DOSSIER TECHNIQUE SAE 4.01A TESLA

7 AVRIL

BUT INFO2

Créé par : BILLY Lucas, GENONCEAU Janelle, LANDRECY Enzo, LEICHTMANN Jimmy, RONDET Pierre

Logo
Nom

Table des matières

Structure **3**

Client : VueJS **3**

API : C# **12**

Structure

Client : VueJS

Le code du Front est composé de plusieurs parties :

- Le dossier api qui contient un fichier Contrôleur Manageur et plusieurs Contrôleurs.
- Le dossier Modèles comportant des classes Modèles
- Le dossier stores avec tous les stores non-permanents (refresh avec F5) et permanent
- Enfin les dossiers views et components comportant les vues et les composants utiles pour le site web.

Tout d'abord, nous allons parler du dossier api. L'architecture est la suivante, il y a une classe Mère appelée "ControllerManager.js" qui va définir la property statique "base url", une property "name" qui va être définie dans les classes filles, et faire hériter les méthodes présentes dans tous les contrôleurs (getAll, getById, delete, put, post).
La classe fille va hériter de "ControllerManager" et va définir la property "name" dans son constructeur qui va être utilisé pour les requêtes.

```js
import axios from 'axios';

class ControllerManager {
    static baseURL = "https://api-tesla-v2.azurewebsites.net/api";
    name; // Voir l'accès à la propriété name dans le constructeur public/private

    constructor() {}

    set name(value) {
        this.name = value;
    }

    get name() {
        return this.name;
    }

    GetAll() {
        return axios.get(`${ControllerManager.baseURL}/${this.name}`);
    }

    GetById(id) {
        return axios.get(`${ControllerManager.baseURL}/${this.name}/ById/${id}`);
    }

    Post(data) {
        return axios.post(`${ControllerManager.baseURL}/${this.name}`, data);
    }

    Put(id, data) {
        return axios.put(`${ControllerManager.baseURL}/${this.name}/${id}`, data);
    }

    Delete(id) {
        return axios.delete(`${ControllerManager.baseURL}/${this.name}/${id}`);
    }
}

export default ControllerManager;
```

La classe fille peut override les méthodes de la classe mère pour les adapter à ses besoins ou encore créer d'autre méthode n'existant pas dans la classe Mère.
Le controller est alors exporté en tant qu'objet ("export default new [Nomducontroller]()") et est importé dans le fichier "index.js" puis dans le store "index.js" (Cela évite juste la multitude de ligne d'import à mettre dans le store).

```
import axios from "axios";
import ControllerManager from "./ControllerManager";

class TypeOptionsController extends ControllerManager {
    constructor() {
        super();
        this.name = "typeoptions";
    }

    GetByName(name) {
        return axios.get(`${ControllerManager.baseURL}/${this.name}/ByName/${name}`);
    }

    GetByIdMotorisation(id){
        return axios.get(`${ControllerManager.baseURL}/${this.name}/ByIdMotorisation/${id}`);
    }
}

export default new TypeOptionsController();
```

Le dossier Modèles contient des classes modèles qui sont des classes qui vont définir les attributs d'un objet.
Ces classes vont être utilisées pour 3 raisons principales :
- Pour définir les attributs / trier un objet reçu par l'api (dans le controller)
- Pour définir des méthodes spécifiques à un objet (définir une classe de carousel ...)
- Pour avancer en parallèle de l'api et créer des objets qui n'existent pas encore dans l'api (dans le contrôleur)

```
class Modele {
    // Private
    idModele;
    nomModele;
    essayable;
    photo;

    get idModele() { return this.idModele; }
    get nomModele() { return this.nomModele; }
    get essayable() { return this.essayable; }
    get photo() { return this.photo; }

    set idModele(idModele) { this.idModele = idModele; }
    set nomModele(nomModele) { this.nomModele = nomModele; }
    set essayable(essayable) { this.essayable = essayable; }
    set photo(photo) { this.photo = photo; }

    constructor(idModele, nomModele, essayable, photo) {
        this.idModele = idModele;
        this.nomModele = nomModele;
        this.essayable = essayable;
        this.photo = photo;
    }

    static fromJson(json) {
        if(json == null)
            return null;
        else
            return new Modele(json.idModele, json.nomModele, json.essayable, json.photo);
    }

    static fromJsonArray(jsonArray) {
        if(jsonArray == null)
            return null;
        else
            return jsonArray.map(json => Modele.fromJson(json));
    }

    CarouselItem() {
        return { title: this.nomModele, link: `http://${this.photo.urlModel[0]}`}
    }

}

export default Modele;
```

En effet, la création des classes modèles va permettre de créer des objets à partir de la base (ou les classes Modèles du code first), sans que pour autant que les contrôleurs soient finis. Mais encore s'il y a une mise à jour de la base de données, modifiant des champs, il suffira de la modifier une fois dans la classe modèle.

Le dossier stores contient tous les stores de l'application. Il y a 2 types de stores :
- Les stores non-permanent. (refresh avec F5)
- Les stores permanents (ne refresh pas avec F5)

Les non-permanents sont les stores qui ont juste une utilité à un moment t de l'application.
Par exemple dans le fichier "index.js" il y a deux variables stores :
- "controller" qui contient toutes les instances des contrôleurs (Cela évite qu'à chaque appel cela créer une nouvelle instance. (impossible à stocker dans le localStorage))
- "request" qui contient plusieurs informations sur l'état de la requête (access(), success(), error() ...)

```js
import api from '../api'
import { defineStore } from 'pinia'
import { ref } from 'vue'

const controller = defineStore( 'controller', () => {
    const AccessoiresController = api.AccessoiresController
    const ModelesController = api.ModelesController;
    const MotorisationsController = api.MotorisationsController;
    const OptionsController = api.OptionsController;
    const CaracteristiquesController = api.CaracteristiquesController;
    const PhotosController = api.PhotosController;
    const TypeOptionsController = api.TypeOptionsController;
    const ComptesController = api.ComptesController;
    const VehiculeDemonstrationsController = api.VehiculeDemonstrationsController;
    const VariantesController = api.VariantesController;
    return { AccessoiresController, ModelesController, MotorisationsController, OptionsController, CaracteristiquesController, PhotosController, TypeOptionsController,
        ComptesController, VehiculeDemonstrationsController, VariantesController }}
)

const request = defineStore( 'request', {

    state: () => {
        return {
            requestState: true, // StatusState de la requête | false = requête en cours | true = requête terminée
            requestCode: '', // Code de la requête | 200 OK | 404 Not Found | 500 Internal Server Error
            requestError: false, // Erreur de la requête | false = pas d'erreur | true = erreur
            typeRequest: '', // Type de la requête | GET | POST | PUT | DELETE
            toastinfo: ref([]), // Toast info
        }
    },
```

```
    actions: {
        success(response) { // Requete terminée avec succès
            this.requestState = true;
            this.typeRequest = response.config.method;
            this.requestCode = `${response.status} ${response.statusText}`;
        },
        access(){ // Requete en cours | Clear les données
            this.requestState = false;
            this.requestError = false;
            this.requestCode = '';
            this.typeRequest = '';
        },
        pass() {
            this.requestState = true;
        },
        error(error) { // Requete terminée avec erreur
            this.requestState = true;
            this.requestError = true;
            this.typeRequest = 'ERROR CODE';
            this.requestCode = 'ERROR CODE';
            if(error.config)
                this.typeRequest = error.config.method;
            if(error.response)
                this.requestCode = `${error.response.status} ${error.response.statusText}`;
            // Add toast
            let index = this.toastinfo.length;
            this.toastinfo.push({id: index, type: `alert-error`, message: `${this.requestCode}`});
        },
        addAlert( type, message ) {
            let id = this.toastinfo.length;
            this.toastinfo.push({id, type, message });
        },
        removeAlert( alert ) {
            let index = this.toastinfo.indexOf(this.toastinfo.find(a => a.id == alert.id));
            this.toastinfo.splice(index, 1);
        },
        debug(){ // Debug
            console.log(`RequestState : ${this.requestState}`);
            console.log(`Error : ${this.requestError}`);
            console.log(`Type of request : ${this.typeRequest}`);
            console.log(`Request code result : ${this.requestCode}`);
        }
    },
});

export {request,controller};
```

Ces deux variables sont utiles entre la navigation des pages, mais ne sont pas utiles tout le temps. C'est pourquoi ils sont stockés dans un store non-permanent.

Les stores permanents sont les stores qui sont utiles tout le temps.

Par exemple, le store "compte" qui contient les informations de l'utilisateur connecté. Ce store est donc permanent (stocké dans le localStorage).

```
import { defineStore } from 'pinia'
import { useStorage, StorageSerializers } from '@vueuse/core'
import router from '../router';

const store_compte = defineStore( 'store-compte', {
    state: () => {
        return {
            _compte: useStorage('compte', null, localStorage, {serializer: StorageSerializers.object}),
            _token: useStorage('token', null),
        }
    },
    getters: {
        compte: (state) => state._compte,
        name: (state) => {
            if(!state._compte)
                return 'Compte';
            else
                return state._compte.nomCompte
        },
        token: (state) => state._token,
    },
```

```
    actions: {
        // Compte
        login(compte, token) {
            console.log(compte);
            this._compte = compte;
            this._token = token;
            router.push('/');
        },
        editCompte(compte) {
            this._compte = compte;
            router.go(0);
        },
        logout() {
            this._compte = null;
            this._token = null;
            router.push('/');
        },
        menu() {
            return {
                'name': this.name,
                'links' : [
                    (this._compte)?{
                        'name': 'Profile',
                        'link': '/profile'
                    }:{'hidden': 'true'},
                    (!this._compte)?{
                        'name': 'Connection',
                        'link': '/login'
                    }:{'hidden': 'true'},
                    (!this._compte)?{
                        'name': 'Créer un Compte',
                        'link': '/createaccount'
                    }:{'hidden': 'true'},
                    (this._compte)?{
                        'name': 'Logout',
                        'event' : 'logout'
                    }:{'hidden': 'true'},
                ]
            }
        }
    },
})
export { store_compte }
```

Voici la liste des stores créer et leurs utilités :

N = Non permanent

P = Permanent

Nom fichier	Variables stores	Méthode Store	Utilité	Type store
Compte.js	store_compte	Login() editCompte() logout() menu()	Stocke les données du compte et le token	P
index .js	Controller request	Sucess() Access() Pass() Error() addAlert() removeAlert() debug()	Stocke les controller ainsi que les états des requêtes et alertes	N
Panier.js	store_panier	addPanier() removePanier()	Stocke le panier de l'utilisateur	P
Pdf.js	pdf	Save()	Permet la transmission des données utiles pour le pdf	N
Saves.js	saves	Save() findValue() findKey()	C'est un store d'optimisation, il stocke les grosses données afin de faire un chargement qu'une fois.	N

Enfin le dossier views et components contient les vues et les composants de l'application. Les vues sont les pages de l'application et les composants sont des éléments réutilisable dans plusieurs vues.
Les vues sont incluses dans un layout défini dans l'App.vue (qui est la vue principale de l'application) à la racine du dossier src.
Les vues sont très différentes en fonction de l'utilisation, mais l'App.vue va permettre d'intégrer sur toutes les pages :
- Une Navbar
- Un Footer
- Le fonctionnement de loading screen
- Le fonctionnement de la notification d'erreur ou de succès (toast)

API : C#

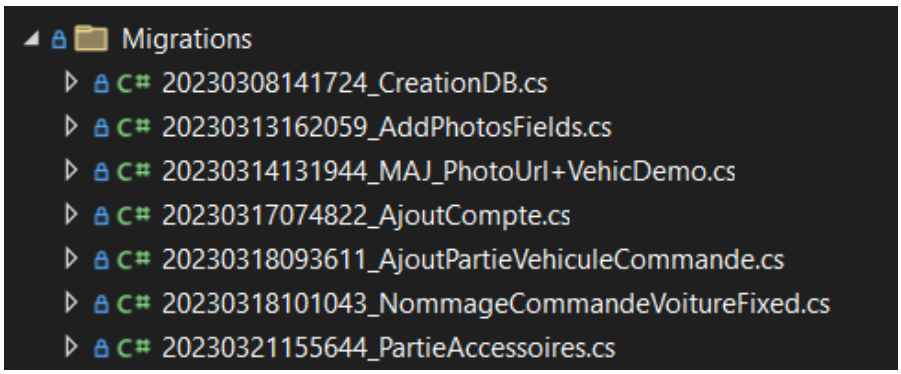
Développement :

Côté back, une API développé en C#. NET Core 6 permet de faire le lien entre le client et notre base de données. Celle-ci est développé en Code First, c'est-à-dire que pour chaque classe modèle correspond à une table dans la base.

```
[Table("t_e_photo_pho")]
public partial class Photo
{
    [Column("pho_id")]
    [Key]
    public int IdPhoto { get; set; }

    [Column("pho_nomcouleur")]
    public string NomCouleur { get; set; }

    [Column("pho_codehexa")]
    public string CodeHexa { get; set; }

    [Column("pho_url")]
    public List<string> Url { get; set; }
```

Les champs sont nommés selon la norme ISO/CEI 9075

Cette méthode de développement est rendue possible grâce à l'Entity Framework Core, qui permet également de générer des migrations pouvant être converties en script SQL afin de mettre à jour une base de données (ici, PostgreSQL).
Ainsi, pour chaque grande partie de développement, une migration est générée :



Afin de réduire le nombre de classes métiers à générer, les tables de jointures Many to Many ne comprenant pas de champs supplémentaires sont générés automatiquement dans la classe TeslaDBContext. Cela nous a permis de gagner du temps et de simplifier l'architecture de l'API.

```
entity.HasMany(d => d.VedCommandesVoitureNavigation).WithMany(p => p.CvtVehiculesDemonstrationNavigation)
    .UsingEntity<Dictionary<string, object>>(
      "VehiculeDemonstrationCommande",
       l => l.HasOne<CommandeVoiture>().WithMany().HasForeignKey("IdCommandeVoiture").OnDelete(DeleteBehavior.Restrict).HasConstraintName("fk_vehiculedemonstrationcommande_idcommandevoiture"),
       r => r.HasOne<VehiculeDemonstration>().WithMany().HasForeignKey("IdVehiculeDemo").OnDelete(DeleteBehavior.Restrict).HasConstraintName("fk_vehiculedemonstrationcommande_idvehiculedemo"),
       j =>
       {
           j.HasKey("IdCommandeVoiture", "IdVehiculeDemo").HasName("pk_vehiculedemonstrationcommande");
           j.ToTable("t_j_vehiculedemonstrationcommande_vdc");
           j.Property("IdCommandeVoiture").HasColumnName("vdc_idcommandevoiture");
           j.Property("IdVehiculeDemo").HasColumnName("vdc_idvehiculedemo");
       });
```

Code dans le DBContext

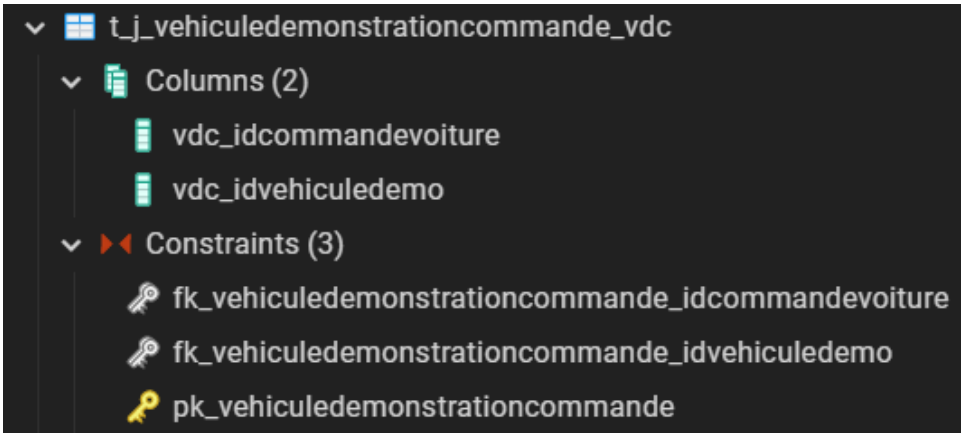


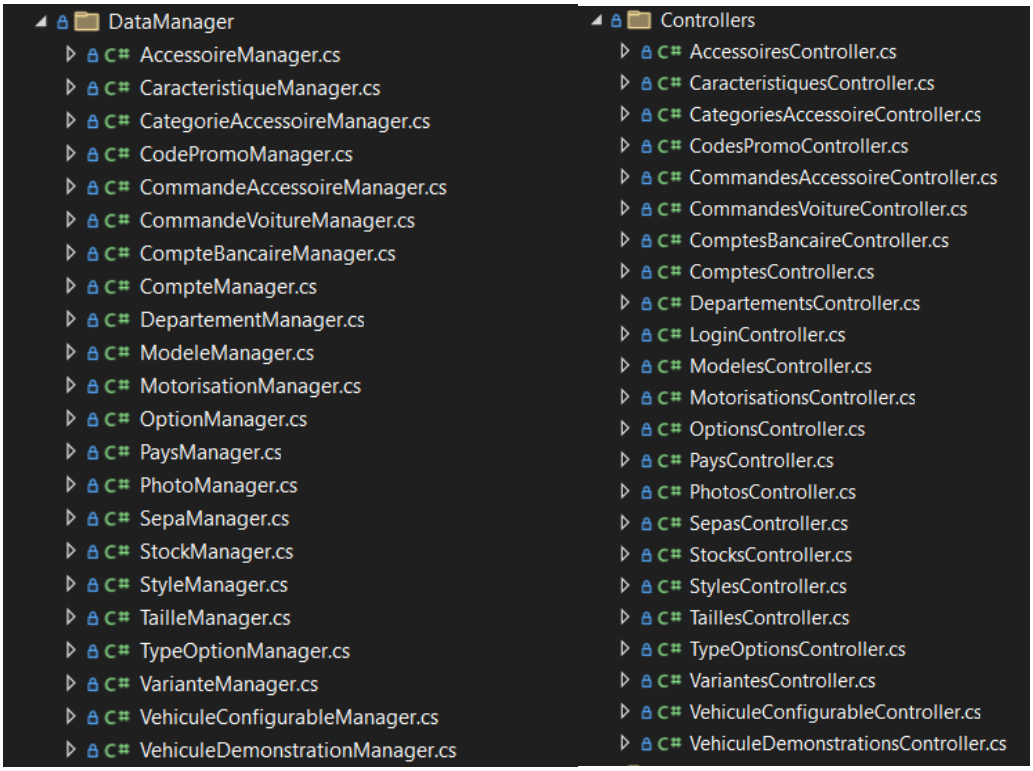
Table autogénérée dans la base PostgreSQL

Toujours dans notre volonté d'appliquer les bonnes pratiques de programmation, nous avons décidé d'appliquer le pattern DataRepository.
De plus, pour ajouter des méthodes propres à une seule classe, nous avons mis en place un système d'héritage d'interfaces.
Ainsi, pour intégrer une méthode GetByIdSepa dans le controller Comptes, on a une interface IDataRepositoryCompte héritant de l'interface IDataRepository classique et qui implémente également les nouvelles méthodes souhaitées.

```
using Microsoft.AspNetCore.Mvc;

namespace API_Tesla.Models.Repository
{
    public interface IDataRepository <TEntity>
    {
        Task<ActionResult<IEnumerable<TEntity>>> GetAllAsync();
        Task<ActionResult<TEntity>> GetByIdAsync(int id);
        Task<ActionResult<TEntity>> GetByStringAsync(string str);
        Task AddAsync(TEntity entity);
        Task UpdateAsync(TEntity entityToUpdate, TEntity entity);
        Task DeleteAsync(TEntity entity);
    }
}
```

```
namespace API_Tesla.Models.Repository
{
    public interface IDataRepositoryCompte<TEntity> : IDataRepository<TEntity>
    {
        Task<ActionResult<IEnumerable<TEntity>>> GetByIdSepaAsync(int idSepa);
    }
}
```

Ces méthodes sont ensuite implémentées dans les Managers associées, puis appelées dans l'API via les controllers correspondants, qui sont au nombre de **23** :



Ce pattern inclut également les injections de dépendances :

```
builder.Services.AddScoped<IDataRepositoryCaracteristique<Caracteristique>, CaracteristiqueManager>();
builder.Services.AddScoped<IDataRepository<Modele>, ModeleManager>();
builder.Services.AddScoped<IDataRepositoryMotorisation<Motorisation>, MotorisationManager>();
builder.Services.AddScoped<IDataRepositoryOption<Option>, OptionManager>();
builder.Services.AddScoped<IDataRepositoryTypeOption<TypeOption>, TypeOptionManager>();
builder.Services.AddScoped<IDataRepository<VehiculeDemonstration>, VehiculeDemonstrationManager>();
builder.Services.AddScoped<IDataRepositoryPhoto<Photo>, PhotoManager>();
builder.Services.AddScoped<IDataRepositoryCompte<Compte>, CompteManager>();
builder.Services.AddScoped<IDataRepository<CompteBancaire>, CompteBancaireManager>();
builder.Services.AddScoped<IDataRepository<Sepa>, SepaManager>();
builder.Services.AddScoped<IDataRepositoryDepartement<Departement>, DepartementManager>();
builder.Services.AddScoped<IDataRepository<Pays>, PaysManager>();
builder.Services.AddScoped<IDataRepository<VehiculeConfigurable>, VehiculeConfigurableManager>();
builder.Services.AddScoped<IDataRepository<CommandeVoiture>, CommandeVoitureManager>();
builder.Services.AddScoped<IDataRepository<CommandeAccessoire>, CommandeAccessoireManager>();
builder.Services.AddScoped<IDataRepository<CodePromo>, CodePromoManager>();
builder.Services.AddScoped<IDataRepository<CategorieAccessoire>, CategorieAccessoireManager>();
builder.Services.AddScoped<IDataRepository<Accessoire>, AccessoireManager>();
builder.Services.AddScoped<IDataRepository<Taille>, TailleManager>();
builder.Services.AddScoped<IDataRepository<Style>, StyleManager>();
builder.Services.AddScoped<IDataRepository<Stock>, StockManager>();
builder.Services.AddScoped<IDataRepository<Variante>, VarianteManager>();
```

Sécurisation :

Certaines méthodes de certains controllers peuvent abriter des données sensibles.
Leur accès doit donc être réglementé.
L'API est donc sécurisée via des tokens JWT.
Par exemple, pour utiliser la méthode GetAll du controller Comptes, exigeant un token de niveau admin, il faut préciser en en-tête de la requête le token, sans quoi la méthode renverra un code 401 Unauthorized.

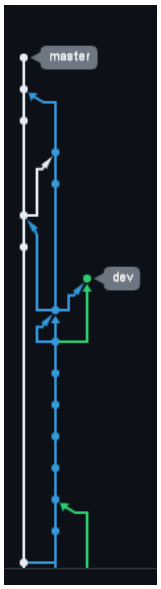
```
// GET: api/Comptes
[HttpGet]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status404NotFound)]
[Authorize(Policy = Policies.Admin)]
public async Task<ActionResult<IEnumerable<Compte>>> GetComptes()
{
    return await dataRepository.GetAllAsync();
}
```

L'accès à ce token se fait lors de la connexion à un compte.
À chaque compte est associé une propriété UserRole de valeur « User » ou « Admin ».

```
[Column("cpt_userrole")]
public string UserRole { get; set; }
```

Un token associé au rôle est renvoyé au VueJS et donne accès à plus de méthodes.
Par exemple, un utilisateur connecté a par exemple accès aux informations de son compte, seuls les comptes administrateur ont accès aux méthodes Delete de tous les controllers.

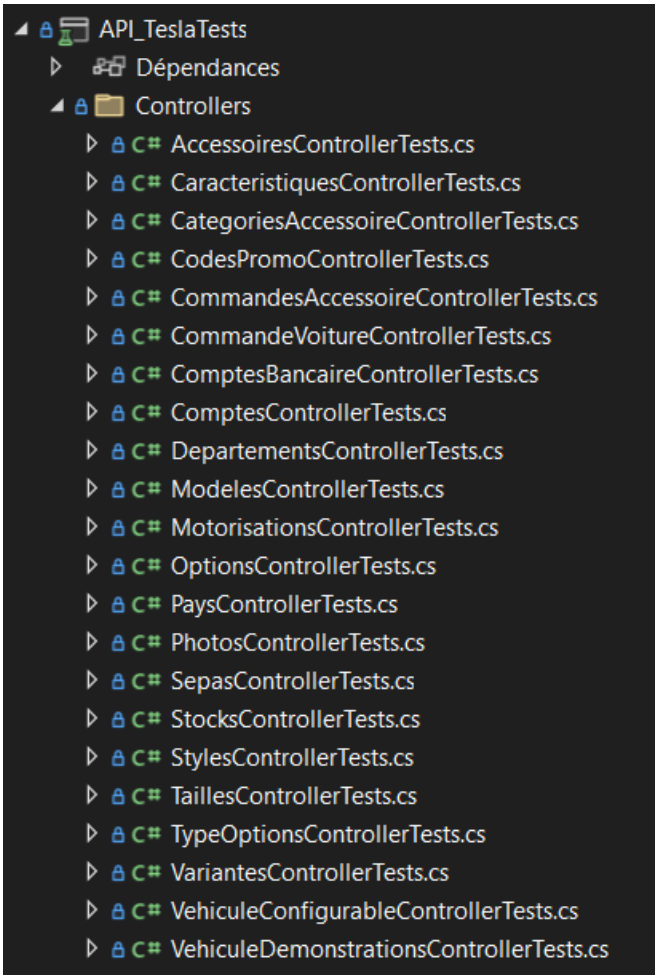
Conduite de projet :



Notre API est hébergée sur Github et déployée en continu via Azure Resource Manager.
Cela signifie que pour chaque commit, l'API est build puis redéployée sur Azure.
Afin d'éviter de trop solliciter cette fonctionnalité, nous avons créé une branche secondaire nommée « dev », où les fonctionnalités en développement seront commit.
Cette branche est merged à la fin de chaque semaine après validation des membres de groupe et des tests concernant les nouvelles fonctionnalités.

Tests unitaires :

Nous avons effectué 290 tests unitaires répartis sur 22 controllers.
Sur ces tests, 180 ont été réalisé en Mock (tests des POST/DELETE/PUT) afin de simuler l'appel à la base et ne pas la polluer avec des opérations d'insert, delete et update.




```
#region Tests Get
[TestMethod()]
public void GetCodesPromoTest()
{
    // Arrange
    List<CodePromo> codespromo = context.CodesPromo.ToList();

    // Act
    var result = controller.GetCodesPromo().Result;

    // Assert
    Assert.IsInstanceOfType(result, typeof(ActionResult<IEnumerable<CodePromo>>));
    Assert.IsNull(result.Result);

    // Act
    List<CodePromo> codespromoresult = result.Value.ToList();

    // Assert
    CollectionAssert.AreEqual(codespromo, codespromoresult);
}
```

```
#region Tests Put
[TestMethod()]
public void PutCodePromoTest_Mock_HttpResponse204()
{
    // Arrange
    CodePromo codePromoToChange = new CodePromo()
    {
        IdCodePromo = 1,
        LibelleCodePromo = "SerieTest",
        DateDebut = new DateTime(2023, 8, 22),
        DateFin = new DateTime(2023, 10, 23),
        Pourcentage = 0.2
    };

    CodePromo codePromoChanged = new CodePromo()
    {
        IdCodePromo = 1,
        LibelleCodePromo = "SerieTest",
        DateDebut = new DateTime(2023, 8, 22),
        DateFin = new DateTime(2023, 10, 23),
        Pourcentage = 0.4
    };

    var mock = new Mock<IDataRepository<CodePromo>>();
    mock.Setup(x => x.GetByIdAsync(1).Result).Returns(codePromoToChange);
    var mockController = new CodesPromoController(mock.Object);

    // Act
    var result = mockController.PutCodePromo(1, codePromoChanged).Result;

    // Assert
    Assert.IsInstanceOfType(result, typeof(NoContentResult));
}
```

```
#region Tests Post
[TestMethod()]
public void PostCodePromoTest_Mock_HttpResponse201()
{
    // Arrange
    CodePromo code = new CodePromo()
    {
        LibelleCodePromo = "SerieTest",
        DateDebut = new DateTime(2023, 8, 22),
        DateFin = new DateTime(2023, 10, 23),
        Pourcentage = 0.2
    };

    var mock = new Mock<IDataRepository<CodePromo>>();
    var mockController = new CodesPromoController(mock.Object);

    // Act
    var actionResult = mockController.PostCodePromo(code).Result;

    // Assert
    Assert.IsInstanceOfType(actionResult, typeof(ActionResult<CodePromo>));
    Assert.IsNull(actionResult.Value);
    Assert.IsInstanceOfType(actionResult.Result, typeof(CreatedAtActionResult));

    var result = (CreatedAtActionResult)actionResult.Result;
    Assert.IsInstanceOfType(result.Value, typeof(CodePromo));
    code.IdCodePromo = ((CodePromo)result.Value).IdCodePromo;
    Assert.AreEqual(code, (CodePromo)result.Value);
}
```

```
#region Tests Delete
[TestMethod()]
public void DeleteCodePromoTest_Mock_HttpResponse204()
{
    // Arrange
    CodePromo code = new CodePromo()
    {
        IdCodePromo = -1,
        LibelleCodePromo = "SerieTest",
        DateDebut = new DateTime(2023, 8, 22),
        DateFin = new DateTime(2023, 10, 23),
        Pourcentage = 0.2
    };

    var mock = new Mock<IDataRepository<CodePromo>>();
    mock.Setup(x => x.GetByIdAsync(1).Result).Returns(code);
    var mockController = new CodesPromoController(mock.Object);

    // Act
    var result = mockController.DeleteCodePromo(1).Result;

    // Assert
    Assert.IsInstanceOfType(result, typeof(NoContentResult));
}
```

Base de données :

Notre base PostgreSQL est hébergée sur Azure à l'adresse suivante :
sae401.postgres.database.azure.com.
Login : sae401_admin
MDP : teslaIUT!

Elle se compose de 39 tables, ainsi qu'une table auto-générée pour historiser les migrations nommé _EFMigrationsHistory. (Voir annexe pour le MPD complet).