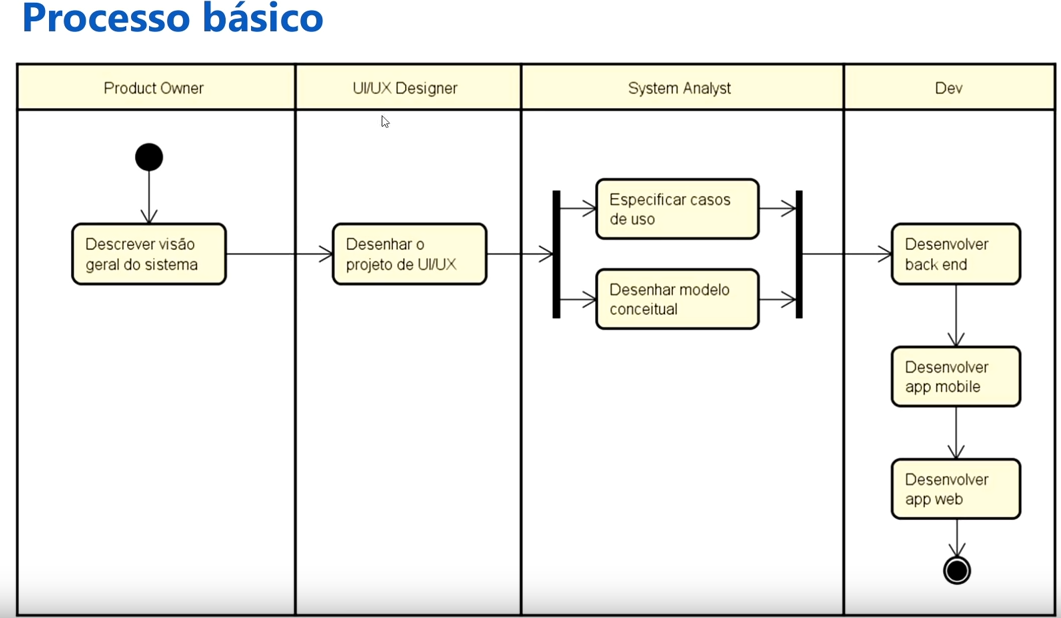
16/01 16:16

Parado vídeo 1 sds 08:00

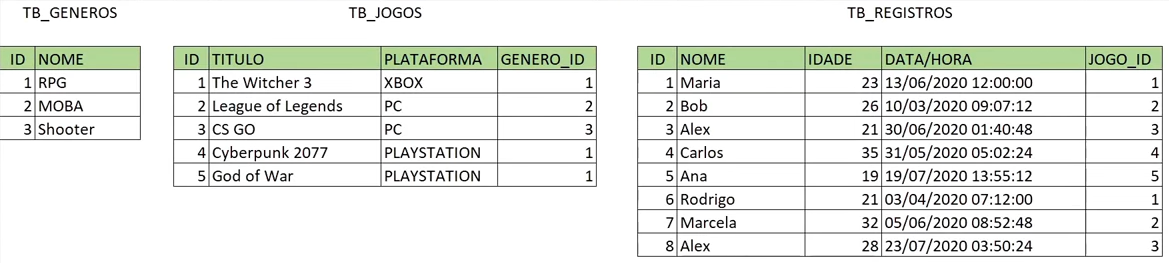
Começar vídeo de 33min. Feita instalação das ferramentas.

17/01 14:10 começando vídeo de apresentação e análise de aplicativo:



Wireframe = mockup do sistema

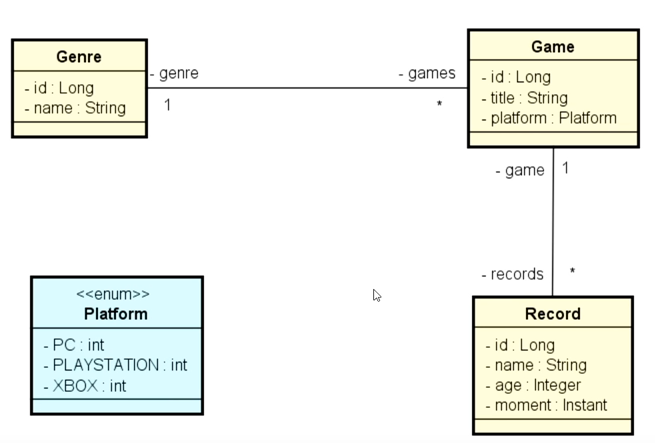
Estruturação do banco de dados:



Esta é forma normalizada, utilizando chave primária e estrangeira. Senão fica tudo misturado em uma tabela com repetições de dados (desnormalizados)

Agora é desenhar o modelo de classes pela UML para depois fazer a parte de banco de dados, pois será utilizada UML para o desenho do banco.

Na UML, plataforma (PS, PC, XBOX) nunca mudará, portanto, deve ser tratado como uma Enumeração para o Java, e não uma tabela.



19/01 20:30 Aula Back-end

Tomcat é o container Web, já vem padronizado com a porta 8080



***Obs: NPM é o gerenciador de dependência do Front e o Maven é do Back.***

Pom.xml: arquivo com a informações do Maven do projeto.

Banco H2 é o banco em memória, Banco Postgresql vai no Heroku na publicação.

Feita classe Gênero: parado aos 32:58.

27/01/2021

Implementação dos ambientes: dev, teste e Prod.

Teste: ambiente instancia BD vazio para teste.

Dev: BD igual do Prod (PostgreSQL).

Prod: rodar na nuvem.

Pegar os documentos de configuração (properties) no github (arquivos de config).

O que significa isso?

spring.profiles.active=test

spring.jpa.open-in-view=false

A segunda linha: quando estiver entregando as respostas (mostrar em tela), o JPA fecha conexão e encerra o monitoramento. Ou seja, não monitorar as Entidades quando estiver entregando as respostas. É para resolver tudo de banco de dados na camada de Serviço.

Para o application.properties de PROD, colocaremos informações pertinentes para rodar em nuvem no Heroku.

Parei as 01:04

Criando o repositor, camada de acesso a banco de dados.

Get/games: output é receber os nomes dos jogos, uma lista. Ter rota de API Rest, retornando os games (foi tirado do projeto UI e casos de uso – vídeo 1).

Entidades: ok.

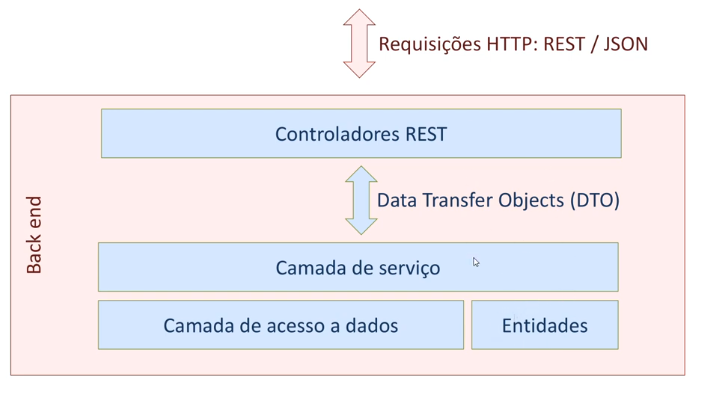
Camada de acesso a dados: ok.

Falta controladores REST e Serviço.

Fazendo agora os Resources, nada mais são do que os Controlers...podendo também ter este nosso de pacote.

Parei 1:26

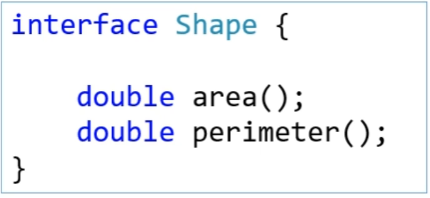
DTO: objetos auxiliares para tráfego de dados. Não será usado o Entities para o Resources: conforme explicação



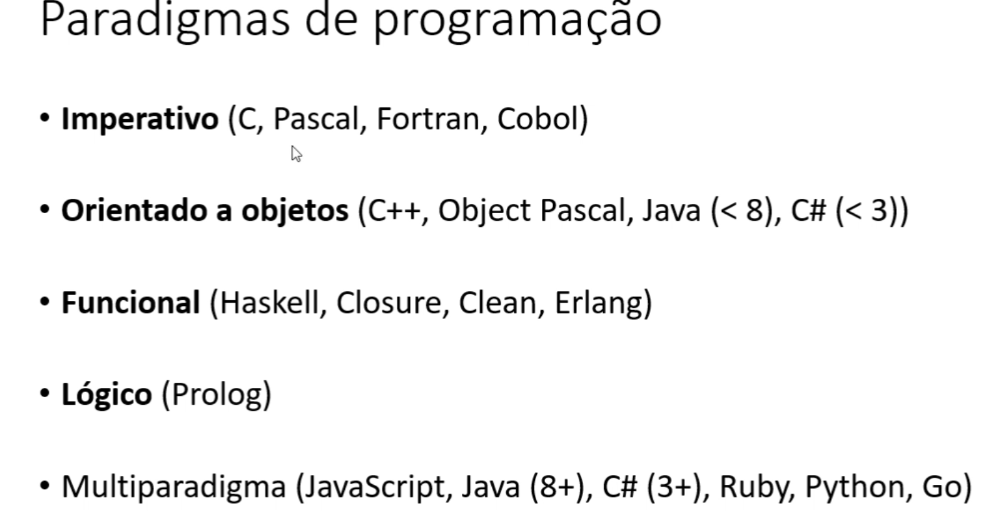
A camada de serviço conversa com o acesso de dados e as Entidades, fazendo assim a transferência com DTO.

Estamos implementando a camada de serviço e DTO.

Interfaces: como se fosse uma classe abstrata com métodos abstratos! Funciona como um contrato.



Quando criamos um triângulo e implementamos Shape, ela DEVE ter área e perímetro.

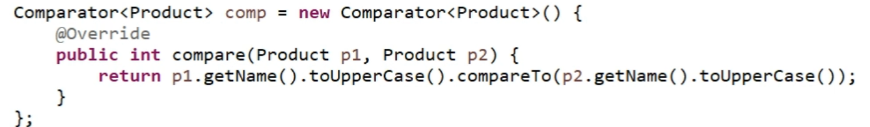


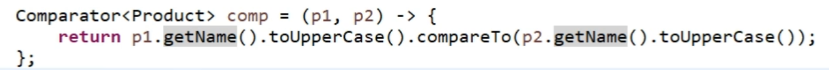
Linguagem multiparadigma implementa o funcional e o O.O.

Interface funcional: tem apenas um método, podendo ter default methods ou static, mas o que tem que implementar é apenas um!

Lambda: funções anônimas. Sempre fica do lado esquerdo os parâmetros, flecha e dentro da função é a sua implementação. Ela nada mais é que a arrow function (visto no JavaScript)

Ex: Sem arrow function – expressão lambda



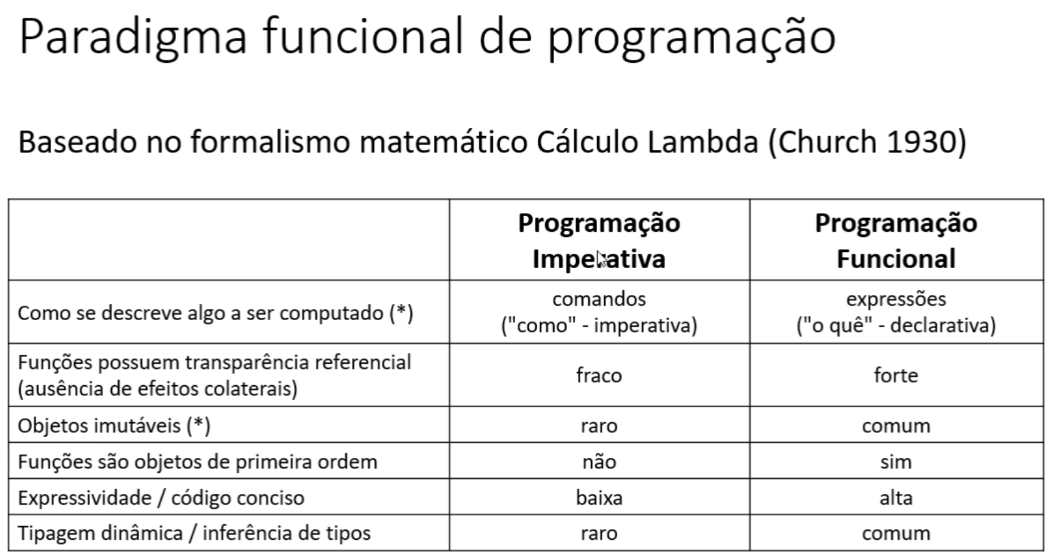


Limpando mais ainda....



Para deixar totalmente limpo, invés de usar o objeto comp, colocar tudo dentro de onde irá utilizá-lo como argumento:





Parei 1:38

Criando endpoint Post/records.

Criando RecordDTO para retornar na API, sabendo se está vindo as inserções feitas.

Rever algo...parei 1:49

Setando o Game no RecordDTO insert(RecordInsertDTO dto):

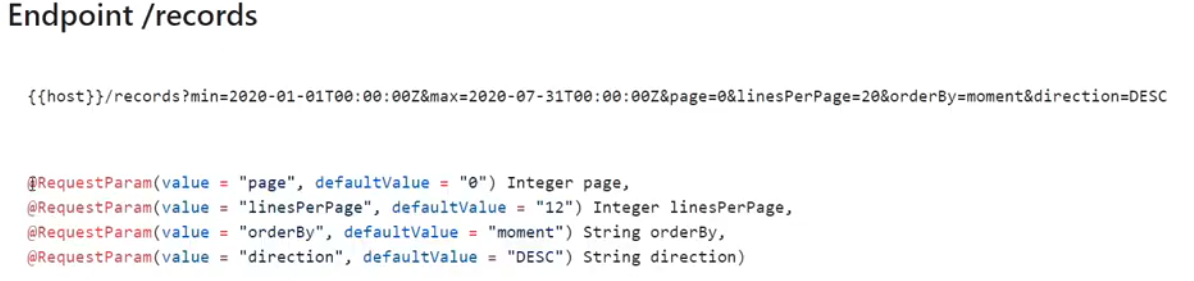
Estamos dentro de uma transação controlada pelo JPA, é preciso pegar uma referência para o Game cujo id seja desse dto do método que chegou pra ele. No RecordInsertDTO, chega pra ele um Long gameId, é necessário instanciá-lo, e como estamos a nível de repository precisamos passar o game todo pra ele. Portanto foi feito um autowired do gameRepository e instanciado para pegar o dto.getGameId().

getOne() do JPA: ele ainda não vai no BD, só instancia o objeto monitorado para você associar, ai salvando sim ele vai no BD.

Parei 1:56

Momento de fazer uma paginação com o Springboot no Get Records.

Como é demorado pra explicar, professor deixou pronto no github



Fazendo paginação no RecordResouce - PageRequest.

REPOSITORY CONHECE ENTIDADE!!!

Utilizado o @Query no RecordRepository (adaptação de uma Query pro JPA)

Parei 2:12

Para continuar a paginação foi criado um endpoint pelo professor:

<http://localhost:8080/records?min=2020-01-01T00:00:00Z&max=2020-07-31T00:00:00Z&page=0&linesPerPage=20&orderBy=moment&direction=DESC>

Momento de rodar o env dev com PostgreSQL.

Mudamos o aplication.properties para dev e assim quando demos um run Springboot app ele criou um arquivo “create.sql” a partir das 4 linhas que estavam no properties dev.

Ele nada mais é que os dados do banco feito já no jeito para PostgreSQL.

Pego todos os dados e colocados no PostgreSQL e não é mais banco de memória!!

Após os testes serem feitos é apenas mudar para PROD conforme arquivo Neli.

No Heroku ele gera uma variável de ambiente:

postgres://idscirvsvccckp:bb01b97eb09bf9240ee16fcc7cf1ce385adcd1d967714acbf946f7803449477f@ec2-34-195-233-155.compute-1.amazonaws.com:5432/d12bm7fuk6jtpq

postgres://

user: idscirvsvccckp

:

Senha: bb01b97eb09bf9240ee16fcc7cf1ce385adcd1d967714acbf946f7803449477f

@

Servidor: ec2-34-195-233-155.compute-1.amazonaws.com

Porta:5432

Base de dados: d12bm7fuk6jtpq

03/04 Deploy feito com sucesso no Heroku

Erro que aconteceu: a pasta backend não é a principal do repositório do git, por isso não conseguia pegar a referência de “language”. Criado um novo repo para dar o deploy pro Heroku.

Feita as mudanças de “Environment” no Postman e agora está funcionando tudo normalmente puxando do servidor do Heroku.

Agora testar o Cors (mecanismo de proteção para quando estamos tentando acessar nossa app de outro domínio). No git do professor tem o teste.