Kotlin Web API com Spring

eBook tutorial sobre como criar uma Web API Kotlin com Spring

Tabela de Conteúdo

Projeto

Home (../README.md)

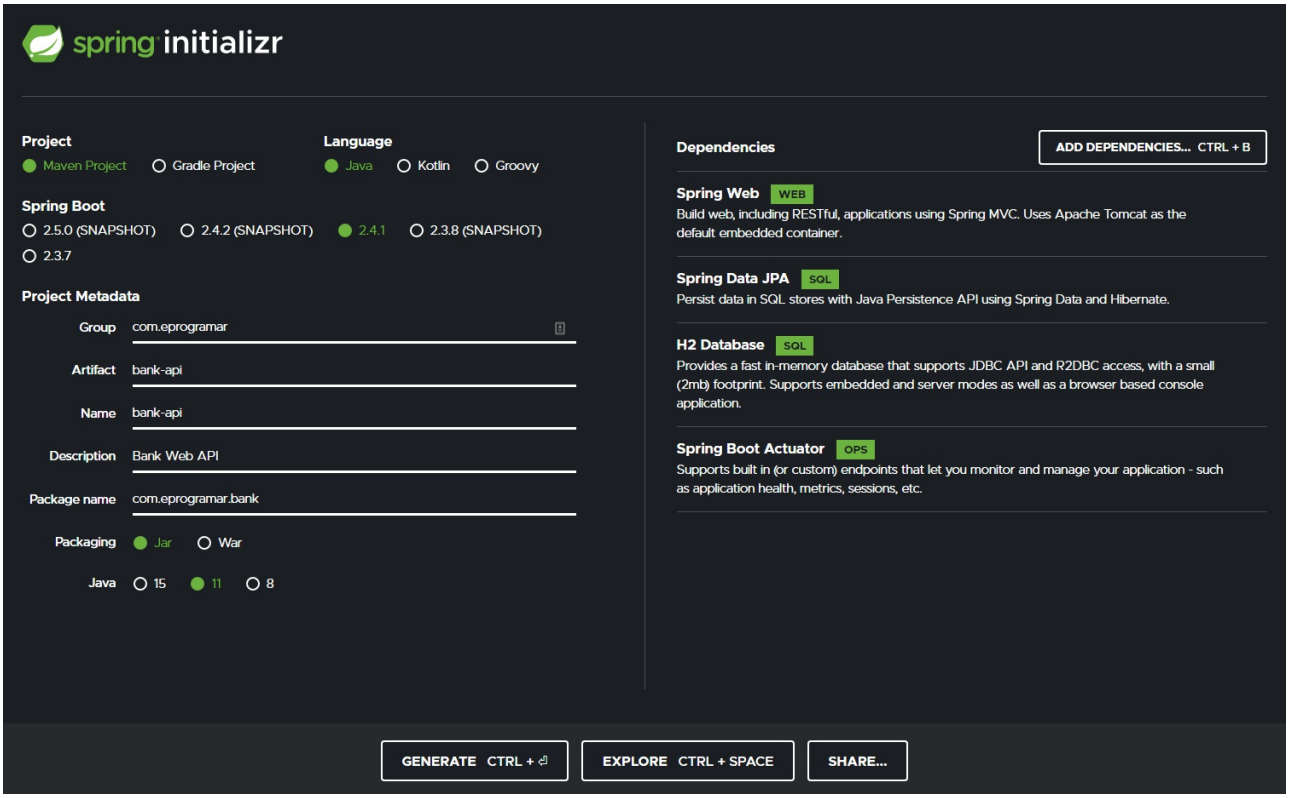
Como **Requisito** para seguir este tutorial é preciso ter instalado o Java a partir da versão 11, Maven 3 e uma IDE como Visual Studio Code ou IntelliJ, eu aconsenlho Instalar a versão **Community** que é gratuita e atende completamente este tutorial.

o IntelliJ pode ser baixado pelo site da **JetBrains**:
https://www.jetbrains.com/pt-br/idea/download/ (https://www.jetbrains.com/pt-br/idea/download/)

Spring Initialzr

Para começar vamos entrar no site do Spring Initializr e criar nosso projeto.

https://start.spring.io/ (https://start.spring.io/)



Projeto

Agora vamos configurar:

1. Projeto
2. Linguagem
3. Versão do Spring Boot

Vamos utilizar como gerenciado de dependencias e build o **Maven**
Em seguida vamo escolher **Kotlin** como Linguagem
Então, vamos escolher a versão do Spring Boot e aqui vale dizer que versão mais nova que não

esteja em **SNAPSHOT**, no meu caso no momento que estou criando este projeto é **2.4.1** mais se no seu caso não tiver esssa versão disponivel você poderá selecionar a ultima depois alterar o arquivo `pom.xml` com a versão que estou utilizando para não ter problema de compatibilidade entre versões.

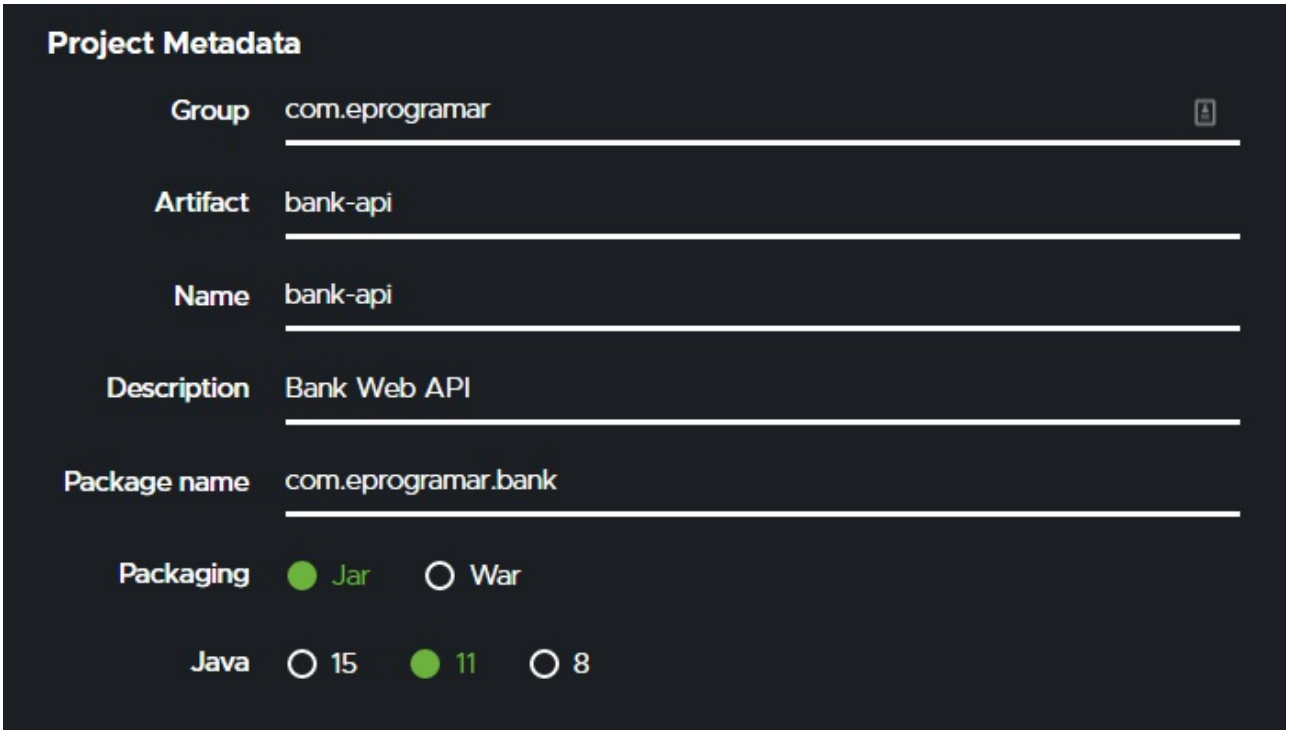
dentro do `pom.xml` altere como a seguir:

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.4.0</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>
```

Metadata

Agora vamos configurar os metadatas do nosso projeto, agora basta seguir como a seguir:

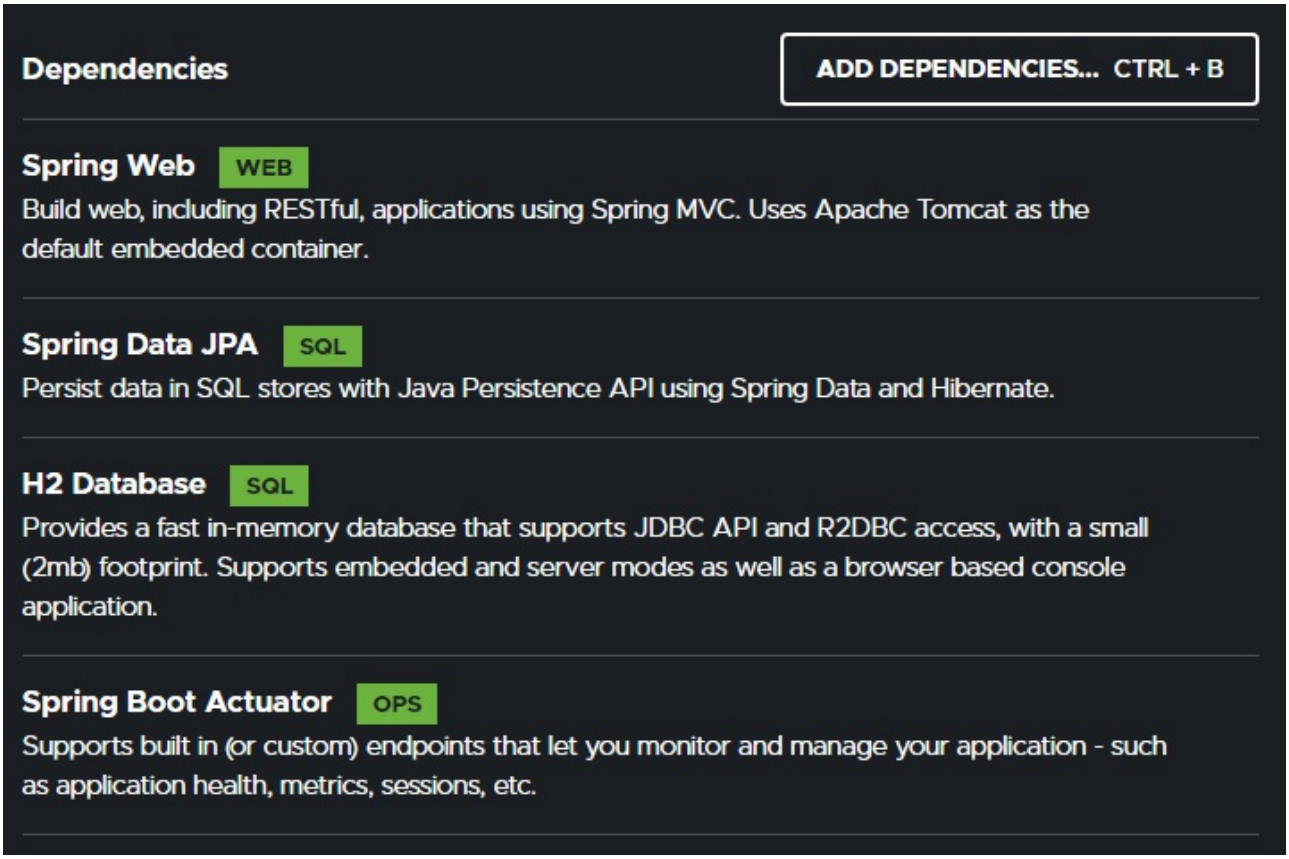
Group: `com.eprogramar`
Artifact: `bank-api`
Name: `bank-api`
Description: `Bank Web API`
package Name: `com.eprogramar.bank`
Packaging: `JAR`
Java: `11`



Dependencias

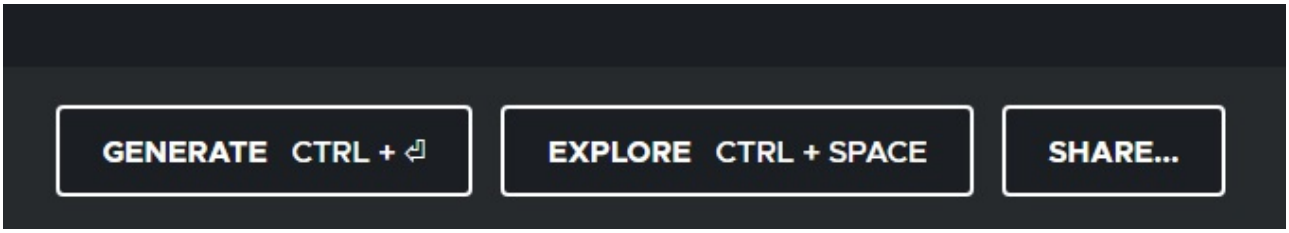
Vamos seguir agora escolhendo as dependencias que precisaremos:

Spring Web: Criar nosso Controller e Rotas.
Spring Data JPA: Nosso `ORM(Object Relational Mapper)` para manipularmos todo o acesso a Banco de Dados.
H2 Database: Como o proprio nome já diz, será nosso Banco de Dados. O H2 trabalha em memória então não precisamos em desenvolvimento instalar nenhum Banco de Dados.
Spring boot Actuator: Criar automaticamente alguns endpoints em nosso API para monitorar e gerenciar a API, como healthcheck, metricas e sessões.



Generação do Projeto

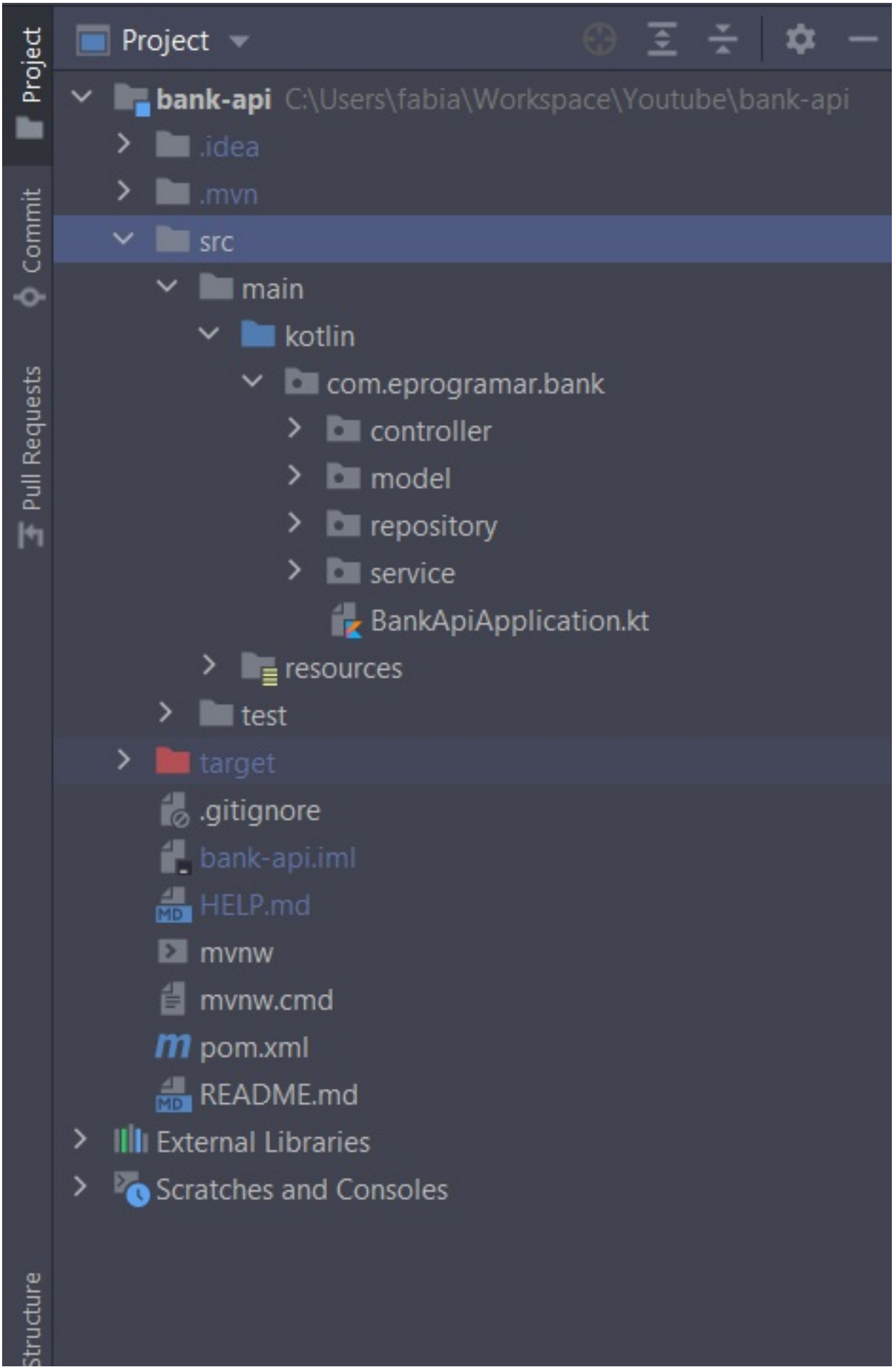
Com tudo configurado podemos clicar no botão `GENERATE`, será disponibilizado um `zip` para download.
Após o download descompacte e mova para sua pasta de trabalho de preferencia.
Abra em sua `IDE` no meu caso estou usando o `Intellij`.



Estrutura

Como o projeto aberto na `IDE` vamos criar os seguitas packages:

- controller
- model
- repository
- service


Project
Commit
Pull Requests
Structure
Project
bank-api C:\Users\fabia\Workspace\Youtube\bank-api
.idea
.mvn
src
main
kotlin
com.eprogramar.bank
controller
model
repository
service
BankApiApplication.kt
resources
test
target
.gitignore
bank-api.iml
HELP.md
mvnw
mvnw.cmd
pom.xml
README.md
External Libraries
Scratches and Consoles

Repositorio

Home (../README.md)

A Repository é a parte mais simples porque aqui quem faz a magica é o Spring Data JPA.
Precisamos apenas de uma interface que extenda de `JpaRepository`.
Então, dentro do package `repository` crie uma interface Kotlin chamada `AccountRepository`

```
package com.eprogramar.bank.repository

import com.eprogramar.bank.model.Account
import org.springframework.data.jpa.repository.JpaRepository

interface AccountRepository : JpaRepository<Account, Long> {
}
```

Perceba que a interface deve extender de `JpaRepository` e usar a nossa classe de Modelo/Entidade `Account` seguido de Long que é o tipo do nosso atributo que leva a anottation `@Id`.

Feito isso, o Spring Data faz toda a magia e cria todos os métodos necessários para um `CRUD`

- **C**reate
- **R**ead
- **U**pdate
- **D**elete

Porém a interface não implementar nenhum método para consulta através de um atributo de nossa classe além do `id`.

Mas calma, ele não faz mas permite uma maneira muito simples para que façamos nós mesmo.
Imagine que queremos uma consulta pelo atributi `document` da nossa claase de modelo.

basta seguir uma convenção de `finBy` + o nome do atrabuto,
como aseguir:

```
fun findByDocument(document: String): Optional<Account>
```

Então passamos como parametro o valor para a consulta.
E retornamos um `Optional` do java.util de nossa classse de modelo `Account`.
Se a consulta retornar uma `Account` bastar fazer um `.get` no optional e para verificar se ter vamos no retorno da consulta usamdo `.isPresent`.

Sem muitas delongas, daria para falarmos muita coisa sobre **Spring Data JPA** mas por hora, para nosso exemplo já é suficiente.

Nossa interface final fica assim:

```
package com.eprogramar.bank.repository

import com.eprogramar.bank.model.Account
import org.springframework.data.jpa.repository.JpaRepository
import java.util.*

interface AccountRepository : JpaRepository<Account, Long> {

    fun findByDocument(document: String): Optional<Account>

}
```

Service

Home (../README.md)

Agora vamos criar a nossa classe `Service` que é onde deveria ficar toda regra de negócio da nossa aplicação.
No nosso caso temos uma aplicação bem simples mas essa é uma estrutura já pronta para a evolução da aplicação.

Vamos começão criando no package `service` uma classe Kotlin chamada `AccountService` e no construtor primário vamos injetar nossa `repository` como um atributo readonly privado, veja como deve ficar a estrutura inicial:

```
@Service
class AccountService(private val repository: AccountRepository) {

}
```

Perceba que usamos a Annotation `@Service` sobre nossa classe.

Então vamos começar criando nossos métodos `CRUD`.

Regra Cread

```
override fun create(account: Account): Account {
    return repository.save(account)
}
```

Regra Read

```
override fun getAll(): List<Account> {
    return repository.findAll()
}

override fun getById(id: Long): Optional<Account> {
    return repository.findById(id)
}
```

Regra Update

```
override fun update(id: Long, account: Account): Optional<Account> {
    val optional = getById(id)
    if (optional.isEmpty) {
        return optional
    }

    return optional.map {
        val accountToUpdate = it.copy(
            name = account.name,
            document = account.document,
            phone = account.phone
        )
        repository.save(accountToUpdate)
    }
}
```

Regra Delete

```
override fun delete(id: Long) {
    repository.findById(id).map {
        repository.delete(it)
    }.orElseThrow { throw RuntimeException("Id not found $id") }

}
```

Controller Refactoring

Home (../README.md)

Então depois de criamos toda nossa base da aplicação, vamos refatorar nosso controller para ficar na versão final e funcional.

a primeria coisa que precisamos fazer é injetar nossa service no construtor primario no controller assim como fizemos com a repository sendo injetada em nossa service.

```
@RestController
@RequestMapping("/accounts")
class AccountController(private val service: AccountService) {
    ...
}
```

agora vamos refatorar todos os nossos métodos:

Create

```
@PostMapping
@ResponseStatus(HttpStatus.CREATED)
fun create(@RequestBody account: Account): Account = service.create(account)
```

Read

```
@GetMapping
fun getAll(): List<Account> = service.getAll()

@GetMapping("/{id}")
fun getById(@PathVariable id: Long) : ResponseEntity<Account> =
        service.getById(id).map {
            ResponseEntity.ok(it)
        }.orElse(ResponseEntity.notFound().build())
```

Update

```
@PutMapping("/{id}")
fun update(@PathVariable id: Long, @RequestBody account: Account) :
ResponseEntity<Account> =
        service.update(id, account).map {
            ResponseEntity.ok(it)
        }.orElse(ResponseEntity.notFound().build())
```

Delete

```
    @DeleteMapping("/{id}")
    fun delete(@PathVariable id: Long) : ResponseEntity<Void> {
        service.delete(id)
        return ResponseEntity<Void>(HttpStatus.OK)
    }
```

e assim temos nosssa versão final e funcional de nosso controlles passando por todas as camadas.

Testes Automátizados

???

CI - Integração Continua

Travis CI

???

Deploy

Heroku

???