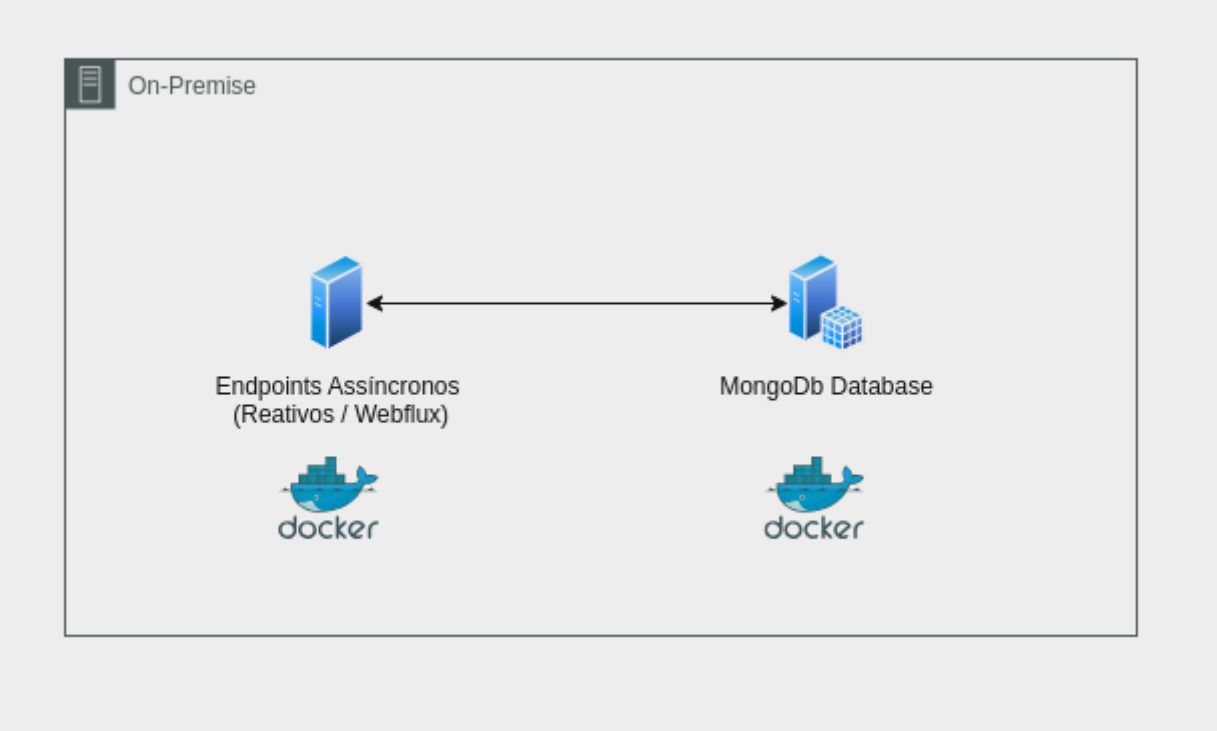
Arquitetura Escolhida

Arquitetura Física

Para este challenge, optamos por fornecer os serviços de maneira on-premises, utilizando o banco NoSQL MongoDB para armazenamento dos dados e também uma imagem docker própria, junto com um docker-compose para distribuição da aplicação. Segue abaixo diagrama simplificado da arquiteura escolhida



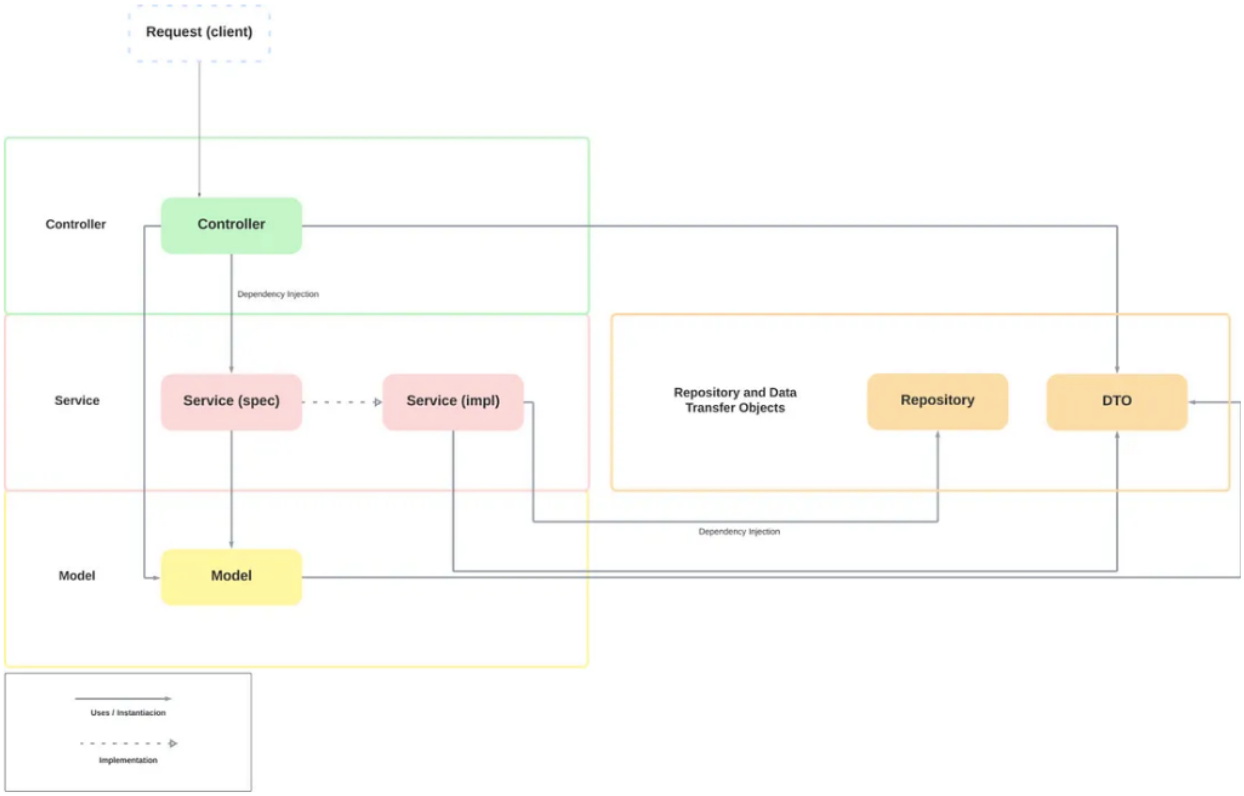
Arquitetura lógica

Para a arquitetura da aplicação (arquitetura lógica), definimos a Clean Architecture por ser um requisito da entrega do trabalho. Cabe ressaltar que o objetivo da clean architecture é focar na lógica e entidades de negócio, tendo como seu ideal fazer com que tecnologias "acessórias" como armazenamento, linguagens, frameworks e etc sejam tratados como mero "detalhe". Dito isso, nem sempre é possível tratar todos os aspectos extra-negociais como mero datalhe, como por exemplo a escolha entre adotar um paradigma reativo (através do WebFlux) ou adotar o tradicional paradigma síncrono. Como será mostrado no decorrer dessa documentação, tentar abstrair e tornar os paradigmas intercambiáveis traria maiores problemas que soluções, portanto, optamos por não abstrair os elementos do paradigma reativo por não enxergarmos benefício (ou até mesmo viabilidade) em tal ação.

Transição do MVC Para Clean Arch

A Arquitetura MVC

Nos desafios anteriores, estruturamos as aplicações da seguinte forma:

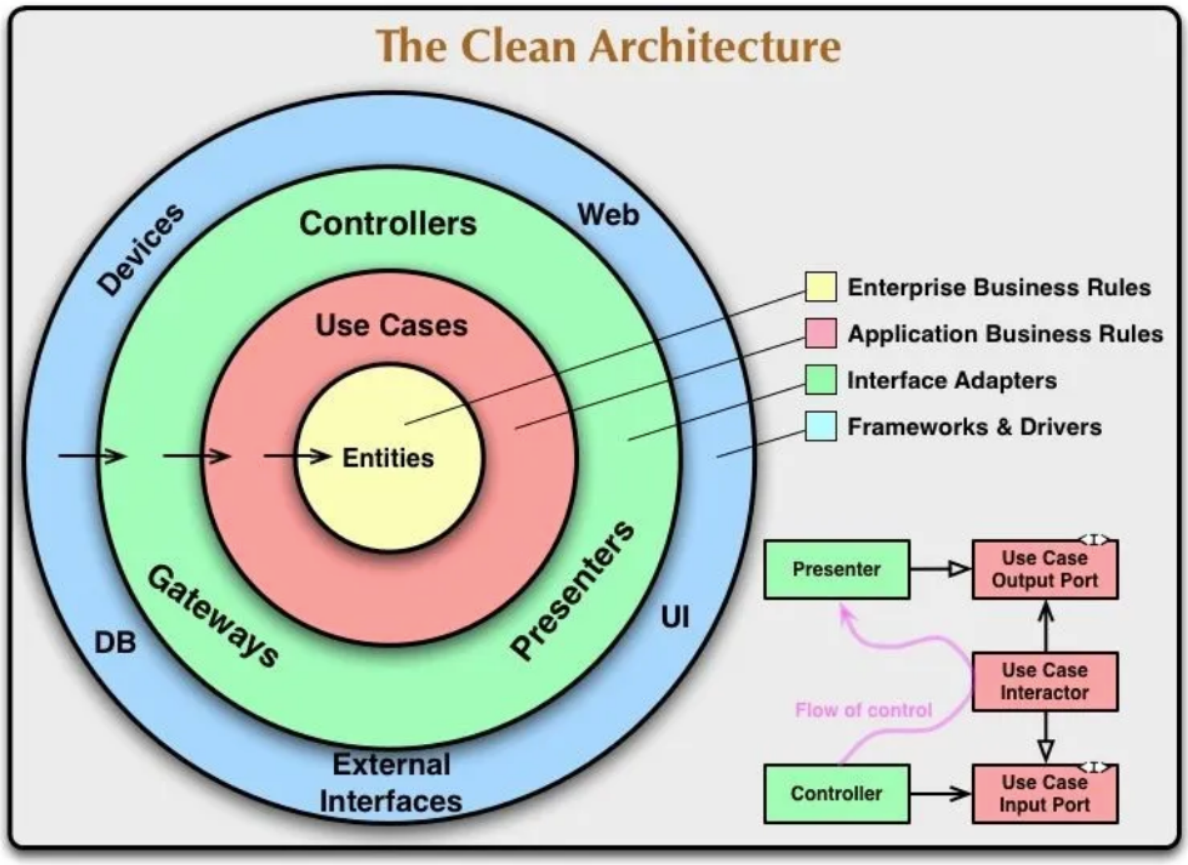


De maneira simplificada, nossa porta de entrada são os Controllers, que se comunicam com os Services através de mensagens trafegando DTOs. Os Services são definidos por interface e sua implementação se dá na mesma camada, tem a responsabilidade de comunicar-se com o modelo de nossa aplicação enquanto sequencia ações, encapsulando parte das regras de negócios e validações. Já o modelo encapsula as entidades e principais regras de negócio de nossa aplicação, comumente encapsulando nossa lógica de persistência também.

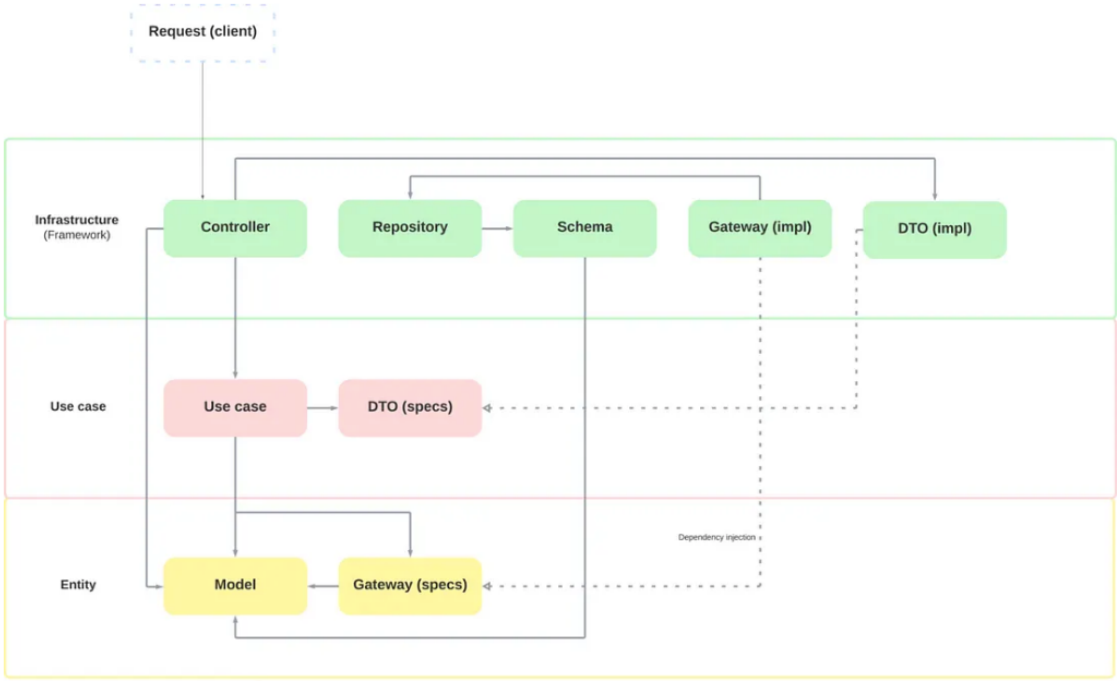
A Clean Arch

A arquitetura Limpa (Clean Arch), traz um outro olhar, procurando adicionar mais camadas de indireção a fim de isolar nossas camadas de negócio, usando a regra de que camadas mais interiores não devem conhecer detalhes das camadas mais exteriores (apenas as mais exteriores devem conhecer as interiores, conforme as setas do diagrama abaixo), enquanto ao mesmo tempo valoriza muito a separação de responsabilidades e todo o arcabouço do SOLID. O tradicional desenho abaixo ajuda a ilustrar como a

arquitetura é vista em sua maneira ideal:



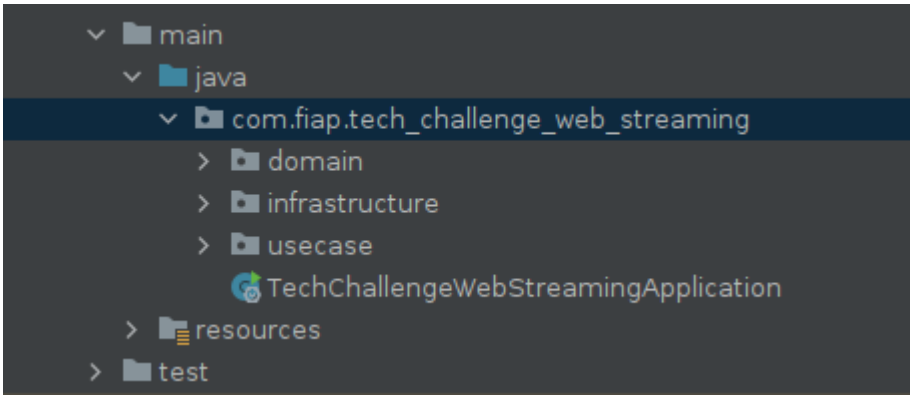
O MVC enxergado através do Clean Arch



Conforme podemos ver na imagem acima (com o mesmo esquema de cores do Clean Arch), podemos exemplificar como se dará a estruturação de nossa arquitetura para o challenge. Basicamente, concentraremos em nossa camada mais interna as nossas entidades e **especificações de gateways**. Na camada intermediária encontraremos o sequenciamento de nossa lógica de negócio, os casos de uso

propriamente ditos e a especificação dos DTOs que servirão de comunicação com nossa camada mais externa.

Já na camada mais externa teremos a parte ligada ao framework, os entrypoints de nossa aplicação, as representações de nossas entidades no banco (os schemas delas) e a implementação dos gateways que foram definidos na camada mais interna. Uma visão bem resumida da organização de nossos pacotes segue abaixo:

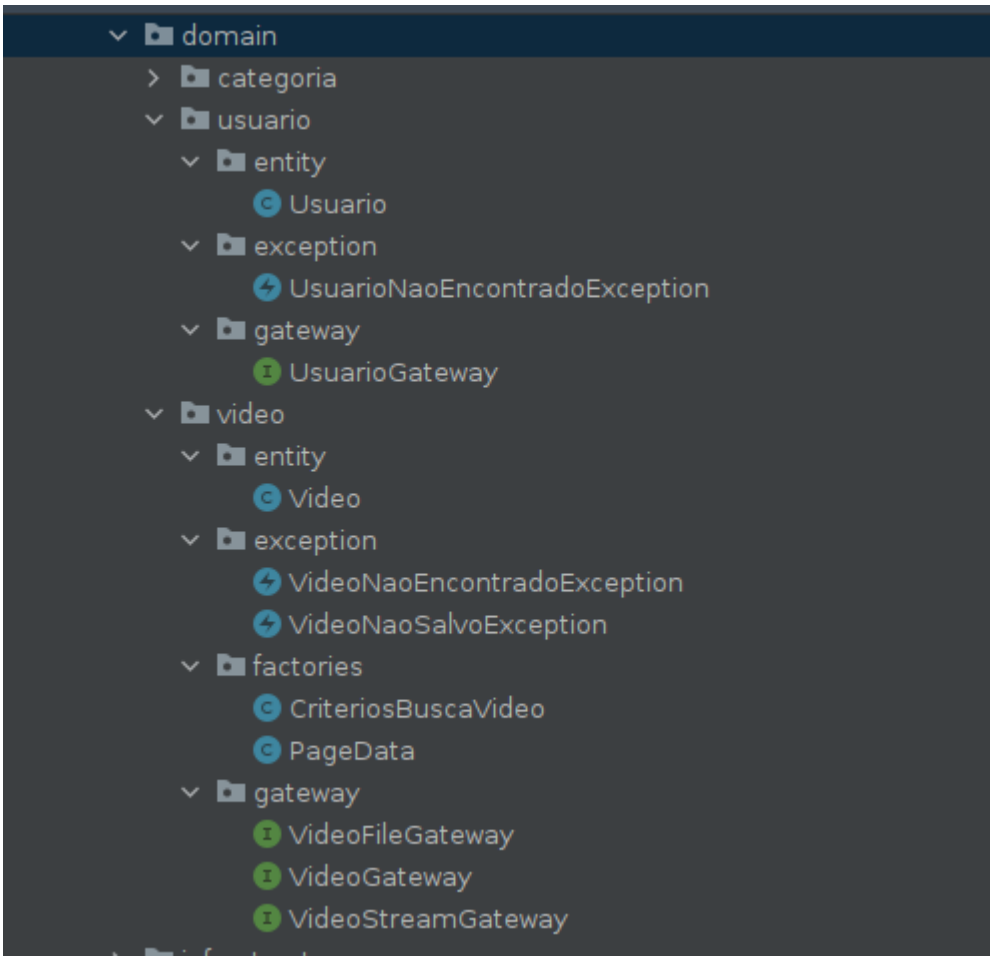


A seguir isolaremos cada camada trazendo exemplos de como foi realizado em nosso projeto.

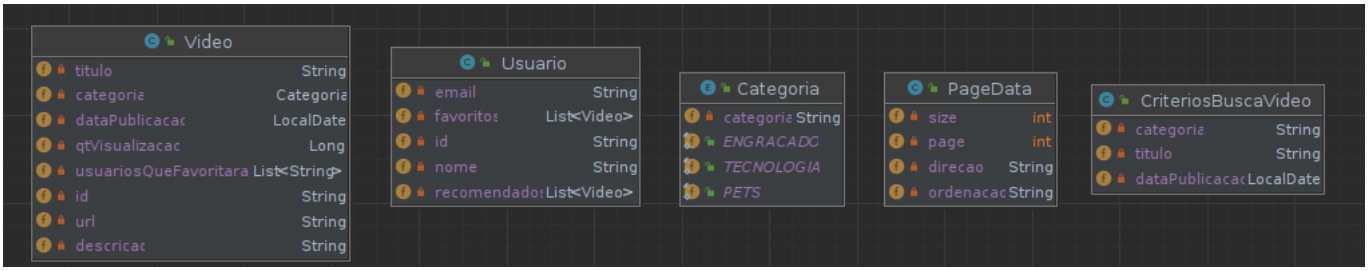
Domain

Como dito acima, essa camada armazenará as entidades de nosso negócio e definirá o contrato que deverá ser implementado por nossos gateways, também inserimos nessa camada a representação das exceções de nosso projeto.

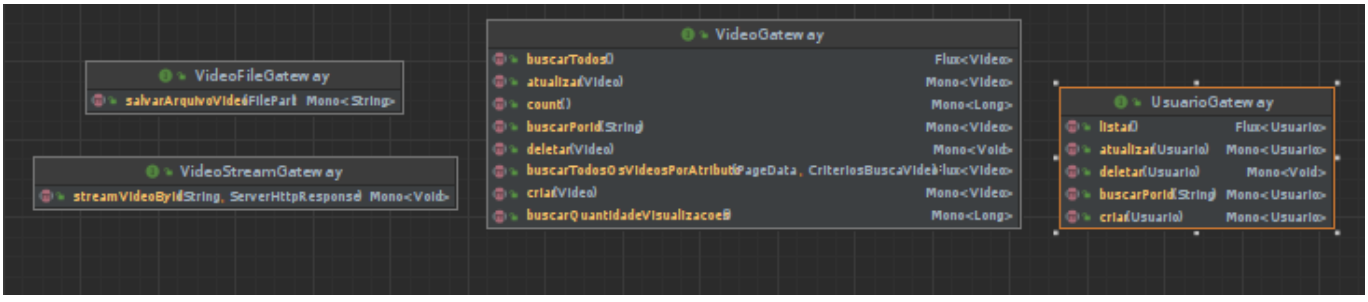
Segue a organização dos pacotes:



Nessa camada nossas entidades são representadas em seu estado puro, não utilizando nenhum tipo de anotação que gere acoplamento a algum framework de persistência, validação ou algo do gênero



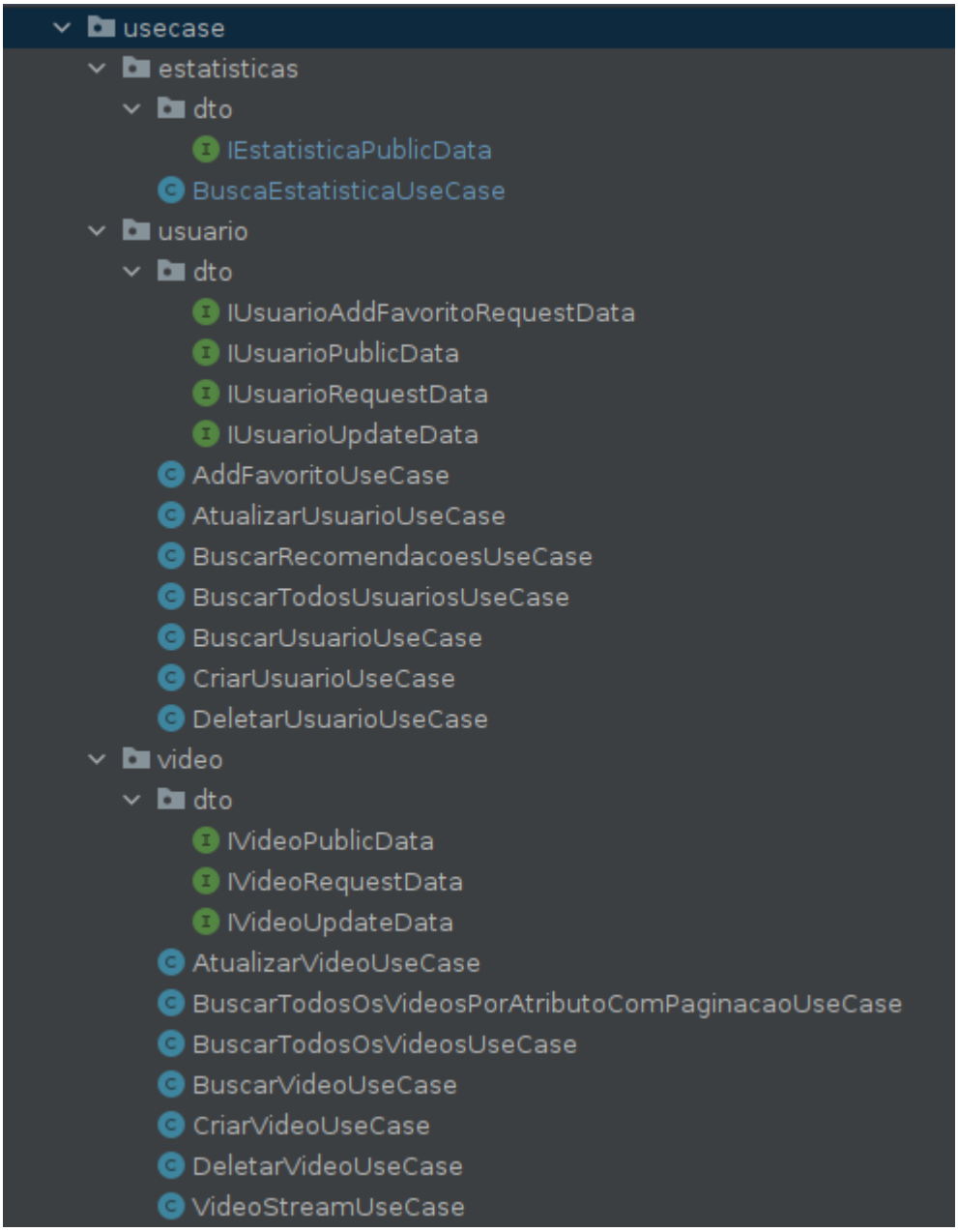
Também definimos as interfaces de nossos Gateways para especificar os contratos que deverão ser implementados para que nossa aplicação converse com os entes das camadas mais externas



Reforçando que nesse ponto definimos apenas os contratos, o Gateway por si só existirá apenas na camada mais externa.

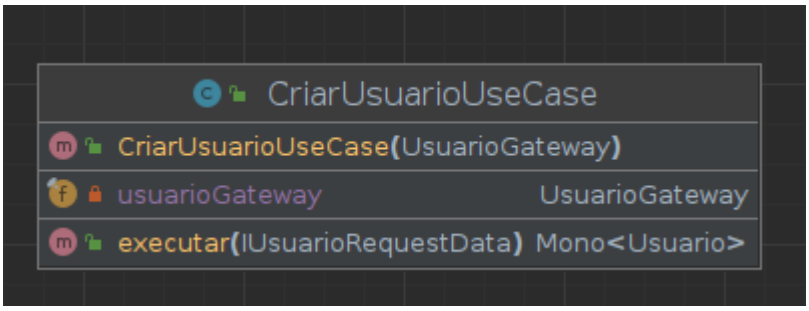
Use Cases

A camada de Use Case é a grande orquestradora de nossa aplicação, através dela realizaremos as ações que nossa aplicação deve prover para atender sua razão de existir.
Basicamente realizamos a divisão de pacotes entre os Use Cases propriamente ditos e a especificação dos DTOs que serão usados para comunicação com a camada acima



Usando o conceito de separação de responsabilidades, observe que cada classe de Use Case faz exatamente uma atividade. Outro detalhe importante de ressaltar é que nessa camada não há nenhuma injeção de dependencias sendo realizada através de anotações, uma vez que a camada de Use Cases também deve ser (o máximo possível) agnóstica de tecnologias externas como frameworks.

conforme pode ser visto no diagrama e exemplo abaixo




```java
public class CriarUsuarioUseCase {

    private final UsuarioGateway usuarioGateway;

    Felipe Santos
    public CriarUsuarioUseCase(UsuarioGateway usuarioGateway) { this.usuarioGateway = usuarioGateway; }


    Felipe Santos
    public Mono<Usuario> executar(IUsuarioRequestData dados) {

        Usuario usuario = new Usuario(dados.nome(), dados.email());

       return this.usuarioGateway.criar(usuario);

    }
}
```

A classe receberá via injeção no construtor (que será explicada posteriormente) o Gateway para se comunicar com o as camadas mais exteriores (observe que passamos a interface e não uma implementação propriamente dita). Seu único método recebe o DTO responsável por representar os dados de entrada e retornará um Mono da entidade Usuário.

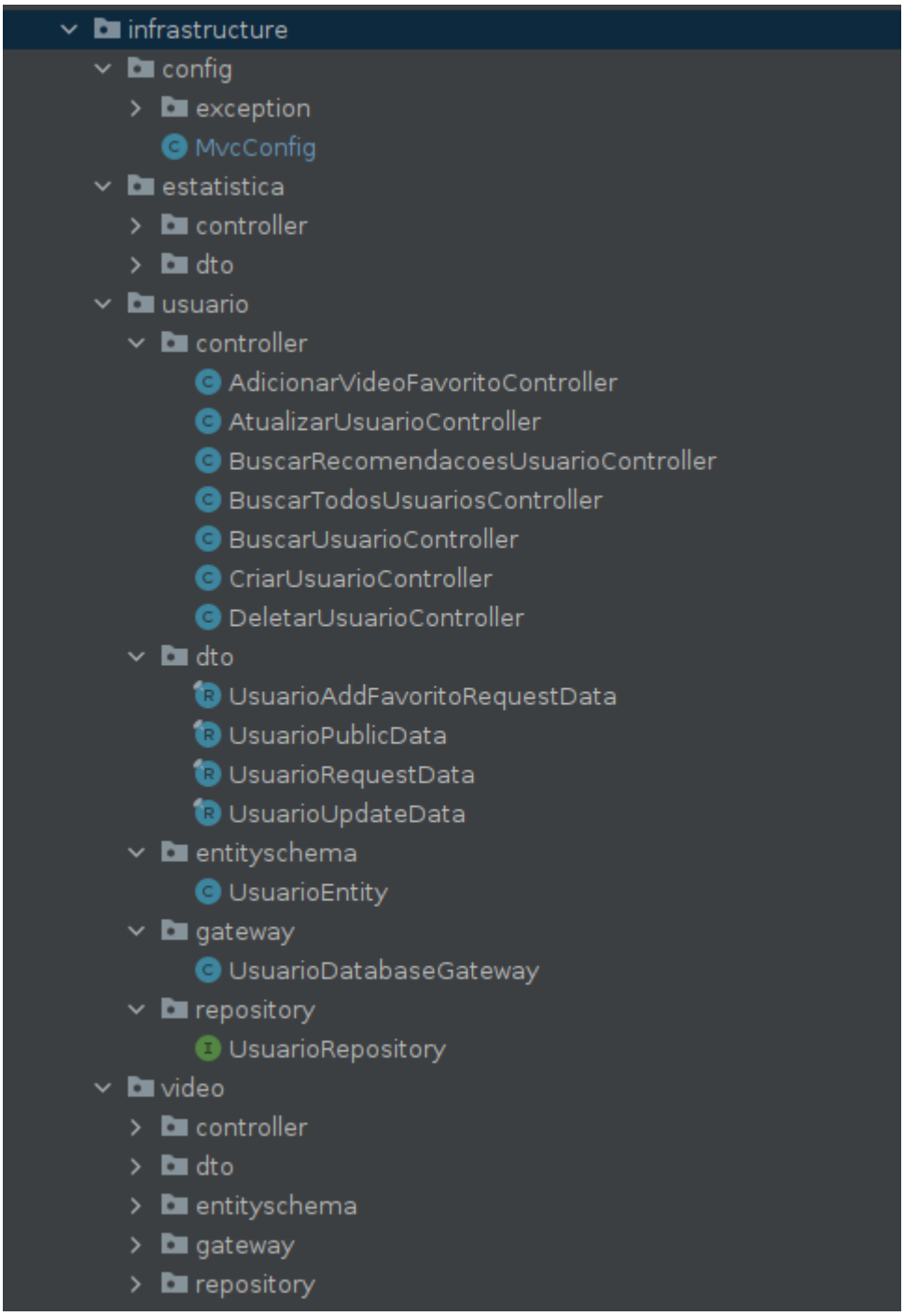
Nesse ponto cabe uma importante observação: o Mono é uma representação da camada mais externa, do framework WebFlux, o que de certa quebraria a regra de não acoplar a camada mais interna com alguma tecnologia / framework. Porém **ao tomarmos decisões arquiteturais muito intrínsecas à aplicação, como trabalhar com o paradigma reativo, algumas abstrações não são proveitosas (ou até mesmo possíveis de serem realizadas)**, por exemplo, caso optássemos por retornar apenas um Usuario, ao invés de um Mono <Usuario> iríamos comprometer a parte reativa de nossa aplicação, pois seria necessária uma chamada "block()" ou "subscribe()", tornando-a virtualmente síncrona, mesmo usando o paradigma reativo.

Há situações em que precisaremos de mais de um gateway para realizar um casos de uso, como por exemplo, em nosso caso de uso de upload de vídeos, onde devemos salvar o arquivo e os metadados do mesmo.

```java
public class CriarVideoUseCase {


    private final VideoGateway videoGateway;
    private final VideoFileGateway videoFileGateway;

    Felipe Santos
    public CriarVideoUseCase(VideoGateway videoGateway, VideoFileGateway videoFileGateway) {
        this.videoGateway = videoGateway;
        this.videoFileGateway = videoFileGateway;
    }

    Felipe Santos *
    public Mono<Video> execute(IVideoRequestData videoMetadata, FilePart videoFile) {

        return videoFileGateway.salvarArquivoVideo(videoFile)
                .flatMap(videoUrl -> {
                    Video video = new Video(videoMetadata.titulo(), videoMetadata.descricao(),
                            Categoria.valueOf(videoMetadata.categoria()),
                            videoUrl);
                    return videoGateway.criar(video);
                });
    }
}
```

Infraestructure

Essa é a camada de maior tamanho de nossa aplicação, ela encapsulará as tecnologias, frameworks, acesso a dados, fronteiras da aplicação, etc. Organizamos nossos pacotes dentro dessa camada da seguinte forma:



De maneira geral, podemos abstrair da seguinte forma

- entidade (usuario, video)
  - controller
  - dto
  - entityschema
  - gateway
  - repository

Controller

Representa a fronteira de nossa aplicação, nessa camada sim podemos utilizar as anotações e todos os recursos do framework pois já estamos falando da camada mais externa.

```java
Felipe Santos
@Tag(name = "Usuário", description = "Usuário API")
@RestController
public class CriarUsuarioController {

    private final CriarUsuarioUseCase criarUsuarioUseCase;


    Felipe Santos
    public CriarUsuarioController(CriarUsuarioUseCase criarUsuarioUseCase) {
        this.criarUsuarioUseCase = criarUsuarioUseCase;
    }

    Felipe Santos
    @PostMapping("/usuarios")
    @Operation(summary = "Novo Usuário")
    public Mono<ResponseEntity<UsuarioPublicData>> criarUsuario(@Valid @RequestBody UsuarioRequestData usuario) {
        Mono<Usuario> usuarioCriado = criarUsuarioUseCase.executar(usuario);
        return usuarioCriado.map(u -> new ResponseEntity<>(new UsuarioPublicData(u), HttpStatus.CREATED));
    }
```

DTO

A implementação das definições de formato de dados feita no Use Case é implementada aqui, bem como as validações de dados já na fronteira da aplicação usando o pacote do Jakarta Validation e aplicando o princípio do fail fast.

```java
Felipe Santos +2
public record UsuarioRequestData(

        //Incluir Spring Validation aqui

        @NotBlank(message = "Nome não pode ser vazio")
        String nome,

        @Email(message = "Email inválido")
        @NotBlank(message = "Email não pode ser vazio")
        String email



) implements IUsuarioRequestData {

}
```

EntitySchema

Aqui temos a representação de nossa entidade voltada para um schema de dados de algum banco de dados, no nosso caso decidimos por utilizar o MongoDB (até por ser um dos requisitos do challenge), mas perceba que essa camada conhece nossa camada de entidade de negócio e ela sabe como construir essa camada mais interior e sabe construir-se a partir dela, respeitando a regra de apenas camadas mais exteriores poderem utilizar as mais interiores.

Utilizarmos uma camada específica para representar os dados de persistencia ao invés de simplesmente mapeá-los diretamente na camada mais interna da arquitetura nos permite maior flexibilidade, como por

exemplo, termos diversas classes nessa camada, cada uma representando o armazenamento de dados de uma maneira diferente (uma para não relacional, outra para relacional, outra em csv, etc).

```java
kmentz +2
@Document(collection = "usuarios")
public class UsuarioEntity {

    @Id
    private String id;
    private String nome;
    private String email;
    private List<Video> favoritos;
    private List<Video> recomendados;

    kmentz +1
    public UsuarioEntity(){
        favoritos = new ArrayList<>();
        recomendados = new ArrayList<>();
    }

    kmentz +1
    public UsuarioEntity(String id, String nome, String email, List<Video> favoritos, List<Video> recomendados){
        this.id = id;
        this.nome = nome;
        this.email = email;
        this.favoritos = favoritos;
        this.recomendados = recomendados;
    }

    kmentz +1
    public UsuarioEntity(String id, String nome, String email){
        this.id = id;
        this.nome = nome;
        this.email = email;
    }

    Felipe Santos
    public UsuarioEntity(Usuario usuario) {
        this.id = usuario.getId();
        this.nome= usuario.getNome();
        this.email = usuario.getEmail();
        this.favoritos = usuario.getFavoritos();
        this.recomendados = usuario.getRecomendados();


    }

    Felipe Santos
    public Usuario toUsuario() {
        return new Usuario(this.id, this.nome, this.email, this.getFavoritos(), this.recomendados);
    }
```

Gateway

Essa é a camada que implementará as especificações das interações com as camadas mais externas, as quais a camada de entidade definiu anteriormente. Essa camada é essencial para que a camada de Use Case realize suas atividades pois permitirá a comunicação com os entes mais externos. Novamente, o benefício do isolamento dessa responsabilidade é podermos substituir por outro gateway conforme a nossa necessidade, por exemplo, na situação abaixo, nosso gateway utiliza a camada de repositório para prover os métodos que serão posteriormente utilizados por nosso Use Case. Porém, seria plenamente possível que esses dados na verdade não fossem acessados através de um banco de dados e sim de através de uma API externa através de chamadas REST.Separar as camadas nos permite criar um novo gateway que respeite o

contrato definido na entidade e obtenha os dados dessa forma, bastando apenas injetar esse novo gateway em nosso Use Case e utilizá-lo.

```java
Felipe Santos +1
public class UsuarioDatabaseGateway implements UsuarioGateway {

    private final UsuarioRepository repository;

    Felipe Santos
    public UsuarioDatabaseGateway(UsuarioRepository repository) { this.repository = repository; }


    Felipe Santos
    @Override
    public Mono<Usuario> criar(Usuario usuario) {
        UsuarioEntity usuarioEntity = new UsuarioEntity(usuario);
        return repository.save(usuarioEntity).map(UsuarioEntity::toUsuario);
    }

    Felipe Santos
    @Override
    public Mono<Usuario> atualizar(Usuario usuario) {
        UsuarioEntity usuarioEntity = new UsuarioEntity(usuario);
        return repository.save(usuarioEntity).map(UsuarioEntity::toUsuario);
    }

    Felipe Santos +1
    @Override
    public Mono<Void> deletar(Usuario usuario) {
        UsuarioEntity usuarioEntity = new UsuarioEntity(usuario);
        return repository.delete(usuarioEntity).then();
    }

    Felipe Santos
    @Override
    public Flux<Usuario> listar() { return repository.findAll().map(UsuarioEntity::toUsuario); }

    Felipe Santos
    @Override
    public Mono<Usuario> buscarPorId(String id) {
         return repository.findById(id).map(
                 UsuarioEntity::toUsuario
         );
    }
}
```

Repository

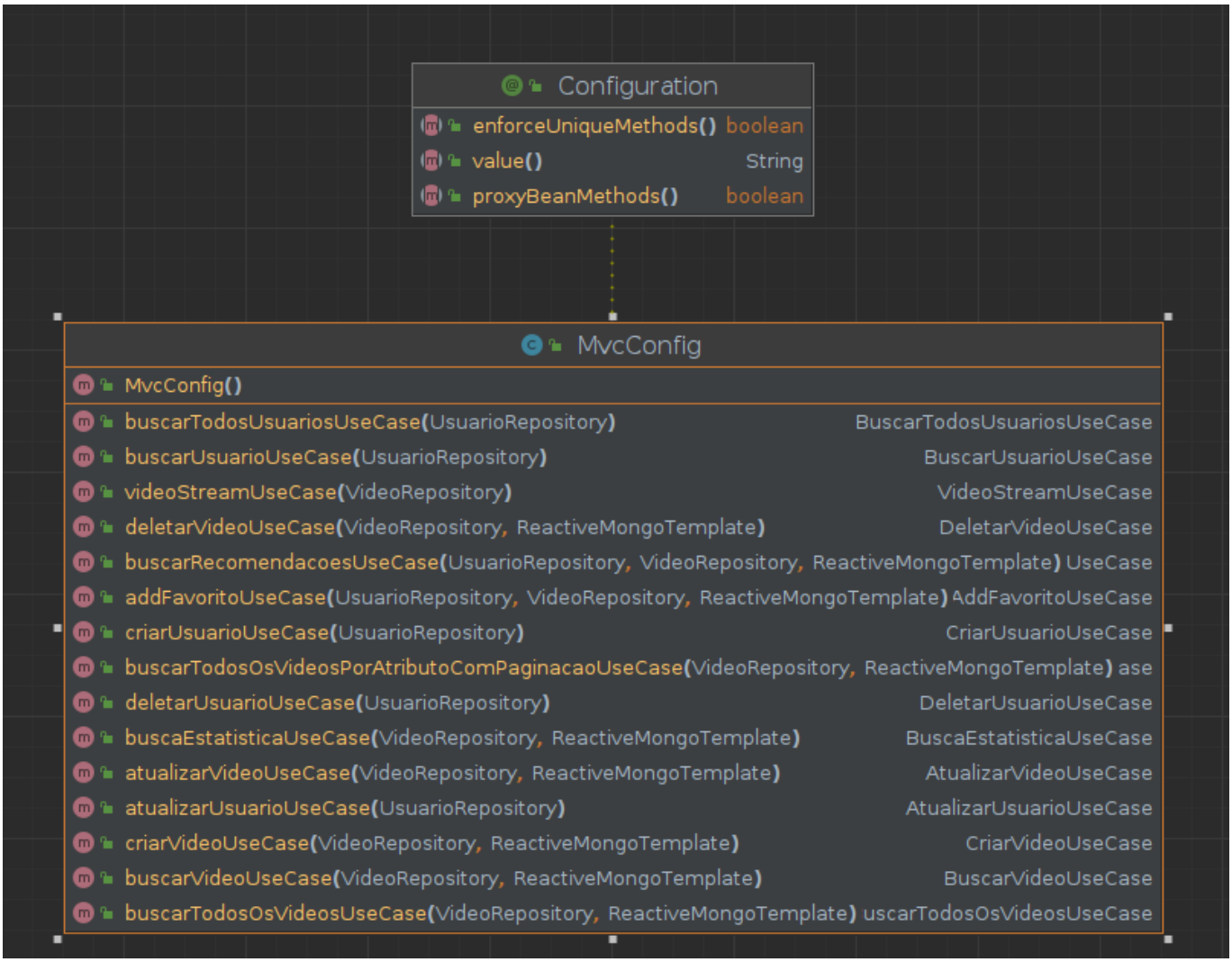
Essa é camada que será utilizada por nosso gateway para as chamadas ao banco de dados, em nosso contexto, ela será provida pelo Spring Data conforme podemos ver abaixo

```java
Felipe Santos
public interface UsuarioRepository extends ReactiveMongoRepository <UsuarioEntity, String> {



}
```

O pacote config Como mencionamos anteriormente, nenhum Use Case está anotado como @Component, @Service ou algo semelhante para evitarmos o acoplamento de tecnologia naquela camanda. Para provermos beans desse tipo e injetá-los em nosso controller, criamos um pacote de configurações e nele

criamos a classe MvcConfig que é responsável por fornecer ao Spring a maneira pela qual os Beans de Use Case serão injetados no Controller.

```java
Felipe Santos +3
@Configuration
public class MvcConfig {

    Felipe Santos
    @Bean
    public CriarUsuarioUseCase criarUsuarioUseCase(UsuarioRepository repository){
        UsuarioGateway usuarioGateway = new UsuarioDatabaseGateway(repository);
        return new CriarUsuarioUseCase(usuarioGateway);

    }

    Felipe Santos
    @Bean
    public BuscarUsuarioUseCase buscarUsuarioUseCase(UsuarioRepository repository){
        UsuarioGateway usuarioGateway = new UsuarioDatabaseGateway(repository);
        return new BuscarUsuarioUseCase(usuarioGateway);

    }

    Felipe Santos
    @Bean
    public BuscarTodosUsuariosUseCase buscarTodosUsuariosUseCase(UsuarioRepository repository){
        UsuarioGateway usuarioGateway = new UsuarioDatabaseGateway(repository);
        return new BuscarTodosUsuariosUseCase(usuarioGateway);
    }

    Felipe Santos
    @Bean
    public AtualizarUsuarioUseCase atualizarUsuarioUseCase(UsuarioRepository repository){
        UsuarioGateway usuarioGateway = new UsuarioDatabaseGateway(repository);
        return new AtualizarUsuarioUseCase(usuarioGateway);
    }

    Felipe Santos
    @Bean
    public DeletarUsuarioUseCase deletarUsuarioUseCase(UsuarioRepository repository){
        UsuarioGateway usuarioGateway = new UsuarioDatabaseGateway(repository);
        return new DeletarUsuarioUseCase(usuarioGateway);
    }
```


Configuration
enforceUniqueMethods() boolean
value() String
proxyBeanMethods() boolean
MvcConfig
MvcConfig()
buscarTodosUsuariosUseCase(UsuarioRepository) BuscarTodosUsuariosUseCase
buscarUsuarioUseCase(UsuarioRepository) BuscarUsuarioUseCase
videoStreamUseCase(VideoRepository) VideoStreamUseCase
deletarVideoUseCase(VideoRepository, ReactiveMongoTemplate) DeletarVideoUseCase
buscarRecomendacoesUseCase(UsuarioRepository, VideoRepository, ReactiveMongoTemplate) UseCase
addFavoritoUseCase(UsuarioRepository, VideoRepository, ReactiveMongoTemplate) AddFavoritoUseCase
criarUsuarioUseCase(UsuarioRepository) CriarUsuarioUseCase
buscarTodosOsVideosPorAtributoComPaginacaoUseCase(VideoRepository, ReactiveMongoTemplate) ase
deletarUsuarioUseCase(UsuarioRepository) DeletarUsuarioUseCase
buscaEstatisticaUseCase(VideoRepository, ReactiveMongoTemplate) BuscaEstatisticaUseCase
atualizarVideoUseCase(VideoRepository, ReactiveMongoTemplate) AtualizarVideoUseCase
atualizarUsuarioUseCase(UsuarioRepository) AtualizarUsuarioUseCase
criarVideoUseCase(VideoRepository, ReactiveMongoTemplate) CriarVideoUseCase
buscarVideoUseCase(VideoRepository, ReactiveMongoTemplate) BuscarVideoUseCase
buscarTodosOsVideosUseCase(VideoRepository, ReactiveMongoTemplate) uscarTodosOsVideosUseCase