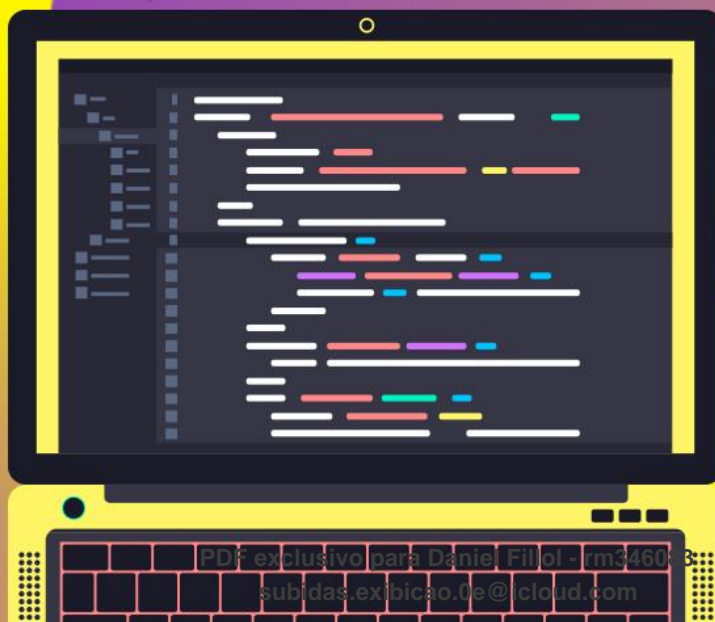


SPRING SPRING FRAMEWORK

FABIO TADASHI MIYASATO



1

LISTA DE FIGURAS

Figura 1.1 – Códigos de resposta HTTP	7
Figura 1.2 – Comparativo de estruturas do Maven e Gradle.....	10
Figura 1.3 – Indicador de desempenho de build do Gradle.....	10
Figura 1.4 – Spring Initializr	11
Figura 1.5 – Apresentação do projeto inicial gerado pelo Spring Initializr	12
Figura 1.6 – Configurações em formatos de propriedades (esquerda) e YAML (direita)	13
Figura 1.7 – Console do IntelliJ com mensagem de sucesso na execução da aplicação	14
Figura 1.8 – Resultado da consulta de livros via REST	16
Figura 1.9 – Escopos de instância do Bean	24
Figura 1.10 – Hierarquia de dependências do Gradle.....	29
Figura 1.11 – Simulação de exceção com parâmetros inexistentes.....	32
Figura 1.12 – Escopos de instância do Bean	35

LISTA DE CÓDIGOS-FONTE

Código-fonte 1.1 – Alteração da porta padrão de execução do servidor.....	13
Código-fonte 1.2 – Estrutura da classe LivroDTO	14
Código-fonte 1.3 – Estrutura da classe LivroController	15
Código-fonte 1.4 – Estrutura do método buscarLivros	15
Código-fonte 1.5 – Estrutura da Query Parameter	18
Código-fonte 1.6 – Estrutura do método buscarPorId	19
Código-fonte 1.7 – Estrutura da classe CreateUpdateLivroDTO.....	20
Código-fonte 1.8 – Estrutura do método criar.....	21
Código-fonte 1.9 – Estrutura do método atualizar	22
Código-fonte 1.10 – Estrutura da classe UpdatePrecoLivroDTO	22
Código-fonte 1.11 – Estrutura do método atualizarPreco.....	23
Código-fonte 1.12 – Estrutura da classe AppConfiguration.....	25
Código-fonte 1.13 – Estrutura da classe DebugController	26
Código-fonte 1.14 – Estrutura da classe LivroService.....	27
Código-fonte 1.15 – Estrutura da classe livroService	27
Código-fonte 1.16 – Exemplo de inclusão de dependências com Gradle	28
Código-fonte 1.17 – Estrutura da classe Livro.....	30
Código-fonte 1.18 – Estrutura da classe livroRepository.....	32
Código-fonte 1.19 – Métodos de serviços utilizando o repositório configurado.....	35
Código-fonte 1.20 – Script para geração da tabela Livro	36

SUMÁRIO

1 SPRING FRAMEWORK	5
1.1 Por que Spring?	5
1.2 REST	5
1.2.1 Recursos	5
1.2.2 Métodos HTTP	6
1.2.3 Códigos HTTP	7
1.3 Spring Boot.....	8
1.3.1 Ambiente de Desenvolvimento	9
1.3.2 Maven x Gradle	9
1.3.3 Spring Initializr	11
1.3.4 Configurações externalizadas	13
1.4 Rest Controller	14
1.4.1 Parâmetro - Query Parameter	16
1.4.2 Parâmetro - Path Parameter	18
1.4.3 Parâmetro - Request Body	19
1.5 Injeção de dependência	23
1.6 Bean.....	24
1.6.1 Conditional Bean	25
1.6.2 Service	26
1.7 Spring Data	28
1.7.1 Configuração	28
1.7.2 Entidade	30
1.7.3 Repository	31
1.7.4 Versionamento de banco de dados – Flyway	35
CONCLUSÃO.....	37
REFERÊNCIAS.....	38
GLOSSÁRIO	39

1 SPRING FRAMEWORK

1.1 Por que Spring?

Como a própria documentação define, Spring torna a programação Java mais rápida, simples e segura. Com foco em produtividade, o Spring é hoje o framework Java mais popular do mundo (<https://snyk.io/blog/jvm-ecosystem-report-2020/>).

Atualmente o Spring Framework conta com inúmeros módulos para gerenciamento de configuração, aplicações web, big data, integração cloud, segurança, entre outros. Ao longo deste capítulo, os principais módulos serão utilizados para o desenvolvimento de uma aplicação web, seguindo o padrão Rest.

1.2 REST

Antes de começar com o Spring, é necessário relembrar os principais conceitos do padrão Rest. Criado em 2000 na dissertação de doutorado de Roy Fielding, Rest (*REpresentational State Transfer*) é uma padronização de arquitetura de software para web services e surgiu como uma alternativa ao protocolo SOAP.

Para que uma aplicação seja considerada RESTful (seguindo a padronização Rest), ela deve implementar uma série de regras, gerando uma interface bem definida e, assim, facilitar a comunicação entre as aplicações.

1.2.1 Recursos

O conceito de recursos (resources) é um dos mais importantes na criação de uma aplicação com padrão Rest. Um recurso é a representação de uma entidade que será disponibilizada para manipulação (operações CRUD, por exemplo) pelo webservice.

Também conhecidos como Resource Models, os recursos muitas vezes (mas não necessariamente) estão relacionados com as entidades do banco de dados ou com qualquer outra forma de armazenamento de modelos da aplicação.

No padrão Rest, os recursos devem estar no plural e farão parte da URL. Por exemplo, no caso do recurso “usuário”, a URL do serviço ficaria:

`http://endereco-api.com/usuarios`

1.2.2 Métodos HTTP

Dado um recurso, serão utilizados os métodos (ou verbos) do protocolo http. Esses métodos representam a ação a ser realizada com o recurso. Por exemplo, o método GET é utilizado para obter (buscar) o recurso. Logo, por meio da combinação do recurso e do método, é possível identificar o que o endpoint deve fazer.

GET => `http://endereco-api.com/usuarios`

No exemplo acima, a utilização do método GET no recurso “usuários” deve retornar uma lista de usuários. O padrão Rest não define o formato que essa lista de usuários deve retornar, porém a prática comum de mercado é o formato Json.

A seguir, são relacionados os 5 principais métodos HTTP utilizados:

Método HTTP	Operação SQL	
Método HTTP	Operação SQL	
GET	READ	Utilizado para buscar/listar recursos
POST	CREATE	Para criação de um recurso
PUT	UPDATE	Atualização completa de um recurso
PATCH	UPDATE (parcial)	Atualização parcial dos dados de um recurso
DELETE	DELETE	Deletar um recurso

Além desses, vale a pena mencionar os métodos OPTIONS bastante utilizados em APIs publicadas em cloud, como uma espécie de “pré-requisição” para saber quais as opções para o recurso e para o HEAD que funcionam como GET, porém indica que o endpoint não irá retornar um corpo na resposta.

1.2.3 Códigos HTTP

O padrão REST utiliza os códigos de resposta HTTP (*response codes*) para definir se a requisição foi realizada com sucesso ou se houve alguma falha.



Figura 1.1 – Códigos de resposta HTTP
Fonte: Rest API Tutorial (2021)

Conforme a imagem acima, os códigos HTTP são compostos por 3 números e estão divididos em cinco categorias diferentes, de acordo com o número que define a centena. Os códigos informacionais quase não são utilizados.

Códigos de sucesso (2xx) indicam que a requisição foi realizada, porém podem variar conforme o tipo de retorno:

- 201 Created – Em requisições POST, quando o recurso é criado.
- 202 Accepted – Requisição foi recebida e aceita pelo servidor, sem resposta imediata (utilizada para processos assíncronos).
- 204 No Content – Aceito, processado, porém sem corpo de resposta.

Na maioria das vezes, códigos de redirecionamento (3xx) estão relacionados à infraestrutura e, raramente, serão utilizados pelo desenvolvedor na aplicação.

- 301 Moved Permanently – O servidor moveu o recurso de forma permanente.
- 302 Found – O servidor moveu o recurso temporariamente.

Códigos do tipo “4xx” indicam que a requisição contém algum erro, e o cliente (aplicação que chama o endpoint) deve alterá-la antes de realizar uma nova requisição.

- 400 Bad Request – Requisição incorreta, falta alguma informação crítica.
- 401 Unauthorised – Erro de autenticação, a aplicação cliente não enviou dados de autenticação (usuário, senha ou token) ou eles estão incorretos.
- 402 Payment Required – Serviço pago.
- 403 Forbidden – O cliente está autenticado na aplicação (usuário, senha ou token corretos), porém não está autorizado para acessar determinado recurso.
- 404 Not Found – Recurso não encontrado.
- 405 Method not Found – Método (verbo) HTTP incorreto.
- 409 Conflict – Cliente tentando criar um recurso de forma duplicada.
- 412 Precondition Failed – Formato dos dados enviados pelo cliente está incorreto.
- 415 unsupported media type – O servidor não entendeu o formato enviado. Por exemplo, se a requisição enviou um xml, mas o server espera um json.
- 419 Too Many Requests – Servidor não conseguiu atender a quantidade de requisições realizadas.

Códigos do tipo “5xx” indicam um erro do lado do servidor, podem indicar tanto problemas na aplicação quanto problemas na infraestrutura.

- 500 Internal Server Error – Erro genérico do servidor; no caso do Spring, qualquer exceção não tratada resulta em um erro 500.
- 503 Service Unavailable – Servidor indisponível, pode ocorrer durante uma manutenção ou durante a inicialização da aplicação.
- 504 Gateway Timeout – O servidor não conseguiu atender a requisição a tempo.

1.3 Spring Boot

O Spring Boot permite a criação de aplicações web stand-alone (sem dependência de um servidor à parte) de forma simples e rápida, e production ready

utilizando o conceito de autoconfiguração, sem necessidade das configurações XML (bastante criticadas no Spring MVC) e sem geração de código em tempo de compilação. Como o próprio time do Spring gosta de resumir, a filosofia do Spring Boot é “just run” ou apenas “execute”.

1.3.1 Ambiente de Desenvolvimento

Para executar uma aplicação criada pelo Spring Boot é necessário instalar Java 8 ou superior (compatível até com Java 15). Os exemplos apresentados neste módulo foram elaborados com a IDE IntelliJ que tem versões gratuitas (community) e licenciadas (ultimate), mas outra IDE, como Eclipse, também pode ser adotada sem restrições.

1.3.2 Maven x Gradle

O Spring pode utilizar tanto Maven quanto Gradle como ferramenta de build e gerenciamento de dependência. Apesar de o Maven ser amplamente conhecido na comunidade Java, o Gradle, que já é bastante difundido na comunidade Android, vem ganhando bastante espaço em desenvolvimento Spring.

Ambos são bastante flexíveis e compatíveis entre si, mas utilizam diferentes linguagens. O Maven utiliza o já conhecido XML, e o Gradle pode utilizar groovy script ou kotlin script. Observando os trechos de código a seguir, nota-se que o Gradle é bem mais conciso. Ele compreende um pouco mais de um terço da quantidade de linhas do Maven, favorecendo a legibilidade de código.

```

1 <plugin>
2   <groupId>org.apache.maven.plugins</groupId>
3   <artifactId>maven-checkstyle-plugin</artifactId>
4   <version>2.12.1</version>
5   <executions>
6     <execution>
7       <configuration>
8         <configLocation>config/checkstyle/checkstyle.xml</configLocation>
9         <consoleOutput>true</consoleOutput>
10        <failsOnError>true</failsOnError>
11      </configuration>
12      <goals>
13        <goal>check</goal>
14      </goals>
15    </execution>
16  </executions>
17</plugin>
18<plugin>
19  <groupId>org.codehaus.mojo</groupId>
20  <artifactId>findbugs-maven-plugin</artifactId>
21  <version>2.5.4</version>
22  <executions>
23    <execution>
24      <goals>
25        <goal>check</goal>
26      </goals>
27    </execution>
28  </executions>
29</plugin>
30<plugin>
31  <groupId>org.apache.maven.plugins</groupId>
32  <artifactId>maven-pmd-plugin</artifactId>
33  <version>3.1</version>
34  <executions>
35    <execution>
36      <goals>
37        <goal>check</goal>
38      </goals>
39    </execution>
40  </executions>
41</plugin>

```

```

1 apply plugin: 'java'
2 apply plugin: 'checkstyle'
3 apply plugin: 'findbugs'
4 apply plugin: 'pmd'
5
6 version = '1.0'
7
8 repositories {
9   mavenCentral()
10 }
11
12 dependencies {
13   testCompile group: 'junit', name: 'junit', version: '4.11'
14 }

```

Figura 1.2 – Comparativo de estruturas do Maven e Gradle
Fonte: Elaborado pelo autor (2021)

O Gradle tem ainda um sistema de cache que diminui consideravelmente o tempo de build da aplicação, conforme demonstrado na imagem a seguir.

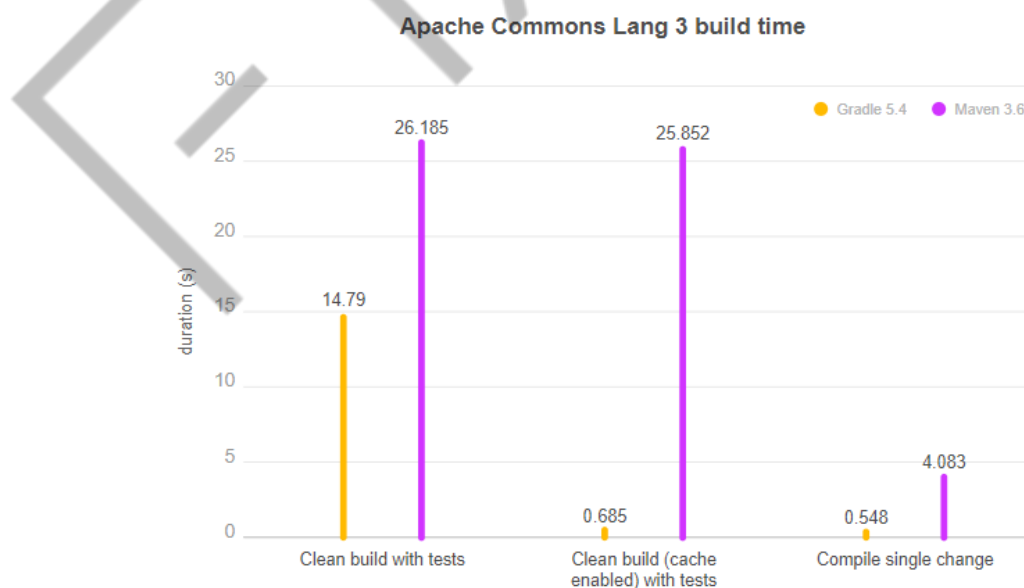
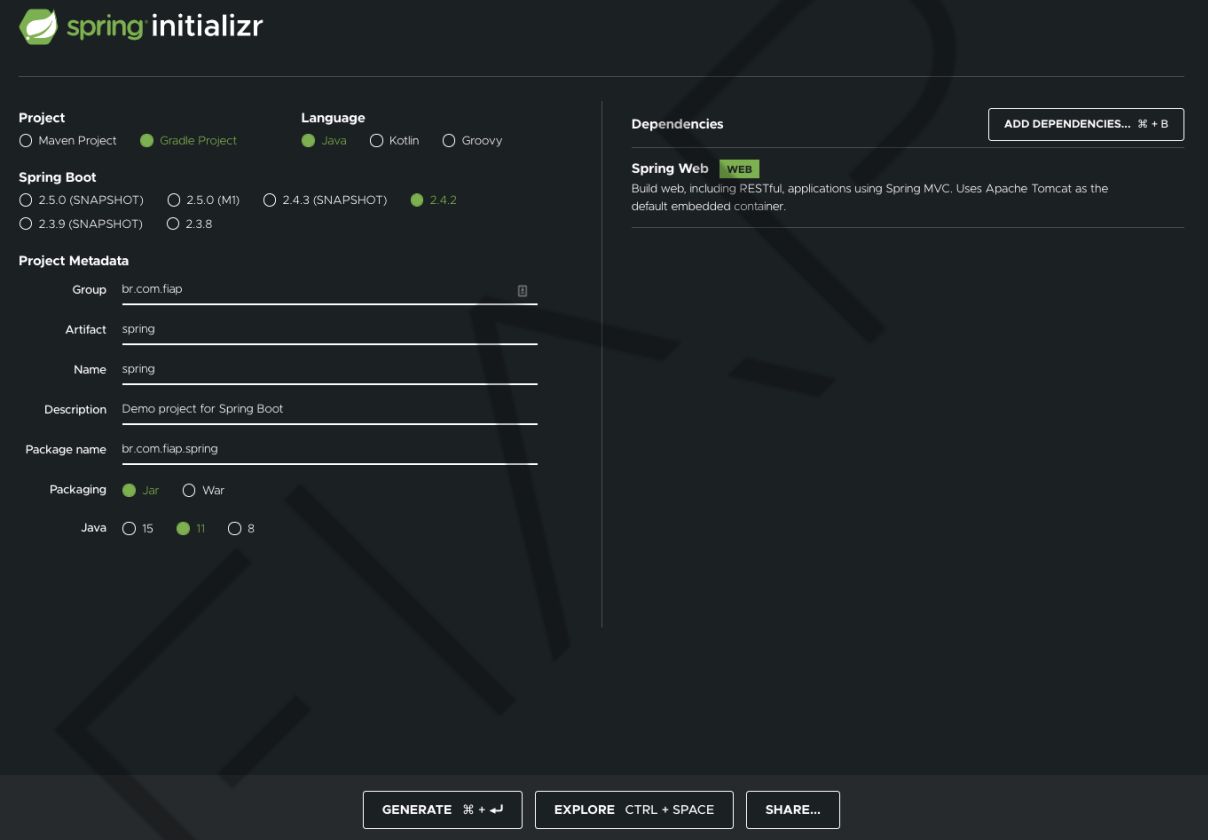


Figura 1.3 – Indicador de desempenho de build do Gradle
Fonte: Apache Commons Lang 3 (2021)

1.3.3 Spring Initializr

A recomendação para iniciar um projeto Spring é por meio do Spring Initializr (<https://start.spring.io/>). Trata-se de uma forma rápida de configurar o projeto e os módulos (dependências) Spring. Além disso, esse projeto sempre está atualizado com os releases estáveis do framework.



The screenshot displays the Spring Initializr web interface. It features a dark theme with a green Spring logo. The interface is divided into several sections: 'Project' with radio buttons for 'Maven Project' and 'Gradle Project' (selected); 'Language' with radio buttons for 'Java' (selected), 'Kotlin', and 'Groovy'; 'Spring Boot' with radio buttons for versions 2.5.0 (SNAPSHOT), 2.5.0 (M1), 2.4.3 (SNAPSHOT), 2.4.2 (selected), 2.3.9 (SNAPSHOT), and 2.3.8; 'Project Metadata' with input fields for Group (br.com.fiap), Artifact (spring), Name (spring), Description (Demo project for Spring Boot), and Package name (br.com.fiap.spring); 'Packaging' with radio buttons for 'Jar' (selected) and 'War'; and 'Java' with radio buttons for versions 15, 11 (selected), and 8. A 'Dependencies' section on the right shows 'Spring Web' selected under a 'WEB' tag, with a description: 'Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.' and an 'ADD DEPENDENCIES... % + B' button. At the bottom, there are three buttons: 'GENERATE % + ↵', 'EXPLORE CTRL + SPACE', and 'SHARE...'.

Figura 1.4 – Spring Initializr
Fonte: Spring Initializr (2021)

Conforme a imagem acima, bastam alguns cliques e o projeto já está configurado e pronto para rodar. A primeira configuração é referente à forma de build e ao gerenciamento de dependências. Com muitos pontos a favor do Gradle e poucos contra, vamos utilizar a opção “Gradle Project” e Java como linguagem.

A recomendação é utilizar a versão estável mais recente na opção do Spring Boot que já vem selecionada por padrão. Na imagem “Spring Initializr” é apresentada a versão 2.4.2, mas existe como opção da última versão estável (2.3.8), além de

alguns SNAPSHOTS, que só devem ser utilizadas em caráter experimental ou em casos muito específicos.

Em “Project Metadata” estão a configuração do grupo, artefato, nome do projeto, descrição e nome do pacote, além da forma de empacotamento (utilizaremos jar) e a versão do Java. Vale lembrar que qualquer configuração feita pelo Spring Initializr pode ser alterada posteriormente.

Na metade direita da tela (imagem “Spring Initializr”) é apresentado o gerenciamento de dependências, onde é possível selecionar as dependências do Spring que serão utilizadas na aplicação. Inicialmente vamos selecionar apenas “Spring Web”.

Para finalizar, basta clicar no botão “generate” e será realizado o download do projeto configurado no formato zip. Descompacte o arquivo e abra a pasta na IDE. Após abrir e compilar, basta clicar na opção RUN (botão play) e o projeto estará disponível para execução local na porta 8080.

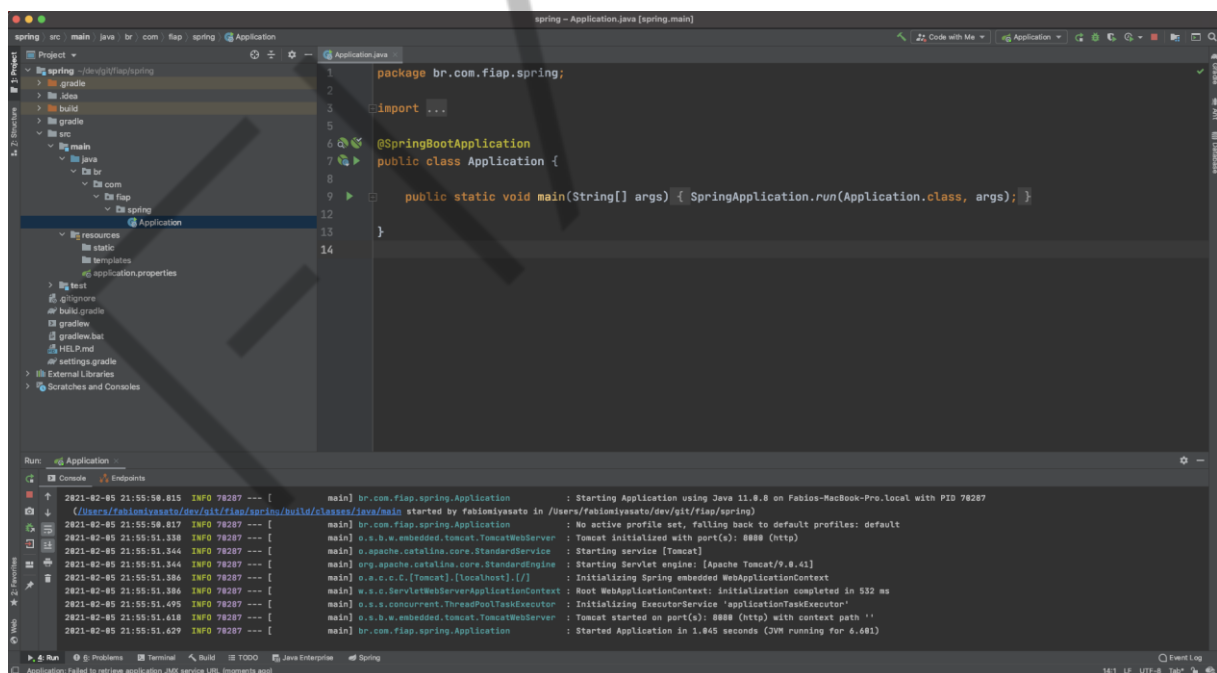


Figura 1.5 – Apresentação do projeto inicial gerado pelo Spring Initializr
Fonte: Elaborado pelo autor (2021)

1.3.4 Configurações externalizadas

O projeto traz como padrão o arquivo `application.properties`, onde é possível configurar variáveis (chave e valor) para a aplicação. O Spring aceita que essas variáveis sejam configuradas em arquivos de propriedades (`.properties`) ou em arquivos no formato YAML, bastando apenas renomear o arquivo de `application.properties` para `application.yml`.



Figura 1.6 – Configurações em formatos de propriedades (esquerda) e YAML (direita)
Fonte: Elaborado pelo autor (2021)

Conforme a imagem “Configurações em formatos de propriedades (esquerda) e YAML (direita)”, o formato YAML evita a repetição e já é padrão para diversos arquivos de configuração de infraestrutura, por exemplo, Docker.

As variáveis de ambiente do Spring seguem uma hierarquia e essa é a forma mais simples de definir variáveis em tempo de desenvolvimento. Os exemplos deste capítulo utilizarão o formato YAML como linguagem.

Para testar, pode-se substituir a porta padrão exposta pela aplicação (8080) com o seguinte código:

```
server:
  port: 8088
```

Código-fonte 1.1 – Alteração da porta padrão de execução do servidor
Fonte: Elaborado pelo autor (2021)

Após a execução da aplicação, será exibida, no console, a mensagem a seguir que indica o início do serviço na porta 8088.

```
w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 489 ms
o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8088 (http) with context path ''
br.com.fiap.spring.Application           : Started Application in 1.013 seconds (JVM running for 6.466)
```

Figura 1.7 – Console do IntelliJ com mensagem de sucesso na execução da aplicação
Fonte: Elaborado pelo autor (2021)

1.4 Rest Controller

O projeto criado utiliza uma arquitetura em camadas. A primeira camada, chamada Controller, é responsável pela interface da API e define quais são as “assinaturas”, como: quais os recursos, parâmetros de entrada e retorno de cada endpoint.

Essa camada da aplicação utiliza objetos específicos de comunicação e transferência de dados, denominados Data Transfer Objects ou DTOs. No contexto do Springs, os DTOs são objetos Java simples também conhecidos como POJOs (Plain Old Java Objects).

A aplicação, criada neste módulo, tem como recursos (resource) livros e usuários, define os atributos e cria a classe LivroDTO dentro do package models.dto.

```
package br.com.fiap.spring.model.dto;

import java.util.Date;

public class LivroDTO {

    private int id;

    private String titulo;

    private String descricao;

    private String ISBN;

    private Date dataDePublicacao;

    private double preco;

    // getters and setters

}
```

Código-fonte 1.2 – Estrutura da classe LivroDTO
Fonte: Elaborado pelo autor (2021)

Para criar o Controller, basta criar a classe LivroController e utilizar a anotação `@RestController`; também podemos utilizar a anotação `@RequestMapping` para definir o path do nosso recurso, no caso “livros”.

```
@RestController

@RequestMapping("livros")

public class LivroController {

}
```

Código-fonte 1.3 – Estrutura da classe LivroController
Fonte: Elaborado pelo autor (2021)

O Spring interpreta essa anotação e, ao executar a aplicação, gera uma instância da classe LivroController. Para criar os endpoints, é só criar métodos e utilizar as anotações “mapping” de acordo com o método (verbo) http do endpoint. Por exemplo, para criar um GET e obter a lista de livros, a anotação é `@GetMapping`.

```
@GetMapping

public List<LivroDTO> buscarLivros() {

    LivroDTO livroDTO = new LivroDTO();

    livroDTO.setId(1);

    livroDTO.setTitulo("Aprenda Spring");

    livroDTO.setDescricao("Passo a passo com Spring Framework");

    livroDTO.setDataDePublicacao(new Date());

    livroDTO.setISBN("938472389472393482");

    livroDTO.setPreco(20.4);

    return Arrays.asList(livroDTO);

}
```

Código-fonte 1.4 – Estrutura do método buscarLivros
Fonte: Elaborado pelo autor (2021)

Ao executar a aplicação, agora é disponibilizado o método GET no recurso livros. Para testar, digite no browser: <http://localhost:8088/livros>

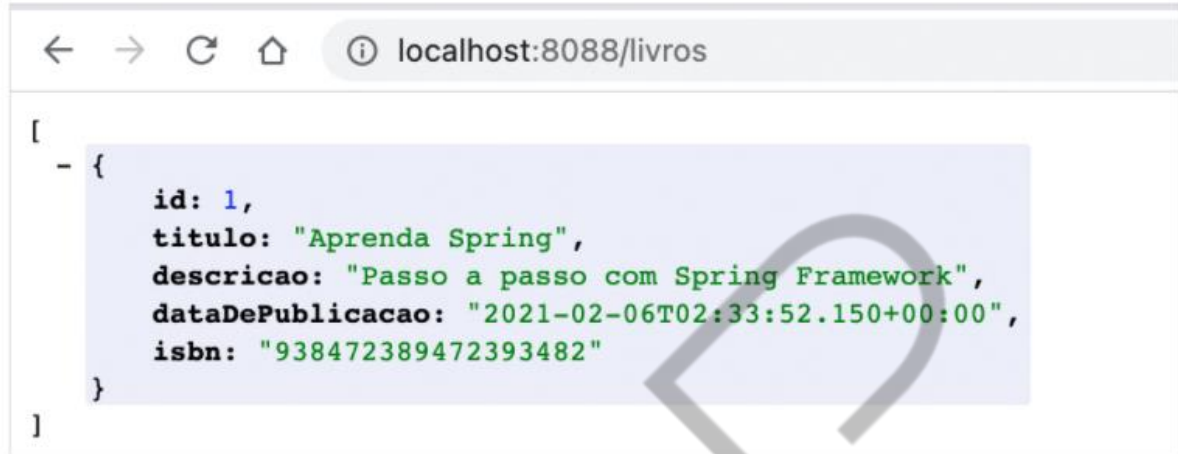


Figura 1.8 – Resultado da consulta de livros via REST
Fonte: Elaborado pelo autor (2021)

Por padrão, o Spring utiliza JSON como tipo de retorno. Na criação do primeiro endpoint, já é possível notar como a filosofia “just run” é aplicada no framework. Em pouco tempo, configurações e classes já temos um servidor rodando com a aplicação e disponibilizando um endpoint.

1.4.1 Parâmetro - Query Parameter

Query Parameter ou Query String é um dos parâmetros que uma aplicação Rest aceita. Esse tipo de parâmetro funciona na estrutura de chave-valor pela url e tem como principais funções filtrar e ordenar o recurso na API. Podem ser passados múltiplos query parameters utilizando “&”.

O Query Parameter vai na URL depois de um sinal de ponto de interrogação “?” e não faz parte da URL. Por exemplo, para filtrar os livros por título, a URL fica:

GET => <http://localhost:8088/livros?titulo=Spring>

Esse parâmetro é representado no Spring pela anotação `@RequestParam` e deve ser colocado como um dos parâmetros do método. Como a aplicação ainda não tem nenhum tipo de persistência de dados, podemos fazer o filtro por meio da função `filter` da Streams API.

```
@GetMapping
public List<LivroDTO> buscarLivros(@RequestParam(required = false,
value = "titulo") String titulo){
    List<LivroDTO> livroDTOList = new ArrayList<>();

    LivroDTO livroDTO = new LivroDTO();

    livroDTO.setId(1);

    livroDTO.setTitulo("Aprenda Spring");

    livroDTO.setDescricao("Passo a passo com Spring Framework");

    livroDTO.setDataDePublicacao(new Date());

    livroDTO.setISBN("938472389472393482");

    livroDTOList.add(livroDTO);

    livroDTO.setPreco(15.5);

    LivroDTO livroDTO1 = new LivroDTO();

    livroDTO1.setId(2);

    livroDTO1.setTitulo("Java");

    livroDTO1.setDescricao("Tudo sobre Java");

    livroDTO1.setDataDePublicacao(new Date());

    livroDTO1.setISBN("9548675464588");

    livroDTO1.setPreco(20.4);

    livroDTOList.add(livroDTO1);

    return livroDTOList.stream()

        .filter(dto -> titulo == null || dto.getTitulo().contains(titulo))

        .collect(Collectors.toList());
}
```

```
}
```

Código-fonte 1.5 – Estrutura da Query Parameter
Fonte: Elaborado pelo autor (2021)

1.4.2 Parâmetro - Path Parameter

O Path Parameter é um parâmetro de especificação de um recurso e é único por recurso. Esse parâmetro faz parte da URL, seguindo o exemplo anterior. Para especificar exatamente qual livro será retornado, a URL fica:

GET => <http://localhost:8088/livros/1>

Nesse caso, está sendo especificado que o recurso a ser retornado é o livro “1”. Logo, não será retornada uma lista de livros, e sim um único objeto.

O Spring permite especificar o path do endpoint na anotação mapping. Como o path parameter faz parte da URL do endpoint, ele também pode ser inserido nessa anotação dentro de “chaves” (“{“ e “}”). Além disso, no parâmetro do método é necessário incluir a anotação `@PathVariable`.

```
@GetMapping("/{id}")
public LivroDTO buscarPorId(@PathVariable int id) {
    List<LivroDTO> livroDTOList = new ArrayList<>();

    LivroDTO livroDTO = new LivroDTO();

    livroDTO.setId(1);

    livroDTO.setTitulo("Aprenda Spring");

    livroDTO.setDescricao("Passo a passo com Spring Framework");

    livroDTO.setDataDePublicacao(new Date());

    livroDTO.setISBN("938472389472393482");

    livroDTO.setPreco(20.4);

    livroDTOList.add(livroDTO);
}
```

```
LivroDTO livroDTO1 = new LivroDTO();

livroDTO1.setId(2);

livroDTO1.setTitulo("Java");

livroDTO1.setDescricao("Tudo sobre Java");

livroDTO1.setDataDePublicacao(new Date());

livroDTO1.setISBN("9548675464588");

livroDTO1.setPreco(15.3);

livroDTOList.add(livroDTO1);

switch(id) {

    case 1:

        return livroDTO;

    case 2:

        return livroDTO1;

    default:

        throw new ResponseStatusException(HttpStatus.NOT_FOUND,

            "Livro não encontrado");

}
```

Código-fonte 1.6 – Estrutura do método buscarPorId
Fonte: Elaborado pelo autor (2021)

O switch case simula o comportamento esperado, retornando o livro do id especificado pelo parâmetro. Caso seja um id diferente dos existentes, é retornado o http code 404 por meio do lançamento da exceção do tipo `ResponseStatusException`, interpretada pelo Spring para mapeamento de erros.

1.4.3 Parâmetro - Request Body

Diferente dos anteriores, o Request Body não é definido na URL e sim no “corpo” da requisição. Ele permite utilizar diferentes formatos, incluindo o JSON. Esse

tipo de parâmetro só pode ser aplicado em requisições com os métodos http POST, PUT e PATCH, ou seja, será utilizado para cadastrar ou alterar os recursos.

POST => http://localhost:8088/livros

```
{
    titulo: "React Avançado",
    descricao: "Tudo sobre React",
    dataDePublicacao: "2021-02-03T02:33:52.150+00:00",
    isbn: "9655236748347654",
    preco: 20.5
}
```

Para o método de cadastro, é uma boa prática criar um DTO apenas com os campos que serão cadastrados ou atualizados. Por exemplo, se a aplicação gerencia o id, esse campo deve ser retirado do DTO de cadastro.

```
public class CreateUpdateLivroDTO {
    private String titulo;
    private String descricao;
    private String ISBN;
    private Date dataDePublicacao;
    private double preco;

    // getters and setters
}
```

Código-fonte 1.7 – Estrutura da classe CreateUpdateLivroDTO
Fonte: Elaborado pelo autor (2021)

Ao incluir a anotação `@RequestBody` no parâmetro, o Spring realiza o “parse” (transformação) de JSON para o objeto DTO de forma transparente, utilizando a biblioteca Jackson.

```
@PostMapping
@ResponseStatus(HttpStatus.CREATED)

public LivroDTO criar(@RequestBody CreateUpdateLivroDTO
createUpdateLivroDTO) {

    LivroDTO livroDTO = new LivroDTO();

    livroDTO.setId(3);

    livroDTO.setTitulo(createUpdateLivroDTO.getTitulo());

    livroDTO.setDescricao(createUpdateLivroDTO.getDescricao());

    livroDTO.setISBN(createUpdateLivroDTO.getISBN());

    livroDTO.setPreco(createUpdateLivroDTO.getPreco());

    livroDTO.setDataDePublicacao(createUpdateLivroDTO.getDataDePublicacao());

    return livroDTO;

}
```

Código-fonte 1.8 – Estrutura do método criar
Fonte: Elaborado pelo autor (2021)

Para retornar o status code 201 (created) a anotação é `@ResponseStatus`, em caso de sucesso no método ou caso nenhuma exceção ocorra. Conforme o padrão REST, o cadastro de um recurso deve retorná-lo completo, no caso desse endpoint com o campo id preenchido.

O método de atualização segue o mesmo princípio, porém é preciso utilizar um path parameter, conforme o tópico anterior para especificar qual recurso será atualizado.

```
@PutMapping("/{id}")

public LivroDTO atualizar(

    @PathVariable int id,
```

```
@RequestBody CreateUpdateLivroDTO createUpdateLivroDTO

){

    LivroDTO livroDTO = new LivroDTO();

    livroDTO.setId(id);

    livroDTO.setTitulo(createUpdateLivroDTO.getTitulo());

    livroDTO.setDescricao(createUpdateLivroDTO.getDescricao());

    livroDTO.setISBN(createUpdateLivroDTO.getISBN());

    livroDTO.setPreco(createUpdateLivroDTO.getPreco());

    livroDTO.setDataDePublicacao(createUpdateLivroDTO.getDataDePublicacao());

    return livroDTO;

}
```

Código-fonte 1.9 – Estrutura do método atualizar
Fonte: Elaborado pelo autor (2021)

O método patch é aplicado em casos em que é necessário atualizar apenas parte do recurso. Um bom exemplo no caso da API de livro é a atualização de preço. O método patch é necessário porque o método put deve, por definição, atualizar o objeto inteiro. Criar um DTO é necessário para manter a consistência das assinaturas dos métodos.

```
public class UpdatePrecoLivroDTO {

    private double preco;

    public double getPreco() {

        return preco;

    }

    public void setPreco(double preco) {

        this.preco = preco;

    }

}
```

Código-fonte 1.10 – Estrutura da classe UpdatePrecoLivroDTO
Fonte: Elaborado pelo autor (2021)

O método patch precisa de um path parameter e pode conter mais alguma especificação de path que, apesar de não ser necessária, torna o entendimento da API mais fácil.

```
@PatchMapping("{id}/preco")

public LivroDTO atualizarPreco(

    @PathVariable int id,

    @RequestBody UpdatePrecoLivroDTO updatePrecoLivroDTO

){

    LivroDTO livroDTO = new LivroDTO();

    livroDTO.setId(id);

    livroDTO.setPreco(updatePrecoLivroDTO.getPreco());

    return livroDTO;

}
```

Código-fonte 1.11 – Estrutura do método atualizarPreco
Fonte: Elaborado pelo autor (2021)

1.5 Injeção de dependência

Os containers de injeção de dependência (Bean Factory e Application Context) do Spring são a característica fundamental para o funcionamento do framework. Injeção de dependência é um design pattern que implementa o princípio de inversão de controle (IoC) em que a responsabilidade do gerenciamento de instância da classe é delegada ao framework.

De forma resumida, por meio de algumas configurações, (anotações) são delegados ao Spring o controle e o gerenciamento de instâncias das classes. Uma das formas de fazer isso é por meio da anotação `@RestController`, como foi feito no tópico anterior.

Como o gerenciamento da instância é de responsabilidade do Spring, é importante conhecer os escopos das instâncias do Bean. Como este módulo trata

apenas de uma aplicação não “web-aware”, os escopos relevantes para a aplicação são singleton e prototype.

Scope	Description
singleton	(Default) Scopes a single bean definition to a single object instance for each Spring IoC container.
prototype	Scopes a single bean definition to any number of object instances.
request	Scopes a single bean definition to the lifecycle of a single HTTP request. That is, each HTTP request has its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring ApplicationContext.
session	Scopes a single bean definition to the lifecycle of an HTTP Session. Only valid in the context of a web-aware Spring ApplicationContext.
application	Scopes a single bean definition to the lifecycle of a ServletContext. Only valid in the context of a web-aware Spring ApplicationContext.
websocket	Scopes a single bean definition to the lifecycle of a WebSocket. Only valid in the context of a web-aware Spring ApplicationContext.

Figura 1.9 – Escopos de instância do Bean
Fonte: Spring Documentation (2021)

1.6 Bean

Para criar um objeto gerenciado, é possível utilizar a anotação `@Bean` em um método. Esse método deve retornar uma instância de objeto, que será delegada ao contexto do Spring. Esse tipo de Bean pode ficar em qualquer classe do contexto do Spring; como forma de organização, pode ser criada uma classe do tipo “configuration”. Classes com a anotação `@Configuration` indicam para o Spring que as classes contêm métodos bean.

```
@Configuration
public class AppConfiguration {

    @Bean

    public LivroValidator livroValidator() {

        return new LivroValidatorImpl();

    }

}
```



```
}
```

Código-fonte 1.12 – Estrutura da classe AppConfiguration
Fonte: Elaborado pelo autor (2021)

1.6.1 Conditional Bean

A criação de instâncias dos Beans pode ser condicional, utilizando diversos critérios, como: outro bean, uma classe, plataforma cloud, expressão, versão jvm, jndi, missing (bean ou classe), property, resource, entre outros. Para utilizar esse recurso, é necessário incluir as anotações do tipo `@Conditional`. Mais detalhes desse recurso podem ser consultados em: <https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-condition-annotations>.

```
@RestController
@RequestMapping("debug")
@ConditionalOnProperty(value = "fiap.debug", havingValue = "true")
public class DebugController {
    private Environment env;

    public DebugController(Environment env) {
        this.env = env;
    }

    @GetMapping
    public String getProperty(
        @RequestParam String property
    ) {
        return env.getProperty(property);
    }
}
```

```
}
```

Código-fonte 1.13 – Estrutura da classe DebugController
Fonte: Elaborado pelo autor (2021)

Na classe anterior, o recurso “debug” é utilizado para exibir alguma propriedade da aplicação. A anotação `@ConditionalOnProperty` garante que esse Bean (ou controller) só será instanciado se houver uma propriedade na aplicação “fiap.debug” com o valor true.

Além desse tipo de uso, esse recurso é bastante útil para desenvolvimento de bibliotecas. O próprio Spring trabalha bastante com esse tipo de instância baseada em configuração.

1.6.2 Service

Na arquitetura em camadas proposta, as regras de negócio são distribuídas nas classes service ou business objects. A classe service também é um Bean gerenciado e possui uma anotação específica chamada `@service`, que é uma variação de `@component`. A única diferença entre elas é o nome usado para facilitar a identificação em tempo de desenvolvimento.

Conforme as boas práticas do SOLID e o princípio da inversão de dependência, não deve depender de uma classe concreta e sim de uma abstração.

```
public interface LivroService {  
  
    List<LivroDTO> buscarLivros(String titulo);  
  
    LivroDTO buscarPorId(int id);  
  
    LivroDTO criar(CreateUpdateLivroDTO createUpdateLivroDTO);  
  
    LivroDTO atualizar(CreateUpdateLivroDTO stockCreateUpdateDTO, int id);  
  
    LivroDTO atualizar(UpdatePrecoLivroDTO updatePrecoLivroDTO, int id);  
  
    void delete(int id);  
}
```

```
}
```

Código-fonte 1.14 – Estrutura da classe LivroService
Fonte: Elaborado pelo autor (2021)

Uma dica para gerar a implementação dessa interface é: clicar com o botão direito do mouse e “implementar interface”, a classe concreta já é gerada com o nome correto LivroServiceImpl. Tanto a interface quanto a implementação ficam no package service.

```
@Service

public class LivroServiceImpl implements LivroService {

    // métodos implementados

}
```

Código-fonte 1.x – Estrutura da classe LivroServiceImpl
Fonte: Elaborado pelo autor (2021)

Não é necessário implementar os métodos ainda, mas já podemos configurar a dependência na classe controller. A forma recomendada pelo Spring para definir uma dependência é pelo construtor. Se um Bean possui parâmetros no construtor, o Spring procura por instâncias desse objeto; então é necessário se atentar para que não haja dependências cíclicas.

```
private LivroService livroService;

public LivroController(LivroService livroService){

    this.livroService = livroService;

}
```

Código-fonte 1.15 – Estrutura da classe livroService
Fonte: Elaborado pelo autor (2021)

1.7 Spring Data

O objetivo do Spring Data é prover um modelo consistente e baseado em modelos orientados a objetos de persistência de dados. Simplificando, dessa maneira, o acesso a diferentes tecnologias de dados, bancos relacionais e não-relacionais entre outras formas.

Spring Data é um projeto “guarda-chuva” que contém diversos subprojetos para diferentes necessidades. É um dos poucos projetos que contém geração dinâmica de código, porém as queries derivadas do nome do método são uma das features mais interessantes. Além disso, oferece auditoria de dados e repositórios poderosos com diversas queries prontas.

O projeto deste módulo utiliza o banco de dados embarcado H2, então não é necessário ter nenhum banco de dados configurado. Porém, a implementação realizada serve para qualquer banco SQL como MySQL ou Postgres.

1.7.1 Configuração

Sempre que for necessário adicionar uma dependência, basta acrescentá-la no arquivo build.gradle (substituto do pom.xml) dentro da chave dependencies.

```
dependencies {  
    implementation 'org.springframework.boot:spring-boot-starter-web'  
    implementation 'org.springframework.boot:spring-boot-starter-data-jpa'  
    implementation 'com.h2database:h2:1.4.197'  
    testImplementation 'org.springframework.boot:spring-boot-starter-test'  
}
```

Código-fonte 1.16 – Exemplo de inclusão de dependências com Gradle
Fonte: Elaborado pelo autor (2021)

Toda vez que esse arquivo é alterado, é necessário importar a dependência por meio do comando “build gradle” ou do botão “load gradle changes”. É possível conferir se as dependências foram corretamente importadas na aba “Gradle” do IntelliJ:

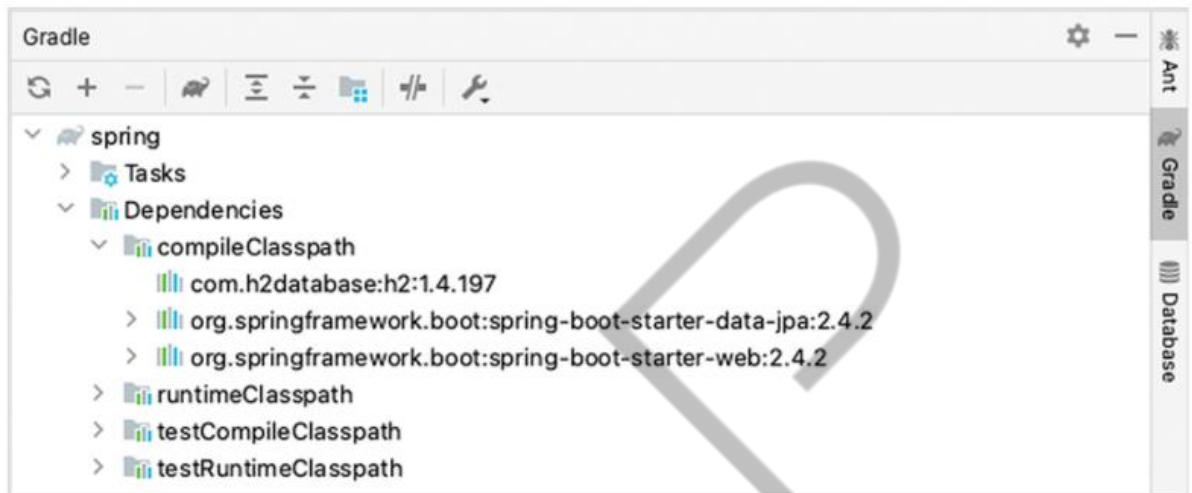


Figura 1.10 – Hierarquia de dependências do Gradle.
Fonte: Elaborado pelo autor (2021)

Para configurar os dados de acesso ao banco, basta incluir essas propriedades no arquivo `application.yml`.

```
spring:
  datasource:
    url: jdbc:h2:/caminho_do_arquivo/fiapspring;DB_CLOSE_ON_EXIT=FALSE
    username: fiap
    password: fiap
```

Essas propriedades configuram um banco de dados H2 que será persistido em um arquivo no caminho especificado. O usuário e a senha são gerados no momento de criação da base e podem ser utilizados para acessar o banco.

1.7.2 Entidade

O próximo passo é mapear a entidade que será persistida no banco de dados. Estamos utilizando o subprojeto spring data jpa (na versão spring boot starter); então o mapeamento da entidade segue a especificação JPA, utilizando o package `javax.persistence`.

```
@Entity
@Table(name = "TB_LIVRO")
public class Livro {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column
    private String titulo;

    @Column
    private String descricao;

    @Column
    private String ISBN;

    @Column
    private Date dataDePublicacao;

    // getters and setters

}
```

Código-fonte 1.17 – Estrutura da classe Livro
Fonte: Elaborado pelo autor (2021)

1.7.3 Repository

A camada repository é responsável pelo acesso à estrutura de banco de dados. No Spring, basta declarar uma interface estendida de JpaRepository e parametrizar a entidade envolvida e a classe da chave primária.

```
public interface LivroRepository extends JpaRepository<Livro, Integer> {  
    }  
}
```

Essa herança já traz os métodos para as operações básicas (CRUD). Para queries customizadas, existem 3 formas: JPQL, nome do método e nativa. As boas práticas dizem que as queries nativas não devem ser utilizadas, pois causam dependência da implementação do banco de dados; então, apesar de não detalhar mais as queries nativas, é importante saber que elas existem.

JPQL é uma forma segura e genérica de fazer as queries, além de possuir uma ampla documentação fornecida pela própria Oracle (https://docs.oracle.com/html/E13946_04/ejb3_langref.html). No caso da busca do livro por título, a query em JPQL fica:

```
@Query("from Livro l " +  
        "where l.titulo like %:titulo%")  
List<Livro> buscaPorTitulo(String titulo);
```

Essa mesma query escrita em “method name”:

```
List<Livro> findAllByTituloLike(String titulo);
```

Vale ressaltar que nenhum desses métodos precisa ser implementado, apesar de ambos estarem em uma interface. Em tempo de compilação, o Spring Data gera as classes necessárias e, no caso de uma query incorreta, ocorre, portanto, um erro

de compilação. Por exemplo, ao trocar “título” por um atributo inexistente, como “nome”, é lançada uma exceção em tempo de compilação:

```
@Query("from Livro l " +
      "where l.nome like %:nome%")

List<Livro> buscaPorTitulo(String nome);
```

```
Error starting ApplicationContext. To display the conditions report re-run your application with 'debug' enabled.
2021-02-06 22:42:49.497 ERROR 4942 --- [main] o.s.boot.SpringApplication : Application run failed

org.springframework.beans.factory.BeanCreationException: Create breakpoint : Error creating bean with name 'livroRepository' defined in br.com.fiap.spring.repository.LivroRepository
defined in @EnableJpaRepositories declared on JpaRepositoriesRegistrar.EnableJpaRepositoriesConfiguration: Invocation of init method failed; nested exception is java.lang
.IllegalArgumentException: Validation failed for query for method public abstract java.util.List br.com.fiap.spring.repository.LivroRepository.buscaPorTitulo(java.lang.String)!
at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.initializeBean(AbstractAutowireCapableBeanFactory.java:1788) ~[spring-beans-5.3.3.jar:5.3.3]
at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.doCreateBean(AbstractAutowireCapableBeanFactory.java:602) ~[spring-beans-5.3.3.jar:5.3.3]
at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.createBean(AbstractAutowireCapableBeanFactory.java:531) ~[spring-beans-5.3.3.jar:5.3.3]
at org.springframework.beans.factory.support.AbstractBeanFactory.lambda$doGetBean$0(AbstractBeanFactory.java:335) ~[spring-beans-5.3.3.jar:5.3.3]
at org.springframework.beans.factory.support.DefaultSingletonBeanRegistry.getSingleton(DefaultSingletonBeanRegistry.java:234) ~[spring-beans-5.3.3.jar:5.3.3]
```

Figura 1.11 – Simulação de exceção com parâmetros inexistentes

Fonte: Elaborado pelo autor (2021)

Com o repositório devidamente configurado, pode-se integrar a classe service, utilizando a injeção de dependência por construtor.

```
private final LivroRepository livroRepository;

public LivroServiceImpl(LivroRepository livroRepository) {

    this.livroRepository = livroRepository;
}
```

Código-fonte 1.18 – Estrutura da classe livroRepository

Fonte: Elaborado pelo autor (2021)

Com o repositório devidamente configurado, ele pode ser integrado à classe service, conforme código a seguir.

```
@Override

public List<LivroDTO> buscarLivros(String titulo) {

    List<Livro> livroList;
```



```
        if (titulo != null) {

            livroList = livroRepository.findAllByTituloLike("%" + titulo
+ "%");

        } else {

            livroList = livroRepository.findAll();

        }

        return livroList.stream()

            .map(livro -> new LivroDTO(livro))

            .collect(Collectors.toList());

    }

    @Override

    public LivroDTO buscarPorId(int id) {

        Livro livro = findLivroById(id);

        return new LivroDTO(livro);

    }

    private Livro findLivroById(int id) {

        Livro livro = livroRepository.findById(id)

            .orElseThrow(() -> new

ResponseStatusException(HttpStatus.NOT_FOUND));

        return livro;

    }

    @Override

    public LivroDTO criar(CreateUpdateLivroDTO createUpdateLivroDTO) {

        Livro livro = new Livro(createUpdateLivroDTO);

        Livro savedLivro = livroRepository.save(livro);

        return new LivroDTO(savedLivro);

    }
```

```
@Override

public LivroDTO atualizar(CreateUpdateLivroDTO stockCreateUpdateDTO,
int id) {

    Livro livro = findLivroById(id);

    livro.setTitulo(stockCreateUpdateDTO.getTitulo());

    livro.setDescricao(stockCreateUpdateDTO.getDescricao());

    livro.setDataDePublicacao(stockCreateUpdateDTO.getDataDePublicacao());

    livro.setISBN(stockCreateUpdateDTO.getISBN());

    livro.setPreco(stockCreateUpdateDTO.getPreco());

    Livro savedLivro = livroRepository.save(livro);

    return new LivroDTO(savedLivro);

}

@Override

public LivroDTO atualizarPreco(UpdatePrecoLivroDTO
updatePrecoLivroDTO, int id) {

    Livro livro = findLivroById(id);

    livro.setPreco(updatePrecoLivroDTO.getPreco());

    Livro savedLivro = livroRepository.save(livro);

    return new LivroDTO(savedLivro);

}

@Override

public void delete(int id) {

    Livro livro = findLivroById(id);

    livroRepository.delete(livro);

}
```

```
}
```

Código-fonte 1.19 – Métodos de serviços utilizando o repositório configurado
Fonte: Elaborado pelo autor (2021)

1.7.4 Versionamento de banco de dados – Flyway

Flyway é uma biblioteca open source de versionamento de banco de dados que se encaixa perfeitamente no conceito de micro serviços e sistemas distribuídos, além de ser totalmente compatível com o Spring.

O primeiro passo para configurar o Flyway é incluir a dependência a seguir.

```
implementation 'org.flywaydb:flyway-maven-plugin:4.0.3'
```

Por padrão, o Flyway procura os scripts de versionamento do banco de dados na pasta db/migrations dentro do classpath. No Spring, podemos acrescentar no classpath ao criar esse diretório dentro da pasta resources. O Flyway aceita arquivos SQL, basta que sigam o padrão de nomenclatura:

V1_000__nome_da_migracao.sql

Após o “V”, fica a numeração. No caso do exemplo, é 1.000 seguido de 02 (dois) underlines e o nome da migração. Todos esses dados ficam armazenados na tabela flyway_schema_history.



installed_rank	version	description	type	script	checksum	installed_by	installed_on
1	<1 null>	<1 Flyway Schema History table created >	TABLE			FIAP	2021-02-07 00:54:01.281000
2	1.000	create table livro	SQL	V1_000__create_table_livro.sql	186699362	FIAP	2021-02-07 00:54:01.236000

Figura 1.12 – Escopos de instância do Bean
Fonte: Elaborado pelo autor (2021)

Para a aplicação de livros, o nome do arquivo será V1_000_create_table_livro.sql, e contém o script a seguir.

```
create table TB_LIVRO
```

```
(  
    id                bigint generated by default as identity,  
    titulo            varchar(100),  
    descricao         varchar(255),  
    preco             decimal(19,2),  
    isbn              varchar(20),  
    data_de_publicacao timestamp,  
    primary key (id)  
)
```

Código-fonte 1.20 – Script para geração da tabela Livro
Fonte: Elaborado pelo autor (2021)

Agora, ao executar a aplicação, o flyway vai gerenciar se o banco está na última versão e, caso não esteja, aplicará as migrações. O banco de dados de teste H2 em arquivo pode se corromper. Caso isso ocorra, basta deletar o arquivo e a aplicação criará outro.

CONCLUSÃO

Neste módulo foi criada uma API desde a configuração do projeto até a integração de todas as camadas, chegando até o banco de dados. É possível perceber as facilidades que o Spring fornece, permitindo ao desenvolvedor focar na implementação das regras de negócio.

O projeto está publicado e pode ser consultado em <https://github.com/fabiotadashi/spring/tree/modulo1>. Um ponto importante é que, apesar da apresentação dos principais pontos, o Spring Framework é muito extenso e está em constante evolução. O melhor ponto de consulta é sempre a documentação oficial. No próximo capítulo, será abordada mais uma parte do Spring muito utilizada: o Spring Batch. Vamos lá!

REFERÊNCIAS

CARNELL, John. **Spring Microservices in Action**. First Edition. Manning, 2017.

Spring Boot Documentation. Disponível em: <<https://docs.spring.io/spring-boot/docs/2.4.2/reference/html/spring-boot-features.html#boot-features-external-config/>>. Acesso em: 23 jan. 2021.

Spring Framework Documentation. Disponível em: <<https://docs.spring.io/spring-framework/docs/current/reference/html/>>. Acesso em: 25 jan. 2021.

TURNQUIST, Greg L. **Learning Spring Boot 2.0**. Second Edition. Packt Publishing, 2017.

GLOSSÁRIO

IoC	Acrônimo de Inverse of Control.
CRUD	Create, Read, Update and Delete
IDE	Integrated Development Enviroment
YAML	Trata-se de um formato de serialização de dados legíveis por humanos, inspirado em linguagens como XML, C e Python.
SOLID	SOLID é um acrônimo dos cinco primeiros princípios da programação orientada a objetos e design de código identificados por Robert C. Martin.