chacune d’entre elle on peut leur préciser des propriétés qui contraindront l’utilisateur a viser celle-ci lors d’une recherche

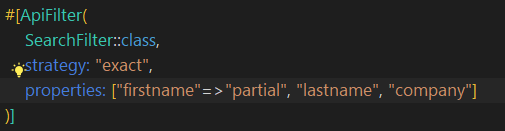
Ex :



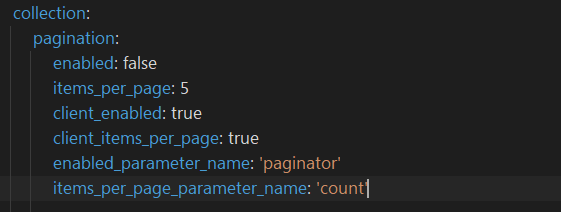
Pour chaque recherche, on peut déterminé une stratégie de recherche,

* Partial stratégie utilise LIKE %texte% pour rechercher les champs qui contiennent texte.
* Start stratégie utilise LIKE texte% pour rechercher les champs commençant par texte.
* End stratégie utilise LIKE %texte pour rechercher les champs qui se terminent par texte.
* Word\_start stratégie utilise LIKE texte% OR LIKE % texte% pour rechercher des champs contenant des mots commençant par texte.

On peut déterminer une stratégie globale, pour toutes les propriétés données, ou bien attribuer une stratégie pour chaque propriété.



Les valeurs par default pour toute les données sont situer dans api\_platform.yaml



Certaine option permette de modifier le nom du paramètre comme enabled\_parameter\_name pour modifier la valeur booléenne de la pagination. Par default pagination=true suite a la modification paginator=true.

Les valeurs commençant par client permettent l’autorisation des paramètres dans la barre d’adresse.

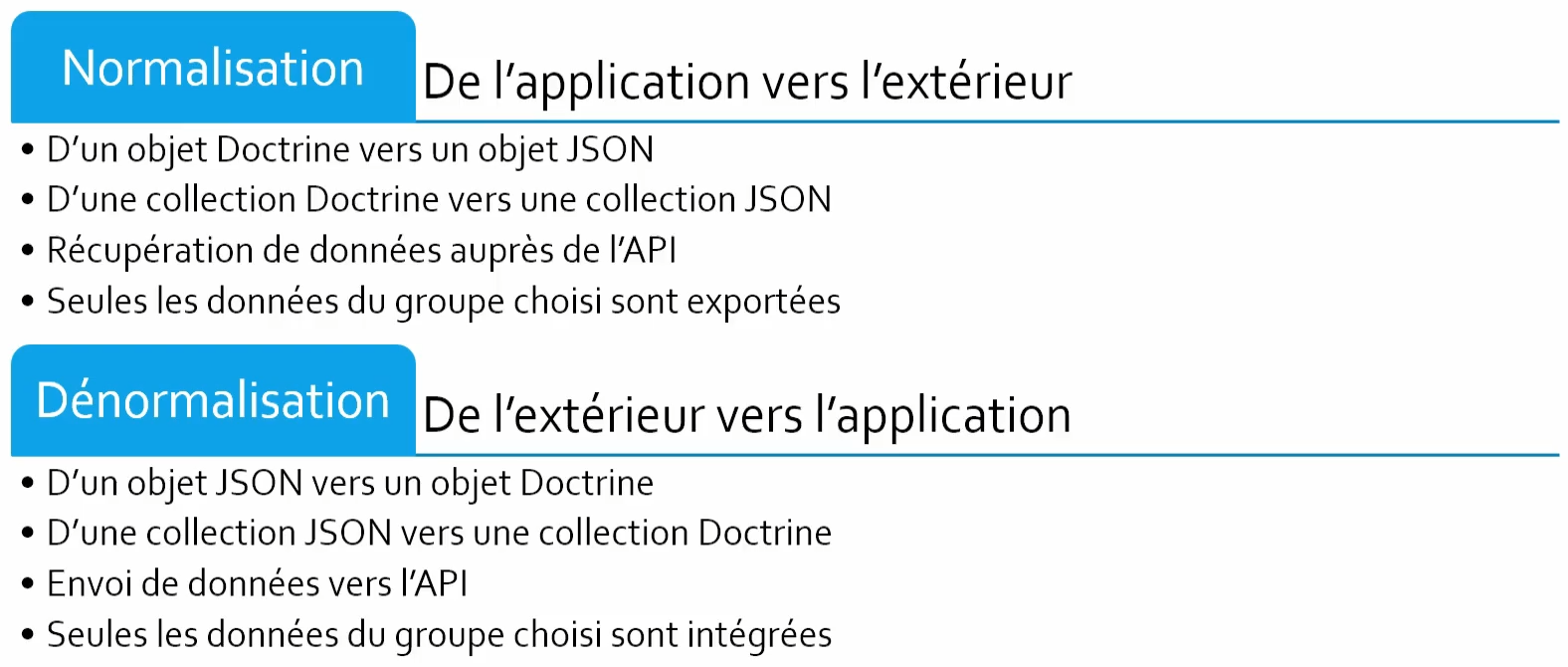
## Transformation json -> Object

Normalisation = Object -> Tableau standard

Sérialisation = Tableau standard -> Tableau JSON

Désérialisation = Tableau JSON -> tableau standard

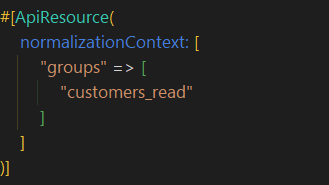
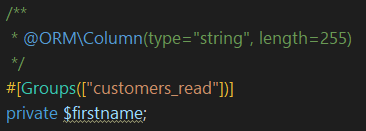
Dénormalisation = Tableau standard -> Object



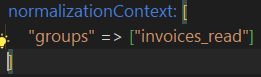
Avec api platform, on peut insérer des champs dans des groupes afin de déterminer quel champ on verra apparaitre dans l’API

On précisera que telle groupe sera visible dans l’api en le présentant comme sérialisable, de ce fait les autres champs de l’entité ne le seront pas

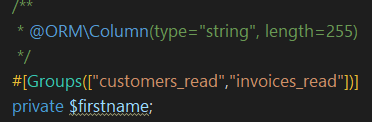
Ex :

Ici j’autorise la normalisation (la transformation Object -> Tableau standard uniquement au champ situer dans le groupe customers\_read

Il est conseillé de placer tous les champs d’une entité dans des groupes, cela permet une meilleure flexibilité

Ensuite si dans les invoices par exemple je souhaite voir le détail du Customer associer, alors je peux attribuer un groupe au invoice :

Et ouvrir la lecture des champs de l’Object invoice, au Customer

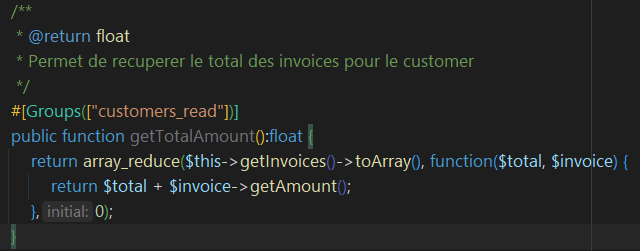


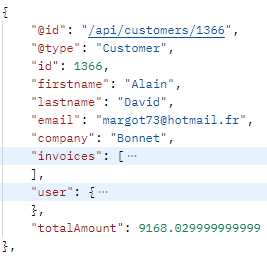
Ainsi ceci donnera 

Dans les invoices on recuperer toute champ marquer comme lisible avec le groupe « invoices\_read » dans l’entité customer

Probleme de boucle infini, Admeton que nous soyon dans l’entité invoice qui est lié a l’entité customer, alors si j’appelle le detail des invoices dans les customer, une boucle infini renvoi une erreur puisque une invoice renvoi ses information plus le customer, qui renvoi les invoice qui renvoi les customer qui… bref boucle. Pour evité cela il suffit de ne pas mettre le champ invoices de l’entité customer dans le groupe invoices\_read.

Il est possible de créer des function dans les entités pour ensuite envoyer le resultat de celle-ci dans l’api imaginons :



Dans l’entité Customer, je recuperer grace a une fonction la somme total des facture du client, je rajoute cette fonction dans le groupe customers\_read et il s’affiche dans mes donné api

Toujours en créant une function, je peux par exemple créer un decalage me permettant un racourcit pour recuperer le user lié a l’invoice

Actuelement si je recherche mes facture je tombe sur

Facture

Customer

User

Il me suffit alors de créer une fonction dans l’entité invoice qui va chercher $this->getCustomer()->getUser.

Je met cette function dans le groupe « invoices\_read », je supprimer le invoices\_read du champ user dans l’entité customer, et je me retrouver avec

Facture

Customer

User

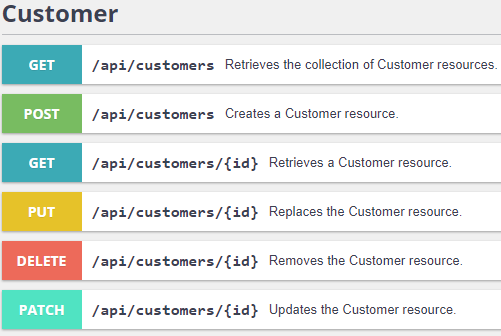
Lors d’une recherche des invoices

## Les operations

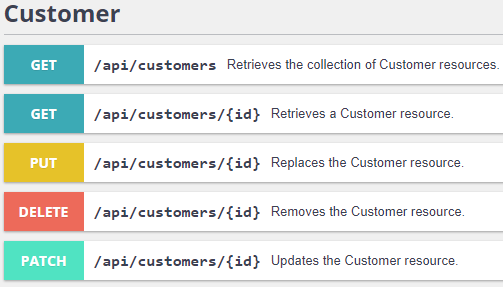
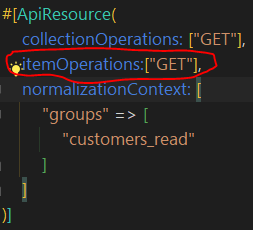
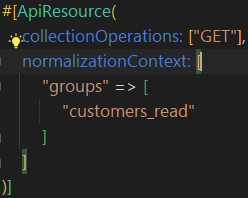
### Opération de collection

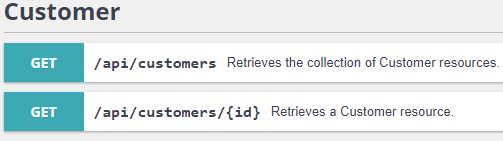
Get post sans identifiant

### Opération d’items

 Get{id} delete{id} put{id}

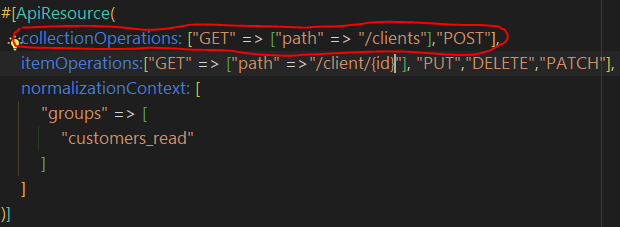
On peut activer ou désactiver tel ou tel opération depuis, exemple si je ne souhaite pas que l’on puisse créer un élément je rajoute collectionOpération [‘ GET ‘ ] dans APIRessource,

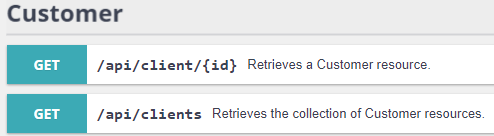
 j’ai ainsi précisé que je ne voulais que le get dans les collectionOperation et le post disparait



Ici je demande qu’il n’y ai que le GET des itemOperation qui soit accessible

Pour chaque opération, je peux attribuer une chemin diffèrent de de celui de base, exemple pour l’instant mon chemin(path) pour récupérer les clients : <http://127.0.0.1:8000/api/customers>

En ajoutant un path dans ‘collectionOperation ‘

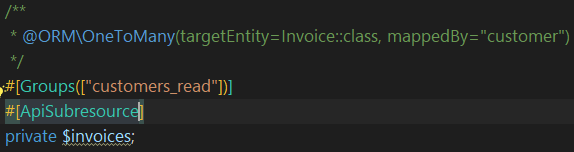


Le nouveau chemin sera <http://127.0.0.1:8000/api/clients>

Cela permet en plus de masquer le nom réel des entités

## Les subRessources

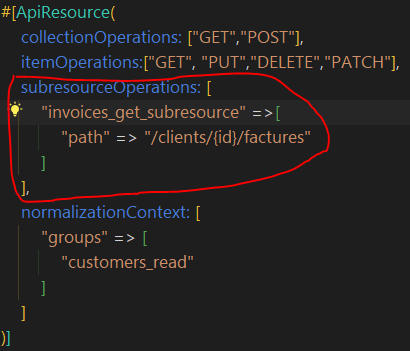
Les subressource crée des liens permettant d’obtenir les ressources d’un seul Customer par exemple :



En ajoutant #[ApiSubresource] au dessu du champ invoices de l’entité Customer, un lien [http://127.0.0.1:8000/api/customers/{id}/invoices](http://127.0.0.1:8000/api/customers/%7bid%7d/invoices) est créer, on remplace {id} par l’id d’un Customer, et on réceptionne toutes les factures de ce client.

#### Configuration :

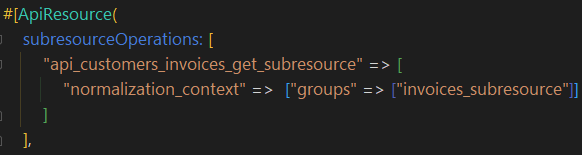
Il est possible de modifier l’adresse de la subRessources, en le précisant dans le apiResource au-dessus de l’entité

Le champ que l’on va chercher est invoices, donc

« invoices\_get\_subresource » et on lui attribue les paramètres souhaités, comme dans l’exemple modification du path



On peut configurer les champ renvoyer lorsque l’on appelle une subressource, actuellement lorsque l’on réclame toute les invoices d’un seul client, on ce retrouver avec toute les information lié a l’invoice. Si l’on souhaite sélectionner les champ désiré la marche a suivre est la suivante :

* On va dans l’entité visé par la subressource : en l’occurrence « invoice »
* On déclare que l’on va paramétrer les résultats d’une subressource 

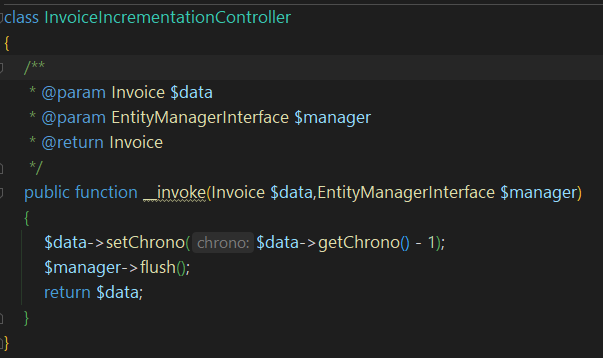
En l’occurrence la demande de subressource vient de customer donc

Api\_l’entité-parente\_la-clé-donné-dans-l’entité-parente

## Creation d’une operation (get post put…)

Pour créer une opération, On commence par créer un controller, type InvoiceIncrementController

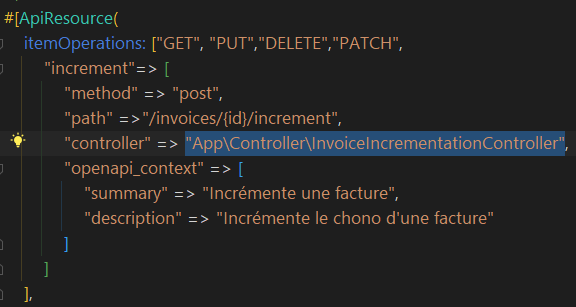
Ensuite on créer un public function invoke avec en paramètre l’Object concerné ici : Invoice, dans une variable nommée $data.

Cette fonction aura pour effet de modifier la valeur du champ chrono de l’invoice visé.

Pour l’instant, aucun effet ne sera visible. Il reste a créer la route dans l’entité

On va créer un nouvel itemOperation, donc on ouvre itemOperation dans APIRessource de l’entité concerné, On détermine les différents itemOperation accessible (put get delete patch) et on ajout un nouvel item que l’on nomme comme on le souhaite. Ici « incrément ». On lui attribue ensuite la configuration que l’on souhaite, ici

* La method est « post »,
* Le chemin sera « /invoices/{id}/incrément »,
* On précise le controller créer précédemment "App\Controller\InvoiceIncrementationController"
* Et pour finir on modifie le nom et la description de notre nouvel itemOperation avec « openapi\_context » qui prendra en paramètre le « summary » qui sera le titre de l’item et la « description » qui sera la description de l’item

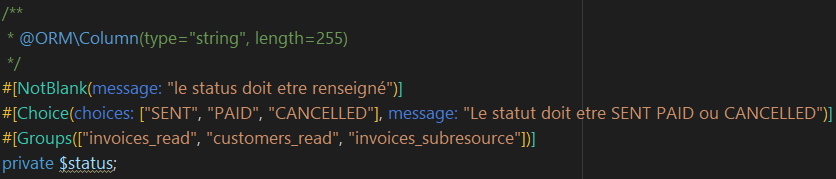


## Se prémunir contre les échecs d’insertion

Ajout d’une spécification dans les services pour formater une date correcte l’ors de la sérialisation services.yaml

services:  
 Symfony\Component\Serializer\Normalizer\DateTimeNormalizer:  
 arguments:  
 $defaultContext:  
 datetime\_format: 'Y-m-d'

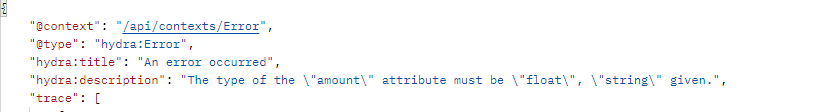
Pour éviter les erreur incompréhensible l’ors de la création d’un élément et pour éviter de dévoiler ouvertement l’erreur entière, on utilise les constraint violation de symfony dans l’entité au-dessus des champ ex :



NotBlank = ne peut être null

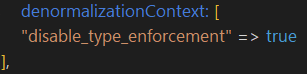
Choice = n’autorise que les valeurs précisées

Un problème de vérification provient lorsque l’ORM doctrine passe avant notre constraint de validation. Si l’on marque « Bonjour » dans l’amount par exemple, l’ors de la dénormalization, le système va capter que l’ORM demande un float et renvoi une erreur avant même l’action de la constraint



Pour résoudre ce problème il faut

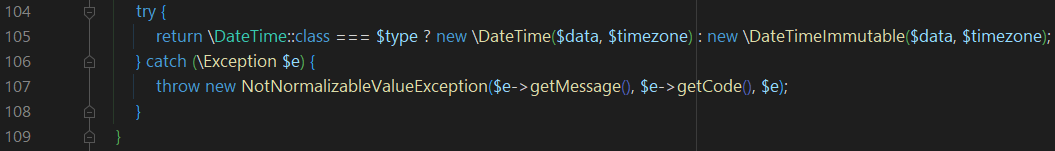
* Supprimer le typage du setAmount par exemple
* Désactivé la vérification au niveau ORM pendant la dénormalization

« disable\_type\_enforcement » supprimer la vérification du a doctrine et en sautant cette étape, notre constraint de validation est bien prise en compte

### Problème de date :

L’erreur du format de date ne peut pas être customisé de base, ceci vient d’une erreur dans le fichier situé dans « vendor/symfony/serializer/Normalizer/DateTimeNormalizer.php »

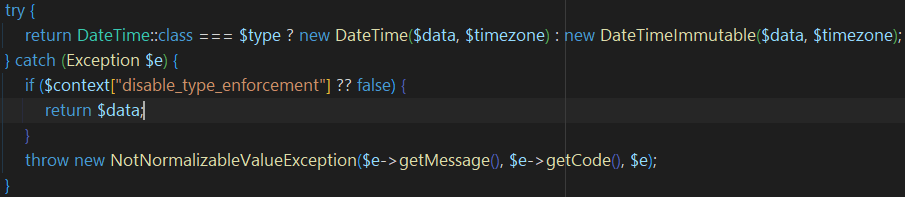
La vérification du format de date ce fait tout en bas a la ligne 104



L’objectif est d’écraser cette vérification pour la corrigée.

Pour ce faire on crée un dossier Sérialiser dans /src, a l’intérieur on crée un fichier nommé PatchedDateTimeSerialiser.php qui contiendra a l’identique du fichier DateTimeNormalizer.php

On modifie la partie buggé en lui précisant que si « disable\_type\_enforcement existe, alors on renvoi la date tel quelle est afin que celle-ci soit ensuite traité par la contrainte de validation.



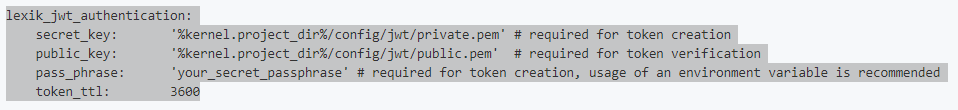
## Sécurisation authentication

Utilisation de JWT

“composer require LexikJWTAuthenticationBundle.”

Installer openssl sur l’ordinateur

Création des clef public et clef privé :



Mise en place du firewall dans security.yaml

login:  
 pattern: ^/api/login  
 stateless: true  
 anonymous: true  
 json\_login:  
 check\_path: /api/login\_check  
 success\_handler: lexik\_jwt\_authentication.handler.authentication\_success  
 failure\_handler: lexik\_jwt\_authentication.handler.authentication\_failure

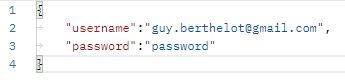
Ce firewall redirige l’utilisateur vers son adresse ^/api/login

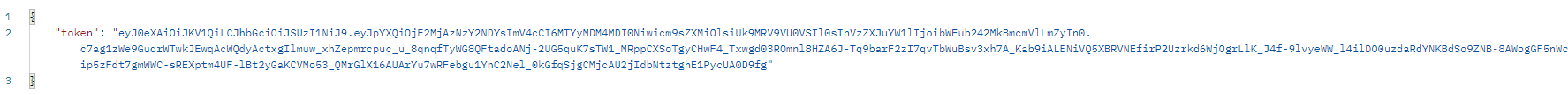
Il est accessible pour tout les utilisateurs, anonyme ou pas

Configuration de la route dans route.yaml

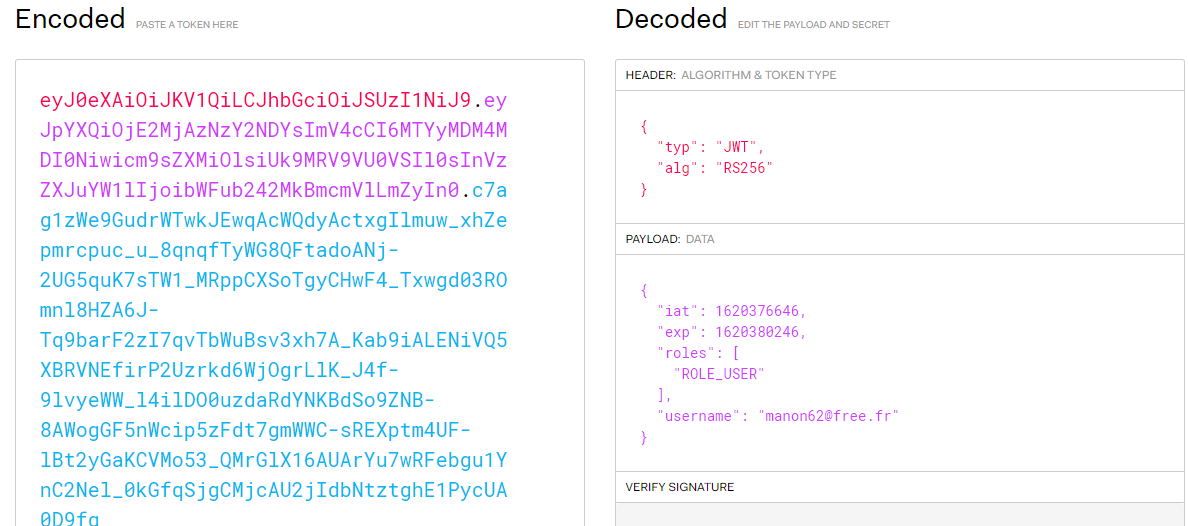
api\_login\_check:  
 path: /api/login\_check

la page <http://localhost:8000/api/login_check> n’est disponible qu’a partir du moment ou du json est envoyer dans la page, soit depuis Postman par exemple en ajoutant dans body



Une fois envoyé si tout ce passe bien, un token est envoyé

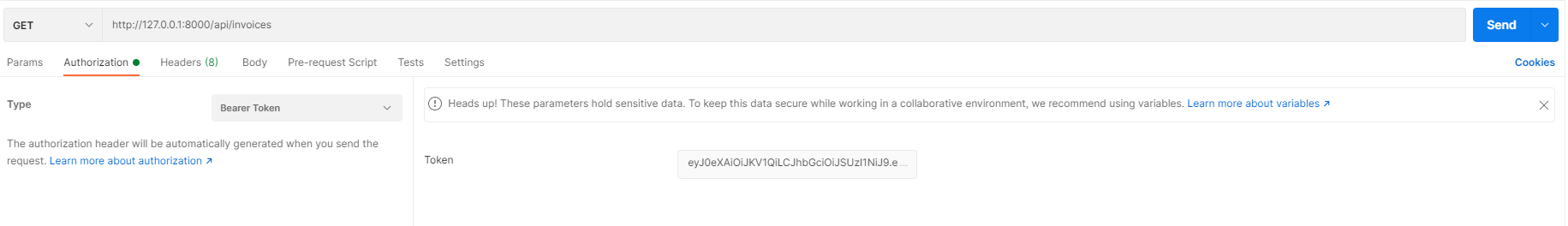
Si on entre ce token jwt sur le site <https://jwt.io/> les infos sont retrouvé



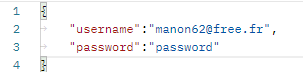
api:  
 pattern: ^/api  
 stateless: true  
 guard:  
 authenticators:  
 - lexik\_jwt\_authentication.jwt\_token\_authenticator

Cette partie-là tjr dans le firewall de security.yaml permet de dire, tout ce qui ce trouve dans /api, sera garder par le jwt auth

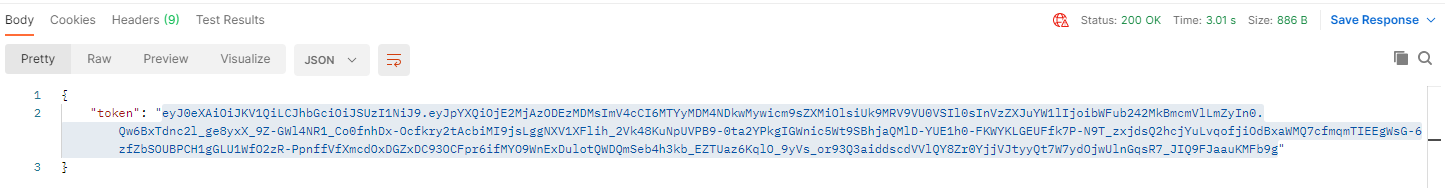
Maintenant afin de récupérer les données en faisant un appel dans Postman, il faudra créer un token et le mettre dans authorization de Postman

donc la procédure a partir de la sera

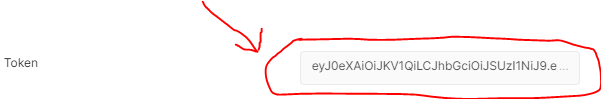
* <http://127.0.0.1:8000/api/login_check> en précisant dans le body, un Username et un Password



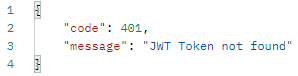
* Ensuite je récupère le token envoyé



* Si je souhaite faite une requête pour récupérer les invoices par exemple, alors il faudra mettre cette clé dans

Sinon je me retrouve avec une erreur :



En clair, je dois m’identifier avec un token pour prouver que je suis bien dans la liste des users récupérer les données.

### Créer un utilisateur malgré la demande d’authentification :

Je crée un nouveau firewall dans security.yaml permettant l’accès a une adresse malgré le fait de ne pas avoir de token puisque je créer un utilisateur, il est normal que celui-ci n’en ai pas.

registration:  
 pattern: ^/api/users  
 stateless: true  
 anonymous: true

methods: [POST]

*Le nom du firewall n’est pas fix n’importe quel nom ferai l’affaire*

On précise ensuite la méthode autorisé « POST » pour éviter les recherches de users non authentifié maintenant seule les requête créant un user sont accepté si l’on n’est pas authentifié avec un token

Ouverture de la page d’accueil a tout le monde même Anonymous, pour utiliser les access\_control : symfony :

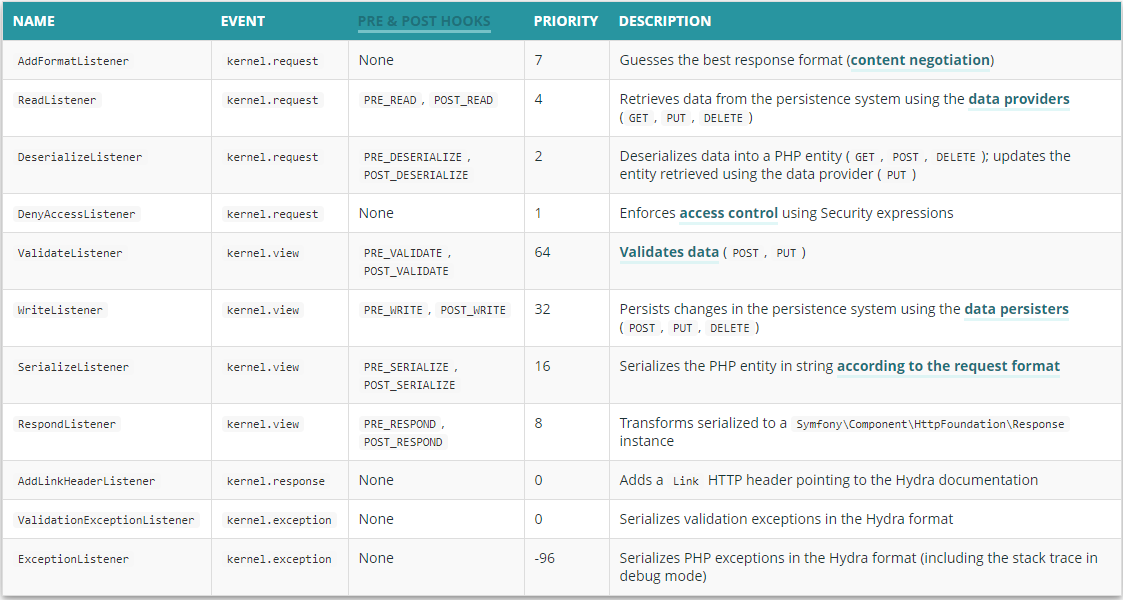
access\_control:

- { path: ^/api/login, roles: IS\_AUTHENTICATED\_ANONYMOUSLY }  
 - { path: ^/api/customers, roles: IS\_AUTHENTICATED\_FULLY }  
 - { path: ^/api/invoices, roles: IS\_AUTHENTICATED\_FULLY }  
 - { path: ^/api/users, roles: IS\_AUTHENTICATED\_FULLY, methods: [GET,PUT,DELETE] }

Ici on dit que seul les gens authentifier pourront accéder aux customers, invoices et users sauf pour la création de user (POST) laissé ouvert pour l’inscription et le login qui sera ouvert pour les personne non connecté.

En clair, tous les firewalls sont en Anonymous true (ouvert pour tout le monde) puisque maintenant c’est l’access\_control qui gère les laisser-passé.

### Créer des évents en se servant du kernel

Timelaps des évents kernel

Il est possible d’agir sur les éléments envoyer ou reçu a certain moment grâce aux évents.

L’exemple pris dans la leçon est l’encryptage du Password juste avant la transformation du tableau standard vers l’Object (dénormalisation), on utilisera donc une class nommée : « PasswordEncoderSubscriber » qui implémentera « EventSubscriberInterface ».

Dans cette implémentation on a une fonction imposer, nommé « getSubscriberEvent » qui permet de récupérer l’évènement a un moment donné.

Ici : on utilisera le kernelEvent :VIEW avec le PRE\_WRITE , c’est-à-dire juste avant l’écriture de l’entité.

Une fois l’évènement récupérer, on utilisera le résultat de cette fonction pour agir sur ce moment sélectionné dans une autre fonction que l’on nommera pour l’exemple : « encodePassword » celle-ci prendra en paramètre une variable $évent de type de ViewEvent. EventView permet de récupérer la donnée en cours de traitement au moment donné dans la function précédente.

On utilise donc « $event->getControllerResult () » qui nous renverra dans l’exemple précis, un User puisque l’on est en train de créer une entité User.

A partir de là on détermine de quelle method l’event provient, ici POST puisque l’on créer un User.

$method = $event->getRequest()->getMethod(); // POST, GET, PUT, ...

Et on fait une vérification pour que notre event de s’active pas a chaque fois qu’un appel est fait

if ($user instanceof User && $method === "POST") {  
 $hash = $this->encoder->encodePassword ($user, $user->getPassword ());  
 $user->setPassword($hash);  
}

Si on traite bien une entité User, et que la method concerne bien une création de User, alors on utilise le « UserPasswordEncoderInterface » pour encoder le Password du User en cours de création.

En clair :

* On créer une classe qui implémente EventSubscriberInterface
* On détermine le moment où l’on souhaite agir en s’appuyant sur le tableau cité précédemment.
* On crée la function de modification en dessous qui prendra en paramètre « ViewEvent »
* On vérifie bien les correspondances entre le moment ou l’on souhaite agir, l’entité, et la méthode utilisé
* Et si tout ça correspond, alors on fait notre modification

Note : pour aller chercher le user connecter, utiliser Security de symfony

Il a fallu faire la même chose pour lié le user connecter au Customer en construction lors du POST Customer. Ala différence près que l’action devais se produire avant la validation du « formulaire » donc :

KernelEvents::VIEW => ["setUserCustomer", EventPriorities::PRE\_VALIDATE]

Ensuite, encore un event mais cette fois ci pour automatiser le chrono d’une invoices pour ceux-ci se suivent sans avoir à les taper manuellement

## Ajout de donnée dans le token JWT de base

Pour injecter des données directement dans le token jwt on passera par une classe que l’on annoncera dans le service.yaml. Ex :

App\Events\JwtCreatedSubscriber:  
 tags:  
 - { name: kernel.event\_listener,  
 event: lexik\_jwt\_authentication.on\_jwt\_created,  
 method: updateJwtData  
 }

Dans cette class on créera une function updateJwtData qui prendra en paramètre « JWTCreatedEvent » qui contraindra les données au moment d’une création de token JWT.

class JwtCreatedSubscriber  
{  
 public function updateJwtData (JWTCreatedEvent $event)  
 {  
 dd($event);  
 }  
}

Ensuite lorsque j’utiliserai $event je pourrais sélectionner les donné et ajouter des informations comme ceci :

/\*\* @var User $user \*/  
$user = $event->getUser ();  
$data = $event->getData();  
  
$data["firstname"] = $user->getFirstname();  
$data["lastname"] = $user->getLastname();  
  
$event->setData($data);

* Je récupère le user qui est compris dans l’event,
* Je récupère les données de l’event sous forme de tableau,
* Je rajoute les données du user dans le tableau data.
* Et pour finir je set le tableau data avec les nouveaux donnés issu du User.

Une fois cette manipulation, on récupère le code JWT on le test sur <https://jwt.io/> et on constate que le token a bien été popullé avec les données de l’utilisateur

L’objectif de cette démarche est d’utilisé infiné un plugin javascript qui décodera ce token pour récupérer les données qu’il contient

## Requête Doctrine à chaque GET envoyé

A chaque requête sur les Invoices par exemple, on récupère toutes les invoices de tout le monde, et ceci est un problème, une requête SQL est requise, pour ne récupérer que les invoices concerné par l’utilisateur connecter.

Pour ce faire

* On créer une classe qui implémentera de QueryCollectionExtensionInterface et QueryItemExtensionInterface.
* Deux fonctions publiques sont obligatoires applyToCollection et applyToItem comportant chacune leur lot de paramètre (Une function concernera les items (methods comprenant un id) et collection ne comprenant que les liste d’item.
* public function applyToCollection(QueryBuilder $queryBuilder, QueryNameGeneratorInterface $queryNameGenerator, string $resourceClass, string $operationName = null)  
  {  
   // TODO: Implement applyToCollection() method.  
  }  
    
  public function applyToItem(QueryBuilder $queryBuilder, QueryNameGeneratorInterface $queryNameGenerator, string $resourceClass, array $identifiers, string $operationName = null, array $context = [])  
  {  
   // TODO: Implement applyToItem() method.  
  }

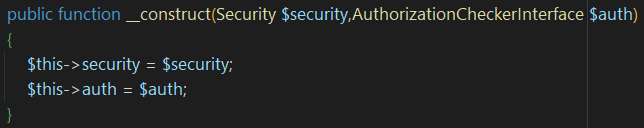
L’objectif ici est de récupérer les items (invoice ou Customer de la personne connecté). Dans tous les cas de figure , item ou collection

Donc je dois :\*

* Je créer une function supplémentaire, « addWhere » par exemple
* Je Récupéré le User connecter avec Security.



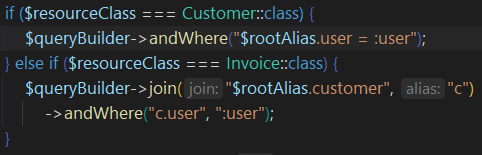
* je récupérer AuthorizationCheckerInterface pour la vérification des rôles



* Je Récupéré l’alias de l’élément concerné, « c » pour Customer ou « i » pour invoice.

On récupère la liste des alias et on prend le premier de la liste qui est sensé représenté l’élément concerné.

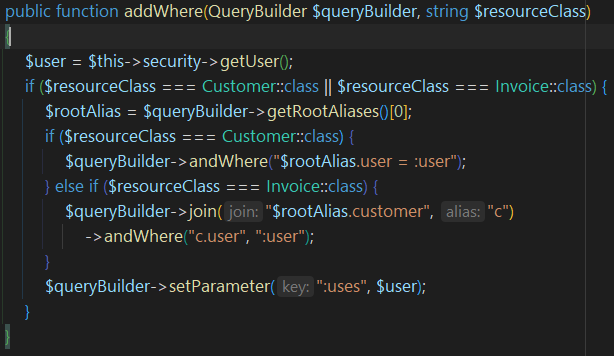
* On filtre avec une vérification si il s’agit d’une invoice ou d’un Customer && si le user est administrateur
* On détermine les alias pour chaque cas avec $queryBuilder->getRootAliases()[0]

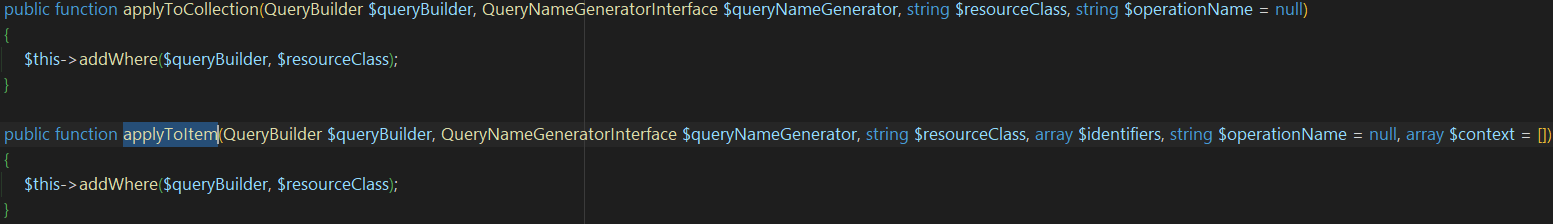


* Une fois que nous avons l’alias, il suffit de créer la requête dql avec celui-ci en fonction du contexte
* Et pour finir on détermine le parameter



Fonction addWhere()



Pour que ma fonction addwhere soit effective dans le cas de figure item ou collection il ne me reste plus qu’a appeler celle-ci dans les public function qu’implemente ma class, applyToCollection et applyToItem. 

Et le tour est joué, dorénavant chaque requête sur les invoices ou customers, ne ramènera uniquement que les éléments de l’utilisateur concerné. Et si l’utilisateur est administrateur, il récupèrera toutes les invoice et Customer

On peut également prendre en compte la faite de changer les consignes du fichier security.yaml, si demain je souhaite rendre les customers accessible a tout le monde même aux user non connecter, je dois m’assurer de pouvoir le faire en ajoutant une condition a

if (($resourceClass === Customer::class || $resourceClass === Invoice::class) && !$this->auth->isGranted("ROLE\_ADMIN")) {

if (($resourceClass === Customer::class || $resourceClass === Invoice::class) && !$this->auth->isGranted("ROLE\_ADMIN") && $user instanceof User) {

$user instanceof User

Ainsi même si je mais api/customers accessible au Anonymously , aucune erreur n’apparaitra l’ors de la recherche.

# React sur symfony

Nb : si des erreurs console apparaisse, utiliser php -S localhost:8000 -t public plutôt que le serve symfony

## Installation Webpack

Terminal :

« Composer require encore »

« npm install »

Librairie utile au projet

* « @babel/preset-react@version –save-dev »
* « react », « react-dom », “react-router-dom”, “axios”

On install

* Babel/preset-react, pour la conversion js->html dans les fichiers js
* React
* ReactDom qui permettra de relier notre code react au dom
* Axios qui permet de faire des requêtes depuis l’API

Il faut penser à décommenté .enableReactPreset() dans le fichier webpack.config.js

Le fichier app.js servira de passerelle entre ma vue twig (base.html.twig) mes fichier react

Test :

import ***React*** from 'react'  
import ReactDOM from "react-dom"  
import '../styles/app.css';

const App = () => {  
 return (  
 <>  
 <h1>Hello world</h1>

</>  
 )  
}

on lance run dev-server et si aucune erreur ne remonte, tout est installé

Trois dossiers principaux sont nécessaires,

* Pages : qui contiendra des pages a de mon site.
* Component : qui contiendra les petits bouts de page (Navbar…)
* Services : qui contiendra les function réutilisable

Création d’un component :

* Tout d’abord ont créé un fichier jsx genre Navbar.jsx, ensuite j’incorpore React

import ***React*** from 'react'

* ensuite un j’initialise une constante qui contiendra le retour html :

const HomePage = (props) => {  
 return (  
 <div className="jumbotron">  
 <h1 className="display-4">Hello, world!</h1>  
 <p className="lead">This is a simple hero unit, a simple jumbotron-style component for calling extra  
 attention to featured content or information.</p>  
 <hr className="my-4"/>  
 <p>It uses utility classes for typography and spacing to space content out within the larger  
 container.</p>  
 <a className="btn btn-primary btn-lg" href="#" role="button">Learn more</a>  
 </div>  
 );  
}

* ensuite j’export cette constante:

export default HomePage;

* et pour finir j’insert ce nouveau composent dans mon fichier app.js qui est relié au DOM

import HomePage from "./Pages/HomePage";  
  
const App = () => {  
 return (  
 <>  
 <div className="container pt-5">  
 <HomePage/>  
 </div>  
 </>  
 )  
}

Afin de pouvoir créer les liens de l’application , il faudra

* HashRouter (qui entourera les éléments dans app.js) il permet de faire des routes qui commence par Dièse « ex : #/customers ».Le # signifie que l’on reste sur la page de base <http://localhost>, mais que l’on change de section <http://localhost/#/customers> <http://localhost/#/invoices> ...
* Switch - Route : switch travail en concert avec Route, toutes les Route qui se situeront dans switch pourront ce voir attribuer un path et un component assigné

<Switch>  
 <Route path="/" component={HomePage}/>  
</Switch>

Ici on appelle la route qui mènera vers la HomePage, « / »

Lorsqu’il y a plusieurs routes les uns derrière les autre, alors il faudra mettre la route la plus détaillé en premier, car a la lecture , si les path commence par « / » il ne cherchera pas plus loin

<Switch>  
 <Route path="/customers" component={CustomersPage} />  
 <Route path="/" component={HomePage}/>  
</Switch>

## Mise en place de la CustomersPage

Tout d’abord on crée un tableau avec des donné fictive afin que le design de la page prenne forme, ensuite, une fois le résultat satisfaisant, on va chercher les éléments dans notre api pour les affichés a la place des donné fictive.

Pour cela :

Tout d’abord on créer ce qui sera le tableau final des donnés récolté avec axios

const [customers, setCustomers] = useState([]);

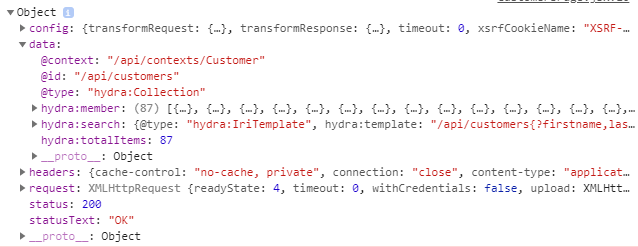
on initialise un tableau nommé customers qu’on set a « rien » ce sera le tableau à remplir.

On se servira de UseEffect pour utiliser axios (useEffect sert à préciser a react, que le ce qu’il y a à l’intérieur doit être exécuté a chaque initialisation de la page)

useEffect(() => {  
 ***Axios***.get("http://localhost:8000/api/customers")  
 .then(response => response.data["hydra:member"])  
 .then(data => setCustomers(data))

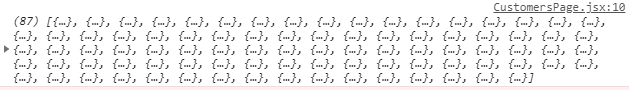
.catch(error => ***console***.log(error.response))  
}, [])

* On appelle Axios, on précise que la method d’appelle est GET « axios.get »,
* 0n précise l’adresse des données <http://localhost:8000/api/customers>
* On récupère un Object avec les données rechercher.



* On récupérer les données contenues dans l’objet

.then(response => response.data["hydra:member"])



* Et on transfert ces données dans la variable setCustomers initialiser en premier lieu

.then(data => setCustomers(data))

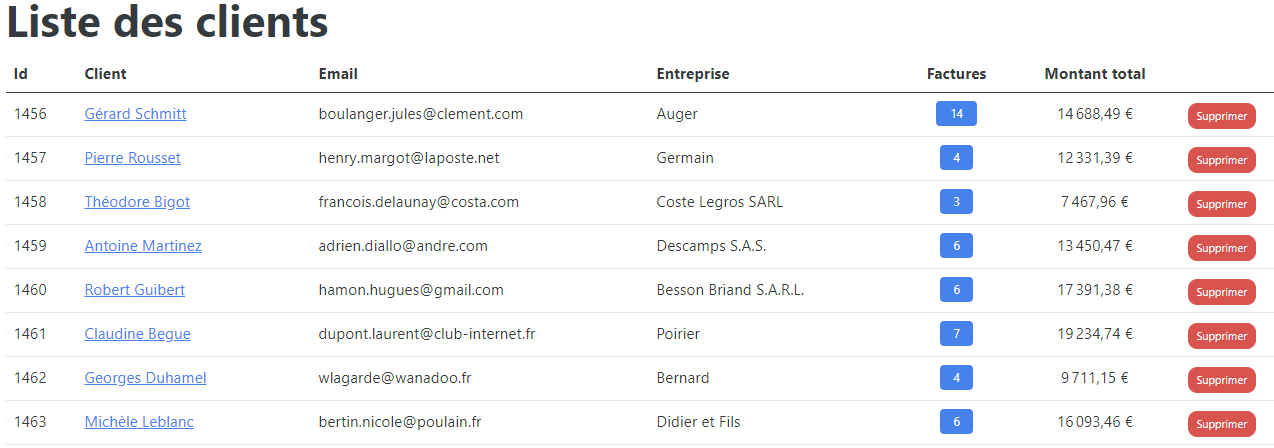
* On capture l’erreur si erreur il y a :

.catch(error => ***console***.log(error.response))

* Il ne reste plus qu’a remplacer les éléments fictifs du tableau par les éléments contenus dans customers grâce à une boucle map :

{customers.map(customer =>  
 <tr key={customer.id}>  
 <td>{customer.id}</td>  
 <td>  
 <a href="#">{customer.firstname} {customer.lastname}</a>  
 </td>  
 <td>{customer.email}</td>  
 <td>{customer.company}</td>  
 <td className="text-center">  
 <span className="badge bg-primary">{customer.invoices.length}</span>  
 </td>  
 <td className="text-center">{customer.totalAmount.toLocaleString()} €</td>  
 <td>  
 <button className="btn btn-sm btn-danger">Supprimer</button>  
 </td>  
 </tr>  
)}

Résultat :



### Suppression des éléments :

* On pose une function sur le bouton déclencheur

<button  
 onClick={() => handleDelete(customer.id)}  
 disabled={customer.invoices.length > 0} className="btn btn-sm btn-danger">Supprimer</button>

* On crée la function handleDelete(id)

function handleDelete(id) {  
 ***Axios***.delete("http://localhost:8000/api/customers/" + id)  
 .then(response => ***console***.log(response))  
}

on appelle le delete d’Axios, on lui passe l’adresse + l’id de l’élément à supprimer

deux approches sont possibles, l’approche dite optimiste, et l’approche dite prudente :

#### approche optimiste :

on part du principe, que tout va bien se dérouler, et donc on supprime l’élément visuellement au moment du clic, cette méthode a pour atout une expérience utilisateur plus fluide en revanche elle a comme default de laisser l’utilisateur dans l’ignorance si jamais l’élément n’a pas été supprimé.

//1. l'approche optimiste  
// On renvoi tous les customers du tableau sauf celui qui est sensé être supprimé  
setCustomers(customers.filter(customer => customer.id !== id))

#### L’approche prudente :

On part cette fois ci du principe, que la suppression peut échouer, et donc l’élément se supprimera visuellement seulement si celui-ci est supprimer de la base de données, atout : pas de méprise utilisateur, default: moins fluide.

function handleDelete(id) {  
 //1. l'approche prudente  
 ***Axios***.delete("http://localhost:8000/api/customers/" + id)  
 .then(response => setCustomers(customers.filter(customer => customer.id !== id)))  
 .catch(error => ***console***.log(error.response))  
}

#### Compromis:

Le compromis consiste a employer l’approche optimiste, mais si lors de la suppression en bdd il y a un problème, alors on réinjecte le tableau de customer original, de ce faite l’utilisateur aura beau cliquer sur l’élément a supprimer celui-ci reviendra systématiquement, il ne restera plus qu’a notifier l’utilisateur que l’élément ne peut être supprimer, on garde la rapidité de l’optimiste et la prudence de la deuxième approche.

function handleDelete(id) {  
 const originalCustomers = [...customers]  
 customers.filter(customer => customer.id !== id)  
 ***Axios***.delete("http://localhost:8000/api/customers/" + id)  
 .then(response => ***console***.log("ok"))  
 .catch(error => {  
 setCustomers(originalCustomers);  
 ***console***.log(error.response)  
 })  
}

## Mise en place d’une pagination

Création d’un useState pour la page current

const [currentPage, setCurrentPage] = useState(1);

commence par la page 1

création de la variable déterminant combien d’éléments il y aura par page

const itemsPerPage = 10;

10 éléments par page

On détermine combien de page il y aura

const pagesCount = ***Math***.ceil(customers.length / itemsPerPage)

on prend le nombre d’éléments total que l’on divise par le nombre d’éléments par page déterminé juste au-dessus.

Pour déterminer le nombre de bouton qu’il y aura en bas de la page il faut boucler sur le nombre de page calculer juste au dessu « pagesCount »

On crée une variable pages [] et pour chaque pagesCount, on ajoute une page a la variable pages.

const pages = [];  
for (let i = 1; i <= pagesCount; i++) {  
 pages.push(i)  
}

ensuite on a plus qu’a fair un map sur le <li> de la pagination.

{pages.map(page =>  
 <li className={"page-item" + (currentPage === page && " active")} key={page}>  
 <button className="page-link" onClick={() => handlePageChange(page)}>{page}</button>  
 </li>  
)}

Pour gérer la modification de page au clic, on crée une function handlePageChange que l’on place sur chaque button de chaque li.

Cette fonction recevra la variable « page » recuperer dans le map, et modifiera la currentPage ( page actuel) grace au setCurrentPage du useState.

const handlePageChange = (page) => {  
 setCurrentPage(page)  
}

il reste a créer le système de chargement d’item au changement de page :

on détermine l’emplacement où l’on se trouve (quel page)

const start = currentPage \* itemsPerPage - itemsPerPage

on prend la page actuelle que l’on multiplie par le nombre d’éléments par page et on soustrait le tout par le nombre d’éléments par page

ensuite on découpe nos customer en fonction de start :

const paginatedCustomers = customers.slice(start, start + itemsPerPage)

on prend tous les éléments on les découpe en donner comme début start, en fin start plus le nombre d’éléments par page

en clair admettons :customers = 80, currentPage = 3, itemsPerPage = 10,

start = 20

paginatedCustomer = on aura les éléments du 20ieme au 30iem soit 10 éléments

pour l’affichage, on aura plus qu’a map sur paginatedCustomer

{paginatedCustomers.map(customer =>

## Refactorisation des éléments réutilisables(ex. système de pagination)

Tout d’abord on crée le component Pagination.jsx, ensuite on créer une function constante dans laquelle on y insert la partie rendu de la pagination.

import \* as React from 'react';  
  
export const Pagination = () => {  
 return (  
 <div>  
 <ul className="pagination pagination-sm">  
 <li className={"page-item" + (currentPage === 1 && " disabled")}>  
 <button className="page-link" onClick={() => onPageChanged(currentPage - 1)}>&laquo;</button>  
 </li>  
 {pages.map(page =>  
 <li className={"page-item" + (currentPage === page && " active")} key={page}>  
 <button className="page-link" onClick={() => onPageChanged(page)}>{page}</button>  
 </li>  
 )}  
 <li className={"page-item" + (currentPage === pagesCount && " disabled")}>  
 <button className="page-link" onClick={() => onPageChanged(currentPage + 1)}>&raquo;</button>  
 </li>  
 </ul>  
 </div>  
 );  
};

Une fois cela fait, alors on invoque ce component a l’endroit ou celui-ci doit apparaitre (ex : en bas de la page des customers)

<Pagination/>

*Dans customersPage*

Dans le component de pagination, on s’aperçoit, que plusieurs paramètres utiles au component sont requis pour que celui-ci puisse fonctionner en autarcie. Il s’agit de déterminé ces paramètres et de les passer dans le component pagination appeler dans la page.



En clair, on a besoin de la page courante, du nombre de page, des pages elle-même, et de la fonction handleChangePage.

On lui passe en paramètre toutes les variables dont elle a besoin + l’équivalent, de la function de changement de page et les calculs utiles à ce component.

export const Pagination = ({currentPage, itemPerPage, length, onPageChanged}) => {  
 const pagesCount = ***Math***.ceil(length / itemPerPage)  
 const pages = [];  
 for (let i = 1; i <= pagesCount; i++) {  
 pages.push(i)  
 }

il ne reste plus qu’a transférer les informations des variables depuis la page parente avec des « alias »

<Pagination currentPage = {currentPage} itemPerPage={itemsPerPage} length={customers.length} onPageChanged={handlePageChange}/>

En clair on transfère tous les éléments, à travers des références, dont le component Pagination a besoin pour se générer tout seul.

Pour terminer la refactorisation, on va créer une function getData() dans le component pagination, qui va se charger de récupérer les customers (items) pour les affiché.

Pagination.getData = (items, currentPage, itemsPerPage) => {  
 const start = currentPage \* itemsPerPage - itemsPerPage  
 return items.slice(start, start + itemsPerPage)  
}

*Ici getData() prendra en parameter les « items » (customers, invoices…) la page actuelle, et le nombre d’item par page.*

const paginatedCustomers = customers.slice(start, start + itemsPerPage)

cette ligne deviendra :

const paginatedCustomers = Pagination.getData(customers, currentPage, itemsPerPage)

utilité, pour crée une pagination dans les autres pages, il suffira de mettre

const paginatedCustomers = Pagination.getData(  
 customers,  
 currentPage,  
 itemsPerPage  
)

et

<Pagination currentPage={currentPage} itemPerPage={itemsPerPage} length={customers.length}  
 onPageChanged={handlePageChange}/>

En bas de la page est le tour est joué.

### Autre alternative de pagination

Méthode pagination manuel, (expliqué précédemment)

* On appelle tous les éléments
* On les repartit sur plusieurs pages différentes
* On les affiche

Méthode Pagination API

* On appelle une adresse qui nous renverra une quantité d’élément prédéterminé dans l’entité de l’élément.
* On les affiche.

API Pagination :

On réutilisera la même customersPage pour l’exemple.

Ajout du useState totalItems

const [totalItems, setTotalItems] = useState(0);

Modification du useEffect pour préciser qu’il devra évoluer à chaque fois que la variable « currentPage » ce modifiera. On set les customers pour leur dire de ne récupérer que les customer afficher à l’adresse donné, on précise le nombre d’élément total et on set dans totalItems.

useEffect(() => {  
***Axios***.get(`http://localhost:8000/api/customers?pagination=true&count=${itemsPerPage}&page=${currentPage}`)  
 .then(response => {  
 setCustomers(response.data["hydra:member"]);  
 setTotalItems(response.data["hydra:totalItems"]);  
 })  
 .catch(error => ***console***.log(error.response))  
}, [currentPage])

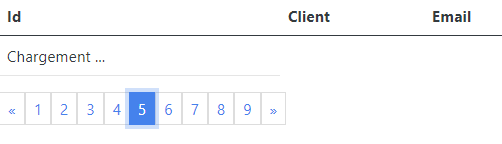
### création d’un system de chargement

* On réinitialise customers a 0 à chaque fois que l’on utilise la méthode handleChangePage()

const handlePageChange = (page) => {  
 setCustomers([]);  
 setCurrentPage(page)  
};

* On ajoute une condition au début du body , si customers est a 0 alors on afficher un

<tbody>  
{customers.length === 0 && (  
 <tr>  
 <td>Chargement ...</td>  
 </tr>  
)}



## Système de filtre

Création de la barre de recherche :

* D’abord on créer le formulaire de recherche en lui appliquant une fonction sur le changement « onChange » chaque fois que l’input sera modifier la function handleSearch sera appeler:

<div className="form-group">  
 <input type="text" name="" onChange={handleSearch} value={search} id="" placeholder="Rechercher..."  
 className="form-contrôle"/>  
</div>

* Création d’un useState search, initialiser sur une chaine de caractère vide.

const [search, setSearch] = useState("");

* Création de la function handleSearch qui remplira la barre de recherche au fur et a mesure que l’on tape.

const handleSearch = event => {  
 const value = event.currentTarget.value;  
 setSearch(value)  
}

* Dérivation de la recherche en fonction de la valeur de la barre de recherche : pour l’exemple, on filtre les éléments en fonction de leur nom et prénom. Donc on prend la liste complète des customers, et on ajout un filter

const filteredCustomers = customers.filter(  
 c =>  
 c.firstname.toLowerCase().includes(search.toLowerCase()) ||  
 c.lastname.toLowerCase().includes(search.toLowerCase())  
);

*c représente les customers, donc on filtre les customers en fonction de leurs prénoms ou leur nom inclus dans search. Le toLowerCase mais tous les mots en minuscule*

* ensuite on remplace la liste de pagination qui comprenait tout les customers, par les customer filtrer dans la function précédente

const paginatedCustomers = Pagination.getData(  
 filteredCustomers,  
 currentPage,  
 itemsPerPage  
)

* Amelioration : on fait en sorte que lorsqu’il n’y a qu’une seule page, alors la pagination n’apparait pas. On utilisera un if react :

{itemsPerPage < filteredCustomers.length && <Pagination  
 currentPage={currentPage}  
 itemPerPage={itemsPerPage}  
 length={filteredCustomers.length}  
 onPageChanged={handlePageChange}/>  
}

* Une dernière étape est nécessaire pour finaliser le processus de recherche. actuellement si on se rend a une autre page que la première, si l’on fait une recherche dans la barre de recherche, il n’y a plus rien qui s’affiche, c’est simple aucune réinitialisation de page n’est utiliser entre les deux action, donc on se retrouve page cinq de la recherche. Pour corriger ça, il faut tout simplement réinitialiser la currentPage a 1 pour chaque recherche.

const handleSearch = event => {  
 const value = event.currentTarget.value;  
 setSearch(value)  
 setCurrentPage(1)  
}

## Refactorisation !

Suppression des requêtes axios de customersPage, création du service qui accueillera les requêtes liées au customers

useEffect(() => {  
 customersAPI.findAll()  
 .then(data => setCustomers(data))  
 .catch(error => ***console***.log(error.response))  
}, [])

*on transfère tous ce qui concerne axios dans une function situer dans le fichier customersAPI.js*

function findAll() {  
 return ***Axios***.get("http://localhost:8000/api/customers")  
 .then(response => response.data["hydra:member"])  
}

### Utilisation des try catch avec await et async

Jusqu’à maintenant on utiliser les .then.catch pour capturer et afficher les erreurs

function handleDelete(id) {  
 const originalCustomers = [...customers]

customers.filter(customer => customer.id !== id)  
 customersAPI.delete(id)  
 .then(response => setCustomers(customers.filter(customer => customer.id !== id)))  
 .catch(error => {  
 setCustomers(originalCustomers);  
 ***console***.log(error.response)  
 })  
}

Une méthode dite “plus propre“ consiste a enfermé la tentative dans un try et à renvoyer une erreur contenue dans catch :

const handleDelete = async id => {  
 const originalCustomers = [...customers]  
  
 setCustomers(customers.filter(customer => customer.id !== id))  
 customers.filter(customer => customer.id !== id)  
  
 try {  
 await customersAPI.delete(id)  
 } catch (error) {  
 setCustomers(originalCustomers);  
 }  
}

a priori une erreur devait survenir dans la console si l’on mettait un async directement dans le useEffect mais forcé de constater qu’il n’est plus utile de déporté le contenu du useEffect dans une function annexe puisque il n’y a plus d’erreur

useEffect(() => {  
 customersAPI.findAll()  
 .then(data => setCustomers(data))  
 .catch(error => ***console***.log(error.response))  
}, [])

useEffect(async () => {  
 try {  
 const data = await customersAPI.findAll()  
 setCustomers(data)  
 } catch (error) {  
 ***console***.log(error.response)  
 }  
},[])

## Les constantes choices

Les constantes choices sont l’équivalent des variables global en php, on les crée au-dessus de la classe, on leurs donnent un titre, une clé, et une valeur.

const STATUS\_LABELS = {  
 PAID: "Payée",  
 SENT: "Envoyée",  
 CANCELLED: "Annulée"  
}

et pour les utilisée, on les concatène à l’endroit voulus

{STATUS\_LABELS[invoice.status]}

*Ici, si le statut est PAID, alors le label sera remplacé par Payée…*

### Utilisation de momentjs formatage des dates

* Npm install moment
* Import de la librairie import ***moment*** from "moment"
* Création de la fonction formatDate :

const formatDate = (str) => moment(str).format("DD/MM/YYYY")

*On envoie une chaine de caractère a moment, on lui impose un format.*

* Utilisation :

<td className="text-center">{formatDate(invoice.sentAt)}</td>

## Création d’un formulaire de connexion

Pour modifier les deux champs dans le même useState :

const [credentials, setCredentials] = useState({  
 username: "",  
 password: ""  
});

on s’appuiera sur le name de l’input lors de la modification de la value :

const handleChange = (event) => {  
 const value = event.currentTarget.value;  
 const name = event.currentTarget.name;  
 setCredentials({...credentials, [name]: value})  
}

<input  
 value={credentials.username}  
 onChange={handleChange}  
 type="email"  
 placeholder="Adresse email de connexion"  
 name="username"  
 id="username"  
 className="form-control"  
/>

On récupère la valeur, on récupère le name, pour préciser que l’on parle bien de cette input-là, en l’occurrence : « username » et on set la valeur de username avec la value déjà entré + la nouvelle valeur ajoutée.

Pour récupérer le token de la personne qui essaye de ce connecté on fait un appel a l’API dans un nouveau fichier « AuthAPI.js » ce chargeant de traité toute les requête axios.

function getToken(credentials) {  
 return ***Axios***.post("http://localhost:8000/api/login\_check", credentials)  
}

*récupération du token en se servant du credentials passé en paramètre lors du login*

Affichage du message d’erreur en bootstrap grâce a la classe « invalid-feedback »

Cette classe affiche un message d’erreur si la classe « is-invalid » est trouver dans l’input.

1. On crée un useState error initialisé a string vide.
2. On crée le paragraphe d’erreur en dessous de l’input concerné ici « username » on fait en sorte qu’il ne s’affiche que si la variable « error » existe.

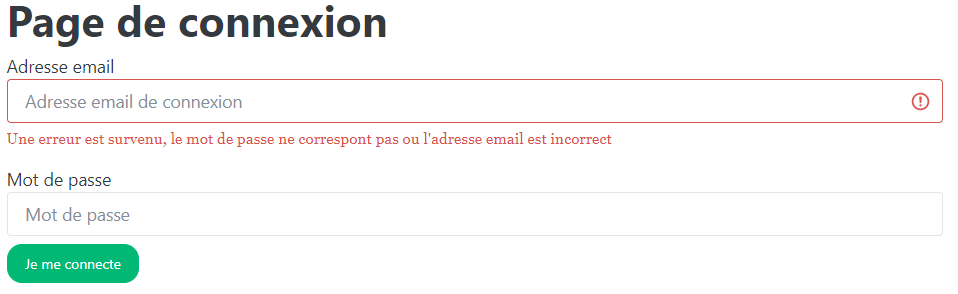
<input  
 value={credentials.username}  
 onChange={handleChange}  
 type="email"  
 placeholder="Adresse email de connexion"  
 name="username"  
 id="username"  
 className="form-control"  
/>  
{error && <p className="invalid-feedback">{error}</p>}

1. Dans le catch de recherche de token dans l’api, on insère dans la variable « error » la phrase d’erreur souhaitée

const handleSubmit = async event => {  
 event.preventDefault();  
 try {  
 await AuthAPI.getToken()  
 .then(response => ***console***.log(response))  
 } catch (error) {  
 setError("Une erreur est survenu, le mot de passe ne correspond pas ou l'adresse email est incorrect")  
 }  
}

1. On ajoute la classe « is-invalid » dans l’input, si « error » existe.

<input  
 value={credentials.username}  
 onChange={handleChange}  
 type="email"  
 placeholder="Adresse email de connexion"  
 name="username"  
 id="username"  
 className={"form-control" + (error && " is-invalid")}  
/>



En revanche si tout se passe bien, le but est de récupéré le token et de l’insérer dans localStorage

1. Suite à la requête api, je récupère le token dans la réponse

const token = await ***Axios***.post("http://localhost:8001/api/login\_check", credentials)  
 .then(response => response.data.token)

1. Je mets le setError a vide.
2. Je stock le token récupéré dans le localStorage

***window***.localStorage.setItem("authToken", token)

1. Je populle mon header avec le token récupéré

***Axios***.defaults.headers['Authorization'] = "Bearer " + token

Pour refactorisé tout ça il faudra

1. Créer un fichier AuthAPI.js qui se chargera de créer le token de le stocker dans le localStorage, et d’ajouter celui-ci dans le header.

function authenticate(credentials) {  
 return ***Axios***.post("http://localhost:8001/api/login\_check", credentials)  
 .then(response => response.data.token)  
 .then(token => {  
 // Stockage du token dans le localStorage  
 ***window***.localStorage.setItem("authToken", token)  
 //On prévient Axios qu'on a maintenant un header par default sur toutes nos futures requête HTTP  
 ***Axios***.defaults.headers['Authorization'] = "Bearer " + token  
 return true;  
 })  
}

1. Refactorisation du handleChange:

Avant:

1. const handleChange = (event) => {  
    const value = event.currentTarget.value;  
    const name = event.currentTarget.name;  
    setCredentials({...credentials, [name]: value})  
   }
2. const handleChange = ({currentTarget }) => {  
    const value = currentTarget.value;  
    const name = currentTarget.name;  
    setCredentials({...credentials, [name]: value})  
   }
3. const handleChange = ({currentTarget}) => {  
    const {value, name} = currentTarget  
    setCredentials({…credentials, [name]: value})  
   }

*On déstructure le paramètre pour n’utiliser que le currentTarget*

### Création du système de déconnection :

1. Création de la fonction logout dans AuthAPI

function logout() {  
 ***window***.localStorage.removeItem("authToken");  
 delete ***Axios***.defaults.headers["Authorization"];  
}

*Ici on détruit la session du localStorage, et supprime le bearer token insérer pendant la connexion.*

1. Transformation du « a » en « button » dans le component Navbar -> déconnection et passage d’une function « handleLogout »

<li className="nav-item">  
 <button onClick={handleLogout} className="btn btn-danger">Déconnexion</button>  
</li>

1. Création de la function appelant AuthAPI.logout

const handleLogout = () => {  
 AuthAPI.logout()  
}

Teste de fonctionnement:

On va chercher les customers avec un findAll a l’authentification, que l’on place dans console.log.

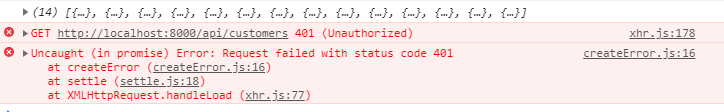
**customersAPI.findAll().then(*console*.log)**

***Si un console.log est placer dans un .then sans parenthèse, alors il affichera en console systématiquement ce que le findAll ou autre renvoi***

On ajoute la même chose dans la déconnection :

function logout() {  
 ***window***.localStorage.removeItem("authToken");  
 delete ***Axios***.defaults.headers["Authorization"];  
 customersAPI.findAll().then(***console***.log)  
}

Si on se connecte on reçoit les customers et lors de la déconnection, on reçois une erreur qui nous signal un 404 comme quoi on a pas accès au customer dans la mesure ou l’on est pas connecté.



### Navbar réactive

Ajout du composant NavLink pour les liens :

<NavLink to="/login" className="btn btn-success">Connexion</NavLink>

*NavLink fait partit de react, on l’associe a un to= ‘’ pour déterminer le chemin*

### Maintenir les accès au page après l’authentification

Pour le moment, si on se connecte on a accès au page, invoices et customers, mais si l’on rafraichi la page, l’accès a sauté bien que le token soit tjr en place. Pour rectifier ça, quelque vérification s’impose à chaque rafraichissement de page.

1. Créer une function que l’on appellera dans app.js, cette fonction ce chargera des vérifications.
2. Création de cette function dans AuthAPI.js
   1. Vérification si un token existe

const token = ***window***.localStorage.getItem("authToken")

* 1. Vérification si le token est encore valide (durée d’expiration) en utilisant jwt-décode, qui nous servira a comparé la date d’expiration au format timestamp du token avec la date actuelle.

if (token) {  
 const jwtData = jwtDecode(token)  
 if (jwtData.***exp*** \* 1000 > new ***Date***().getTime()) {  
 ***Axios***.defaults.headers['Authorization'] = "Bearer " + token  
 }  
}

*on récupère la date d’expiration du token que l’on multiplie par 1000 pour avoir un résultat en milliseconde comme a la création d’une date avec getTime()*

*et on compare les deux, si la date d’exp est plus grande que la date d’aujourdui, alors, on réinjecte le bearer token dans le headers*

1. je logout si le token n’existe pas ou si l’expiration du token est passé

if (token) {  
 const jwtData = jwtDecode(token)  
 if (jwtData.***exp*** \* 1000 > new ***Date***().getTime()) {  
 ***Axios***.defaults.headers['Authorization'] = "Bearer " + token  
 } else {  
 logout()  
 }  
} else {  
 logout()  
}

### gestion de la réactivité de la navbar avec l’utilisation d’un Boolean :

« isAuthenticated » initialisé dans app.js

const [isAuthenticated, setIsAuthenticated] = useState(false);

cette variable sera false en déconnecté et true en connecté

pour ce faire, on envois cette variable en props a la navbar, lui permettant d’être modifier ou consulté depuis celle-ci a volonté. OnLogout permet de modifier la variable isAuthenticated.

<Navbar isAuthenticated={isAuthenticated} onLogout={setIsAuthenticated()}/>

On veut également envoyer cette variable, props a la LoginPage, et pour cela on a une propriété pouvant contenir des variables à travers une function :

<Route path="/login"  
 render={props => <LoginPage isAuthenticated={isAuthenticated}

onLogin={setIsAuthenticated}/>}  
/>

*Au lieu de component={LoginPage} render nous permet d’envoyer des variables dans LoginPage*

Ainsi la LoginPage est capable de savoir si on est authentifié ou non est peut également modifier la valeur de celle-ci.

Ensuite une fois la variable envoyée à la navbar, on peut les utiliser et modifier leur valeur comme ceci :

const Navbar = ({isAuthenticated, onLogout}) => {  
  
 const handleLogout = () => {  
 AuthAPI.logout();  
 onLogout(false)  
 }

*Ici on récupère les props envoyé depuis l’app.js et on modifie la valeur de onLogout pour le mettre a false lors de l’appel a handleLogout*

Même chose pour le login, on a passé la variable isAuthenticated a la LoginPage donc on peut l’utilisé,

export const LoginPage = ({onLogin}) => {  
 const [credentials, setCredentials] = useState({  
 username: '',  
 password: ''  
 });

*Ici comme dans la Navbar, on déstructure les props récupéré et on récupère donc onLogin*

On a plus qu’a ajouter la modification de la valeur de onLogin lors du submit si tout va bien

const handleSubmit = async event => {  
 event.preventDefault();  
 try {  
 await AuthAPI.authenticate(credentials)  
 onLogin(true)  
 setError("");  
 } catch (error) {  
 setError("Une erreur est survenu, le mot de passe ne correspond pas ou l'adresse email est incorrect")  
 }  
}

on peut ensuite utiliser la variable isAuthenticated que l’on a passer dans la navbar, pour vérifier si elle est true ou false et ainsi afficher « déconnection » ou « s’inscrire et connexion »

{!isAuthenticated && (  
 <>  
 <li className="nav-item">  
 <NavLink to="/register" className="nav-link">Inscription</NavLink>  
 </li>  
 <li className="nav-item mx-2">  
 <NavLink to="/login" className="btn btn-success">Connexion</NavLink>  
 </li>  
 </>  
) || (  
<li className="nav-item">  
 <button onClick={handleLogout} className="btn btn-danger">Déconnexion</button>  
</li>  
)}

## Prendre en compte l'état de l'authentification au démarrage de l'application

L’objectif est de créer une fonction qui renverra un Boolean qui déterminera si oui ou non on est connecté. Et d’utiliser cette fonction comme initialisation du useState isAuthentication

function isAuthenticated() {  
 const token = ***window***.localStorage.getItem("authToken")  
 if (token) {  
 const jwtData = jwtDecode(token)  
 if (jwtData.***exp*** \* 1000 > new ***Date***().getTime()) {  
 ***Axios***.defaults.headers['Authorization'] = "Bearer " + token  
 return true  
 } else {  
 return false  
 }  
 } else {  
 return false  
 }  
}

on utilise ensuite cette fonction dans le useState d’app.js comme valeur par défaut a la variable isAuthenticated

const [isAuthenticated, setIsAuthenticated] = useState(AuthAPI.isAuthenticated());

ainsi nous avons la valeur d’isAuthenticated tel quelle a été laissé lors d’un rafraîchissement de page

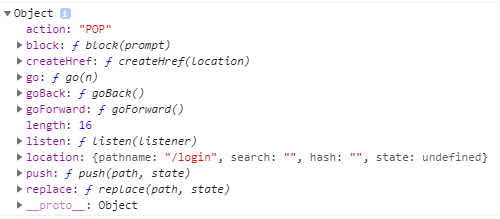
## Les redirections après connexion ou déconnection

Lorsqu’un élément de HashRouter Route est utilisé, il contient des props qui lui son propre, et qui peuvent être utilisé dans la page du lien, tel que « history » history possède des méthodes de redirection tel que replace (qui remplace l’adresse actuel par une autre) ou push (qui ajoute ladresse suivante a celle en place précédemment)

Dans un premier temp on précise sur la route que « login » que l’on souhaite récupérer tout props contenu dans la méthode Route

<Route  
 path="/login"  
 render={props => <LoginPage onLogin={setIsAuthenticated} **{...props}/>}**  
/>

A partir de la, on retrouvera toute les propriété de la route dans sur la page « LoginPage »



Au moment de la submission du formulaire de login, si tout ce passe bien (dans le try), on remplace l’adresse actuelle par la page des customers

const handleSubmit = async event => {  
 event.preventDefault();  
 try {  
 await AuthAPI.authenticate(credentials)  
 onLogin(true)  
 setError("");  
 history.replace("/customers")  
 } catch (error) {  
 setError("Une erreur est survenu, le mot de passe ne correspond pas ou l'adresse email est incorrect")  
 }  
}

un problème survient si l’on souhaite avoir toutes les propriétés d’un route dans un component.

Exemple si on souhaite que le component Navbar ai toute les props d’une Route :

On peut lui ajouter « withRouter » de cette façon :

Const NavbarWithRouter = withRouter(Navbar) en n’oubliant pas d’importer WithRouter de « react-router-dom »

import {HashRouter, Route, Switch, withRouter} from "react-router-dom";

const NavbarWithRouter = withRouter(Navbar)

Ensuite il ne me reste plus qu’a appeler history en paramètre de ma class Navbar

const Navbar = ({isAuthenticated, onLogout, history}) => {

et à mettre un history.replace("/login") dans le logout :

const handleLogout = () => {  
 AuthAPI.logout();  
 onLogout(false)  
 history.replace("/login")  
}

## Protéger les routes non accessibles aux personnes non connectées

Au lieu d’utiliser component={CustomersPage} dans mes routes, j’utiliserai plutôt render, qui me permet de placer une function au milieu de ma route :

On utilisera le component de react-router-dom « redirect », je pose la question « suis-je authentifiée » si je le suis, je vais vers CustomersPage en n’oubliant pas de lui redonner les props lié à la Route, en revanche si je ne suis pas authentifié je suis redirigé vers « /login »

<Route  
 path="/customers"  
 render={  
 props => isAuthenticated ? (<CustomersPage {...props} />) : (<Redirect to="/login"/>)}  
/>

Faire ca pour chaque route est très récurent et très fastidieu, on créera plutôt une classe PrivateRoute afin de refactorisé ce bout de code :

const PrivateRoute = ({path, isAuthenticated, component}) =>  
 isAuthenticated ? (  
 <Route  
 path={path}  
 component={component}/>  
 ) : (  
 <Redirect to="/login"/>  
 );

pour donner finalement :

<PrivateRoute path="/customers" isAuthenticated={isAuthenticated} component={CustomersPage}/>

*Une ligne sans le teste qui est compris dans la classe PrivateRoute()*