

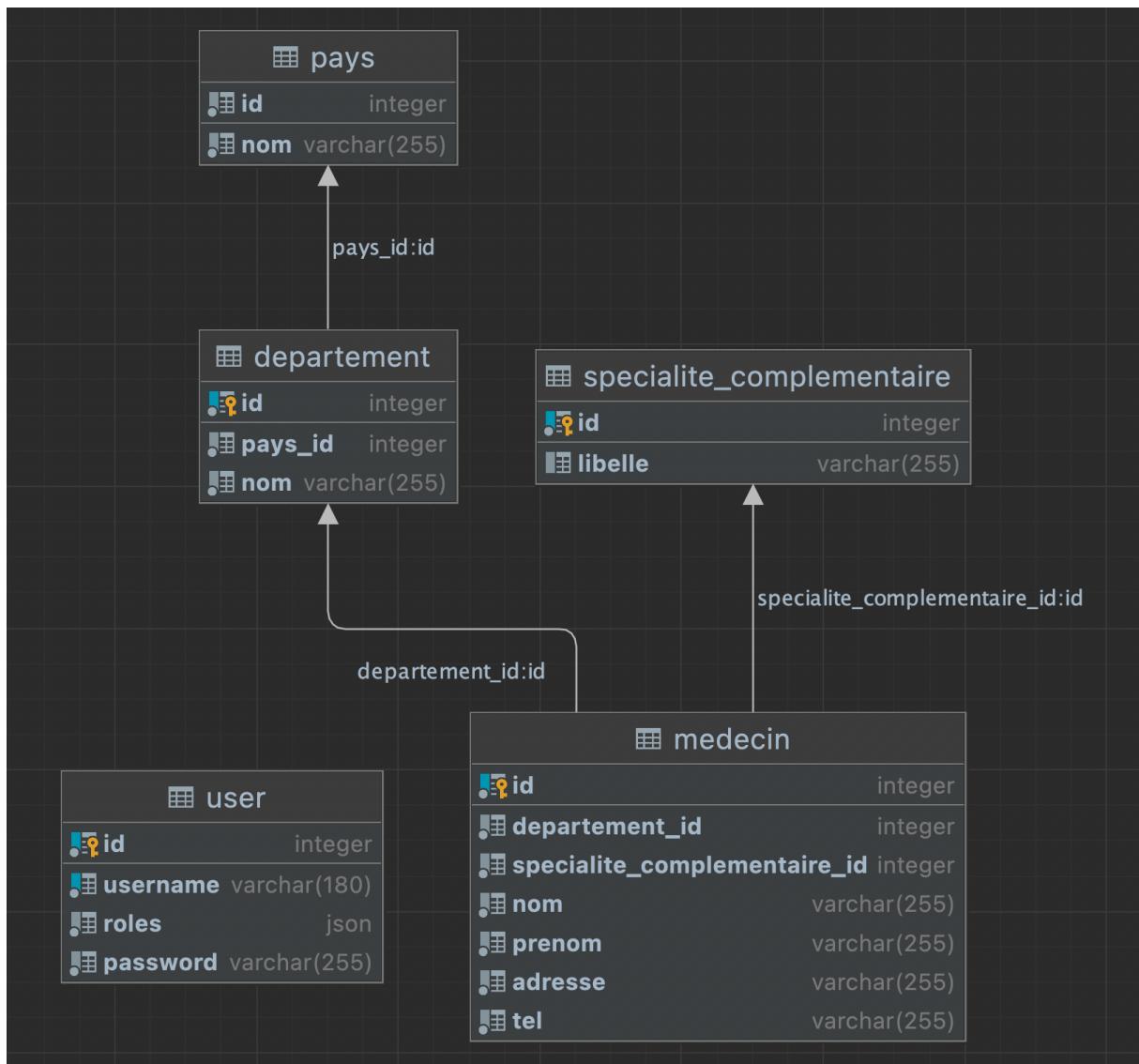
# GSB Médecins

Documentation technique

## Présentation

Le projet GSB Médecins est composé de deux applications : une API REST avec un client web basé sur Symfony et une application mobile Flutter qui consomme l'API. Le SGBD utilisé est PostgreSQL.

## Structure des données



## Application Symfony

L'application Symfony se charge de l'API REST pour l'application mobile et du client Web d'administration.

## Structure de l'application

```
src App\
  └── Controller
      ├── .gitignore
      ├── DepartementController.php
      ├── IndexController.php
      ├── LoginController.php
      ├── MedecinController.php
      ├── PaysController.php
      └── SpecialiteComplementaireController.php
  └── Entity
      ├── .gitignore
      ├── Departement.php
      ├── Medecin.php
      ├── Pays.php
      ├── SpecialiteComplementaire.php
      └── User.php
  └── Form
      ├── DepartementType.php
      ├── MedecinType.php
      ├── PaysType.php
      └── SpecialiteComplementaireType.php
  └── Repository
      ├── .gitignore
      ├── DepartementRepository.php
      ├── MedecinRepository.php
      ├── PaysRepository.php
      ├── SpecialiteComplementaireRepository.php
      ├── UserRepository.php
      └── Kernel.php
  └── templates
      > departement
      > index
      > login
      > medecin
          ├── _delete_form.html.twig
          ├── _form.html.twig
          ├── edit.html.twig
          ├── index.html.twig
          ├── new.html.twig
          └── show.html.twig
      > pays
      > specialite_complementaire
          └── base.html.twig
```

## Description des classes

### Les entités

Ce sont les objets métier de l'application, la plupart représentent une table dans la base de données.

```
#ORM\Entity(repositoryClass: MedecinRepository::class)
class Medecin
{
    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column(type: 'integer')]
    private $id;

    #[ORM\Column(type: 'string', length: 255)]
    #[Assert\NotBlank(message: "Cette valeur ne doit pas être vide.")]
    private $nom;

    #[ORM\Column(type: 'string', length: 255)]
    #[Assert\NotBlank(message: "Cette valeur ne doit pas être vide.")]
    private $prenom;

    #[ORM\Column(type: 'string', length: 255)]
    #[Assert\NotBlank(message: "Cette valeur ne doit pas être vide.")]
    private $adresse;

    #[ORM\Column(type: 'string', length: 255)]
    #[Assert\NotBlank(message: "Cette valeur ne doit pas être vide.")]
    #[Assert\Regex(
        pattern: "/^(\\+)(297|93|244|1264|358|355|376|971|54|374|1684|1268|61|43|994|257|32|229|226|880|359|973|1242|\\s+){1}([0-9]{1,15})$",
        message: "Le numéro de téléphone doit être saisis au format international sans le 0. Ex : +33123456789")]
    private $tel;

    #[ORM\ManyToOne(targetEntity: Departement::class, inversedBy: 'medecins')]
    #[ORM\JoinColumn(nullable: false)]
    #[Assert\NotBlank(message: "Cette valeur ne doit pas être vide.")]
    private $departement;

    #[ORM\ManyToOne(targetEntity: SpecialiteComplementaire::class, inversedBy: 'medecins')]
    private $specialiteComplementaire;
```

On utilise des annotations PHP pour configurer les entités. Ainsi, l'ORM (Doctrine) se chargera de générer la structure de la base de données et gérer l'accès aux données à l'aide des repositories.

Les annotations tel que `#[Assert\NotBlank]` ou `#[Assert\Regex]` permettent d'assurer la validation de l'objet et de retourner les éventuelles erreurs.

Les annotations tel que `#[ORM\ManyToOne]` ou `#[ORM\OneToMany]` permettent de définir les relations entre les entités et de générer les clés étrangères dans la base de données.

### Les contrôleurs

Les contrôleurs permettent de définir les endpoints de notre application Web, ainsi qu'envoyer et recevoir les données des templates (vues en html). Le contrôleur appelle ensuite les services pour charger et enregistrer les données.

```

#[Route('/medecins')]
#[IsGranted('ROLE_ADMIN')]
class MedecinController extends AbstractController
{
    #[Route('/', name: 'medecin_index', methods: ['GET'])]
    public function index(Request $request, MedecinRepository $medecinRepository, LoggerInterface $logger): Response
    {
        $name = $request->query->get('name');
        return $this->render('medecin/index.html.twig', [
            'medecins' => $medecinRepository->findByNomOrPrenomContaining($name),
        ]);
    }

    #[Route('/new', name: 'medecin_new', methods: ['GET', 'POST'])]
    public function new(Request $request, EntityManagerInterface $entityManager): Response
    {
        $medecin = new Medecin();
        $form = $this->createForm(MedecinType::class, $medecin);
        $form->handleRequest($request);

        if ($form->isSubmitted() && $form->isValid()) {
            $entityManager->persist($medecin);
            $entityManager->flush();

            return $this->redirectToRoute('medecin_index', [], Response::HTTP_SEE_OTHER);
        }

        return $this->renderForm('medecin/new.html.twig', [
            'medecin' => $medecin,
            'form' => $form,
        ]);
    }
}

```

Les annotations `#[Route` permettent de définir les routes ainsi que les verbes HTTP (GET, POST etc...)

## Les vues

Le moteur de template utilisé par Symfony est Twig. Il fonctionne avec un système de block qu'on importe dans les fichiers puis on utilise des accolades pour définir les variables chargées depuis les contrôleurs.

```
{% extends 'base.html.twig' %}

{% block title %}Médecins{% endblock %}

{% block searchBar %}
    <input aria-label="Search" class="form-control form-control-dark w-100" id="search"
        placeholder="Rechercher un médecin..." name="name" type="text">
{% endblock %}

{% block body %}
    <div class="d-flex justify-content-between flex-wrap flex-mdnowrap align-items-center pt-3 pb-2 mb-3 border-bottom">
        <h1 class="h2">Médecins</h1>
        <div class="container p-0 m-0">
            <div class="row justify-content-end p-0 m-0">
                <div class="col-auto me-0 pe-0">
                    <a class="btn btn-success" href="{{ path('medecin_new') }}>Nouveau</a>
                </div>
            </div>
        </div>
    </div>
</div>

<table class="table">
    <thead>
        <tr>
            <th scope="col">#</th>
            <th scope="col">Nom</th>
            <th scope="col">Prénom</th>
            <th scope="col">Adresse</th>
            <th scope="col">Tel</th>
            <th scope="col">Actions</th>
        </tr>
    </thead>
    <tbody>
        {% for medecin in medecins %}<br>
            <tr>
                <td>{{ medecin.id }}</td>
                <td>{{ medecin.nom }}</td>
                <td>{{ medecin.prenom }}</td>
                <td>{{ medecin.adresse }}</td>
```

## API REST

L'API REST est implémentée sur l'application Symfony par le biais d'API Platform.

### Configuration des endpoints

Avec API Platform, la configuration des endpoints REST se fait via des annotations.

```
#[ORM\Entity(repositoryClass: MedecinRepository::class)]
#[ApiResource(
    collectionOperations: [
        'get' => ['normalization_context' => ['groups' => ['medecins:read']]],
    ],
    itemOperations: [
        'get' => ['normalization_context' => ['groups' => ['medecin:read']]],
    ],
)]
class Medecin
{
    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column(type: 'integer')]
    #[Groups(['departement:read", "medecin:read", "medecins:read", "specialite_complementaire:read", "departements:read"])]
    private $id;

    #[ORM\Column(type: 'string', length: 255)]
    #[Groups(['departement:read", "medecin:read", "medecins:read", "specialite_complementaire:read"])]
    #[Assert\NotBlank(message: "Cette valeur ne doit pas être vide.")]
    private $nom;

    #[ORM\Column(type: 'string', length: 255)]
    #[Groups(['departement:read", "medecin:read", "medecins:read", "specialite_complementaire:read"])]
    #[Assert\NotBlank(message: "Cette valeur ne doit pas être vide.")]
    private $prenom;

    #[ORM\Column(type: 'string', length: 255)]
    #[Groups(['departement:read", "medecin:read", "medecins:read", "specialite_complementaire:read"])]
    #[Assert\NotBlank(message: "Cette valeur ne doit pas être vide.")]
    private $adresse;
```

#[ApiResource] permet d'exposer l'entité comme une ressource d'API et de définir les différentes opérations possibles sur la collection et les items. #[Groups] permet de configurer les champs qui seront inclus dans le JSON en fonction de l'opération.

## Interface Swagger

API Platform génère automatiquement une documentation Swagger sous la forme d'une interface web interactive.

The screenshot shows the API Platform Swagger interface at `localhost:8000/api`. The top navigation bar includes a logo, the text "API PLATFORM", and a search bar. Below the header, there's a status bar showing "0.0.0 OAS3". The main content area is titled "Departement" and lists two GET methods:

- `GET /api/departements` Retrieves the collection of Departement resources.
- `GET /api/departements/{id}` Retrieves a Departement resource.

Below "Departement" are sections for "Medecin", "Pays", and "SpecialiteComplementaire", each with their respective GET methods listed. A "Schemas" section is also present at the bottom. On the right side, there's a decorative illustration of a character holding a sword.

The screenshot shows a detailed view of the "Medecin" resource at `localhost:8000/api`. The top navigation bar and status bar are identical to the previous screenshot. The main content area starts with a "Parameters" section indicating "No parameters" and a "Responses" section for the 200 status code:

Code	Description	Links
200	Medecin collection	No links

Under "Responses", the "Media type" is set to "application/json". An "Example Value" is provided as a JSON array:

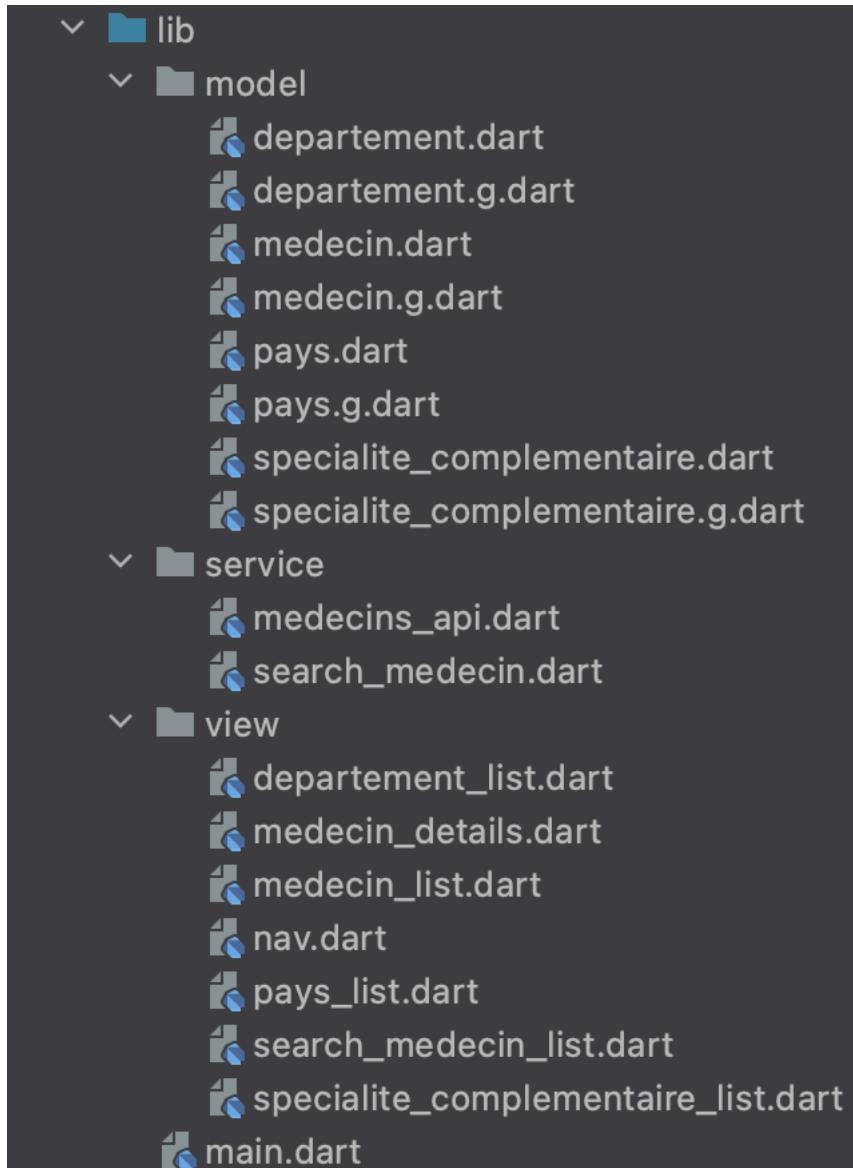
```
[{"id": 0, "nom": "string", "prenom": "string", "adresse": "string", "tel": "string"}]
```

Below this, another "GET /api/medecins/{id}" method is listed. At the bottom, there's a "Pays" section.

## Application mobile

L'application mobile est réalisée en Dart avec le framework Flutter.

### Structure de l'application



### Description des classes

#### Les modèles

Ce sont les objets métiers, ils contiennent également les méthodes pour convertir depuis et vers le JSON. Ces méthodes sont générées automatiquement dans les fichiers .g.dart.

```

part 'medecin.g.dart';

@JsonSerializable()
class Medecin {
    int id;
    String? nom;
    String? prenom;
    String? adresse;
    String? tel;
    SpecialiteComplementaire? specialiteComplementaire;
    Departement? departement;

    Medecin({required this.id, this.nom, this.prenom, this.adresse, this.tel, this.specialiteComplementaire, this.departement});

    factory Medecin.fromJson(Map<String, dynamic> json) =>
        _$MedecinFromJson(json);

    Map<String, dynamic> toJson() => _$MedecinToJson(this);
}

```

## Le fichier .g.dart

```

// GENERATED CODE - DO NOT MODIFY BY HAND

part of 'medecin.dart';

// *****
// JsonSerializableGenerator
// *****

Medecin _$MedecinFromJson(Map<String, dynamic> json) => Medecin(
    id: json['id'] as int,
    nom: json['nom'] as String?,
    prenom: json['prenom'] as String?,
    adresse: json['adresse'] as String?,
    tel: json['tel'] as String?,
    specialiteComplementaire: json['specialiteComplementaire'] == null
        ? null
        : SpecialiteComplementaire.fromJson(
            json['specialiteComplementaire'] as Map<String, dynamic>, // SpecialiteComplementaire.fromJson
        ),
    departement: json['departement'] == null
        ? null
        : Departement.fromJson(json['departement'] as Map<String, dynamic>),
); // Medecin

Map<String, dynamic> _$MedecinToJson(Medecin instance) => <String, dynamic>{
    'id': instance.id,
    'nom': instance.nom,
    'prenom': instance.prenom,
    'adresse': instance.adresse,
    'tel': instance.tel,
    'specialiteComplementaire': instance.specialiteComplementaire,
    'departement': instance.departement,
};

```

## Les services

Il contient les classes qui permettent d'accéder aux données depuis l'API.

```
class MedecinsApi {
    late Response response;
    Dio dio = Dio();
    String baseURL = "http://192.168.1.16:8000/api";

    Future<List<Medecin>> getMedecins({String? query}) async {
        response = await dio.get(baseURL+"/medecins",
            options: Options(headers: {Headers.acceptHeader: "application/json"}));
        var jsonlist = response.data as List;
        var medecinList = jsonList.map((jsonElement) {
            return Medecin.fromJson(jsonElement);
        }).toList();

        if (query != null) {
            medecinList = medecinList.where((element) => element.nom!.toLowerCase().contains((query.toLowerCase()))).toList();
        }

        return medecinList;
    }
}
```

## Les vues

Ces fichiers représentent les écrans de l'application. Chaque fichier est composé de deux classes, une s'occupant des états et l'autre qui contient le widget.

Chaque classe widget doit être composée d'une méthode build() qui contient les différents widgets qui composent ce widget.

```
class MedecinList extends StatefulWidget {
  const MedecinList({Key? key}) : super(key: key);

  static const routeName = '/medecinList';

  @override
  State<MedecinList> createState() => _MedecinListState();
}

class _MedecinListState extends State<MedecinList> {
  @override
  void initState() {
    super.initState();
  }

  @override
  Widget build(BuildContext context) {
    final args =
      ModalRoute.of(context)!.settings.arguments as Future<List<Medecin>>;

    return Scaffold(
      appBar: AppBar(
        title: const Text("Médecins"),
      ), // AppBar
      body: Center(
        child: FutureBuilder<List<Medecin>>(
          future: args,
          builder: (context, snapshot) {
            if (snapshot.hasData) {
              return ListView.builder(
                itemBuilder: (context, i) {
                  return ListTile(
                    title: Row(
                      children: [
                        Text(snapshot.data![i].prenom!),
                        const Text(" "),
                        Text(snapshot.data![i].nom!)
                      ],
                    ), // Row
                    subtitle: Text(snapshot.data![i].adresse!),
                    leading: const Icon(Icons.account_circle),
                    onTap: () {
                      Navigator.pushNamed(context, MedecinDetails.routeName,
                        arguments: snapshot.data![i].id);
                    },
                  ); // ListTile
                },
                itemCount: snapshot.data!.length,
              ); // ListView.builder
            } else {
              return const CircularProgressIndicator();
            }
          },
        ), // FutureBuilder
      ), // Center, Scaffold
    );
  }
}
```