HELHa
Haute École Louvain en Hainaut

Travail de fin d'études

Les différentes facettes du framework Nuxt.js

Présenté par

Wambe Thomas

En vue de l'obtention du grade de

Bachelier en Informatique et Systèmes

finalité Informatique Industrielle

Année Académique 2020-2021

Parlement européen

EUROPEAN CONSERVATIVES AND REFORMISTS

TECHNIQUE

Remerciement

Je tiens à les remercier pour leurs conseils et leur aide précieuses :

Mesdames :

- Bandinu Louna
- Desurpalis Nicole
- Lechien Zazie

Messieurs :

- Jadoul Philippe
- Ledru Robinson
- Michaux Bertrand
- Saldic Dzafer
- Smolders Laurent
- Wambe Patrick

Ainsi que :

- Le groupe politique ECR pour m'avoir donné l'opportunité d'effectuer mon stage en son sein.
- L'ensemble du corps enseignant de l'HELHa de Charleroi m'ayant suivi de près ou de loin dans mon parcours.

Abstract

During my last year of Bachelor in informatics, I had the opportunity to participate to a 15-weeks internship in a company.
This company is the ECR group, one of the many political groups of the European Parliament.
The purpose of this internship was to develop an application to test the abilities of a new technology known as Nuxt.js.

The goal was reached and I was able to develop a website dealing with public data of ECR deputies.
In order to do this, I had to use known technical concepts learned in school, study unknown concepts and apply them in the 'real working environment'.

This final project will describe in detail all the steps needed to achieve a working application, which will serve as a foundation for further developments.

Finally, those 15 weeks have proven me that this kind of development project was well within my reach and that I was technically competent to deal with it.

I hope that reading this will be as interesting for you as it was for me writing it.

Table des matières

1.Introduction

Lors de ma dernière année de Bachelier en informatique et systèmes à finalité industrielle, j'ai eu l'occasion de réaliser un projet lors d'un stage en entreprise de 15 semaines. Cette entreprise est le groupe ECR, l'un des groupes politique du parlement européen.

Lors de ce stage, il m'a été demandé de préparer le terrain afin de moderniser la mise en place d'une infrastructure applicative pour la rendre disponible sous un navigateur web. Cette mise en place s'est effectuée sous une nouvelle technologie de rendu universel, Nuxt.js.

L'objectif final de cette application est de rendre disponible des données générales au sein de l'entreprise afin que chaque membre puisse les consulter librement.

Pour ce faire, j'ai dû effectuer des recherches sur différents concepts techniques, tel que le rendering ou le framework Nuxt.js, que j'expliquerai dans ce travail.
Ensuite, il a fallu mettre en place ces différents concepts afin de répondre aux objectifs de ce projet.

Enfin, différents tests ont été nécessaires afin de valider le bon fonctionnement de l'application dans les différents cas de figure possibles lors de son utilisation.

De nouvelles fonctionnalités seront mises en place à l'avenir. Celles-ci sont, de manière non exhaustive, la localisation de l'emplacement des lieux de travail des députés (bureaux alternativement à Strasbourg et à Bruxelles), le remboursement des différents frais liés aux activités du personnel du groupe ECR, une gestion d'utilisateurs afin de permettre l'accès à certaines informations, etc…

Ces développements futurs ne seront pas abordés dans ce travail, car ils font toujours l'objet de négociations au sein du groupe.

Avant d'aborder le projet en lui-même, ce stage m'a permis de me familiariser avec le monde de l'entreprise et la gestion d'un projet de développement en Informatique dans un cadre professionnel.

La situation sanitaire actuelle n'a pas empêché de mener le projet à son terme, et ce grâce aux nombreux échanges avec le maître de stage. La disponibilité et les compétences techniques de ce dernier m'ont permis d'acquérir de nouvelles connaissances dans le cadre de ma formation.

Je souhaite que la lecture de ce travail fasse apparaître tout l'enthousiasme que j'ai mis pour le réaliser.

2.Présentation de l'entreprise

2.1.Le Parlement européen

Le Parlement européen (PE) est l'une des sept institutions de l'Union Européenne (UE). Ces institutions sont réparties en différentes catégories : Les institutions politiques, les institutions économiques et l'institution judiciaire.

Les institutions politiques sont composées du Parlement européen, du Conseil européen, du Conseil des ministres et de la Commission européenne. Les quatre institutions susnommées détiennent les pouvoirs exécutif et législatif de l'UE.

Les institutions économiques regroupent les institutions de la Banque centrale européenne et de la Cour des comptes européenne. La première contrôle la politique monétaire de l'ensemble des 19 états de l'Eurozone[1] et ainsi, elle maintient la stabilité des prix sur l'ensemble de ces états. La seconde s'assure de la bonne utilisation du budget de l'UE.

L'institution judiciaire est uniquement composée de la Cour de justice de l'UE. Cette institution a pour but d'assurer que le droit communautaire est appliqué de la même façon dans tous les états et d'arrêter les discordes entres les institutions et les États.

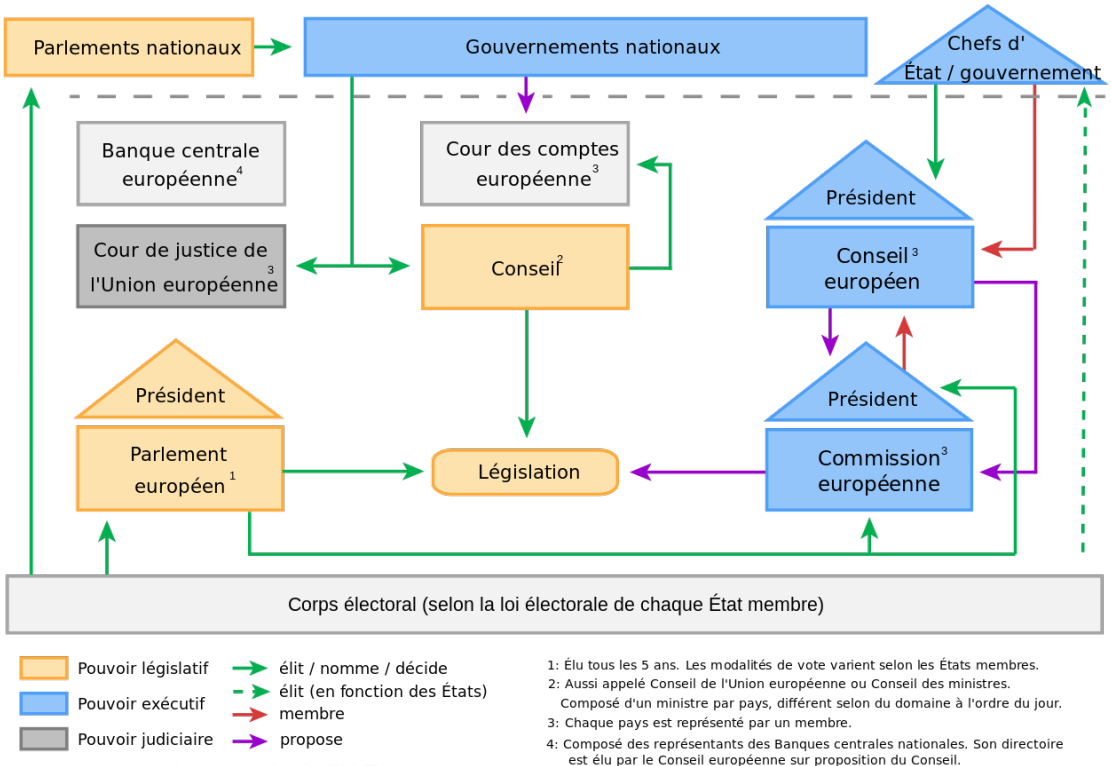

Figure 1 : Organisation au sein de l'U.E.

Le PE est composé de 705 députés élus dans les 27 pays membres de l'UE élargie. Ce parlement est élu au suffrage universel direct pour une période de 5 ans.

[1] Eurozone : Les états membres qui ont adoptés l'euro comme monnaie officielle.

Celui-ci décide également la législation de l'Union, y compris sur le budget pluriannuel, avec le Conseil de l'Union européenne. Les autres institutions, dont la Commission européenne, rendent des comptes au Parlement.

Le PE élit le président de la Commission européenne et joue un rôle clé dans l'examen des Commissaires-désignés, en les auditionnant individuellement. Le collège des commissaires - c'est à dire l'ensemble des vingt-sept commissaires réunis - doit ensuite recueillir le vote de consentement du Parlement.

Les députés au Parlement européen sont élus dans les états membres de l'UE et représentent environ 447 millions d'habitants. Au fil des années et des modifications apportées aux traités européens, le Parlement a acquis des compétences législatives et budgétaires considérables.

Le PE compte 7 groupes politiques répartis dans toute l'UE. Les députés ne peuvent appartenir qu'à un seul groupe politique ; Il suffit de 23 députés pour former un groupe politique qui doit comprendre des membres représentant au moins un quart des États membres. Les députés au Parlement européen ne sont pas organisés par nationalité, mais en fonction de leurs affinités politiques.

Ces groupes sont :
-Groupe du Parti Populaire Européen (PPE).
-Groupe de l'Alliance Progressiste des Socialistes et Démocrates au Parlement européen (S&D).
-Renew Europe Group (Renew).
-Groupe des Verts/Alliance Libre Européenne (Verts/ALE).
-Groupe « Identité et Démocratie » (ID).
-Groupe des Conservateurs et Réformistes Européens (ECR).
-Groupe de la Gauche au Parlement européen (La Gauche).

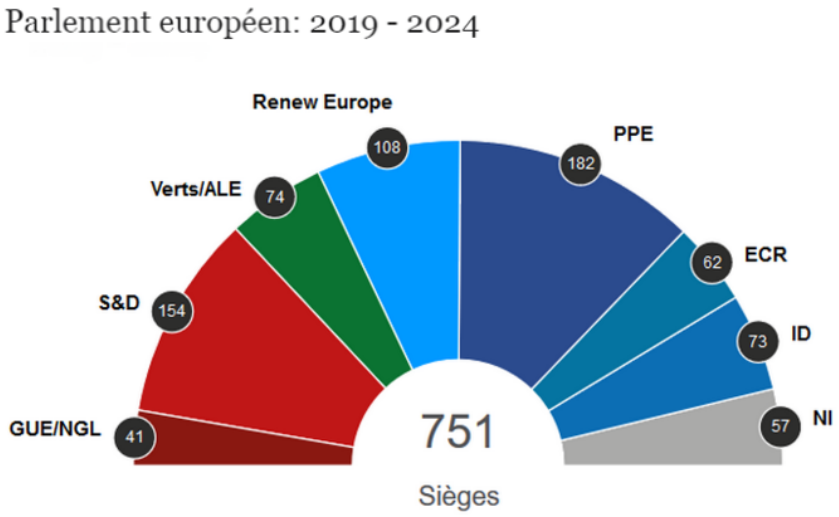

Figure 2 : Sièges occupés par groupes politique au P.E.

En plus des 7 groupes politiques cités ci-dessus, on retrouve les non-inscrits, notés NI. Ces députés politiques sont présents au Parlement européen mais ne font pas partie d'un quelconque groupe politique.

2.2.Le groupe ECR

Le groupe Politique ECR, où le stage s'est déroulé, retiens donc ici notre attention.

Le groupe des Conservateurs et Réformistes Européens (ECR ou CRE en français) est un groupe politique européen qui regroupe l'ensemble des partis de droite et de la droite nationaliste en Europe. Les membres sont pour le « libéralisme économique ». Ils disposent d'un groupe au Parlement européen mais également d'un groupe à l'Assemblée parlementaire du Conseil de l'Europe.

Ce groupe fut fondé le 30 mai 2009 par le Parti conservateur britannique ayant quitté l'ex-groupe du Parti populaire européen et des Démocrates européens (PPE-DE) devenus le groupe du PPE. Néanmoins, la création du groupe fut officielle le 22 juin 2009 et est annoncée à Londres et à Prague simultanément par les conservateurs et le Parti démocratique civique (ODS). Ils comptaient 54 députés (26 conservateurs et unionistes britanniques, 15 Polonais de Droit et Justice, 9 membres de l'ODS, un Belge, un Hongrois, un Letton et un Néerlandais). Il est officialisé lors de la première représentation au Parlement européen du 14 juillet 2009.

Composition du groupe ECR par pays lors de la 7ième législature (2009-2014).

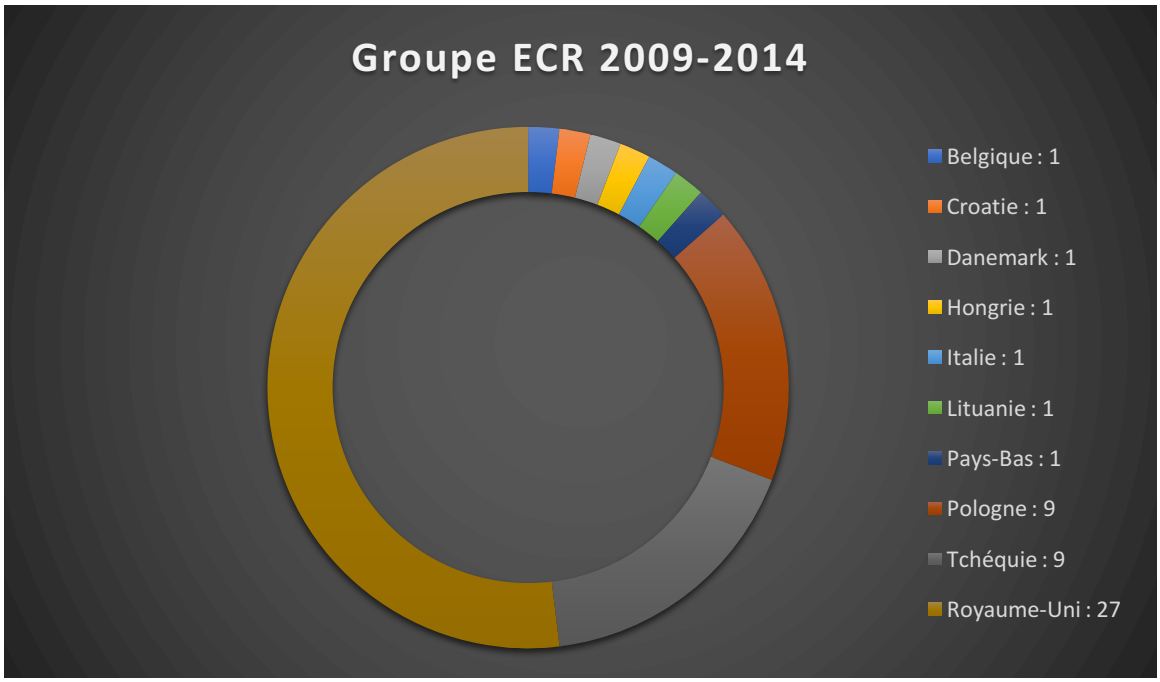

Figure 3 : Composition du groupe ECR par pays (2009-2014)

À la présidence du groupe on retrouve Michał Kamiński, qui exerça ce rôle de juillet 2009 à mars 2011. Ensuite, Jan Zahradil assura la présidence jusqu'en décembre 2011 pour céder sa place à Martin Callanan, en poste jusqu'à la fin de la législature : le 30 juin 2014.

À ce jour, le groupe est présidé par deux personnes co-présidentes qui sont : Ryszard Legutko et Raffaele Fitto. Le groupe est constitué, pour la 9ème législature (2019-2024), de 62 sièges à la chambre du PE.

En voici sa composition classifiée par pays :

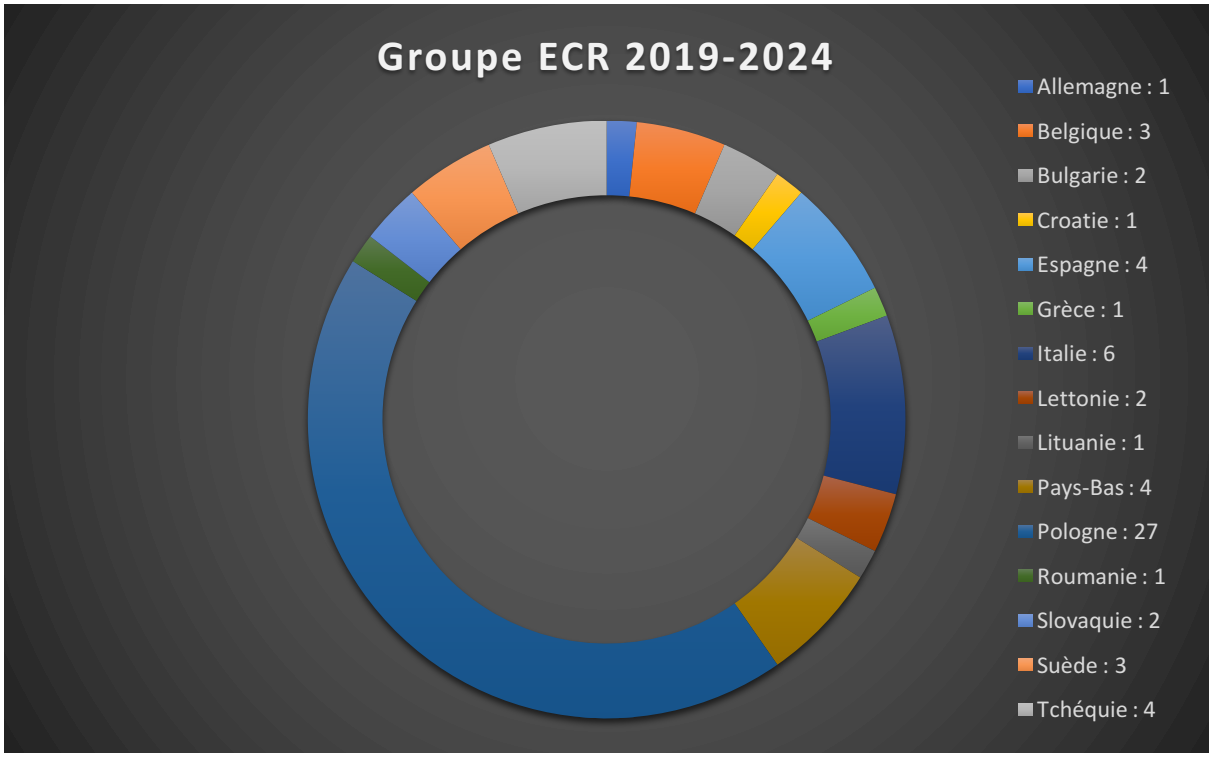

Figure 4 : Composition du groupe ECR par pays (2019-2024)

3.Présentation du stage

3.1 Introduction

Le groupe ECR met à la disposition de ses membres un site internet permettant de connaître les activités de ceux-ci.

En parallèle, un système basé sur des documents papiers existe, pour tout ce qui concerne, en autre, les remboursements et autres frais générés.

Le groupe souhaiterait disposer d'un outil informatisé permettant de rassembler ces différents points dans une application qui pourrait évoluer, grâce à de nouvelles fonctionnalités, dans le futur.

C'est là l'objectif principal de mon stage ; vérifier la faisabilité de cette demande, en tenant compte des impositions du service informatique du groupe, à savoir, utiliser la technologie Nuxt.js afin de jeter les bases d'une application pouvant servir de POC (proof of concept) pour les différents développements à venir.

Pour cela, certains concepts techniques sont nécessaires.

3.2.Différents concepts techniques abordés lors du stage

Lors de ce stage, plusieurs concepts techniques ont dû être utilisés ou simplement mentionnés. On y retrouve le protocole HTTP, la Single Page Application, le SEO, le DOM et l'Ajax expliqués ci-dessous.

3.2.1.HTTP

1. Introduction

En 1989 Tim Berners-Lee, travaillant alors au CERN en suisse, a rédigé une proposition visant à créer un système hypertexte sur internet. La mise en œuvre de ce système hypertexte en 1990, appelé plus tard World Wide Web, est construit sur les protocoles TCP et IP existants.

Il est fait de 4 composants :

- Un format textuel pour représenter des documents hypertextes : le HTML.
- Un client pour afficher ces documents : le premier navigateur web (appelé aussi WorldWideWeb).
- Un serveur pour donner accès aux documents : une première version de httpd.
- Un protocole simple pour échanger ces documents : le protocole HTTP.

Ce protocole est un protocole de communication client-serveur. Il utilise par défaut le port 80. Il existe une variante sécurisée, HTTPS qui utilise le port 443. Le protocole HTTP permet la communication entre un client et un serveur, lors d'un accès sur un site internet par exemple.

Cet accès va se faire comme suit :

1) L'utilisateur va saisir, dans la barre d'adresse de son navigateur, l'URL du site auquel il veut accéder : http://www.example.com. Après avoir résolu cette adresse et obtenu d'un serveur DNS l'IP du serveur distant lui correspondant, la requête est envoyée à celui-ci.

2) La requête HTTP est envoyée par le navigateur au serveur Web qui héberge le contenu du site. Cette requête est notamment composée d'une méthode, de l'URL et de la version de HTTP utilisée. Elle a pour but de demander au serveur s'il peut envoyer le fichier HTML qui compose la page.

3) Le serveur reçoit la requête, la traite et prépare une réponse qu'il renvoie au navigateur.

4) Le navigateur traite la réponse et affiche la page à l'écran.

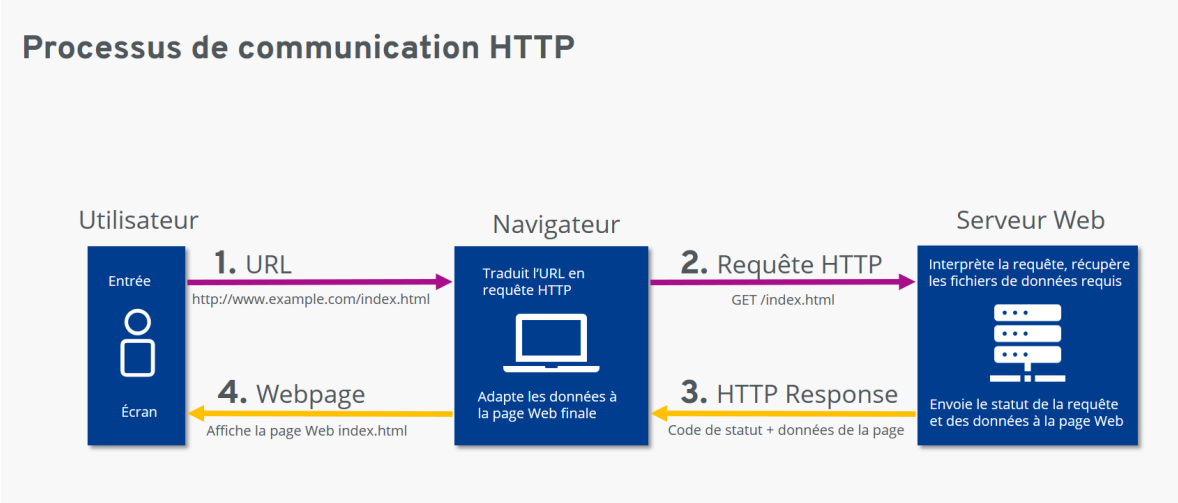

Figure 5 : Processus de communication du protocole HTTP

Comme dit aux points 1 et 2, le protocole HTTP est fait de requêtes. Celles-ci respecteront la structure suivante :

2. Structure d'une requête

2.1. « Start Line »

Elles commenceront par ce qu'on appelle une « Start Line » (ligne de départ). Cette « Start Line » sera composée d'une méthode, d'une URL et d'une version.

2.1.1. Méthode

Ici la méthode décrit l'action à effectuer.
Ces méthodes sont nombreuses, mais les plus utilisées sont les méthodes GET, HEAD et POST.

2.1.2. URL

L'URL est la cible de la requête. Celle-ci peut être écrite de façons différentes.

- Un chemin absolu, également appelé « forme d'origine ». Cette forme d'URL est utilisée avec les méthodes GET, POST, et HEAD. Celle-ci commencera par un « / » et peut également n'être constituée que de ce caractère pour cibler la racine du site. Quelques exemples :

  -GET /background.png HTTP/1.1
  -POST / HTTP/1.0
  -HEAD /test.html ?query=test HTTP/1.1

- Une URL complète, également appelé « forme absolue ». Celle-ci est principalement utilisée avec la méthode GET.
  Cette URL sera relative au serveur et sera formée de cette façon :

  -GET http://www.example.com. HTTP/1.1

- Une URL comprenant un composant d'autorité, composé d'un nom de domaine et éventuellement du port, précédé du caractère « : ». Elle n'est utilisée que par la méthode CONNECT lors de la configuration d'un tunnel HTTP (cette méthode ne sera pas reprise dans ce travail) ... Elle se présente comme suit :

  -CONNECT example.com:80 HTTP/1.1

- Une URL redirigeant sur l'entièreté du serveur. Cette URL se compose uniquement d'une astérisque « * » et est utilisée avec la méthode OPTIONS (cette méthode ne sera pas reprise dans ce travail).
  En voici un exemple :

  -OPTIONS * HTTP/1.1

2.1.3. Version

La version HTTP définit la structure du message. Elle agit comme un indicateur de la version attendue à utiliser pour la réponse.

Ensuite, viennent les « Headers » (en-têtes) qui complètent la requête.

2.2. « Headers »

Les « Headers » HTTP d'une requête suivent tous la même structure. Une chaîne insensible à la casse[2] suivie d'un séparateur, deux points, « : » et d'une valeur dont la structure dépend du type de « Header ». Cette structure n'est faite que sur une seule ligne ce qui peut être, parfois, assez long.

Un exemple de « Headers » :

-Content-Length : 9000
-Content-Type: text/html
-Connection: keep-alive

Après les « Headers » vient le corps de la requête également appelé « Body ».

[2] Insensible à la casse : L'action sera la même que ce soit écrit en majuscules ou en minuscule

2.3. « Body »

C'est la dernière partie de la requête bien qu'elle puisse être optionnelle. Effectivement toutes les méthodes n'ont pas besoin de corps. Celui-ci n'est utile que pour les requêtes servant à envoyer des données au serveur afin de les mettre à jour.

3. Structure d'une réponse

Une fois la requête reçue, le serveur envoie une réponse au navigateur.
Cette réponse a, à l'instar de la requête, une structure.

3.1. « Status Line »

Cette réponse commence par une « Status Line » (ligne d'état).
La « Status Line » contient les informations suivantes : la version d'HTTP, un « Status Code » (code de statuts) et un texte d'état.

3.1.2. Version

La version d'HTTP est en relation directe avec la requête. Effectivement, si la requête envoie une certaine version d'HTTP, la réponse aura la même version.

3.1.3. « Status Code »

Le « Status Code » indique l'état de la réponse. Nous reviendrons plus en détails sur ces codes de statut.

3.1.4. « Status text »

Le texte d'état est une brève description textuelle purement informative du « Status Code » pour qu'un humain puisse comprendre le message HTTP plus facilement.

Exemples de « Status Line »:

-HTTP/1.1 404 NOT FOUND
-HTTP/1.0 200 OK

3.2. « Headers »

Ensuite, comme pour la requête, on retrouve le « Header » de la réponse.
Cet en-tête a la même structure que celle de la requête ; c'est-à-dire une chaîne insensible à la casse suivie d'un séparateur, deux points, « : » et d'une valeur. Ceci s'appelle une paire de clé/valeur. Au même titre que l'en-tête de la requête, la structure de la valeur dépend du type de « Header ». Celui-ci peut, encore une fois, être assez long car toujours sur la même ligne.

3.3. « Body »

Comme dans une requête, le dernier élément de la réponse est son corps.
Toutes les réponses n'ont pas nécessairement un corps. Effectivement, les réponses, avec un « Status Code » qui répond suffisamment à la demande sans avoir besoin de contenu correspondant, n'en ont pas. Par exemple les « Status Code » 204 ne nécessitent pas de corps de réponse.

4. Méthodes plus en détails.

4.1. Méthode GET

La méthode GET est la méthode la plus courante pour récupérer une ressource statique ou dynamique sur un serveur. Ces requêtes avec la méthode GET ne peuvent, théoriquement, servir qu'à récupérer des données mais dans la pratique certains développeurs pourraient l'utiliser pour envoyer des données au serveur pour aller, par exemple, modifier ou écrire des informations dans une base de données.

La méthode GET peut posséder des « Headers » mais ne possède pas de corps dans la requête.
Cependant, la réponse peut également posséder des « Headers » mais elle, au contraire de la requête, possède généralement un corps de réponse.

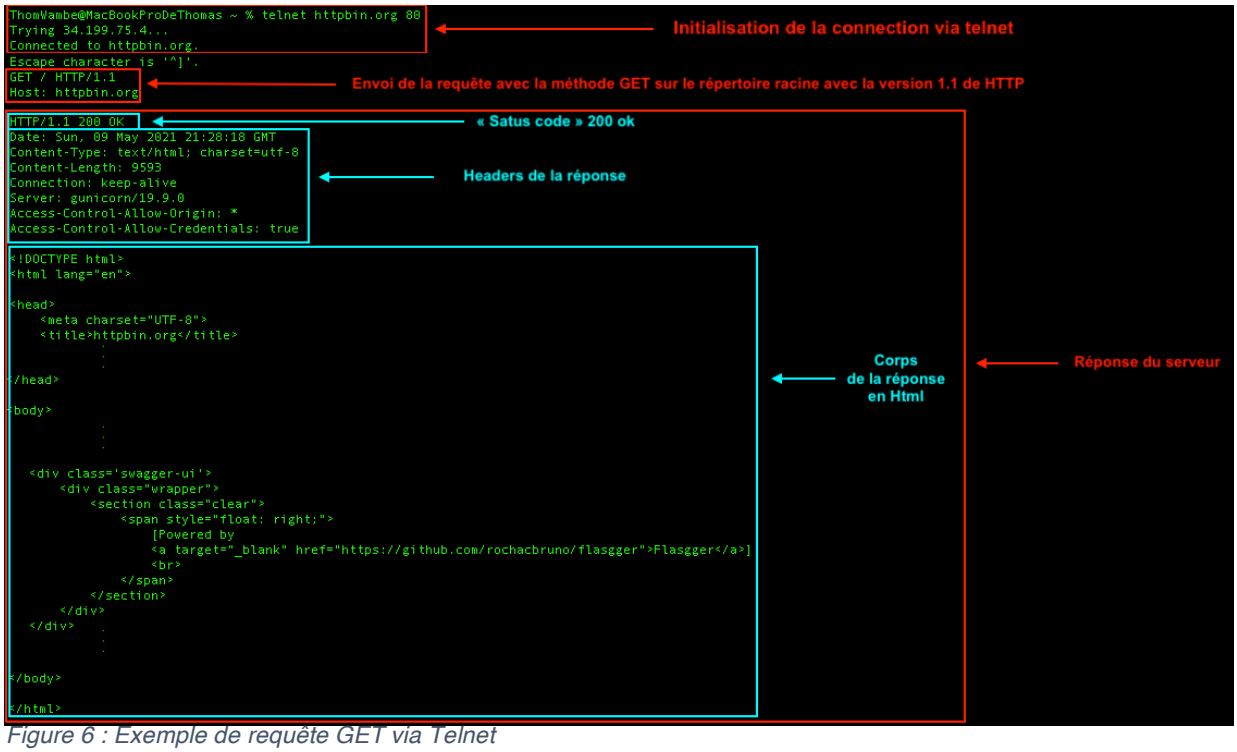

Figure 6 : Exemple de requête GET via Telnet

4.2. Méthode HEAD

La méthode HEAD est surtout utilisée lors de tests ou à des fins informatives. Cette méthode permet, entre autres, à l'utilisateur de savoir si la page à afficher n'est pas trop volumineuse avant de la télécharger entièrement. Si la page semble trop grande pour l'utilisateur, il pourra éviter de faire une requête GET pour, par exemple, économiser de la bande passante.

Cette méthode peut posséder des « Headers » mais ne possède pas de corps dans la requête.

La réponse peut également posséder des « Headers » mais ne possède généralement pas de corps. Si, malgré tout, corps il y a dans la réponse, celui-ci sera rogné et ignoré.

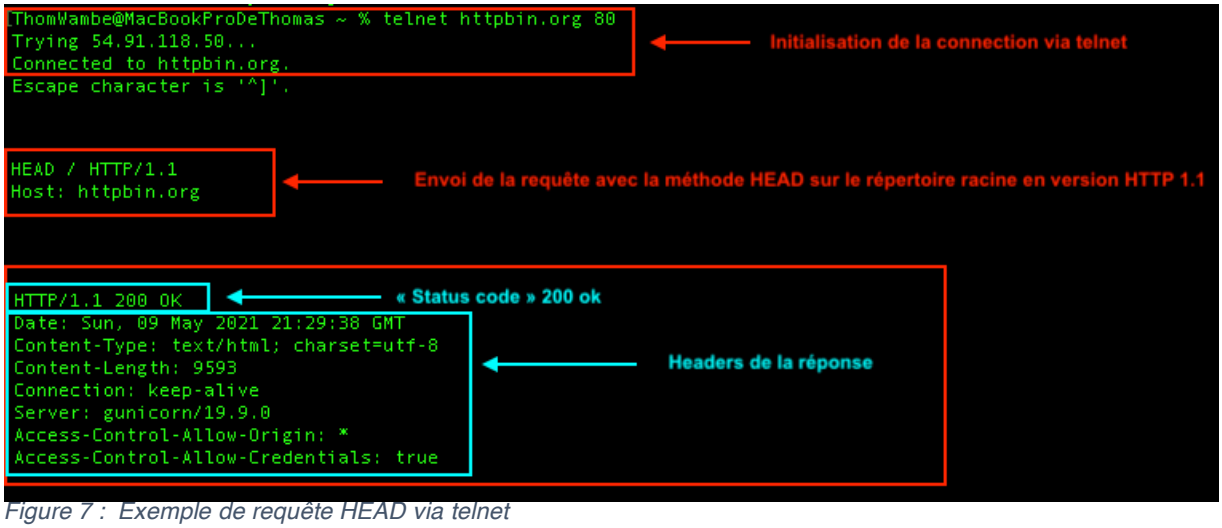

Figure 7 : Exemple de requête HEAD via telnet

4.3. Méthode POST

La méthode POST, a contrario des autres méthodes décrites, sert à envoyer des données au serveur.

Cette méthode peut posséder des « Headers » et possède un corps dans la requête.
La réponse peut également posséder des « Headers » et possède généralement un corps.

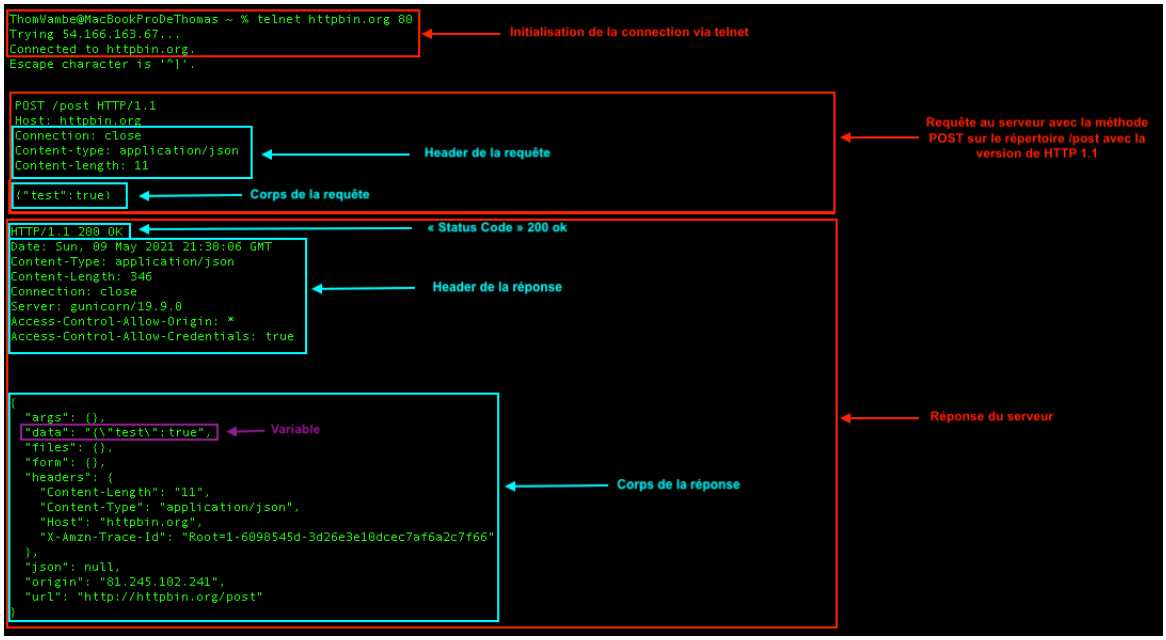

Figure 8: Exemple de requête POST via Telnet

5. Précisions des « Status Codes »

Les codes de statuts HTTP font partie des réponses fournies par les serveurs lors de chaque requête effectuée sous n'importe quelle méthode.

Ces codes, composés de 3 chiffres, permettent au navigateur de savoir si la requête a bien été formulée, si celle-ci a abouti, etc. Ces « Status Codes » sont divisés en plusieurs classes selon le chiffre se situant en première position :

- La classe 1XX : ce sont les codes d'information. Ces codes commencent par le chiffre 1, ce qui indique au client que la requête est en train d'être effectuée. Cette classe regroupe donc tous les codes des requêtes étant en cours de traitement et ou d'envoi.

- La classe 2XX : ce sont les codes de succès. Ces codes commencent par le chiffre 2 et indiquent l'aboutissement de la requête. Cette requête a donc été reçue par le serveur, a été comprise et acceptée. Ces codes sont envoyés en même temps que les informations des pages web demandées.

- La classe 3XX : ce sont les codes de redirection. Ces codes commencent par le chiffre 3. Ceux-ci stipulent au client que le serveur a bien reçu la requête mais que le client doit encore effectuer une action supplémentaire pour que le traitement soit conduit à sa résolution finale. Ces codes apparaissent lors de cas de redirections.

- La classe 4XX : ce sont les codes d'erreur du client. Ces codes commencent par le chiffre 4 et renvoient à une erreur commise par le client. Ils signifient que le serveur a bien reçu la requête mais ne peut pas l'exécuter. Dans la majorité des cas, ces codes sont dus à une erreur de syntaxe. Les développeurs peuvent avoir mis en place du contenu, à afficher, sur le serveur lorsque ce genre de code est répondu.

- La classe 5XX : ce sont les codes d'erreur du serveur. Ces codes commencent par le chiffre 5. Ils font référence à une erreur commise par le serveur. Ils indiquent que la requête est complètement ou provisoirement impossible à effectuer. Une page HTML est généralement affichée.

Certains codes notables sont : 200, 304, 400, 403, 404, 500 et 503.

- Le code 200 : généralement suivi de « Ok » indique au client que la requête a bien été traitée avec succès. La réponse dépendra, bien évidemment, de la méthode de requête utilisée.

- Le code 304 : ce code, suivit de « Moved Permanently », indique que la ressource a été définitivement déplacée à L'URL contenue dans le Headers.

- Le code 400 : suivi de « Bad Request » est une réponse à une requête à syntaxe erronée. Ce code survient fréquemment lors de test sur un serveur.

- Le code 403 : suivi de « Forbidden » indique que le serveur a compris la requête mais refuse de l'exécuter. Ce code d'erreur est souvent retourné lorsque l'utilisateur ne peut pas accéder à la ressource demandée. S'authentifier ne servira à rien dans ce cas, la ressource ne pourra pas être accessible.

- Le code 404 : toujours suivi de « Not Found » est une réponse récurrente qu'on rencontre lorsque la ressource demandée n'est pas trouvée. Généralement, une page dédiée aux erreurs 404 est située sur un site internet. Cette erreur peut être facilement générée en se trompant ou en ajoutant des caractères dans une URL.

- Le code 500 : ce code est retourné en réponse à une requête lorsqu'une erreur interne au serveur est présente. Si le serveur ne peut traiter la requête, ce code est automatiquement affiché. Généralement seul l'administrateur du site pourra régler ce problème.

- Le code 503 : ce code est envoyé en réponse à une requête quand le service requis est temporairement ou indéfiniment indisponible ou en maintenance. Cela peut aussi se produire lorsque le serveur est surchargé. L'utilisateur doit, en général, se dire qu'un administrateur travaille, au moment même, sur le problème et que le service sera de nouveau disponible sous peu.

3.2.2.Single Page Application

Une « single page application (SPA) » ou une application web mono-page est une application web, comme son nom l'indique, accessible via une et une seule page web. Le but est d'éviter le chargement d'une nouvelle page à chaque action demandée.

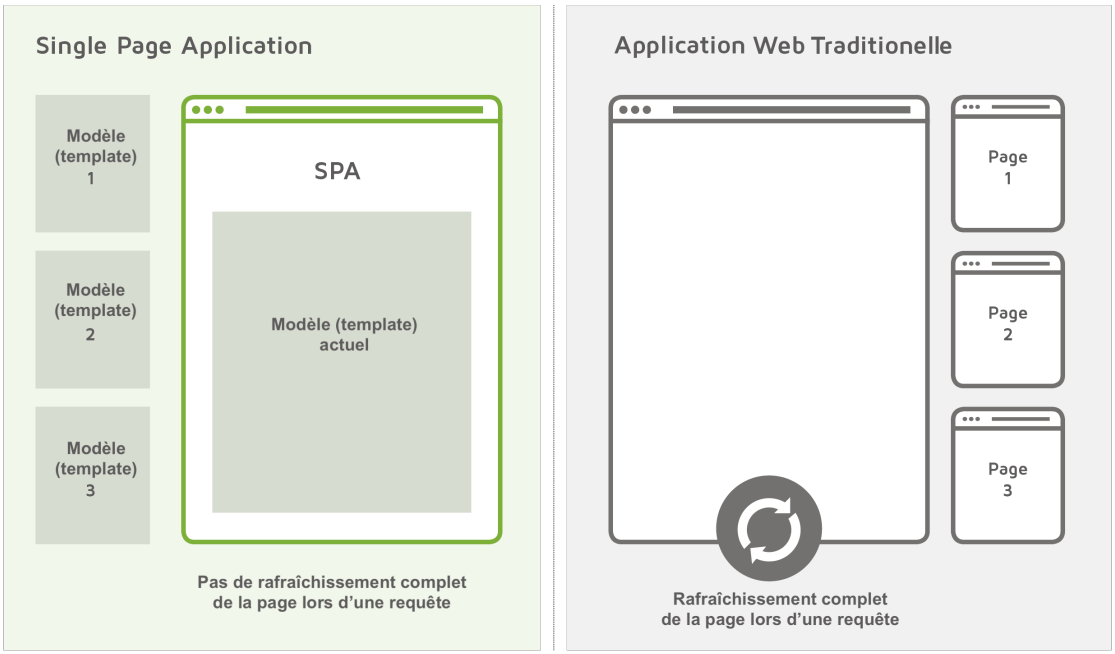

Figure 9 : rafraîchissement d'une Single Page Application.

Cela fluidifie ainsi l'expérience utilisateur. Pour ce faire, l'ensemble des éléments de l'application est chargé en une fois. Lors de l'utilisation d'une SPA, le navigateur devient, en quelques sorte, un ordinateur qui exécuterait localement un programme téléchargé sur internet. Cette façon de faire des applications web apporte quelques avantages, tels que :

- La vitesse de chargement : le plus grand avantage du SPA est sa rapidité, car une fois l'application chargée, la quantité de données transitant entre le client et le serveur est très faible. Les temps de chargement en sont donc réduits au minimum.

- Le développement mieux organisé : lors du développement d'une application à page unique, le code côté serveur est réutilisé et est effectivement découplé de l'interface utilisateur frontale. Cela signifie que les équipes « Back-End » et « Front-End » peuvent se concentrer sur leur travail respectif. Elles doivent néanmoins communiquer entre elles pour s'assurer que les bonnes informations seront bien envoyées et reçues.

Par contre, cette façon de coder des applications web a un défaut majeur : la mauvaise gestion du SEO (expliqué ultérieurement).

3.2.3.Document Object Model

Le Document Object Model ou le DOM est la représentation objet des données qui composent la structure et le contenu d'une page web. Le DOM est une représentation du fichier HTML source. Il le transforme, en quelque sorte, en un modèle utilisable par d'autres programmes. Ce modèle utilisable a une structure spécifique, appelée « Node Tree » (arborescence en nœud). Sa représentation se fait en forme d'arbre où à chaque nœud du document HTML, une nouvelle ramification se crée. Le premier élément est la balise « <html> », ce qu'on pourrait associer à la racine de l'arbre. Chaque élément ou balise se situant dans ce document fera partie des branches de cet arbre.

Par exemple, pour le code suivant :

```
<html>
    <head>
        <title>Ma page HTML</title>
        <meta charset="utf-8">
    </head>
    <body>
        <h1>Bonjour le monde!</h1>
        <p>Comment allez vous?</p>
    </body>
</html>
```

Figure 10 : Exemple de code HTML simple

On obtiendra ce genre de DOM :

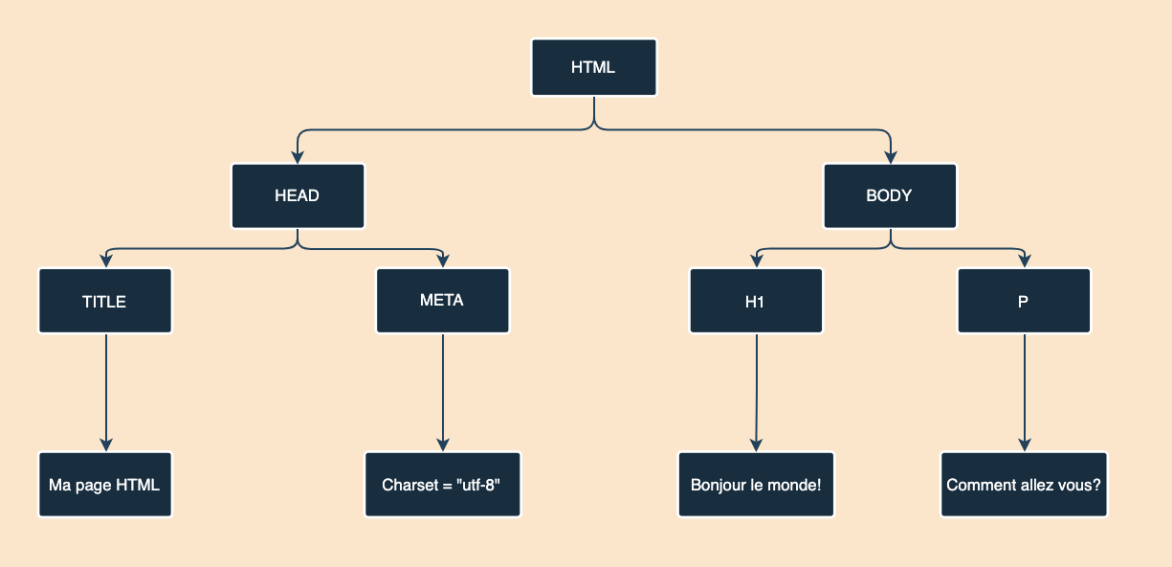

Figure 11 : Exemple de ce que peut donner un DOM

Le DOM définit également la façon dont la structure du document HTML peut être modifiée par les programmes ou scripts, en terme de style et de contenu.

Le JavaScript s'exécute en partie sur le DOM pour en modifier son style et son contenu pour afficher les différents éléments voulus à l'écran.

Pour généraliser, le DOM relie les pages HTML aux scripts ou langages de programmation. Le DOM est donc ce que le navigateur utilise pour afficher la page à l'écran. Le DOM sert ainsi de liant entre le HTML et les scripts pour ne faire plus qu'un seul fichier à afficher.

3.2.4.SEO

Le Search Engine Optimization (SEO), ou optimisation pour les moteurs de recherche, est l'ensemble des techniques visant à améliorer le positionnement d'une page, d'un site ou d'une application web dans la page des résultats d'un moteur de recherche. Ce positionnement est considéré comme bon lorsque le site est classé dans la première page des résultats de recherches faites grâce à des mots clés correspondant à sa thématique.

Le SEO, contrairement au SEA (Search Engine Advertising), est gratuit et est basé sur la bonne optimisation du site. Cette optimisation peut être technique, grâce à l'indentation, au contenu, à sa forme (meta tags) ou stratégique, basé sur les clients et leurs besoins.

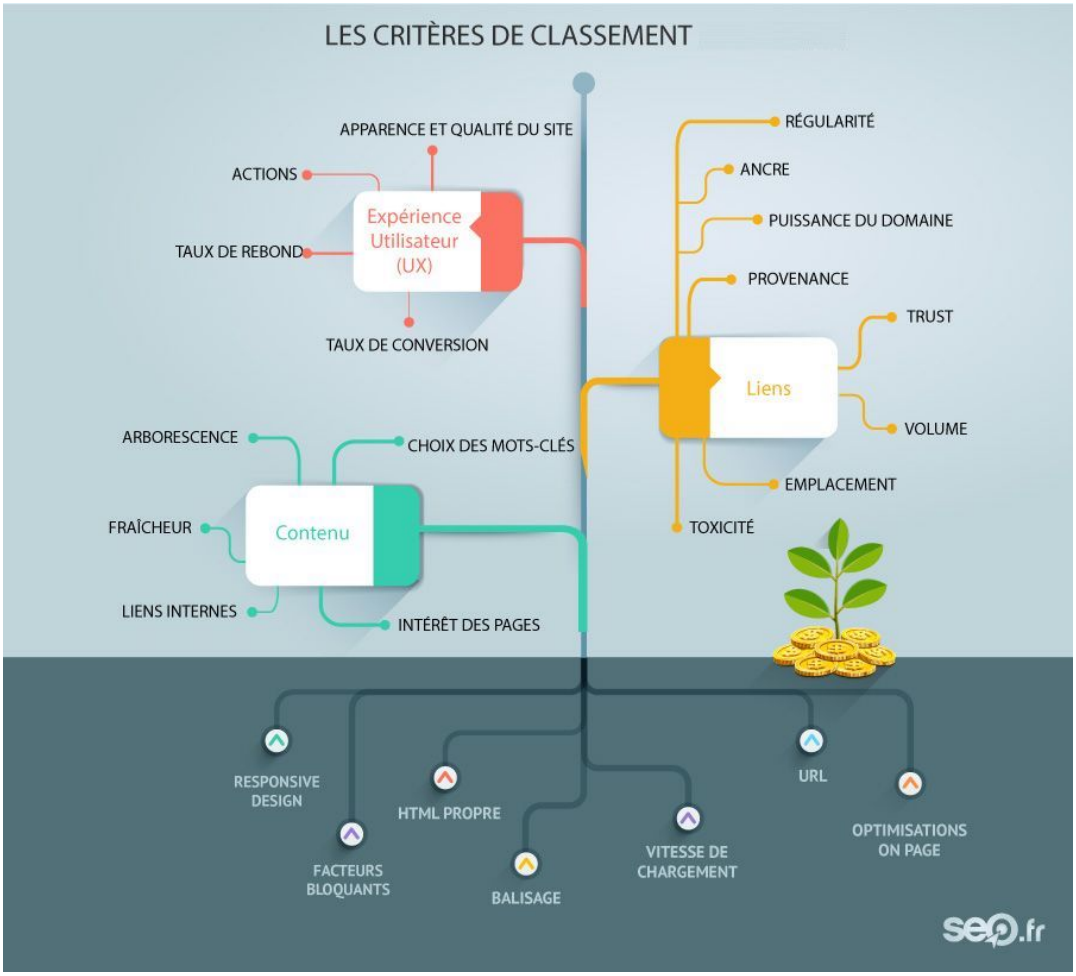

Figure 12 : Critères de classement SEO.

« Le SEO est l'art et la technique de persuader les moteurs de recherche comme Google, Bing, et Yahoo, de recommander votre contenu à leurs utilisateurs comme la meilleure solution à leur problème. »[3]

La plupart du temps, les développeurs se concentrent sur le référencement sur le moteur de recherche Google. En effet ce moteur de recherche est, de loin, le plus utilisé en Europe.

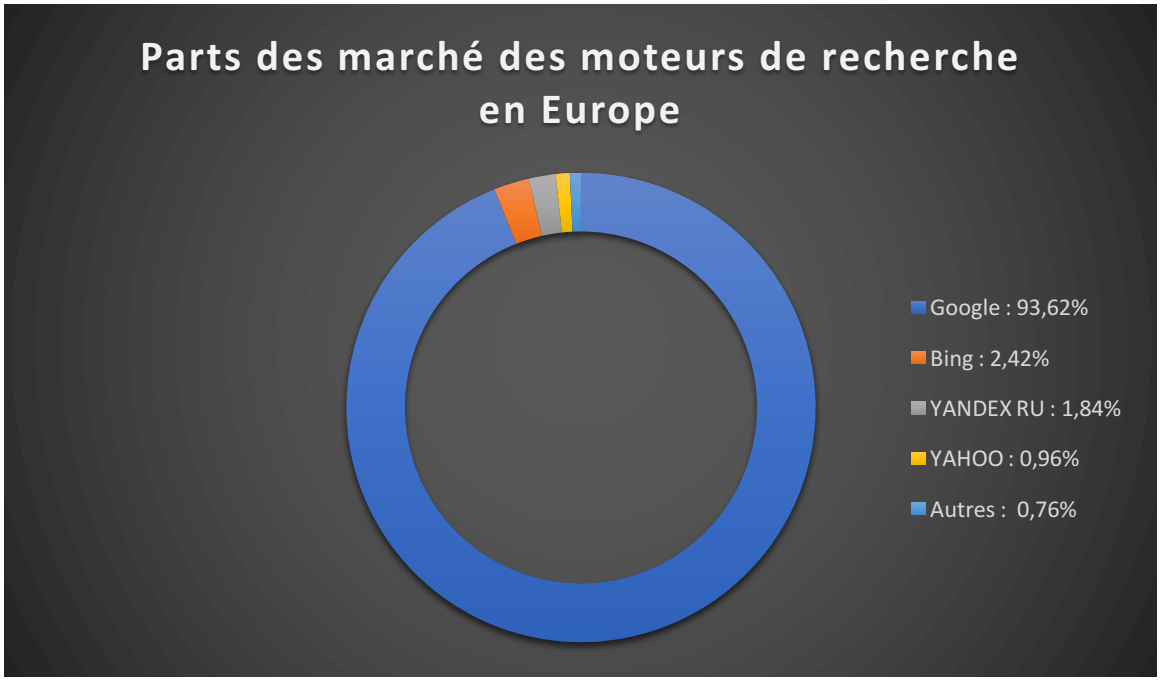

Figure 13: Parts des marchés des moteurs de recherche en Europe.

3.2.5.Ajax

L'Asynchronous JavaScript and XML (Javascript et XML asynchrones), dit Ajax, est une méthode utilisant différentes technologies des navigateurs web qui permet d'effectuer des requêtes aux serveurs web depuis le code JavaScript. Cette méthode permet de mettre en place des applications web et des sites web dynamiques interactifs. XML présent dans l'acronyme d'Ajax était historiquement le format utilisé pour échanger des données entre le navigateur et le serveur web. De nos jours le JSON (JavaScript Object Notation) lui est, le plus souvent, préféré dû à sa facilité d'interprétation par le moteur JavaScript.

La méthode classique de communication entre un serveur et un navigateur d'une application web standard se fait comme suit :
Lors de chaque action effectuée par l'utilisateur, le browser (Navigateur internet) envoie au serveur une requête HTTP, vue précédemment, contenant un lien vers une page web.
Le serveur va alors effectuer des « calculs » et envoyer le résultat, au navigateur, sous forme d'une page web.

Une fois reçue, celui-ci l'affichera.

[3] Source : https://fr.semrush.com/blog/definition-seo-guide-2020-debutants/?kw=&cmp=FR_SRCH_DSA_Blog_Core_BU_FR&label=dsa_pagefeed&Network=g&Device=c&utm_content=486542000146&kwid=aud-296306606820:dsa-1100351999444&cmpid=11849486850&agpid=113156852777&BU=Cor

Chaque manipulation faite par l'utilisateur entraînera cette séquence d'actions.
Ce qui engendre une perte de temps lors des rafraîchissements intempestifs à chaque action, contrairement à l'Ajax qui modifie la façon dont ce dialogue se déroule.

En effet, lorsque l'utilisateur fait une action, un programme écrit en JavaScript, présent sur la page web, est exécuté par le navigateur. Ce programme envoie, en arrière-plan, les requêtes au serveur web, puis modifie le contenu de la page actuellement affichée par le browser en fonction du résultat reçu du serveur. Ce procédé évite la transmission et l'affichage, c'est-à-dire, le rafraîchissement de la page au complet.

Cette méthode nécessite de programmer, en JavaScript, les échanges entre le navigateur web et le serveur web. Il est nécessaire, également, de programmer les modifications à effectuer dans la page web lorsque les réponses sont reçues.

Ces dialogues sont, comme le nom Ajax l'indique, fait de manière asynchrone. Cela veut dire que le navigateur continue d'exécuter le programme JavaScript lorsque la requête est effectuée. Celui-ci n'attend donc pas la réponse du serveur et l'utilisateur peut ainsi continuer à effectuer des manipulations pendant ce temps. Une fois la réponse reçue, celle-ci sera traitée par le gestionnaire d'événement défini lors de l'appel à cette fonction.

3.3.Le rendering

3.3.1.Introduction

Dans le monde du développement web, on différencie 2 types de rendering.

Le premier fait référence au « Rendering Engine », que l'on peut traduire en français par « Moteur de rendu » du navigateur.
Le moteur de rendu est, en quelque sorte, le noyau d'un navigateur internet, par analogie avec le moteur des véhicules.

Effectivement, le moteur de rendu a une fonction très importante : il permet d'afficher ce qu'on voit à l'écran. Il communique avec la couche réseau du navigateur pour récupérer le code HTML et d'autres éléments transmis, tels que des fichiers JavaScript depuis un serveur web distant.

Une fois que tous ces éléments sont présents dans le moteur de rendu, il analyse les fichiers HTML et crée le DOM grâce à ceux-ci et aux fichiers JavaScript reçus. Il construit ensuite, à l'aide des attributs CSS et du DOM, l'arborescence de rendu. Ensuite, il commence le processus de mise en page en parcourant de manière récursive les éléments HTML de l'arborescence et détermine où ceux-ci doivent être placés. Pour finir, il affiche chaque branche de l'arborescence de rendu à l'écran, en communiquant avec l'interface du système d'exploitation qui, contient des conceptions et des styles indiquant à quoi doivent ressembler les éléments de l'interface utilisateur.

Le second, celui qui nous intéresse ici, est celui qui « génère » le fichier HTML. Ce fichier est ensuite utilisé par le navigateur afin de faire le premier type de rendering expliqué ci-dessus. Ce deuxième rendering se décline de différentes façons, à savoir : le *Static rendering*, le *Server Side rendering*, le *Client Side rendering* et l'*Universal rendering*.

3.3.2.Static Rendering

Les technologies du développement web ont beaucoup changé au fil des années.
Avec les débuts d'internet, seule une ou plusieurs pages statiques étaient nécessaires pour créer un site web. Ces pages contenaient souvent du texte ou des images et étaient bien souvent codées sous HTML et CSS. Ces sites utilisaient la méthode la plus conventionnelle de l'époque qui était le *Static Rendering* (rendu statique).

Cette méthode ne permettait pas de faire de site dynamique. Aucun appel à une ou plusieurs sources d'informations (une base de données, une API[4], un service web, …) n'étant fait, seuls les éléments affichés sur la page étaient disponibles.

Cette méthode est la plus sûre, son rendu est le plus rapide, son déploiement est le moins complexe à mettre en place et, de plus, elle ne demande que quelques Ko de stockage. Cependant, comme elle ne permet aucun rendu dynamique et que sa maintenance n'est pas chose aisée, elle n'est plus aussi utilisée qu'avant.

[4] API : Application Programming Interface ou interface de programmation d'application. Source de données accessible.

Effectivement, si, sur un site de vente fait en *Static Rendering*, nous étions amenés à ajouter plusieurs produits différents, nous devrions créer chaque page une à une. De la même façon, si nous souhaitions modifier les caractéristiques d'un produit, nous devrions le faire sur chaque page où ce produit apparait. Comme dit plus haut, aucune requête à une quelconque source d'informations n'est effectuée dans ce cas.

Son fonctionnement se fait comme ceci :

1) Lors de l'accès au site web, ici fruits.com, une requête est envoyée au serveur pour pouvoir afficher le site web.

2) Le serveur envoie une requête à la source de fichiers statiques qui contiennent les pages déjà traitées et rendues.

Figure 14: SR image 1

3) La source de fichiers renvoie au serveur la ou les ressources demandées.

4) Le serveur envoie au navigateur la ressource reçue par la source de fichiers statiques.

5) Affichage de la ressource sur le navigateur.

Figure 15 : SR image 2

6) Lors du clic sur l'un des boutons du site, ici banane, une nouvelle requête est envoyée au serveur afin de recevoir la ressource appropriée.

7) Le serveur envoie ensuite une requête à la source de fichiers pour qu'il puisse retourner la ressource demandée.

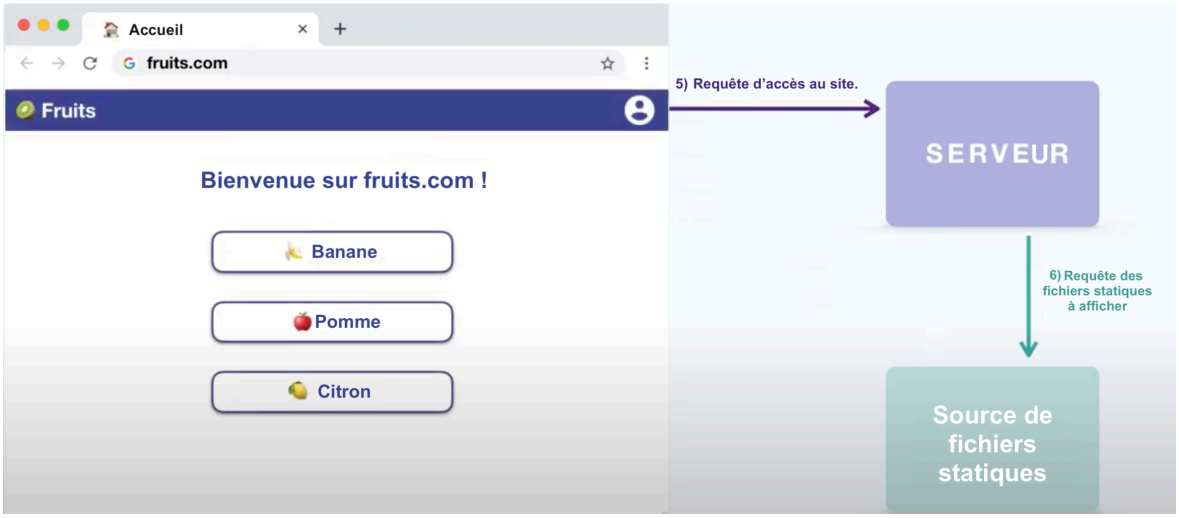

Figure 16 : SR image 3

8) La source de fichiers statiques envoie les ressources demandées au serveur.

9) Le serveur envoie les ressources reçues au navigateur, qui les affiche à l'écran.

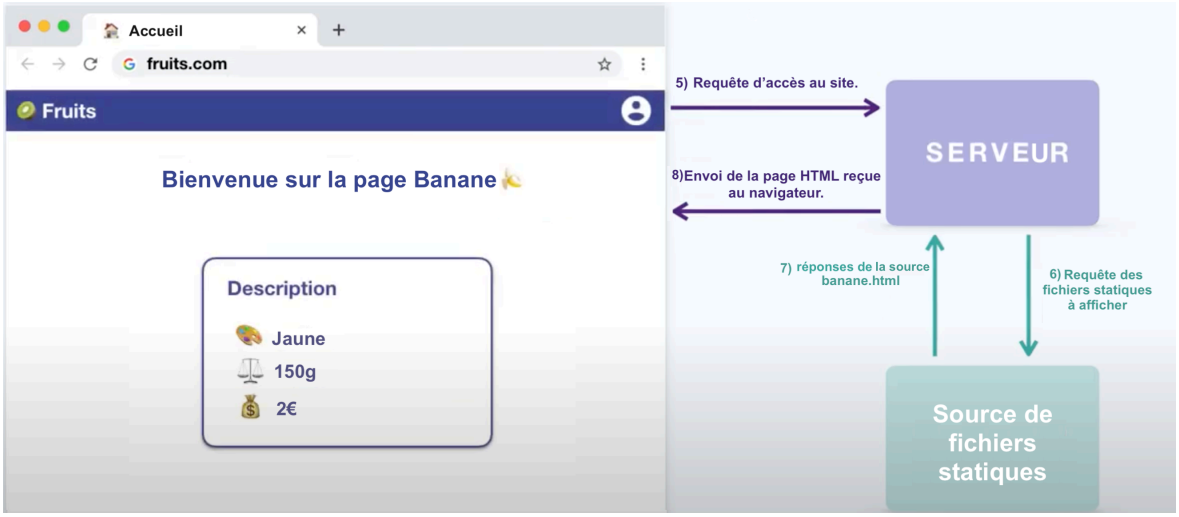

Figure 17 : SR image 4

Si nous devions résumer cette méthode de *Rendering* en une image :

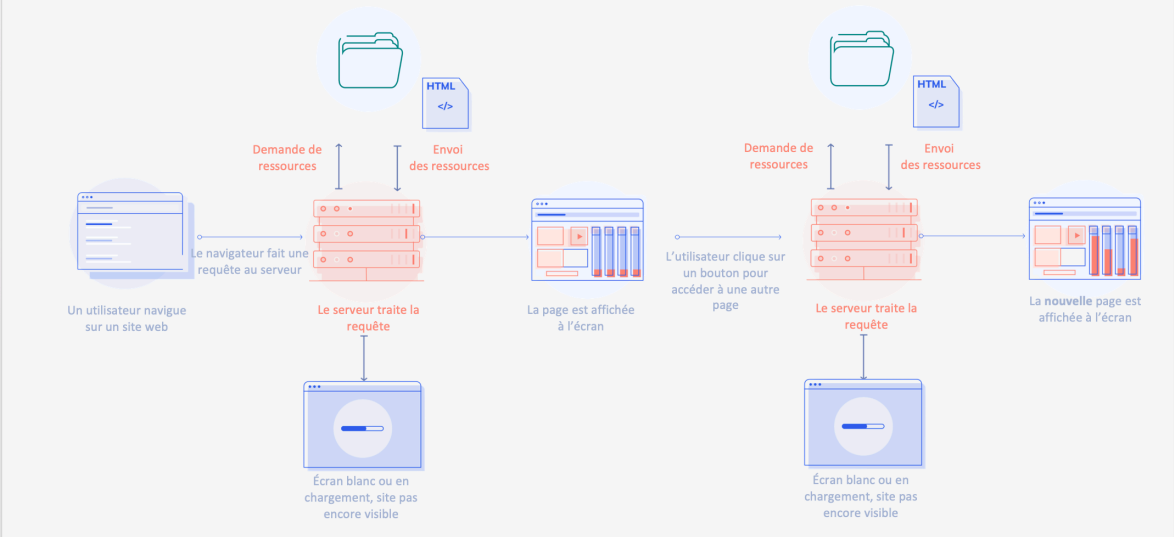

Figure 18 : Schéma Static Rendering

3.3.3.Server Side Rendering

Pour pallier le problème de rendu dynamique et de maintenance, la méthode du SSR a été inventée.
Cette méthode consiste à charger tout le rendu graphique, les traitements et les appels à la source d'informations au niveau du serveur puis de les renvoyer, une fois chargés, au browser.
Ceci permet à l'usager de ne pas devoir posséder un ordinateur puissant pour afficher une page internet. A l'époque, les ordinateurs à usage personnel vendus dans le commerce n'étant pas vraiment puissants, cette méthode était relativement efficace.

Elle fonctionne comme suit :

1) Lors de l'accès au site web, ici fruits.com, une requête est envoyée au serveur pour pouvoir afficher le site web.
2) Le serveur envoie une requête à la source avec les données demandées, ici la page « fruits.com ».

3) La source d'informations lui répond en lui donnant les données demandées, ici les sortes de fruits.

4) Le serveur traite les différents fichiers html et scripts.

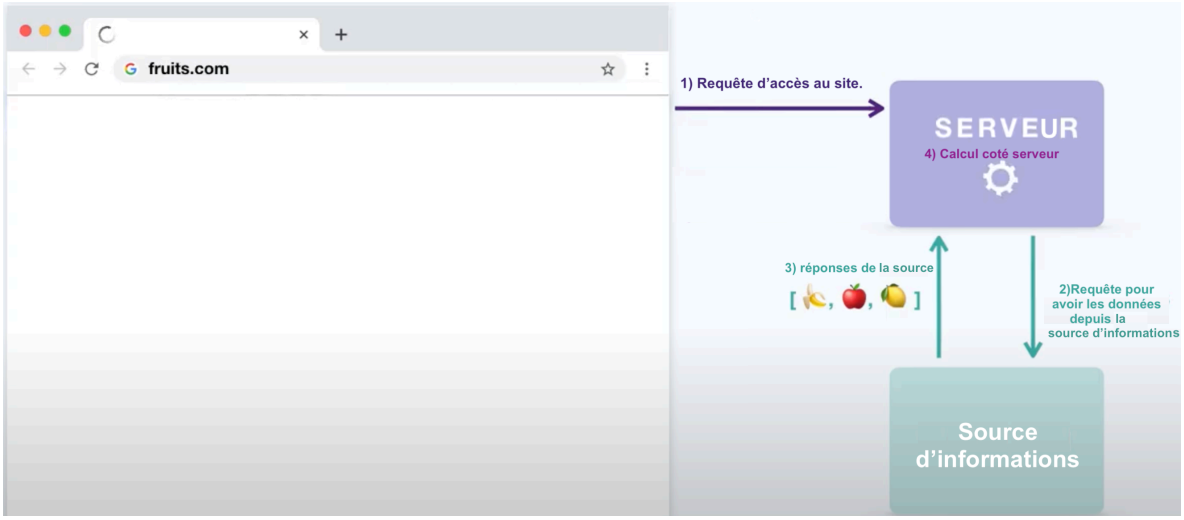

Figure 19 : SSR image 1

5) Une fois les traitements de rendu effectués, le serveur envoie l'affichage de la page d'accueil sur le navigateur de l'utilisateur.

Figure 20 : SSR image 2

6) Lors du clic sur l'un des boutons de la page, ici le bouton « banane », une nouvelle requête au serveur est effectuée pour y accéder.

7) Le serveur envoie une requête à la source d'informations en demandant des infos sur la page en question, ici la page banane.

8) La source d'informations répond au serveur et lui envoie les données qu'elle possède à ce sujet.

9) Le serveur traite le rendu graphique de la page.

10) Une fois le rendu graphique traité, le code html, avec les données de la source d'informations, est envoyé au navigateur web.

Figure 21: SSR image 3

Cependant, les requêtes faites par le navigateur au serveur, qui contiennent les données du site internet, peuvent prendre un certain délai. Théoriquement, ce délai n'est censé être que de quelques millisecondes, mais dans la pratique, cela varie énormément à cause des conditions du réseau dans lequel on se trouve, de la rapidité de sa connexion, de la localisation du serveur, du nombre de requêtes simultanées faites sur ce serveur, de l'optimisation du site internet, etc.
Tous ces facteurs peuvent allonger ce délai de quelques millisecondes à quelques secondes, voire même, dans les cas les plus extrêmes, plusieurs minutes. Ce qui peut être dérangeant pour un simple affichage de site internet.

Si nous devions résumer cette méthode de *Rendering* en une image :

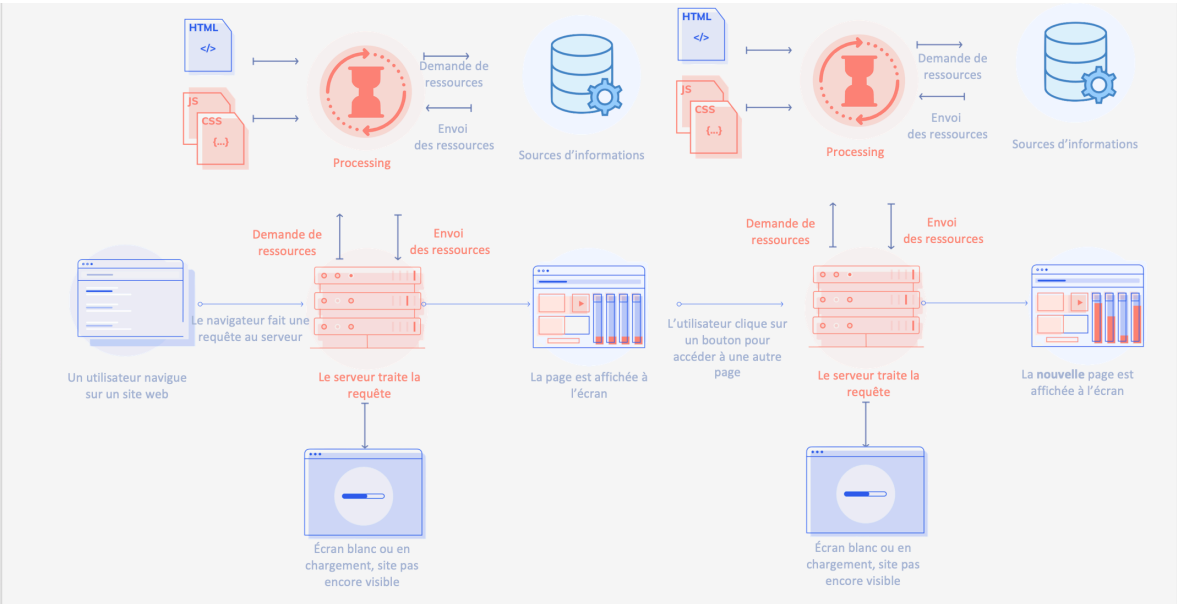

Figure 22 : Schéma Server Side Rendering

3.3.4.Client Side Rendering

Pour pallier les éventuelles latences dues aux sites toujours plus volumineux, on préfère l'utilisation d'un *Client Side Rendering* (rendu côté client ou CSR). Effectivement, cette méthode demande à l'usager de faire certains traitements et rendus graphiques grâce à son navigateur.
Cette dernière est codée majoritairement en HTML, comme pour le SSR, mais avec des parties de JavaScript incrustées au code. Ces parties permettent également de rendre les sites internet beaucoup plus dynamiques. Cette méthode demande plus de puissance au niveau de l'ordinateur de l'usager mais offre des possibilités que la méthode du SSR ne peut pas proposer.

Elle fonctionne comme suit :

1) Lors de l'accès au site web, ici fruits.com, une requête est envoyée au serveur pour pouvoir afficher le site web.
2) Le serveur répond directement au navigateur et lui envoie la page html non compilée qui contient essentiellement des scripts et des balises de style. Sans contenu, la première chose que l'utilisateur voit, est une page blanche.

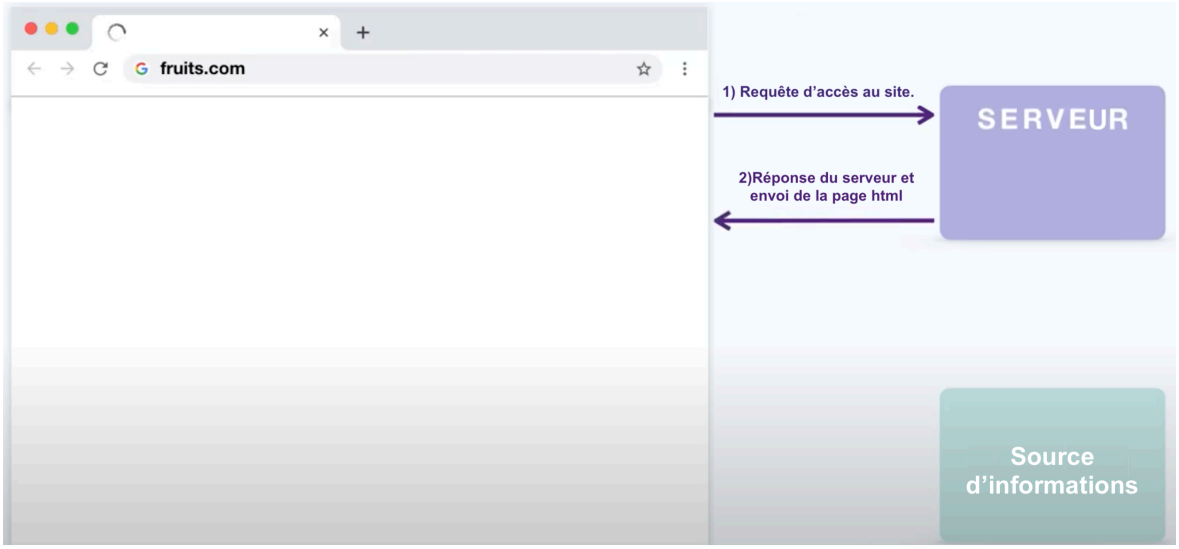

Figure 23 : CSR image 1

3) Le navigateur fait une ou plusieurs requêtes pour que le serveur exporte ensuite les paquets JavaScript.

4) Le serveur envoie les paquets JavaScript au navigateur, si ceux-ci ne sont pas encore téléchargés côté client.

5) Une fois les paquets récupérés, le navigateur compile les paquets et en ressort un « layout » (plan du site) HTML sur le navigateur. Celui-ci sera certainement un plan commun à toutes les pages ; la page n'a donc pas toutes les informations et devra donc demander l'export des données à la source d'informations.

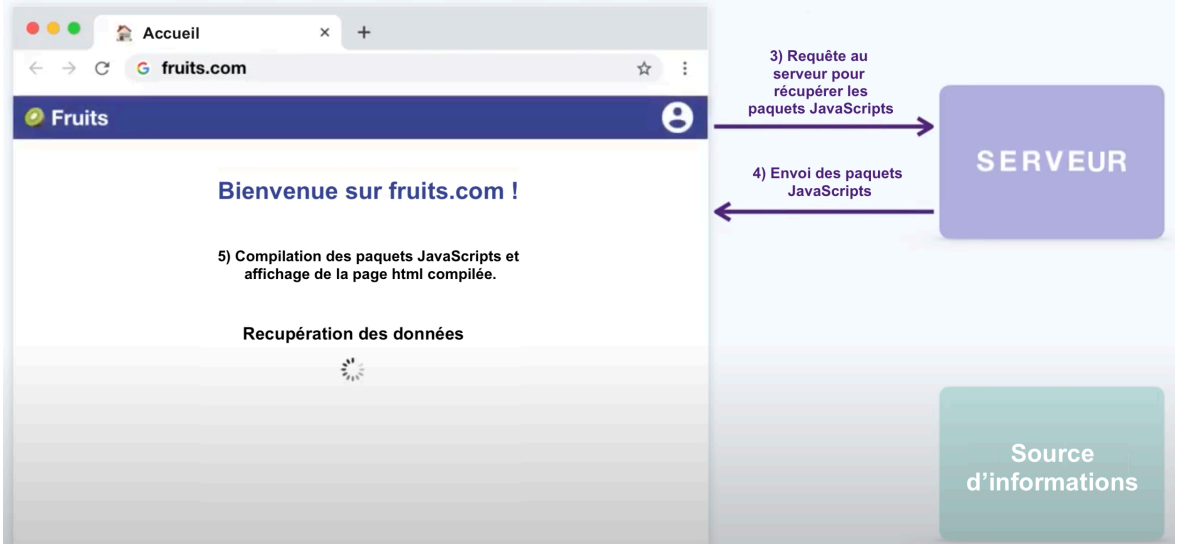

Figure 24 : CSR image 2

6) Le navigateur demande donc à la source d'informations de lui envoyer les données relatives à la page fruits.com pour étoffer la page d'accueil.

7) La source d'informations envoie les données pour que le navigateur puisse afficher correctement le contenu de la page.

8) Une fois ces données reçues, le contenu dynamique est affiché à l'écran.

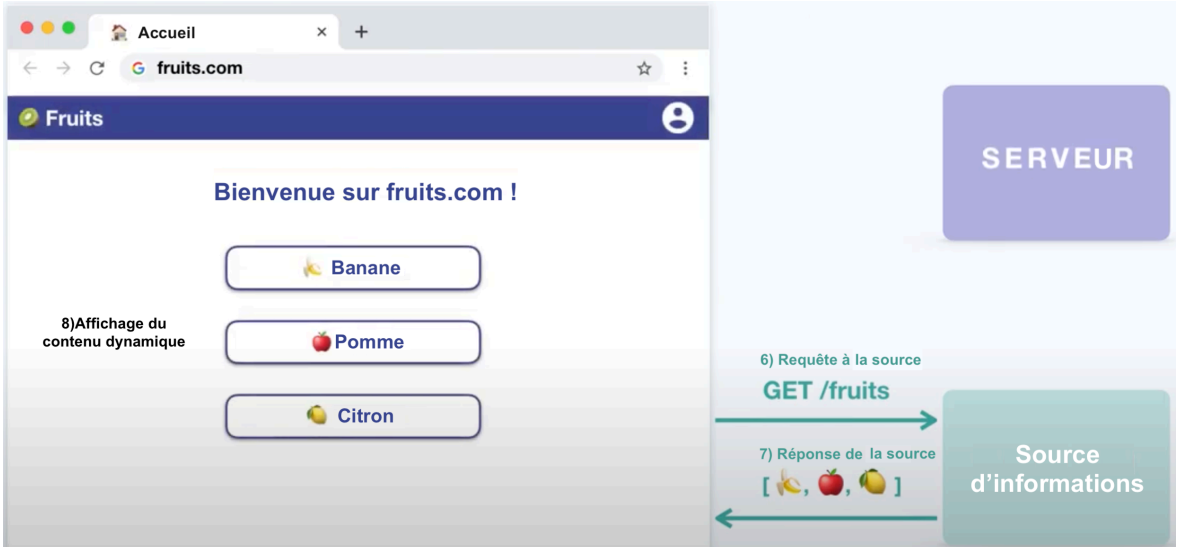

Figure 25 : CSR image 3

9) Lors du clic sur un élément pour naviguer sur une autre page, ici la page banane par exemple, le layout sera directement affiché à l'écran mais une demande d'export des données relatives à la page banane est faite à la source d'informations.

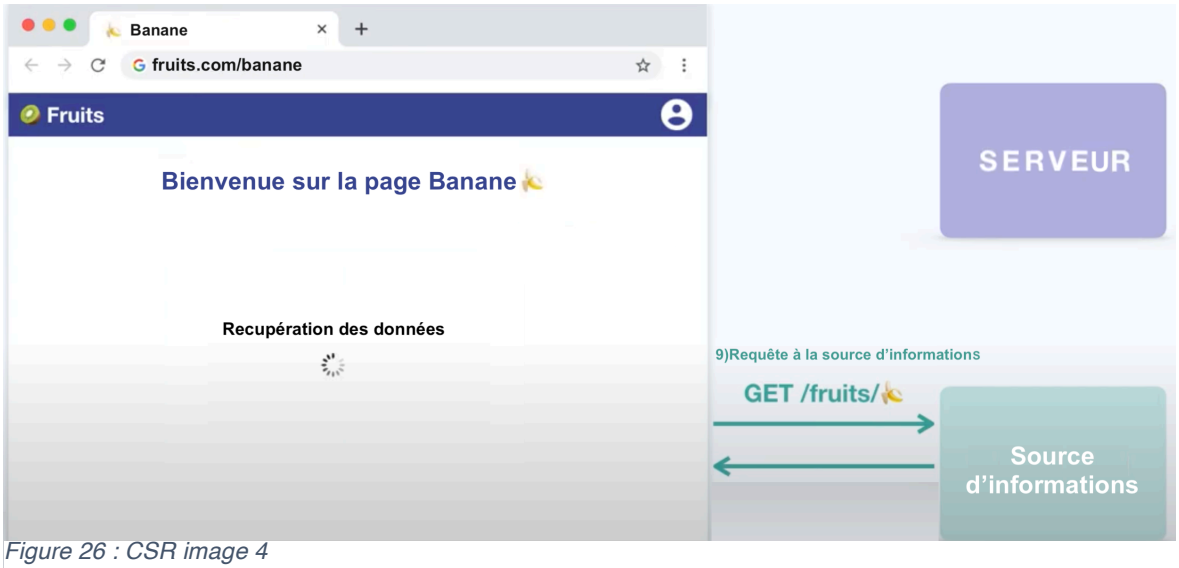

Figure 26 : CSR image 4

10) La source d'informations répond au navigateur en lui envoyant les données relatives à la page souhaitée, ici la page banane.

11) Une fois ces données reçues, le contenu dynamique est affiché à l'écran.

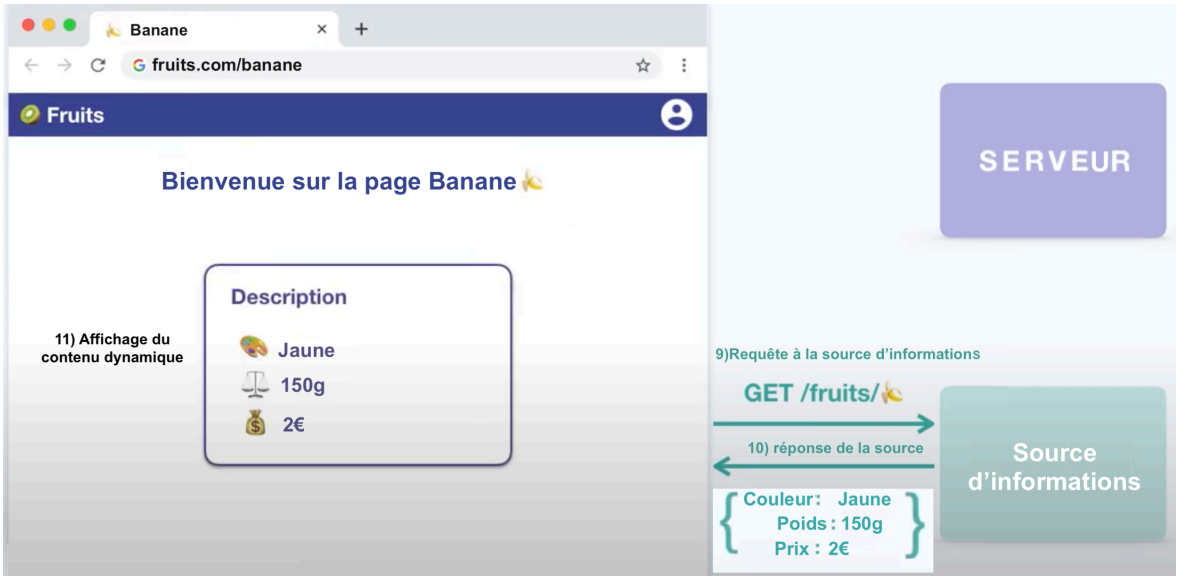

Figure 27 : CSR image 5

Cependant, cette méthode a quelques défauts :
Vu que le rendu se fait du côté client, le navigateur est soumis à une plus grande contribution que dans le cas d'un rendu côté serveur. Cela peut engendrer certaines gênes si le site est souvent utilisé sur mobile (comme une utilisation excessive de la batterie par exemple).

Le défaut majeur du *Client Side Rendering* est son inefficacité en termes de SEO.
Effectivement, les robots des moteurs de recherche, qui s'occupent d'indexer les sites, ne sont que peu ou pas capables d'exécuter du code JavaScript. Donc, si tout le rendu est fait en JavaScript, ces robots ne « voient » aucun contenu à indexer. Ceci rend le référencement SEO impossible ou presque.

Depuis quelques années, Google a développé un robot appelé « Googlebot » qui peut exécuter quelques lignes de JavaScript pour aider à ce référencement. Mais il ne faut pas que le code JavaScript soit trop conséquent, auquel cas, il n'arrivera pas à référencer ce site.

Si nous devions résumer cette méthode de *Rendering* en deux images :

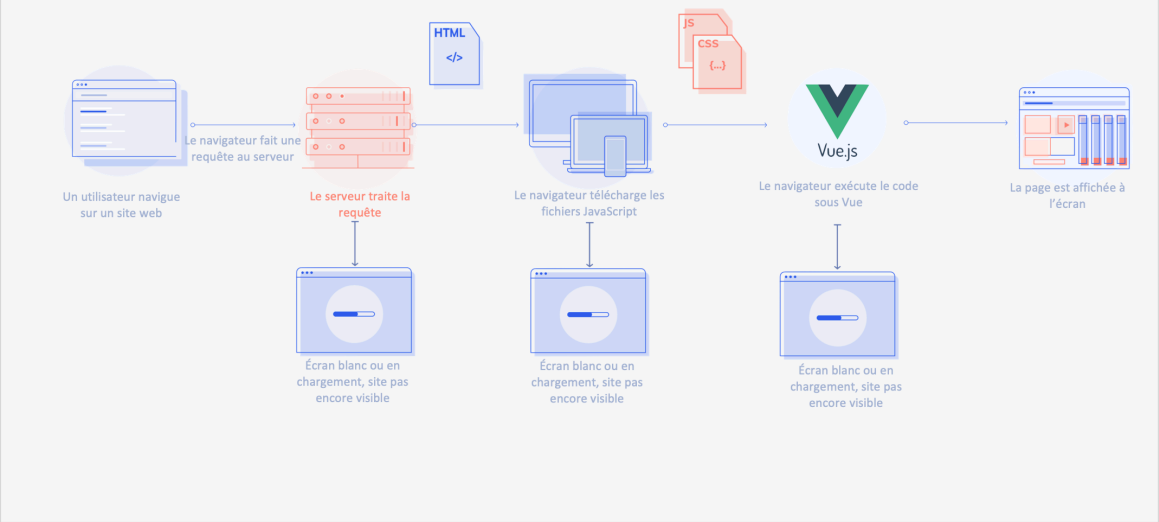

Figure 28 : Schéma Client Side Rendering 1

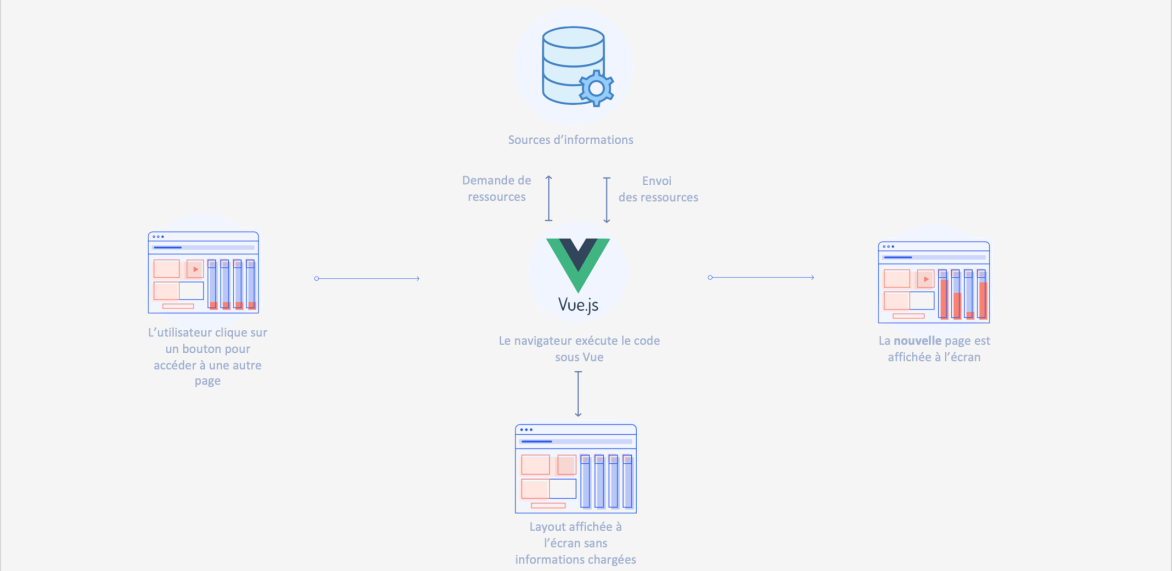

Figure 29 : Schéma Client Side Rendering 2

3.3.5.Universal Rendering

Depuis quelques temps une nouvelle méthode de rendu a fait son apparition dans le monde du développement web. Celle-ci est appelée *Universal Rendering*. Elle combine les avantages des deux méthodes que sont le SSR et le CSR. Elle permet d'avoir un affichage même si les données de la page ne sont pas encore envoyées depuis la source d'informations. Cette page ne doit pas totalement être rechargée lors de la navigation.
Cette méthode commence, comme le rendu côté serveur, tout en ayant la finalité d'un rendu côté client. C'est-à-dire que les fichiers JavaScript sont téléchargés en local sur le client.

Elle fonctionne comme suit :

1) Lors de l'accès au site web, ici fruits.com, une requête est envoyée au serveur pour pouvoir afficher le site web.

2) Le serveur envoie une requête à la source d'informations avec les données demandées, ici la page « fruits.com ».

3) La source d'informations lui répond en lui donnant les données demandées, ici les sortes de fruits.

4) Le serveur traite les différents fichiers HTML et scripts.

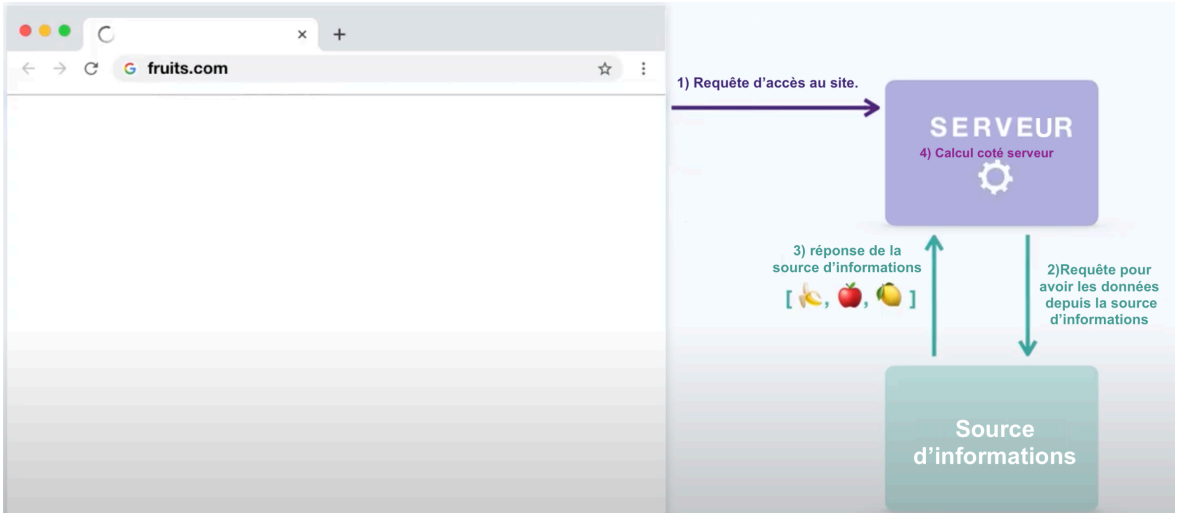

Figure 30 : UR image 1

5) Une fois les traitements de rendu effectués, le serveur envoie la page d'accueil sur le navigateur de l'utilisateur.

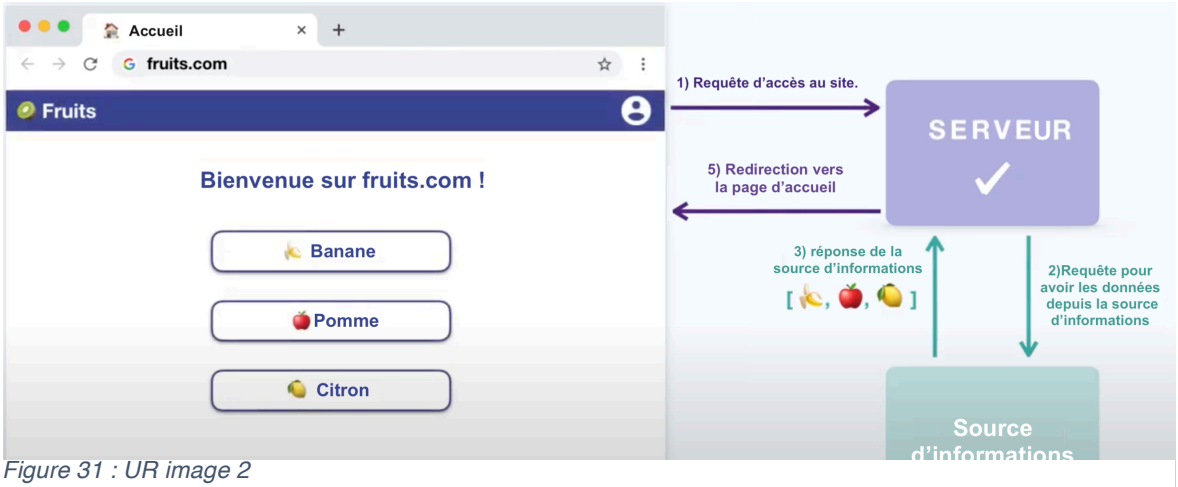

Figure 31 : UR image 2

6) Même si l'affichage ici est complet, le site n'est pas encore interactif et donc l'utilisateur ne pourra appuyer sur aucun bouton pour afficher une autre page.

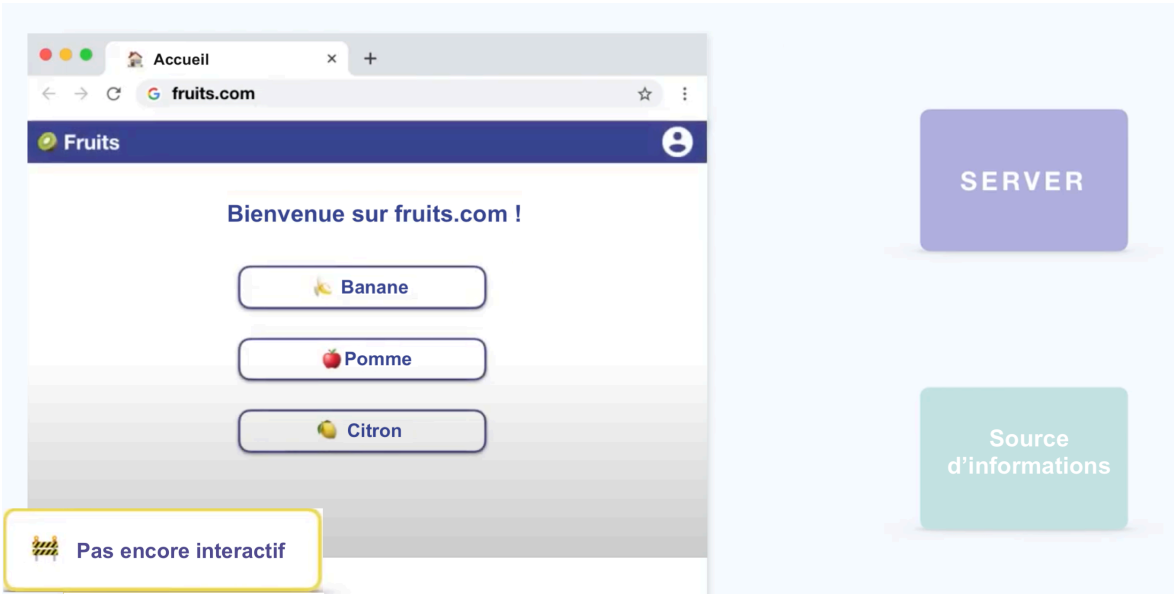

Figure 32 : UR image 3

7) Il faut que le navigateur reçoive les fichiers JavaScript pour que le site soit totalement fonctionnel.

8) Une fois les fichiers JavaScript reçus, une période de traitement commence. Cette période consiste à synchroniser l'état de l'interface utilisateur de l'application avec l'état initial reçu du serveur (synchronisation du JavaScript avec le HTML reçu auparavant).

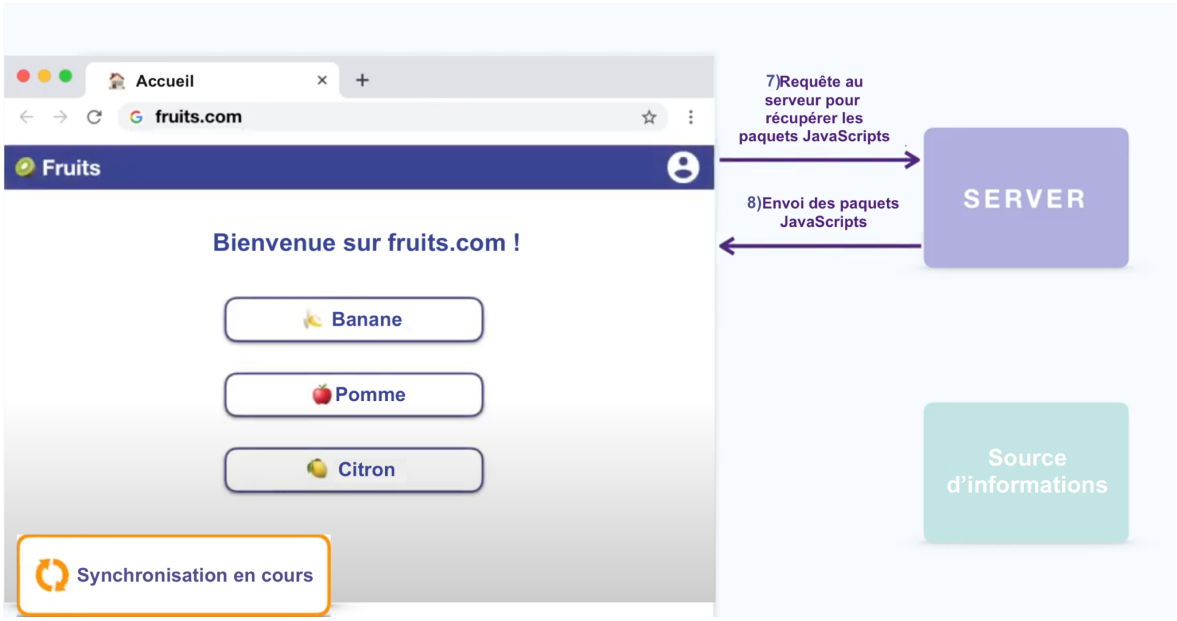

Figure 33 : UR image 4

9) Une fois cette période de synchronisation effectuée, la page devient totalement interactive et donc l'utilisateur pourra être redirigé vers les pages lors d'un clic sur un bouton.

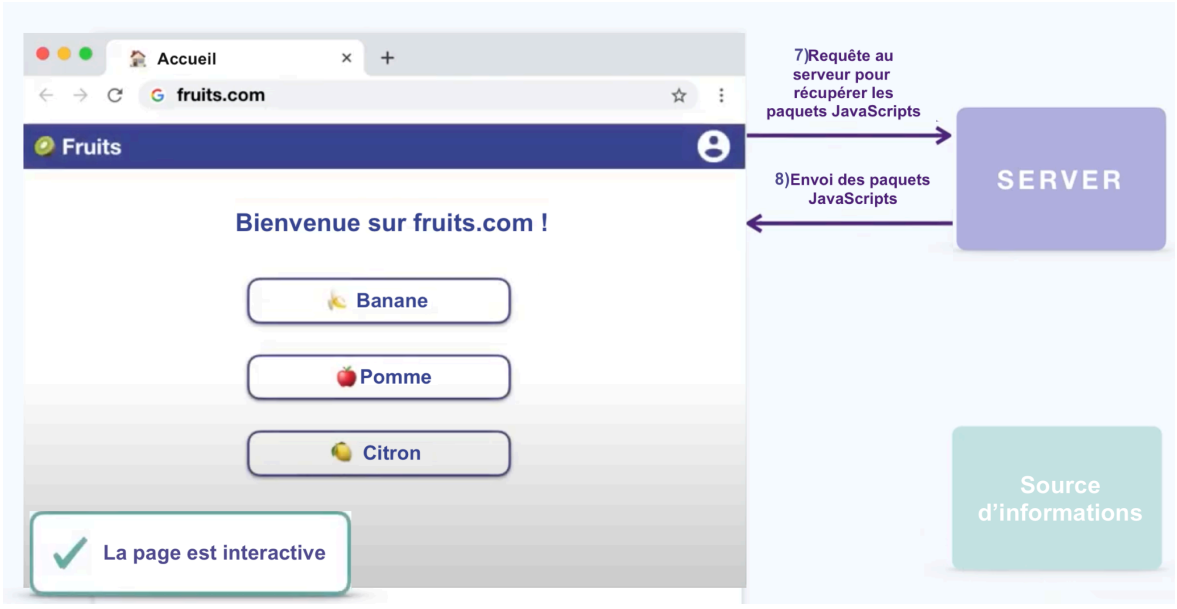

Figure 34 : UR image 5

10) Lors du clic sur un élément pour naviguer sur une autre page, ici la page banane, le layout sera directement affiché à l'écran. Une demande d'export des données relatives à cette page sera faite à la source d'informations.

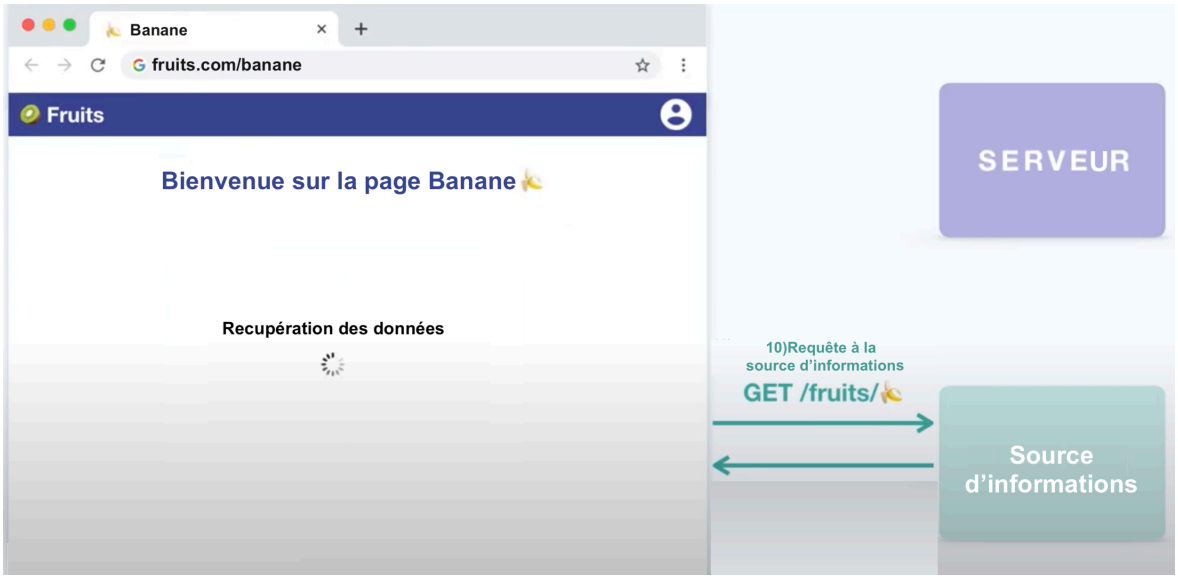

Figure 35 : UR image 6

11) La source d'informations répond au navigateur en lui envoyant les données relatives à la page souhaitée, ici la page banane.

12) Une fois les informations reçues, le contenu dynamique est affiché à l'écran.

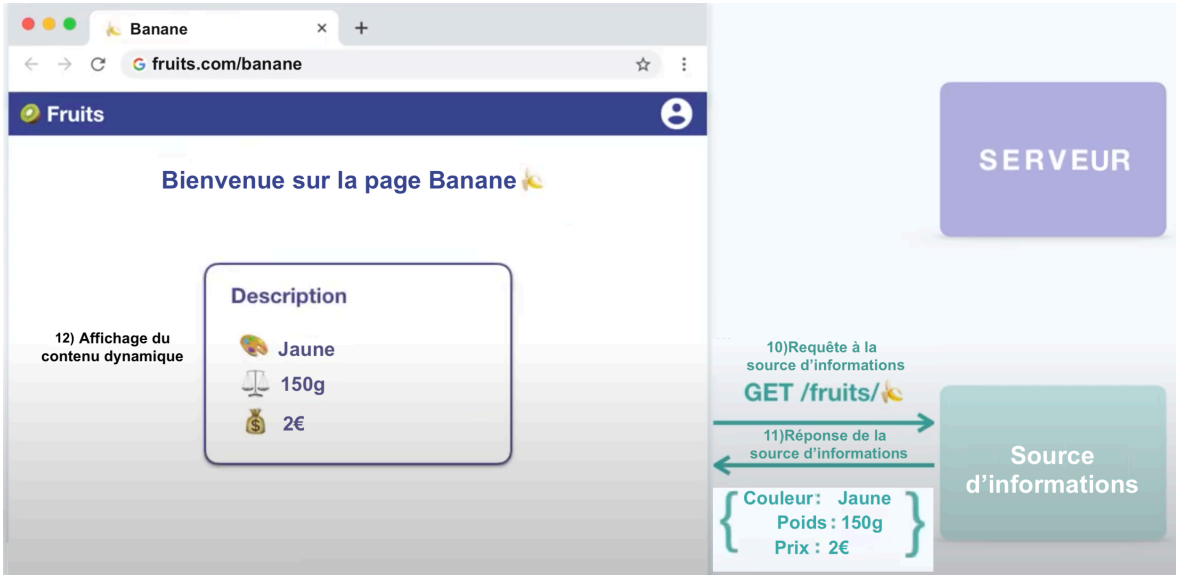

Figure 36 : UR image 7

Si nous devions résumer cette méthode de *Rendering* en deux images :

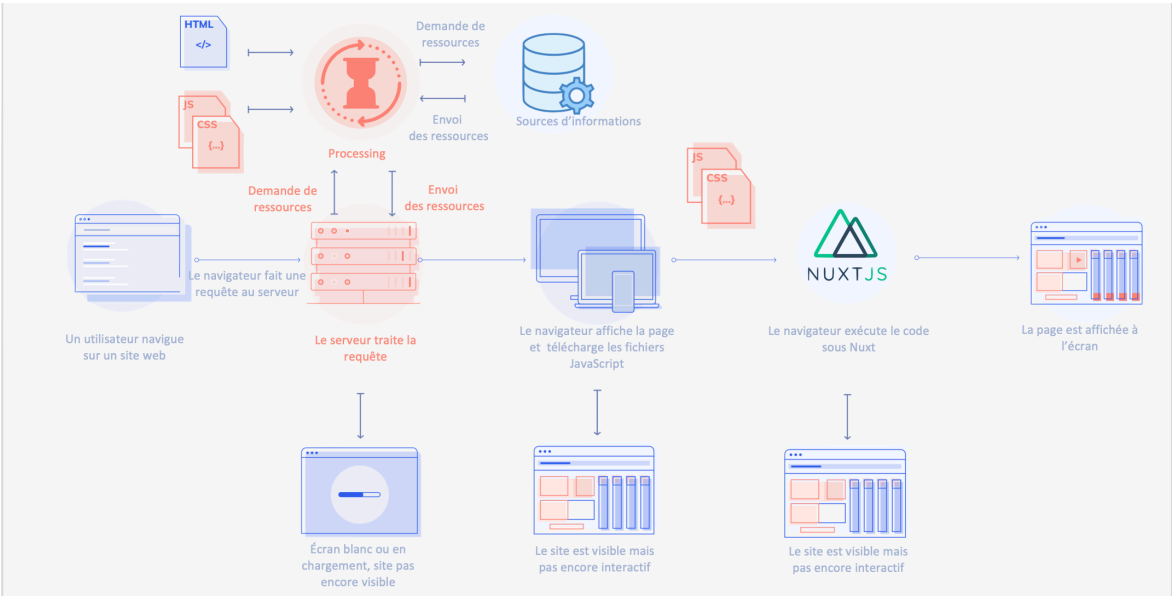

Figure 37 : Schéma Universal Rendering 1

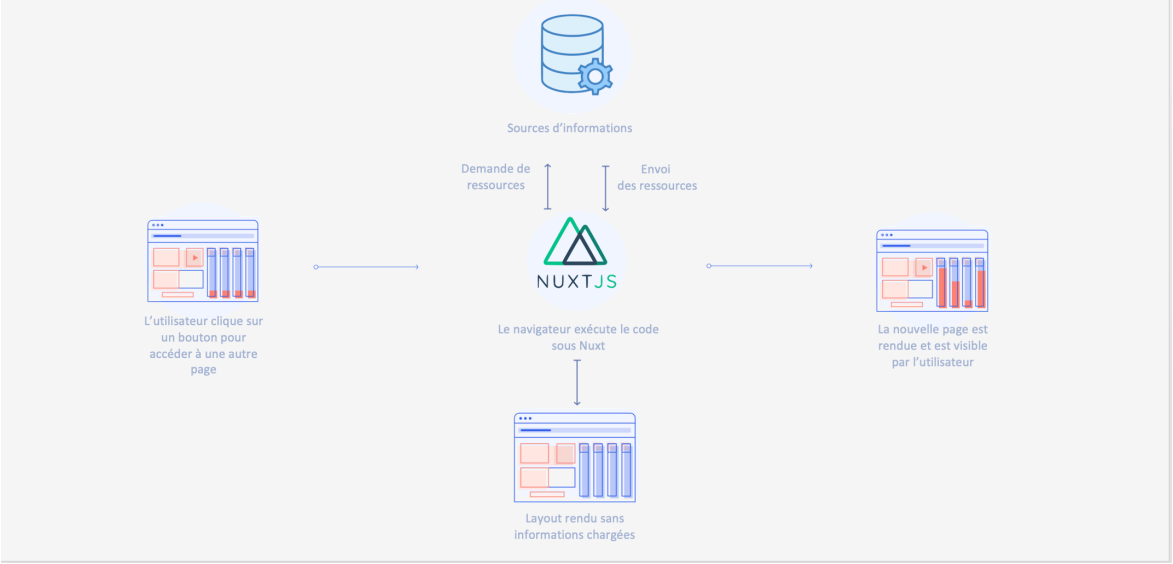

Figure 38 : Schéma Universal Rendering 2

3.3.6.Static, Server Side, Client Side ou Universal Rendering?

Suite à toutes ces explications, une comparaison s'impose :

Il y a quelques points importants lors de la création de site internet qui peuvent entrer en compte lors du choix de sa méthode de rendu.

Il convient donc de se baser sur ces points pour avoir une comparaison.

	Static Rendering	Server Side Rendering	Client Side Rendering	Universal Rendering
Dynamique	❌❌	✅✅	✅✅	✅✅
Rapidité de rendu à la première requête	✅✅	✅	❌	✅
Rapidité de rendu de la page après la première requête	✅	❌	✅✅	✅✅
SEO	✅✅	✅✅	❌❌	✅✅
Flexibilité	❌❌	✅✅	✅✅	✅✅
Rafraîchissement	Toute la page	Toute la page	Uniquement les éléments à changer	Uniquement les éléments à changer
Intelligence	Pas d'intelligence. Le serveur renvoie un fichier statique. Le navigateur l'interprète.	Sur le serveur web. Le « code » s'exécute sur le serveur majoritairement.	Sur le client. Le serveur web ne donne que des fichiers statiques. Un serveur d'application fournit les données.	Sur le client et au build pour générer les fichiers statiques. Pas d'intelligence sur le serveur. Un serveur d'application fournit les données.

Figure 39 : Tableau récapitulatif Rendering

Légende : ❌❌=Médiocre, ❌=Mauvais, ✅= Bon, ✅✅= Excellent

3.4.Présentation de Nuxt.js

Nuxt.js est un framework[5] open source[6] gratuit basé sur Vue.js (framework de JavaScript) et de Node.js. Ce framework a été créé afin qu'on puisse concevoir des applications web dites Universelles. Avec Nuxt.js il est possible d'élaborer des applications web dites « Single Page Application » en CSR mais également de produire des applications sur base du *Universal Rendering*. Ce framework a l'avantage d'offrir la possibilité de mettre en place un très bon SEO. Nuxt.js permet de créer son application web très facilement, car celui-ci rend également possible l'intégration de modules et de librairies, ce qui permet aux développeurs de gagner énormément de temps.

De plus, le « Routing » est autogéré. C'est-à-dire que la navigation entre les pages est déjà prévue. Au contraire de Vue.js par exemple, dans lequel un fichier de routage est nécessaire grâce à vue-router. Sur Nuxt.js, toutes les pages dans le même dossier parent peuvent être accessibles en mentionnant une balise « NuxtLink » et le lien de la page.

```
<NuxtLink to="/"
    class="text-blue-700
    Log out
</NuxtLink>
```

Figure 40 : Exemple NuxtLink Simple

Ici on retrouve la balise ouvrante et fermante « NuxtLink » qui permet de stipuler qu'un clic sur le champ « Log out » redirigera vers l'adresse « / » qui est la racine de l'application web.
Le champ « class » ici présent ajoute simplement des éléments de style pour rendre ce champ « Log out » plus agréable visuellement.

Ces redirections peuvent également se faire en passant des paramètres. En voici un exemple concret :

```
<NuxtLink
  :to=" {name:'HomePage', params: { offset : 0}}"
  class="hover:from-pink-500 hover:to-orange-500
  >
    HOME
</NuxtLink>
```

Figure 41 : Exemple NuxtLink avec paramètres

Ici on retrouve la balise ouvrante et fermante « NuxtLink » qui permet de stipuler qu'un clic sur le champ « HOME » redirigera vers le lien portant le nom « HomePage » avec le paramètre « offset » qui sera égal à 0.
Le champ « class » ici présent ajoute simplement des éléments de style pour rendre ce champ « HOME » plus agréable visuellement.

[5] Un framework est une infrastructure de développement.
[6] Open source : Logiciels dont le code est public. Ces projets sont généralement le fruit d'une collaboration entre programmeurs.

Grâce à ces balises « NuxtLink » la page est déjà « prefetch » c'est-à-dire que le JavaScript est déjà traité ; la page est déjà prête avant même que le clic sur l'élément balisé soit fait. Cela est dû au système de « Smart Prefetching ». Ce système fait en sorte de charger les pages à lien visibles et de préparer la page afin de l'afficher plus rapidement si l'utilisateur clique sur le lien. Ce processus ne s'effectue que lorsque le navigateur n'est pas occupé. Il n'a pas lieu lorsque l'utilisateur est hors ligne ou a une connexion internet plus faible (connexion 2G).
Nuxt.js est également très léger. Avec sa configuration minimale en version *Universal Rendering* il ne dépasse pas les 100Mo.

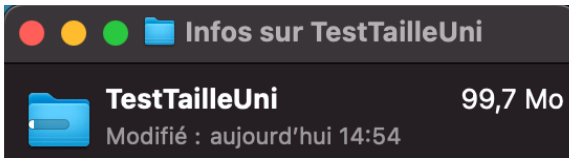

Figure 42 : Taille minimale Universal Rendering

Comme une application sous Nuxt.js peut être développée en vue d'être en *Universal Rendering*, il est possible d'ajouter un « Layout » commun à toutes les pages. Ce qui permet par exemple d'ajouter un en-tête et un pied de page identiques à toutes les pages. Néanmoins il est possible de ne pas activer cette fonctionnalité sur certaines pages.

Nuxt.js offre la possibilité d'avoir et d'utiliser des composants. Un bout de code répétitif peut être exporté dans un composant afin de ne pas devoir réécrire ce code à chaque besoin. Cela permet une meilleure lecture du code et évite la redondance (principe DRY : Don't Repeat Yourself). Dans Nuxt.js les composants sont dits « Globaux » et donc peuvent être utilisés partout dans le projet, que ce soit dans les pages ou dans les « Layout ».

3.5.TailwindCss

Lors du projet de stage, il m'a été demandé d'ajouter tailwindcss pour une gestion plus facile de l'habillage du site grâce à CSS.
Tailwindcss est un framework CSS permettant une meilleure gestion du CSS ainsi qu'une facilité de développement. Ce framework a été développé en 2017 par Adam Wathan et est encore aujourd'hui développé par son équipe et lui-même.

Ce framework est basé sur les principes de classes utilitaires, qui sont des classes à un seul et unique but. Il ne reprend pas le principe des classes sémantiques qui demandent d'inventer des noms de classes pour chaque élément d'interface que l'on voudrait ajouter.

Toutefois, il est toujours possible de créer ses propres classes sémantiques pour pouvoir identifier un élément plus rapidement, par exemple.

Tailwindcss a également la particularité d'être intégré au fichier HTML. Il n'est donc pas nécessaire de faire un second fichier dans lequel le CSS se trouvera.

Comme chaque classe a sa fonction bien précise, il nous en faudra plus pour pouvoir effectuer la même action que sur d'autres framework tel que Bootstrap par exemple. Si l'on souhaite afficher un bouton bleu sur Bootstrap, deux classes suffisent :

```
<a href="#" class="btn btn-primary">
  Bouton
</a>
```

Figure 43 : Exemple bouton sous Bootstrap

Tandis qu'avec tailwindcss, pour afficher le même bouton, il en faudra 6 :

```
<a href="#" class="bg-blue-700 text-white px-5 py-3 font-bold rounded">
  Bouton
</a>
```

Figure 44: Exemple bouton sous Tailwindcss

Ceci peut paraître perturbant au premier abord lors du développement mais cela permet une grande personnalisation des éléments à afficher. Ces classes sont nombreuses et fournies. Que ce soit pour de la typographie avec la classe « font-size » ou « font-color » ou même des effets avec la classe « shadow », il en existe suffisamment pour personnaliser à souhait.
Toutes ces classes sont directement présentes dans le code HTML ce qui donne la possibilité de ne pas devoir changer de fichier ou de quitter son code pour devoir changer un simple élément comme une taille de police par exemple. L'emploi intensif de composants est cependant à recommander pour assurer la maintenabilité.

Tailwindcss permet d'ajouter le fait qu'un élément visuel soit dit « Responsive ». Effectivement avec l'arrivée massive des mobiles dans le marché du web, il est nécessaire, voire obligatoire, d'avoir son application utilisable sur un plus petit écran.
Ce que permet tailwindcss de faire est d'ajouter des classes « lg », « md », « sm » et bien d'autres pour pouvoir afficher la classe adéquate selon la taille de l'écran.

Ce framework a la chance d'avoir une très grande communauté qui met en ligne de nombreux et différents composants disponibles sur le site de tailwindcss même :
https://tailwindcomponents.com/

Sur ce site, on retrouve une documentation qui détaille toutes les classes disponibles avec tailwindcss notamment, des exemples d'utilisation avec le code associé, ce qui facilite le travail de recherche du développeur.

Ces différents concepts techniques étant désormais connus, nous pouvons entrer dans le vif du sujet.

3.6.Projet de stage

Lors de ce stage en entreprise, l'objectif qui m'a été donné était de préparer la migration d'une Single Page application codé sous React.js, à une application en *Universal Rendering* sous Nuxt.js.

Le passage de React.js à Vue.js, pour pouvoir l'installer sur Nuxt.js, n'étant pas ce qui a été demandé, nous ne nous attarderons pas sur cette partie.

Pour prendre en main le framework Nuxt.js, il m'a été demandé de développer une application en « Single Page Application » pour me familiariser avec le code Nuxt.js.

Cette application consiste à afficher des fiches de présentation des députés politique du groupe ECR. Ces données sont disponibles dans une base de données. Cette application étant en SPA, il a été nécessaire de passer par un serveur Node.js pour aller récupérer ces données.

3.6.1.Serveur Node.js / Back-End

En premier lieu, nous avons dû faire le lien entre le serveur Node.js et la base données PostgreSQL. Pour ce faire nous avons dû importer les librairies compatibles avec la base de données et créer une connexion entre cette dernière et le serveur node.js. Pour des questions de facilité de relecture du code, nous avons préféré exporter le fichier de configuration de la connexion sur un autre fichier. La connexion se fait comme suit :

Un appel au fichier de configuration de la base de données.

```
const pool = require('./config/db.config')
```

Figure 45 : Connexion au fichier db.config

Ensuite la configuration de la connexion se fait et un export de cette connexion est envoyé au code l'ayant appelé.

```
NodeServer > config > JS db.config.js > ...
1   const Pool = require('pg').Pool
2
3   const pool = new Pool(
4     {
5       host: "localhost",
6       user: "postgres",
7       password:
8       database: "ECR",
9       port: 5432 ,
10    }
11  )
12
13  module.exports = pool
```

Figure 46 : Configuration de la base de données

Une fois la connexion effectuée, il a fallu exporter les informations nécessaires au bon fonctionnement du code. Comme nous n'avons pas besoin de toutes les informations de la base de données en même temps, nous faisons des requêtes SQL[7] précises.

Ces requêtes sont envoyées à la base de données qui répond avec un objet que nous transformons en format Json. Celui-ci est exporté afin que l'application SPA puisse l'utiliser.

Nous avons également configuré le serveur node.js pour que celui-ci envoie ces mêmes requêtes SQL lorsqu'il reçoit une certaine requête depuis le « Front-End ». Cette configuration s'est faite avec la méthode POST sur une certaine URL.

La méthode GET aurait pu être utilisée, mais l'envoi de requêtes entre le « Front-End » et le « Back-End » avec la méthode POST a été imposée par le maître de stage.

Un exemple de cette configuration sur l'URL « /ListPerson » du serveur node.js :

```
app.post('/ListPerson', async (req, res)  => {
    try{
        const { search } = req.body
        const { offset } = req.body
        const { pageSize} = req.body

        const query = {
             text :
                `SELECT persid,login,firstname,lastname,aspicture,nationality,salutation.title
                FROM person
                INNER JOIN salutation ON person.salutid=salutation.salutid
                WHERE CONCAT(firstname,' ',lastname) ILIKE $1
                ORDER BY lastname
                OFFSET $2
                LIMIT $3
                `,
              values : ['%'+search+'%',offset,pageSize]
        }
        const result = await pool.query(query);
        return res.status(200).json({
            cards : result.rows
        });
    } catch (err){
        console.log(err.message)
    }
  })
```

Figure 47 : Exemple d'export de données Back-End

On retrouve bien la méthode POST utilisée mais également l'URL sur laquelle la requête doit être effectuée pour exécuter le code. On trouve également un tag « async » dans celui-ci simplement pour stipuler qu'il s'exécutera de façon asynchrone.

Ensuite, si la connexion à la base de données a bien été effectuée, nous exécutons le code, si ce n'est pas le cas, un message d'erreur, comprenant cette erreur, est envoyé à la console. Cette erreur est plus souvent écrite dans des « log files [8]».

[7] SQL (Structured Query Language) est un langage de requête structuré. Ces requêtes sont utilisées pour exploiter des bases de données relationnelles.

[8] Log files : Fichiers de logs. Ces fichiers comprennent des informations liées à l'utilisation d'un serveur.

Si l'on rentre dans le code, alors on récupère dans le corps de la requête (« req.body ») les paramètres envoyés (nous verrons comment est constituée la requête ci-dessous).
Une fois les paramètres stockés dans leurs variables respectives, nous décrivons une nouvelle variable (« query ») qui prendra deux paramètres : un champ texte et un champ valeurs. Cette variable sera en fin de compte notre requête SQL.

Après ceci, un envoi de la requête SQL à la base de données est effectué (« await pool.query(query) » et celle-ci nous envoie les informations que nous formatons en format Json (« return res.status(200).json ». Ce texte en format Json sera stocké dans un tableau d'objets appelé « cards » (« cards : result.rows »).

Ce tableau d'objet « cards » comprend chaque élément du « SELECT » de la requête SQL ainsi que la valeur « salutation » présente dans la fonction « INNER JOIN » pour chaque député.

Ce genre de configuration est effectuée autant de fois que des paquets d'informations différentes devront être envoyées à la « Single Page Application » qui est, dans ce cas-ci, la partie « Front-end » du projet.

Cette dernière n'est plus codée sous node.js mais bien sous Nuxt en version SPA.

3.6.2.Single Page Application / Front-End

Le framework Tailwindcss, cité précédement, a été utilisé pour mettre en forme cette partie.

Pour récupérer des informations sur une page du côté « Front-end » depuis le serveur Node.js également appelé « Back-end », il faut configurer l'envoi de ces requêtes citées auparavant.

Un exemple de configuration de l'envoi de ces requêtes du coté SPA :

```
let request= await axios
  .post('http://localhost:8080/ListPerson',{
      search : mysearch,
      offset : this.offset,
      pageSize: this.pageSize
  }).then(res => {
  (this.cards = res.data.cards)

  }
  ).catch(err =>console.log(err.message))
```

Figure 48 : Exemple d'import de données Front-End

Dans ce code on retrouve le champ « await » qui est en lien étroit avec le tag « async » situé plus haut dans le code (le champ « async » non montré par facilité d'explications).

On retrouve également la requête avec la méthode POST et son URL sur laquelle on doit aller chercher les informations (cette URL correspond à celle du serveur Node.js configuré plus haut). Cette requête contient des paramètres dans son corps et sont également envoyés sur le côté « Back-end ».

Une configuration spécifique au niveau du serveur Node.js est nécessaire pour que la requête soit entendue sur le bon port :

```
app.listen(8080, function() {
    console.log('Server is running on PORT:',8080)
})
```

Figure 49 : configuration de l'écoute du serveur node.js

Grâce à cette configuration présente sur le serveur Node.js, celui-ci écoute les requêtes arrivant sur ce port. Le serveur peut donc répondre aux requêtes selon les adresses renseignées du coté « Front-End »

Ensuite, une fois la requête envoyée et sa réponse reçue avec le tableau d'objets « cards », déjà vu, on égale un nouveau tableau d'objet, appelé, lui aussi, « cards », présent sur le « Front-end » à la valeur du « cards » envoyé par le serveur Node.js (« this.cards=res.data.cards »).
En cas d'erreur dans cet envoi de requête et lors de manque de réponse du serveur, un message d'erreur, comprenant ladite erreur, est envoyé à l'écran.

Ce genre de configuration est effectuée autant de fois que des paquets d'informations différents sont reçus sur l'une des pages de la « Single Page Application ».

Par contre, ce tableau d'objets « cards » n'est présent que sur la page sur laquelle il a été appelé. Si nous voulons le réutiliser ailleurs sur le « Front-End », il faut passer cet objet en paramètre de l'appel de la page ou du composant.

3.6.3.Affichage des cartes

Avec ces données présentes sur la page, nous avons pu commencer à les traiter et à les afficher. Mais comme chaque entrée (ligne) de la base de données comprend les informations de chaque député au sein du groupe politique ECR. Nous devons donc faire en sorte d'afficher ces personnes sous forme de cartes, et ce, dynamiquement, afin de pouvoir ajouter, ou retirer, des députés, aisément, sans devoir changer une quantité importante de code dans l'application.

Pour ce faire Nuxt.js, propose de créer des composants afin de pouvoir réutiliser un même élément plusieurs fois. Les codes de ces éléments sont donc placés dans un dossier « Composants », ce qui facilite l'appel à ces codes.

L'affichage des différentes cartes se fait par ce biais. Donc, nous avons dû créer un composant, appelé « Card [9]», qui affichera certaines des données d'un député.

[9] Il faut bien différencier le tableau d'objets « cards », le composant « Card » et l'objet « card ».

Ce composant n'a pas la possibilité de lire le tableau d'objets « cards » présent sur la page. L'appel de ce composant doit donc se faire avec un paramètre lui passant le tableau en question.

Pour que les cartes de chaque député au sein du groupe politique ECR soient affichées, nous avons besoin de mettre en place une boucle au niveau de l'affichage.

Le code de l'appel de ce composant :

```
<Card
  v-for="card in cards"
  :card="card"
  :key="card.persid"
  class="rounded-lg bg-gray-400 p-4"
/>
```

Figure 50 : Appel du composant Card

Grâce à Nuxt.js, l'appel des composants se fait simplement en mentionnant le nom du composant (tant que le code de celui-ci se trouve dans le dossier composant du projet).

On voit effectivement que l'instruction « v-for » est présente dans ce code. Cette instruction Vue.js permet de faire une boucle « for ». Ce genre de boucle est une instruction commune a bien des langages de programmation et permet de faire des itérations.
Ici, sa version « For-in » est utilisée. Elle sert simplement à faire une boucle qui parcourra tous les éléments présents dans le tableau « cards ». Un nouvel objet « card » prendra la valeur de l'élément suivant du tableau « cards » à chaque itération.

Ensuite, on retrouve le tag «: card = "card" » qui passe ledit objet au composant.
Et le tag « :key= "card.persid" » qui permet de donner un identifiant unique à chaque objet « card ».

Le champ « class », ici présent, ajoute simplement des éléments de style pour rendre l'affichage des cartes plus agréable visuellement.

Une fois les informations de l'objet « card » passés au composant « Card », celui-ci peut les traiter et les afficher comme nous le souhaitons.

L'affichage de ces données se fait, par exemple, comme suit :

```
<p class="mx-3 text-left  font-semibold text-gray-900 ">
    {{card.title}} {{card.lastname}}

  <span class="text-sm text-gray-600 text-opacity-75 ">
      <br>
  {{card.firstname}}
  </span>
</p>
```

Figure 51 : Exemple d'affichage des données

Le champ « class » sert encore une fois à ajouter des éléments de style à l'affichage des données pour les rendre plus agréables visuellement.

La balise «   » sert simplement à dire que c'est un paragraphe.
Dans celui-ci on retrouve les champs « {{card.title}} » et « {{card.lastname}} ».
Cette façon d'écrire permet d'afficher, à l'écran, la valeur de la variable citée entre les "{{ }}"

Ici, les différentes variables ont une nomenclature spécifique. En effet celles-ci sont relatives à l'objet initial « card ». Cet objet comprend bien des variables (on peut les retrouver au niveau de la requête SQL). Ces variables sont donc accessibles depuis l'objet « card » en spécifiant la variable en question. Pour afficher le titre de la personne stocké dans l'une des cartes, nous écrirons « card.title ». Et ce pour toutes les variables se trouvant dans « card ».

La balise «   », quant à elle, n'a pas une spécificité d'utilisation. Elle sert simplement à grouper des éléments pour qu'ils puissent, par exemple, avoir le même attribut « class ».

La balise «
 » est une balise qui ne se ferme pas, au contraire des deux balises précédemment citées, et intègre simplement un saut de ligne dans le texte.

Pour finaliser la carte, nous avons décidé d'ajouter une photographie du député, si nous disposons de celle-ci. Dans ce but, chaque image disponible a été stockée au sein d'un même dossier. Et chaque image a été renommée par le login du député.

On affiche cette photographie grâce à ce code :

```
<img
v-if="card.aspicture"
class=" w-12 h-12 rounded-full object-cover mr-1 shadow"
:src="require(`~/assets/images/photos/${card.login}.jpg`)"
alt="avatar">
```

Figure 52 : Affichage de la photographie du député.

Cet affichage se fera grâce à la balise « <img> ». Seulement, comme dit précédemment, chaque député n'a pas forcément de photographie lui étant liée. Pour tester cela nous utilisons l'instruction « v-if ».
Cette instruction permet de poser une condition sur l'affichage du bloc. Dans ce cas-ci, la condition est la variable « card.aspicture ». Si cette variable est égale à vrai ou est supérieure à zéro, alors le bloc sera affiché.

Si ledit député a bien une image qu'il lui est associée, alors nous allons la chercher grâce au tag « :src ». Ce tag permet à la balise « <img> » de savoir où aller chercher l'image. Dans ce cas-ci, comme l'image doit être affichée dynamiquement selon le député, nous devons intégrer une variable dans la source. C'est pour cela que la fonction « require() » est utilisée ici.

Le tag « alt » permet de remplacer, en cas de problème d'affichage de l'image, celle-ci par un texte prédéfini.

Nous avons décidé de faire apparaître une image par défaut si l'un des députés n'a pas de photographie lui étant associée. Ces images par défaut étant genrées, nous avions donc deux possibilités qui s'offraient à nous.

Soit le député était de genre féminin, donc l'image par défaut était de genre féminin.
Soit le député était de genre masculin, donc l'image par défaut était de genre masculin.

En voici le code associé :

```
<img
v-else-if="card.title === 'Mrs' || card.title === 'Miss' "
class=" w-12 h-12 rounded-full object-cover mr-1 shadow"
src="~/assets/images/photos/mrs.jpg"
alt="avatar">
<img
v-else
class=" w-12 h-12 rounded-full object-cover mr-1 shadow"
src="~/assets/images/photos/mr.jpg"
alt="avatar">
```

Figure 53 : Affichage de la photographie genrée par défaut

Si la réponse du test précédent « v-if="card.aspicture" » est « faux » ou égal à zéro alors l'instruction « v-else » est appelée. Ici comme un test supplémentaire est nécessaire afin de déterminer le genre du député, ce n'est pas cette instruction qu'on a utilisée mais l'instruction « v-else-if » qui permet de refaire un test si le précédent est non concluant.

Ce test s'effectue maintenant sur le titre de la personne. Si ce titre est « Mrs » ou « Miss » alors l'image associée et répertoriée dans le tag « src » est l'image par défaut de genre féminin.
Sinon l'image par défaut de genre masculin est utilisée.

Un aperçu du rendu visuel de ces cartes avec les valeurs d'affichage par défaut :

Figure 54 : Aperçu des cartes

3.6.4.Page « détails »

En plus d'afficher les photographies et les informations de chaque député sur cette carte, nous avons lié une nouvelle page, qui donnera plus de détails sur le député en question.

Pour ce faire nous avons utilisé l'outil « d'autoRouting ». Comme cette page ne comprend pas les mêmes informations, vu que chaque carte est différente, il faut, en plus de passer la page, passer également des paramètres. Ceux-ci permettront de différencier une carte d'une autre et donc, un député d'un autre.

Cette redirection est faite comme suit :

```
<NuxtLink
:to="{name:'Details', params: { persid: card.persid}}"
class="bg-gray-200 min-w-full rounded-lg shadow">
  <div class="flex items-center "> ⋯
  </div>
  <div class="rounded-full">⋯
  </div>
</NuxtLink>
```

Figure 55 : Navigation vers la page de détails

Nous retrouvons la balise « d'autoRouting » « NuxtLink » avec ses paramètres comme expliqué ci-dessus. Ici on redirigera, lors du clic sur une carte, l'utilisateur vers le lien ayant comme nom « Details ». De plus, avec le paramètre « persid », on envoie la variable « card.persid » à la page pour afficher les informations de la bonne carte.

Une fois la page de détail appelée, celle-ci doit aller chercher, sur le serveur, des informations complémentaires sur le député en question. Cette requête se fait comme les précédentes. Cependant l'URL visée et les paramètres envoyés ne sont pas les mêmes.

```
let request = await axios
.post('http://localhost:8080/Details',{
    persid : this.$nuxt._route.params.persid
}).then(res => {
this.card = res.data.card
}
).catch(err =>console.log(err.message))
```

Figure 56 : Requête au serveur Node.js de la page de détails

Ici l'URL pointe toujours vers l'adresse du serveur Node.js mais sur le lien « /Details »
Le paramètre passé à cette requête est également la valeur de la variable « persid » passé lors de la redirection via « NuxtLink ».

Du coté Node.js, la configuration est partiellement identique à celle sur l'URL « /ListPerson ».

```
app.post('/Details', async(req,res)=>{
  try{
      const { persid } = req.body

      console.log("values of persid :" + persid)
      const query = {
          text :
          `SELECT *
          FROM person
          INNER JOIN salutation ON person.salutid=salutation.salutid
          INNER JOIN countries ON person.nationality=countries.shortcut
          WHERE persid=$1`,
          values : [persid]
      }
      const result = await pool.query(query)

      return res.status(200).json({
          card : result.rows[0]
      });

  }catch(err){
      console.log(err.message)
  }
})
```

Figure 57 : Configuration de l'écoute de la requête /Details

Ici l'URL touchée est « /Details » et le paramètre récupéré est « persid ».

La requête SQL change également. Celle-ci va aller chercher toutes les informations (caractérisé par l'astérisque) présentes dans la base de données, pour le député ayant comme valeur de « persid », la valeur passée.

Un aperçu du rendu visuel de cette page de détails :

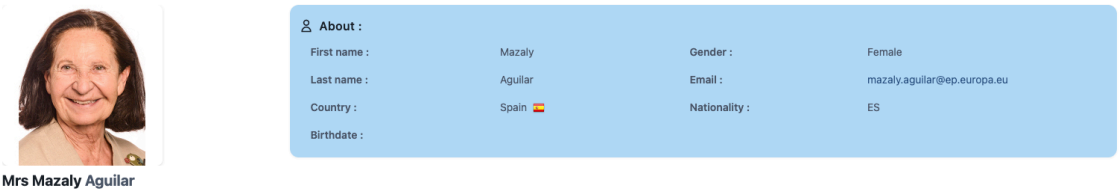

Figure 58 : Aperçu de l'une des pages de détails

3.6.5.Barre de navigation.

Après cela, nous avons décidé de créer un composant « NavBar » qui sera une barre de navigation commune à la majorité des pages du site. Cette barre de navigation comprendra le logo du groupe ECR et des boutons pour pouvoir naviguer entre les différentes pages. Le bouton « Home » qui permettra de revenir à la page d'accueil. Le bouton « Log in / Sign in » qui permettra de rediriger vers une page d'authentification ou de création de compte quand celui-ci sera mis en place. Et un bouton « Log out » pour se déconnecter.

Les différentes redirections sont faites grâce à « l'autoRouting » de Nuxt.js avec les balises « <NuxtLink> ».

La barre de navigation sera présente sur la page d'accueil et la page de détails mais absente de la page d'authentification. Cette barre de navigation se présente comme suit :

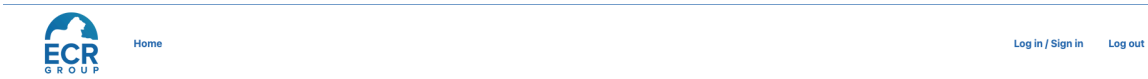

Figure 59 : Aperçu de la barre de navigation

3.6.6.Recherche dans les cartes

Une autre fonctionnalité a été mise en place sur le site. En effet, nous avons pensé qu'une possibilité de recherche d'un député ou d'une liste de députés par leurs nom ou prénom pourrait être pratique.

Dans ce but nous avons dû styliser une zone d'entrée de caractères pour effectuer cette recherche.

Figure 60 : Aperçu de la barre de recherche

Cela a été majoritairement fait grâce aux tags « class » et un import d'images sous format « svg ».

Par contre, la recherche en elle-même se fait grâce à l'appel d'une fonction.

Voici la configuration de la zone d'entrée :

```
<input
  class="h-10 py-2 pr-12 pl-2 rounded-full text-xl focus:outline-none"
  placeholder="Search"
  v-model="search"
  @input ="newsearch()"
>
```

Figure 61 : Configuration de la zone d'entrée

La balise <input> est une balise qui permet à l'utilisateur de saisir des données.
Le tag « placeholder » sert à afficher du texte par défaut lorsque l'utilisateur ne renseigne rien dans la zone d'entrée.

L'instruction « v-model » permet de stocker dans une variable, ici « search », le texte saisi dans la zone d'entrée.

Le champ « @input » est un tag d'évènement. Chaque fois que l'utilisateur va écrire quelque chose dans la zone d'entrée, la fonction « newsearch() », déclenchée par l'événement « @input », va être appelée.

La fonction est la suivante :

```
async newsearch(){
    let mysearch

    if(this.search){
            mysearch =this.search
      }else{
            mysearch = '%'
    }
    let request= await axios
      .post('http://localhost:8080/ListPerson',{
            search : mysearch,
            offset : this.offset,
            pageSize: this.pageSize
      }).then(res => {
      (this.cards = res.data.cards)
      }
      ).catch(err =>console.log(err.message))
 },
```

Figure 62 : Code de la fonction Mysearch

Lors de l'appel de cette fonction asynchrone, une variable nommée « mysearch » est instanciée. Ensuite, un test sur la variable « search » présente dans l'instruction « v-model » est effectué : la variable « search » existe-t-elle ?
Si c'est le cas, cela veut dire que l'utilisateur a déjà inscrit quelque chose dans la zone de texte. Auquel cas, ce texte est enregistré dans la nouvelle variable « mysearch ».
Si « search » n'existe pas, alors la variable « mysearch » vaudra le caractère « % » (n'importe quel caractère ou groupe de caractères).

Ensuite, la variable « mysearch » est envoyée dans la requête au serveur Node.js comme vu précédemment.

Sur le serveur Node.js, la variable est récupérée et est utilisée dans la requête SQL.
Voici la requête SQL en question :

```
const query = {
    text :
       `SELECT persid,login,firstname,lastname,aspicture,nationality,salutation.title
       FROM person
       INNER JOIN salutation ON person.salutid=salutation.salutid
       WHERE CONCAT(firstname,' ',lastname) ILIKE $1
       ORDER BY lastname
       OFFSET $2
       LIMIT $3
       `,
     values : ['%'+search+'%',offset,pageSize]
}
```

Figure 63 : Requête SQL de la page d'accueil

On retrouve, dans la fonction« WHERE » une concaténation du prénom et du nom et une fonction « ILIKE » suivit d'un « $1 » ($1 prend la valeur de la première variable présente dans le champ « values »).
Cette ligne consiste à préciser sur quel paramètre la sélection dans la base de données va être effectuée. Ici on peut traduire cela par : « Où "prénom + nom" contient une suite de caractères égale à la variable "search" »

Par exemple si le nom et prénom sont « Mazaly Aguilar », la concaténation des deux feront « MazalyAguilar». Si la variable « search » équivaut à toute suite de caractères qui compose cette concaténation, alors les données de Mrs Mazaly Aguilar seront envoyées.

3.6.7.Pagination

Au départ, un nombre fixe de cartes était affiché à l'écran. Ce nombre délibérément choisi était 16. Ce qui permettait d'avoir quatre cartes par rangée et par colonne. Seulement, la base de données contenant 48 députés nous ne pouvions les voir tous. Il a alors fallu mettre en place un système de pagination. Avec cette navigation entre pages, il était souhaitable qu'une sélection du nombre de cartes par page soit aussi mise en place.

Nous avons d'abord commencé par mettre en place quelques boutons fixes pour pouvoir accéder à ces pages contenant différentes cartes.

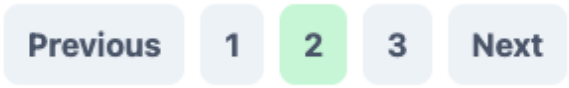

Figure 64 : Affichage de la pagination

L'affichage des cartes se fait sur la totalité des cartes présentes dans l'objet « cards » envoyé par le serveur Node.js. Le serveur, lui, récupère ces cartes grâce à la requête SQL.

Il faut donc limiter la récupération des données de la requête SQL pour limiter le nombre de cartes à l'affichage. Cette fonction se nomme « LIMIT » dans cette requête.

De plus, pour afficher les cartes à partir du deuxième lot de cartes (lorsqu'on navigue sur la deuxième page), il faut spécifier à la requête à partir de quelle carte il commencer à les récupérer. Cette deuxième fonction se nomme « OFFSET » dans la requête SQL.

Il faut donc agir sur la fonction « OFFSET » si on veut changer de page, celle-ci contenant également 16 cartes. En théorie, si la première page comporte 16 cartes (de 0 à 15), alors la deuxième commencera à « l'OFFSET » de valeur 16 et ainsi de suite avec les multiples de 16.

Un exemple sur le bouton nommé « Previous » :

```
<ul class="flex ml-10 mb-10">
  <li>
    <span v-if=" this.offset === 0 " disabled class="mx-1
        Previous
    </span>
  <span v-else @click="offsetter($event)">
    <NuxtLink :to=" {name:'HomePage'}"
      name="Prev"
      class="mx-1 px-3 py-2 bg-gray-200 text-gray-700
      >
      Previous
    </NuxtLink>
    </span>
  </li>
```

Figure 65 : code du bouton Previous et Next

Ici, la balise <ul> permet de créer une liste structurée d'éléments grâce aux balises <li>. Chaque bouton sera présent dans une balise <li> afin de structurer la liste des boutons.

Une variable « offset » est créée au niveau de la page « Home » dans le « Front-end » (code non présent sur cette image). Elle est ensuite passée au niveau du server Node.js grâce aux requêtes HTTP. Cette variable est ensuite introduite dans la requête SQL.

On retrouve à nouveau ici l'instruction « v-if » permettant de faire un test sur la valeur de cette variable « offset ». Ce test est une vérification de sa valeur. Si elle équivaut strictement à 0 alors le bouton est désactivé.

Sinon, un événement « @onclick » qui appellera la fonction « offsetter » est créé.
Cet événement permet, lors du clic, ici sur le bouton, d'appeler une fonction par exemple.
Ici, on passe également un paramètre à cette fonction. Ce paramètre « $event » permet de connaître sur quel bouton l'utilisateur a appuyé. Nous reviendrons sur cette fonction plus loin.

Ensuite, nous retrouvons une redirection vers le nom du lien « HomePage » et un attribut « name » qui est donné à ce bouton lors de la redirection.

Le code de ce bouton est identique au code du bouton « Next ».

Par contre, en vue de passer sur un affichage dynamique de bouton, il a été préférable de coder les boutons entre « Previous » et « Next » de façon dynamique.

Il a fallu, tout d'abord, connaître le nombre de pages qu'il y aurait sur le site. Chacune comprenant 16 cartes par page.

Nous devions connaître le nombre total de cartes. Pour ensuite diviser ce nombre par le nombre de cartes par page. Nous aurions ainsi notre nombre total de pages.

Le nombre total de cartes est susceptible de changer si l'un des députés devait quitter le groupe ECR ou que de nouveaux députés venaient à être insérés dans la base de données. Une nouvelle requête à cette base de données est donc requise afin de connaître le nombre exact de députés.

Cette requête s'initialise d'abord du coté « Front-end » comme-suit :

```
async InitPager(){
  let request= await axios
    .post('http://localhost:8080/CountPersons',{

    }).then(res => {
      (this.totalPersons=res.data.persons)
    }).catch(err =>console.log(err.message))
  this.numPages = this.totalPersons/this.pageSize
},
```

Figure 66 : Code de la fonction InitPager

L'URL est celle du server Node.js sur le lien « /CountPersons ».
La réponse sera introduite dans la variable, créée au préalable, « totalPersons ». Et cette variable est directement utilisée pour faire le calcul, expliqué ci-dessus, afin d'avoir le nombre de pages total.

Du côté serveur, la réception de la requête est identique aux autres expliquées précédemment. Cependant, la requête SQL change énormément de ce qui a déjà été expliqué :

```
const query = {
    text :
    `SELECT COUNT(*) numPersons
    FROM person`
}
const result = await pool.query(query);
return res.status(200).json({
  persons : result.rows[0].numpersons
```

Figure 67 : Requête SQL comptant le nombre de député

Ici, la fonction « SELECT COUNT() » permet de compter le nombre de lignes dans une table. En lui spécifiant le caractère astérisque « * », nous lui demandons de compter toutes les lignes présentes dans la table « person ». Ce résultat sera stocké dans une colonne appelée « numPersons ».
Une fois la requête SQL effectuée, la réponse est stockée dans la variable « persons » et est envoyée au « Front-end ».

Une fois ce nombre de pages récupéré sur la page d'accueil, l'affichage des boutons de redirections par pages, peut se faire.
Ces boutons se font comme-suit :

```
<li
  v-for="page in this.numPages"
  :key="page"
  >
  <span v-if="offset === ((page-1)*pageSize)"
    disabled class="mx-1 px-3 py-2 bg-green-200 text-gray-700
    {{page}}
  </span>

  <span v-else @click="offsetter($event)">
    <NuxtLink :to=" {name:'HomePage'}"
      class="mx-1 px-3 py-2 bg-gray-200 text-gray-700 hover:
      :name="page">
      {{page}}
    </NuxtLink>
  </span>
</li>
```

Figure 68 : Affichage des boutons dynamiques

On retrouve l'instruction « v-for » permettant d'afficher le nombre exact de boutons.

Un test sur la valeur de la variable « offset » est également présent. Ce test permet de savoir si cette valeur est égale au nombre de cartes sur une page multiplié par le numéro de la page moins 1. Avec ce test, on regarde si la page affichée n'est pas la page sur laquelle le bouton redirige. Auquel cas, ce bouton doit être désactivé et une indication visuelle doit être mise en place.

Si ce test n'est pas bon, alors un événement « @click » est créé. Cet événement appelle également la fonction « offsetter() » en lui passant la variable « $event ».

Ensuite, une redirection vers la page d'accueil est effectuée et un nom est donné à ce bouton. Son nom est donné grâce à la variable « page » présente dans l'instruction « v-for ». Celui-ci est un nombre compris entre 1 et le nombre total de pages.

Voici le code de la fonction « offsetter() » :

```
async offsetter(e){

  if(e.srcElement.name=='Prev'){
      this.offset=this.offset-this.pageSize
  }
  if(e.srcElement.name=='Next'){
      this.offset=this.offset + this.pageSize
  }
  for (let page = 1; page <= this.numPages; page++) {
      if(e.srcElement.name==page){
        this.offset=((page-1)*this.pageSize)
    }
  }
  this.newsearch()
}
```

Figure 69 : Code de la fonction offSetter

On retrouve, encore une fois, une fonction asynchrone.
Celle-ci reçoit un paramètre comprenant les informations relatives au bouton sur lequel l'utilisateur a cliqué.

Cette fonction comprend trois tests qui ont pour but de donner la bonne valeur à la variable « offset ». Ces tests sont effectués sur le nom des boutons. Il y a trois possibilités suite aux explications données ci-haut.

La première, le nom est égal à « Prev », nom donné au bouton « Previous ». Auquel cas, une simple soustraction du nombre de cartes par pages à la variable « offset » est opérée. Cela nous donnera la page précédente.

Deuxième cas, le nom testé est égal à « Next », le nom du bouton « Next ». Alors une addition du nombre de cartes par pages et de la valeur de la variable « offset » est effectuée. Cela nous affichera la page suivante.

Troisième cas, le nom est un nombre compris entre 1 et le nombre total de pages. Pour effectuer ce test une boucle est nécessaire. On retrouve ici une instruction « for » afin de tester tous les noms possibles dans ce cas. Si le nom du bouton cliqué est égal à l'une de ces possibilités, la valeur de la variable « offset » sera modifiée. Cette valeur sera égale à la valeur du bouton cliqué, diminué de 1, multiplié par le nombre de cartes par page.
Ce troisième cas nous affichera la page du bouton cliqué.

Un exemple : si le nombre de cartes par page égale 16 et que le bouton cliqué est le troisième, le calcul est le suivant : (3-1) x 16 ce qui donne un « offset » de 32.

Pour éviter tout débordement sur la valeur de la variable « offset », une fonction « offsetSecure() » a été mis en place.

Cette fonction a pour seul but d'égaler cette valeur à zéro lorsque celle-ci descend dans les négatifs. Et de l'égaler à sa valeur maximale lorsque celle-ci la dépasse.

Cela se fait comme-suit :

```
async offsetSecure(){

    if( this.offset < 0){
        this.offset = 0
    }
    if(this.totalPersons!=0){
      if( this.offset > this.totalPersons-this.pageSize){
          this.offset = this.totalPersons-this.pageSize
    }
    }
  },
```

Figure 70 : Code de la fonction offsetSecure

Lorsque la valeur de la variable « offset » est strictement inférieure à zéro, celle-ci est égalée à zéro.

Le deuxième test doit s'effectuer lorsque la requête permettant de connaître le nombre de députés a été envoyée (quand la valeur de totalPersons n'est plus égale à zéro). Une fois que cela est vérifié, alors, nous testons si l'offset est strictement supérieur au nombre maximal. Ce nombre est défini par la soustraction du nombre de cartes affichées par page du nombre total de députés. Si ce nombre est dépassé alors la valeur de la variable « offset » est égalée à cette valeur.

Ensuite, il a fallu mettre en place la sélection du nombre de cartes par page.

Des boutons fixes ont donc été créés. Ceux-ci resteront fixes et prendront la valeur souhaitée par le développeur. Ici, les quatre valeurs ont été choisies délibérément. Il s'agit de 4, 8, 12 et 16 cartes par page.

Ces boutons sont codés de la même façon que pour la pagination à l'exception de la fonction appelée « paginer() » et non « offsetter() ». Un paramètre comprenant l'événement a été également passé à la variable.

```
async paginer(e){
  this.offset=0
  this.pageSize=1*e.srcElement.id
  this.newsearch()
},
```

Figure 71 : Code de la fonction paginer

Cette fonction, aussi asynchrone, permet de remettre la valeur de « l'offset » à zéro. Et d'égaler le nombre de cartes à afficher par page à la valeur de l'id du bouton cliqué.

Il est à noter que, l'id du bouton a été au préalable égalé à la valeur du bouton.

La fonction « newsearch() » visible souvent en fin de fonction est la fonction permettant l'envoi de la requête sur « /ListPerson » vu au point « Single Page Application / Front-End » de ce chapitre.

Pour finaliser ce point, cette fonction « newsearch() » appelle également les fonctions « InitPager() » et « offsetSecure() » vues précédemment, avant d'envoyer la requête au serveur.

3.6.8.Passage sous Universal Rendering

Une fois toutes ces fonctionnalités ajoutées au site, nous pouvions le passer sous un environnement dit *Universal Rendering*. Il a fallu recommencer un nouveau projet sous Nuxt.js afin de s'assurer que celui-ci prenne bien les paramètres d'un site sous *Universal Rendering*.

Dans cette configuration, le serveur Node.js est interne au projet. Le projet Nuxt.js intègre totalement le serveur Node.js, celui-ci étant lancé en même temps que l'application. Ainsi le « Font-End » et le « Back-end » sont lancés simultanément au sein du même projet. L'avantage de cette configuration est qu'il n'est plus nécessaire d'avoir plusieurs URL indépendantes entre le « Front-End » et le « Back-end ». En effet, dans la configuration précédente, le serveur tournait, dans notre cas, sur le port 8080 de l'adresse locale de l'ordinateur. Et la partie « SPA » tournait, quant à elle, sur le port 3000 de cette même adresse. Avec cette nouvelle configuration, les deux parties tournent sur le même port qui est le port 3000 de l'adresse locale de l'ordinateur utilisé.

Notre configuration de l'application dans ce mode de *Rendering* permet également un export des données vers l'extérieur. C'est-à-dire que les données sont également accessibles via une adresse indépendante de la partie « SPA ». Cette exportation se fait sur l'adresse « /api » du port 3000 de l'adresse locale. L'URL complète pour y accéder est la suivante : http://localhost:3000/api.

Cette adresse n'est accessible que sur la machine qui héberge l'application. En production, cette adresse ne sera plus en local mais il restera toujours « /api » en fin d'adresse.

Cette exportation vers l'extérieur se fait grâce à ce code :

```
export default {
  path: '/api',
  handler: app
}
```

Figure 72 : Exportation /api

Ici, nous pouvons renseigner l'adresse d'export de ces données en format Json.

Grâce à cet export de données, le code reste presque inchangé.

Comme l'adresse d'accès aux données a changé, nous devions également modifier l'adresse des requêtes envoyées depuis la partie « Front-End ». Celles-ci prennent la nomenclature suivante :

```
.post('http://localhost:3000/api/ListPerson',{
```

Figure 73 : Import des données /ListPerson

```
.post('http://localhost:3000/api/Details',{
```

Figure 74 : Import des données /Details

```
.post('http://localhost:3000/api/CountPersons',{
```

Figure 75 : Import des données /CountPersons

3.6.9.Sessions et cookies

Un ajout de fonctionnalité a été proposé par le maître de stage ; il s'agit d'un système de sessions et de cookies. Ceci permettra d'initier un système d'authentification des utilisateurs par la suite.

Ce système se configure comme-suit :

```
app.use(session({
    store: new pgStore({
        pool:pool,
        tableName: 'user_sessions'
    }),
    cookie: {
        maxAge: 1000 * 30,//60 * 60 * 24
        httpOnly: true,
    }
}))
```

Figure 76 : configuration des sessions et cookies

Grâce à la librairie « express-session » nous avons la possibilité d'ajouter des sessions au sein de chaque réponse aux requêtes reçues.
Ces sessions sont également stockées dans notre base de données dans une table s'appelant « user_sessions ».

Ces sessions permettent d'ajouter à ces réponses des cookies. Ces cookies nous donnent, notamment, la possibilité de donner une date d'expiration à ces sessions.

L'avantage des sessions est qu'elles permettent une persistance de l'information propre à cet utilisateur, à cette session, côté serveur. Le web étant par définition « stateless » (sans état), cette persistance n'est pas possible sans session.

3.6.10.Site complet

La page d'accueil du site, avec tous ces ajouts, ressemble actuellement à ceci :

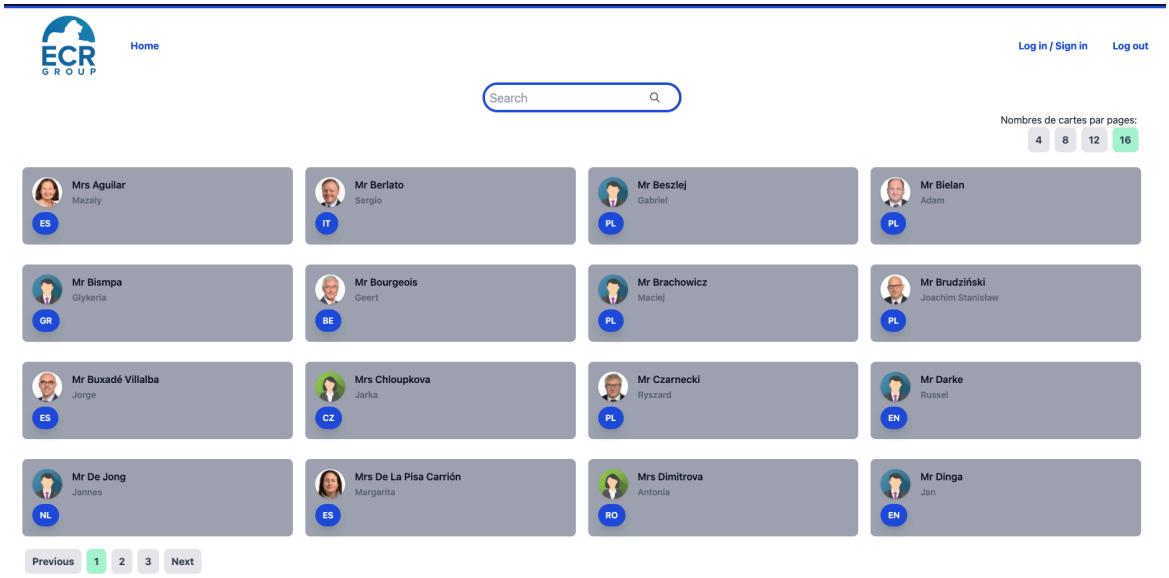

Figure 77 : affichage de la page d'accueil au complet

Lors du clic sur l'une des cartes, la page de détails s'affiche comme suit :

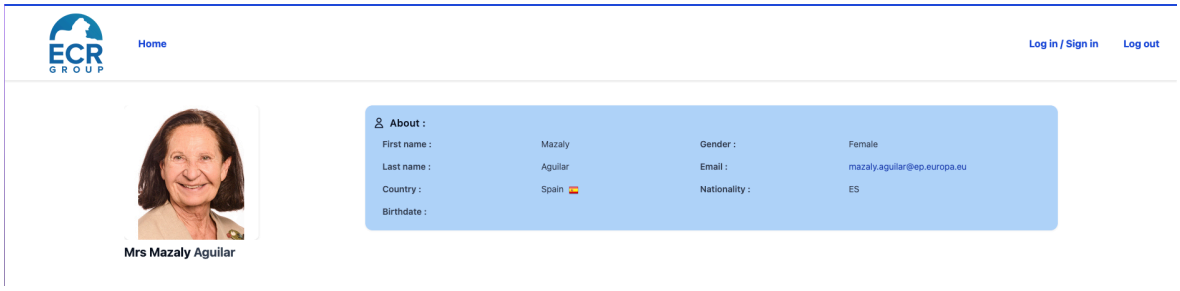

Figure 78 : Affichage de la page de détails au complet

Et lors de l'accès à la page de connexion, cette page, ci-dessous, s'affichera :

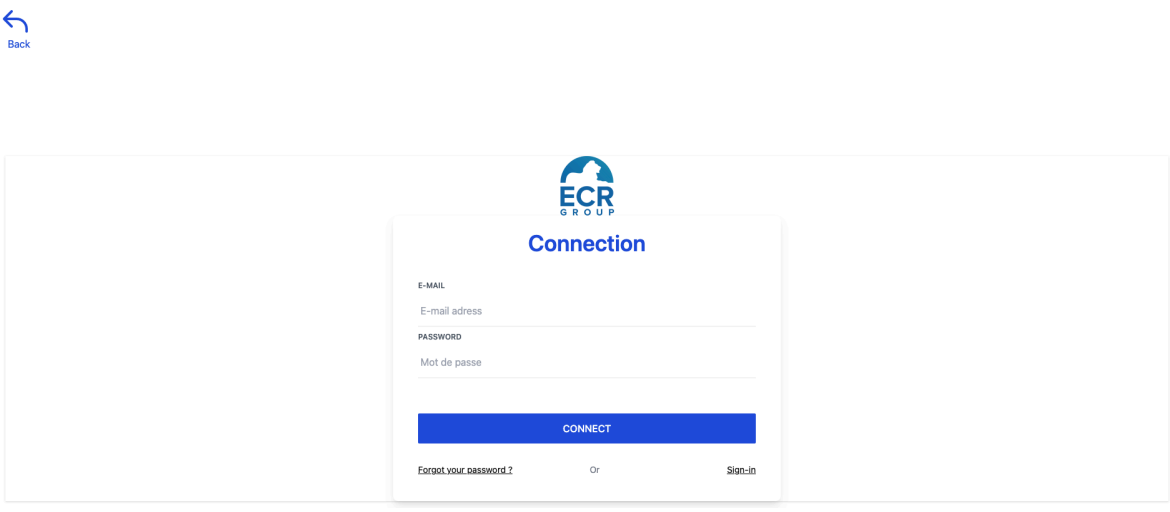

Figure 79 : Affichage de la page statique de connexion

3.7.Alternatives

Des alternatives à l'utilisation de Nuxt.js sont disponibles tel que Next.js, Nestjs, …

Next.js est la version « Universal » de la bibliothèque « React ». Ce framework est créé par « Vercel » avec l'aide de Google et de Facebook. Next.js est utilisé par de célèbres sites tel que Netflix, Nike, TikTok, … La liste n'est, bien sûr, pas exhaustive.

Nestjs est la version « Universal » de la bibliothèque Angular. Ce framework est créé par Google. Nestjs est utilisé par de célèbres sites tel que Adidas, Autodesk, Décathlon, … La liste n'est, là non plus, non exhaustive.

Toutes ces alternatives ont leurs avantages et leurs inconvénients. Mais le choix de Nuxt.js fut imposé par le maître de stage. La question de quel framework utiliser ne s'est pas posée.

4.Conclusion

En conclusion, lors de ce stage, j'ai dû approfondir des concepts techniques déjà vus lors de ma formation, m'intéresser à d'autres concepts techniques non vus et les étudier afin de les mettre en pratique. Enfin, apprendre à utiliser une nouvelle technologie, Nuxt.js, dont les avantages sont indéniables.

L'objectif initial de ce stage a été atteint. C'est-à-dire, réaliser un POC (proof of concept) sur base d'une application ayant pour vocation de montrer la faisabilité et l'efficacité de cette nouvelle technologie.

Le serveur Node.js est effectivement intégré à Nuxt.js et est efficace dans le cadre de ce projet.

Cette technologie a permis de récupérer le code d'origine d'une application sous Vue.js réalisé au préalable par mes soins et de l'implémenter dans sa version finale sous Nuxt.js.

Ce stage m'a également permis de développer des compétences non techniques importantes pour la gestion de tout projet. En effet, la situation sanitaire m'ayant obligé à travailler à distance, il a fallu mettre en place une organisation rigoureuse pour garantir le bon déroulement de cette expérience professionnelle particulière.

De plus, de fréquentes interactions ont été nécessaires afin de s'assurer que le travail réalisé corresponde aux attentes des personnes impliquées.

Au final, ce stage m'aura permis d'avoir un aperçu des défis qui m'attendent à l'avenir. Grâce à ma formation technique, je suis confiant en mes capacités à y faire face.

5.Médiagraphie

Abgrall, F. (s.d.). *Demystifying SSR, CSR, universal and static rendering with animations.* Consulté en Mai 2021, sur Dev.to: https://dev.to/kefranabg/demystifying-ssr-csr-universal-and-static-rendering-with-animations-m7d

Alves, R. (s.d.). *What Is a Single Page Application (SPA)?* Consulté en Mai 2021, sur Outsystems: https://www.outsystems.com/blog/posts/single-page-application/?utm_source=google&utm_medium=cpc&utm_campaign=Awareness_G_GBL_Search&utm_term=single%20page%20application&utm_content=awareness&gclid=CjwKCAjwnPOEBhA0EiwA609RecQ2Pe4WKeLshvbLJB6WcfQ6XSzTf61uUeSh6ud7UvysIgT07n7IAxoCjQUQAvD_BwE

Barnard, J. (s.d.). *Qu'est-ce que le SEO ?* Consulté le Mai 2021, sur SemRush Blog: https://fr.semrush.com/blog/definition-seo-guide-2020-debutants/?kw=&cmp=FR_SRCH_DSA_Blog_Core_BU_FR&label=dsa_pagefeed&Network=g&Device=c&utm_content=486542000146&kwid=aud-296306606820:dsa-1100351999444&cmpid=11849486850&agpid=113156852777&BU=Cor

FreeCodeCamp. (s.d.). *Client-side vs. server-side rendering: why it's not all black and white.* Consulté en Mai 2021, sur FreeCodeCamp: https://www.freecodecamp.org/news/what-exactly-is-client-side-rendering-and-hows-it-different-from-server-side-rendering-bd5c786b340d/

httpbin.org. (s.d.). *httpbin.org.* Consulté le Mai 2021, sur httpbin.org: http://httpbin.org/

Malekal.com. (s.d.). *QU'EST-CE QUE JAVASCRIPT.* Consulté en Mai 2021, sur Malekal.com: https://www.malekal.com/javascript/

Mozilla. (s.d.). *HTTP request methods .* Consulté en Mai 2021, sur MDN WebDocs: https://developer.mozilla.org/fr/docs/Web/HTTP/Methods

NodeSource. (s.d.). *Choosing the right Node.js Framework: Next, Nuxt, Nest?* Consulté en Mai 2021, sur The NodeSource Blog: https://nodesource.com/blog/next-nuxt-nest/

Omoyeni, T. (s.d.). *Differences Between Static Generated Sites And Server-Side Rendered Apps.* Consulté en Mai 2021, sur Smaching Magazine: https://www.smashingmagazine.com/2020/07/differences-static-generated-sites-server-side-rendered-apps/

Osmani, J. M. (s.d.). *Rendering on the web.* Consulté en Mai 2021, sur Developpers google: https://developers.google.com/web/updates/2019/02/rendering-on-the-web

Parlement européen. (s.d.). *Qu'est-ce que le Parlement européen?* Consulté en Mai , 2021, sur Le Parlement européen: https://www.europarl.europa.eu/news/fr/faq/16/qu-est-ce-que-le-parlement-europeen

SEO.fr. (s.d.). *Définition du SEO (Search Engine Optimisation).* Consulté en Mai 2021, sur SEO.FR: https://www.seo.fr/definition/seo-definition

Toute l'Europe. (s.d.). Consulté en Mai 2021, sur Toute l'Europe: https://www.touteleurope.eu/fileadmin/_processed_/3/2/Screenshot_2019-07-02_Accueil_Resultats_des_elections_europeennes_2019_Parlement_europeen-04856b33fd.png

Wikipédia. (s.d.). *Ajax (informatique).* Consulté en Mai 2021, sur Wikipédia: https://fr.wikipedia.org/wiki/Ajax_(informatique)

Wikipédia. (s.d.). *Conservateurs et réformistes européens.* Consulté en Mai 2021, sur Wikipédia: https://fr.wikipedia.org/wiki/Conservateurs_et_réformistes_européens

Wikipédia. (s.d.). *Document Object Model.* Consulté en Mai 2021, sur Wikipédia: https://fr.wikipedia.org/wiki/Document_Object_Model

Wikipédia. (s.d.). *Hypertext Transfer Protocol.* Consulté en Mai 2021, sur Wikipédia: https://fr.wikipedia.org/wiki/Hypertext_Transfer_Protocol

Wikipédia. (s.d.). *Institutions de l'Union européenne.* Consulté en Mai 2021, sur Wikipédia: https://fr.wikipedia.org/wiki/Institutions_de_l%27Union_européenne

Wikipédia. (s.d.). *Nuxt.js.* Consulté en Mai 2021, sur Wikipédia: https://fr.wikipedia.org/wiki/Nuxt.js

Wikipédia. (s.d.). *Optimisation pour les moteurs de recherche.* Consulté en Mai 2021, sur Wikipédia: https://fr.wikipedia.org/wiki/Optimisation_pour_les_moteurs_de_recherche

6.Lexique

A

B

Browser : Navigateur internet.

C

C.S.R : Rendu côté client
CSS : Langage de styles.

D

E

E.C.R : Groupe des Conservateurs et Réformistes européens
Eurozone : Les états membres qui ont adopté l'euro comme monnaie officielle

F

G

H

HTML : Langage de balises
HTTP : Hypertext Transfer Protocol

I

I.D : Groupe « Identité et Démocratie »

J

JSON : JavaScript Object Notation

K

L

La Gauche : Groupe de la Gauche au parlement européen

M

N

O

O.D.S : Parti Démocratique Civique

P

P.E : Parlement européen
P.P.E : Groupe du Parti Populaire Européen
P.P.E-D.E : Parti Populaire Européen et des Démocrates Européens

Q

R

Renew : Renew Europe Group

S

S&D : Groupe de l'Alliance Progressiste des Socialistes et Démocrates au Parlement européen
S.P.A : Application à page unique
S.S.R : Rendu côté serveur

T

U

U.E : Union européenne

URL : Uniform Resource Locator

V

Verts/ALE : Groupe des Verts/Alliance Libre Européenne

W

X

Y

7.Annexes

Figure 6 : Exemple de requête GET via Telnet

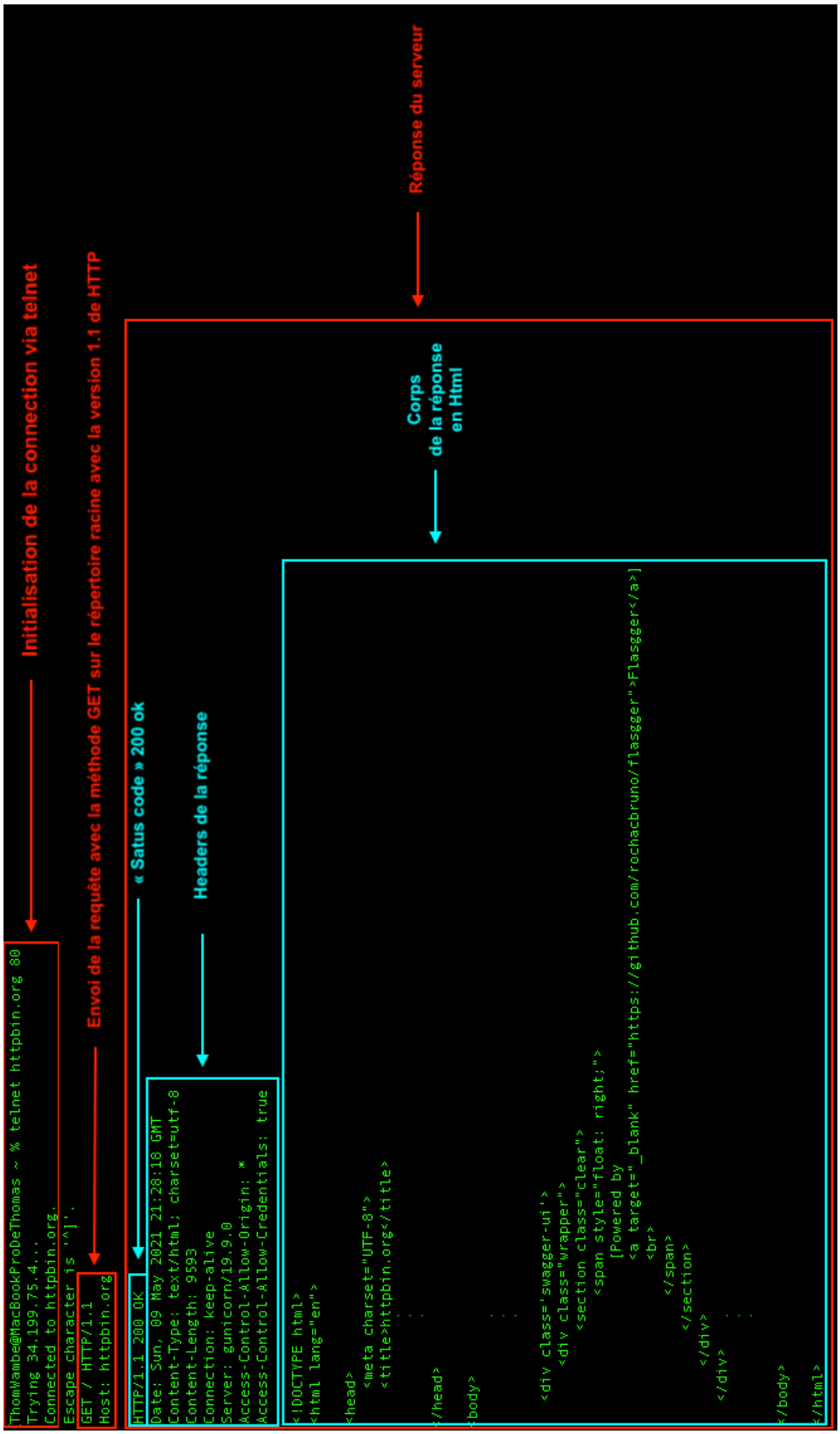

Figure 7 : Exemple de requête HEAD via telnet

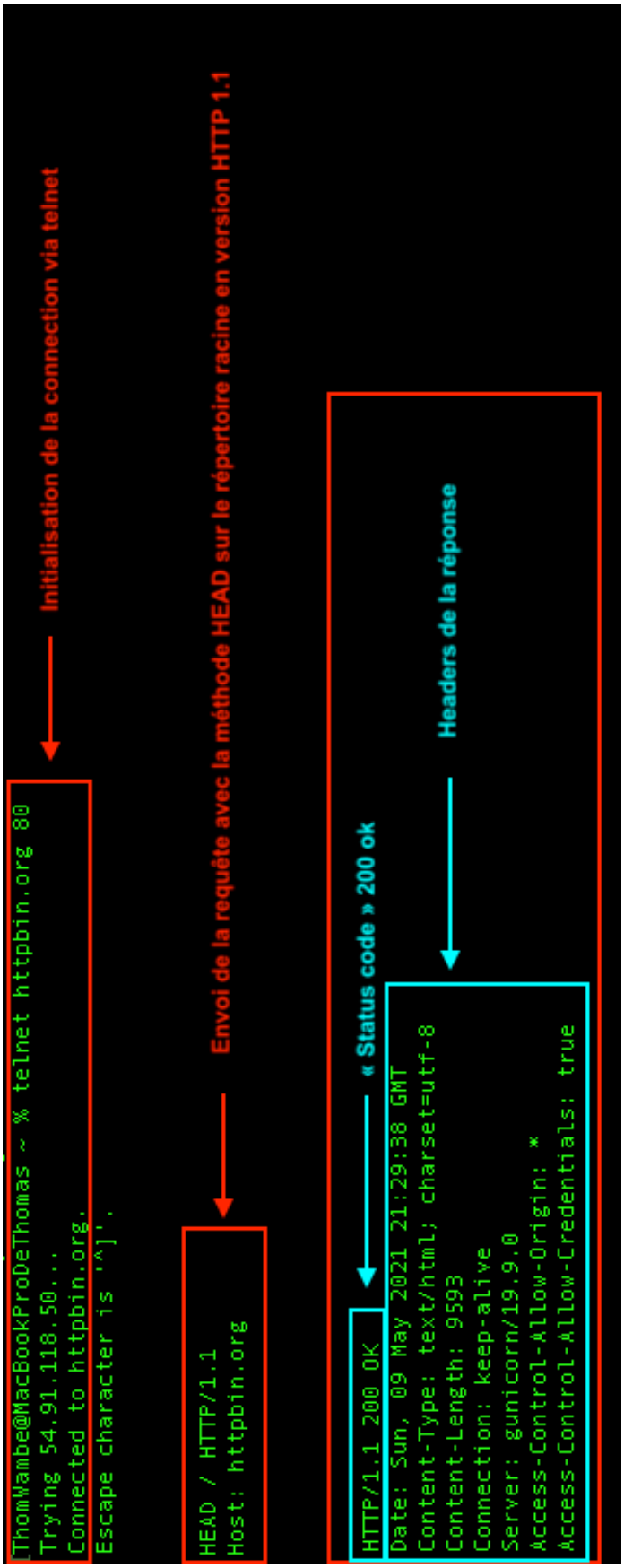

Figure 8: Exemple de requête POST via Telnet

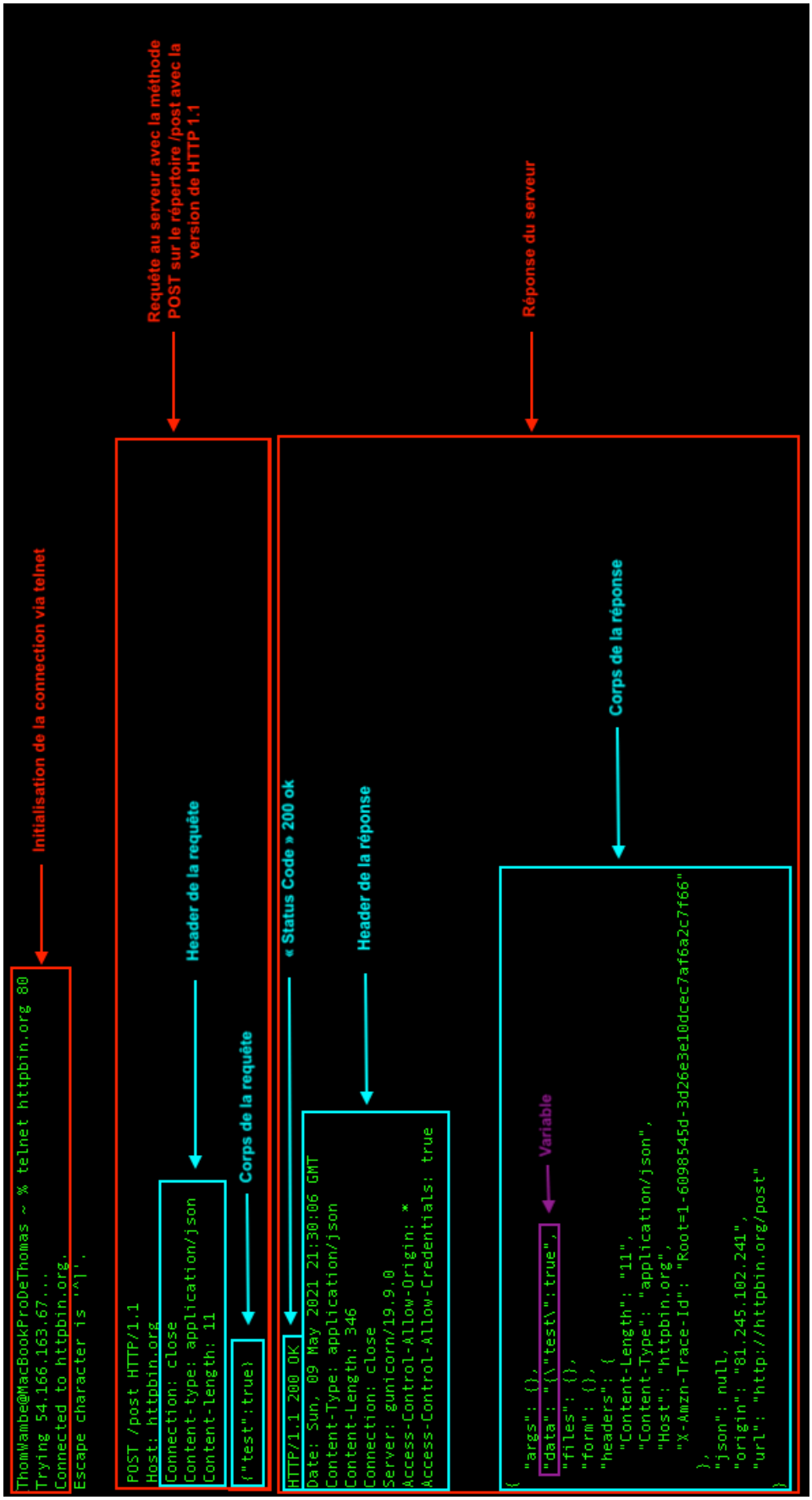

Figure 18 : Schéma Static Rendering

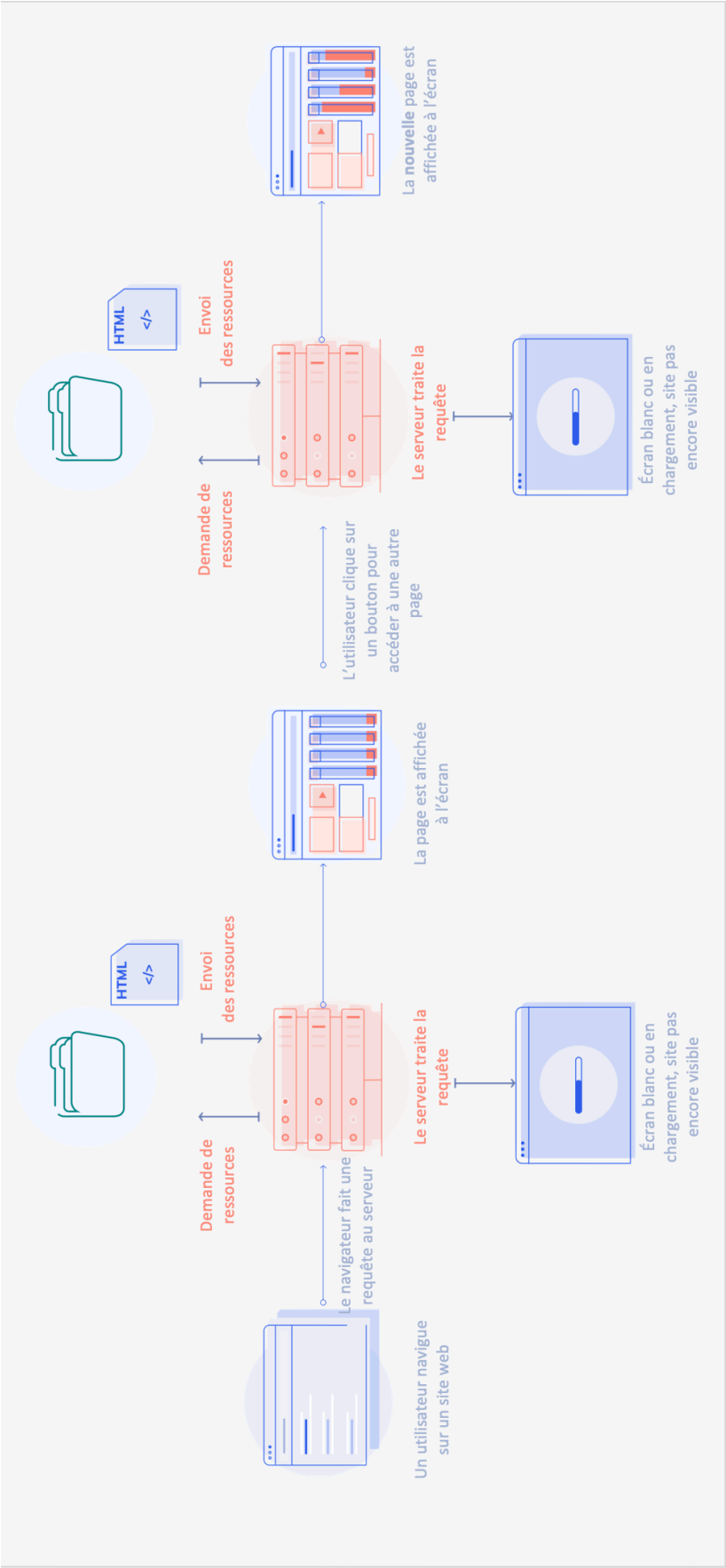

Figure 22 : Schéma Server Side Rendering

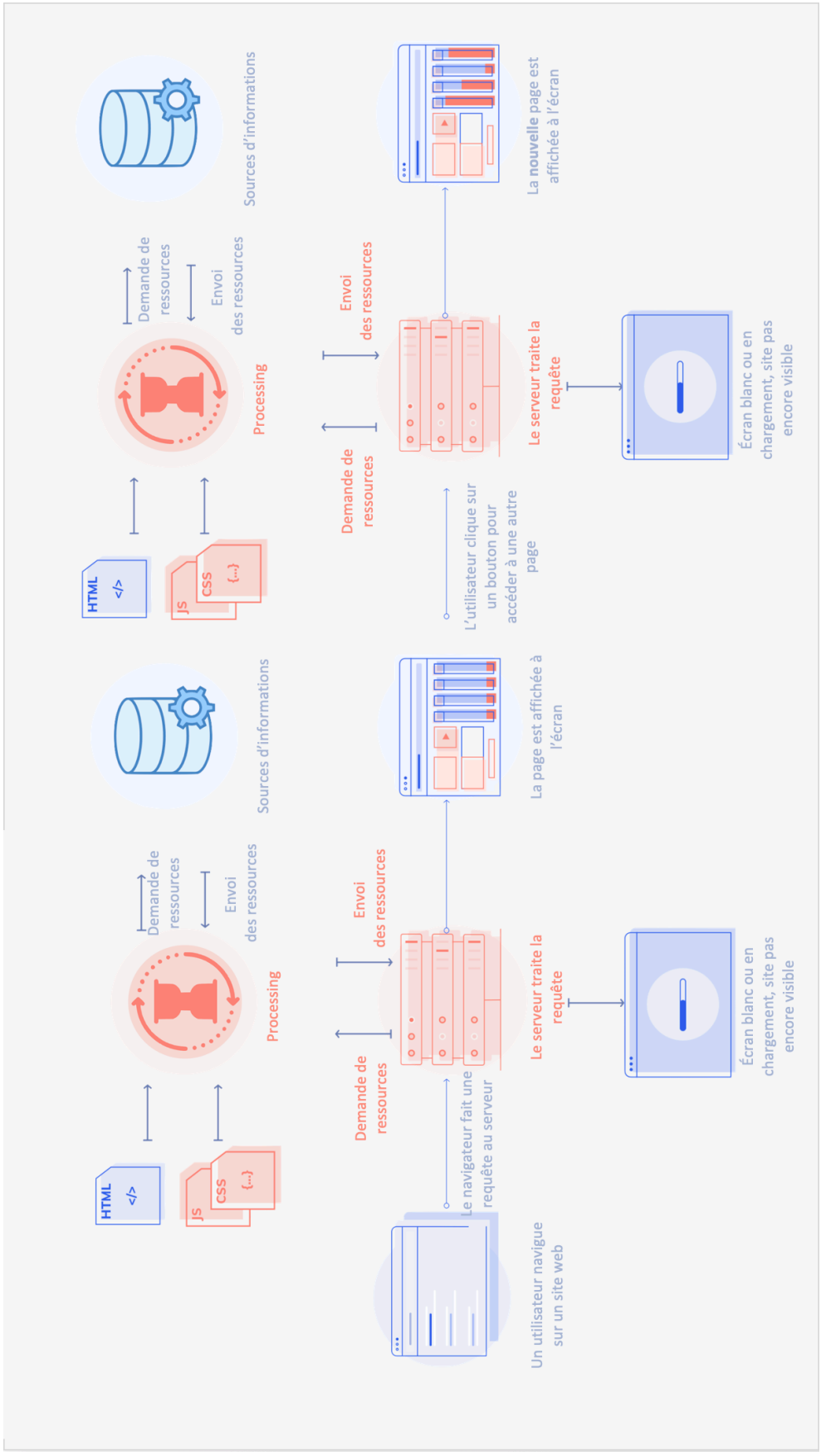

Figure 28 : Schéma Client Side Rendering 1

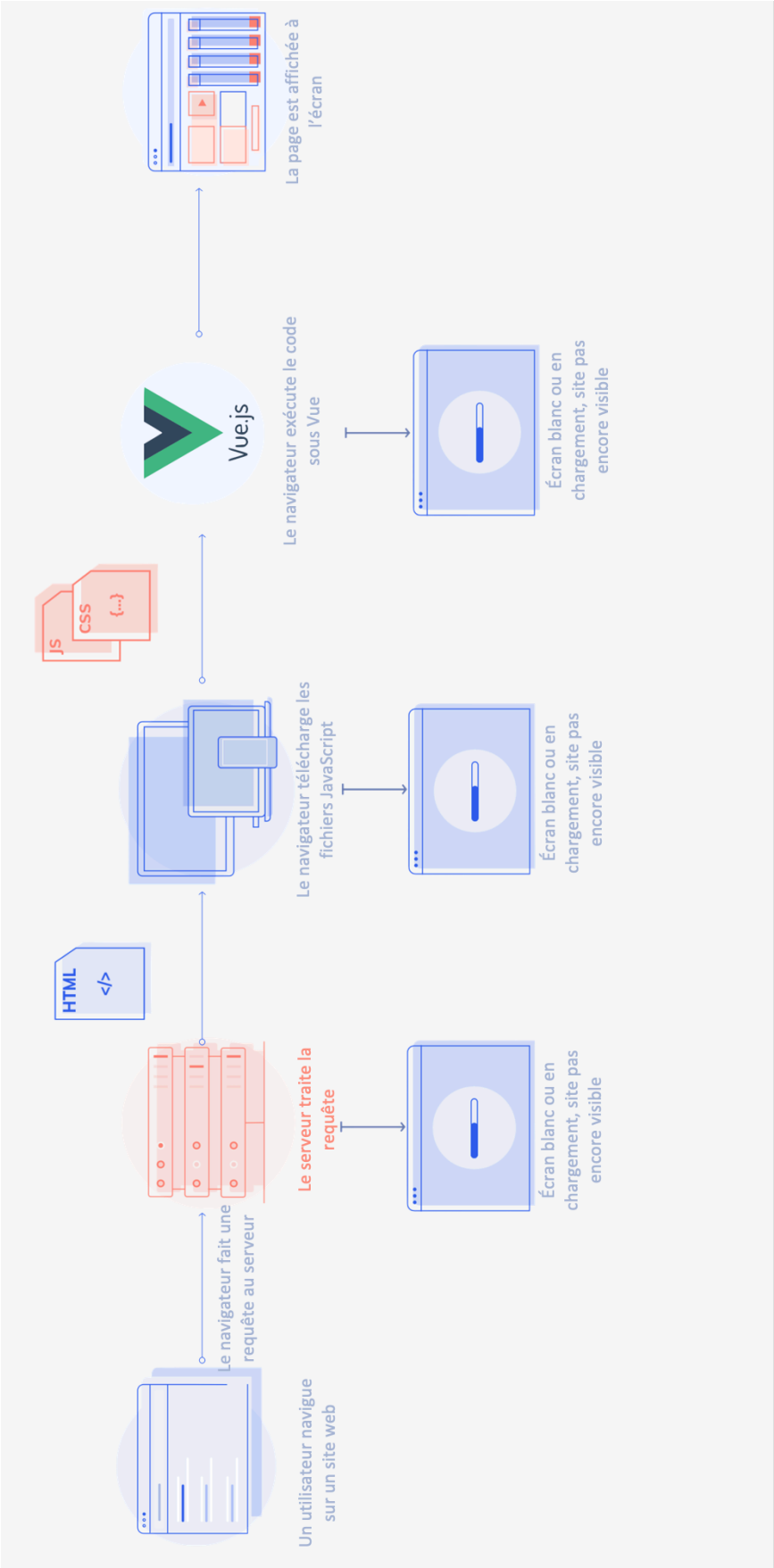

Figure 29 : Schéma Client Side Rendering 2

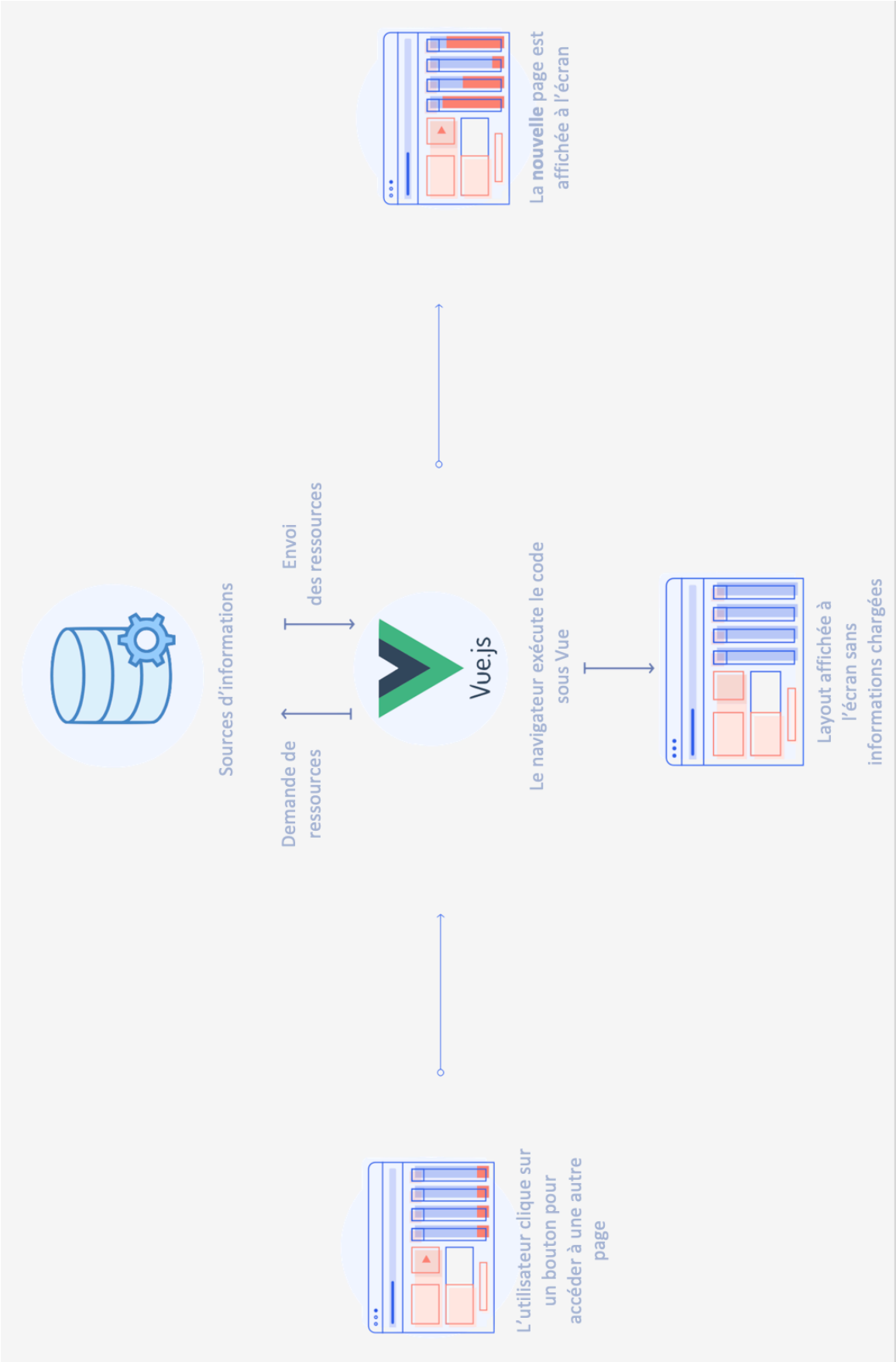

Figure 37 : Schéma Universal Rendering 1

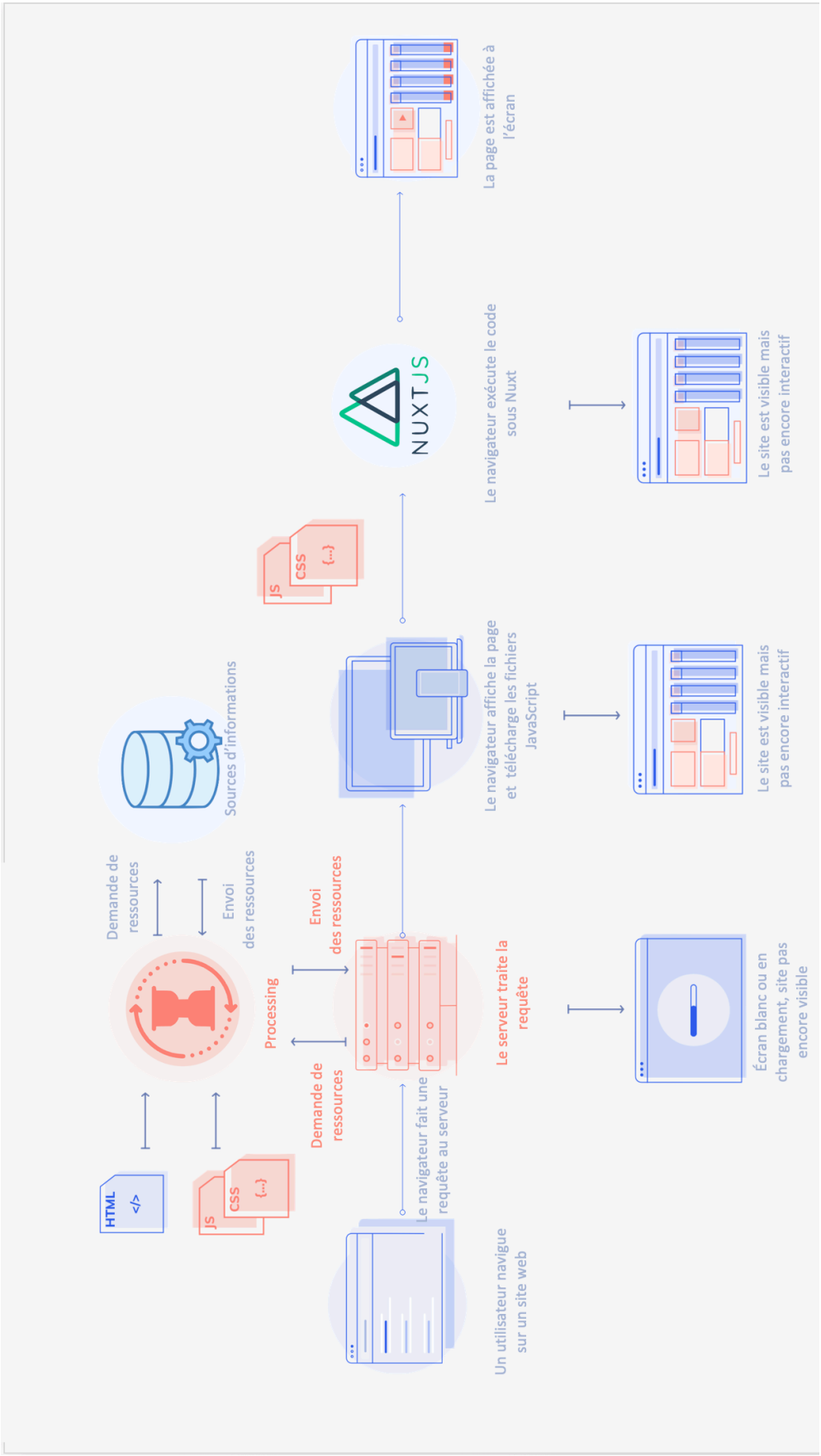

Figure 37 : Schéma Universal Rendering 2