Programação assíncrona com CompletableFuture

Emerson Venâncio @ Dígitro Tecnologia

Filipi da Silva Fuchter @ Dígitro Tecnologia

Digitro 40 anos

PROGRAMAÇÃO ASSÍNCRONA

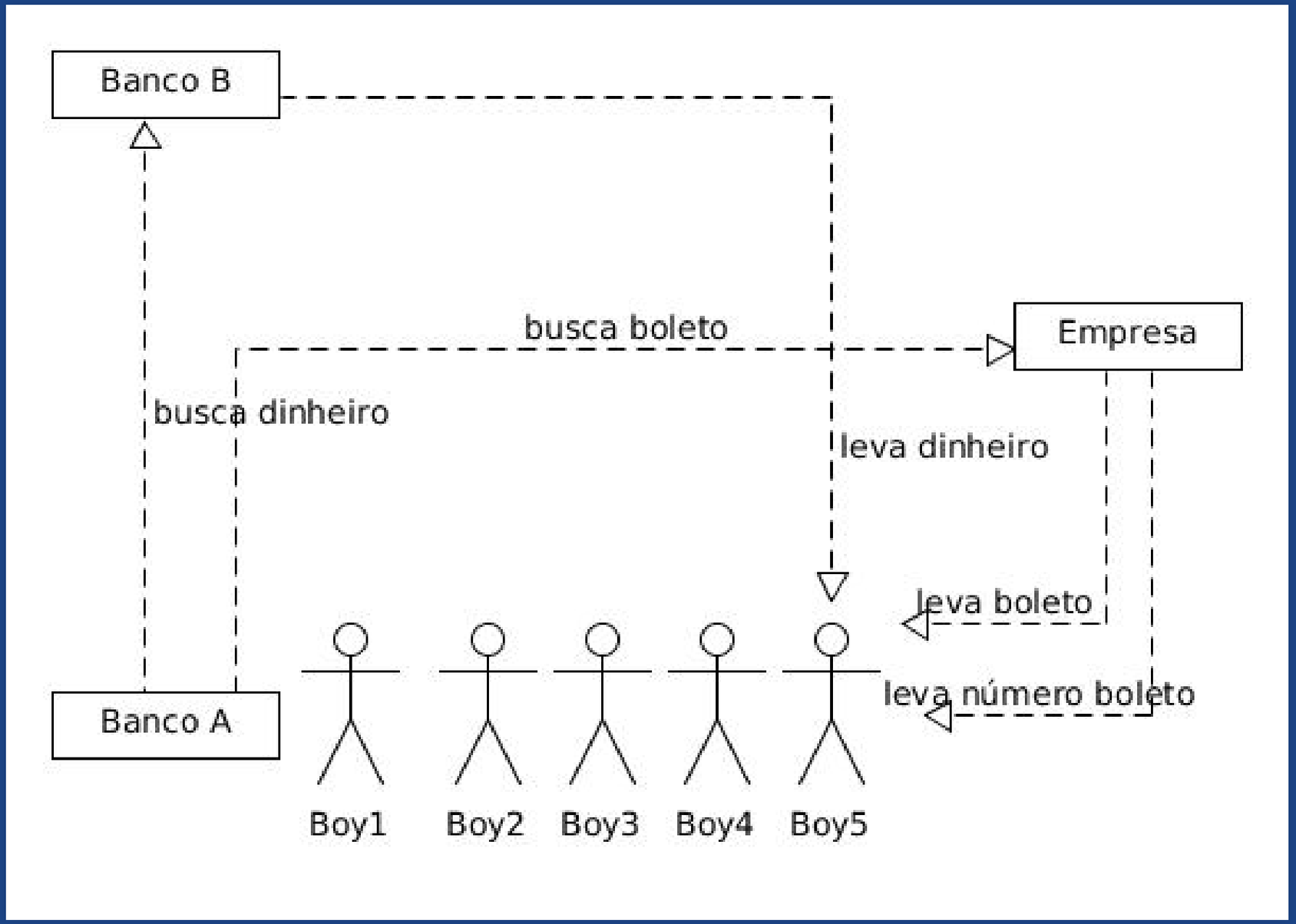
POR QUE?

Programação assíncrona deve ser usada para otimizar o uso de recursos do hardware

- *Adicionar mais threads não traz performance em todos os casos*
- *Quando o número de threads excede o número de processadores o efeito é inverso*


OFFICE BOY
Banco
Cartório


zzzz

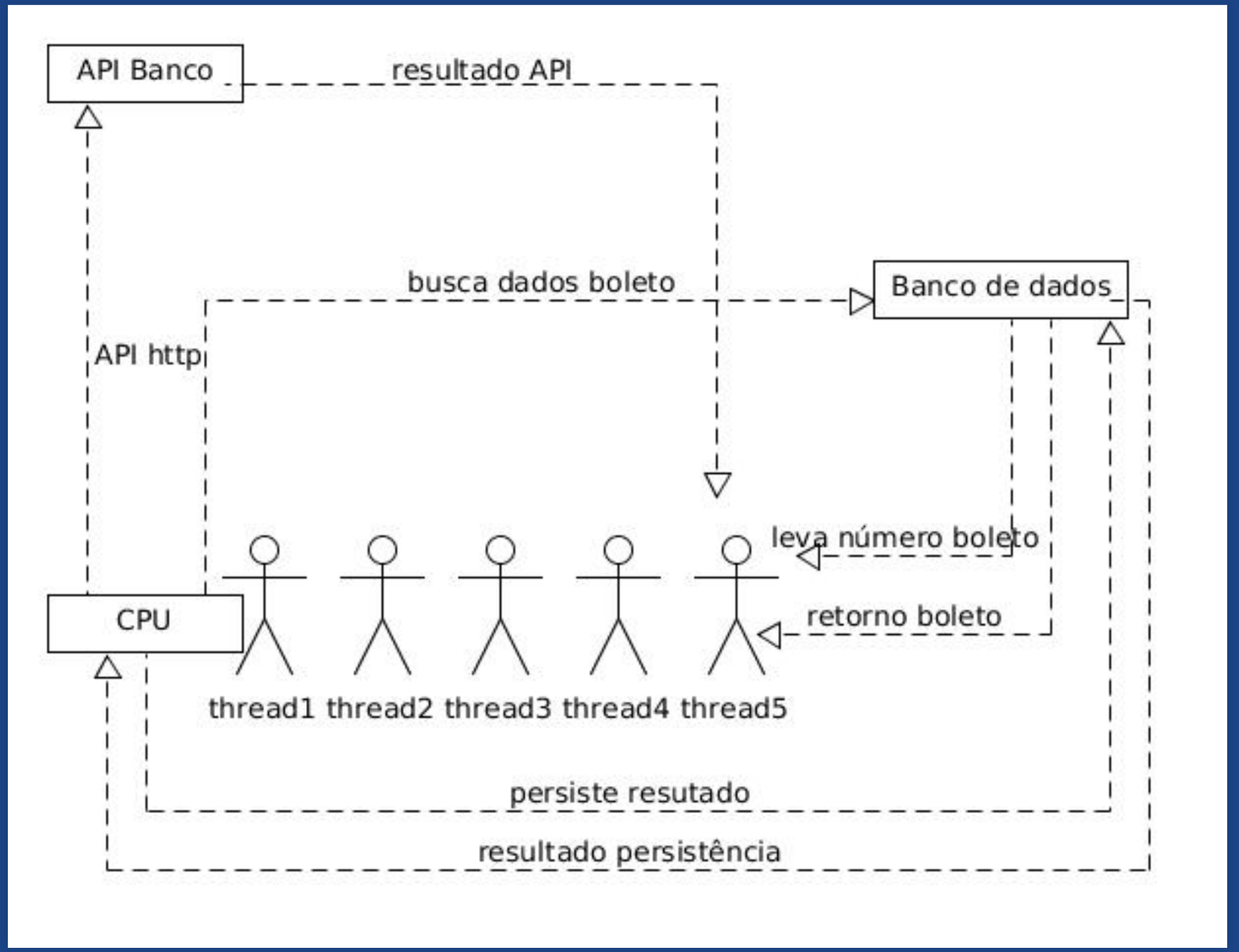

Banco B
busca boleto
Empresa
busca dinheiro
leva dinheiro
leva boleto
leva número boleto
Banco A
Boy1
Boy2
Boy3
Boy4
Boy5

Sistema OfficeBoy

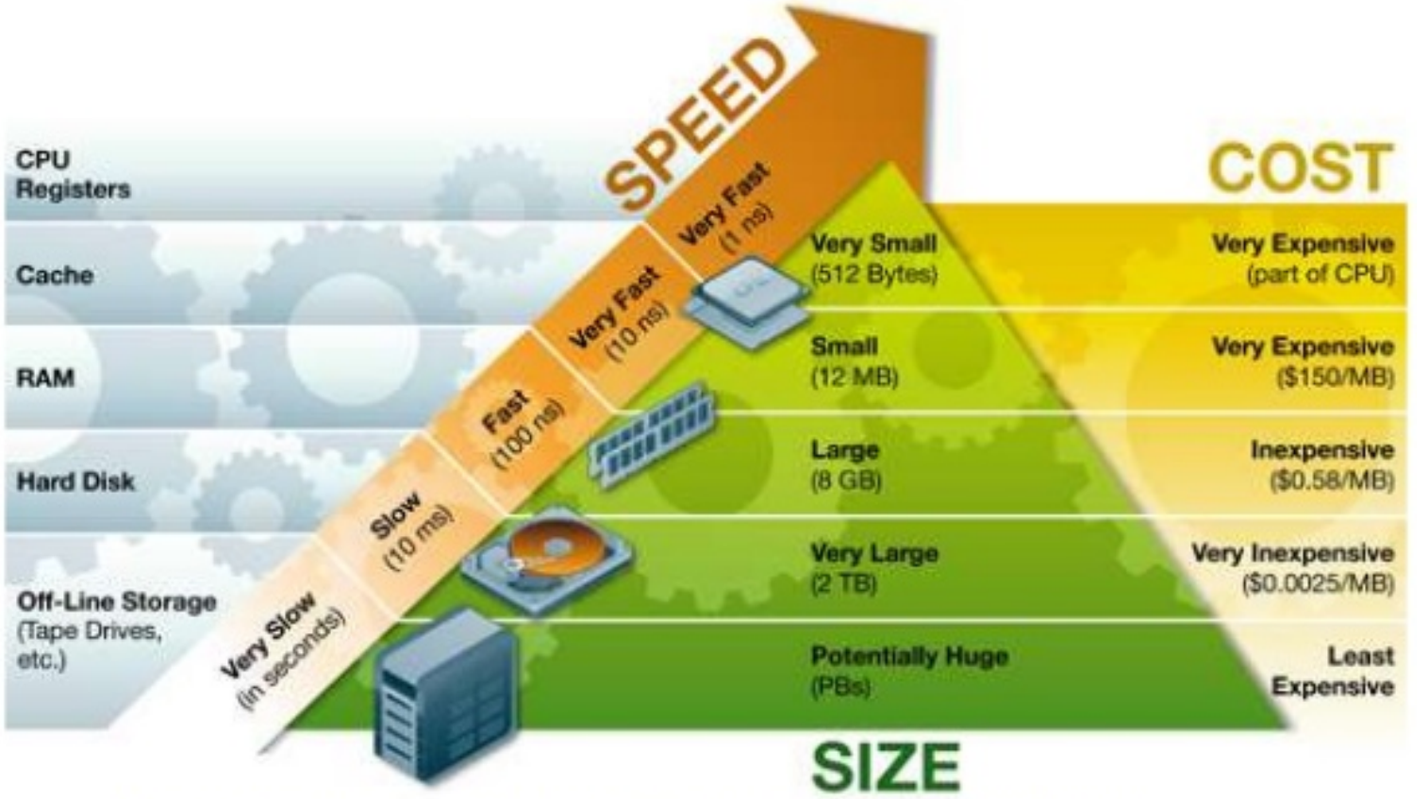
```
    public class OfficeBoy {

  @PersistenceUnit
  EntityManager manager;

  public void pagaBoletos(List ids) {
    for (Long idBoleto : ids) {
      Boleto boleto = manager.getReference(Boleto.class, idBol
      Float valor = boleto.getValor();
      boolean pago = BancoWSApi.pagaBoleto(boleto, valor);
      boleto.setPago(pago);
      manager.merge(boleto);
    }
  }
}
```


API Banco
resultado API
busca dados boleto
Banco de dados
API http
leva número boleto
retorno boleto
CPU
thread1 thread2 thread3 thread4 thread5
persiste resutado
resultado persistência

Extended Memory Hierarchy



Source: http://www.ts.avnet.com/uk/products_and_solutions/storage/hierarchy.html

QUANDO?

- *Servlets*
- *Clientes HTTP*
- *I/O*
- *Grande volume de dados*

QUANDO NÃO?

- *Quando houver necessidade de lock*
- *Dentro de transações*

EVITANDO LOCK

Adicionando e consulmindo sem lock

```
     <!--
List<String> tmp = new ArrayList<>();

private List<String> getWorkList() {
    List<String> work = tmp;
    tmp = new ArrayList<>();
    return work;
}-->
```

EVITANDO LOCK

Use objetos imutáveis para salvar estado

```
<!--
List<String> work = Collections.unmodifiableList(new ArrayList

public synchronized void addToWorkList(String o) {
        List<String> tmp = new ArrayList<>(work);
        tmp.add(o);
        work = Collections.unmodifiableList(tmp);
} -->
```

COMPLETABLE FUTURE

Representa um objeto Futuro que pode ser explicitamente completado, e que pode ser utilizado para agregar funções dependentes que serão disparadas quando isto ocorrer.

INTERFACES IMPLEMENTADAS

- Future
  - Não notifica quando o resultado está disponível.
  - Métodos: get() ou isDone()

- CompletionStage
  - Realiza uma ação quando uma outra ação for concretizada.

MÉTODOS

- 38 métodos total
- Características dos métodos:
  - somethingAsync(..., Executor)
  - somethingAsync(...)
    - Utiliza ForkJoinPool
  - something(...)

MÉTODOS

- Argumentos e retornos:
  - Apply
  - Accept
  - Run

MÉTODOS

- Forma de execução do método
  - single input
    - thenApply, thenAccept, thenRun
  - binary or
    - applyToEither, acceptEither, runAfterEither
  - binary and
    - thenCombine, thenAcceptBoth, runAfterBoth

MÉTODOS:

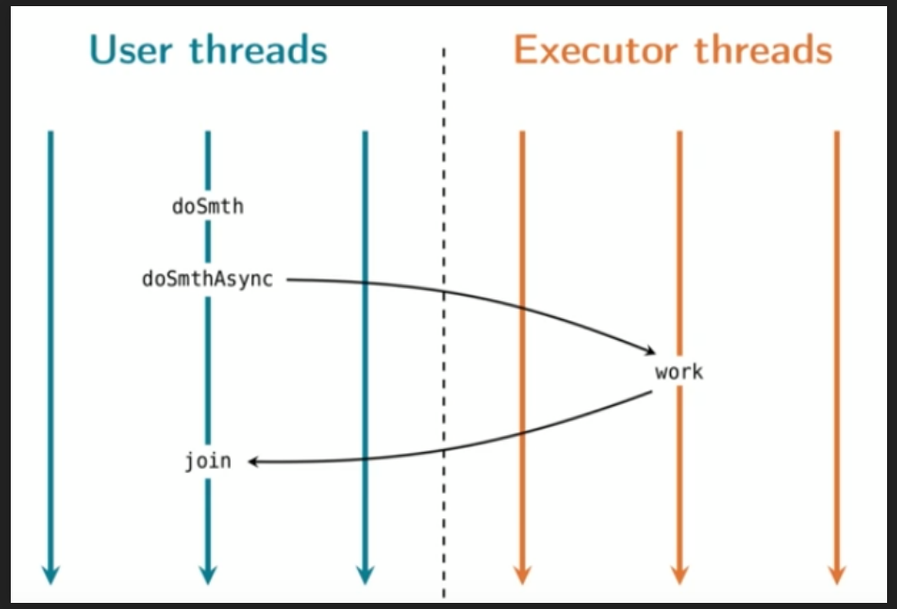
- thenCompose
- whenComplete
- exceptionally
- handle

MÉTODOS

- Diferenças:
  - thenCompose: retorna outro stage que retornará o resultado
  - thenApply: retorna resultado em sí

COMPORTAMENTO DAS THREADS

- ForkJoinPool
- Executors



Threads demais podem levar a cenários de erro!

COMPORTAMENTO DAS THREADS

Quiz

Thread 1

```
future.thenApply((...) -> foo());
```

Thread 2

```
future.complete(...);
```

Which thread will execute `foo()`?

A) thread 1
B) thread 2
C) thread 1 or thread 2
D) thread 1 and thread 2

COMPORTAMENTO DAS THREADS

Quiz

Thread 1	Thread 2
`future.thenApply((...) -> foo());`	`future.complete(...);`

Which thread will execute `foo()`?

A) thread 1
B) thread 2
C) thread 1 or thread 2 ← correct answer
D) thread 1 and thread 2

COMPORTAMENTO DAS THREADS

CompletableFuture chaining

- `thenSomethingAsync(...)` gives predictability.
- `thenSomething(...)` gives performance.

TRATAMENTO DE EXCEÇÃO

- whenComplete
- exceptionally
- handle

EXEMPLO

- Exemplo utilizando os principais métodos:
- TestCompletableFuture

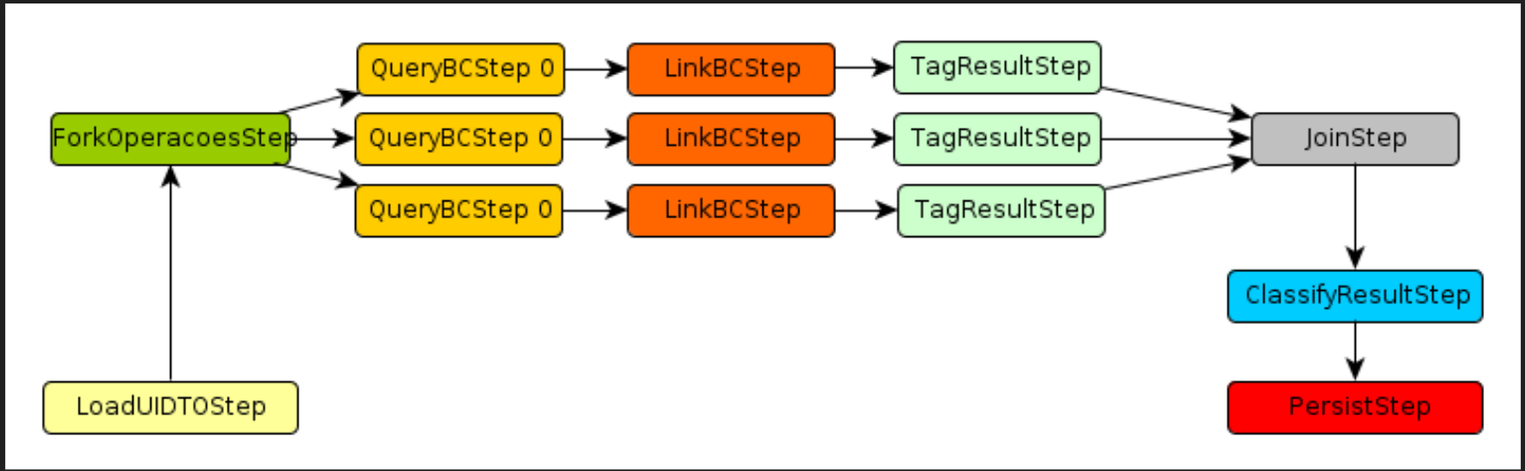
PERFORMANCE

- Asynchronous API with CompletableFuture
- JavaOne Oracle - Sergey Kuksenko

- Evite transições entre uma Thread para outra. Isto custa!
- Evite excesso de threads!
- Evite bloquear as Threads!
  - get()
  - join()

CASOS DE USO

Workflow para enriquecimento de dados



Workflow para enriquecimento de dados

```
var templateStep = new QueryBCStep(stepConfig);
templateStep.nextStep(new LinkBCStep(stepConfig))
                        .nextStep(new TagResultStep(stepConfig

var stepInicial = new LoadUIDTOStep(stepConfig);
stepInicial.nextStep(new ForkOperacoesStep(stepConfig, request
                .nextStep(new JoinStep(stepConfig))
                .nextStep(new ClassifyResultStep(stepConfig))
                .nextStep(new PersistTemporaryStep(stepConfig)
                .nextStep(callback);

stepInicial.execute(stepContext, request.getUIID());
```

Workflow para enriquecimento de dados

```
<!--
List<CompletableFuture<List<UIDTO>>> promises = this.executaOp
proceed(stepContext, promises);
-->
```

Workflow para enriquecimento de dados

```
<!--
protected CompletableFuture<List<UIDTO>> criarPromiseComTodosV
        try {
                CompletableFuture allOf = CompletableFuture.al
                CompletableFuture<List<UIDTO>> allPromises = a
                return allPromises.thenApply((uis) -> {
                        List<UIDTO> valores = new ArrayList<UI
                        for (CompletableFuture<List<UIDTO>> pr
                                try {
                                        List<UIDTO> lista = pr
                                        if (lista != null) {
                                                valores.addAll
                                        }
                                } catch (Exception e) {
                                        tratarErro(e);
```

Workflow para enriquecimento de dados

```
<!--
CompletableFuture<List<UIDTO>> uisFuture = this.criarPromiseCo
uisFuture.whenComplete((uis, erro) -> {
        proceed(stepContext, uis);
});
-->
```

BENEFÍCIOS

- Componentes com escopo bem definido e reutilizáveis (arquitetura)
- Legibilidade do fluxo de negócio (manutenção)
- Melhor aproveitamento dos recursos de hardware (performance)

RISCOS

- Hábitos de programação síncrona
- Callback-hell
- Estados inesperados por modificação concorrente
- Deadlocks

Emerson Venâncio @ Dígitro Tecnologia

Filipi da Silva Fuchter @ Dígitro Tecnologia

https://github.com/filipi87/completablefuture-revealjs