CAVAZZONI Christophe

Mémoire de projet

1

* Abstract
* Compétences couvertes par le Projet
* Analyse du Besoin
  + - Présentation
    - Contexte
    - Contraintes
* Spécifications fonctionnelles
  + - Use Case
    - Diagrammes d’activité
    - Diagrammes de séquence
    - Maquettes et arborescence
* Conception
  + - MCD
    - MLD
* Fonctionnalités
* Annexes
* Remerciements

2

* Abstract

Le projet que je réalise est un site ressemblant aux sites de cours en ligne (type « openclassroom ») qui aura pour thème l’informatique, la programmation, l’algorithmique, ainsi que les technologies et sciences associées à l’informatique, il réunira les différents concepts que j’ai appris lors de la formation et en tant qu’autodidacte, pour ensuite pouvoir les consulter à loisir. C’est un projet qui a d’abord un bénéfice personnel (utile pour moi, pour comprendre plus facilement ce que je veux apprendre, et retrouver facilement ce que je veux chercher).

Dans le futur, le projet comportera aussi des interfaces interactives pour taper du code et tester les différents concepts directement sur le site ou réaliser des jeux basés sur l’algorithmique.

Ce projet a d’abord eu un but purement pédagogique, c’est-à-dire que j’ai cherché à créer un site avec le maximum de fonctionnalités intéressantes dans mon apprentissage du développement.

Le site comporte donc une interface d’inscription et de connexion, de récupération de mot de passe et de modification de compte.

La première fonctionnalité que je présenterais sera une fonctionnalité d’Analytics et de visualisation de données par l’administrateur.

En effet, chaque « évènement » ayant lieu sur le site (inscriptions, connexion, erreurs, messages envoyés, etc …), est récupéré dans la base de données et j’ai implémenté un système de graphiques interactifs permettant de consulter l’historique de ces évènements.

J’ai aussi implémenté une interface de chat instantané permettant aux utilisateurs de dialoguer avec l’administrateur du site. Ce sera la seconde fonctionnalité que je présenterais dans ce mémoire.

* Compétences couvertes par le projet

Les compétences suivantes ont été mises en place pendant la réalisation de ce projet :

* Maquetter une application
* Réaliser une interface utilisateur web statique et adaptable  
  - Développer une interface utilisateur web dynamique
* Créer une base de données   
  - Développer les composants d’accès aux données   
  - Développer la partie backend d’une application web ou web mobile
* Analyse du besoin

Contexte - Besoin :

Ce site est réalisé dans le contexte actuel où les personnes ont un intérêt croissant pour l’informatique, l’apprentissage en ligne depuis chez soi, l’e-learning en général, et comme expliqué dans l’abstract, mon besoin personnel de créer un recueil de ce que j’apprends.  
  
Les besoin principaux seront d’avoir des interfaces claires et simples d’utilisation, et du contenu abordable pour ce qui est des cours.  
Typiquement, le but idéal d’un site de ce type serait d’avoir un niveau de difficulté qui croit assez lentement, et partant d’un niveau « lambda ».  
Ca permet l’accès au cours au plus grand nombre.  
  
Suivant les standards du web moderne, il faudra aussi évidemment que le site soit accessibles et lisible sur les divers formats des médias actuels (responsive design).  
  
Aussi, accorder de l’importance à l’accessibilité, par exemple au personnes malvoyantes, est un besoin réel.

Contraintes techniques :

Ce projet comporte plusieurs contraintes techniques spécifiques :

Tout d’abord l’interface de chat instantané ainsi que l’interface de visualisation de données sont toutes les deux interactives et il n’est donc pas possible de nécessiter un rechargement de la page pour actualiser l’interface. Cela rendrait la navigation très peu plaisante et empêcherait la création d’un chat instantané.

La première contrainte est donc que les interactions client-serveur des fonctionnalités principales du site se feront via des requêtes AJAX (on utilisera l’api « fetch » à cette fin).

La seconde contrainte porte surtout sur l’interface de visualisation de données, en effet pour une fonctionnalité comme celle-là, il faut présupposer que le nombre de données à traiter peut être très grand, que des calculs pourront être nécessaire sur ce grand nombre de données et/ou que la connexion du client peut être médiocre.  
Donc j’ai jugé très important que les requêtes à la base de données devront être très performantes, j’ai essayé de minimiser le nombre d’échanges client – serveur pour cette interface dans le but de minimiser l’impact de la latence du client.  
Pour ce faire, j’ai effectué beaucoup de tests sur mes requêtes ainsi que sur les algorithmes nécessaires (ici de construction des requêtes et de formatage des données finales), pour les rendre plus optimales.  
  
La troisième contrainte porte aussi sur l’interface de visualisation de données, en effet pour créer des graphiques, pour une application web, il existe globalement deux possibilités :

* Dessiner soi-même les graphiques via des manipulations de svg, ce qui se révèle être très complexe, et très chronophage
* Utiliser une dépendance externe comme une librairie, qui s’occupe de l’affichage et du dessin des graphiques.

J’ai opté pour la deuxième possibilité pour faciliter mon travail et optimiser mon temps.  
J’utilise donc la libraire « chartJS » pour générer mes graphiques.

En termes de design pattern et d’architecture, j’ai pris le parti pris de construire l’architecture de mon serveur sous le modèle « API » plutôt que sur le modèle classique, ce qui implique plusieurs contraintes importantes :

* Le serveur ne devra être capable de renvoyer qu’un seul type de donnée normalisé (ici le serveur reverra *exclusivement* du JSON au client).  
  Ceci à plusieurs avantages, en termes de sécurité tout d’abord, le fait que le serveur soit incapable de renvoyer du code HTML permet de limiter drastiquement la possibilité de XSS.  
  En effet, si on prend l’exemple de l’injection malveillante d’un script JS de type « alert » sur toutes les pages d’un site web, étant donné que mon serveur ne sert pas les pages HTML directement, si les possibilités de XSS sont toujours présentes, la vulnérabilité ne proviendra pas du code PHP (mais plus probablement du serveur apache et de ses réglages et sécurités).
* Les redirections seront effectuées par le code coté client, ce qui implique que l’activation du JS est nécessaire au fonctionnement du site en général (ce qui par ailleurs était déjà une contrainte de l’utilisation de requête AJAX).

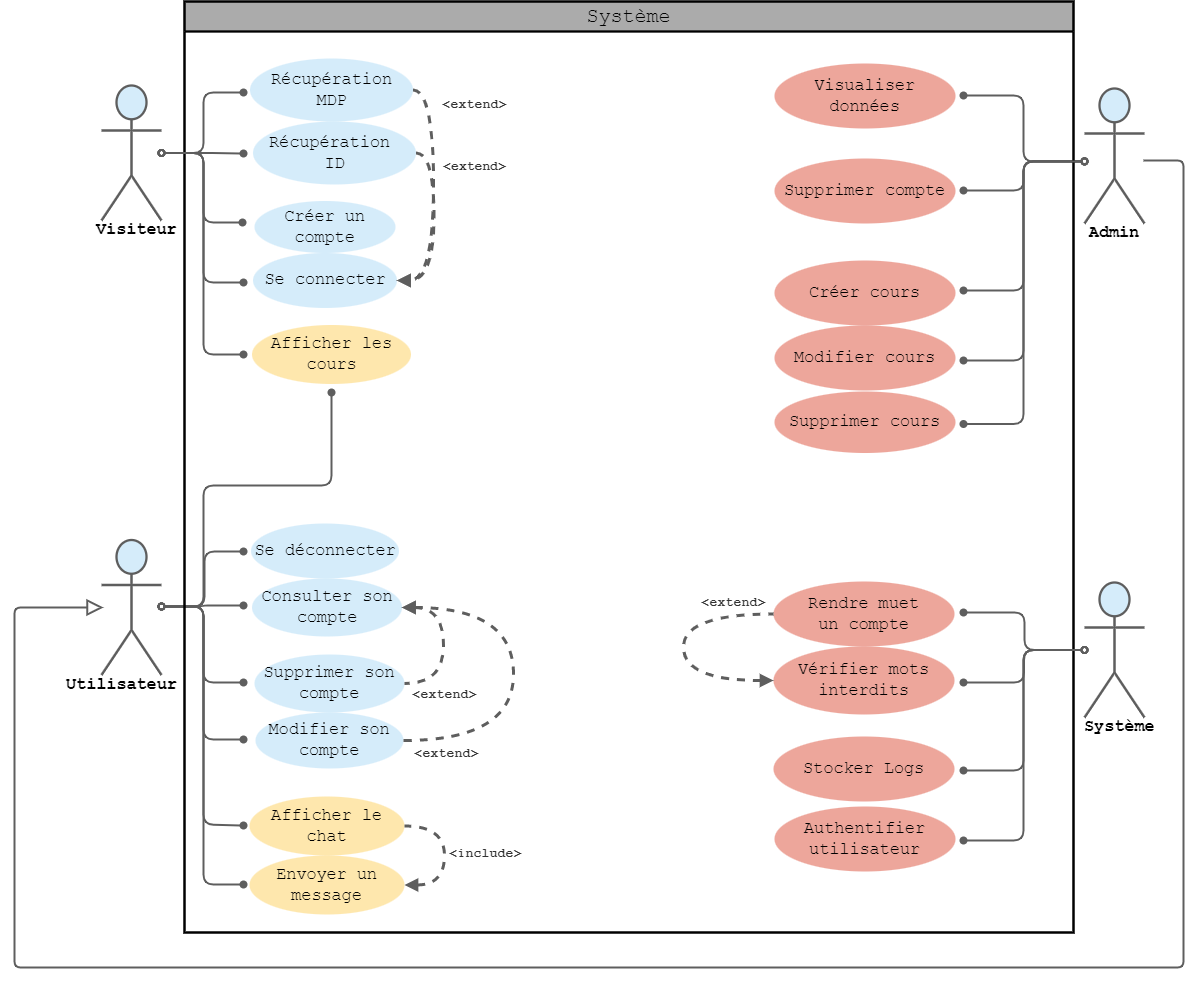
J’ai aussi pris le parti d’avoir les interfaces spécifiques au type de compte (utilisateur / admin) coté client ce qui implique une contrainte importante mais assez facile à résoudre en pratique :

* L’interface ne doit contenir aucun indice / aucune information sur l’architecture du serveur ou le code du serveur, on considère l’interface et le code coté client comme vulnérable *a priori.*
* Tout l’aspect sécurité sera focalisé sur le serveur et l’accès au contenu réel du site, les données.

Enfin, j’ai aussi pris le parti de ne pas utiliser le paradigme objet pour la réalisation de ce site, mais plutôt une approche procédural inspirée au maximum du paradigme fonctionnel, ceci pour plusieurs raisons :

* Il n’y a aucun problème que j’ai pu rencontrer dans ce projet qui auraient pu être résolue par de l’héritage ou du polymorphisme qui sont pour moi les principaux intérêts du paradigme objet.
* Je réalise l’encapsulation des variables que je veux rendre inaccessibles dans d’autres fichiers ou d’autres parties du script directement par l’utilisation de la portée des variables.
* L’approche fonctionnelle garde le même principe de « modularité » qui permet de rajouter simplement des nouvelles fonctionnalités au code. Et le code reste tout aussi lisible.
* On a aussi la liberté de rendre lisible le code par l’utilisation de wrappers entre autres, et créer des abstractions ou des modules qui correspondent à nos besoins de la même manière qu’en POO.
* C’est un des paradigmes qui me passionne particulièrement et que je souhaitais approfondir et comparer au paradigme objet par la pratique.
* Spécifications Fonctionnelles

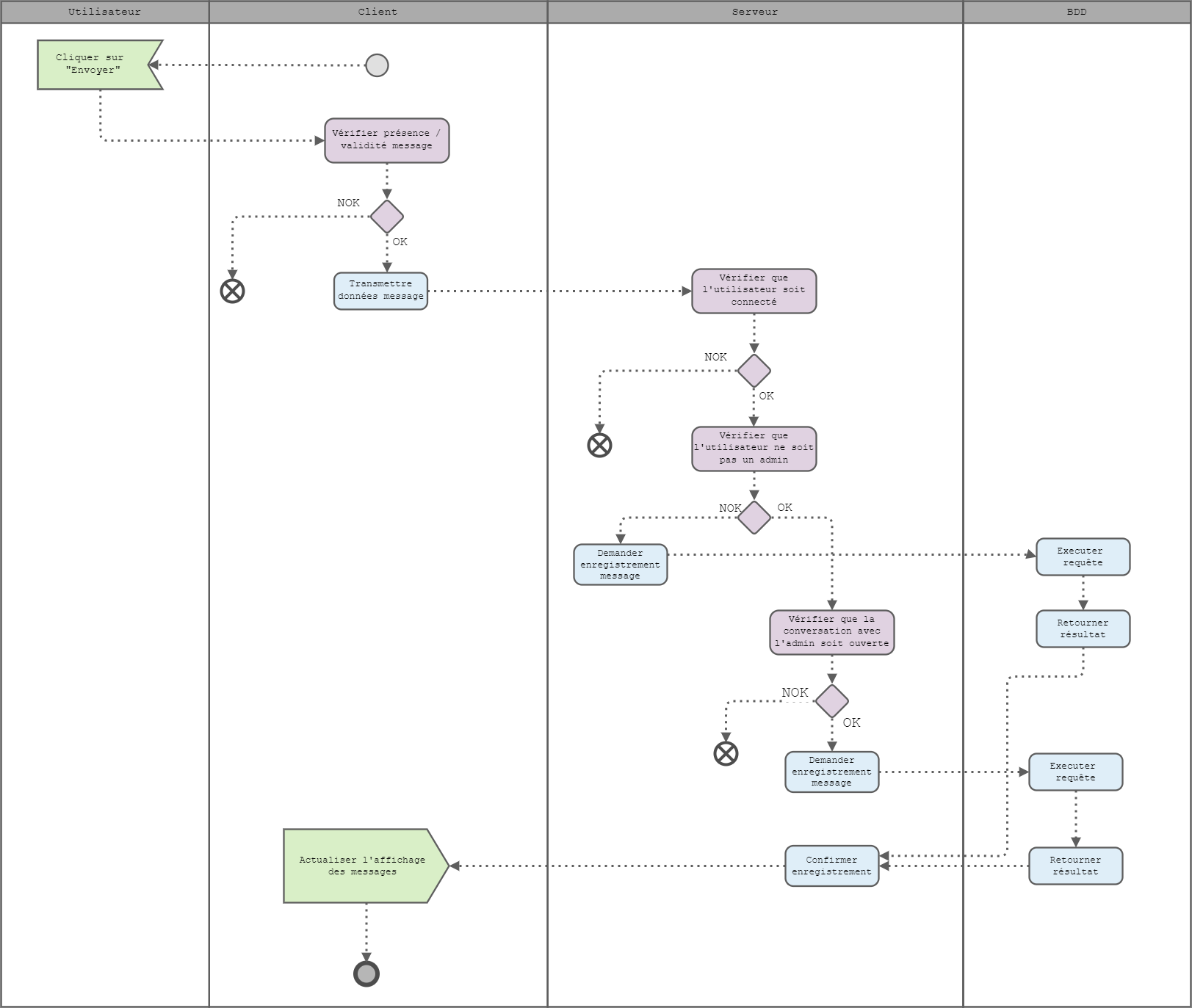
Use Case :



On constate ici les cas d’utilisations principaux. L’acteur système représente ici les actions automatiques réalisées par le serveur qui constituent des fonctionnalités à part entières.

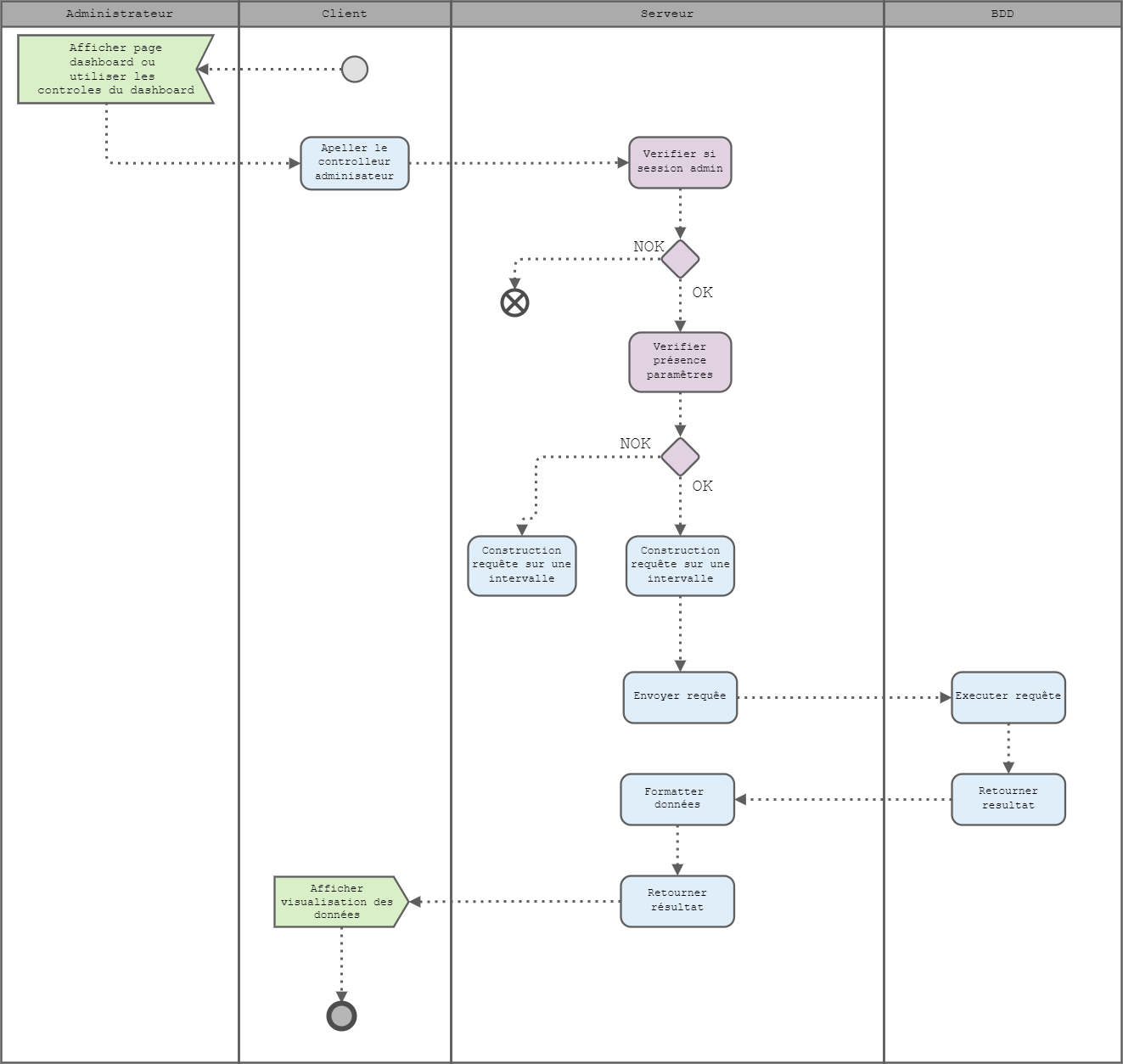
On peut remarquer aussi que l’admin hérite de tous les cas d’utilisateurs accessibles à un utilisateur lambda.

Diagrammes d’activité :

Pour le système de chat instantané, j’utilise un système de conversations ouvertes ou fermées :

* Si un utilisateur envoie un message, il ouvre une conversation avec l’administrateur, si elle n’est pas déjà ouverte.
* L’administrateur à la possibilité de fermer les conversations qu’il juge terminées.

Ceci permet à l’admin, même avec un potentiel grand nombre d’utilisateurs, de ne pas avoir une liste très longue de tous les utilisateurs, et d’avoir à chercher avec quels utilisateurs il était en train d’échanger.



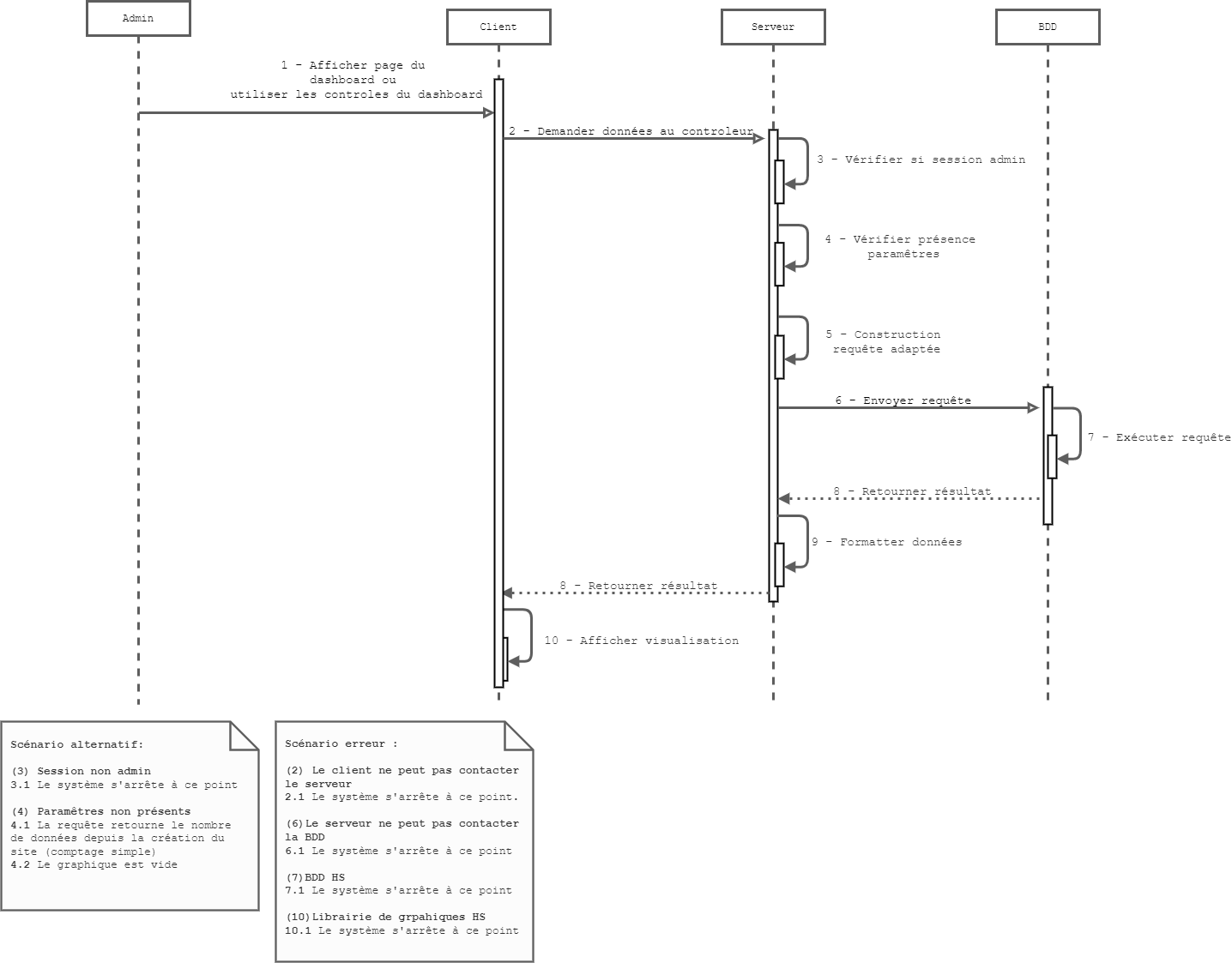
On aperçoit sur ce diagramme que le serveur bloque l’accès aux données utilisées par le Dashboard à quiconque n’ayant pas de session administrateur.

Et un embranchement important qui permet d’aller chercher les données de deux manières différentes :

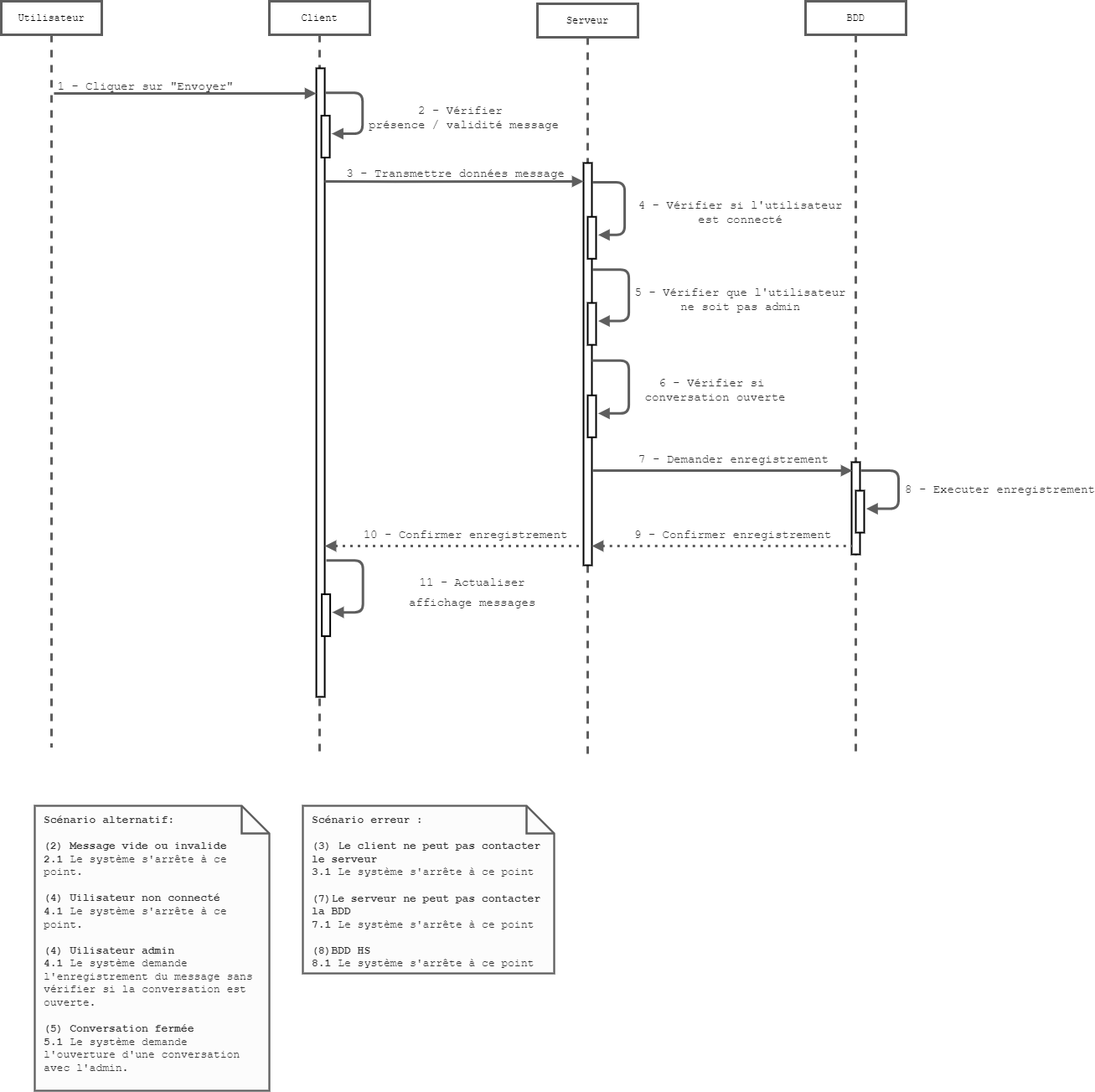
- Si on fournit des variables d’intervalle et de résolution à la requête, on construira alors une requête adaptée qui ira chercher les données d’un certain type (exemple : connexions) et retournera une liste d’intervalle de temps associée au nombre d’occurrence de la donnée en question pendant cet intervalle (exemple : la liste du nombre de connexions journalières des 14 derniers jours).

- Sinon on récupère alors le total du nombre d’occurrences depuis la création du site. Ce qui peut être utile pour avoir, par exemple, le nombre de connexion ou de messages envoyés depuis la création du site.

Diagrammes de séquence :

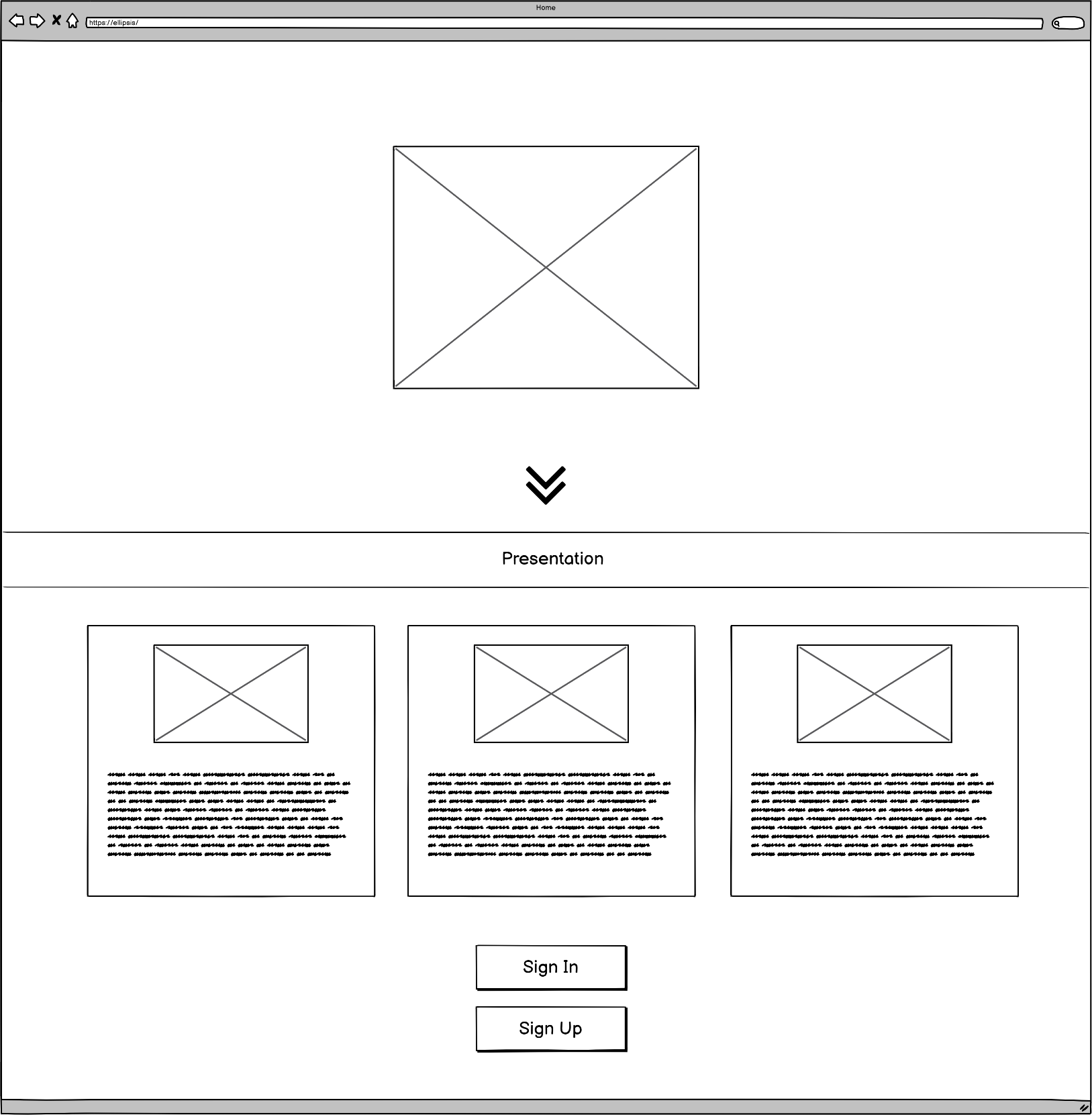


Sur ce diagramme on peut constater qu’un des cas d’erreur est liée à la dépendance utilisée pour cette fonctionnalité, la libraire qui servira à afficher les graphiques.   
En effet, si cette libraire vient à ne plus être supportée ou maintenue, l’ensemble de l’interface deviendrait hors service.   
La meilleure solution à ce problème étant d’utiliser des dépendances maintenues et utilisées à grande échelle, pour minimiser ce risque.



Maquettes :

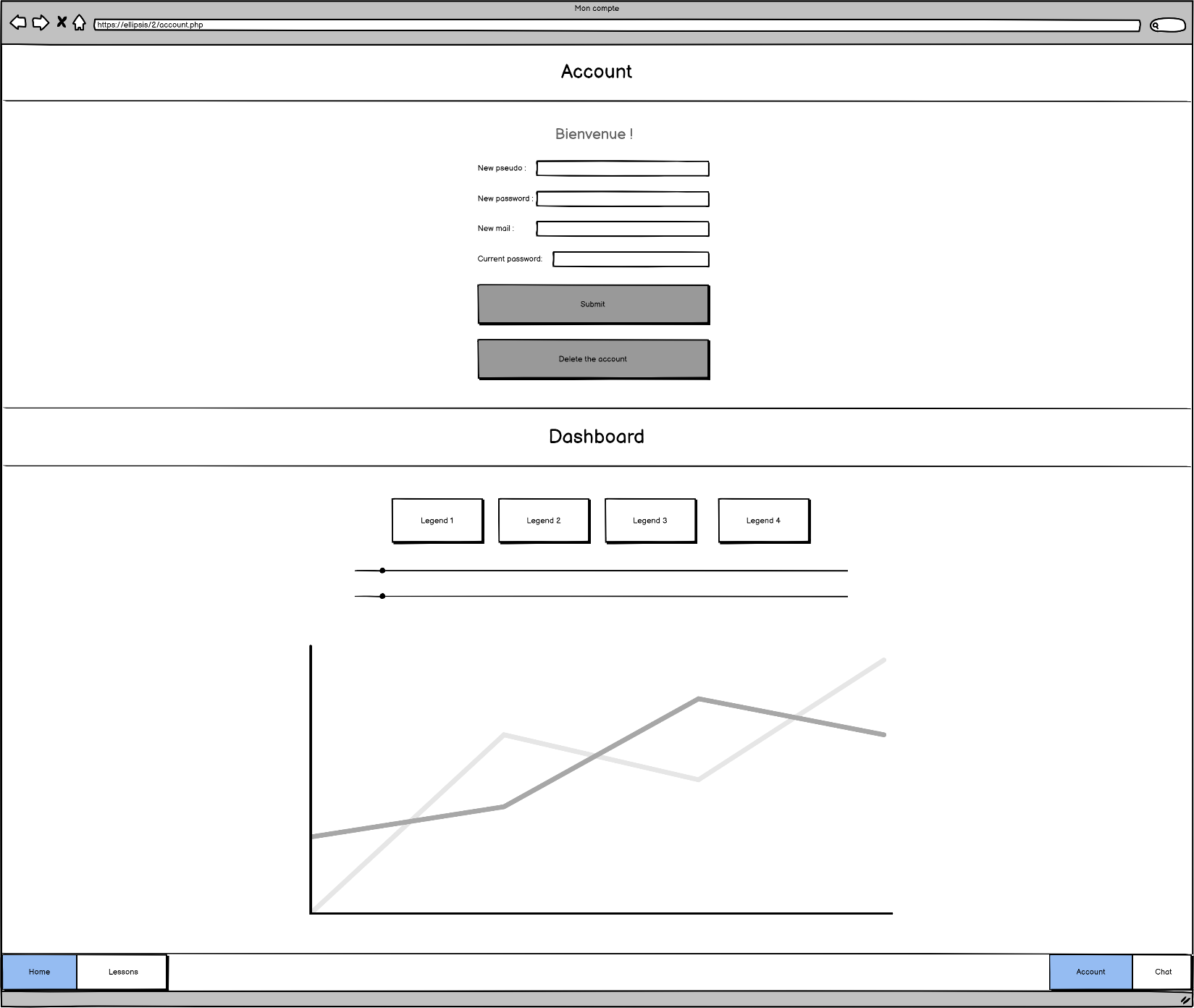
Page d’accueil :



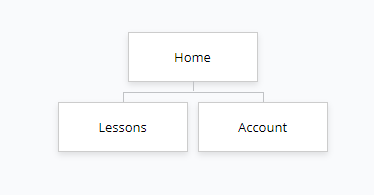
Page de contenu :



Page de gestion du compte :

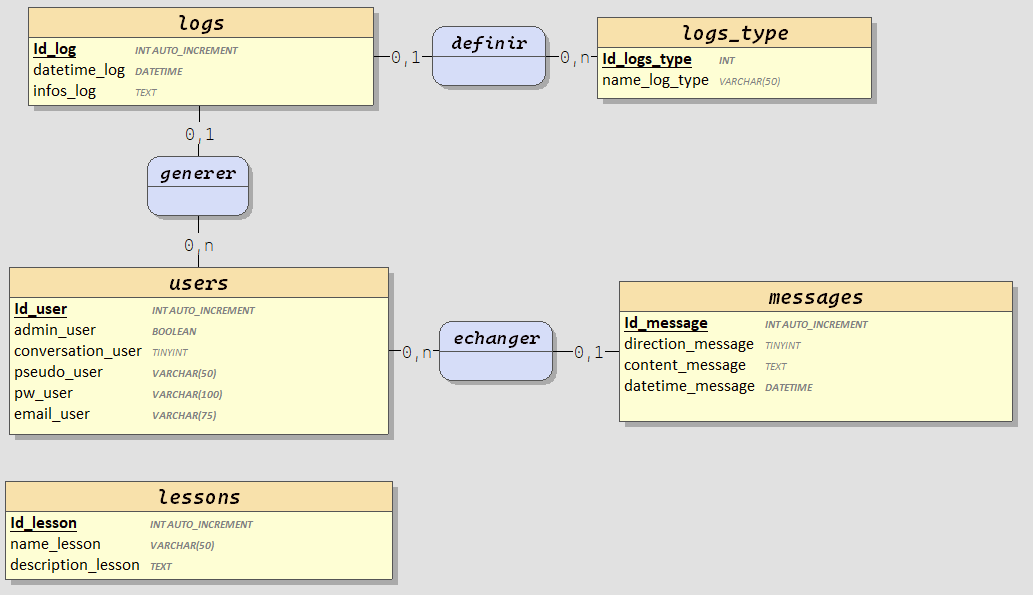


Arborescence :



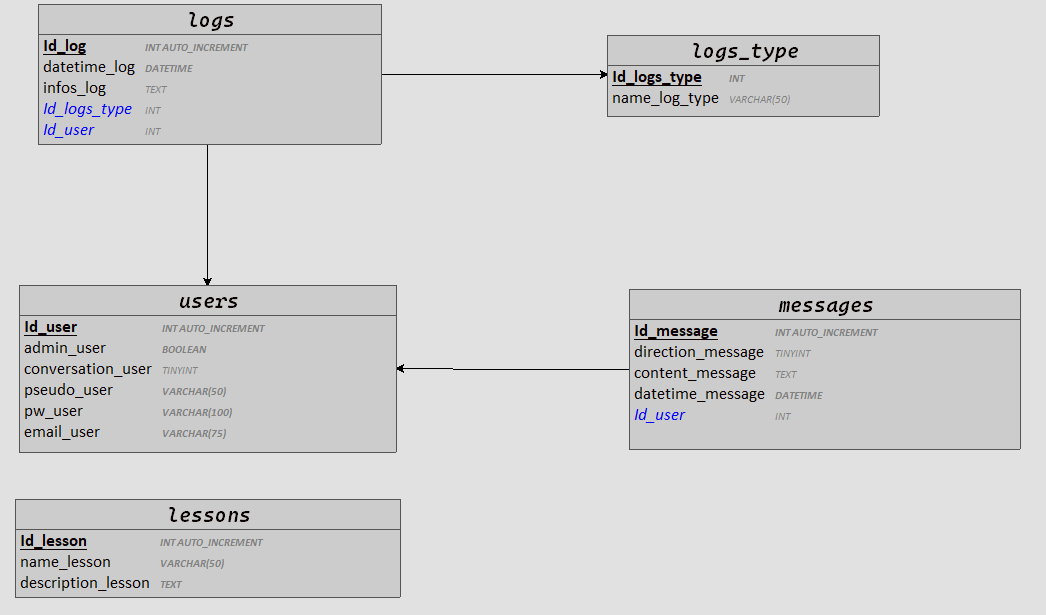
* Conception

MCD :



Chaque utilisateur est associé aux messages qu’il envoie, les messages sont associés à une direction, qui permet d’identifier les envoyés et reçus. Le sens par défaut étant pris en référence à l’utilisateur commun (0 reçu, 1 envoyé) tandis que l’admin à le sens inverse.  
  
Chaque logs généré sur le site est associé à un utilisateur (par exemple logs de connexion / suppression de compte).  
Les logs dits « système » (par exemple les logs d’erreurs ou d’inscription) sont associé à l’administrateur.   
Ceci permet une plus grande facilité pour aller récupérer les logs par la suite.  
J’ai créé une table séparée pour identifier le type de logs, et permet un parcours plus rapide de la base de données lors des recherches.

MLD :



Sur le MLD on aperçoit clairement les clés étrangères qui seront utilisées par les tables, par exemple la table messages qui référence l’id de l’utilisateur.

* Fonctionnalité 1 : Utilisation de l’outil visualisation de données

Structure HTML :

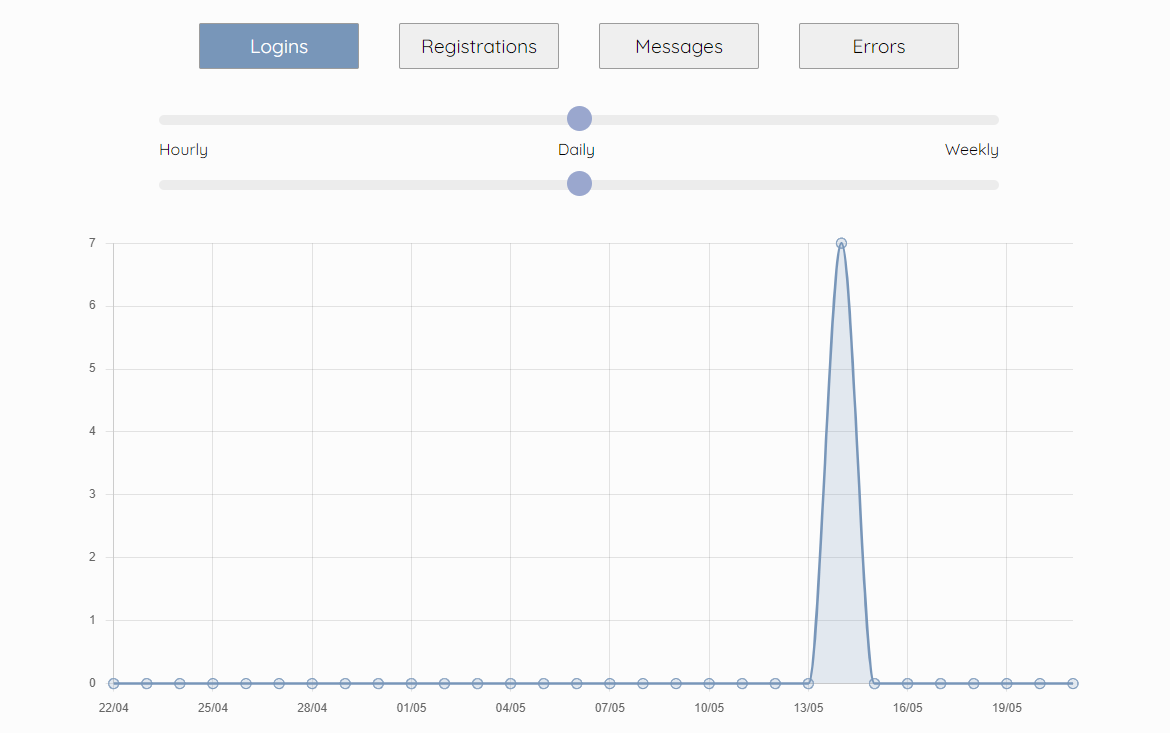


Toute l’interface est englobée dans une section avec un id correspondant.

L’interface se découpe en plusieurs blocs :

* Un titre qui introduit le dashboard
* Un bloc qui contient tous les inputs qui serviront à interagit avec le graphique (des boutons qui activent/désactivent une courbe en particulier type « légende », un slider qui servira à choisir la résolution souhaitée (par heure, par jour ou par semaine pour le moment) et un second slider qui permettra de choisir l’intervalle de temps (7 options)).
* Le canva qui sera utilisée par la librairie pour dessiner le graphique.

Aperçu de l’interface :



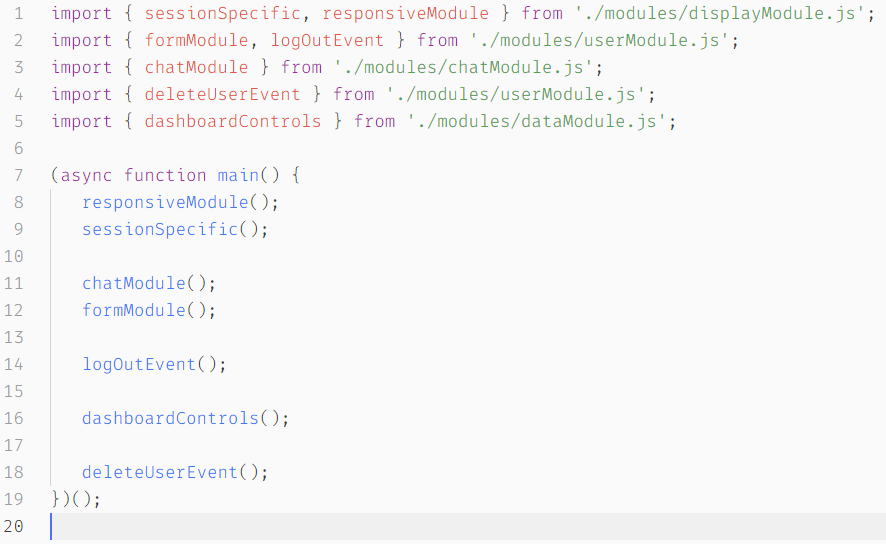
Requêtes du client :

Script principal :



Ce script exécute toutes les fonctions utilisées sur la page account.php.

La fonctionnalité dont je vais parler est celle qui appartient au dataModule (surligné ci-dessous) et sa fonction principale appelée dashboardControl.





Module utilisé pour gérer l’interface :



La première variable globale au début du module me permet de définir quels graphiques seront affichés au chargement de la page, ici seulement le premier sera affiché. La seconde me permet simplement de définir une liste de couleurs qui seront utilisés par la librairie plus tard pour colorer les couleurs.

Ce module utilise et importe plusieurs fonctions que je vais devoir décrire avant de pouvoir décrire les fonctions spécifiques du dashboard.

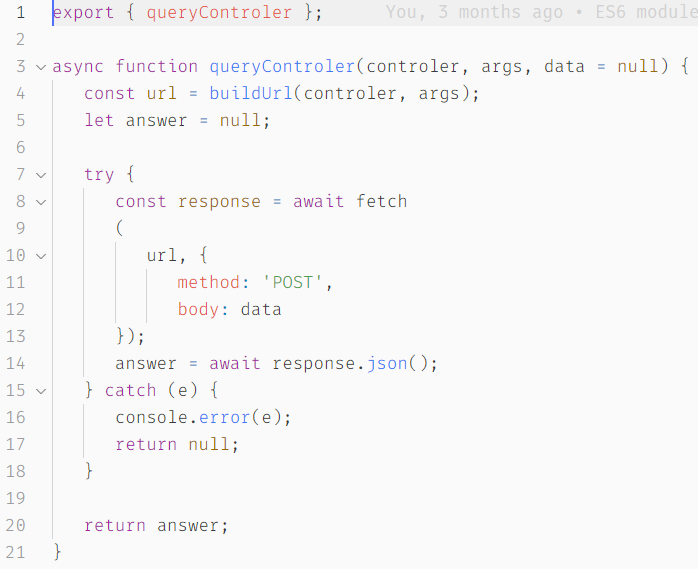
Fonction *queryControler* :

La fonction *queryControler* que j’importe dans le script ci-dessus est la fonction générique qui me permet d’envoyer des requêtes AJAX à n’importe quelle route sur mon serveur.

C’est la fonction la plus importante car elle me permet de contacter le serveur peu importe le type de requête AJAX que j’aurais besoin d’utiliser (y compris pour la fonctionnalité 2 que je présenterais par la suite dans ce mémoire).

Le but ici était donc de réussir à écrire la fonction la plus générale possible et la plus indépendante du contexte d’exécution possible, de telle sorte que je n’ai pas besoin de me repencher sur la fonction par la suite.

Elle est donc capable de fonctionner de manière totalement transparente, dans la plupart des cas d’utilisations que j’ai pu imaginer.



Elle prend comme paramètres :

- Le nom du contrôleur que je veux appeler.

- Un objet qui contient des arguments qui me seront utiles pour ajouter des paramètres à ma requête HTTP, et pouvoir avoir un routage précis et efficace.

- Des données (optionnelles) dans le cas où la requête AJAX vient d’un formulaire par exemple et nécessite de passer des données au serveur.

Elle utilise cette fonction secondaire qui me permet de construire dynamiquement l’URL dont j’aurais besoin :



La fonction *buildUrl* utilise les arguments passés à la première fonction et parcours la liste des arguments.

Les URL que j’utilise pour mes routes AJAX sont formatés de la sorte :

- Un premier paramètre est nommé « *type* » et me permet de déterminer quel type de requête est faite.

- Dans le cas où je passe un argument avec la clé « *target[]* », cela signifie que je veux utiliser un paramètres http sous forme de tableau. Ceci me sera très utile pour envoyer une requête de données relatives à plusieurs courbes.

- Enfin, si l’argument n’est ni de type « *type* » ni de type « *target[]* », il est simplement rajouté à la fin de l’URL. Ceci me permet de garder de la flexibilité, et de pouvoir rajouter des arguments que je n’aurais pas prévu à la base.

J’illustrerais cette fonction par un appel, lors de la description de la fonction dashboardControl ci-dessous.

Fonction *dashboardControl* :

Cette fonction se découpe en deux partie, une partie relativement triviale qui s’occupe de récupérer les éléments du DOM utilisés (les inputs, et le canva) et de rajouter quelques animations au « hover » des boutons (avec les couleurs des courbes définies plus haut).



Et la deuxième partie juste au-dessous qui contient la logique du dashboard, tout d’abord, on définit les variables qui seront utilisées, et l’état initial du graphique :



*stepval* *:* Celle-ci correspond aux différents intervalles de temps disponibles actuellement quand la résolution est journalière.

*resolution :* La résolution initiale au chargement de la page est journalière.

*range :* L’intervalle de temps initial au chargement de la page est de 30 jours.

*args :* Variable importante, c’est celle qui définit les arguments utilisés pour la requête qui ira chercher les jeux de données au chargement de la page.

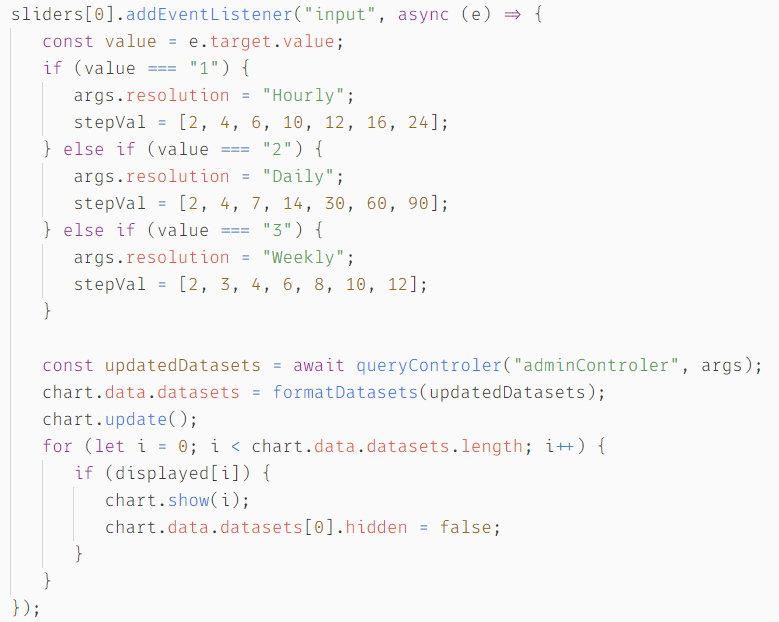
*datasets* : les jeux de données dont on a besoin pour afficher le graphique initial. Voilà l’appel à la fonction que j’ai défini plus haut, qui va envoyer la requête AJAX à mon serveur, avec les arguments définis dans la variable *args*.

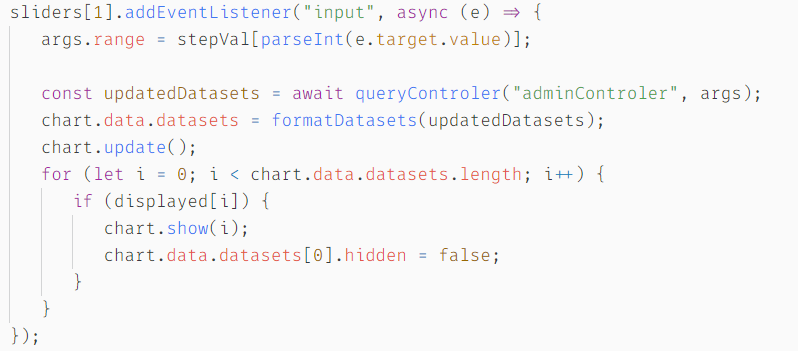
Une fois la requête effectuée, je peux fournir les jeux de données à la libraire pour qu’elle s’occupe de l’affichage grâce à la fonction *drawChart.*

Initialement, j’affiche seulement le premier graphique grâce aux deux dernières lignes.

Mais la fonction ne s’arrête pas là, en effet les graphiques sont voulus interactifs, donc les afficher seulement au chargement de la page n’est pas satisfaisant.

Voici donc la suite de la fonction dashboardControl :





Cette partie se résume assez simplement, à chaque interaction avec le premier slider, par exemple, (celui de la résolution), on change la clé *args.resolution* pour refléter la nouvelle valeur, et on change aussi la variable *stepVal*, pour avoir des valeurs cohérentes avec la résolution (typiquement, avoir des multiples de 15 ou 30 pour la résolution journalière).

Une fois la mise à jour de mes arguments effectués, je peux appeler à nouveau ma fonction *queryControler* avec ma nouvelle variable *args*, et ainsi obtenir les nouveaux jeux de données, sans rechargement de la page. Je remplace les datasets par les nouveaux dans mon objet *chart* et j’utilise la méthode *update* de ma libraire, qui permet de rafraichir les graphiques*.*

La boucle finale des deux écouteurs d’événement me permet de ne montrer que les courbes que l’utilisateur a sélectionné via les boutons.

C’est un point important que je pense important de détailler, car si on considère à nouveau ma variable *args* au tout début de ma fonction, on remarque qu’à chaque fois que je vais chercher des jeux de données, je vais chercher tous les jeux de données, même ceux des courbes qui ne sont pas affichées.  
  
La raison est assez simple, après des tests, je me suis vite rendu compte qu’avec un système où à chaque clics sur un bouton de la légende pour rajouter une courbe, on fait une nouvelle requête, il en résultait une latence à chaque clic et rendait l’utilisation assez désagréable.   
Aller chercher tous les jeux de données à chaque modification de l’intervalle ou de la résolution impose la contrainte d’avoir un certain temps de chargement initial du graphique, mais une fois ce temps de chargement fini, la manipulation est très réactive. On peut donc ensuite, cacher et afficher les courbes, avec une latence minimale.  
  
Enfin voici la partie finale de la fonction dashboardControl : 

C’est simplement la boucle qui me permet d’afficher/cacher les courbes grâce aux boutons de la légende.

*toogleCurve* est une fonction que j’ai séparé pour plus de lisibilité dans ma fonction principale, déjà assez chargée :



Grace à ma variable globale *displayed,* je gère les courbes affichées et cachées et cette fonction s’occupe donc d’activer / désactiver une courbe correspondant à un bouton.

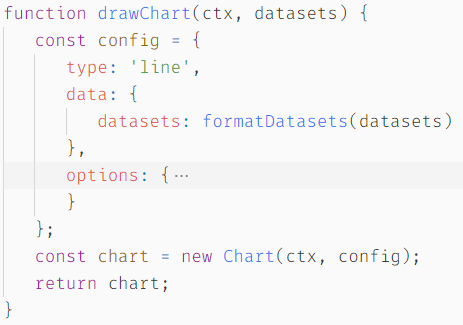
Dans la fonction dashboardControl, j’ai aussi utilisé la fonction formatDatasets qui me permet de configurer les couleurs et les paramêtres des courbes, aussi séparée de ma fonction principale pour plus de lisibilité :

Pour chaque jeu de donnée dans le jeu de donnée global, je lui assigne une couleur et des paramètres génériques.

C’est cette liste de jeu de données qui sera finalement fournie à la librairie à l’étape de l’affichage.



Fonction qui dessine les courbes :



C’est la fonction finale qui instancie la classe fournie par la librairie, et prends en paramètre un canva et un objet config qui contient, mes jeux de données ainsi que diverses options.

Réception des requêtes côté serveur :

Après avoir décrit les différentes requêtes que le client peut faire pour demander des jeux de données, je vais maintenant décrire le fonctionnement du serveur et la gestion de ces requêtes.  
Le contrôleur appelé par les différentes requêtes est nommé *adminControler.php* et voici sa structure générale :





Ce contrôleur effectue d’autres taches que celle que vais décrire, je vais décrire exclusivement les conditions et routes surlignées ci-dessus.

Ce contrôleur utilise les fonctions présentes dans le script *logModel.php*, ce script contient toutes les fonctions d’accès à la base de données et les requêtes.

Voici un aperçu global du modèle (qui sera décrit en détail par la suite) :



Par ailleurs, ce modèle utilise aussi la fonction de connexion à la base de données, qui est située dans le fichier *helpers.php :*



Dans le contrôleur que j’ai présenté au début, la variable *$getdataCondition* est un booléen qui valide la route des requêtes relatives au dashboard, voici cette condition :





Je vérifie tout d’abord la condition essentielle, c’est-à-dire que la session qui effectue la requête soit une session administrateur grâce à des variables liées à l’instance de session, importées depuis *helpers.php.*  
Puis, comme évoqué dans la partie client, je vérifie que le paramètre http *« type »* soit bien *getData*, et que le paramètre *target* soit non-vide*.*Puis, dans la suite du script, je teste ce booléen pour valider la route ou non. Si la route est valide alors je passe tous mes paramètre http à la fonction principale, la fonction *getData* :



On peut constater qu’ici se passe l’embranchement principal évoqué à l’étape des diagrammes d’activité :

* Soit la requête contient un intervalle de temps, et une résolution, auquel cas on utilisera une fonction spécifique du modèle (getDataInterval).
* Soit elle n’en contient pas, auquel cas une autre fonction sera utilisée (getDataSince ou count).

Fonction *getData* :

Voici un aperçu global de la fonction (du code a été caché pour plus de lisibilité du screenshot, le détail est présent ci-après)



On constate que la fonction prend en paramètres les trois paramètres évoqués dans la partie client, et retourne un tableau de jeux de données.  
On aperçoit ici l’embranchement évoqué plus haut.

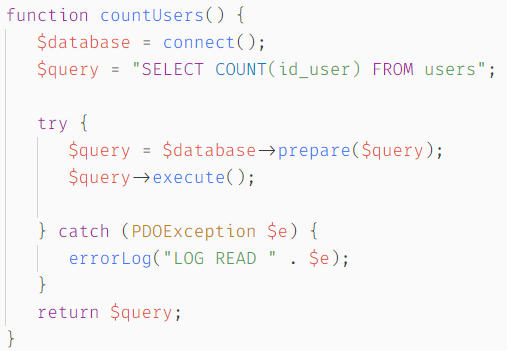
Si les deux paramètres d’intervalles sont des chaînes vides, la fonction utilisera une requête simple avec la possibilité de retourner un type de donnée au choix parmi :

* Le compte total des erreurs depuis la création du site
* Le compte total des utilisateurs inscrits depuis la création du site
* Le compte total des messages depuis la création du site

Exemple du code complet d’un des embranchements en question :



J’y utilise une fonction du modèle nommée *countUser* :



Fonction simple qui se content d’envoyer une requête qui comptera tous les utilisateurs dans la table utilisateurs.

Cette partie est la partie ou l’administrateur demande simplement un compte simple, je vais maintenant présenter la première partie de la fonction getData, c’est-à-dire le cas où l’administrateur veut des données par intervalle de temps avec une résolution donnée.

Considérons cette partie de la fonction :



Plusieurs fonctions y sont utilisées, je vais décrire leurs utilités brièvement, puis présenter le code associé :

*buildBinds :* Cette fonction a pour objectif de créer un tableau de chaines de caractères qui va contenir les paramêtes de bind de la requête finale, le tableau se présentera de la sorte :

[

« :0 » => « intervalle1 »

« :1 » => « intervalle2 »

« : 2 » => « intervalle3 »

……

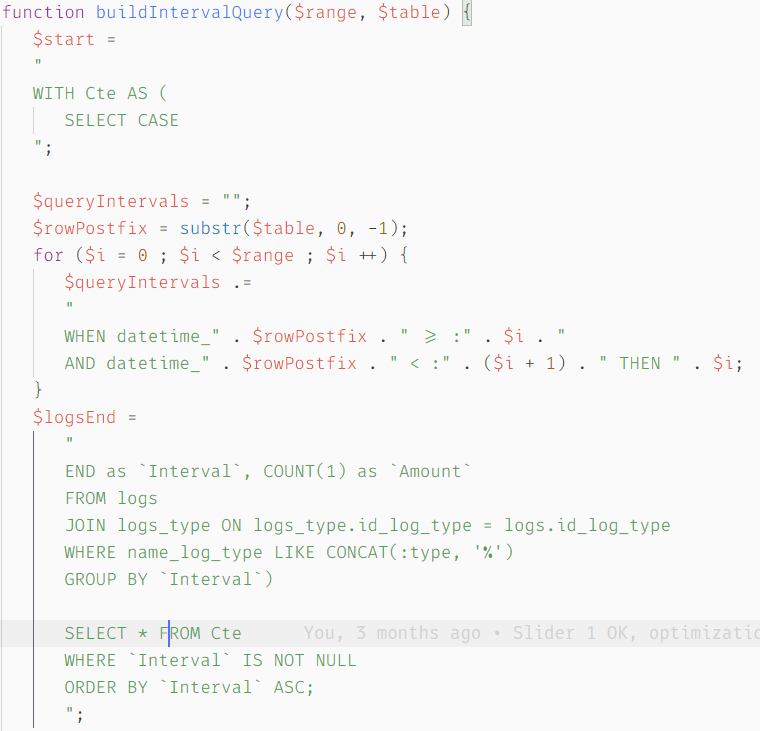
]

Le nombre de lignes du tableau dépendra de l’intervalle de temps considéré et de la résolution de temps considéré.



Pour générer des intervalles cohérents, on utilise la fonction native *strtotime* qui permet de trouver une date un certain intervalle donnée dans le passé.  
Cette boucle commence donc par créer le premier intervalle (le plus vieux) puis se rapproche, tour après tour du dernier intervalle (le plus récent).

*buildIntervalQuery :* Cette fonction s’occuper de l’autre partie nécessaire à la construction de la requête et de la syntaxe générale SQL de la requête, elle prends en paramètres l’intervalle de temps demandé et le nom de la table qu’on veut consulter.



La requête finale est séparée en 3 parties :

* La première est simplement le début de la requête qui restera constant peut importe le nombre d’intervalle. On utilise ici une table temporaire via le mot clé *WITH,* qui est l’équivalent d’une requête imbriquée, mais permet un peu plus de lisibilité.
* La seconde partie est la partie dynamique et dépend du nombre d’intervalles, et on sélectionne les différents intervalles via une requête de type WHEN x > y AND x < z.
* La dernière partie stipule sur quelle table chercher les intervalles, la jointure pour récupérer le nom du type dans la table logs\_type par rapport à l’id stocké dans table logs.

On sélectionne *seulement* les intervalles qui contiennent des occurrences de la donnée demandée. Et on ordonne tout par intervalle chronologiquement.  
On remarque qu’il existe un bind appelé « :type » qui permettra de chercher n’importe quelle type de donnée spécifique dans la table logs et qui correspond au paramètre http « type » envoyé par le client (je précise ce détail plus bas).

Fin de la fonction et concaténation de la requête :



A ce point, la requête est concaténée pour atteindre sa forme finale, selon le type de donnée demandée, la dernière partie de la requête peut varier (du fait des jointures nécessaires et de la partie FROM de la requête qui change si la table qu’on cherche est différente).

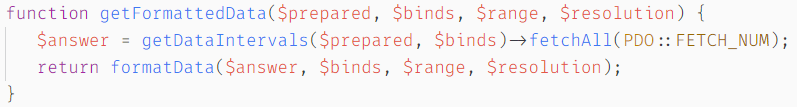
J’ai donc présenté les fonctions intermédiaires utilisées dans *getData,* je vais pouvoir décrire le fonctionnement la fonction :



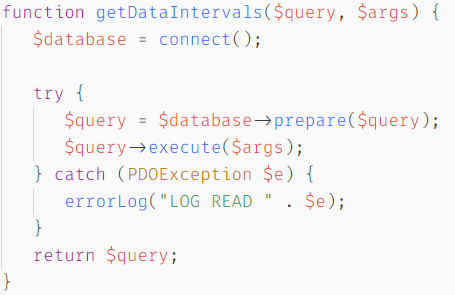
Je crée donc le tableau de binds des intervalles demandés, via la fonction buildBinds au tout début du bloc.

Ensuite pour chaque courbe demandée par l’utilisateur (paramètre http *target[]* comme expliqué plus haut), on va créer un bind qui correspond au type de la courbe (ceci servira chercher les données correspondantes), et pour chaque type de donnée demandé, on créer une requête SQL adaptée via *buildIntervalQuery.*Une fois cette requête crée et le tableau de binds crée, on peut apeller la fonction finale *getFormattedData* qui nous retournera le jeu de donnée formaté comme souhaité.

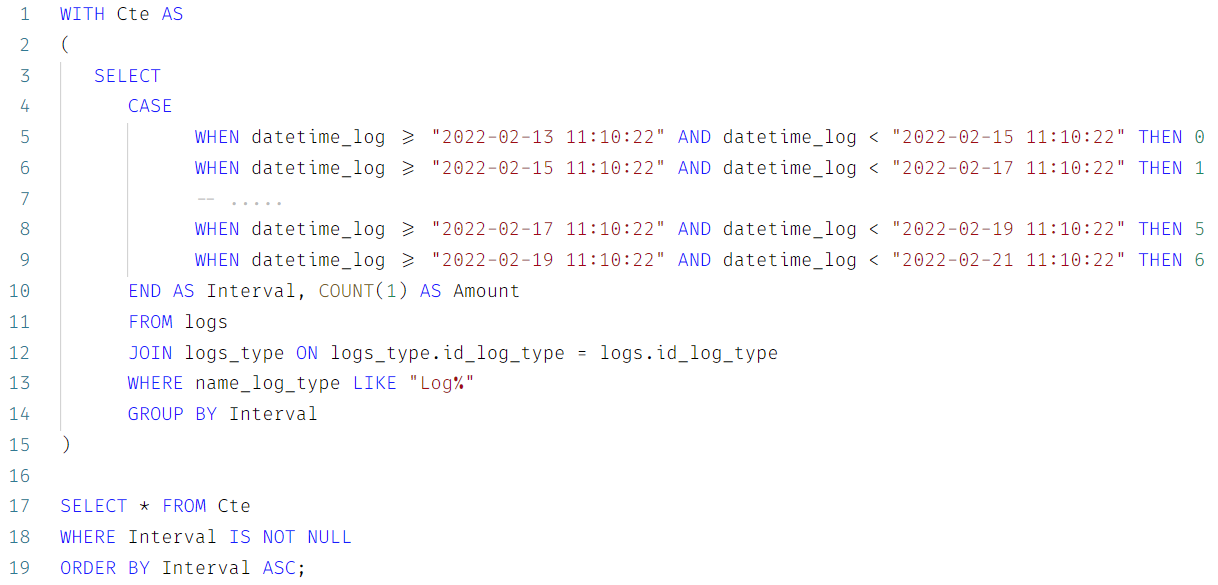
Fonction *getFormattedData :*



Cette fonction appelle la fonction du modèle qui exécutera la requête de recherche :



Cette fonction assez triviale se contente de préparer et d’exécuter la requête grâce à la liste de binds qu’on a créée juste avant et au squelette de requête qu’on a créé dynamiquement.

Exemple de requête sur un intervalle : 

Les points de suspensions illustrent ici que le nombre d’intervalles est dynamique et choisi par l’utilisateur.  
  
Exemple de résultat de la requête :

[

Logins :

{

0 : 12,

1 : 2,

2 : 40

},

Messages :

{

0 : 120,

1 : 69,

2 : 42

}

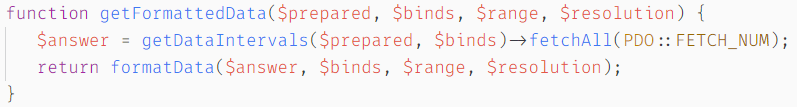
]

On remarque ici que le tableau final écrase complétement les intervalles qui seraient nécessaires pour un affichage normal.

C’est l’objectif de la fonction suivante qui récupère le résultat de la requête pour le formater de telle sorte qu’il soit utilisable facilement par le client.



Cette fonction utilise les index des binds crées plus tôt, et les remplace par des dates, elle replace aussi les intervalles où il n’y a eu aucune occurrence (par défaut MySQL les remplit par la valeur NULL), par zéro pour avoir un type consistant sur tout le jeu de donnée.



Une fois les données récupérées et formatées, elles sont retournées à getData, qui elle-même les retourne sur le premier embranchement du serveur vu au début, qui les retourne par la suite au client après encodage en JSON.  
Une fois sur le client, les données sont fournies à la librairie puis affichées.

Exemple de jeu de donnée formatté :

{

Logins : {

05/07/2022 : 5,

06/07/2022 : 9,

07/07/2022 : 12,

} ,

Messages : {

05/07/2022 : 25,

06/07/2022 : 19,

07/07/2022 : 42,

}

}

* Fonctionnalité 1 : Chat instantané utilisateur

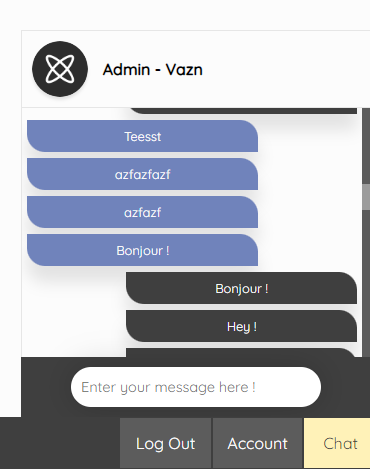
Structure HTML :



La structure du chat est assez simple :

* Un header qui contient le pseudo de l’interlocuteur
* Une fenêtre qui contient les messages
* Un champ permettant d’entrer un nouveau message

Aperçu de l’interface :



Requêtes du client :

Module dédié à l’envoi de formulaires :



Ce module me permet d’envoyer les données de tous les formulaires du site facilement.  
Je parlerais ici du cas ou le formulaire est le chat.

La partie considérée vérifie simplement que le champ est non-vide et tente d’envoyer un message via la fonction *sendMessageAttempt* que je vais décrire.

Ce module utilise également la fonction queryControler pour les requêtes AJAX que j’ai décrit pour la fonctionnalité 1.

Fonction *sendMessageAttempt* :



Cette fonction prend donc les données du formulaire en paramètre et (dans le cas ou l’utilisateur n’est pas admin) appelle simplement le contrôleur *messageControler.php avec* le paramêtre http *createMessage*, et les données du formulaire.

A propos de la fonction *admin* et de sécurité, comme je l’exposais dans la partie « contraintes techniques », l’interface est considérée a priori comme accessible par un utilisateur malveillant, donc quelqu’un peut facilement afficher l’interface de chat en modifiant le script client.

Mais ça ne peut pas perturber le comportement du serveur et donc du site, car une phase d’authentification est effectuée à chaque requête comme nous le verront plus tard.

Réception des requêtes côté serveur :

Le contrôleur appelé par les différentes requêtes est nommé messageControler.php et voici sa structure générale :





Comme précisé, je ne décrirais que ce qui concerne l’envoi et l’enregistrement d’un message (surligné ci-dessus)

Comme pour la fonctionnalité 1 une variable contenant un booléen permet de valider la route :



Elle vérifie que la requête est bien de type *createMessage* et que le message est non-vide.

On vérifie si la session est connectée grâce aux mêmes variables que pour la fonctionnalité 1 :

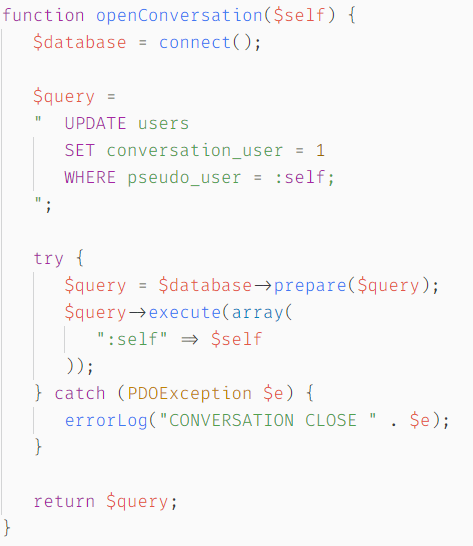


Suite de la fonction :



Si la requête est une requête d’envoie de message, et que nous ne sommes pas un compte administrateur, alors nous faisons appel à deux fonctions du modèle lié aux messages, *messageModel.php*

Fonction *openConversation* :



Cette fonction permet d’ouvrir une conversation avec l’admin en modifiant le champ *conversation\_user* dans la table user pour l’utilisateur en question.

Après avoir ouvert la conversation, on enregistre le message à proprement parler.

Fonction *createMessage* :



Cette fonction enregistre le message, avec le timestamp associé, la direction du message (qui me permet de savoir qui est le destinataire d’un message donné et permet de faciliter les affichages par la suite (notamment l’alternance gauche-droite des bulles du chat, qui sont les mêmes que ce soit sur un compte utilisateur ou admin).  
  
Le message est donc stocké en base de données, et à la prochaine connexion de l’administrateur, les différentes requêtes de rafraichissement lui permettront de le consulter.

* Conclusion

Ce projet a été une réalisation très enrichissante pour moi, en effet, j’ai pu découvrir toutes les parties qui constituent une application web, et toutes les manières de penser les fonctionnalités que l’on souhaite mettre en place. Je suis rentré dans cette formation pour apprendre la programmation et l’algorithmique, et ce projet entre autres, m’aura permis d’avancer dans ce sens.

Au-delà du code, c’est une nouvelle forme d’organisation, et d’autonomie que je pense avoir retiré de ce projet. Ainsi qu’une bonne dose de réalisme, une capacité de juger de ce qui est réalisable en l’état de mes connaissances et ce qui ne l’est pas.  
  
J’ai rencontré quelques difficultés qui ont été tout aussi enrichissantes, par exemple j’ai mis plus de temps que j’imaginais pour comprendre *vraiment* le rôle d’une architecture client-serveur, le rôle spécifique du client et du serveur, et pourquoi on réalise les applications de la sorte.  
Apprendre le langage SQL n’a pas été une tache triviale non plus, c’est très différents de tout les langages que je connaissais, et il a fallu une bonne dose de persévérance pour que « ça rentre ».  
  
A propos du futur de ce projet, je compte le continuer. Une grande étape est que lors de mon stage (que j’ai réalisé seul, pour réaliser le site d’une mairie) j’ai pu découvrir les technologies qui m’intéressaient particulièrement au début de la formation (nodeJS notamment).

Cela a changé mon point de vue sur beaucoup de choses (notamment sur PHP).  
Donc une des étapes futures du projet est la migration de tout le projet sur le stack que j’apprécie le plus, à savoir Typescript/Node/MongoDB/Handlebars, avec Express pour la création du serveur.  
Aussi comme je l’avais dit dans la présentation, j’aimerais implémenter des interfaces pou coder directement sur le site, idéalement dans plusieurs langages pas forcément interprétés.  
Ce qui sera un travail probablement très difficile, j’imagine que faire compiler du C par le navigateur ou le serveur n'est pas trivial, mais ça peut être sacrément intéressant.  
  
Pour ce qui est de mon futur professionnel personnel, cette formation m’a conforté dans la conviction que j’avais au début de de celle-ci. Je souhaite continuer mes études, pas seulement dans le développement web, mais dans le développement logiciel en général.  
Aussi, après avoir pris un peu de temps pour lire certaines librairies dont le fonctionnement m’intéressait (je pense à THREEjs par exemple), je me suis rendu compte que la limitation principale qui perdure après ce type de formation n’est pas spécialement technique mais surtout théorique.  
Donc je souhaite composer cette limite pour pouvoir vraiment utiliser la programmation pour pouvoir faire tout ce que je pourrais avoir envie de faire.  
Concrètement, je compte faire une licence en informatique générale, et si la motivation perdure après ça, pourquoi pas un Master.

* Annexes
* Ici je mettrais la traduction d’un texte anglais, et peut être d’autres choses ?
* Remerciements

Je tiens à remercie tout d’abord l’ensemble des formateurs qui nous ont accompagné, qui ont été très pédagogues et à l’écoute.  
Mais surtout je souhaite remercier particulièrement :

- Rodolphe BRUMENT pour avoir sa bonne humeur tout le long de cette formation et pour avoir répondu à mes questions, même à des heures peu conventionnelles.

* Jonathan GRUSON pour avoir été notre formateur principal une grande partie de la formation, malgré les difficultés.
* Pierre COUPECHOUX pour ses cours d’une profondeur incroyable, et particulièrement passionnants.

Et bien sûr je remercie aussi tous mes collègues de la session, pour les questions, les blagues, les problèmes. 😉