Dossier présentant les informations qui seraient utiles à un développeur : architecture et bien plus encore.

DOSSIER TECHNIQUE

Groupe 12 – BMW Motorrad

BAJOREK Romain, CARRY David, CHRISTOPHE Léo, CLERC-RENAUD Hugo, GEHANT Dorian

TABLE DES MATIERES

I. L'API Rest Asp .NET Core en C#

1. Architecture

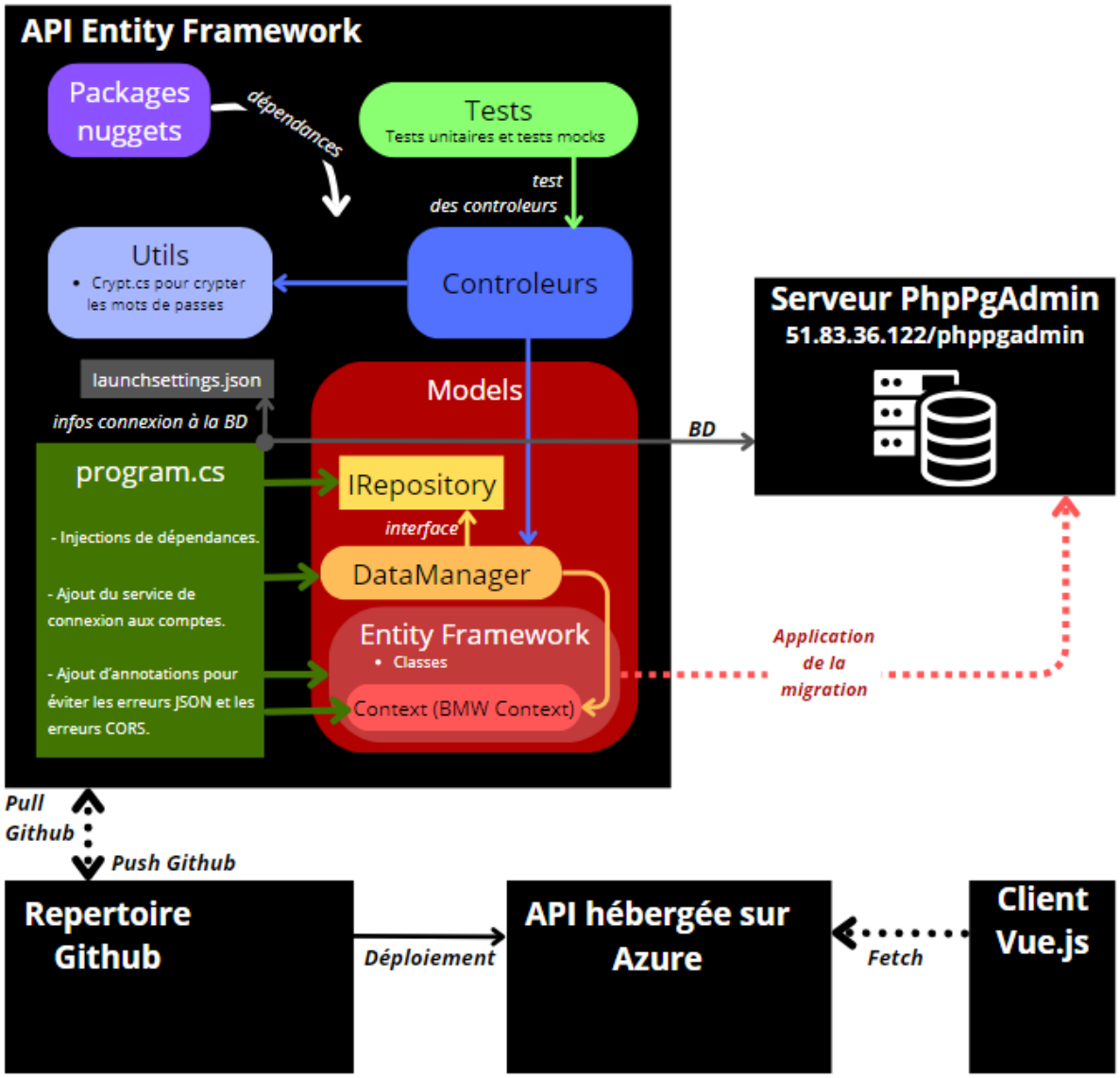


Figure 1 Schéma simplifié de l'architecture de l'API

a. Repository

On utilise les repositories car ils ont plusieurs avantages. Ce concept centralise la logique d'accès aux données, ce qui rend le code plus propre et plus facile à maintenir. Il favorise la réutilisation du code en évitant la duplication de la logique d'accès aux données.

En accord avec les normes, nous avons donc réalisé une interface IDataRepository qui permet d'indiquer aux data managers les méthodes à implémenter. Ensuite, on a séparé les contrôleurs en deux parties : chaque méthode dans le contrôleur a une méthode « sœur » dans le data manager.

Dans le contrôleur, on reçoit la requête, on retourne les possibles erreurs puis on récupère la donnée dans le data manager.

b. *Design Pattern*

Les paternes de conceptions sont un ensemble de solutions aux problèmes les plus courant lors du développement d'application. Les points importants sont : la réutilisabilité (on utilise une même solution à plusieurs problèmes qui se ressemblent), la communication (on utilise des normes de nommage, une nomenclature), les meilleurs pratiques (on utilise les pratiques les plus efficaces et optimisées de la conception), la classification (on classe les différentes solutions en plusieurs catégories), la flexibilité (les paternes de conceptions se doivent d'être flexibles).

c. *Injection de dépendance*

L'injection de dépendances est un principe important de la sécurité lors du développement. Au lieu de créer un objet B dépendant dans un objet A, on rajoute l'objet B déjà existant à l'objet A. C'est plus optimisé et plus sécurisé.

On a réalisé une injection de dépendance pour la chaîne de connexion à la base de données.

Dans le fichier de configuration de l'application appsettings.json, on rajoute une section « ConnectionStrings » :

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "BMWDBContext": "Server=51.83.36.122;port=5432;Database=sa12; uid=sa12; password=[redacted]; SearchPath=bmwbd"
  }
}
```

Ensuite, dans le constructeur de l'application, on réalise l'injection :

```
public class Program
{
    public static void Main(string[] args)
    {
        var builder = WebApplication.CreateBuilder(args);

        // Add services to the container.

        builder.Services.AddControllers();
        // Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle
        builder.Services.AddEndpointsApiExplorer();
        builder.Services.AddSwaggerGen();

        // Injections
        builder.Services.AddDbContext<BMWContext>(options => options.UseNpgsql(builder.Configuration.GetConnectionString("BMWDBContext")));

        var app = builder.Build();

        // Configure the HTTP request pipeline.
        if (app.Environment.IsDevelopment())
        {
            app.UseSwagger();
            app.UseSwaggerUI();
        }
```

d. Utilisation d'un singleton

```
/// <example>
/// Ce code montre comment instancier la classe Singleton Crypt:
///     <code>
///         Crypt crypt = Crypt.Instance;
///     </code>
/// </example>
///
/// </summary>
7 références | LeoChris7, il y a 23 jours | 2 auteurs, 2 modifications
public class Crypt
{
    private static Crypt? _instance;
    private static readonly object _lock = new object();

    1 référence | LeoChris7, il y a 23 jours | 2 auteurs, 2 modifications
    public Crypt()
    {
    }

    /// <summary>
    /// Méthode statique permettant de créer une instance de Crypt
    /// </summary>
    2 références | Léo Christophe, il y a 27 jours | 1 auteur, 1 modification
    public static Crypt Instance
    {
        get
        {
            lock (_lock)
            {
                if (_instance == null)
                {
                    _instance = new Crypt();
                }
                return _instance;
            }
        }
    }
```

Un singleton est très utile quand on instancie qu'une seule fois une classe, évitant les conflits de ressources et garantissant qu'une seule instance de classe est créée.

Un singleton peut donc primordialement améliorer l'optimisation d'une application.

Exemple :

Figure 2 Création d'un singleton

2. Base de données

La base de données a été générée dans le principe du code first. C'est-à-dire qu'on a écrit le code en C# pour qu'il génère une base de données.

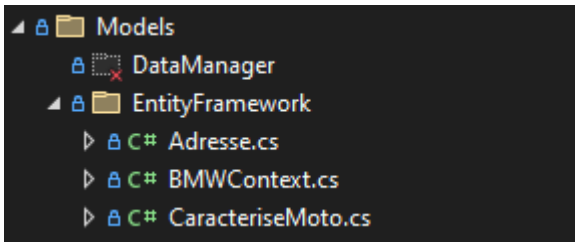
a. Principes du Code First

Setup

- Projet API Web ASP.NET Core

- Installation de divers packages :
  1. Npgsql.EntityFrameworkCore.PostgreSQL (Version 6.0.X)
  2. Npgsql.EntityFrameworkCore.PostgreSQL.Design
  3. Microsoft.EntityFrameworkCore.Tools (Version 6.0.X)

- Mis en place de l'architecture suivante :



On a créé diverses classes qui ont permis de décrire les propriétés de chaque entité SQL à générer et un contexte qui a permis de décrire les contraintes de clé primaire, de clé étrangère, de valeurs par défauts…

Création d'une migration

La création d'une migration est nécessaire pour valider la création de la base de données. Le principe est le suivant : On transmet le code first à la migration qui fait le lien avec la base de données.

La création de la base en code first est très utile car flexible, affichant les possibles erreurs, il est possible de générer un script SQL de création de base mais aussi de l'annuler.

Déroulement de la migration

Pour commencer une migration il est nécessaire d'utiliser dotnet -ef : dotnet tool install --global dotnet-ef --version 6.0.**x** de **même version** que le package Microsoft.EntityFrameworkCore.Tools.

Puisqu'on a fait de l'injection de dépendance pour la chaîne de connexion, on peut définir la base de données dans laquelle on va faire la migration comme ceci :

```
// Injections
builder.Services.AddDbContext<BMWContext>(options => options.UseNpgsql(builder.Configuration.GetConnectionString("BMWDBContext")));
```

Ajout d'une migration dans la console de gestionnaire des packages nuggets :

dotnet-ef migrations add 'migration_name' --project 'project_name'

Ajout de la migration

dotnet-ef database update 'migration_name' --project 'project_name'

La migration a bien été appliquée à la base de données.

3. Hashage du mot de passe

Une application bien protégée se doit d'avoir un mot de passe renforcé et sécurisé. Pour cela il est nécessaire de le hacher (et non de le crypter car on ne va pas le déchiffrer). Ainsi, nous avons choisi de crypter les mots de passes avec l'algorithme bcrypt. On a choisi bcrypt car c'est un algorithme renforcé et efficace.

a. Fonctionnement de Bcrypt

1. On génère un « sel » aléatoire. Cela rend chaque hachage unique.
2. On hache de multiples fois avec le sel.
3. Le sel et le hachage sont combinés en une seule chaîne de caractère. Cette chaîne va ensuite être utilisée pour vérifier si le mot de passe est correct.

4. Respect notable de certaines règles

a. Documentation

Tous les contrôleurs sont documentés, contenant généralement une description, les arguments possibles, la valeur de retour et les types de réponse possibles.

Exemple de documentation d'une méthode :

```
// PUT: api/Adresse/5
/// <summary>
/// Permet de modifier une adresse selon son ID de façon synchrone.
/// </summary>
/// <param name="id">ID de l'adresse</param>
/// <param name="adresse">adresse dont on veut les données</param>
/// <response code="404">Not found si aucune adresse n'a été trouvé.</response>
/// <response code="400">Bad Request si les IDs ne correspondent pas.</response>
/// <response code="204">No Content si tout va bien</response>
/// <returns>Le résultat de l'action</returns>
[HttpPut("{id}")]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
[ProducesResponseType(StatusCodes.Status404NotFound)]
[ProducesResponseType(StatusCodes.Status204NoContent)]
2 références | Hugo Clerc-Renaud, il y a 6 jours | 2 auteurs, 4 modifications
public async Task<IActionResult> PutAdresseAsync(int id, Adresse adresse)
{
    if (id != adresse.Id)
    {
        return BadRequest();
    }

    var adresseById = await dataRepository.GetByIdAsync(id);
    if (adresseById.Value == null)
    {
        return NotFound();
    }
    else
    {
        await dataRepository.UpdateAsync(adresseById.Value, adresse);
        return NoContent();
    }
}
```

b. *Commentaires*

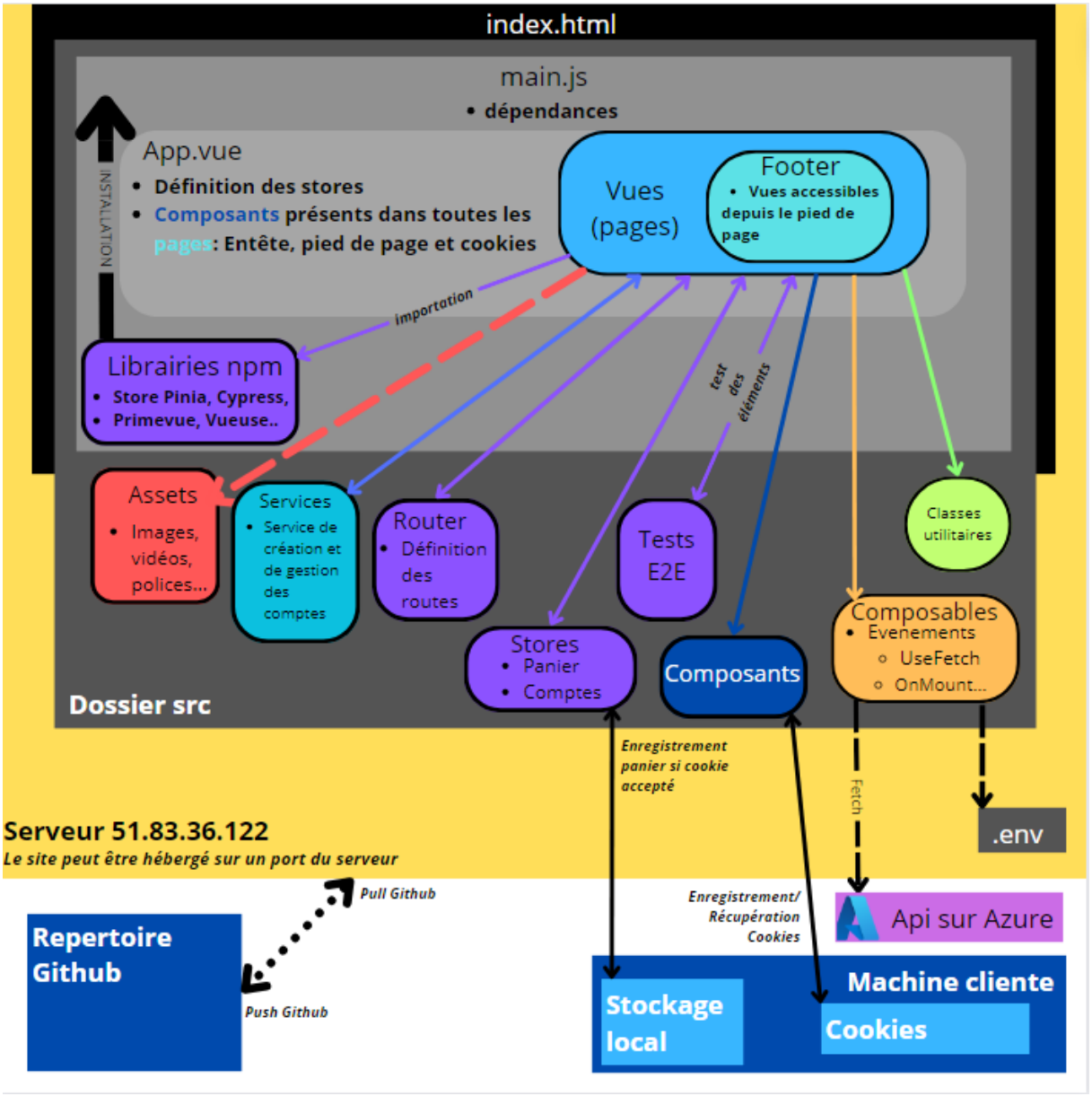
Certaines fois, les méthodes sont un peu compliquées donc des commentaires sont présents pour accompagner le développeur.

Exemple d'une méthode commentée :

```
/// <summary>
/// Permet à un utilisateur de se connecter grace à un compte client en arguments.
/// </summary>
/// <param name="login">Informations de connexion</param>
/// <returns>Réponse de la tentative de connexion.</returns>
/// <response code="200">Ok si l'utilisateur a été authentifié avec succès.</response>
/// <response code="401">Unauthorized si l'utilisateur n'a pas été authentifié.</response>
[HttpPost]
[AllowAnonymous]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status401Unauthorized)]
0 références | Léo Christophe, Il y a 1 heure | 1 auteur, 2 modifications
public IActionResult Login([FromBody] CompteClient login)
{
    // Réponse retournée par défaut: non autorisé.
    IActionResult response = Unauthorized();

    // Tentative d'authentification de l'utilisateur.
    CompteClient user = AuthenticateUser(login);

    // Si l'utilisateur a bien été authentifié, on procède.
    if (user != null)
    {
        // Génération d'un token de sécurité avec l'utilisateur en arguments
        var tokenString = GenerateJwtToken(user);

        // Retour d'une réponse positive 200
        response = Ok(new
        {
            token = tokenString,
            userDetails = user,
        });
    }
    // Retour d'une réponse négative Unauthorized (401).
    return response;
}
```

II. Le client réalisé en Vue.Js

1. Architecture



a. Séparation des fonctions

En programmation, séparer les fonctions est essentiel tant pour l'optimisation, que pour la lisibilité ou la cohérence. Ainsi, **on a séparé nos diverses méthodes javascript** et composants dans divers dossiers. Les exemples les plus courants sont :

- **Composants :** Contient des parties de code Javascript servant à factoriser. Par exemple, pour l'affichage d'une moto, on privilégiera un composant car on affichera plusieurs fois, dans différents contextes des motos.
- **Composables :** Contient des évènements. Dans notre cas, on utilise la librairie useFetch. Dans ce cas, on a accès à useFetch, permettant de fetch. En résultat, on a réalisé une classe useApiFetch qui contient une classe permettant de get (récupérer), post (créer), put (modifier)…

- **Views :** dossier contenant toutes les pages. Dans views, on a aussi le dossier footer sépare les vues avec un dossier exprès pour les vues accessibles depuis le pied de page.

b. *Bonnes pratiques :*

- **Utilisation de .env :** .env offre un moyen très optimal de stocker des variables que l'on veut récupérer dans tout le client, comme le lien de l'API.

- **Création d'une classe pour récupérer les données de l'API :** useApiData est une classe dans composable permettant d'automatiser la récupération des données, rendant l'appel dans les vues plus simple et plus lisible.

- **Création de fonctions utilitaires dans le fichier Utils.js** permettant habituellement de factoriser le code.

- **Utilisation de Github :** Très utile pour la collaboration, pour la sauvegarde du site et son déploiement.

2. Librairies

a. *Stores Pinia*

Store Pinia est une librairie installable. Elle consiste à garder les données de manière rapide sur toutes les pages. On utilise notamment les stores pour le panier d'achat et la connexion utilisateur.

Avantage	Inconvénient	Solution
Store Pinia offre une manière très facile d'enregistrer les données, et cela synchronisé dans toutes les pages.	Lorsqu'on rafraîchît la page, le store est réinitialisé.	L'utilisation des variables locales devient alors très utile puisqu'elles sont enregistrées au rafraichissement.

b. *Router*

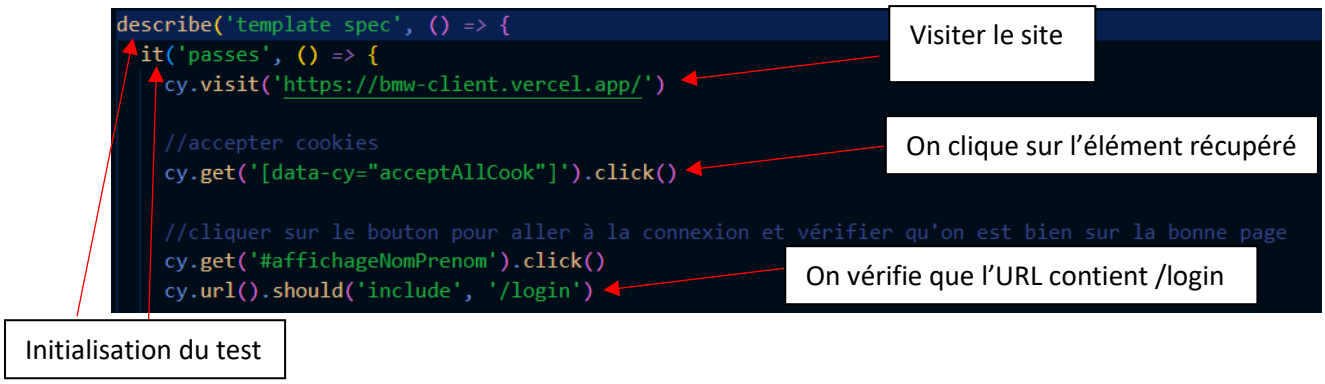
Instaure un système de « **navigation** » entre pages à travers l'application. On définit un fichier index.js où l'on définit les différents chemins. Ensuite, il suffit d'utiliser la syntaxe habituelle de router pour naviguer, revenir en arrière, créer des liens semblables aux balises <a>.

c. *Tests*

La syntaxe des librairies de test utilisés Cypress et Vitest est semblable au français

Cypress

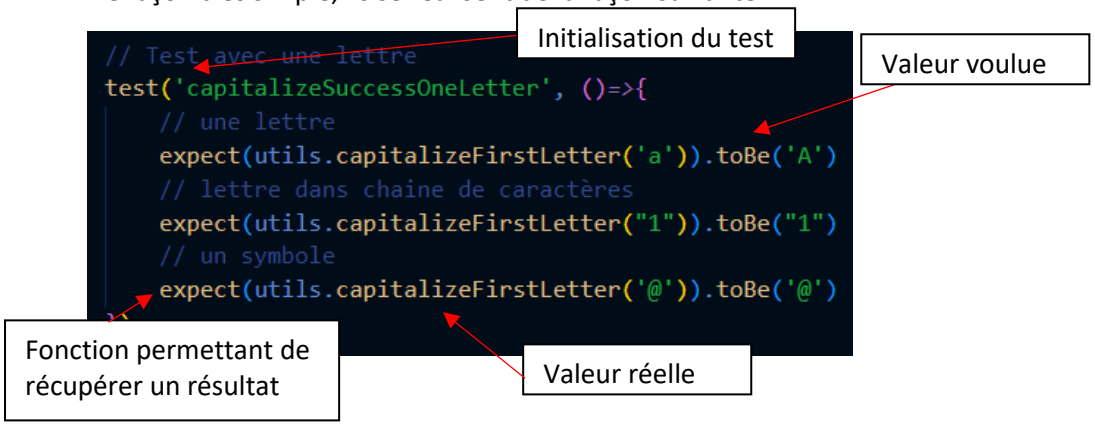
Les tests E2E (End to End) permettent de tester les interfaces et plus encore. La technologie utilisée est Cypress. De façon très simple, ils se déroulent de la façon suivante :



Vitest

Vitest est une libraire permettant de réaliser des tests unitaires. Ainsi, des tests unitaires ont été réalisé sur les méthodes utilitaires présentes dans Utils.js.

De façon très simple, ils se réalisent de la façon suivante :



3. Front End

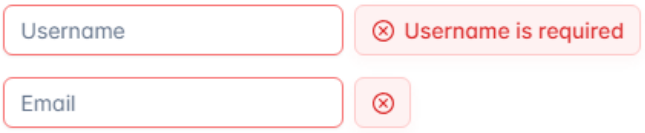
a. *Style.css*

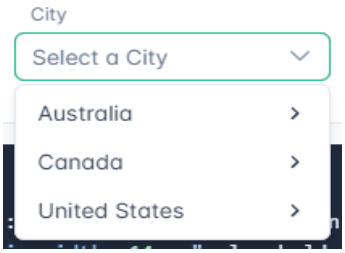
Lors de la création d'un projet Vue.Js, un fichier style.css est généré automatiquement. Ce fichier applique du style à toutes les vues, ce qui peut devenir problématique. Ainsi, style.css doit contenir que du style générique. Par exemple, si on veut toujours le même genre de bouton, on peut définir un style qui s'appliquera à tous les boutons, ou si on utilise plusieurs boutons différents, on peut définir différents IDs qui s'appliqueront dans toutes les pages.

b. *PrimeVue*

PrimeVue est une librairie qui permet d'utiliser des composants existants. Cela permet d'économiser du temps par rapport au style et des fois, gagner du temps par rapport aux fonctionnalités.

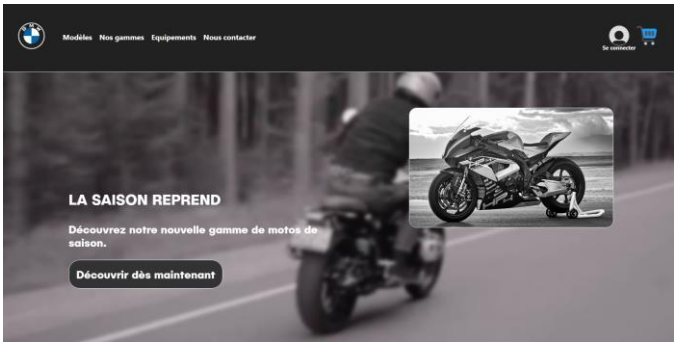
Certaines fois, on a eu des problèmes de types, impossible à régler donc pas tous les composants sont utilisables. De plus, il y a très peu d'aide en ligne donc c'est un peu difficile de débugger.





c. *Responsive*

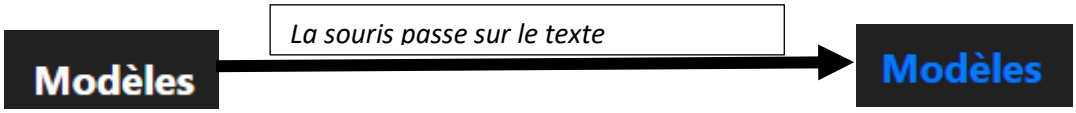
Notre site est responsive, le format sur ordinateur peut s'adapter aux smartphones.

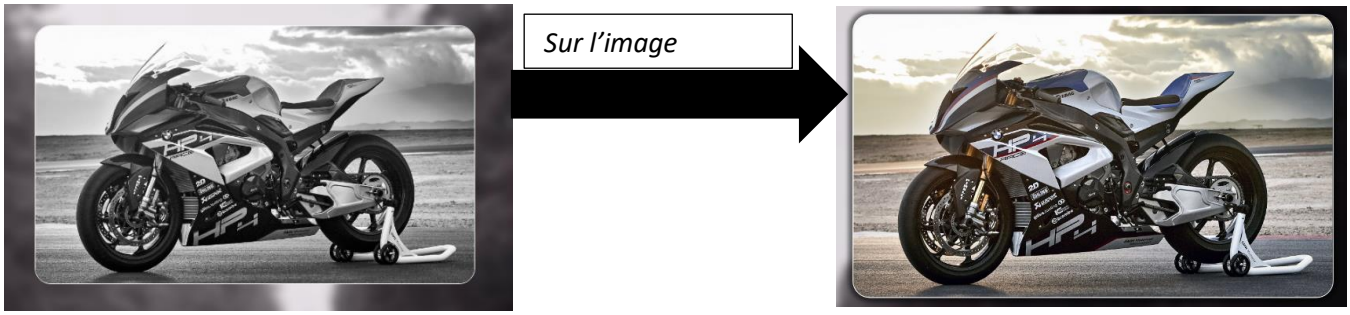


Ce passage au format smartphone est notamment souligné par le menu hamburger et le repositionnement des éléments de la page

d. Accessibilité et animations

Pour de meilleures indications sur quoi cliquer ou juste pour plus de beauté, il peut être nécessaire de faire des animations, notamment lorsqu'on passe la souris sur un objet. Voici quelques exemples :





4. Déploiement continu

Grâce à Github Actions, il est possible de faire des vérifications à chaque push. C'est notamment ce qu'on fait avec Cypress et Vitest.

a. Vérification des tests

On a créé un « workflow » permettant d'exécuter des commandes de terminal directement sur le projet à chaque fois que quelqu'un push le projet sur Github. Ainsi, on exécute à chaque fois **npx run cypress** permettant de démarrer les tests cypress et **npm run test** permettant de démarrer les tests vitest. Si les tests sont bons, une croix verte apparaît, sinon une croix rouge.

b. Déploiement sur Vercel

On utilise vercal.app pour déployer notre site. Ainsi, à chaque push Github, Vercel se met à jour à l'adresse suivante : BMW Motorrad (bmw-client.vercel.app).

III. La connexion utilisateur Client → API

L'utilisateur entre ses informations de connexion (email, mot de passe) dans le formulaire du site. Ses données sont envoyées au LoginController de l'API par le biais de Axios. L'API vérifie que le compte existe bien dans la base de données et renvoie à l'utilisateur son token JWT ainsi que ses informations de compte sous forme d'objet. Ces deux données sont enregistrées dans un store user.js. Le token JWT servant à accéder aux contrôleurs sensibles tel que CarteBancaireController.

Nous n'avons pas eu le temps de mettre en place le système d'enregistrement de carte bancaire mais si l'utilisateur voulait POST ou GET sa carte bancaire, il aurait fallu qu'il envoie son token dans le header de sa requête.

```
'Authorization': Bearer ${this.getBearerToken()}
```

IV. Les points à améliorer

API

- Vérification de la BDD : avant d'insérer les données, nous aurions dû vérifier si notre contexte avait bien généré la base de données (ex : clefs étrangères dans le mauvais sens).
- Modification de la BDD : certaines tables et champs de la BDD existantes était manquantes, nous aurions dû la modifier dès le début.

CLIENT

- Nomenclature : il aurait fallu nous mettre d'accord sur la langue à utiliser pour définir nos vues, composants et variables.
- CSS : nous aurions dû choisir une librairie (comme PrimeVue) plus vite pour éviter de s'attarder sur le CSS

CLIENT ET API

- Hébergement : nous aurions dû héberger notre API et notre Client plus tôt afin d'éviter les problèmes persistants de CORS qui sont apparus le jour du rendu
- On a pas pu réaliser le dossier sur l'impact environnemental de manière très correcte à cause d'un manque de connaissance.