**Diagrammes UML Front End :**

MVVM :

Ce modèle de conception définit l'architecture du projet, après avoir entré dans le dossier «./app » chaque grande section a son propre dossier, ils peuvent être trouvés sous les noms qui se termine par « Module ». Les principaux modules du projet sont : « Welcome », « Login », « Register », « OTP », « Home », « Add », « Associate » et « View ».

Dans chaque sous-dossier de ces modules, le motif MVVM est clairement visible. Les sous-modules du projet ont leur propre « View », « ViewModel » et, « Model ».

Le View : contient uniquement la partie HTML/CSS du sous-module.

Le ViewModel : contient toute la logique (appels API, manipulation des données dans le modèle, fonctions onLoad ...)

Le Model : contient les données du sous-module.

Un exemple se trouve dans le dossier : «./app/addModule/addCampus/ ».

Diagramme UML (exemple pour le dossier addCampus) :

|  |
| --- |
| View Model  «./viewModel/ » |
| * campusModel : AddCampusModel |
| + onLoad() ;  + submitAddCampus() ;  + bind() ;  + showLoader(Bool show) ;  + checkFields() ;  + hideError() ; |

Data binding

|  |
| --- |
| View  « ./view/ » |
|  |

|  |
| --- |
| Model  « ./model/ »  Class : AddCampusModel ; |
| * model : JSON Object |
| + constructor() ;  + getCurrentObj() ;  + filleName(String name) |

Send Data

Recieve Data Data

Factory :

Ce patron de conception m’aide à créer des objets d’une manière plus souple, efficace et pratique. Il se trouve dans le fichier « ./factories/formFactory/formFactory.js ».

Je l'ai utilisé dans le projet pour créer les formulaires de saisie de l'utilisateur de tout le projet. Chaque formulaire se trouve dans le dossier «. /factories/formFactory/forms»

Chaque formulaire est une classe composer d’une fonction « getForm » pour retourner le formulaire.

La méthode principale pour le Factory est « createForm » elle prend en paramètre le type du formulaire et retourne l’objet correspondant à se formulaire.

Les type de formulaire se trouve dans un enum dans le fichier « ./factories/formFactory/formTypes.js »

Diagramme UML :

|  |
| --- |
| Forms  «./factories/formFactory/forms » |
|  |
| + getForm() |

|  |
| --- |
| Factory  « ./factories/formfactory/formfactory.js » |
|  |
| + createForm(FormType type) |

Creates

Singleton :

Ce patron de conception est utilisé pour s'assurer qu'une seule instance d'un objet existe dans le système à un moment donné. Le singleton est utile pour les objets et les fonctions qui doivent être accessibles globalement dans le projet. Il peut être trouver dans le fichier : « ./helpers/helpers.js »

L’instance utiliser est nommer « sharedHelperInstance »

En JavaScript au lieu d’utiliser la méthode getInstance() pour retourner l’instance j’utilise la méthode « Object.freeze() »

Diagramme UML:

|  |
| --- |
| Singleton  «./helpers/helpers.js»  Class : Helpers |
| * sharedHelperInstance : Helpers |
| + constructor()  + hashPassword(String password)  + isValidMail(String mail) ………… |

Template :

Le modèle Template Method fournit un aperçu d'une série d'étapes pour un algorithme. Les objets qui mettent en œuvre ces étapes conservent la structure originale de l'algorithme mais ont la possibilité de redéfinir ou d'ajuster certaines étapes.

Ce patron est trouvé dans le fichier : « ./services/template.js »

Il est utilisé dans le projet pour les appelle APIs pour s’assurer que tous les appellent suivent une même structure et le même ordre d’appel de fonctions.

Un exemple de d’utilisation se trouve dans le fichier « ./services/login/loginService.js » où l’appelle d’API se fait de la même façon du Template.

La fonction : « inherit » m’aide à initialiser ce Template.

Je définie chaque méthode et à la fin j’appelle la fonction « call » du Template qui va en ordre appeler :

1. Montrer le loader.
2. Appel API.
3. Cacher le loader.

Diagramme UML :

|  |
| --- |
| Template  « ./services/template.js »  RequestTemplate |
|  |
| + call() : (étapes : +showLoader(), +request(), +hideLoader())  + inherit() |

|  |
| --- |
| LoginService  « ./services/login/loginService.js» |
| * loginRequest : inherit(Template request) |
| Je definie chaque methode du template. Puis j’appelle call |

Module :

JavaScript ne possède pas le concept de méthodes privées et publiques. Pour mettre en œuvre le comportement nécessaire, j'ai donc utilisé le modèle de conception Module. L'un des nombreux avantages de ce modèle est l'encapsulation, c'est-à-dire la protection des états et des comportements contre l'accès par d'autres catégories. Ce modèle de conception donc autorise les méthodes publiques et privées.

Le patron est utilisé dans tous les ViewModels du projet.

Exemple dans le fichier « ./app/addModule/addCampus/viewModel/addCampusViewModel.js » :

Dans ce fichier la seule méthode publique est la méthode « onLoad » (dans le return) que j’appelle dans l’HTML. Toute autre méthode et variables sont privées.

Observer :

Le modèle Observer offre un modèle d'abonnement dans lequel les objets s'abonnent à un événement et sont notifiés lorsque l'événement se produit.

Il est trouvé dans le fichier : « ./observer/observable.js »

La classe Observable contient les méthodes pour s’abonner et pour notifier.

Ce patron est utilisé dans tous les ViewModels pour notifier le Model que l’utilisateur a fait une action qui doit changer les données

Un exemple d’utilisation de ce patron dans le projet :

Exemple dans le fichier « ./app/addModule/addCampus/viewModel/addCampusViewModel.js » :

La méthode « bind » du ViewModel crée des nouveaux Observable et abonne les éléments concerner a des méthodes qui doit s’exécuter quand ils seront notifiés.

Diagramme UML :

|  |
| --- |
| Observable  «./observer/observable.js» |
| * observers : Array Of Observers |
| + constructor()  + subscribe(Function f)  + unsubscribe(Function f)  + notify(Data data) |

|  |
| --- |
| Observer  «./app/addModule/addCampus/  viewModel/addCampusViewModel.js» |
|  |
| + notify() : found in the « bind » function of the viewModel |

Iterator :

Le modèle de conception Iterator me permet de boucler efficacement sur une collection d'objets. Il se trouve dans le fichier : « ./iterator/iterator.js ».

La class Iterator a toutes les méthodes nécessaires pour le comportement attendus :

La variable « items » contient la collection d’objet.

La méthode « first » : met le pointeur à zéro et retourne le premier objet de la liste.

La méthode « next » : retourne l’objet suivant dans la liste et incrémente le pointeur

La méthode « hasNext » : retourne vraie si l’objet a un objet suivant.

La méthode « reset » : met le pointeur à zéro

La méthode « each » : prends en paramètre une fonction qui sera exécuter pour chaque élément de la liste

Un exemple d’utilisation :

Dans le fichier « ./app/addModule/addCampus/viewModel/addCampusViewModel.js » :

Dans la méthode « onLoad » du viewModel, je crée un nouvel objet Iterator et j’appelle la fonction « each » pour boucler dans la liste.

Diagramme UML :

|  |
| --- |
| Iterator  «./iterator/iterator.js» |
| * index : Int(pointeur) * items : Array of objects |
| + constructor(Array items)  + first()  + next()  +hasNext()  + reset()  + each(Function callback) |

crée

addCampusViewModel.js