

Kotlin Web API com Spring

eBook tutorial sobre como criar uma Web API Kotlin com Spring

Tabela de Conteúdo

1. Criando o Projeto
2. Controller
3. Repositorio
4. Serviços
5. Controller Refactoring
6. Testes Automatizados
7. Integração Continua
8. Deploy

Esta obra está licenciada com uma Licença [Creative Commons Atribuição-NãoComercial-SemDerivações 4.0 Internacional](https://creativecommons.org/licenses/by-nc-nd/4.0/) (<https://creativecommons.org/licenses/by-nc-nd/4.0/>).



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 aconselho 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 Initializr

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

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

The screenshot shows the Spring Initializr web interface. It has a dark theme with green accents. The form is divided into several sections:
1. **Project**: Includes radio buttons for 'Maven Project' (selected) and 'Gradle Project'.
2. **Language**: Includes radio buttons for 'Java' (selected), 'Kotlin', and 'Groovy'.
3. **Spring Boot**: Includes radio buttons for versions 2.5.0 (SNAPSHOT), 2.4.2 (SNAPSHOT), 2.4.1 (selected), and 2.3.8 (SNAPSHOT). There is also a 2.3.7 option below.
4. **Project Metadata**: Fields for Group (com.eprogramar), Artifact (bank-api), Name (bank-api), Description (Bank Web API), and Package name (com.eprogramar.bank).
5. **Packaging**: Radio buttons for 'Jar' (selected) and 'War'.
6. **Java**: Radio buttons for versions 15, 11 (selected), and 8.
7. **Dependencies**: A list of dependencies with checkboxes: 'Spring Web' (WEB, selected), 'Spring Data JPA' (SQL, selected), 'H2 Database' (SQL, selected), and 'Spring Boot Actuator' (OPS, selected). A button 'ADD DEPENDENCIES... CTRL + B' is at the top right of this section.
8. **Buttons**: At the bottom, there are three buttons: 'GENERATE CTRL + G', 'EXPLORE CTRL + SPACE', and 'SHARE...'.
The Spring logo and 'spring initializr' text are at the top left.

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 essa versão disponível 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.

A screenshot of the Spring Boot CLI interface. It shows two main sections: 'Project' and 'Language'. Under 'Project', there are three radio buttons: 'Maven Project' (selected), 'Gradle Project', and 'Groovy Project'. Under 'Language', there are three radio buttons: 'Java' (selected), 'Kotlin', and 'Groovy'. Below these, there is a 'Spring Boot' section with five radio buttons for versions: '2.5.0 (SNAPSHOT)', '2.4.2 (SNAPSHOT)', '2.4.1' (selected), '2.3.8 (SNAPSHOT)', and '2.3.7'.

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 metadados 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

Project Metadata

Group

com.eprogramar

Artifact

bank-api

Name

bank-api

Description

Bank Web API

Package name

com.eprogramar.bank

Packaging

☒ Jar

☐ War

Java

☐ 15

☒ 11

☐ 8

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, mettricas e sessões.

Dependencies

ADD DEPENDENCIES... CTRL + B

Spring Web WEB

Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Spring Data JPA SQL

Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

H2 Database SQL

Provides a fast in-memory database that supports JDBC API and R2DBC access, with a small (2mb) footprint. Supports embedded and server modes as well as a browser based console application.

Spring Boot Actuator OPS

Supports built in (or custom) endpoints that let you monitor and manage your application - such as application health, metrics, sessions, etc.

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.

GENERATE CTRL + ⌘

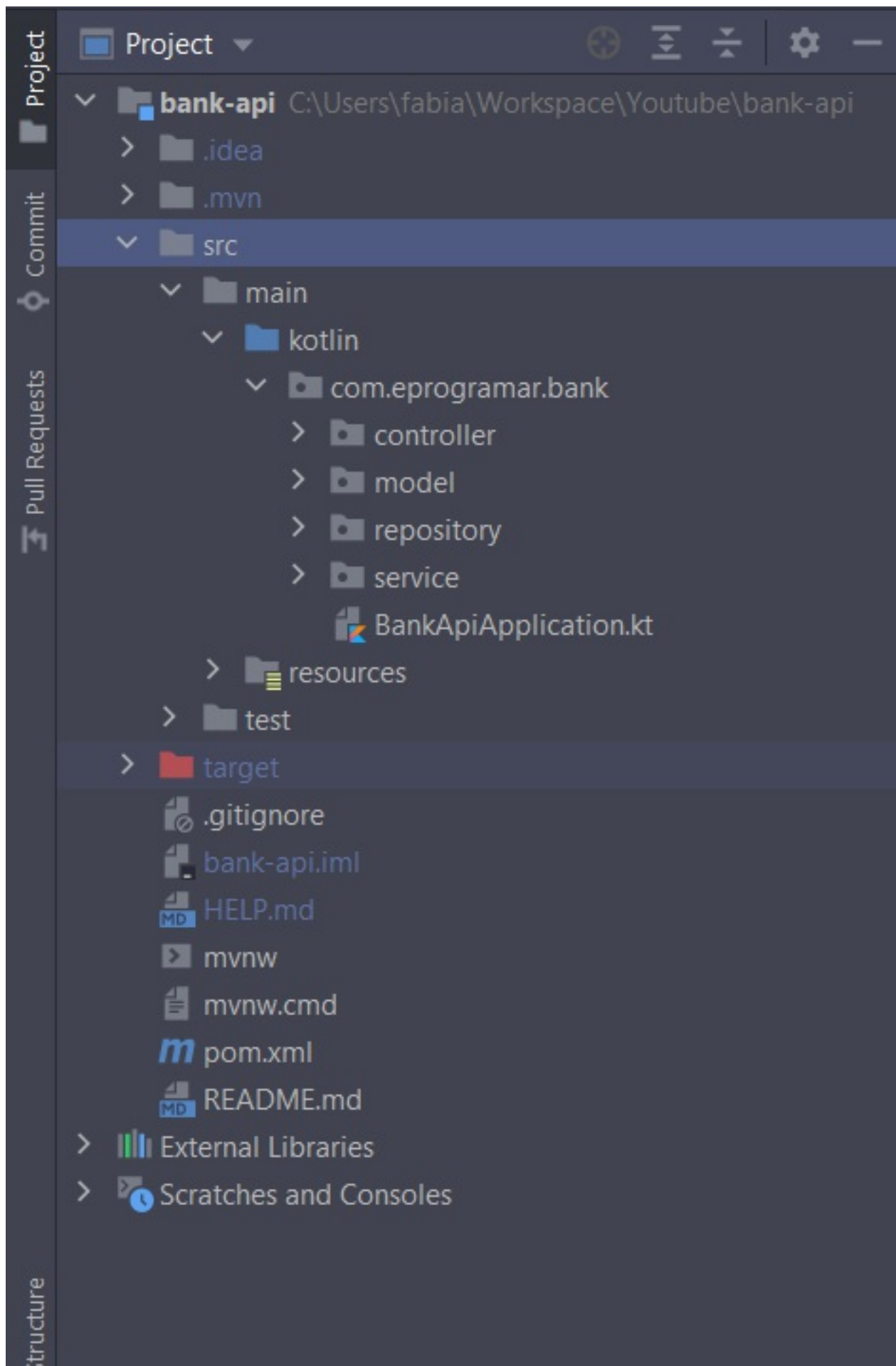
EXPLORE CTRL + SPACE

SHARE...

Estrutura

Como o projeto aberto na IDE vamos criar os seguintes packages:

- controller
- model
- repository
- service



Repositorio

[Home \(../README.md\)](#)

A Repository é a parte mais simples porque aqui quem faz a magia é o Spring Data JPA. Precisamos apenas de uma interface que estenda 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 estender de JpaRepository e usar a nossa classe de Modelo/Entidade Account seguido de Long que é o tipo do nosso atributo que leva a anotation @Id.

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

- Create
- Read
- Update
- Delete

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 atributo document da nossa classe de modelo.

basta seguir uma convenção de findBy + o nome do atributo, como a seguir:

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

Então passamos como parametro o valor para a consulta.

E retornamos um Optional do java.util de nossa classe de modelo Account.

Se a consulta retornar uma Account basta fazer um .get no optional e para verificar se ter vamos no retorno da consulta usando .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 começar 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 criarmos toda nossa base da aplicação, vamos refatorar nosso controller para ficar na versão final e funcional.

a primeira 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 nossa versão final e funcional de nosso controlles passando por todas as camadas.

Testes Automatizados

???

CI - Integração Continua

Travis CI

???

Deploy

Heroku

???