**Gamelog [1.0]**

Alunos: João Marcus Cardoso e Igor Meurer.

Professor: Thiago Funk

Disciplina: Padrões de Projetos

**SUMÁRIO**

[**INTRODUÇÃO 2**](#_mz8h5omtbmcu)

[**Versionamento 3**](#_p8r96gkgmbs6)

[**REQUISITOS FUNCIONAIS 3**](#_xi6k5blp5efq)

[**REQUISITOS NÃO FUNCIONAIS 3**](#_5f5exc3imn4q)

[**REGRAS DE NEGÓCIO 3**](#_1xiyl77ph4z5)

[**RELATÓRIO 3**](#_jszylvlso0uw)

[**REFERÊNCIAS 4**](#_hm0mzdgcjnzx)

# **INTRODUÇÃO**

O projeto gamelog tem intuito de permitir que o usuário mantenha um catálogo de jogos e dlcs concluídos, permitindo assim que ele tenha um local onde ele possa organizar os seus jogos concluídos ou que esteja jogando. Podendo até mesmo criar reviews/notas sobre o que achou sobre o jogo e manter até mesmo as configurações como a plataforma em que ele jogou tal jogo.

Nesse software o usuário conseguirá criar conta e se autenticar, após isso será redirecionado para a página inicial, que seria sua “biblioteca” a onde ele poderá cadastrar seus jogos jogados, avaliá los e escrever reviews sobre eles. voltando ao seu perfil onde ele vai poder editar o mesmo, ele será redirecionado para uma tela a onde ele pode tanto mudar seu nickname, foto de perfil, bio e também adicionar seus jogos favoritos, que vão ser um total de 5 o qual o jogo que ele ditar como o seu top 1 favorito terá uma borda diferente do restante, voltando a página de perfil terá além dos jogos favoritos vai possuir também a contagem das horas e jogos adicionados na parte inferior do site, o site contará com um menu que possibilitará o redirecionamento para a página de exibição do catálogo e de jogos favoritos. Neste trabalho utilizaremos a linguagem Java com o Framework Spring Boot, juntamente do CSS, HTML5 e Javascript além do banco de dados Postgres.

# **Versionamento**

| Versão | Data | Alterações | Fonte | Autor |
| --- | --- | --- | --- | --- |
| 1.0 | 08/01/2023 | Criação do Escopo e Requisitos | João Marcus, Igor Meurer | João Marcus, Igor Meurer |

# **REQUISITOS FUNCIONAIS**

| ID | Descrição | Versão | Data | Fonte | Status |
| --- | --- | --- | --- | --- | --- |
| RF01 | Manter usuário | 1.0 | 09/01/2024 | Documento de requisição do aplicativo | Em construção |
| RF02 | Manter jogos | 1.0 | 09/01/2024 | Documento de requisição do aplicativo | Em construção |
| RF03 | Manter reviews | 1.0 | 09/01/2024 | Documento de requisição do aplicativo | Em construção |
| RF04 | Manter configs | 1.0 | 17/01/2024 | Documento de requisição do aplicativo | Em construção |
| RF05 | O sistema deve permitir filtrar os jogos por meio do filtro escolhido pelo usuário | 1.0 | 09/01/2024 | Documento de requisição do aplicativo | Em construção |
| RF06 | O sistema deve pegar a imagem do jogo por meio de uma url | 1.0 | 09/01/2024 | Documento de requisição do aplicativo | Em construção |
| RF07 | Ao clicar em um jogo na biblioteca o sistema deve abrir um modal com as informações do jogo | 1.0 | 09/01/2024 | Documento de requisição do aplicativo | Em construção |
| RF08 | O usuário pode escolher os seus jogos favoritos em seu perfil | 1.0 | 09/01/2024 | Documento de requisição do aplicativo | Em construção |
| RF09 | O jogo favorito principal deve ter uma borda dourada em volta dele | 1.0 | 09/01/2024 | Documento de requisição do aplicativo | Em construção |
| RF10 | Os jogos na biblioteca devem mostrar a nota da review que o usuário deu | 1.0 | 09/01/2024 | Documento de requisição do aplicativo | Em construção |
| RF11 | O sistema vai permitir que o usuário baixe seus jogos adicionados em formato de csv ou pdf | 1.0 | 17/01/2024 | Documento de requisição do aplicativo | Em construção |

# **REQUISITOS NÃO FUNCIONAIS**

| ID | Descrição | Versão | Data | Fonte | Status |
| --- | --- | --- | --- | --- | --- |
| RNF01 | O perfil de usuário deve contabilizar a quantidade de jogos cadastrado do usuário | 1.0 | 09/01/2024 | Documento de requisição do aplicativo | Em construção |
| RNF02 | O sistema deve Fazer uma média das review do usuário | 1.0 | 09/01/2024 | Documento de requisição do aplicativo | Em construção |
| RNF03 | O sistema deve mostrar uma prévia da imagem quando o usuário coloca a url dela | 1.0 | 09/01/2024 | Documento de requisição do aplicativo | Em construção |
| RNF04 | O usuário pode ter uma bio e uma foto de perfil | 1.0 | 09/01/2024 | Documento de requisição do aplicativo | Em construção |
| RNF05 | O Sistema deve mandar um e-mail para o usuário | 1.0 | 17/01/2024 | Documento de requisição do aplicativo | Em construção |

# **REGRAS DE NEGÓCIO**

| ID | Descrição | Versão | Data | Fonte | Status |
| --- | --- | --- | --- | --- | --- |
| RN01 | O sistema deve ser desenvolvido utilizando spring boot | 1.0 | 09/01/2024 | Documento de requisição do aplicativo | Em construção |
| RN02 | O sistema deve ser web | 1.0 | 09/01/2024 | Documento de requisição do aplicativo | Em construção |
| RN03 | O sistema deve utilizar postgresql | 1.0 | 09/01/2024 | Documento de requisição do aplicativo | Em construção |
| RN04 | O usuário pode ter até 5 jogos favoritos | 1.0 | 09/01/2024 | Documento de requisição do aplicativo | Em construção |
| RN05 | O usuário só pode adicionar ao seus jogos favoritos jogos que ele tenha cadastrado | 1.0 | 09/01/2024 | Documento de requisição do aplicativo | Em construção |
| RN06 | O sistema deve ter o e-mail como campo único. | 1.0 | 09/01/2024 | Documento de requisição do aplicativo | Em construção |
| RN07 | O sistema deve validar por meio de jwt | 1.0 | 09/01/2024 | Documento de requisição do aplicativo | Em construção |
| RN08 | O sistema deve validar se o e-mail existe | 1.0 | 09/01/2024 | Documento de requisição do aplicativo | Em construção |

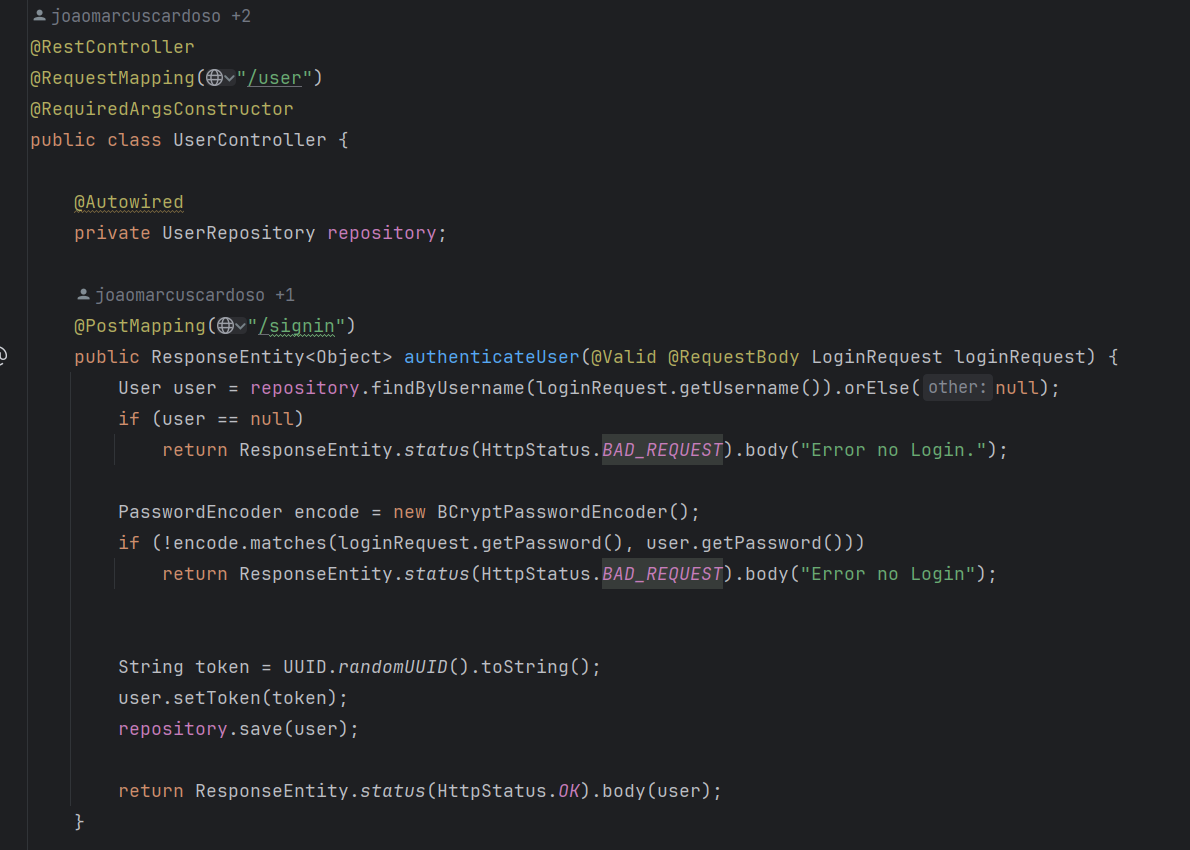
# **RELATÓRIO**

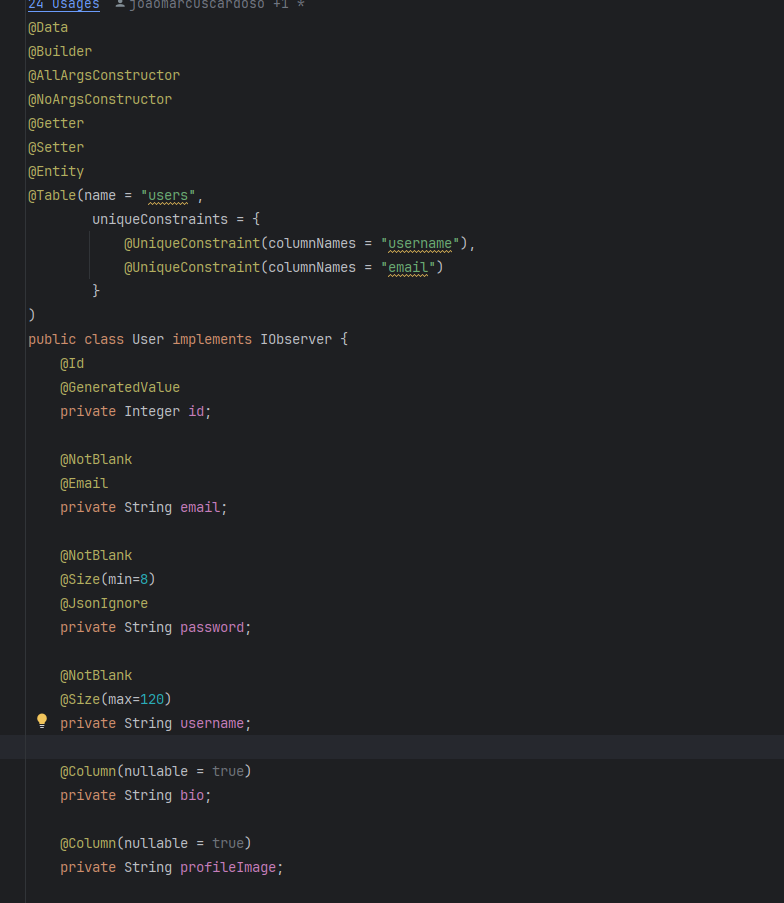
Neste trabalho aplicamos os padrões de projetos lesionados na aula, tendo como intuito entender melhor o funcionamento de cada padrão e suas respectivas estruturas, tendo isso em vista daremos uma explicação como foi aplicado e por que foi utilizado em cada módulo.

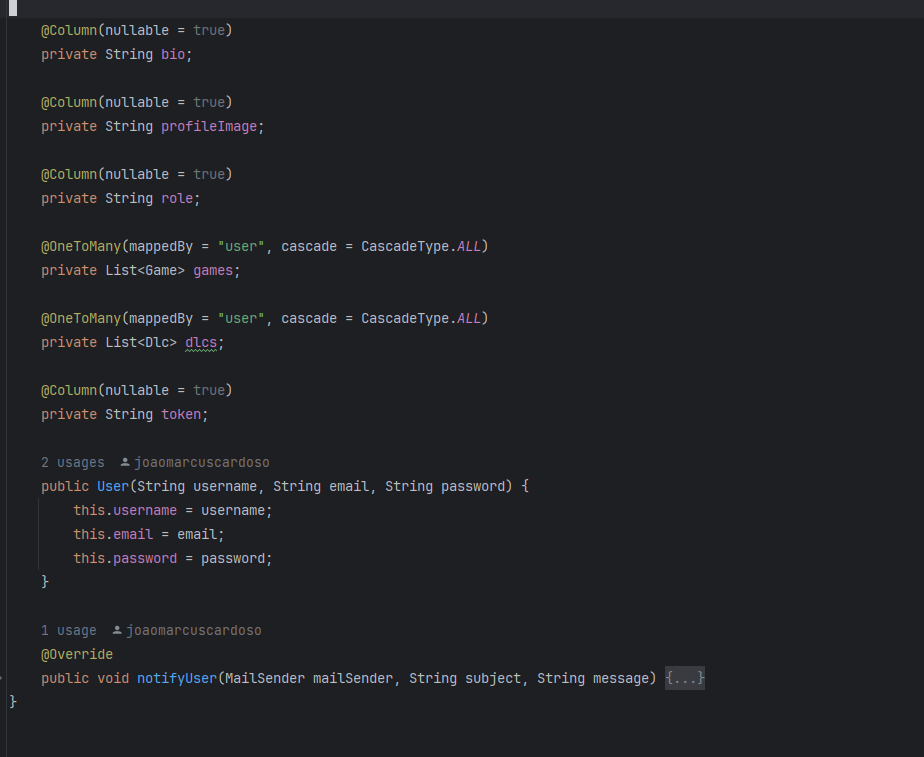
Por conta do curto tempo disponível para a execução do trabalho, focamos na parte do backend, onde aplicamos todos os padrões pedidos.

## **MVC**

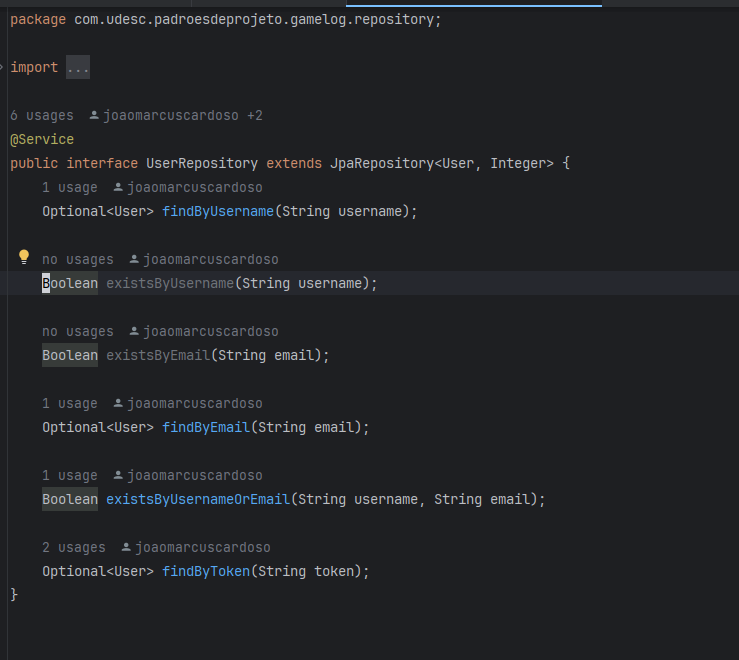
## O padrão MVC foi utilizado ao longo de todo o projeto de API, sendo assim tivemos um Model que comporta o modelo dos dados que são utilizados e salvo no banco, o Controller que recebe os dados de cada endpoint e chama o repositório para fazer a consultas no banco, a View não foi implementada pois no nosso contexto de API, não faria sentido. Temos um endpoint de criar usuário no sistema. O responsável por recebe e passar os dados seria nossa classe UserController, possuímos um método de realizar login, esse método recebe por médio da inicialização do método o DTO que é responsável por atribuir validação e tipagem para as variáveis de entrada, como por exemplo no código abaixo:

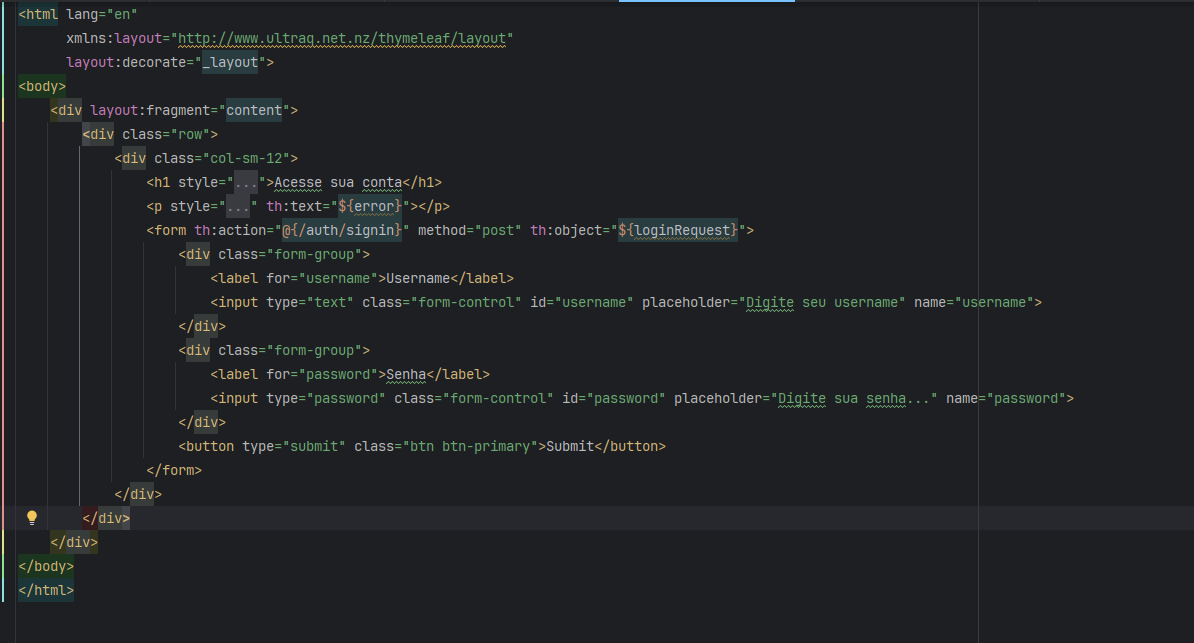
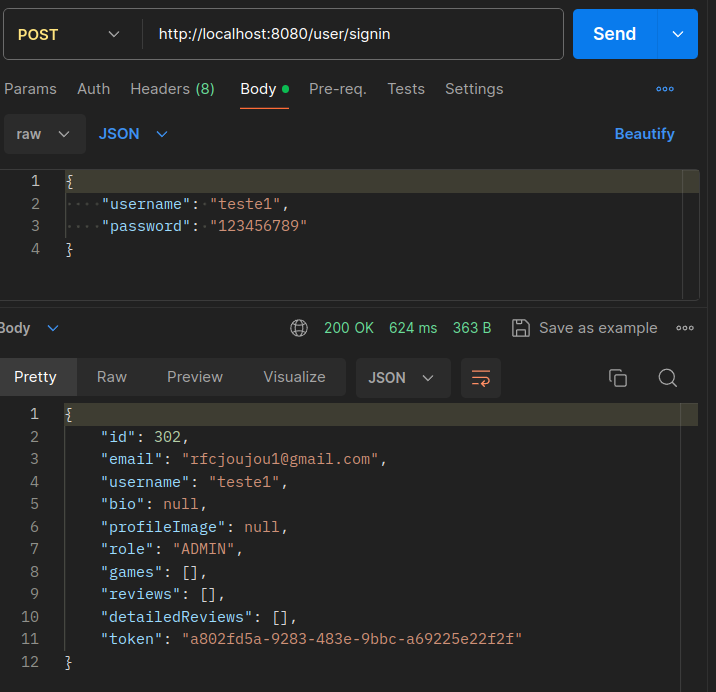


Então pegamos o username do usuário e buscamos no banco através do repositório, se não encontramos usuário, retornaremos uma mensagem de “Error no Login.”, caso encontremos usuário, iremos fazer a validação da senha e se tudo estiver de acordo, retornaremos os dados do usuário, confirmando assim a efetuação do Login do usuário. O nosso model de usuário, possui o modelo e validação conforme a necessidade do sistema:  




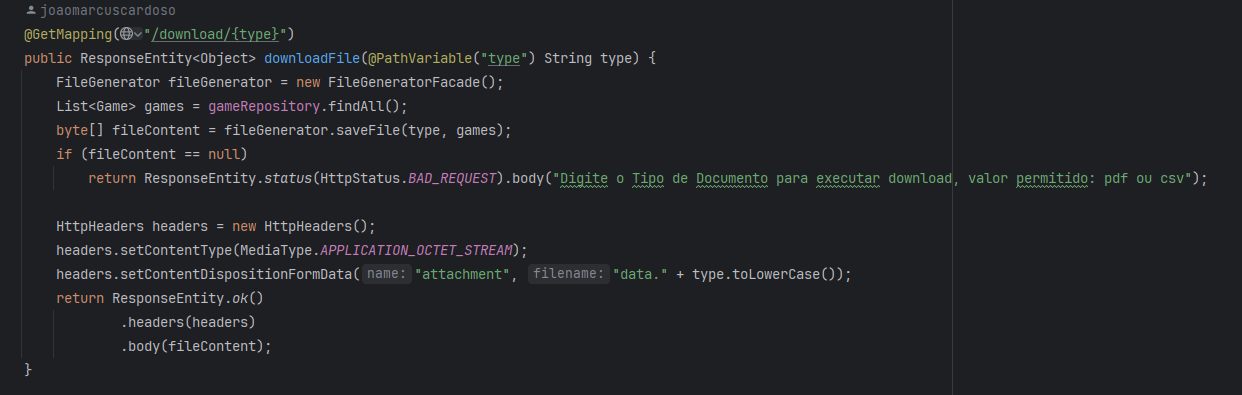
Outra classe seria o repositório que foi amplamente utiliza, ela é a classe para realizar as ações no banco de dados:



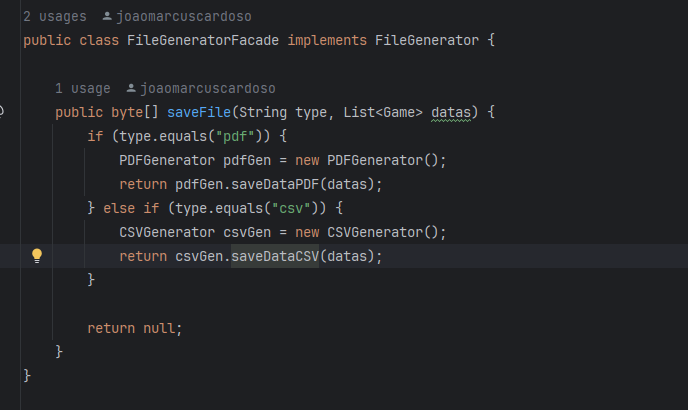
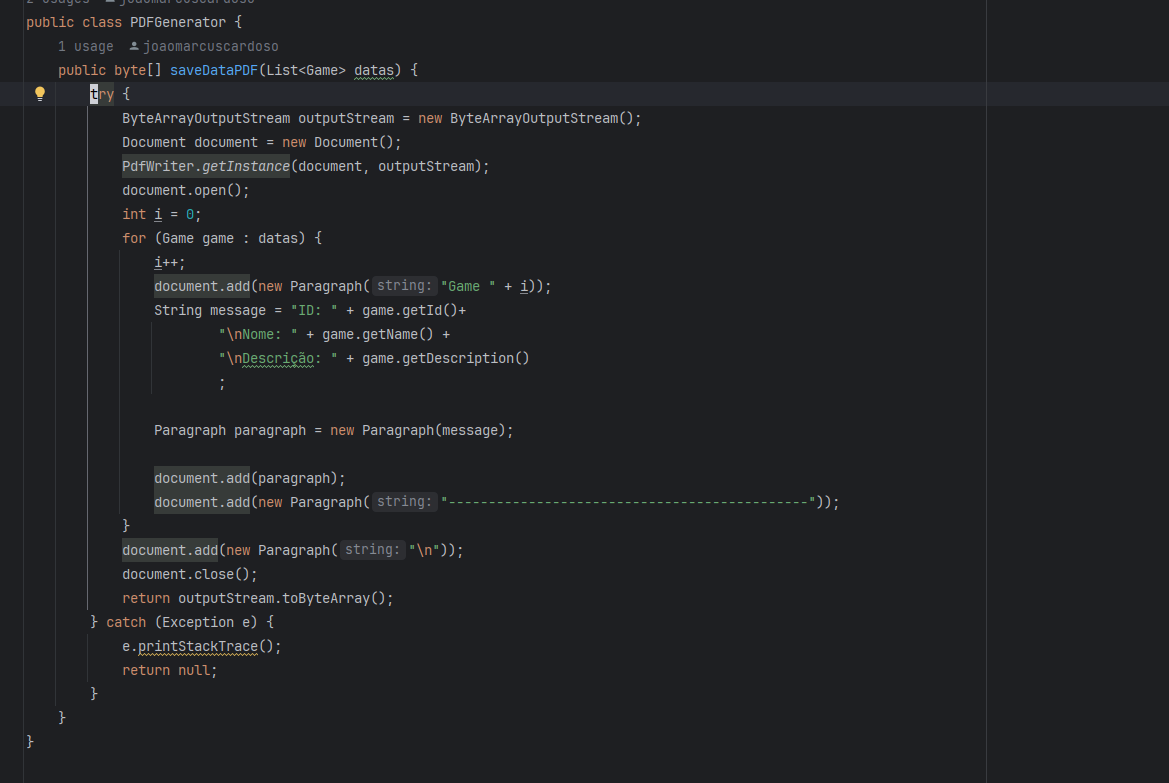
Apesar do nosso sistema ser uma API, criamos uma demonstração de um View de login usando Botstrap e Thymeleaf:  
  
  
A requisição do Login via postman:  


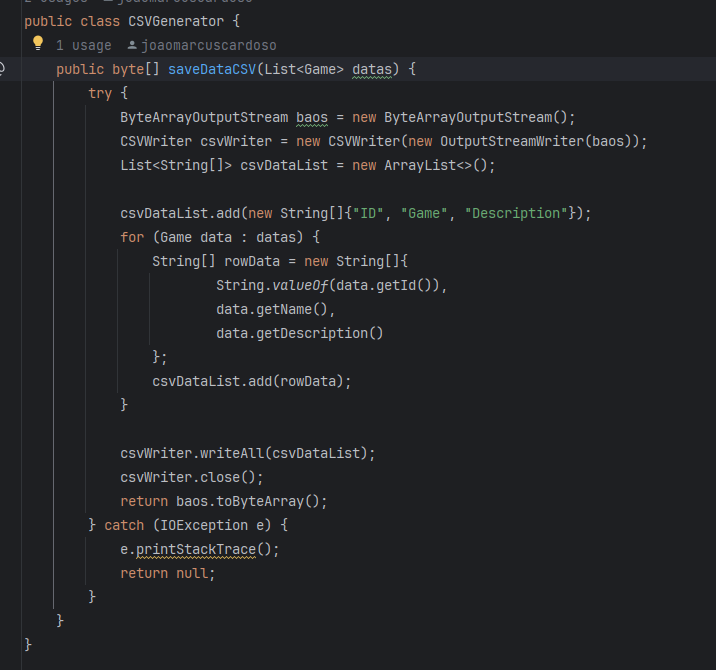
## **Facade**

O padrão Facade auxilia a implementar códigos complexos de uma maneira mais estruturada e simplificada. Sendo assim, ele facilita a chamada das ações e realizações de tarefas.

Utilizamos o Facade para realizar o Download dos jogos cadastrados pelo usuário de maneira que o sistema permite baixar diferentes formatos de arquivos(CSV e PDF).  
Temos um controller GameController, que possui um método que recebe a requisição de usuário e instância o FileGeneratorFacade, passando o tipo de arquivo solicitado pelo usuário e os dados dos jogos:  


O FileGeneratorFacade recebe a assinatura da interface FileGenerator e implementa o método saveFile, esse método tem como responsabilidade valida o tipo de arquivo digitado pelo usuário e de acordo como isso chamar o método responsável para salvar o arquivo

   
 Caso a requisição for de pdf, então instanciamos o PDFGenerator, que o processamento do dado e retornará o PDF como os dados tratados:  


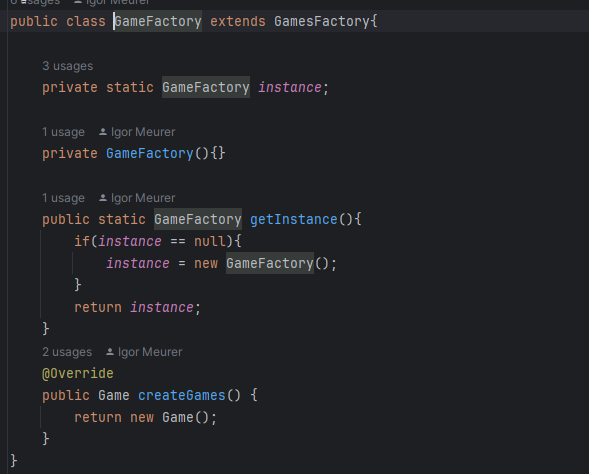
No mesmo intuito, quando o usuário solicitar na requisição csv será chamado CSVGenerator que irá tratar os dados, salvar no arquivo e retornar esse arquivo:  


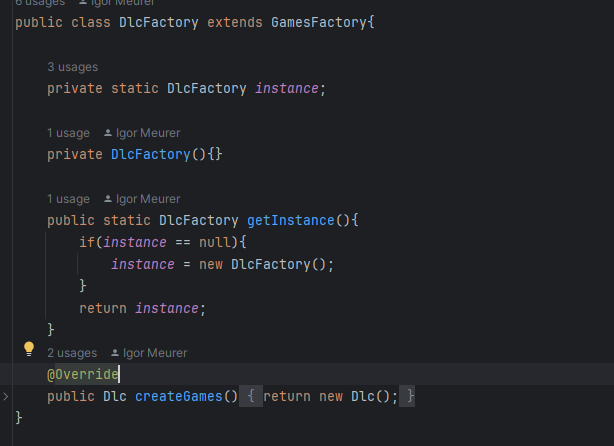
Através da estrutura do Facade será possível implementar novos tipos de arquivo de maneira facilitada, considerando que a responsabilidade de salvar os dados em um tipo de arquivo será dada a classe especializada nessa ação, com isso conseguimos fazer download para diferentes arquivos.

## **Singleton**

O padrão Singleton em primeiro momento foi pensado em ser utilizado nos Repository, porém como o spring boot gerência eles automaticamente, aplicando já o Singleton acabou sendo preciso que a gente implementasse o padrão em outras áreas do código.

Então o padrão foi implementado em conjunto com outros 2 padrões, o Factory e no command, no factory foi implementado em ambas as factories feitas para Game e Dlc

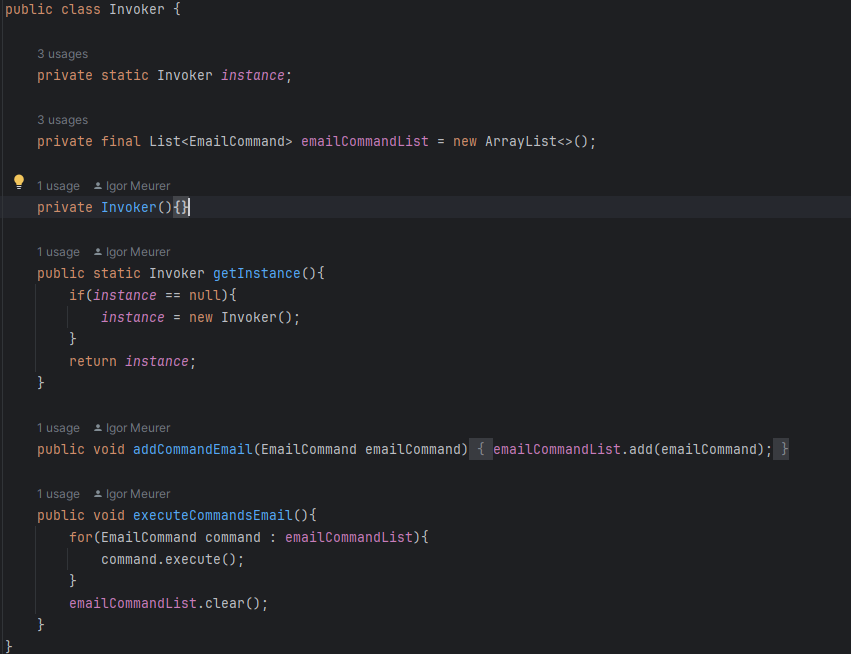




Que por conta de ambas serem apenas chamadas na hora de cadastrar tanto Game quanto Dlc, faz sentido que chamemos sempre a mesma instância, pois utilizamos ambos apenas para nos retornar um new Game ou dlc, como é mostrado no Controller a onde chamamos a instância

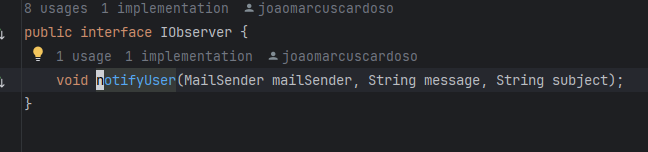


Outra classe que foi utilizada foi no Invoker de commands

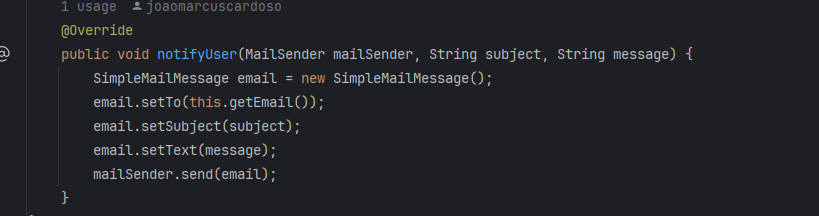


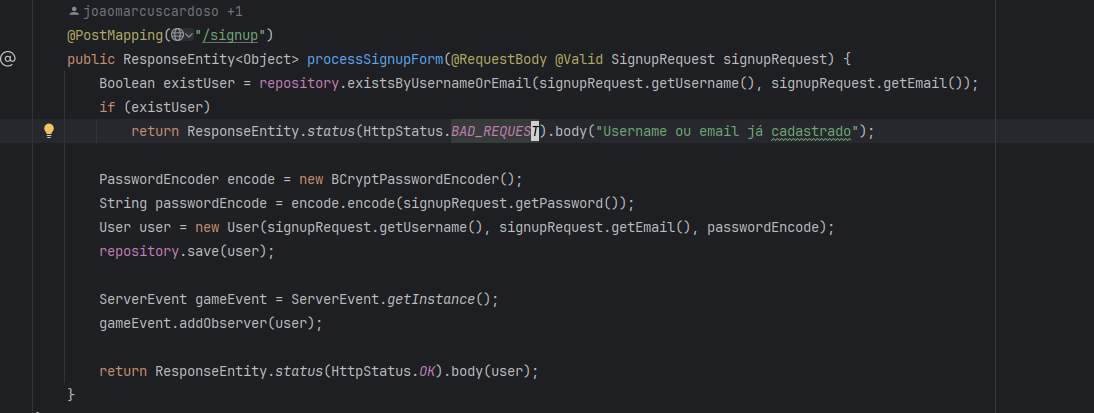
O qual pelo mesmo motivos dos factory foi aplicado o Singleton, pois ele seria utilizado apenas para armazenar os commands que seriam utilizados e logo em seguida executar os mesmos, e como ele limpa a lista de commands a serem executados logo após executar todos, se tornava interessante utilizar do Singleton nele também.

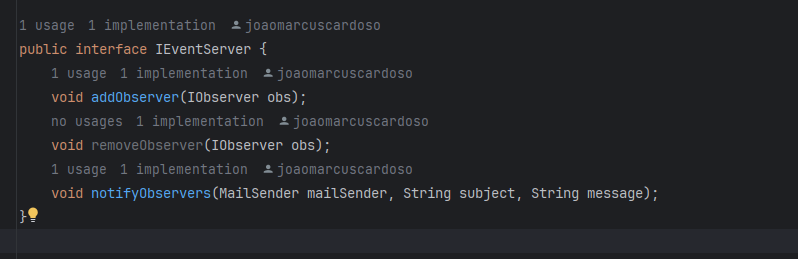
## **Observer**

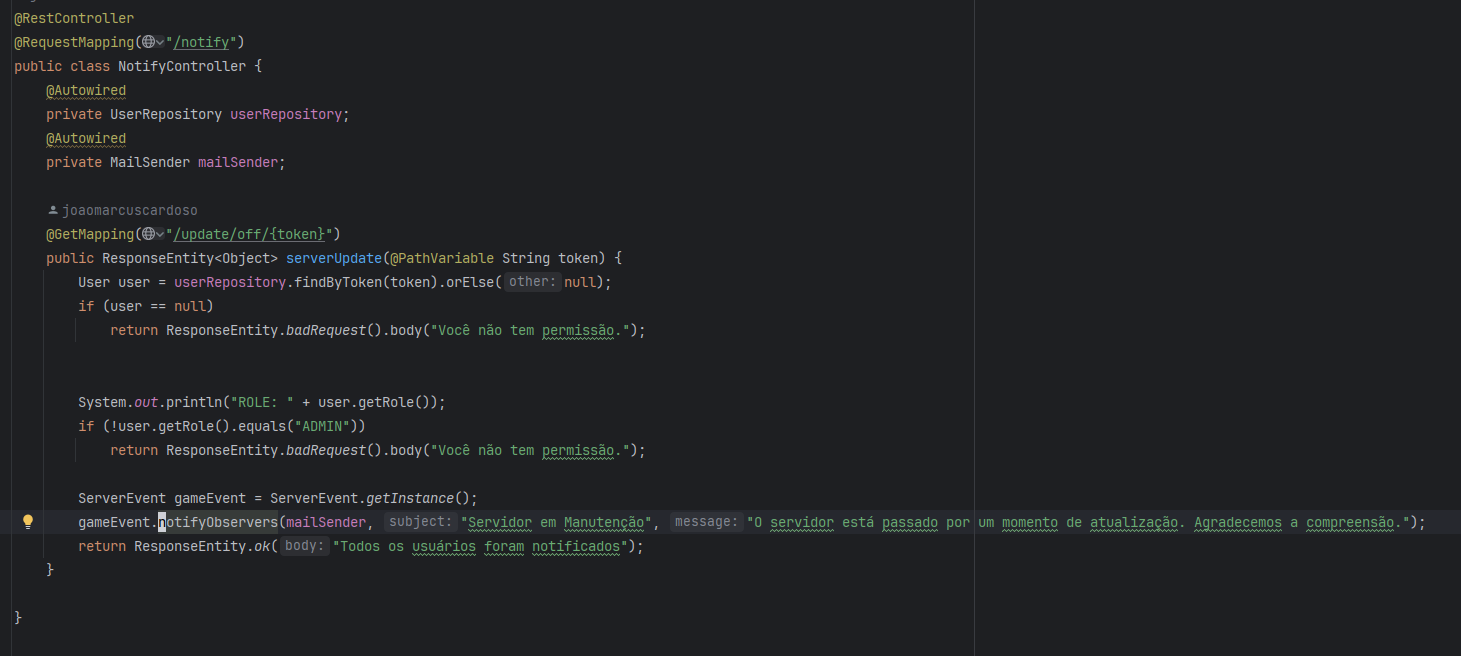
Utilizamos o Observer para enviar mensagem ao usuário cadastrado na nossa base de dados quando o servidor entrar em manutenção. Quando o usuário realiza o cadastro no sistema efetuamos uma inserção de usuário como Observado, sendo assim quando um Admin fizer uma requisição para o endpoint de manutenção iremos realizar um envio de e-mail a fim de notificar o usuário sobre a manutenção:  
Temos um IObserver que seria uma interface para a assinatura dos módulos a serem o observados:  


No caso temos no Model de User a implementação para mandar mensagem quando a função notifyUser for acionada:

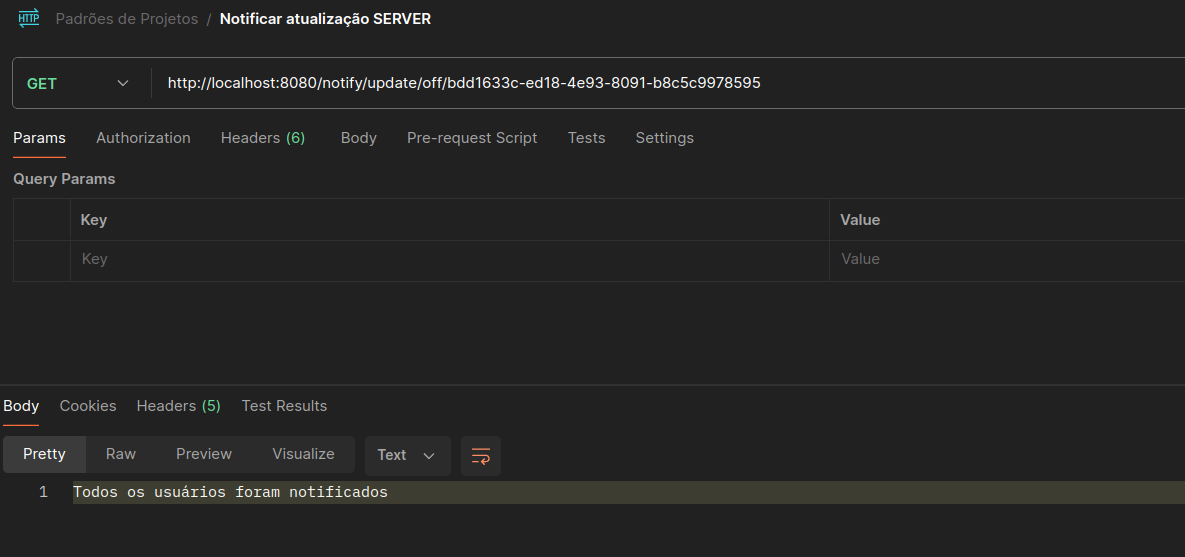


No método de signup, recebemos os dados para cadastro do usuário e adicionamos o usuário a lista de observers:  


Temo a interface IEventServer que seria a interface que possui as assinaturas do método:  


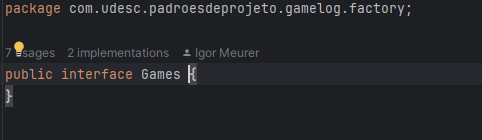
Por fim temos um Controller que receberá as chamadas das requisições do administrador e realizará a implementação de enviar notificação para cada Observer instanciado:  


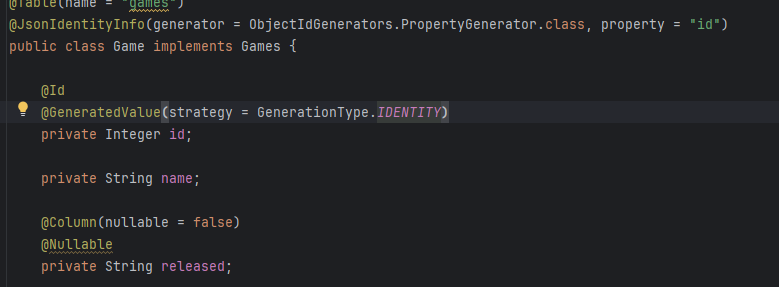
Resultado da requisição via postman:

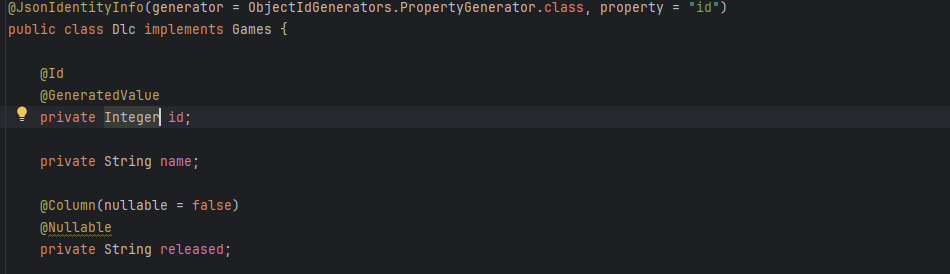


## **Factory Method**

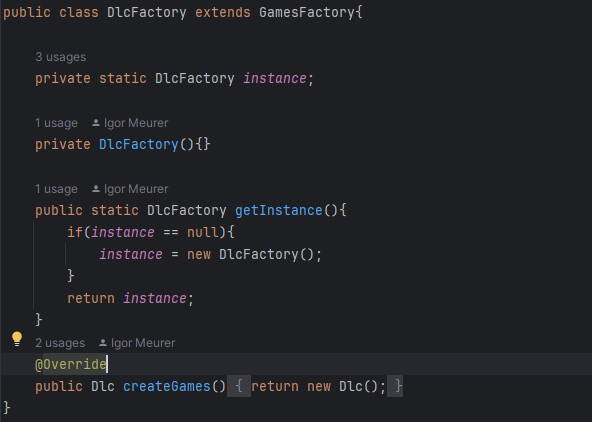
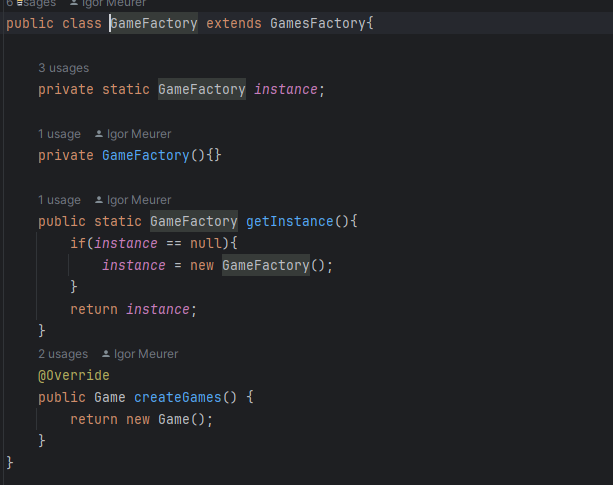
O Factory method como dito anteriormente foi utilizado para a criação de Game e Dlc, pois ambos são semelhantes o bastante para terem uma interface em comum, e caso fosse necessário aplicar um método a ambos em um futuro poderíamos apenas adicionar no interface ao invés de duplicar o código, no momento a interface não possui nenhum pois não se provou necessário até o momento trabalhando apenas no backend.







Então temos os Factorys os quais são utilizados apenas para retornar novos Games e Dlcs, por conta de que deixamos a criação de ambos o mais simples possível, mas caso fosse de nossa vontade no futuro modificar a criação de ambos para agora passar todos os campos no construtor seria possível. E caso a gente quisesse adicionar uma nova etapa na criação de dlc na factory para que ela desse set no game na própria factory seria possível também adicionar um novo método na factory da mesma sem que fosse preciso alterar qualquer coisa na factory de Game.



Como utilizamos Singleton nas factory, quando a chamamos no Controller é sempre a mesma instance de factory para ambas, e como utilizamos as factory para apenas retornar um novo Game ou dlc, ambas são chamadas apenas para que possamos ter o objeto e assim settar nele todas as informações que foram enviadas pelo usuário e depois salvar o mesmo no banco de dados.

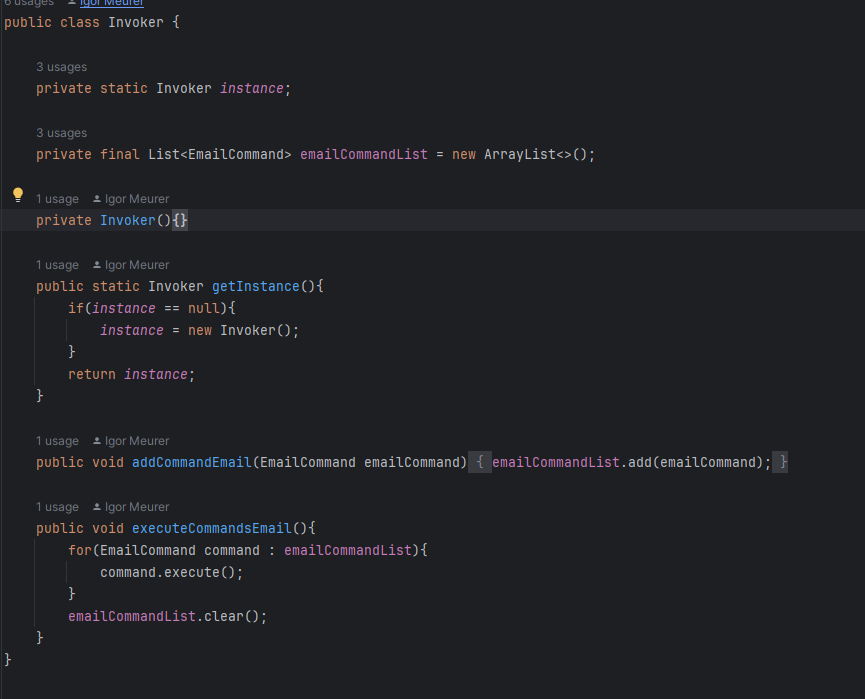




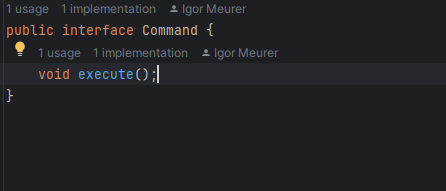
## **Command**

O command foi utilizado para a nossa função de enviar emails, o local onde foi aplicado foi na criação de Games, onde toda vez que um game fosse criado e adicionado a biblioteca de um usuário o sistema pegaria qual game foi cadastrado e enviaria um email para o usuário o avisando que o game foi criado com sucesso e o lembrando de avaliar o mesmo.

Então foi criado uma classe de Invoker o qual armazena os commands em uma lista para o caso de querer executar diversos comandos em sequência, como por exemplo em um futuro adicionar a opção de cadastrar diversos jogos, mas não queremos mandar todos no mesmo email e sim em e mails separados ou talvez logo depois de enviar o email poderíamos adicionar um command que faz aparecer um modal com as informações do Game cadastrado aparecer na tela do usuário.



Juntamente foi criado a interface para os commands o qual tem apenas o execute que todos os commands terão e que é chamado no invoker enquanto o mesmo varre a lista.



Depois temos a classe de EmailCommand o qual implementa a interface, nele possuímos a classe que efetivamente realiza o envio de emails que é o JavaMailSenderService, que é instanciado quando se cria o command. O execute da classe cria o corpo do email que se consiste dele pegar o nome do usuário e o nome do jogo que foi criado e colocar ambos em uma String e depois enviar para o service de envio de email o email do usuário, juntamente do Assunto do email e do corpo do email.

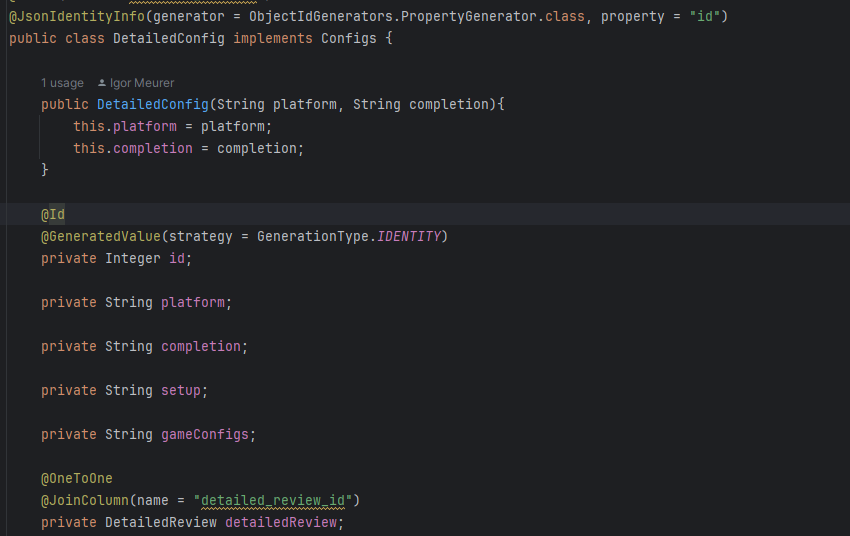
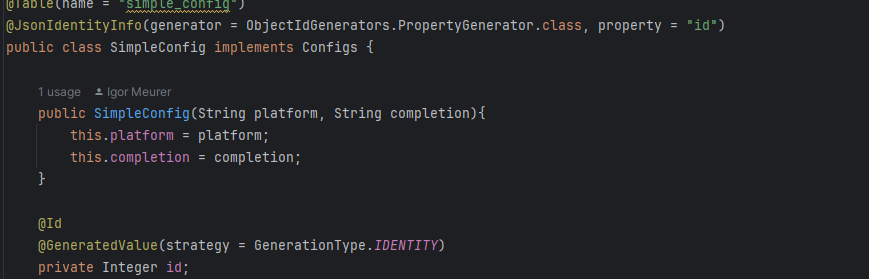
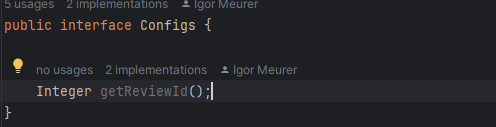
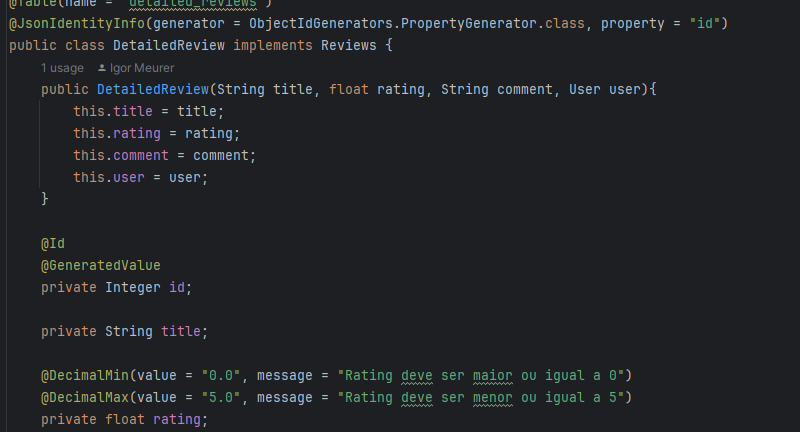
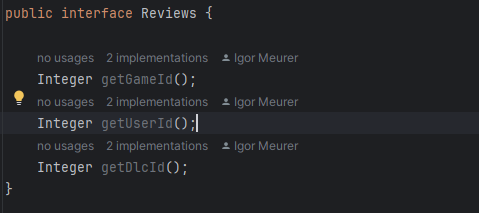


Para o funcionamento do command então foi feito as classe de JavaMailSenderService o qual tem seu método chamado no objeto EmailCommand, e também foi criado o JavaMailConfiguration o qual configura o envio de email para o Spring boot. Então quando o EmailCommand chama o método passando tudo que é necessário o JavaMailSenderService apenas setta tudo que é necessário para que a biblioteca de envio de emails do java precisa para funcionar.

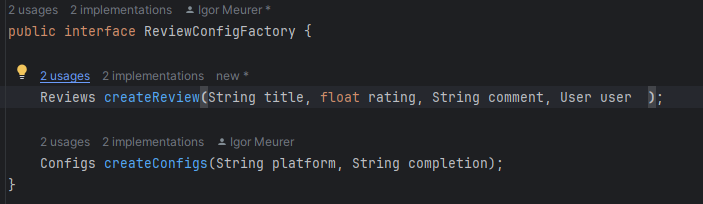


## **Abstract Factory**

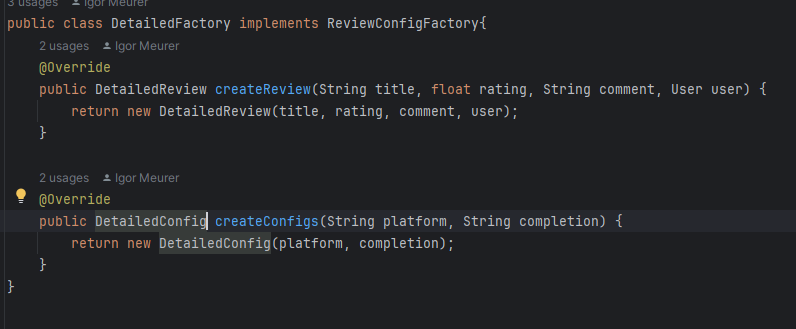
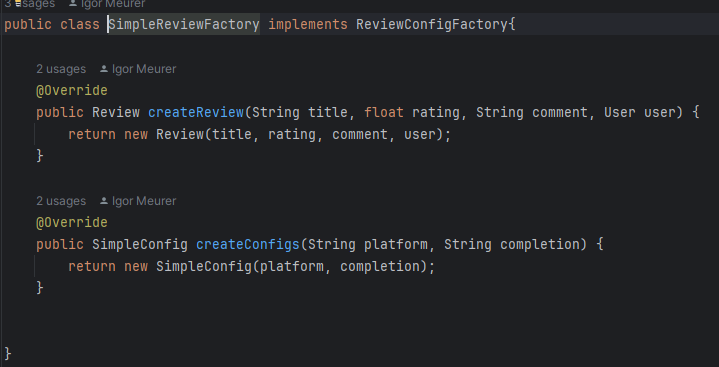
O abstract factory foi aplicado nas classes de review do sistema, pois decidimos adicionar configs a review, e uma opção de como seria a review do usuário, podendo ela ser uma review simples ou uma review detalhada, então as classes foram divididas entre review, simpleConfig onde ambos vão ser criadas pela mesma factory e detailed review e detailed configs a onde ambos serem criados pela mesma factory tbm. As reviews possuem a mesma interface de Reviews e as configs a interface de Configs, também foi criado repositorios para todas essas classes, mas todas são mt simples já que o spring gerência elas automaticamente e os métodos como save e findById já vem junto.



Então possuímos as factorys, temos ReviewConfigsFactory que é a interface que as factorys concretas implementam



A factory SimpleReviewFacty é a que cria a review e simpleConfig, dessa vez nessas factorys estamos passando parâmetros para os construtores, já na factory DetailedFactory criamos a DetailedReview e DetailedConfig, na DetailedConfig não se passa todas os parâmetros que a classe possui, pois por conta do método ser implementado da interface seria preciso passar colocar parâmetros que o simpleConfig não possui então foi deixado apenas os que ambos possuem e no detailed é settado durante o controller os que faltam

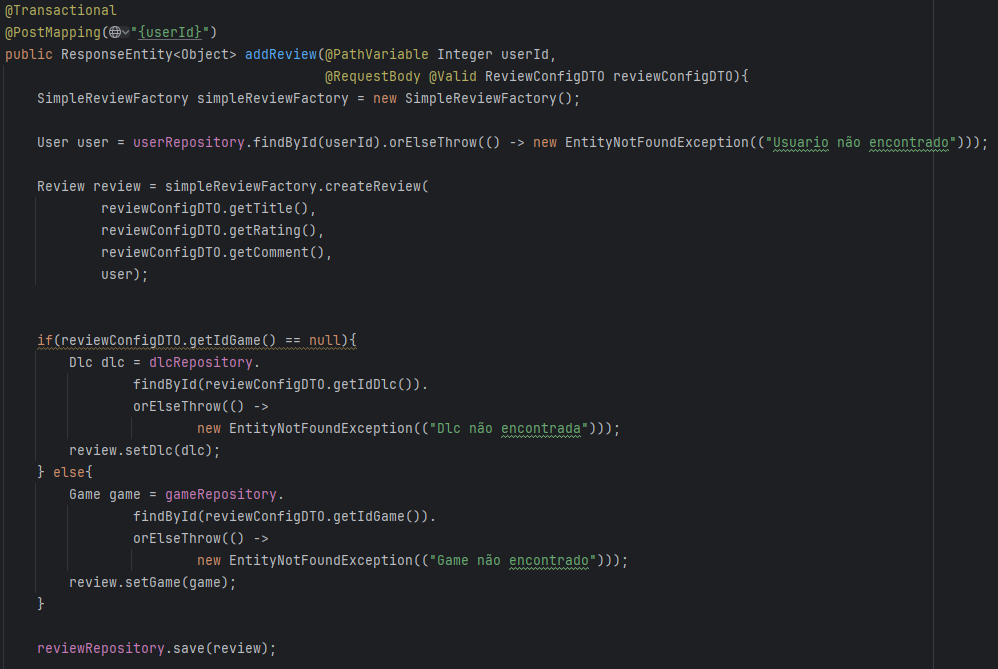


Por fim temos o ReviewController o qual tem dois métodos de post feitos para criar a review junto da config respectiva, como uma review simples vai criar junto a simple config, de início damos autowired em todos os repositórios que vamos utilizar.

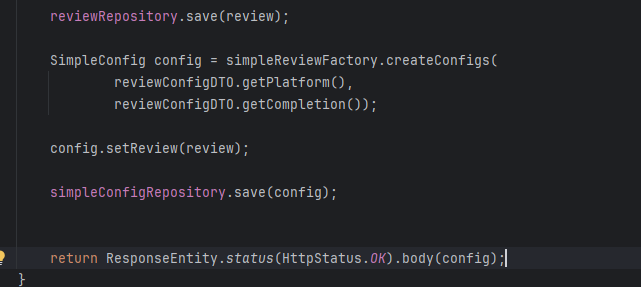


Depois possuímos os dois métodos, que são bem parecidos porém um é para simple review e outro para detailed review, começando pelo simple, mesmo valendo para o detailed, de início recebemos um request com o id do usuário e o requestBody, o qual utilizando um Dto, ReviewConfigDTO, criado para receber as variáveis tanto de review quanto de config, também é usado para validar os campos obrigatórios.

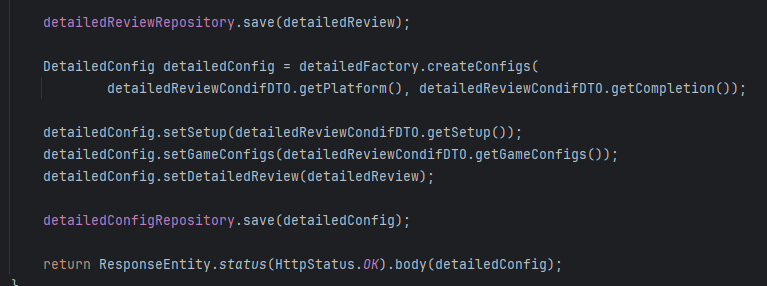
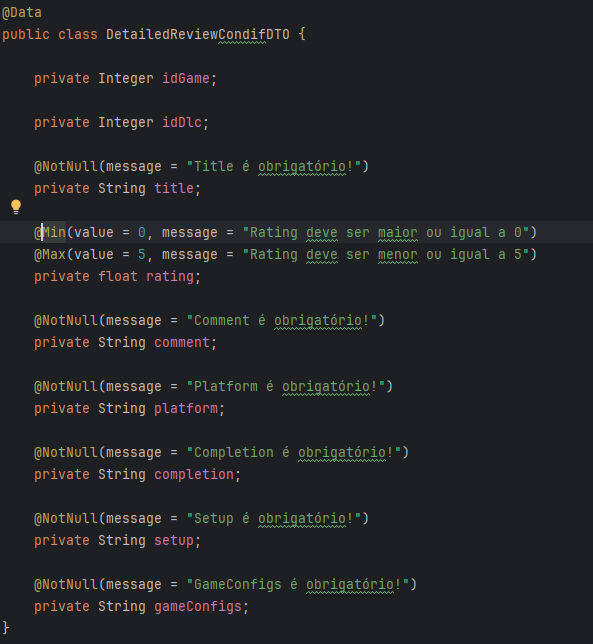
Depois criamos a simple review factory e a utilizamos para criar a review, depois verificamos se é a review está sendo feito para um game ou dlc, depois pegamos o jogo especifico pelo id e damos set na review, depois o salvamos no banco



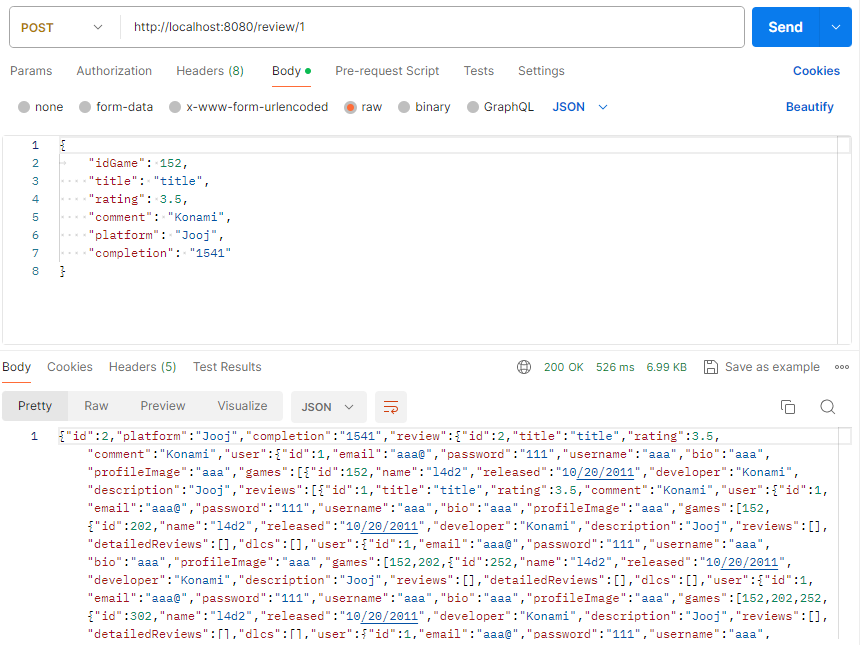
Depois a mesma factory é utilizada para criar criar a config, finalizando com o set da review que foi previamente criada no método e depois a config também é salva no banco de dados.



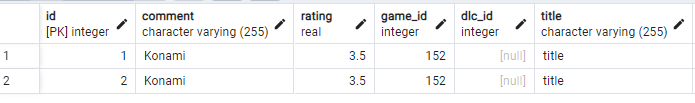
Quanto ao Detailed review a sequencia de como é feito é praticamente igual, a diferença é que ao inves da simple review factory é utilizada a Detailed Factory, e na parte da criação da detailedConfig também é dado set nas variáveis que não foram passadas no construtor, também foi criado um Dto para o detailed review o DetailedReviewCondifDTO.



Por fim temos um exemplo de post utilizado para testar o método



Select das tabelas de review e simple\_config





# 

# **REFERÊNCIAS**

<https://github.com/joaomarcuscardoso/gamelog>

<https://refactoring.guru/design-patterns/facade>

<https://refactoring.guru/design-patterns/observer>

<https://refactoring.guru/design-patterns/singleton>

<https://refactoring.guru/design-patterns/command>

<https://refactoring.guru/design-patterns/factory-method>

<https://refactoring.guru/design-patterns/abstract-factory>