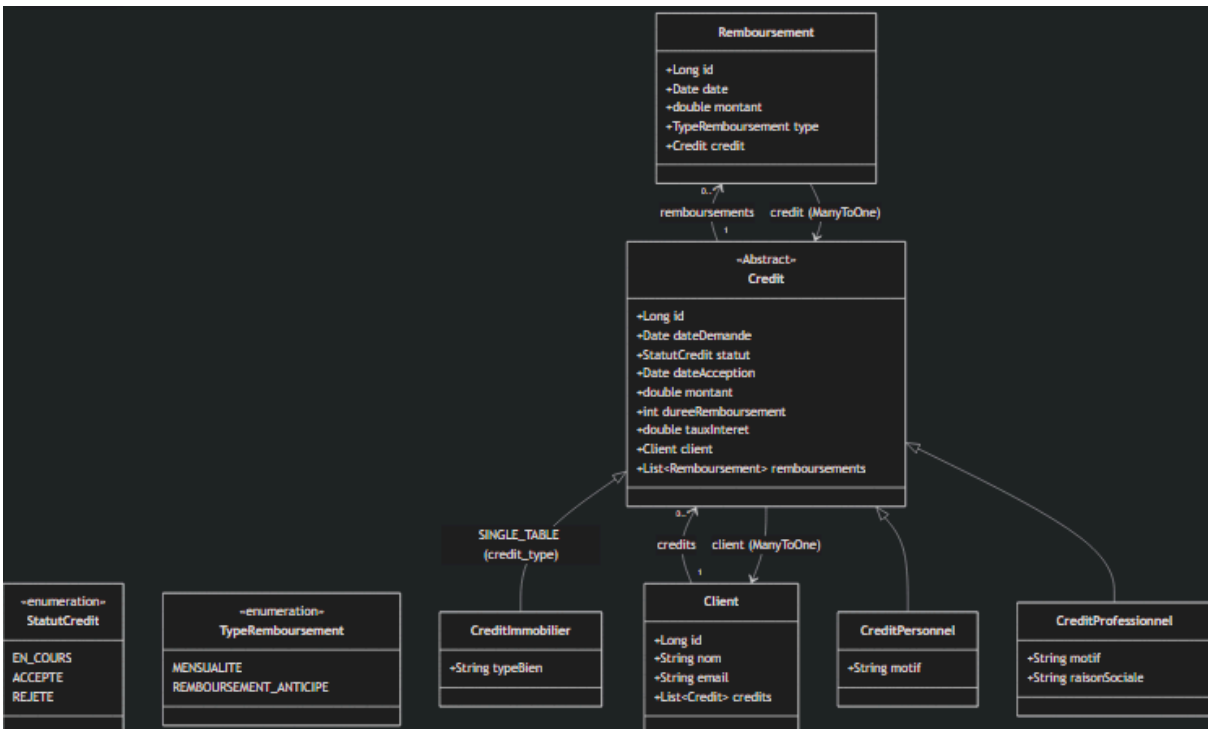
Introduction

L’examen d’Architecture Distribuée et Middleware consiste à concevoir et développer une application web full-stack JEE basée sur les frameworks Spring (backend) et Angular (frontend) pour la gestion de crédits bancaires. L’application permet de gérer des clients, leurs demandes de crédits (personnel, immobilier, professionnel) et leurs remboursements, en respectant des règles métier spécifiques telles que les attributs distinctifs par type de crédit (motif, type de bien financé, raison sociale) et les statuts dynamiques des crédits (en cours, accepté, rejeté). Structurée en couches, elle s’appuie sur Spring Boot pour le backend, avec une persistance des données via **Spring Data JPA et Hibernate** (base H2/MySQL), une couche métier encapsulant la logique via des DTOs et mappers, et une API REST documentée avec Swagger/OpenAPI. La sécurité est assurée par **Spring Security et des JSON Web Tokens (JWT), implémentant une authentification et une autorisation basées sur des rôles (ROLE_CLIENT, ROLE_EMPLOYE, ROLE_ADMIN). Le projet exige la création d’un dépôt GitHub avec des commits réguliers, un rapport détaillé incluant l’architecture technique, un diagramme de classes des entités, ainsi que des tests rigoureux des couches DAO, service et API. Des améliorations optionnelles (validation des données, journalisation, etc.) et une interface Angular intuitive complètent les livrables, mettant en avant les bonnes pratiques de conception distribuée, de modularité et de gestion sécurisée des données.

Conception

diagramme de classe

Le diagramme de classes illustre une modélisation modulaire et hiérarchique conçue pour une application de gestion de crédits bancaires. Il met en évidence une structure centrée autour des entités principales Client, Credit (avec ses sous-classes spécialisées : `CreditImmobilier`, `CreditPersonnel`, `CreditProfessionnel`) et Remboursement, reliées par des relations JPA bidirectionnelles. L’héritage `SINGLE_TABLE` simplifie la persistance des types de crédit via une colonne discriminatrice (`credit_type`), tandis que les associations OneToMany/ManyToOne assurent la traçabilité entre clients, crédits et remboursements. Les enums (`StatutCredit`, `TypeRemboursement`) garantissent la cohérence des données, et les cascades (`CascadeType.ALL`) automatisent la gestion des dépendances. Ce schéma, à la fois **flexible et normalisé**, reflète une architecture adaptée aux règles métier complexes, favorisant l’extensibilité (ajout de types de crédit) et l’intégrité des données.



Architecture

Implémentation :

Entities

Entite Client

```java
@Data  26 usages
@Entity
public class Client {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String nom;
    private String email;

    @OneToMany(mappedBy = "client", cascade = CascadeType.ALL)
    private List<Credit> credits;
}
```

Entite Credit

```java
@Data  3 inheritors
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "credit_type", discriminatorType = DiscriminatorType.STRING)
public  class Credit {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Temporal(TemporalType.DATE)
    private Date dateDemande;

    @Enumerated(EnumType.STRING)
    private StatutCredit statut;

    @Temporal(TemporalType.DATE)
    private Date dateAcception;
    private double montant;
    private int dureeRemboursement;
    private double tauxInteret;

    @ManyToOne
    private Client client;

    @OneToMany(mappedBy = "credit", cascade = CascadeType.ALL)
    private List<Remboursement> remboursements;
}
```

Entite CreditImmoblier

```java
@Data  6 usages
@Entity
@DiscriminatorValue("IMMOBILIER")
public class CreditImmobilier extends Credit {
    private String typeBien; // Appartement, Maison ou Local Commercial
}
```

Entite CreditPersonel

```java
@Data  6 usages
@Entity
@DiscriminatorValue("PERSONNEL")
public class CreditPersonnel extends Credit {
    private String motif;
}
```

Entite CreditProfessionel

```java
@Data  3 usages
@Entity
@DiscriminatorValue("PROFESSIONNEL")
public class CreditProfessionnel extends Credit {
    private String motif;
    private String raisonSociale;
}
```

Entite Rembours

```java
@Data  27 usages
@Entity
public class Remboursement {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Temporal(TemporalType.DATE)
    private Date date;
    private double montant;

    @Enumerated(EnumType.STRING)
    private TypeRemboursement type;

    @ManyToOne
    private Credit credit;
}
```

Repositories

client repository

```java
// Client Repository
@Repository  2 usages
public interface ClientRepository extends JpaRepository<Client, Long> {}
```

Credit Repository

```java
// Credit Repository
@Repository  2 usages
public interface CreditRepository extends JpaRepository<Credit, Long> {}
```

```java
@Repository  2 usages
public interface RemboursementRepository extends JpaRepository<Remboursement, Long> {}
```

Test le code initiale

le code dans main et tros long je vais attacher les screens shots du base de donner

clients:

← T →				id	email	nom
☐	Éditer	Copier	Supprimer	1	john.doe@example.com	John Doe
☐	Éditer	Copier	Supprimer	2	jane.smith@example.com	Jane Smith

credits:

date_demande	duree_remboursement	montant	taux_interet	client_id	id	credit_type	motif	raison_sociale	type_bien	statut
2025-05-19	24	10000	5.2	1	1	PERSONNEL	Achat de voiture	*NULL*	*NULL*	EN_COURS
2025-05-19	360	150000	4.1	2	2	IMMOBILIER	*NULL*	*NULL*	Appartement	ACCEPTE

j’ai pas ajouter raison social

rembourcement

date	montant	credit_id ▼ 1	id	type
2025-05-19	750	2	2	REMBOURSEMENT_ANTICIPE
2025-05-19	450	1	1	MENSUALITE

DTOS

maintenant je vais présenter les dtos pour envoyer les donner au ui ou bien front end pour ne pas entrer dans des boucle infinie à cause des entités

```java
@Data  10 usages
public class ClientDTO {
    private Long id;
    private String nom;
    private String email;
    private List<Long> creditIds; // IDs of associated credits
}
```

```java
@Data  10 usages
public class CreditDTO {
    private Long id;
    private Date dateDemande;
    private String statut; // Enum as String
    private Date dateAcception;
    private double montant;
    private int dureeRemboursement;
    private double tauxInteret;
    private String specificDetails; // To hold special credit details like
    private Long clientId; // ID of the associated client
}
```

```java
@Data  10 usages
public class RemboursementDTO {
    private Long id;
    private Date date;
    private double montant;
    private String type; // Type as String (MENSUALITE, REMBOURSEMENT_ANTICIPE)
    private Long creditId; // The associated credit ID
}
```

Mappers

les mappers a une but pour mapper les donne du dtos vers entities et vise versa, voila des examples

```java
public class Mapper {  15 usages

    // Convert Client to ClientDTO
    public static ClientDTO mapToClientDTO(Client client) {  3 usages
        ClientDTO clientDTO = new ClientDTO();
        clientDTO.setId(client.getId());
        clientDTO.setNom(client.getNom());
        clientDTO.setEmail(client.getEmail());
        clientDTO.setCreditIds(client.getCredits() != null
                ? client.getCredits().stream().map(Credit::getId).collect(Collectors.toList())
                : null);
        return clientDTO;
    }
```

```java
// Convert Remboursement to RemboursementDTO
public static RemboursementDTO mapToRemboursementDTO(Remboursement remboursement) {  3 usages
    RemboursementDTO remboursementDTO = new RemboursementDTO();
    remboursementDTO.setId(remboursement.getId());
    remboursementDTO.setDate(remboursement.getDate());
    remboursementDTO.setMontant(remboursement.getMontant());
    remboursementDTO.setType(remboursement.getType().name());
    remboursementDTO.setCreditId(remboursement.getCredit() != null ? remboursement.getCredit().getId() : null);
    return remboursementDTO;
}
```

Controllers

controllers du clients

```java
@RestController
@RequestMapping("/api/clients")
public class ClientController {

    @Autowired
    private ClientService clientService;

//    @PostMapping
//    public Client addClient(@RequestBody Client client) {
//        return clientService.saveClient(client);
//    }
    @PostMapping
    public ClientDTO saveClient(@RequestBody ClientDTO clientDTO) {
        Client client = Mapper.mapToClient(clientDTO);
        Client savedClient = clientService.saveClient(client);
        return Mapper.mapToClientDTO(savedClient);
    }
```

```java
@GetMapping("/{id}")
public ClientDTO getClient(@PathVariable Long id) {
    Client client = clientService.getClientById(id);
    return client != null ? Mapper.mapToClientDTO(client) : null;
}

@GetMapping
public List<ClientDTO> getAllClients() {
    return clientService.getAllClients().stream() Stream<Client>
            .map(Mapper::mapToClientDTO) Stream<ClientDTO>
            .collect(Collectors.toList());
}
```

```java
@DeleteMapping("/{id}")
public void deleteClient(@PathVariable Long id) {
    clientService.deleteClient(id);

}
```

controller du credit :

```java
@RestController
@RequestMapping("/api/credits")
@CrossOrigin
public class CreditController {

    @Autowired
    private CreditService creditService;

    // Create or update a credit (Can handle different types of credits)
    @PostMapping
    public CreditDTO addCredit(@RequestBody CreditDTO creditDTO) {
        Credit credit = Mapper.mapToCredit(creditDTO, new Credit());
        Credit savedCredit = creditService.saveCredit(credit);
        return Mapper.mapToCreditDTO(savedCredit);
    }
```

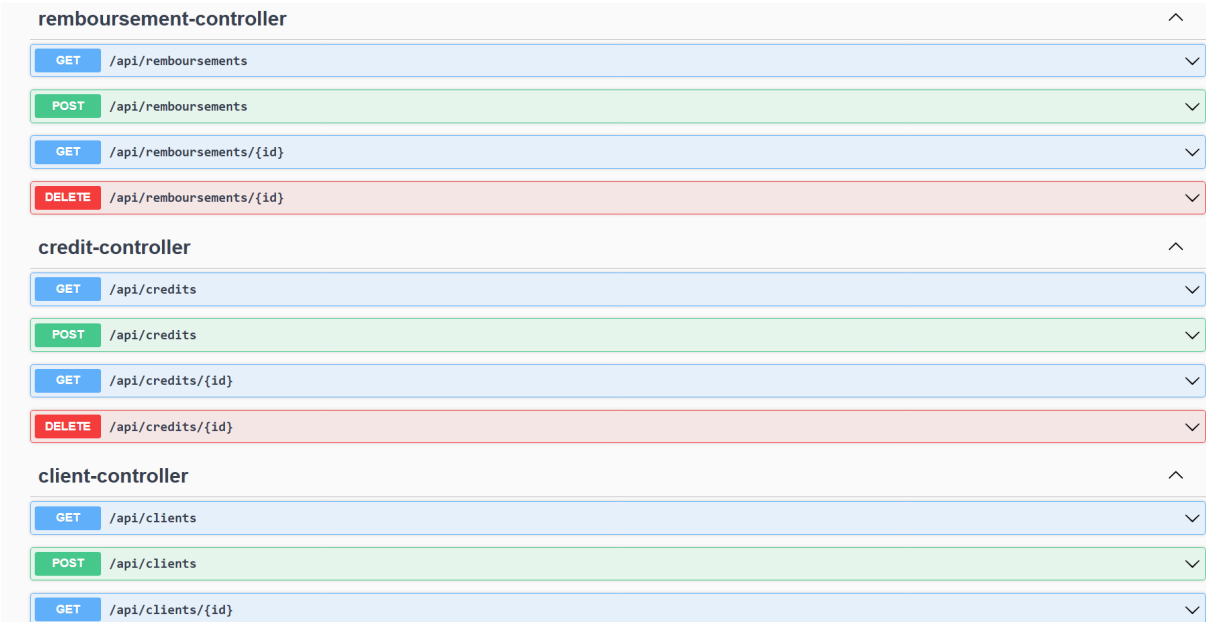
```java
@GetMapping("/{id}")
public CreditDTO getCreditById(@PathVariable Long id) {
    Credit credit = creditService.getCreditById(id);
    return credit != null ? Mapper.mapToCreditDTO(credit) : null;
}

// Get all credits
@GetMapping
public List<CreditDTO> getAllCredits() {
    return creditService.getAllCredits() List<Credit>
            .stream() Stream<Credit>
            .map(Mapper::mapToCreditDTO) Stream<CreditDTO>
            .collect(Collectors.toList());
}
```

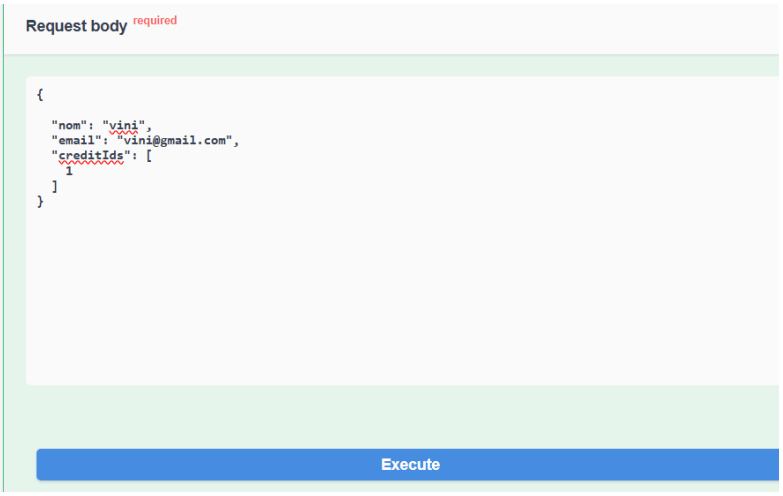
```java
// Delete a credit by ID
@DeleteMapping("/{id}")
public void deleteCredit(@PathVariable Long id) {
    creditService.deleteCredit(id);
}
```

Swagger

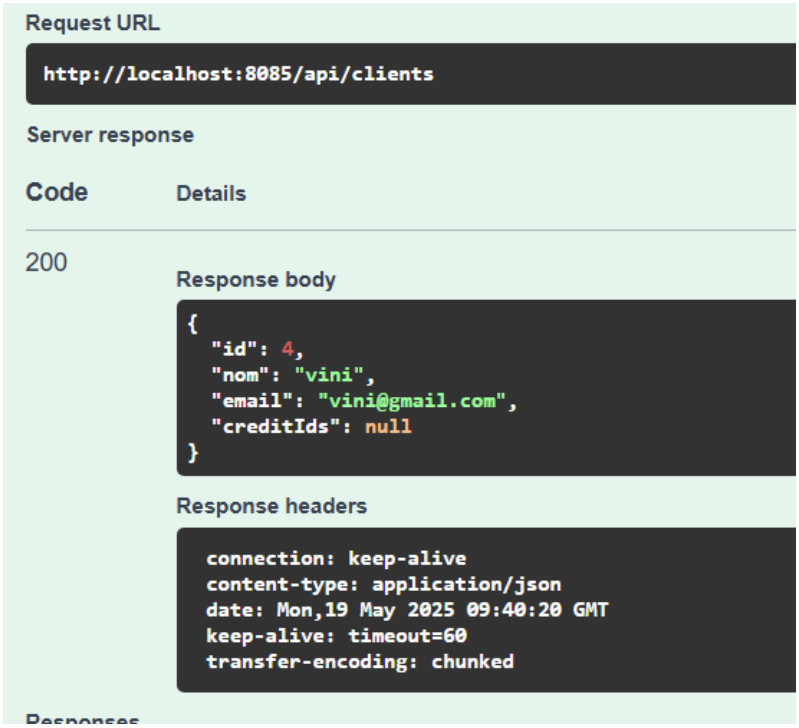
dans cette partie on va interagire aven notre backend utilisons swagger je vais faire les different methodes pour les client, car c'est la meme processuse pour les autres classes



por ajout d’un client :

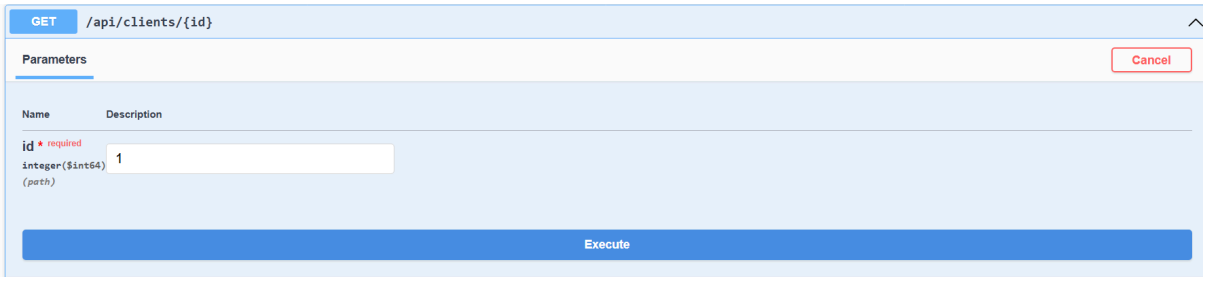


result



get all clients

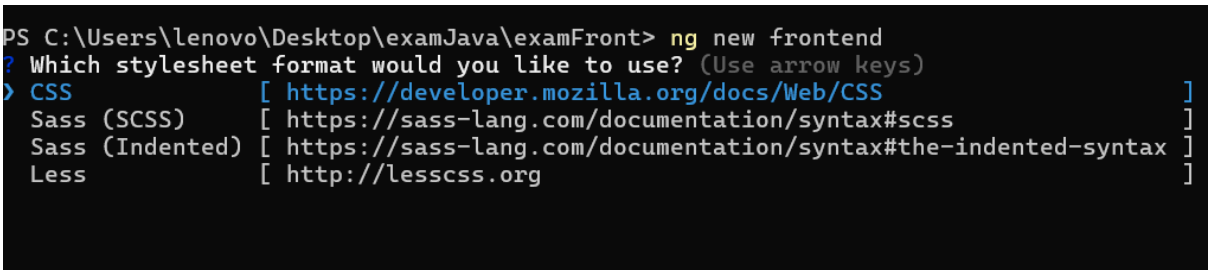
get client by id :



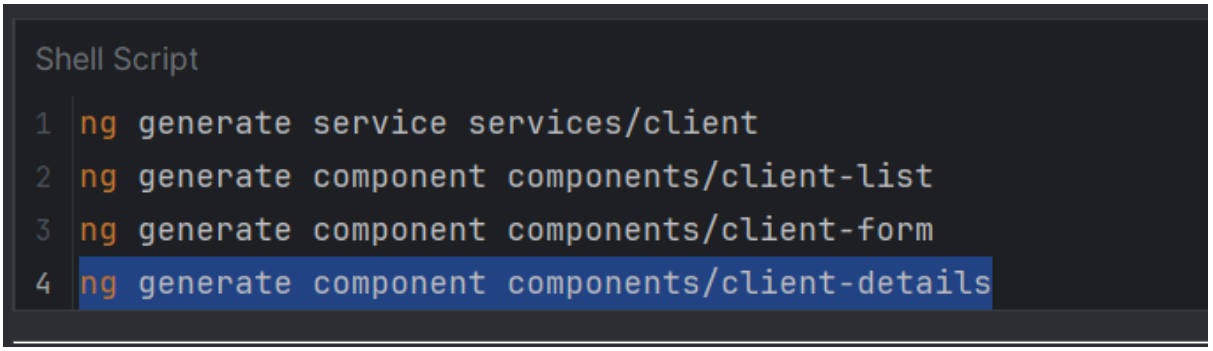
result

Swagger

j’ai cree la partie front utilisons le cmd



generating the required components



j’ai pas

conclusion

En conclusion, cette implémentation pour gérer les clients via l'API Spring Boot, comprenant la liste des clients, l'ajout et la suppression, en utilisant des composants modulaires, un service dédié pour les appels REST, le routing entre les vues. Elle suit les bonnes pratiques de consommation d'API RESTful et nécessite une configuration CORS côté serveur, tout en offrant une base extensible pour ajouter des fonctionnalités comme l'édition, la validation avancée, ou la pagination selon les besoins.