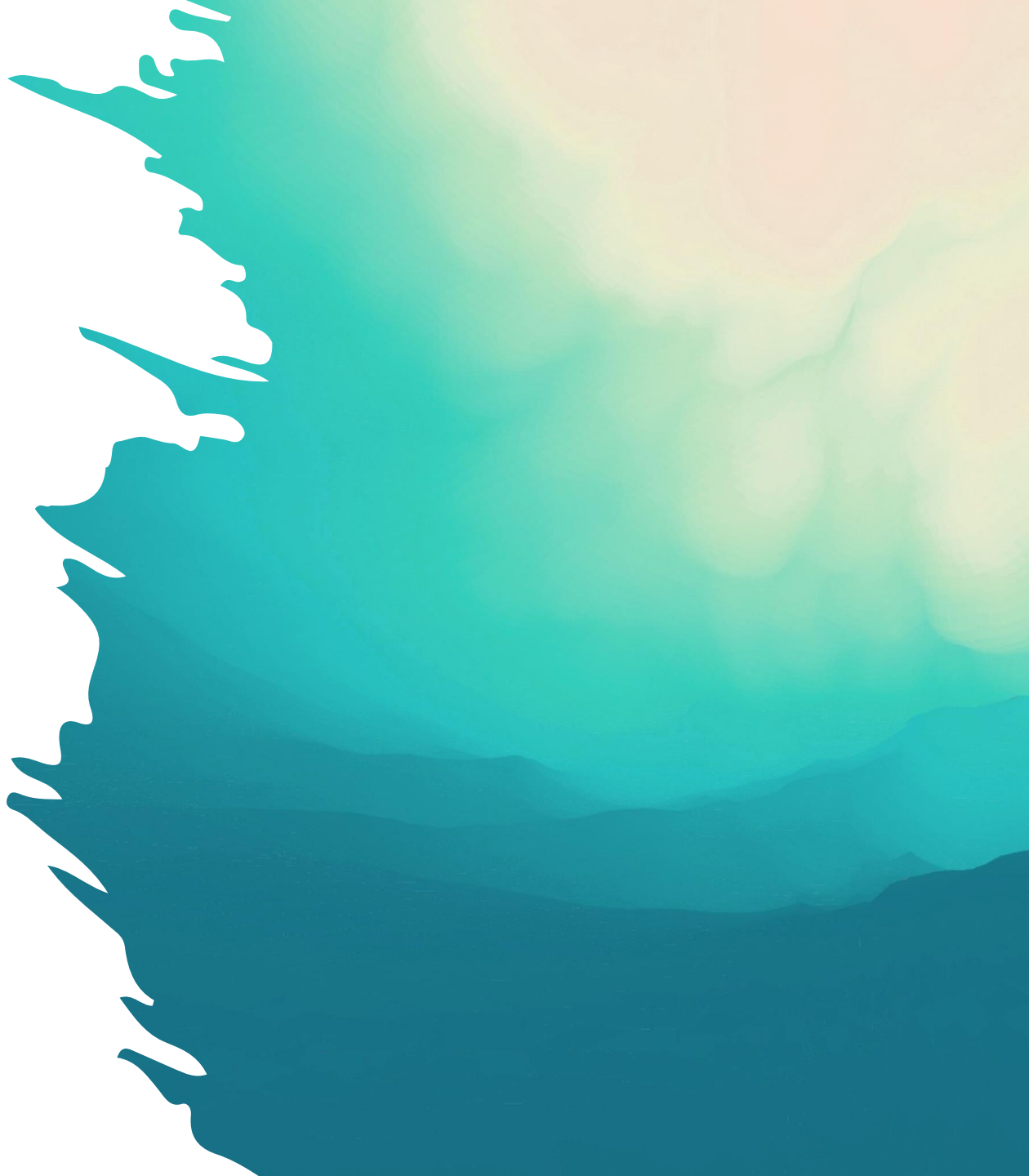
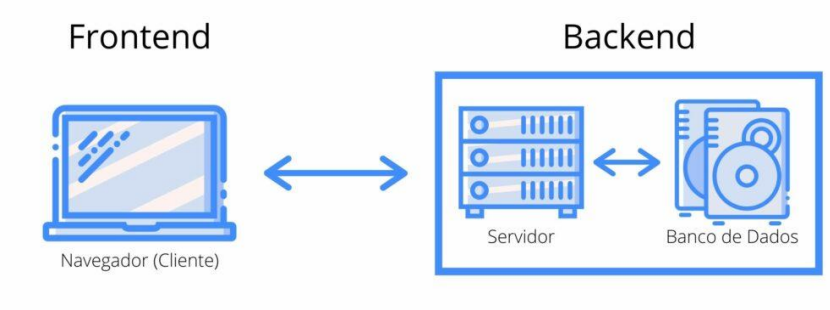
O Framework Spring-Boot

Desenvolvendo um Backend



Desenvolvendo um primeiro exemplo

- O objetivo desta apresentação é desenvolver nossa primeira aplicação “backend”
- Será basicamente um módulo que roda no servidor e fica aguardando por requisições HTTP
- Sempre que chegar uma requisição prevista, o “endpoint” correspondente será ativado para executar o código adequado para processar aquela solicitação.



MARISTA

Leitura complementar

- Material complementar a estas aulas pode ser encontrado em Spring.io.

MARISTA

O framework Spring-Boot

- O framework Spring-Boot fornece um modelo abrangente de programação e configuração para aplicações coorporativas desenvolvidas em Java para todos os tipos de plataforma.
- Principais recursos
  - Criar aplicações Spring “stand-alone”
  - Embutir diretamente Tomcat, Jetty or Undertow
  - Fornecer dependências de inicialização visando simplificar as configurações
  - Configurar automaticamente as opções do Spring e de terceiros automaticamente.
  - Fornece recursos de produção tais como metricas, “health checks”, e configurações externas

MARISTA

Pré-requisitos

- Existem diversas formas de trabalhar com Spring-Boot. Uma delas exige os seguintes softwares previamente instalados:
  - Java 17
  - Maven
  - VSCode
- Além do IDE e das ferramentas de desenvolvimento propriamente ditas, vamos precisar de uma ferramenta capaz de simplificar o envio de requisições HTTP para o nosso “backend” sem a necessidade de termos de construir um módulo “frontend”. Em nossos exemplos usaremos a ferramenta “Postman” (Postman API Platform), porém, existem diversas soluções semelhantes a disposição. Mais adiante iremos detalhar o uso do PostMan.

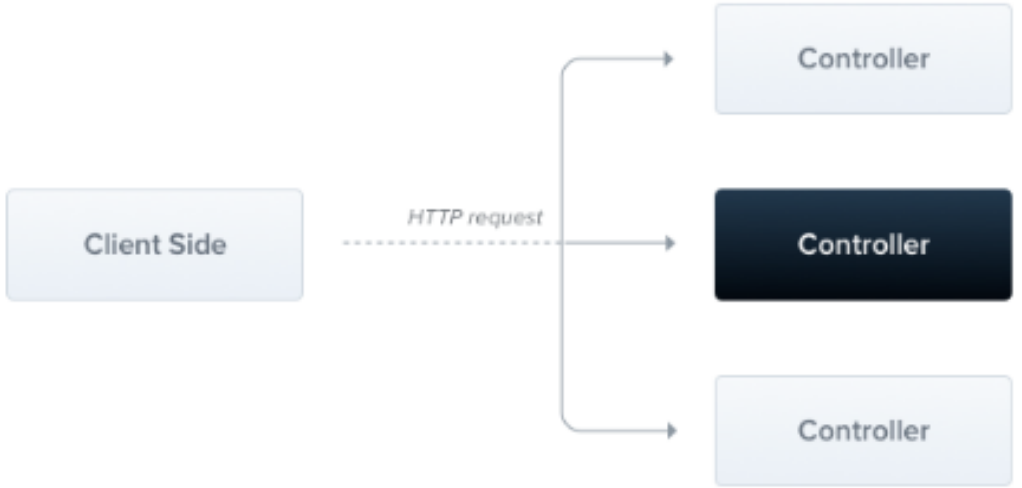
MARISTA

Revisando

- Uma aplicação “backend” não tem “interface com o usuário”
- Ao invés disso ela implementa uma série de “endpoints”
- Cada “endpoint” é acionado em resposta a um tipo diferente de requisição (no caso requisições HTTP)
- A(s) classe(s) que implementam os “endpoints” normalmente são chamadas de “controllers”.
- Os “controllers” são a forma de comunicação das aplicações “backend” com o mundo externo.

Entendendo os “controllers”

- Uma aplicação pode ter vários “controllers” de maneira a lidar com subconjuntos de requisições
- Nossas primeiras aplicações terão apenas 1 “controller”



MARISTA

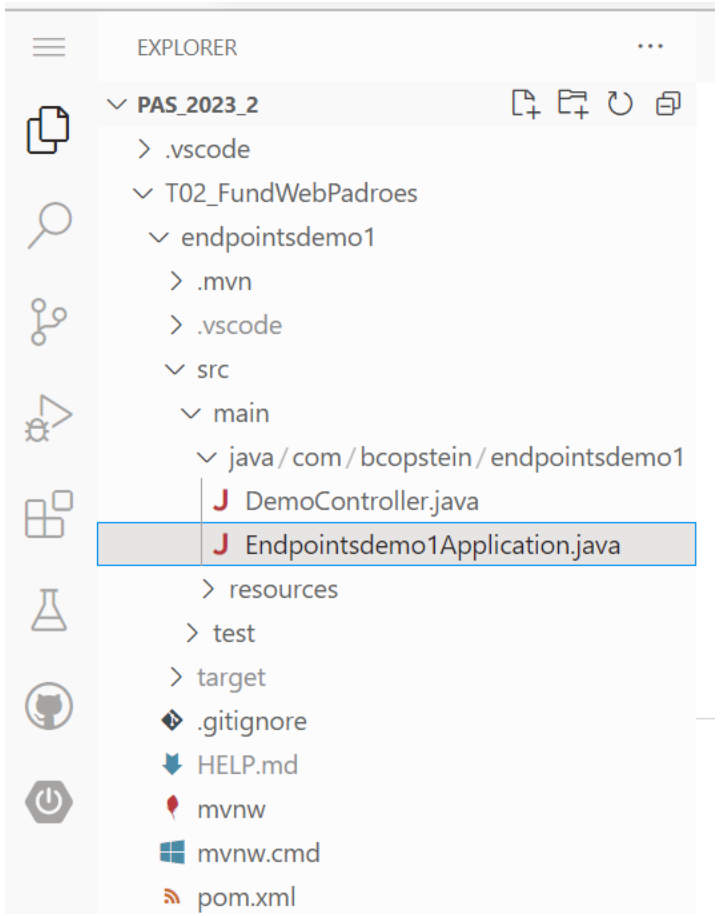
Criando o primeiro projeto com Spring-Boot

- Para criar um projeto Spring-Boot usar:
  - O Spring Initializer: Spring Initializr
  - A ferramenta de criação de projetos “Spring” do próprio VSCode
- Nos dois casos é necessário indicar uma série de informações tais como:
  - Tipo de projeto: “Maven”
  - Versão do Spring-Boot: 3.12
  - “Group Id”: com.bcopstein
  - Nome do artefato: demo1
  - Packing: jar
  - Versão do Java: 17
  - Dependências: (inicialmente usaremos apenas estas duas)
    - Spring-Boot Dev Tools
    - Spring WEB
- A ferramenta Spring Initilizer gera um zip com todo conteúdo do projeto
- A ferramenta do VSCode já gera a estrutura do projeto no local indicado.

MARISTA

Estrutura do projeto

- O projeto será criado em uma pasta com o nome do projeto: endpointsdemo1
- O programa principal (EndPointsApplication.java) será criado na pasta src.
- O arquivo “pom.xml” contém toda a descrição do projeto.



MARISTA

O Arquivo EndPointsApplication.java

```
package com.bcopstein.endpointsdemo1;


import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;


@SpringBootApplication
public class Endpointsdemo1Application {
        public static void main(String[] args) {
            SpringApplication.run(Endpointsdemo1Application.class, args);
        }
}
```

- O programa principal é responsável por toda as inicializações automáticas do Spring-Boot.
- A única anotação necessária, por enquanto, é “@SpringBootApplication” que indica o tipo de aplicação.
- Esta aplicação coloca no ar uma instancia do Tomcat, porém, não temos nenhum “endpoint” configurado para atender requisições HTTP.

Acrescentando um “controller”

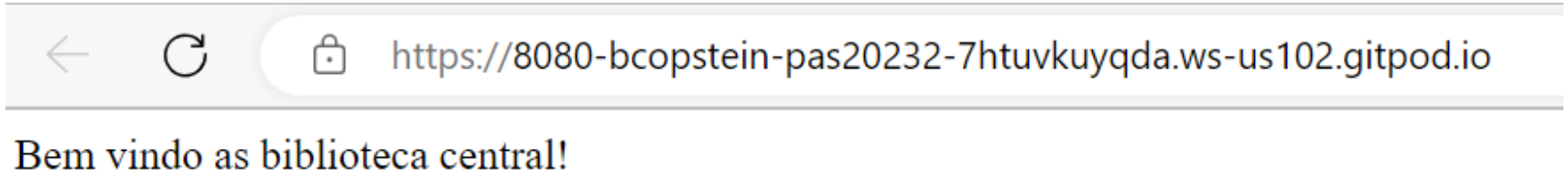
```
package com.bcopstein.endpointsdemo1;


@RestController
public class DemoController{
    @GetMapping("/")
    @CrossOrigin(origins = "*")
    public String consultaCidadesAtendidas() {
        return "Bem vindo a biblioteca central!";
    }
}
```

- O arquivo “DemoController.java” implementa um “controller”.
- A identificação da classe que implementa um “controller” é feita pela anotação “@RestController”
- Cada função anotada com “@GetMapping” indica um endpoint
- O roteamento correspondente é anotado no parâmetro.
- No exemplo ao lado não temos nenhuma rota anotada, logo este será o endpoint padrão quando nada for indicado

Executando o projeto

- Para executar o projeto localize o console na pasta raiz do projeto (aquela que tem o arquivo “pom”) e digite: mvn spring-boot:run
- O projeto irá compilar e executar. A execução começa colocando no ar uma instancia do TOMCAT e depois fica escutando a porta 8080 aguardando eventuais requisições.
- Se estiver executando o programa na máquina local, abra o navegador e acesse a URL: http://localhost:8080”
- Se estiver executando no GitPod, selecione a aba “ports” do terminar e clique no link exibido.

Visualizando a execução



Interrompendo a execução

- Para parar a execução digite “Ctrl+C” no console do terminal.
- Para garantir que a aplicação libera a porta execute: mvn spring-boot:stop

Requisições sem rotas

- Até o momento nossa aplicação só é capaz de atender uma única requisição HTTP: http://localhost:8000
- Esta é uma requisição GET simples
  - Não tem “caminhos” (path)
  - Não tem parâmetros
- Para entendermos como fazer para atender requisições mais elaboradas vamos revisar o protocolo HTTP

HTTP: introdução

- Comunicar-se com servidores e aplicativos web se dá através do protocolo *Hypertext Transfer Protocol*
  - Protocolo de nível de aplicação
  - Protocolo textual
  - Protocolo baseado em mensagens de requisição/resposta no modelo cliente/servidor
  - Protocolo sem manutenção de estado

HTTP: URLs

- Lembrando: HTTP foi originalmente criado para lidar com hipertextos
- Neste contexto uma URL *Uniform Resource Locator* foi pensada para identificar arquivos em um servidor Web
  - Ex.: http://java.sun.com/index.html
- O conjunto de elementos de uma URL, porém, permitiu que seu uso evoluísse com o tempo
- Estrutura de uma URL:**"http:" "//" host[":" port] [path["?" query]]**
  - Ex: http://java.sun.com/books/index.html?id=1322

HTTP: estrutura de uma requisição

- Uma requisição HTTP consiste de:
  - Uma linha inicial
  - Um ou mais campos de cabeçalho
  - Uma linha em branco
  - Possivelmente um corpo da mensagem
- Uma resposta HTTP consiste de:
  - Uma linha de status com seu código (ver RFC, Wikipédia) e mensagem associada
  - Um ou mais campos de cabeçalho
  - Uma linha em branco
  - Possivelmente um corpo da mensagem

HTTP: métodos

- Alguns métodos (também chamados de verbos):

GET*	Solicita um recurso ao servidor
POST*	Fornece a entrada para um comando do lado do servidor e devolve o resultado
PUT	Envia um recurso ao servidor
DELETE	Exclui um recurso do servidor
TRACE	Rastreia a comunicação com o servidor

*métodos mais utilizados para fornecer entrada de dados aos programas no lado servidor

HTTP: GET

- GET:
  - Método mais simples
  - Quantidade de dados muito limitada
    - Limite implementado nos navegadores
  - Dados acrescentados à URL após um caractere “?”, no formato “campo=valor”, separados pelo caractere “&”
    - Recebe o nome de *query-string*

  - **Ex.: http://www.biblioteca.com/livros?autor=Ze&ano=2020**

HTTP: POST

- Método mais robusto
- Quantidade de dados não é limitada como no GET
- Dados *(query-string)* enviados no corpo da requisição do protocolo
- Permite efeitos colaterais na execução no lado do servidor

- Requisição:

```
POST /index.html HTTP/1.0
Accept: text/html
If-modified-since: Sat, 29 Oct 1999 19:43:31 GMT
Content-Type: application/x-www-form-urlencoded
Content-Length: 41

Nome=NomePessoa&Idade=20&Curso=Computacao
```

- Resposta:

```
HTTP/1.1 200 OK
Date: Mon, 23 May 2005 22:38:34 GMT
Server: Apache/1.3.3.7 (Unix)  (Red-Hat/Linux)
Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT
Etag: "3f80f-1b6-3e1cb03b"
Accept-Ranges: bytes
Content-Length: 438
Connection: close
Content-Type: text/html; charset=UTF-8
```

Incluindo caminhos nos “controllers”

- **No exemplo ao lado** indicamos sub caminhos no decorador “@GetMapping”
- O caminho indicado em “**@GetMapping(‘’livros”)** é ativado pela URL que segue: http://localhost:8000/livros
- Altere e teste o projeto conforme ao lado

```
@RestController
public class DemoController{
    @GetMapping("/")
    @CrossOrigin(origins = "*")
    public String getSaudacao() {
        return "Bem vindo a biblioteca central!";
    }
    @GetMapping("/livros")
    @CrossOrigin(origins = "*")
    public String getLivros() {
        return "Lista de livros";
    }
}
```

Indicando subcaminhos

- Por vezes as aplicações possuem muitos endpoints e é necessário agrupar os mesmos em subcaminhos.
- Imagine, por exemplo, que a aplicação ao lado gerencia uma univrsidade, e que a biblioteca é apenas um dos itens.
- Então define-se um subcaminho “\biblioteca” a partir do qual concatenam-se os demais.
- Usa-se a anotação “@RequestMapping” para indicar um subcaminho.
- A partir desta alteração teste as URLs:
  - http://localhost:8080/biblioteca/”
  - http://localhost:8080/biblioteca/livros”

```
@RestController
@RequestMapping("/biblioteca")
public class DemoController{
    @GetMapping("/")
    @CrossOrigin(origins = "*")
    public String getSaudacao() {
        return "Bem vindo a biblioteca central!";
    }
    @GetMapping("/livros")
    @CrossOrigin(origins = "*")
    public String getLivros() {
        return "Lista de livros";
    }
}
```

Relembrando POO

- Vamos relembrar alguns conceitos e estruturas de programação orientada a objetos criando uma resposta melhor para o caminho “/biblioteca/livros”.
- O objetivo deste caminho é retornar a lista de livros contidos no acervo da biblioteca
- Para tanto acrescente uma lista de livros como propriedade privada da classe “DemoController” e inicialize a mesma no método construtor desta classe.
- Não se esqueça de criar uma classe que modele um livro.
- Na sequência altere o método “getLivros” para que retorne a lista de livros contida no acervo. Não se esqueça de alterar o parâmetro de retorno do método para “List<Livro>”.

Observações

- Note como o Spring-Boot automatiza a questão das respostas as requisições HTTP. Não importa se é uma string ou uma lista de objetos a resposta é convertida em um JSON automaticamente.

Dinâmica

D1) Acrescente dois novos caminhos na aplicação da biblioteca sendo desenvolvida:

a) /biblioteca/titulos → devolve a lista dos títulos pertencentes ao acervo

b) /biblioteca/autores → devolve a lista dos autores dos livros pertencentes ao acervo (sem repetição)

MARISTA