Decisões Técnicas Relavantes

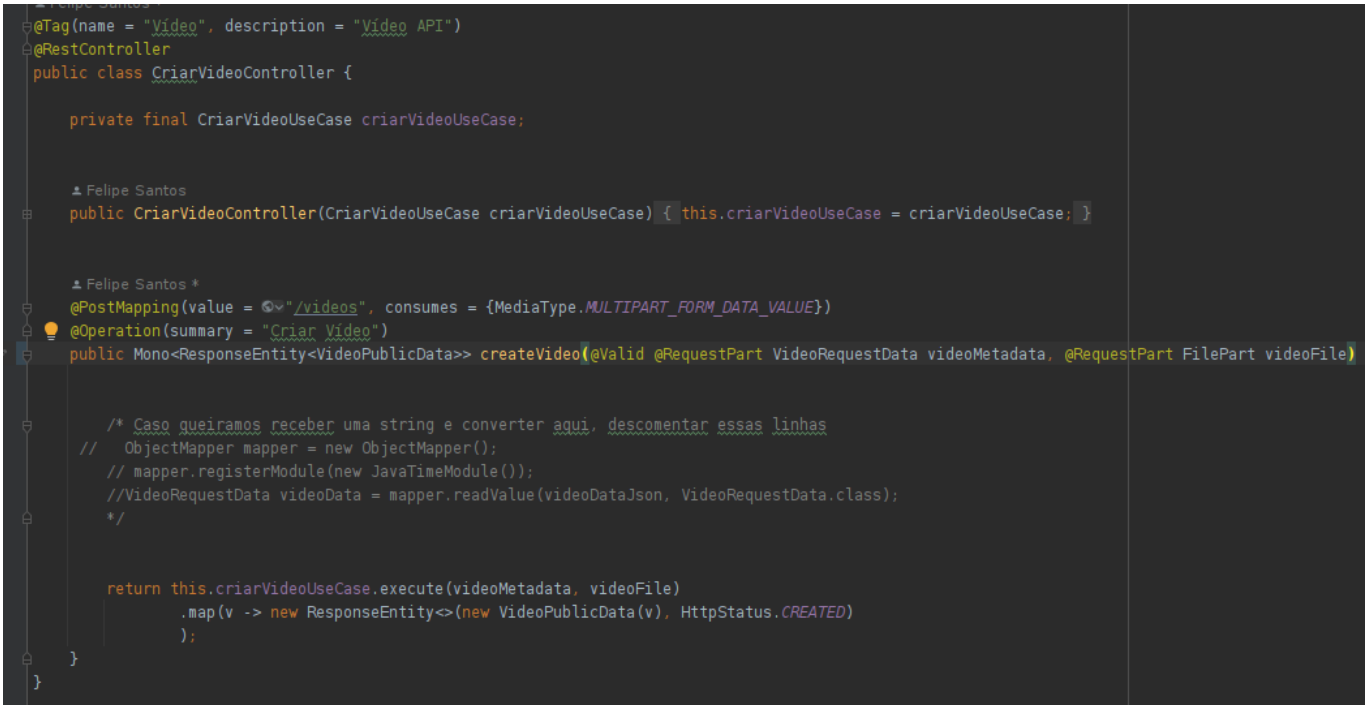
Arquitetura

Conforme já descrito no decorrer do documento de arquitetura, utilizamos a CleanArch e por estarmos utilizando o paradigma reativo através do WebFlux, optamos por fazer com que os Controllers retornem as respectivas representações reativas Mono e Flux, a justificativa para tal (e contexto) foi dado na própria documentação de arquitetura, especificamente na área de Use Cases: Arquitetura

> 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.

Forma de upload do vídeo

Optamos por utilizar o formato **Multi-Part-Form-Data** para receber na mesma requisição o arquivo do vídeo e seus meta dados, conforme pode ser visto abaixo:



```java
@Tag(name = "Vídeo", description = "Vídeo API")
@RestController
public class CriarVideoController {

    private final CriarVideoUseCase criarVideoUseCase;


    Felipe Santos
    public CriarVideoController(CriarVideoUseCase criarVideoUseCase) { this.criarVideoUseCase = criarVideoUseCase; }


    Felipe Santos *
    @PostMapping(value = "/videos", consumes = {MediaType.MULTIPART_FORM_DATA_VALUE})
    @Operation(summary = "Criar Vídeo")
    public Mono<ResponseEntity<VideoPublicData>> createVideo(@Valid @RequestPart VideoRequestData videoMetadata, @RequestPart FilePart videoFile)


        /* Caso queiramos receber uma string e converter aqui, descomentar essas linhas
     //   ObjectMapper mapper = new ObjectMapper();
        // mapper.registerModule(new JavaTimeModule());
        //VideoRequestData videoData = mapper.readValue(videoDataJson, VideoRequestData.class);
        */


        return this.criarVideoUseCase.execute(videoMetadata, videoFile)
                .map(v -> new ResponseEntity<>(new VideoPublicData(v), HttpStatus.CREATED)
                );
    }
}
```

A motivação para tal escolha foi poder receber ambos os dados em uma mesma requisição, de forma que caso uma validação falhe ou por algum motivo o arquivo não possa ser salvo, toda a operação de inserção é cancelada, evitando assim possíveis registros fantasmas que poderiam surgir.

Para consumir o endpoint e enviar metadados e vídeo ao mesmo tempo, precisamos fazer a requisição de forma semelhante a essa:

```
curl --location 'localhost:8080/videos' \
--header 'Content-Type: multipart/form-data' \
--form 'videoMetadata="{\"titulo\": \"teste 1\",
\"descricao\": \"inserção de vídeo teste 2\",
\"categoria\": \"TECNOLOGIA\",
\"dataPublicacao\": \"2024-01-25\"}";type=application/json' \
--form 'videoFile=@"/home/felipe/Downloads/videoplayback.mp4"'
```

Atenção pois na forma acima, é necessário informar o type do vidoMetadata para que o Jackson entenda que a string é um json e faça o parse do RequestPart automaticamente para nós.

Deixamos comentada no código uma forma alternativa que faz o parse manual da String em Json, possibilitando que a necessidade de passar o type seja suprimida.

```
   ObjectMapper mapper = new ObjectMapper();
        mapper.registerModule(new JavaTimeModule());
       VideoRequestData videoData = mapper.readValue(videoDataJson,
VideoRequestData.class);
```

Stream do Vídeo

Apesar de não ser um requisito, também fornecemos um endpoint para que o vídeo pudesse ser baixado, acessível através do recurso:

```
\stream\{videoId}
```

Isolamento dos testes Unitários / Integração

Optamos por filtrar a execução dos testes de integração na chamada mvn test para agilizar a execução do comando. Esse procedimento foi feito através dos profiles do maven, conforme podemos ver abaixo:

```xml
<profiles>
<profile>
    <id>integration-test</id>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-surefire-plugin</artifactId>
                <version>3.1.2</version>
                <configuration>
                    <includes combine.self="override">
                        <include>**/*IT.java</include>
                    </includes>
                    <excludes combine.self="override">
                        <exclude>**/*Test.java</exclude>
                    </excludes>
                </configuration>
            </plugin>
        </plugins>
    </build>
</profile>
</profiles>
```

Utilização do Make

Criamos um Makefile para simplificar e agilizar nossos deploys e criação de nossa infra.

Criação da Docker Image

Criamos nossa própria imagem docker da aplicação e optamos por utilizar o multi-staged build, de forma que a aplicação final ficasse menor. Também optamos por explodir nosso jar final em camadas para tirarmos vantagem do OverlayFS (Filesystem do Docker), dessa forma, obtemos um reaproveimento maior das camadas inferiores e diminuimos o custo de armazenamento e o tempo para criação das imagens. Segue abaixo uma imagem de nosso DockerFile

```dockerfile
FROM eclipse-temurin:17-jdk-alpine as build
ARG MVN_OPTIONS="-Dspring.data.mongodb.uri=mongodb://mongo:27017/db_web_streaming"
ARG MVN_PROFILE
WORKDIR /workspace/app


COPY mvnw .
COPY .mvn .mvn
COPY pom.xml .
COPY src src

RUN ./mvnw install -DskipTests ${MVN_OPTIONS} ${MVN_PROFILE}
RUN mkdir -p target/dependency && (cd target/dependency; jar -xf ../*.jar)

FROM eclipse-temurin:17-jdk-alpine
VOLUME /tmp
ARG DEPENDENCY=/workspace/app/target/dependency

COPY --from=build ${DEPENDENCY}/BOOT-INF/lib /app/lib
COPY --from=build ${DEPENDENCY}/META-INF /app/META-INF
COPY --from=build ${DEPENDENCY}/BOOT-INF/classes /app

ENTRYPOINT ["java","-cp","app:app/lib/*","-Dspring.data.mongodb.uri=mongodb://mongo:27017/db_web_streaming" ,"com.fiap.tech_challenge_web_streaming.TechChallengeWebStreamingApplication"]
```

a primeira área é responsável por buildar nossa aplicação, baixando as dependencias do maven. O artefato final gerado por esse estágio é explodido através do jar -xf

o segundo estágio copia os arquivos produzidos pelo primeiro, cada cópia irá gerar uma nova camada, sendo assim caso mudemos apenas as classes, o build não precisará gerar novamente as bibliotecas pois a mesma está em um camada mais inferior, inalterada. Como explodimos o jar, o entrypoint da aplicação não

pode ser o java -jar tradicional, no lugar dele usamos o java -cp, especificando as pastas produzidas como classpath e executando nosso método main. O código abaixo é a cópia do entrypoint da imagem

```
ENTRYPOINT ["java","-cp","app:app/lib/*","-
Dspring.data.mongodb.uri=mongodb://mongo:27017/db_web_streaming"
,"com.fiap.tech_challenge_web_streaming.TechChallengeWebStreamingApplicatio
n"]
```

também passamos o endereço do mongo como argumento para a execução da aplicação.

Test Coverage

Utilizamos o Jacoco para produzir nossas métricas de test coverage, segue abaixo imagem das estatísticas produzidas (atendendo aos requisitos técnicos)

tech_challenge_web_streaming

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.fiap.tech_challenge_web_streaming.infrastructure.video.gateway		69%		75%	7	21	15	57	5	17	0	3
com.fiap.tech_challenge_web_streaming.usecase.usuario		75%		40%	10	30	12	65	5	25	0	7
com.fiap.tech_challenge_web_streaming.domain.usuario.entity		73%		n/a	3	16	13	45	3	16	0	1
com.fiap.tech_challenge_web_streaming.infrastructure.usuario.entityschema		80%		n/a	4	16	11	41	4	16	0	1
com.fiap.tech_challenge_web_streaming.infrastructure.video.entityschema		90%		50%	6	23	3	37	2	19	0	1
com.fiap.tech_challenge_web_streaming.domain.video.entity		95%		n/a	2	21	4	56	2	21	0	1
com.fiap.tech_challenge_web_streaming.usecase.video.dto		0%		n/a	4	4	4	4	4	4	1	1
com.fiap.tech_challenge_web_streaming.domain.video.exception		36%		n/a	2	3	4	6	2	3	1	2
com.fiap.tech_challenge_web_streaming.usecase.video		96%		50%	4	21	1	48	1	18	0	7
com.fiap.tech_challenge_web_streaming.infrastructure.video.controller		96%		n/a	1	17	1	47	1	17	0	7
com.fiap.tech_challenge_web_streaming.infrastructure.config.exception		95%		75%	1	16	1	39	0	14	0	3
com.fiap.tech_challenge_web_streaming		37%		n/a	1	2	2	3	1	2	0	1
com.fiap.tech_challenge_web_streaming.usecase.estatisticas		98%		50%	1	4	0	10	0	3	0	1
com.fiap.tech_challenge_web_streaming.infrastructure.config		100%		n/a	0	16	0	34	0	16	0	1
com.fiap.tech_challenge_web_streaming.infrastructure.usuario.controller		100%		n/a	0	18	0	40	0	18	0	7
com.fiap.tech_challenge_web_streaming.domain.video.factories		100%		n/a	0	20	0	36	0	20	0	4
com.fiap.tech_challenge_web_streaming.infrastructure.video.dto		100%		n/a	0	4	0	11	0	4	0	3
com.fiap.tech_challenge_web_streaming.infrastructure.usuario.dto		100%		n/a	0	5	0	11	0	5	0	4
com.fiap.tech_challenge_web_streaming.infrastructure.usuario.gateway		100%		n/a	0	6	0	11	0	6	0	1
com.fiap.tech_challenge_web_streaming.domain.categoria.entity		100%		n/a	0	3	0	8	0	3	0	1
com.fiap.tech_challenge_web_streaming.infrastructure.estatistica.controller		100%		n/a	0	2	0	6	0	2	0	1
com.fiap.tech_challenge_web_streaming.infrastructure.estatistica.dto		100%		n/a	0	1	0	1	0	1	0	1
com.fiap.tech_challenge_web_streaming.domain.usuario.exception		100%		n/a	0	1	0	2	0	1	0	1
Total	254 of 2,261	88%	17 of 38	55%	46	270	71	618	30	251	2	60

Pode-se ver que **obtivemos 88%** de cobertura total.