NestJS

NestJS est un framework de développement backend pour NodeJS permettant de créer des applications complètes, évolutives et performantes.

Il repose sur une architecture stricte, standardisée et opiniâtre, facilitant la conception de code de qualité et maintenable qui répond efficacement à tous les besoins d'une application backend moderne.

> **NOTE**
>
> La documentation officielle reste la référence principale. Cette page a pour objectif de présenter uniquement les concepts fondamentaux du framework.
>
> https://docs.nestjs.com/

Spécificités

Nest est fondé sur le principe de l'inversion de contrôle (IoC), un concept fondamental en programmation orientée objet.

Ce principe stipule que le flux d'exécution d'un logiciel n'est plus directement contrôlé par l'application elle-même, mais par le framework ou la couche logicielle sous-jacente.

Contrairement à la programmation procédurale traditionnelle où le développeur écrit du code qui utilise des bibliothèques tierces, dans l'IoC, c'est le framework qui utilise le code produit par le développeur.

https://en.wikipedia.org/wiki/Inversion_of_control

Découpage

NestJS impose une structure claire à l'application. Chaque composant a un rôle spécifique et se trouve dans un répertoire dédié avec une nomenclature

précise.

Pour chacun des composants que nous aborderons par la suite, NestJS applique automatiquement un comportement spécifique, qui peut être personnalisé à l'aide de décorateurs.

Décorateurs

Les décorateurs, bien que relativement peu utilisés en JavaScript standard, constituent un élément central dans NestJS. Ils représentent le principal moyen d'exploiter les fonctionnalités avancées du framework.

Exemples de décorateurs

```tsx
// Déclaration d'un module
@Module({
  controllers: [CatsController],
  providers: [CatsService],
})

// Injection de dépendances
constructor(
  @InjectRepository(CatEntity)
  private readonly catRepository: Repository<CatEntity>
) {}

// Description d'une route
@ApiResponse({ status: 200, description: 'Returns a cat' })
@Get("/:id")
findOne(@Param('id') id: string): Promise<CatResponseDto> {
  return this.catService.findOne(id, true);
}

// Validation d'un objet
export class CatDto {
  @IsString()
```

```
  @IsNotEmpty()
  name: string;

  @IsUUID()
  breedId: string;
}
```

CLI

https://docs.nestjs.com/cli/overview

NestJS est fourni avec un outil en ligne de commande puissant qui facilite le développement d'applications, notamment grâce à la génération automatique de composants.

```bash
# Installer la CLI
npm i -g @nestjs/cli

# Créer un nouveau projet
nest new my-project

# Générer une nouvelle ressource complète
# (génère module, controller, service, routes, DTOs)
nest g resource cats

# Générer des composants individuels
nest g module cats
nest g controller cats
nest g service cats

# Afficher les commandes disponibles
nest g -h
```

Ressources

- Documentation officielle
- Projet d'exemple

- Awesome Nest / Exemples de boilerplates

Découpage de l'application

NestJS impose une structure modulaire précise pour organiser chaque composant de l'application.

Cette architecture stricte favorise une organisation claire du code, améliore la collaboration en équipe, garantit une meilleure qualité de développement et assure une forte évolutivité du projet.

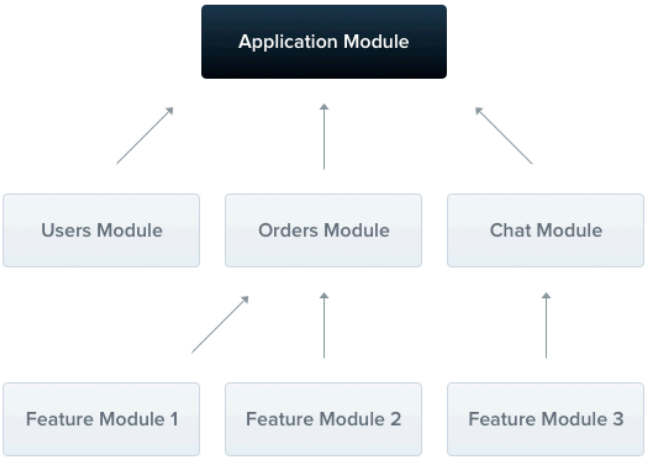
```
src/
  |__user/
  |      |__user.entity.ts
  |      |__user.service.ts
  |      |__user.controller.ts
  |      |__user.module.ts
  |__app.module.ts
  |__main.ts
```

Modules

https://docs.nestjs.com/modules

La logique métier de l'application est divisée en modules distincts, chacun représentant une fonctionnalité spécifique comme l'authentification, la gestion des utilisateurs ou la gestion des produits.

Ces modules encapsulent tous les composants nécessaires à leur fonctionnement, et l'application principale est elle-même un module racine qui orchestre l'ensemble.



Déclaration des modules

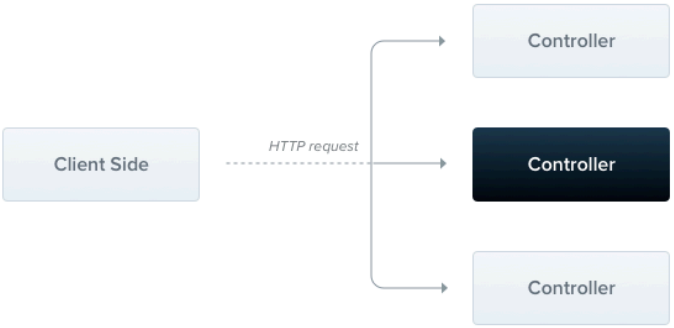
```ts
// cat.module.ts
@Module({
  controllers: [CatController],
  providers: [CatService],
  imports: [
    TypeOrmModule.forFeature([CatEntity, BreedEntity]),
    BreedModule,
  ],
  exports: [CatService],
})
```

Contrôleurs

https://docs.nestjs.com/controllers

Les contrôleurs constituent le point d'entrée de l'application depuis l'extérieur. Ils définissent les routes (URLs), spécifient les paramètres attendus en entrée et structurent les types de réponses.

Leur rôle se limite à recevoir les données entrantes, les transmettre aux services appropriés et renvoyer une réponse formatée. Ils ne doivent pas contenir de logique métier complexe.




```ts
// cat.controller.ts
@Controller("cat") // route '/cat'
export class CatController {
  constructor(private catService: CatService) {}

  @Get("/find-all") // route '/cat/find-all'
  findAll(): Promise<CatResponseDto[]> {
    return this.catService.findAll();
  }
}
```

Providers

https://docs.nestjs.com/providers

Dans NestJS, tous les composants sont des classes qui dépendent souvent d'autres classes. Le framework intègre un système d'injection de dépendances sophistiqué, permettant d'accéder facilement à n'importe quelle classe depuis une autre, sans se préoccuper de leur instanciation.

Ces classes injectables, appelées Providers, sont annotées avec le décorateur `@Injectable()`.

Pour utiliser un Provider, il suffit de le déclarer dans le module approprié et de l'injecter dans le constructeur de la classe qui en a besoin.

Services

Les services sont des Providers spécialisés qui implémentent la logique métier de l'application : interactions avec la base de données, calculs, traitements complexes, etc. Ils sont principalement utilisés par les contrôleurs, qui délèguent ainsi le traitement des données.

```ts
@Injectable()
export class CatService {
  constructor(
    @InjectRepository(CatEntity)
    private readonly catRepository: Repository<CatEntity>
  ) {}

  async findAll(): Promise<CatEntity[]> {
    return this.catRepository.find();
  }
}
```

Middleware

https://docs.nestjs.com/middleware

Les middlewares manipulent les objets de requête et de réponse, par exemple en extrayant et décodant un token JWT pour identifier l'utilisateur, ou en interrompant une requête non autorisée.

Les middlewares peuvent être chaînés et transférer le contrôle au middleware suivant via l'appel à `next()`, avant ou après avoir effectué leur traitement.

Ils sont appliqués globalement à un ensemble de routes dès l'initialisation de l'application, sans distinction de comportement spécifique à chaque route.

Guards

https://docs.nestjs.com/guards

Les guards déterminent si une requête doit être traitée ou rejetée. Ils interviennent après les middlewares et avant les intercepteurs.

Contrairement aux middlewares, les guards peuvent être appliqués de manière ciblée à un module, un contrôleur ou une route spécifique. Ils sont particulièrement utiles pour implémenter des contrôles d'accès basés sur les rôles utilisateurs.

Intercepteurs

https://docs.nestjs.com/interceptors

Les intercepteurs permettent de modifier les données reçues ou renvoyées, ainsi que d'interrompre la chaîne de traitement si nécessaire.

En entrée, ils peuvent implémenter des mécanismes de cache (évitant ainsi l'appel au contrôleur), mesurer le temps d'exécution d'une requête, ou effectuer d'autres traitements préalables.

En sortie, ils peuvent transformer le format d'une exception, filtrer les données sensibles avant leur envoi, ou adapter la réponse selon d'autres critères.

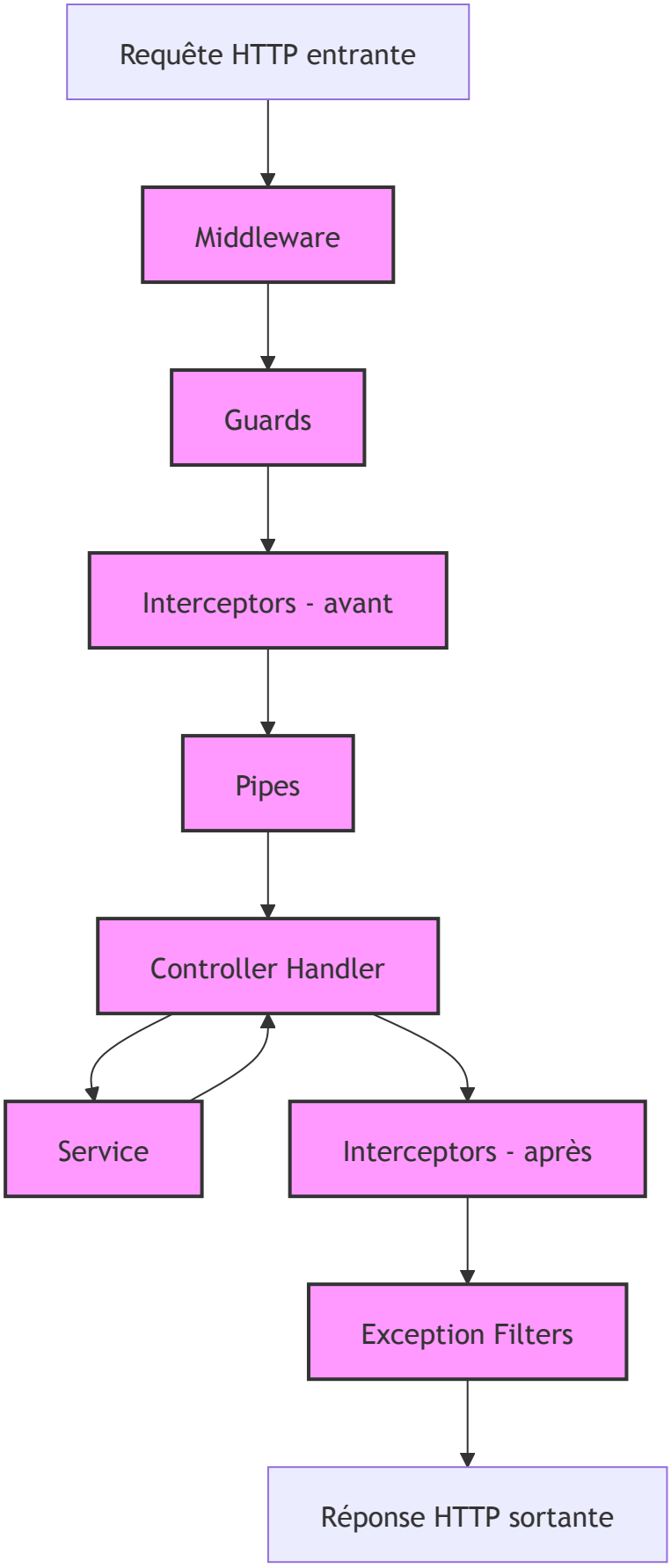
Pipes

https://docs.nestjs.com/pipes

Les pipes transforment les données reçues par les contrôleurs.

Ils sont principalement utilisés pour valider et convertir (désérialiser) les données entrantes dans le format attendu par l'application.

Pipeline d'exécution


Requête HTTP entrante
Middleware
Guards
Interceptors - avant
Pipes
Controller Handler
Service
Interceptors - après
Exception Filters
Réponse HTTP sortante

API REST

L'une des fonctionnalités principales de NestJS est la création d'API REST robustes, complètes et évolutives.

Contrôleurs

https://docs.nestjs.com/controllers

Les **Contrôleurs** sont les composants qui définissent les routes et traitent les requêtes entrantes.

```ts
@Controller("cat") // définit la route de base '/cat'
export class CatController {
  constructor(private catService: CatService) {}

  @Get("/") // GET '/cat'
  findAll(): Promise<CatResponseDto[]> {
    return this.catService.findAll({ includeBreed: true });
  }

  @Get(":id") // GET '/cat/:id'
  findOne(@Param("id") id: string): Promise<CatResponseDto> {
    return this.catService.findOne(id, true);
  }

  @Post() // POST '/cat'
  create(@Body() cat: CreateCatDto): Promise<CatResponseDto> {
    return this.catService.create(cat);
  }

  @Put(":id") // PUT '/cat/:id'
  async update(
    @Param("id") id: string,
    @Body() cat: UpdateCatDto
  ): Promise<CatResponseDto> {
```

```
    return this.catService.update(id, cat);
  }
}
```

DTOs (Data Transfer Objects)

Les DTOs définissent la structure des données échangées par l'API, tant pour les requêtes entrantes que pour les réponses.

Ils sont distincts des modèles de base de données et se concentrent uniquement sur les données nécessaires aux échanges avec le client.

class-transformer: Utilisé pour la sérialisation et désérialisation des données, c'est-à-dire la transformation entre objets JSON et instances de classes.

class-validator: Permet de définir et valider les types et contraintes des propriétés d'une classe.

ts

```ts
// cat.dto.ts
import { Expose, Exclude } from "class-transformer";
import { IsNotEmpty, IsNumber, IsOptional, IsString } from "class-va

class CreateCatDto {
  @IsString()
  @IsNotEmpty()
  name: string;

  @IsNumber()
  @IsOptional()
  age?: number;
}

@Exclude()
class CatResponseDto {
  @Expose()
  id: string;
```

```ts
  @Expose()
  name: string;
}
```

Validation des données entrantes

https://docs.nestjs.com/techniques/validation

Une fois les DTOs définis et utilisés dans les contrôleurs, nous pouvons valider automatiquement les données entrantes grâce à un pipe de validation.

```ts
app.useGlobalPipes(
  new ValidationPipe({
    whitelist: true, // supprime les propriétés non définies dans le
    transform: true, // convertit les valeurs entrantes aux types ap
  })
);
```

Sérialisation des données sortantes

https://docs.nestjs.com/techniques/serialization

Pour contrôler le format des données renvoyées par l'API, nous pouvons utiliser un intercepteur de sérialisation.

```ts
app.useGlobalInterceptors(new ClassSerializerInterceptor(app.get(Ref
```

> **Attention**: Si un service renvoie un `Entity` alors que le controlleur renvoie un DTO, même si les deux classes sont compatibles, il faudra explicitement indiquer quelle classe doit être utilisée pour la sérialisation avec `SerializeOptions`.

```ts
@SerializeOptions({type: CatResponseDto})
@Get()
findAll(): Promise<CatResponseDto[]> {
```

```
    return this.catService.findAll();
  }
```

Exceptions

https://docs.nestjs.com/exception-filters

À tout moment dans l'application, il est possible d'interrompre le traitement d'une requête et de renvoyer une erreur en lançant une exception.

```tsx
throw new HttpException("Forbidden", HttpStatus.FORBIDDEN);
throw new ForbiddenException();
```

OpenAPI / Swagger

https://docs.nestjs.com/openapi/introduction

Toute API professionnelle doit fournir une documentation complète, et la norme actuelle est OpenAPI.

NestJS intègre nativement la génération de documentation OpenAPI via la bibliothèque Swagger. Il suffit d'annoter les contrôleurs et les DTOs avec des décorateurs spécifiques pour décrire les routes et les structures de données.

```ts
// cat.controller.ts
class CatController {
  @ApiOperation({ summary: "Find all cats" })
  @ApiResponse({ status: 200, description: "Returns a list of cats"
  @Get()
  findAll(@Query() query: FindAllQueryDto): Promise<CatResponseDto[]
    return this.catService.findAll(query);
```

```
  }
}
```

```ts
// cat.dto.ts
class CatResponseDto {
  @ApiProperty({ description: "The name of the cat" })
  @IsString()
  @IsNotEmpty()
  name: string;

  @ApiProperty({ description: "The age of the cat" })
  @IsNumber()
  @IsOptional()
  age: number;
}
```

```ts
// main.ts
const config = new DocumentBuilder()
  .setTitle("Cats example")
  .setDescription("The cats API description")
  .setVersion("1.0")
  .build();
const documentFactory = () => SwaggerModule.createDocument(app, conf
SwaggerModule.setup("swagger", app, documentFactory);
```

La documentation sera accessible à l'URL `/swagger`, et la specification OpenAPI sera disponible sous forme de JSONà l'URL `/swagger-json`.

GraphQL dans NestJS

NestJS propose nativement la création d'API GraphQL tout en beneficiant de la puissance du framework pour construire des resolvers avancés.

https://docs.nestjs.com/graphql/quick-start

Code/Schema first

Lors de l'utilisation de GraphQL au sein de NestJS, vous devrez choisir parmi deux approches:

Schema first

Dans cette approche, vous créez d'abord votre schéma GraphQL, et NestJS génerera les types Typescript que vous devrez suivre dans vos resolvers.

Code first

Dans cette approche, vous créez votre schema en tapant uniquement du code Typescript. Les types seront définis par des classes annotées et les resolvers serviront pour la définition des requêtes.

NestJS génèrera alors automatiquement le schema GraphQL.

> Nous étudierons l'approche code-first dans la suite du cours

---

Définition du schema

Les types sont définis en tant que classes, et les décorateurs servent pour indiquer les modificateurs. On pourra créer des classes à part uniquement pour GraphQL (èntity.model.ts`) ou bien réutiliser les DTO existants pour le REST.

Modèles

Les types de données renvoyées par l'API sont définis avec `@ObjectType()`.

Chaque champ doit etre annoté avec `@Field()`.

Le type `String` est par défaut, il n'est donc pas necessaire de le preciser dans `Field`.

```typescript
import { Field, ID, Int, ObjectType } from "@nestjs/graphql";

@ObjectType()
class User {
  @Field(() => ID)
  id: string;

  @Field(() => Int)
  age: number;

  @Field({ nullable: true })
  name?: string;
}
```

Inputs

Les types envoyés à l'API `input` doivent être annotés avec `@InputType()`.

```typescript
@InputType()
class UserInput {
  @Field()
  name: string;
}
```

Resolvers

Les resolvers sont l'équivalent des controllers pour GraphQL. Ils sont définis avec `@Resolver()`.

- Les queries sont définies avec `@Query()`.
- Les mutations sont définies avec `@Mutation()`.
- Les resolvers chainés sont définies avec `@ResolveField()`.

On peut récupérér les arguments, le parent et le contexte de la requete avec `@Args()`, `@Parent()` et `@Context()`.

typescript

```typescript
@Resolver(() => Cat)
export class CatResolver {
  constructor(
    private catService: CatService,
    private breedService: BreedService
  ) {}

  @Query(() => [Cat])
  async cats(): Promise<Cat[]> {
    return this.catService.findAll({ includeBreed: true });
  }

  @Query(() => Cat)
  async cat(@Args("id") id: string): Promise<Cat> {
    const cat = await this.catService.findOne(id, true);
    return new CatResponseDto(cat);
  }

  @ResolveField(() => Breed)
  breed(@Parent() cat: Cat): Promise<Breed> {
    return this.breedService.findOne(cat.breedId);
  }

  @Mutation(() => Cat)
  async createCat(@Args("cat") cat: CreateCatInput): Promise<Cat> {
    return this.catService.create(cat);
```

```
  }
}
```

En pratique

Vous pouvez retrouver un exemple de projet NestJS avec GraphQL sur le projet demo.

TypeORM

Pour gérer les interactions avec les bases de données, NestJS propose le module `@nestjs/typeorm`, qui fournit une intégration élégante avec TypeORM. Cette combinaison permet de manipuler efficacement les entités et les repositories, formant ainsi une abstraction puissante au-dessus de la base de données.

Ce module simplifie considérablement la connexion, la lecture et l'écriture dans différents systèmes de gestion de bases de données relationnelles depuis les services de l'application.

- Documentation NestJS : https://docs.nestjs.com/techniques/database
- Documentation TypeORM : https://typeorm.io/

Entités

Les entités sont des classes qui représentent les tables de la base de données. Décorées avec `@Entity()`, elles définissent la structure des données persistantes. Chaque propriété d'une entité correspond à une colonne dans la table associée, et les décorateurs permettent de configurer ces colonnes ainsi que les relations entre les différentes entités.

```ts
@Entity()
export class Cat {
  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  name: string;

  @Column()
  age: number;
```

```ts
  @ManyToOne(() => Breed, (breed) => breed.cats)
  breed: Breed;
}

@Entity()
export class Breed {
  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  name: string;

  @OneToMany(() => Cat, (cat) => cat.breed)
  cats: Cat[];
}
```

Utilisation

```ts
// Configuration globale dans app.module.ts
@Module({
  imports: [
    TypeOrmModule.forRoot(databaseConfig),
    // ...
  ],
})
// Import des entités dans le module spécifique
// cat.module.ts
@Module({
  imports: [TypeOrmModule.forFeature([Cat])],
})
// Injection et utilisation du repository dans un service
// cat.service.ts
@Injectable()
export class CatService {
  constructor(
    @InjectRepository(Cat)
    private catRepository: Repository<Cat>
```

```
  ) {}

  async findAllJohns(): Promise<Cat[]> {
    return this.catRepository.find({
      where: {
        name: "John",
      },
      relations: ["breed"], // Charge également les données de race
    });
  }
}
```

Migrations

Les migrations sont un moyen de gérer les modifications de la structure de la base de données au fur et a mesure que le projet évolue.

En mode developpement, on utilise parfois la methode dite de "synchronisation", qui comparera la base de donnée courante avec le code TypeORM et appliquera les modifications nécessaires automatiquement. **Attention: Il ne faut surtout pas utiliser cette methode en production !** En effet, cela peut avoir pour effet de supprimer des données existantes ou generer des incohérences.

Pour générer des migrations, il faut dabord desactiver la synchronisation, puis on peut utiliser la commande suivante:

```bash
npx typeorm migration:generate src/migrations/[migration-name]
```

Cela comparera la base de données avec le code TypeORM et générera une migration.

Pour appliquer les migrations, on peut utiliser la commande suivante:

```bash
npx typeorm migration:run
```

- Documentation TypeORM : https://typeorm.io/migrations

Tests

- https://docs.nestjs.com/testing
- https://jestjs.io/

Tester de manière exhaustive les applications backend est crucial, car elles sont responsables de la gestion et de la sécurisation des données, contrairement aux applications frontend qui se concentrent principalement sur l'interface utilisateur.

Si les tests manuels sont souvent privilégiés pour les applications frontend (où l'expérience visuelle est prépondérante), cette approche devient rapidement inefficace pour les backends. Les tests programmatiques sont donc indispensables pour garantir la fiabilité des applications backend, et sont généralement plus simples à mettre en œuvre que pour le frontend. L'intelligence artificielle peut d'ailleurs considérablement accélérer l'écriture de ces tests.

NestJS intègre un système de tests robuste et intuitif qui facilite grandement cette tâche essentielle.

Tests unitaires

Les tests unitaires se concentrent sur des portions isolées du code.

Ils vérifient le bon fonctionnement de composants spécifiques de l'application, en prenant en compte différents scénarios d'utilisation, notamment les diverses combinaisons d'arguments que les fonctions peuvent recevoir et les différents contextes d'exécution possibles.

```ts
// cat.service.spec.ts
describe("CatService", () => {
  let service: CatService;
```

```ts
  beforeEach(async () => {
    const module: TestingModule = await Test.createTestingModule({
      providers: [CatService],
    }).compile();

    service = module.get<CatService>(CatService);
  });

  it("should be defined", () => {
    expect(service).toBeDefined();
  });

  it("should create and find all cats", async () => {
    await service.create({ name: "John", age: 10 });
    const result = await service.findAll();

    expect(result).toEqual([{ name: "John", age: 10 }]);
  });
});
```

Mocks

Lorsque nous testons des composants qui dépendent d'autres parties du code, il est souvent nécessaire de créer des mocks. Cette technique permet d'isoler précisément le code testé en remplaçant ses dépendances par des implémentations contrôlées qui simulent le comportement attendu dans le contexte du test.

ts

```ts
// cat.controller.ts
@Controller("cat") // route '/cat'
export class CatController {
  constructor(private catService: CatService) {}

  @Get("/find-all") // route '/cat/find-all'
  findAll(): Promise<CatResponseDto[]> {
    return this.catService.findAll({ breed: true });
  }
}
```

```
// cat.controller.spec.ts
it("calls catService with correct parameters", async () => {
  jest.spyOn(catService, "findAll").mockResolvedValue([]);

  const result = await controller.findAll();

  expect(result).toEqual([]);
  expect(catService.findAll).toHaveBeenCalledWith({ breed: true });
});
```

Coverage

Les outils de tests proposent une analyse de la couverture du code, c'est à dire le pourcentage de code qui est testé. Cela permet de rapidement découvrir les zones de code qui ne sont pas testés pour assurer que tous les scenarios possibles sont pris en compte.

Pour lancer le test avec la couverture, il faut lancer la commande suivante:

bash
```
npm run test:cov
```

Un dossier `coverage` sera créé contenant les rapports de couverture, avec des fichiers descriptifs pour les outils de CI et des pages html pour le developpeur.

```
  async findOne(id: string, includeBreed?: boolean): Promise<CatEntity> {
1x  const cat = await this.catRepository.findOne({
      where: { id },
      relations: includeBreed ? ['breed'] : undefined,
    });
1x  if (!cat) {
      throw new NotFoundException('Cat not found');
    }
1x  return cat;
  }
```

Tests d'intégration (End to End)

https://ladjs.github.io/superagent/

Les tests d'intégration évaluent l'application dans son ensemble, en la démarrant dans un environnement proche de la production. Ils simulent des requêtes HTTP réelles et vérifient les réponses obtenues. Ces tests examinent l'application de bout en bout, incluant tous les composants, leurs interactions et les dépendances externes comme les bases de données.

Bien que plus complexes à élaborer et plus lents à exécuter que les tests unitaires, ils sont essentiels pour valider le comportement global de l'application. Ils se concentrent généralement sur les principaux scénarios d'utilisation et les exigences critiques de sécurité plutôt que sur une couverture exhaustive de toutes les possibilités.

```ts
describe("AppController (e2e)", () => {
  let server: Server;

  beforeEach(async () => {
    const moduleFixture: TestingModule = await Test.createTestingMod
      imports: [AppModule],
    }).compile();

    const app = moduleFixture.createNestApplication();
    await app.init();
    server = app.getHttpServer();
  });

  it("Health check should return OK", () => {
    request(server).get("/").expect(200).expect("OK");
  });

  describe("Breed API", () => {
    it("should create a new breed and retrieve it in the list", asyn
      // Données de test pour la race de chat
      const inputBreed = {
        name: "Persian",
        description: "A fluffy breed",
      };
```

```
      // Test de création d'une race
      const createResponse = await request(server)
        .post("/breed")
        .send(inputBreed)
        .expect(201);

      const createdBreed = createResponse.body;
      expect(createdBreed.name).toBe(inputBreed.name);
      expect(createdBreed.description).toBe(inputBreed.description);
      expect(createdBreed.id).toBeDefined();
      expect(createdBreed.seed).not.toBeDefined(); // Vérifie que le

      // Test de récupération de toutes les races
      const getAllResponse = await request(server).get("/breed").exp

      // Vérifie que la race créée est bien présente dans la liste
      expect(getAllResponse.body).toContainEqual(createdBreed);
    });
  });
});
```

Microservices

https://docs.nestjs.com/microservices/basics

NestJS propose un module intégré pour les microservices, il est disponible dans le paquet `@nestjs/microservices`.

Le principal changement par rapport à une API REST est que, au lieu d'écouter des requêtes HTTP, on écoutera des messages émis, generalement via un message broker. Une application microservice ne peut pas être à la fois un microservice et une API REST. En revanche, l'API REST pourra communiquer avec les microservices.

Il existe differents types de communication compatibles avec NestJS:

- TCP
- Redis
- RabbitMQ
- Kafka ...

Implémentation

Microservice

Le microservice "écoute" les messages émis et y répond (`MessagePattern`), ou ne répond pas (`EventPattern`).

ts

```ts
// main.ts
// Configuration d'un microservice
async function bootstrap() {
  const app = await NestFactory.createMicroservice<MicroserviceOptio
    AppModule,
    {
      transport: Transport.REDIS,
```

```
      options: redisConfig,
    }
  );
  await app.listen();
}
bootstrap();
```

```ts
// *.controller.ts
@Controller()
export class ColorsController {
  // Ecoute de l'évenement "generate_color" avec réponse
  @MessagePattern("generate_color")
  generateColor(breedSeed: string): string {
    return this.colorsService.getColors(breedSeed);
  }

  // Ecoute de l'évenement "cat_created" sans réponse
  @EventPattern("cat_created")
  async handleCatCreated(data: Record<string, unknown>) {
    //...
  }
}
```

Client

Le client (producer) émettra des messages et attendra une réponse ou non.

```ts
// *.module.ts
@Module({
  ...,
  imports: [
    ClientsModule.register([
      {
        name: 'COLORS_SERVICE',
        transport: Transport.REDIS,
        options: redisConfig,
      },
    ]),
```

```
    ...,
  ],
  ...,
})
```

```ts
// *.service.ts
export class CatService {
  constructor(
    @Inject('COLORS_SERVICE') private client: ClientProxy,
  ) {}

  async create(cat: CreateCatDto): Promise<CatEntity> {
    const { seed } = breed;
    // Appel du microservice avec attente de réponse (Observable)
    const colorObservable = this.client.send<string, string>('genera
    const color = await firstValueFrom(colorObservable);
    ...

    // Emission d'un évenement sans attente de réponse
    this.client.emit('cat_created', cat);
  }
}
```

Observables

https://rxjs.dev/guide/observable

Les réponses des microservices sont des Observables.

Un Observable est un objet qui peut émettre des valeurs, des erreurs ou se terminer de manière asynchrone.

```ts
const observable = new Observable((observer) => {
  observer.next("Hello");
  observer.error("Error");
  observer.complete();
});
```

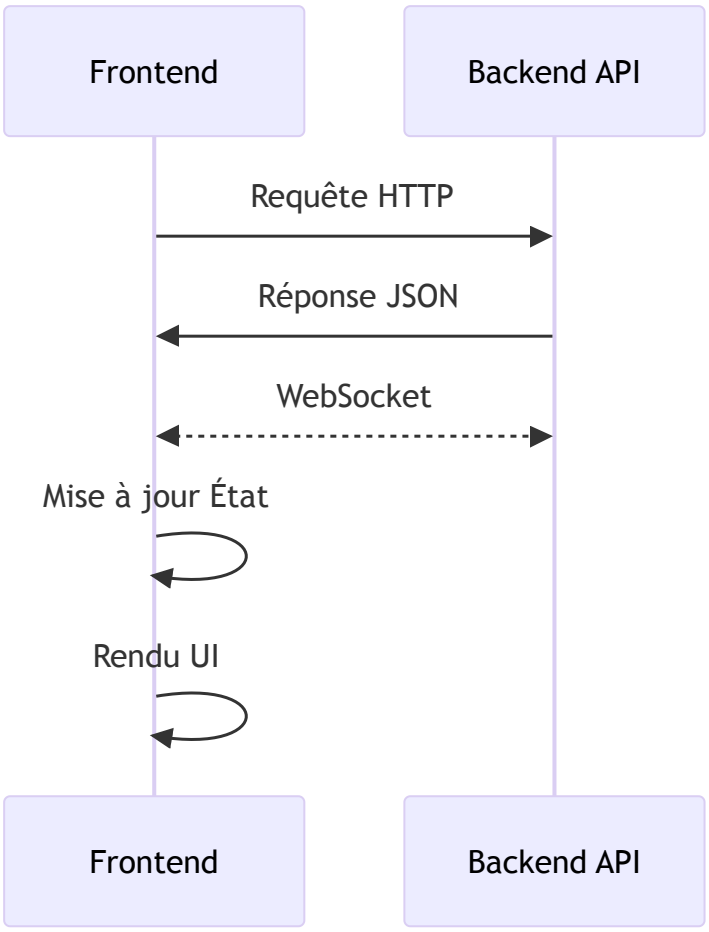
```
observable.subscribe((value) => {
  console.log(value);
});

const first = await firstValueFrom(observable);
```

Websockets

Introduction

Les websockets sont un protocole de communication bidirectionnel et full-duplex entre le client et le serveur.

Contrairement aux API REST, ou le client doit systèmatiquement requêter le serveur pour obtenir des informations à jour, les websockets permettent de maintenir une connexion persistante entre le client et le serveur, permettant ainsi de recevoir des notifications en temps réel émises par le serveur.



Implémentation

https://docs.nestjs.com/websockets/gateways

NestJS propose un module intégré pour les websockets, il est disponible dans le paquet `@nestjs/websockets`.

L'équivalent d'un contrôleur pour les websockets est la `Gateway`.

```ts
@WebSocketGateway()
export class LiveGateway implements OnGatewayConnection {
{
  // Surveiller les connexions des clients
  handleConnection(client: Socket) {
    console.log(`Client id: ${client.id} connected`);
    // Envoyer un message à un client specifique
    client.emit('hello', { message: 'Hello from the server' });
  }

  // Souscrire à un evenement
  @SubscribeMessage('hello')
  handleMessage(
    @ConnectedSocket() client: Socket,
    @MessageBody() data: HelloRequestDto,
  ): HelloResponseDto {
    // Recevoir un message d'un client et y répondre
    return {
      message: `Hello ${data.name}`,
    };
  }
}
```

Projet d'exemple

Un projet de démonstration illustrant les principales fonctionnalités de NestJS est disponible dans le dépôt GitHub nest-demo.

Structure du projet

Cette application de démonstration implémente un système de gestion de chats et de races félines, permettant de comprendre les concepts fondamentaux de NestJS dans un contexte concret.

Technologies et concepts mis en œuvre

- API REST complète avec CRUD
- Persistance des données dans PostgreSQL via TypeORM
- Documentation interactive avec OpenAPI/Swagger
- Validation rigoureuse des données entrantes et sérialisation contrôlée des réponses
- Couverture de code par tests unitaires
- Validation fonctionnelle par tests d'intégration
- Communication en temps réel via WebSockets
- Démonstration des Guards avec un exemple de contrôle d'accès aléatoire
- Middleware de journalisation des requêtes
- Application séparée en mode microservice qui permet de calcul de la couleur des chats

Fonctionnalités métier

- Gestion complète des chats (création, modification, suppression, consultation)
- Gestion complète des races félines (création, modification, suppression, consultation)

- Système de notifications en temps réel lors de l'ajout ou la modification d'entités
- Mécanisme avancé : les races possèdent une valeur seed (non exposée dans l'API) qui détermine la génération des couleurs des chats, assurant que les chats d'une même race présentent des caractéristiques chromatiques similaires

Exercices

Préparation

Ces exercices doivent être réalisés en se basant sur le projet d'exemple fourni.

Pour commencer :

1. Forker le Projet NestJS sur votre compte GitHub
2. Cloner votre fork sur votre machine locale
3. Créer une nouvelle branche pour vos modifications

Une fois votre travail terminé, vous pourrez créer une pull request vers le projet original.

Exigences techniques

- Le code doit être propre, bien structuré et respecter les conventions de NestJS
- L'API doit exposer une spécification OpenAPI complète et précise
- Les données sensibles doivent être sécurisées et masquées aux utilisateurs non autorisés
- Chaque fonctionnalité doit être couverte par des tests unitaires appropriés
- Des tests d'intégration doivent valider les principaux scénarios d'utilisation

Tâches

> Les tâches proposées ci-dessous sont des suggestions. Vous pouvez adapter les spécifications selon votre vision, tant que les fonctionnalités principales sont implémentées correctement et que la qualité du code est maintenue.

1. Gestion des utilisateurs

Étendre le projet pour intégrer une entité utilisateur complète.

Objectifs :

- Créer une entité utilisateur avec des informations de profil pertinentes (nom d'utilisateur, description, etc.)
- Établir une relation entre les chats et les utilisateurs : chaque chat appartient à un seul utilisateur, et un utilisateur peut posséder plusieurs chats
- Développer les routes CRUD pour la gestion des utilisateurs
- Implémenter des fonctionnalités de recherche permettant de filtrer les chats par propriétaire et d'inclure les données utilisateur lors de la récupération des informations sur les chats

---

2. Système d'authentification

Mettre en place un système d'authentification sécurisé pour l'application.

Objectifs :

- Définir et implémenter les attributs nécessaires dans l'entité utilisateur (email, mot de passe hashé, etc.)
- Développer le système d'authentification complet avec inscription, connexion, et gestion des jetons
- Sécuriser les routes existantes en fonction des besoins d'authentification :
  - Restreindre la modification ou suppression des profils aux propriétaires respectifs
  - Limiter l'accès aux fonctionnalités de gestion des chats/races aux utilisateurs authentifiés
  - Associer automatiquement un chat nouvellement créé à l'utilisateur authentifié qui effectue la création

3. Système de commentaires

Permettre aux utilisateurs d'interagir via des commentaires sur les profils de chats.

Objectifs :

- Concevoir l'entité commentaire avec les relations appropriées
- Créer les routes nécessaires pour gérer les commentaires (création, lecture, modification, suppression)
- Établir les relations entre commentaires, utilisateurs et chats
- Implémenter les règles de sécurité pour que les utilisateurs ne puissent modifier ou supprimer que leurs propres commentaires

> Penser à supprimer les commentaires de l'utilisateur lors de sa suppression via une transaction.

---

4. Croisements entre chats

Développer un système permettant le croisement de chats pour créer de nouveaux chatons.

Objectifs :

- Créer une route dédiée au croisement de deux chats existants
- Vérifier que les deux chats appartiennent à l'utilisateur connecté
- Gérer l'hérédité des races :
  - Si les parents sont de même race, le chaton hérite de cette race
  - Si les parents sont de races différentes, créer une nouvelle race avec une seed dérivée des races parentales
  - Alternativement, permettre la création de plusieurs chatons avec attribution aléatoire des races parentales

5. Croisement inter-propriétaires

Enrichir le système de croisement pour permettre des croisements entre chats de différents propriétaires.

Objectifs :

- Concevoir un système de demande et d'approbation pour les croisements (requête, acceptation, refus)
- Développer les entités et routes nécessaires à la gestion des demandes de croisement
- Implémenter la logique d'exécution des croisements approuvés et la gestion de la propriété des chatons résultants

---

6. Modélisation des données

Élaborer une représentation visuelle de l'architecture de données de l'application.

Objectifs :

- Créer un diagramme UML présentant les entités et leurs relations
- Intégrer ce diagramme dans un fichier Markdown au sein du projet
- Utiliser la syntaxe Mermaid pour une visualisation claire et interactive

---

7. Système de rôles et permissions

Implémenter un système de contrôle d'accès basé sur les rôles.

Objectifs :

- Définir différents rôles utilisateur avec leurs permissions respectives
- Adapter le système d'authentification pour intégrer la gestion des rôles
- Créer ou modifier les routes pour restreindre l'accès en fonction des rôles

Exemple de hiérarchie de rôles :

- **Administrateur** : accès complet à toutes les fonctionnalités, gestion des rôles utilisateur
- **Modérateur** : capacité à modérer les commentaires de tous les utilisateurs
- **Utilisateur standard** : accès limité aux fonctionnalités de base

---

8. Deplacement de chats

Creer un systeme où les chats possèdent une position sur un plateau peuvent être déplacés par leur propriétaire.

Ce système devra utiliser uniquement les websockets pour les communications.

- Le propriétaire pourra demander à deplacer son chat
- Tous les déplacements devront être notifiés à tous utilisateurs connectés
- Les utilisateurs pourront demander la position actuelle de tous les chats

---

9. Statistiques

Creer un systeme de statistiques pour l'application sous forme de micro service (à rajouter dans le projet `/app-microservice`).

Ce micro service devra réagir aux modifications effectuées en bases de données (création, modification, suppression, déplacement) des chats, et calculer des statistiques.

Les statistiques doivent être enregistrées en base de données et retournées par l'API REST.

Exemples de statistiques

- Moyenne du nombre de chats par utilisateur
- Moyenne de nombre de chats par race

- Race avec le plus de chats
- Utilisateur avec le plus de chats
- ...

Et plus ...

Si vous cherchez de nouvelles techniques à apprendre, vous pouvez regarder la todolist du projet d'exemple et essayer d'implémenter une ou plusieurs de ces fonctionnalités.