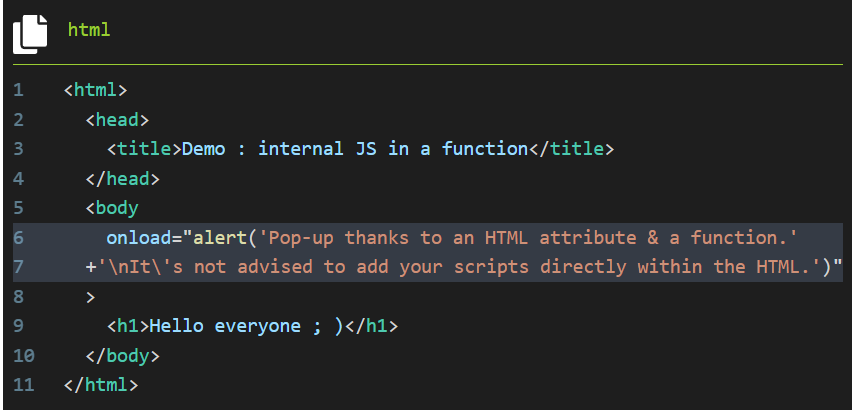
Notes cours WEB2 (JS Vanilla)

Cours de Web 2 => axé Vanilla JS, no framework

Préférer séparer les catégories de code (en .js, .html, .css,...)

Et surtout éviter le JS intégré dans un element html (les scripts servent à ça.) => valide en codant en JS Vanilla. Autrement cela est souvent utilisé.

MODULE 1 : IHM CLASSIQUE & INTERACTIONS

Ex : JS dans du html (onload :event declénché après le tag body)

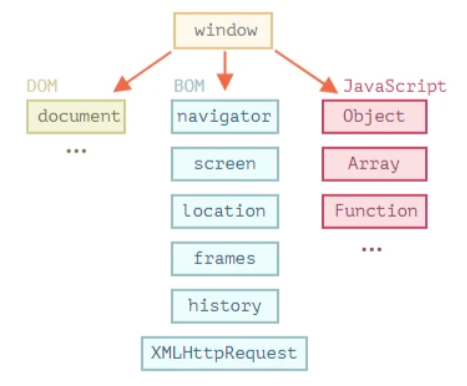
Chaque catégorie de code séparé : L’alerte se trouve dans external-scripts.js et on utilise le tag script.

Comment le browser interprete du code js ?

HTML (+CSS) & JS sous forme de bytes into parsing into tokens into node è DOM

La construction du DOM sera bloquée jusqu’à la fin du l’exec du script JS.

* IMPORTANCE DE METTRE LE <script> APRES LA BALISE <body>

Son equivalent CSS est le CSSOM. (mais ils ne sont pas interdépendants).

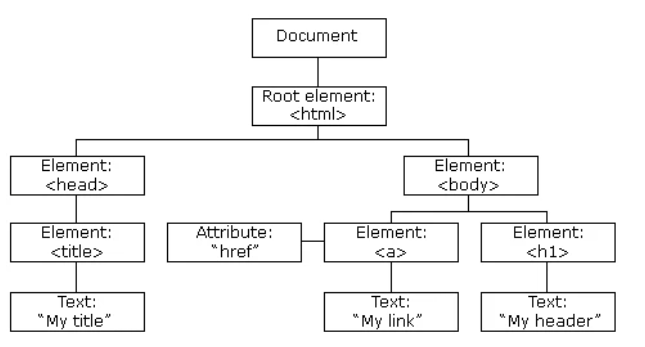
Les 3rd party APIS sont les API localisées dans des serveurs externes.

Les Browser APIS (ex : DOM API, Web Storage APIs,…) permettent d’interagir avec ceux-ci.

En JS il existe des objets qui permettent d’y accéder

Le DOM est une SD sous forme d’arbre qui structure les données de ma page. Quand je change le script, il est automatiquement réafficher. Je peux modifier l’HTML et le CSS grâce à ça.

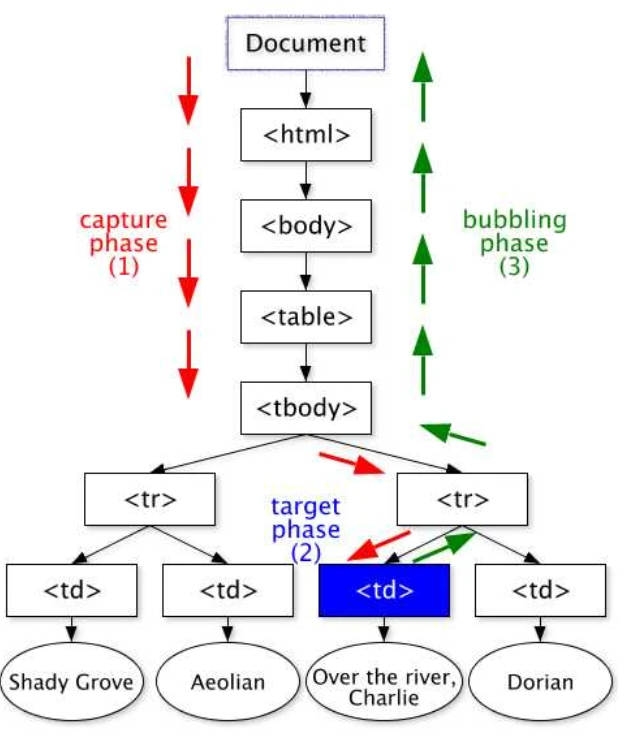
C’est le DOM qui offre les méthodes à cet effet.



Ces méthodes se retrouvent ici : <https://developer.mozilla.org/en-US/docs/Web/API/Document>

Quelques exs : getElementById(), querySelector(),…

Gestion des events en ajoutant des events listeners (prog. Evenementielle).

Flux des evenements du DOM

Ajouter ou enlever des events listeners : **addEventListener() et removeEventListener()**

**à l’intérieur de l’écouteur d’evenement je peux y mettre un event parmi plusieurs qui existent => ex : ‘’click’’ è addEventListener(‘’click’’)**

**Callbacks**

Les callbacks sont des fonctions appelées une fois le temps venu. On donne une fonction en paramètre pour qu’elle s’exécute en temps voulu.

La callback est une fonction passée en paramètre à une autre fonction.

Il faut y passer la DEFINITION d’une fonction et non son appel !

Ex : const displayAlert = titre.addEventListener(‘’click’’, () => alert(‘’Hello World’’)) è je définis la fct

Et pas : titre.addEventListener(‘’click’’, alert(‘’Hello World’’)) è je l’appelle tout de suite

Ensuite pour l’appeler => titre.addEventListener(‘’click’’, displayAlert) (sans parenthèses !!)

**L’objet event**

Par défaut l’objet event est passée au callback d’un écouteur d’évènement

On peut le définir (e par exemple)

Synchrone => en temps réel, réponse immédiate

Asynchrone => réponse dès que possible, une fois que les actions simultanées sont finies (permet de renvoyer un element complet par exemple)

MODULE 2 : IHM MODERNE & DYNAMIQUE

Si je veux use une librairie externe (ou dépendance) è <script> dans mon html

Les modules bundlers sert a mieux gérer les dépendances, scripts, etc… (cf npm, le gestionnaire de packages de npm).

Ça « bundle » tous mes fichiers contenant les scripts, les dépendances… et ça le wrap dans des fichiers .js, .css,… plus compact.

En ayant importé un projet git dans mon projet git, mon repo ne pourra pas le reconnaitre. Pour remédier à cela je dois supprimer le fichier .git (& .gitignore) du projet importé.

Le .json est le fichier de configuration de mon projet.

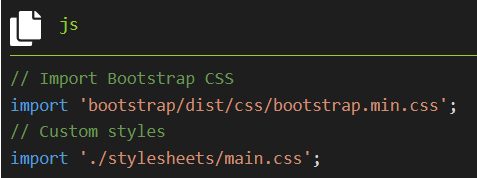
Le **linter** sert a nous aider a détecter les erreurs de programmation (extension ESLint)

Son fichier de configuration se trouve a la racine du projet

Le **formatage** est offert grâce à l’extension prettier. Il figure dans le fichier de config de linter.js mais il faut installer l’extension vsc, et il a aussi son fichier config a la racine du projet (automatiquement généré)

* Pour formater avec l’extension :
  + Soit SHIFT + ALT + F
  + Clic droit => format document

L’importation de scripts & dépendances : Webpack ne prend pas en compte l’importation de scripts dans index.html, il prend les **importations dans index.js.**



La commande NPX (pas NPM) permet de créer un serveur temporaire et de run l’application tout de suite. Le package serve offre un serveur de fichiers statiques.

Responsive Web Design è un design qui rend bien sur tous les supports. Pour nous aider on peut utiliser SASS ou Tailwindcss/Bootstrap

**Module 3 : IHM Structurée, routage & Anims**

Utilisation de modules pour faciliter le code JS à empaquetages de code, module possèdent des objets dedans.

Pour en créer un : faire son script JS (créer son objet) puis export avec le nom de mon fichier js.

Si je veux qu’un objet soit privé au sein d’une librairie, il suffit de ne pas l’exporter. L’encapsulation n’existe pas en JS, il faut utiliser d’autres modules.

Diff entre default et named export : le default permet d’importer avec le nom qu’on veut, au contraire du named. Pour exporter des fonctions, ou pour exporter plusieurs éléments, le named est à utiliser

Pour en importer, je spécifie le chemin et je lui donne un nom (préférer le nom que l’on a choisi pour l’export). Si c’est une librairie externe, en général il se trouvera dans le /lib/.

Pour les named import, je dois les mettre entre accolades avec son nom exact. Je peux ajouter un « as » (toujours entre accolades) pour lui faire changer de nom.

Pour le routing (l’affichage de différents layouts), on utilise normalement une librairie pré-existance (cf express), mais dans le cours on va bientôt apprendre comment créer son propre routeur.

**Module 4 : Service web & clients**

Pour exporter : module.exports = objet (1)

Module.exports = {o1,o2} (plusieurs)

À placer à la fin du script

Pour importer : nom = require(‘../route’) ;

On a aussi l’accès aux packages de la communauté, après installation on mentionne son nom

// module used after package installation

const shortid = require('shortid');

Questions séance 2 : Un middleware c quoi ?

C’est un app.use qui sera utilisé à chaque requête. L’ordre des middlewares a une importance. On les reconnait par app.use(…) ou app est une instance de express()

On peut y passer 3 params : req,res, next,…

On peut aussi filtrer les middlewares pour qu’ils s’utilisent par un certain chemin (‘/’,’/users’,…)

Les middlewares sont censé s’enregistrer par routeurs (facilité de lire)

Middlewares d’app => middlewares autiliser pour CHAQUE REQUETE

Routeurs => app les appelle

**Serveur backend** => machine qui va fournir les données

La SPA est ce qu’on va voir cette année, avec des fichiers html rendus sur le browser. (client site rendering) en MPA c la génération d’html.

Dans les deux cas on va avoir besoin de serveurs backend qui envoie les données

Le frontend envoie les fichiers et communique avec le navig, c’est lui qui render

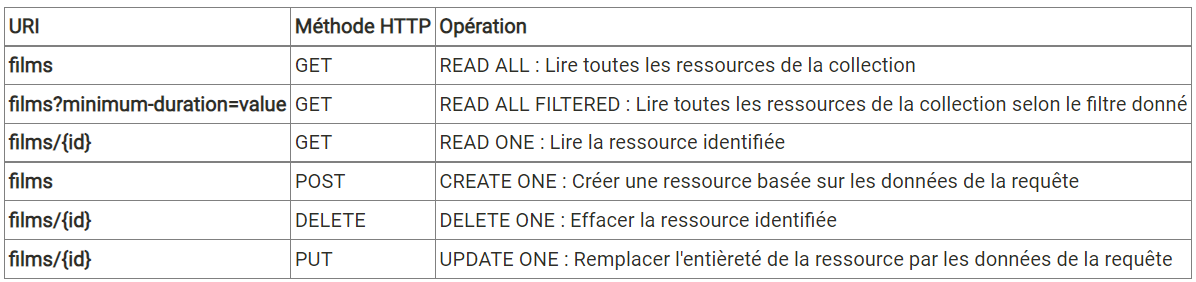
Le backend va enregistrer, modifier les données, mais il ne sait pas comment les render.

Quand il effectue une modif, le Backend l’envoie au browser, et c’est le rendering chargé au navigateur qui s’occupe de le rendre. En bref, frontend et backend servers ne communiquent jamais.

Note : on peut avoir les fichiers HTML/CSS/JS dans les serveurs backend, mais si utilisation de JS compliqué/bcp de modules, un serveur frontend est nécessaire

Backend => JSON only

On n’a pas vu encore comment communiquer avec notre backend depuis le navigateur.

**Paramètres de route ?**

Penser au CRUD.

Get => READ

Post => CREATE

Delete => DELETE

PUT => UPDATE

RESTful API nous imposent d’utiliser un parametre de **chemin** au lieu d’un **parametre de requête ( ?id=2)**

Mais on va utiliser les param de requetes pour filtrer à la place.

Module 5 Notes

Il existe des données stockées dans le browser. On appelle ça le localStorage et la route peut récupérer ces données pour les faire persister dans l’app.

Sérialiser -> passer d’un langage humain à une sérialisation de mes données en mémoire : de objet vers json

Parsing -> inverse de sérialiser : de json vers object ;

Sérialiser permet d’utiliser les données pour tout support possible, car langage commun a tous les devices (mobile, browser, langage de programmation,…).

Fat model c quoi ?

En express on a l’architecture MVC. Mais dans l’API elle n’a aucun sens car on n’a pas de vue. On a les controllers tho.

On appelle ça un fat model (en express : MODELS) car c’est une architecture a deux couches : couche controleur (routes, requetes etc, en express : ROUTES), et la couche fat model (contient les données, les reçoit renvoie ex : les fonctions qui récupère les pizzas et traite pour récupérer ce qui est prêt a être envoyé par notre controller)

Il y a des vérifications d’input au niveau du client mais une deuxième fois aussi au niveau du backend.

Module 6 Notes

On utilise une API et un Frontend : 2 apps, deux ports différents.

Les fonctions asynchrones (ex : fetch) ont une priorité plus basse que les autres, c’est ce pourquoi elles sont exécutées par la suite.

En utilisant le .then(), toutes les fonctions asynchrones sont chainées au fetch.

Le fetch peut être : en cours, réussi ou echec. Pour récupérer l’état du fetch(), on utilise .then(callback)

Q&A :

Localhost est un synonyme de 127.0.0.1 …. (adresse ip locale du réseau de l’école)

Ainsi, les gens qui partagent ce réseau peuvent charger le frontend de quelqu’un d’autre et y apporter des modifications. La personne qui partage son ip devient un provider. Et dans le cas ou mon frontend et mon api sont splités, il faut 2 providers.

Problem de CORS :

SOP (single origin policy) :

Données de session (cookies) vont voyager automatiquement et des sites peuvent s’en servir pour injecter des requêtes pour voler mes données bancaires, effectuer des virements, etc.

C’est pour cela que le SOP existe. Je n’autorise qu’une origine. Les browser imposent une seule origine pour prévenir ce genre de situation

CORS : je relâche cette sécurité et j’accepte que des ressources proviennent d’autres origines. Cela s’autorise via du code JS.

Proxy ?

Serveur qui redirige des requêtes

Frontend : recupere serv de fichier statique

Si je veux faire une requête avec l’alias /api/pizzas, je passe par le proxy.

L’origine d’une requête vers API doit autoriser à chaque fois une nouvelle origine. Ce n’est pas pratique de faire manuellement ; c’est pour ça qu’on relâche la sécurité OU on met en place un proxy.

Proxy OU rajout de CORS.

Le proxy tourne sur le même serveur que le serveur de fichier statiques.

Ainsi, je peux utiliser le frontend d’un autre (réseau partagé, qui lui a autorisé les autres origines), afin d’executer mes requetes RESTClient.

Je peux configuer url et dns va résoudre cet url into adresse IP du serveur.

Sujet n2 : APPELS ASYNCHRONES

C’est quoi une Promesse ?

Je donne la promesse, il y a qqch a l’intérieur pas encore a maturité. A l’intérieur il y a un résultat ou une erreur. Pendant que ça mature je fais autre chose a un autre client.

Il faut savoir que quand on utilise un fetch, le chainement des traitement EST DIFFERENT.

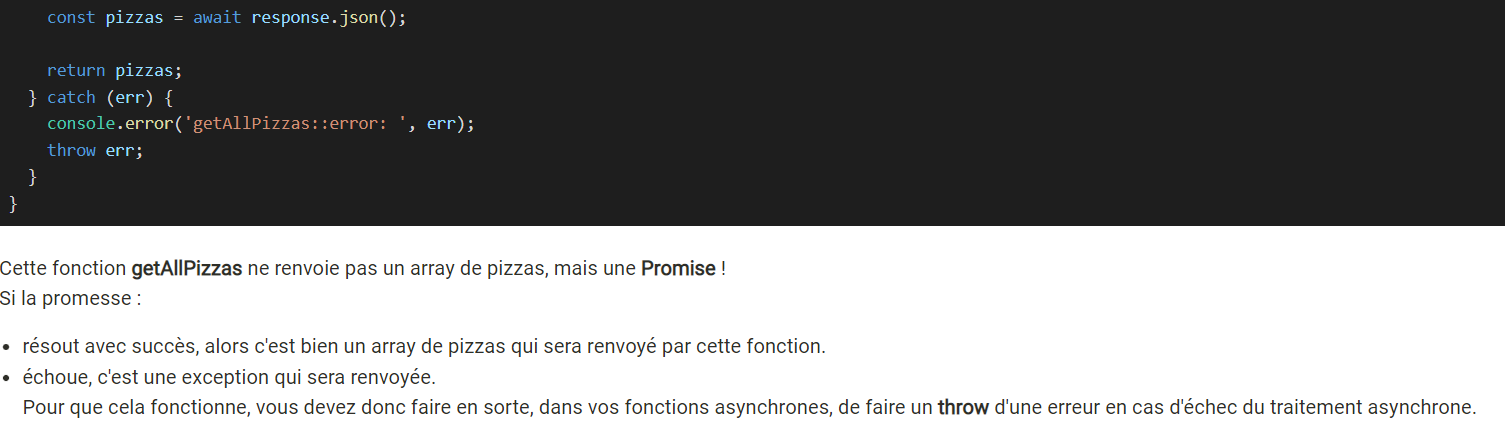
Si je veux chainer les traitements, je dois rester dans le then, dans la promesse.

Si j’en sors, la promesse donne sa reponse avant même que probleme se resolve

AWAIT ? tu passes pas a la ligne d’apres avant que mon paquet ne s’ouvre.

Await ne va pas sans async.

ASYNC ?

Dans la definition de ma fonction.

Pour faire un eventListener sur un submitBtn (elementHTML) et que je cherche l’ID du film (en l’ayant setqqpart dans un element html parent ou target) :

Une image contenant texte

Description générée automatiquement



Une image contenant texte

Description générée automatiquement

**Module 7 : Authentifications/Sécurisation**

Authentification stateful : le serveur s’occupe de sauvegarder les données de connexion de l’user (id => persiste sur la session par les cookies)

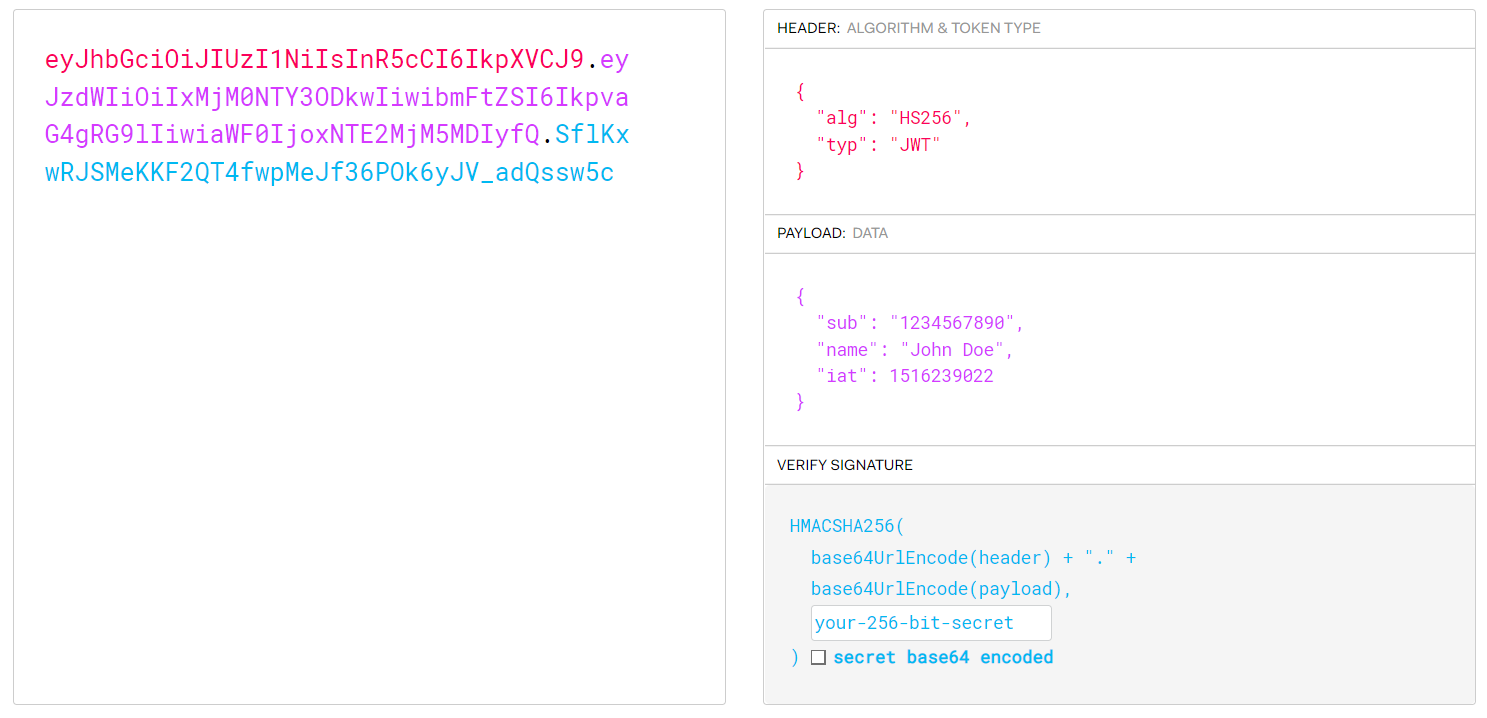
Authentification stateless : Le client (le browser) sauvegarde les données de connexion via les JWT (JSON Web Tokens)

Cela réside sur le principe que le serveur ne retient pas les données de l’utilisateur

Si le serveur redémarre, il n’y a pas d’authentification nécessaire. En revanche si je change de browser, bah logique quoi.

Il y a le header, le payload, et la signature (secret) dans un token. Le payload est décodable en base64, il ne faut donc jamais mettre de secret dedans.

L’API détecte tout de suite si lui-même ou un autre veut altérer le token, les tentatives de hack sont donc facilement détectables



Notes questions réponses sur le module 7 :

Seul le serveur a la clé pour identifier un token

Dans le tutoriel, nous avons séparer le mécanisme de sessions en deux :

1. La route register/login simple, qui serialise les input obtenus.
2. Le module auths du repertoire utils, qui vérifie la légitimité du TOKEN.

Etapes/Moyens pour **sécuriser mon API** :

* Installer escape-html (cf mod 7 d.2.3)
* Sécuriser les password via bcrypt et le salt (si besoin importer le code du projet)

Autorisations via Token JWT : serveurs stateless (cf mod.8 a.3 ou importer projet)

Fonctions authorize & isAdmin dans le AUTHS du utils de l’api

Quelques explications sur le code de la fonction middleware **authorize** :

* Grâce à req.get('authorization'), on récupère le token qui a été envoyé par le client au sein de l'**authorization header** de la requête, c'est une string.
* La méthode jwt.verify(token, jwtSecret) vérifie tant la signature du token que le fait que le token n'a pas encore expiré. Elle utilise le secret **jwtSecret** qui doit être le même que celui pris en compte lors de la création du token.
* Elle charge toutes les données de l'utilisateur authentifié au sein de l'objet **request**, dans **user**. Cela est une bonne pratique, cela permet, pour toute la durée du traitement de cette requête, de mettre à disposition ces données à toutes les fonctions middleware.  
  💭 Mais pourquoi faire cela ? Imaginez que vous faites un appel à une base de données externes à chaque fois que vous souhaitez obtenir les informations d'un utilisateur... Cela est très consommateur en temps... Lorsque vous allez utiliser d'autres fonctions middleware comme **isAdmin**, vous n'avez plus besoin de faire appel à la base de données.

Utiliser isAdmin & authorize :

Une image contenant texte, capture d’écran, écran, noir

Description générée automatiquement

Ainsi, nous avons juste fait l'appel de deux fonctions middleware pour vérifier :

* **que l'utilisateur est authentifié** via **authorize** ; si tout est OK au niveau du token fournit par l'application cliente, **authorize** fait appel via **next()** à la prochaine fonction middleware. Dans ce cas-ci, c'est **isAdmin**. S'il y a un problème, **authorize** termine le traitement de la requête en envoyant un code d'erreur au client et les prochaines fonctions middleware (**isAdmin**, puis la fonction arrow) ne sont pas exécutées.
* **que l'utilisateur est admin** via **isAdmin** ; si tout est OK, que l'utilisateur authentifié est l'admin, **isAdmin** fait appel via **next()** à la prochaine fonction middleware. Dans ce cas-ci, c'est la fonction arrow qui appelle createOnePizza pour créer la pizza, la fonction traitant l'opération demandée. Si l'utilisateur authentifié n'est pas l'admin, alors **isAdmin** termine le traitement en envoyant un code d'erreur au client et la fonction traitant de l'opération de création n'est pas exécutée.

En gros : authorize vérifie qu’un user est connecté, et isAdmin verifie qu’il a les droits de performer cette opération

Ainsi ce genre de opération a besoin d’un autorize. Ceci marche pas :

Une image contenant texte

Description générée automatiquement

Et ceci marche :

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

# **Module 8 : Sessions sur le local storage/web storage + Déploiement**

Le local storage est comme un système de fichiers propre a lui (sur le navigateur). Ainsi la session d’un user peut persister même après un refresh.

* Pourquoi cela serait-il un probleme d’être considéré comme user dès que j’accède a mon site ?

Imaginons que je sois supprimée de la DB. Avec mes données persistant dans la session, je suis toujours connecté alors que je ne peux pas. Dans ce cas, ce n’est pas si grave car je ne peux pas faire de requêtes POST. Dans une SPA, c’est l’API qui vérifie la légitimité des envoyeurs de requêtes en checkant si le token est legit.

En bref, l’utilisateur non legit aura l’apparence d’un frontend d’utilisateur legit mais ne pourra quand même pas performer d’action non-autorisée. (ex : add pizza)

« *Remember me* » ?

Si je coche, je mets les données dans le local storage. Si non, je les mets dans le session storage.

Avec le cookie, la seule donnée qu’on sauvegarde le local storage est le username. Le token voyage dans le cookie.

La gestion du cookie se fait au niveau de l’application API

1. Web Storage
   1. SessionStorage
   2. LocalStorage

**🡺 localStorage.setItem(clé,valeur)** ou **sessionStorage.setItem(clé,valeur):** clé nom de l’user, valeur une String qui a été stringify depuis JSON

🡺 **localStorage.getItem(clé)** **ou sessionStorage.setItem(clé)** : donne valeur que je peux parse en JSON

🡺 **localStorage.removeItem(clé) ou sessionStorage.removeItem()** : fait ce que ça dit

🡺 **localStorage.clear()** ou **sessionStorage.clear()**

DANS UTILS/auths.js (cf mod.8 a.2.6) (Frontend)

Une image contenant texte

Description générée automatiquement

1. Cookies

Besoin du web storage

Suivre les étapes précisées dans le a.3.2) ( backend !)

Déploiement

En général en web on utilise le déploiement PaaS (platform as a service) => on va utiliser ça pour le projet (aws/azure/google/etc)

Déploiement pour le frontend => GitHub Pages

Pour l’API => Azure (dl Azure Tools sur VSC)