O que é Spring Boot e por que usá-lo?

O que é Spring Boot?

- **Framework para desenvolvimento rápido de aplicações Java:** Simplifica a criação de aplicações prontas para produção, com foco em microsserviços.
- **Parte do ecossistema Spring:** Aproveita as funcionalidades do Spring Framework, como injeção de dependências, Web MVC, Spring Data, etc.

O que é Spring Boot?

- **Convenção sobre configuração:** Minimiza a necessidade de configurações manuais, adotando convenções sensatas.
- **Servidor embutido:** Facilita o deploy, eliminando a necessidade de configurar um servidor externo.

O que é Spring Boot?

- **"Starters" para dependências:** Agiliza a inclusão de bibliotecas e funcionalidades comuns.
- **Autoconfiguração:** Configura automaticamente componentes com base nas dependências do projeto.
- **Actuator para monitoramento:** Oferece endpoints para monitorar a saúde e métricas da aplicação.

Por que usar Spring Boot?

- **Produtividade:** Reduz o tempo e o esforço de desenvolvimento, permitindo focar na lógica de negócio.
- **Simplicidade:** Elimina a complexidade da configuração e do setup do ambiente.

Por que usar Spring Boot?

- **Facilidade de deploy:** Gera aplicações autossuficientes, fáceis de implantar em qualquer ambiente.
- **Ecossistema rico:** Ampla comunidade, documentação e suporte.

Por que usar Spring Boot?

- **Microsserviços:** Ideal para construir arquiteturas de microsserviços escaláveis e resilientes.
- **Popularidade:** Uma das tecnologias mais utilizadas no mercado para desenvolvimento Java.

Exemplos de uso

- **Aplicações web:** APIs REST, sites, sistemas web.
- **Microsserviços:** Arquiteturas distribuídas e escaláveis.
- **Aplicações de linha de comando:** Ferramentas, scripts, jobs.
- **Aplicações reativas:** Sistemas de alta performance e escalabilidade.

Público-alvo

- Desenvolvedores Java que desejam criar aplicações de forma rápida e eficiente.
- Equipes que buscam simplificar o desenvolvimento e o deploy de aplicações.
- Empresas que precisam de soluções escaláveis e de alta performance.

Em resumo

- Spring Boot é um framework que facilita a criação de aplicações Java modernas, produtivas e escaláveis. Ele oferece uma série de recursos e benefícios que o tornam uma excelente escolha para projetos de todos os portes.

Principais recursos e benefícios do Spring Boot

Recursos que impulsionam a produtividade

Autoconfiguração

- Simplifica a configuração de componentes, detectando automaticamente as necessidades do projeto.
- Elimina a necessidade de arquivos XML complexos.
- Permite focar na lógica de negócio em vez de configurações.

Recursos que impulsionam a produtividade

Starters

- Dependências pré-configuradas que agrupam funcionalidades comuns.
- Agilizam a inclusão de recursos como Spring Web, Spring Data, Spring Security, etc.
- Reduzem a necessidade de gerenciar manualmente as dependências do projeto.

Recursos que impulsionam a produtividade

Servidor embutido

- Tomcat, Jetty ou Undertow já inclusos no projeto.
- Elimina a necessidade de configurar e implantar um servidor externo.
- Facilita o desenvolvimento e o teste local.

Recursos que impulsionam a produtividade

Actuator

- Fornece endpoints para monitorar a saúde, métricas e informações da aplicação em tempo real.
- Permite acompanhar o desempenho, identificar problemas e otimizar a aplicação.

Recursos que impulsionam a produtividade

Spring Boot CLI

- Interface de linha de comando para criar e executar aplicações Spring Boot rapidamente.
- Ideal para prototipação e desenvolvimento rápido.

Benefícios que aceleram o desenvolvimento

Maior produtividade

- Menos tempo gasto com configuração e mais tempo dedicado à lógica de negócio.
- Desenvolvimento mais rápido e eficiente.

Benefícios que aceleram o desenvolvimento

Menor curva de aprendizado

- Convenções e padrões estabelecidos facilitam o aprendizado e a adoção.
- Documentação abrangente e comunidade ativa.

Benefícios que aceleram o desenvolvimento

Fácil integração

- Amplo suporte a diversas tecnologias e bibliotecas.
- Integração simplificada com bancos de dados, sistemas de mensagens, etc.

Benefícios que aceleram o desenvolvimento

Escalabilidade e performance

- Recursos para construir aplicações robustas e de alta performance.
- Suporte a microsserviços e arquiteturas distribuídas.

Benefícios que aceleram o desenvolvimento

Qualidade e confiabilidade

- Baseado no sólido Spring Framework, com anos de experiência e maturidade.
- Testes automatizados e práticas recomendadas para garantir a qualidade do código.

Em resumo

- O Spring Boot oferece um conjunto poderoso de recursos que simplificam e aceleram o desenvolvimento de aplicações Java, tornando-o uma escolha ideal para projetos modernos e escaláveis.

Arquitetura básica de uma aplicação Spring Boot

Camada de apresentação (Controllers)

- Responsável por receber requisições HTTP, processar dados e retornar respostas.
- Utiliza anotações como @RestController, @GetMapping, @PostMapping, etc.
- Interage com a camada de serviço para obter os dados necessários.

Camada de serviço (Services)

- Contém a lógica de negócio da aplicação.
- Implementa regras de negócio, validações e orquestração de outros componentes.
- Interage com a camada de repositório para acessar e persistir dados.

Camada de repositório (Repositories)

- Responsável por acessar o banco de dados ou outras fontes de dados.
- Utiliza o Spring Data para simplificar o acesso a dados e abstrair a tecnologia de banco de dados.
- Fornece métodos para realizar operações CRUD (criar, ler, atualizar e deletar) em entidades.

Camada de domínio (Entities):

- Representa os objetos de negócio da aplicação.
- Define as classes e atributos que correspondem às tabelas do banco de dados.
- Pode conter lógica de validação e relacionamentos com outras entidades.

Principais componentes

- ***Spring Boot Application***: Classe principal que inicia a aplicação.
  application.properties (ou application.yml): Arquivo de configuração da aplicação.
- ***pom.xml (ou build.gradle)***: Arquivo de gerenciamento de dependências.

Principais componentes

- ***Controllers***: Classes que definem os endpoints da API.
- ***Services***: Classes que implementam a lógica de negócio.
- ***Repositories***: Interfaces que definem o acesso aos dados.
- ***Entities***: Classes que representam os objetos de negócio.

Benefícios da arquitetura em camadas

- ***Organização***: Separação clara de responsabilidades, facilitando a manutenção e evolução do código.
- ***Reusabilidade***: Componentes podem ser reutilizados em diferentes partes da aplicação.
- ***Testabilidade***: Facilita a criação de testes unitários e de integração para cada camada.
- ***Escalabilidade***: Permite escalar cada camada de forma independente, conforme a demanda.

Em resumo

- A arquitetura em camadas do Spring Boot oferece uma estrutura sólida e flexível para construir aplicações Java, promovendo a organização, reusabilidade, testabilidade e escalabilidade do código.

Configuração e inicialização

Anotação @SpringBootApplication

- ***Marcador principal***: Indica que a classe é a principal da aplicação Spring Boot.
- ***Combinação de anotações***: Equivale a usar @Configuration, @EnableAutoConfiguration e @ComponentScan juntas.

Anotação @SpringBootApplication

- @Configuration: Permite definir beans e configurações na própria classe.
- @EnableAutoConfiguration: Habilita a autoconfiguração do Spring Boot, que configura automaticamente os componentes necessários com base nas dependências do projeto.
- @ComponentScan: Habilita a varredura de componentes do Spring, que procura por classes anotadas com @Component, @Service, @Repository, etc., e as registra como beans no contexto da aplicação.

Ponto de entrada (método main)

```
@SpringBootApplication
public class MinhaAplicacao {

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

Ponto de entrada (método main)

- Inicia a aplicação: O método main é o ponto de entrada da aplicação Spring Boot.
- SpringApplication.run():
  - Cria um contexto de aplicação Spring.
  - Inicia o servidor web embutido (Tomcat, Jetty ou Undertow).
  - Realiza a varredura de componentes e a autoconfiguração.
  - Executa a aplicação, tornando-a disponível para receber requisições.

Exemplo

```java
package com.example.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication
@RestController
public class DemoApplication {

    @GetMapping("/")
    public String hello() {
        return "Olá, mundo!";
    }

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

Em resumo

- A anotação @SpringBootApplication marca a classe principal da aplicação Spring Boot e habilita a autoconfiguração.
- O método main é o ponto de entrada da aplicação e inicia o servidor web embutido.
- Juntos, eles permitem criar aplicações Spring Boot de forma rápida e fácil, sem a necessidade de configurações manuais complexas.

Spring Initializr e criação de projetos

Spring Initializr

- **Ferramenta online:** Disponível em https://start.spring.io/
- **Agiliza a criação de projetos Spring Boot:** Gera um projeto básico com a estrutura e as dependências necessárias.

Spring Initializr

- **Personalização:** Permite escolher:
  - Linguagem (Java, Kotlin, Groovy)
  - Build system (Maven ou Gradle)
  - Versão do Spring Boot
  - Dependências (Spring Web, Spring Data, Spring Security, etc.)
  - Metadados do projeto (nome, descrição, groupId, artifactId)

Spring Initializr

- **Download:** Gera um arquivo ZIP com o projeto pronto para ser importado no IDE.
- **Integração com IDEs:** Muitos IDEs (Eclipse, IntelliJ IDEA, VS Code) possuem integração com o Spring Initializr, facilitando ainda mais a criação de projetos.

Criação de projetos com Spring Initializr

1. **Acesse o site:** Abra o Spring Initializr em https://start.spring.io/.
2. **Selecione as opções:** Escolha a linguagem, build system, versão do Spring Boot, dependências e metadados do projeto.
3. **Gere o projeto:** Clique em "Generate" para baixar o arquivo ZIP com o projeto.
4. **Importe no IDE:** Descompacte o arquivo ZIP e importe o projeto no seu IDE favorito.

Vantagens

- **Rápido e fácil:** Cria projetos Spring Boot em poucos minutos.
- **Personalizável:** Permite escolher as tecnologias e dependências desejadas.
- **Estrutura consistente:** Gera projetos com uma estrutura padrão, facilitando a organização e manutenção.
- **Integração com IDEs:** Facilita a importação e o desenvolvimento do projeto.

Em resumo

- O Spring Initializr é uma ferramenta essencial para iniciar projetos Spring Boot de forma rápida e eficiente, permitindo que você se concentre na lógica de negócio em vez de na configuração inicial do projeto.

Arquivo application.properties (ou application.yml)

O que é o arquivo `application.properties` (ou `application.yml`)?

- **Arquivo de configuração central:** Armazena as configurações da aplicação Spring Boot.
- **Formatos:** Pode ser em `.properties` (chave=valor) ou `.yml` (estruturado em YAML).

O que é o arquivo `application.properties` (ou `application.yml`)?

- **Localização:** Fica na pasta `src/main/resources` do projeto.
- **Sobrescrita:** As configurações podem ser sobrescritas por variáveis de ambiente, argumentos de linha de comando ou outros arquivos de configuração.

Principais configurações

- **Servidor:**
  - `server.port` : Porta em que a aplicação vai rodar (padrão 8080).
  - `server.servlet.context-path` : Caminho base da aplicação (opcional).

Principais configurações

- **Banco de dados:**
  - `spring.datasource.url` : URL de conexão com o banco de dados.
  - `spring.datasource.username` : Usuário do banco de dados.
  - `spring.datasource.password` : Senha do banco de dados.

Principais configurações

- **Logging**:
  - `logging.level.*` : Nível de log para diferentes pacotes (ex: `logging.level.root=INFO`).
  - `logging.file.name` : Nome do arquivo de log.

Principais configurações

- **Outras:**
  - `spring.application.name` : Nome da aplicação.
  - `spring.profiles.active` : Perfil de configuração ativo.
  - `spring.jpa.hibernate.ddl-auto` : Estratégia de criação/atualização do esquema do banco de dados.

Exemplo em `application.properties`

```
server.port=8081
spring.datasource.url=jdbc:mysql://localhost:3306/mydb
spring.datasource.username=root
spring.datasource.password=password
logging.level.root=INFO
logging.file.name=logs/myapp.log
```

Exemplo em `application.yml`

```yaml
server:
  port: 8081
spring:
  datasource:
    url: jdbc:mysql://localhost:3306/mydb
    username: root
    password: password
logging:
  level:
    root: INFO
  file:
    name: logs/myapp.log
```

Boas práticas

- **Organização:** Agrupe as configurações por categoria (server, database, logging, etc.).
- **Comentários:** Use comentários para explicar as configurações.
- **Externalização:** Armazene configurações sensíveis (senhas, chaves de API) em variáveis de ambiente.
- **Profiles:** Use profiles para definir diferentes configurações para diferentes ambientes (desenvolvimento, teste, produção).

Em resumo

- O arquivo `application.properties` (ou `application.yml`) é fundamental para configurar uma aplicação Spring Boot. Ele permite personalizar o comportamento da aplicação de forma fácil e flexível.

Empacotamento e Deploy de Aplicações Spring Boot

Empacotamento

- **JAR executável:** Spring Boot empacota a aplicação e suas dependências em um único arquivo JAR executável.
- **Facilidade de deploy:** Basta ter um ambiente com Java instalado para executar o JAR.

Empacotamento

- **Comando:** `mvn package` (Maven) ou `gradle build` (Gradle)
- **Estrutura do JAR:**
  - Camadas da aplicação (classes, recursos)
  - Bibliotecas e dependências
  - Servidor web embutido (Tomcat, Jetty ou Undertow)
  - Script de inicialização (`start.sh` ou `start.bat`)

Opções de deploy

- **Execução direta:**
  - `java -jar nome-da-aplicacao.jar`
  - Simples e rápido, ideal para ambientes de desenvolvimento e teste.

Opções de deploy

- **Containers (Docker):**
  - Crie uma imagem Docker com a aplicação e suas dependências.
  - Permite fácil portabilidade e escalabilidade.
  - `docker build -t nome-da-imagem .`
  - `docker run -p 8080:8080 nome-da-imagem`

Opções de deploy

- **Cloud:**
  - Implante em plataformas como AWS, Azure, Google Cloud, Heroku, etc.
  - Oferece escalabilidade automática, balanceamento de carga e outras funcionalidades.
  - Utilize serviços como AWS Elastic Beanstalk, Azure App Service ou Google App Engine.

Opções de deploy

- **Servidores de aplicação tradicionais:**
  - Implante o JAR em um servidor Tomcat, Jetty ou WildFly externo.
  - Requer mais configuração, mas pode ser necessário em alguns casos.

Boas práticas

- **Profiles:** Utilize profiles para definir configurações específicas para cada ambiente (desenvolvimento, teste, produção).
- **Externalização de configurações:** Armazene configurações sensíveis (senhas, chaves de API) em variáveis de ambiente ou arquivos externos.
- **Monitoramento:** Utilize o Spring Boot Actuator para monitorar a saúde e métricas da aplicação em produção.
- **Logs:** Configure o logging para registrar informações importantes e facilitar a resolução de problemas.

Em resumo

- O Spring Boot simplifica o empacotamento e deploy de aplicações Java, oferecendo diversas opções para atender às necessidades de diferentes ambientes e requisitos.

Spring Web MVC

Controladores REST

- **Componentes chave:** Permitem criar APIs RESTful para expor recursos e funcionalidades da aplicação.
- **Recebem requisições HTTP:** Respondem com dados no formato JSON, XML ou outros.
- **Manipulam diferentes métodos HTTP:** GET, POST, PUT, DELETE, etc.
- **Baseados em anotações:** Facilitam a definição de endpoints e mapeamento de URLs.

Anotação `@RestController`:

- **Combinação de `@Controller` e `@ResponseBody`:** Simplifica a criação de controladores REST.
- **`@Controller`:** Marca a classe como um controlador Spring MVC.
- **`@ResponseBody`:** Indica que os métodos do controlador retornam diretamente o corpo da resposta HTTP (JSON, XML, etc.), sem precisar de uma View.

Exemplo

```java
@RestController
@RequestMapping("/api/produtos")
public class ProdutoController {

    @Autowired
    private ProdutoService produtoService;

    @GetMapping
    public List<Produto> listarProdutos() {
        return produtoService.listarTodos();
    }

    @GetMapping("/{id}")
    public Produto buscarProdutoPorId(@PathVariable Long id) {
        return produtoService.buscarPorId(id);
    }

    @PostMapping
    public Produto criarProduto(@RequestBody Produto produto) {
        return produtoService.salvar(produto);
    }

    // ... outros métodos para atualizar e deletar produtos
}
```

Funcionamento

1. **Requisição HTTP:** O cliente envia uma requisição para um endpoint da API (ex: `/api/produtos`).
2. **Mapeamento:** O Spring MVC mapeia a requisição para o método do controlador correspondente, com base no método HTTP e na URL.
3. **Processamento:** O método do controlador executa a lógica de negócio, acessando o serviço e/ou repositório.
4. **Resposta:** O método retorna um objeto que é automaticamente convertido para JSON (ou outro formato) e enviado como resposta ao cliente.

Vantagens do `@RestController`

- **Simplicidade:** Elimina a necessidade de configurar manualmente a serialização de objetos para JSON.
- **Concisão:** Reduz a quantidade de código boilerplate nos controladores.
- **Foco na lógica de negócio:** Permite que os desenvolvedores se concentrem na implementação da lógica de negócio, em vez de detalhes de configuração.

Em resumo

- A anotação `@RestController` é uma ferramenta poderosa para criar controladores RESTful de forma rápida e fácil no Spring Boot. Ela simplifica o desenvolvimento de APIs, permitindo que você se concentre na lógica de negócio e entregue resultados mais rapidamente.

Mapeamento de endpoints

- **Definir URLs:** Determina quais URLs da API acionam cada método do controlador.
- **Anotações específicas para cada método HTTP:**
  - `@GetMapping` : Requisições GET (obter dados).
  - `@PostMapping` : Requisições POST (criar dados).
  - `@PutMapping` : Requisições PUT (atualizar dados).
  - `@DeleteMapping` : Requisições DELETE (remover dados).
  - `@PatchMapping` : Requisições PATCH (atualização parcial de dados).

Sintaxe

```
@GetMapping("/recurso/{id}")
public Recurso obterRecurso(@PathVariable Long id) {
    // ... lógica para obter o recurso com o ID especificado
}

@PostMapping("/recurso")
public Recurso criarRecurso(@RequestBody Recurso recurso) {
    // ... lógica para criar um novo recurso
}

// ... outros métodos com @PutMapping, @DeleteMapping, @PatchMapping
```

Exemplos

- `@GetMapping("/usuarios")`: Retorna a lista de todos os usuários.
- `@GetMapping("/usuarios/{id}")`: Retorna o usuário com o ID especificado.
- `@PostMapping("/usuarios")`: Cria um novo usuário com os dados fornecidos no corpo da requisição.

Exemplos

- `@PutMapping("/usuarios/{id}")`: Atualiza o usuário com o ID especificado com os dados fornecidos no corpo da requisição.
- `@DeleteMapping("/usuarios/{id}")`: Remove o usuário com o ID especificado.
- `@PatchMapping("/usuarios/{id}")`: Atualiza parcialmente o usuário com o ID especificado com os dados fornecidos no corpo da requisição.

Nível de classe

- `@RequestMapping`: Define um prefixo para todos os endpoints do controlador.

```
@RestController
@RequestMapping("/api/usuarios")
public class UsuarioController {
    // ... métodos com @GetMapping, @PostMapping, etc.
}
```

Em resumo

- As anotações `@GetMapping`, `@PostMapping`, etc., permitem mapear de forma clara e concisa os endpoints da sua API RESTful, facilitando o desenvolvimento e a manutenção do código.

Parâmetros de requisição em controladores Spring Boot

- `@RequestParam`: Captura parâmetros da query string da URL.
- `@PathVariable`: Captura valores de variáveis presentes no caminho da URL.
- `@RequestBody`: Captura o corpo da requisição, geralmente em formato JSON.

@RequestParam

- **Uso:** Indicado para parâmetros opcionais ou filtros.
- **Exemplo:**

```
@GetMapping("/produtos")
public List<Produto> buscarProdutos(
    @RequestParam(name = "nome", required = false) String nome,
    @RequestParam(name = "categoria", required = false) Long categoriaId
) {
    // ... lógica para buscar produtos com base nos parâmetros
}
```

`@PathVariable`

- **Uso:** Indicado para identificar recursos específicos.
- **Exemplo:**

```
@GetMapping("/produtos/{id}")
public Produto buscarProdutoPorId(@PathVariable Long id) {
    // ... lógica para buscar o produto com o ID especificado
}
```

@RequestBody

- **Uso:** Indicado para enviar dados complexos no corpo da requisição (ex: criação ou atualização de recursos).
- **Exemplo:**

```
@PostMapping("/produtos")
public Produto criarProduto(@RequestBody Produto produto) {
    // ... lógica para criar um novo produto
}
```

Outros detalhes:

- **Tipos de dados:** Os parâmetros podem ser de tipos primitivos (int, long, etc.), Strings, objetos ou coleções.
- **Validação:** Utilize Bean Validation (`@NotNull` , `@NotBlank` , `@Size` , etc.) para validar os parâmetros.
- **Conversão:** O Spring Boot converte automaticamente os parâmetros para o tipo esperado.

Em resumo

- `@RequestParam` : Parâmetros da query string (ex: `/produtos?nome=teclado&categoria=1`).
- `@PathVariable` : Valores de variáveis no caminho da URL (ex: `/produtos/123`).
- `@RequestBody` : Corpo da requisição (ex: `{ "nome": "teclado", "preco": 150.0 }`).

Exemplo completo

```
@GetMapping("/produtos")
public ResponseEntity<List<Produto>> buscarProdutos(
        @RequestParam(name = "nome", required = false) String nome,
        @RequestParam(name = "categoria", required = false) Long categoriaId,
        @RequestParam(name = "pagina", defaultValue = "0") int pagina,
        @RequestParam(name = "tamanho", defaultValue = "10") int tamanho
) {
    Pageable paginacao = PageRequest.of(pagina, tamanho);
    Page<Produto> produtos = produtoService.buscarPorNomeECategoria(nome, categoriaId, paginacao);
    return ResponseEntity.ok(produtos.getContent());
}
```

- Com este exemplo, podemos buscar produtos por nome e categoria, com paginação.

Retorno de respostas em controladores Spring Boot

- `@ResponseBody` : Anotação que indica que o valor de retorno do método deve ser serializado (convertido) para o formato especificado (JSON por padrão) e enviado diretamente como corpo da resposta HTTP.

Retorno de respostas em controladores Spring Boot

- `ResponseEntity` : Classe que encapsula a resposta HTTP completa, incluindo:
  - **Corpo da resposta:** O objeto a ser serializado (opcional).
  - **Status HTTP:** Código de status da resposta (200 OK, 404 Not Found, etc.).
  - **Cabeçalhos:** Informações adicionais sobre a resposta (Content-Type, Location, etc.).

@ResponseBody

- **Uso:** Simplifica o retorno de dados em formatos como JSON ou XML.
- **Exemplo:**

```
@GetMapping("/produtos/{id}")
@ResponseBody
public Produto buscarProdutoPorId(@PathVariable Long id) {
    return produtoService.buscarPorId(id); // Retorna um objeto Produto serializado em JSON
}
```

ResponseEntity

- **Uso:** Permite controle total sobre a resposta HTTP, incluindo status e cabeçalhos.
- **Exemplo:**

```
@GetMapping("/produtos/{id}")
public ResponseEntity<Produto> buscarProdutoPorId(@PathVariable Long id) {
    Produto produto = produtoService.buscarPorId(id);
    if (produto != null) {
        return ResponseEntity.ok(produto); // Retorna 200 OK com o produto no corpo
    } else {
        return ResponseEntity.notFound().build(); // Retorna 404 Not Found
    }
}
```

Boas práticas

- **Use `ResponseEntity` para:**
  - Retornar códigos de status específicos (além de 200 OK).
  - Adicionar cabeçalhos personalizados à resposta.
  - Redirecionar o cliente para outra URL.
- **Use `@ResponseBody` para:**
  - Retornar dados simples em formato JSON ou XML.
  - Simplificar o código quando não precisa de controle total sobre a resposta.

Em resumo

- `@ResponseBody` : Simplifica o retorno de dados em formato JSON ou XML.
- `ResponseEntity` : Permite controle total sobre a resposta HTTP, incluindo status e cabeçalhos.
- Escolha o método mais adequado para cada situação, levando em consideração a necessidade de controle sobre a resposta e a complexidade do código.

Formatação de dados com Jackson

- **Biblioteca de serialização/desserialização JSON:** Converte objetos Java em JSON e vice-versa.
- **Integração com Spring Boot:** Já incluído por padrão, facilitando a formatação de dados em APIs REST.
- **Personalização:** Permite configurar a formatação do JSON de diversas formas.

Serialização (Java para JSON)

```java
@RestController
public class ProdutoController {

    @GetMapping("/produtos/{id}")
    public Produto buscarProdutoPorId(@PathVariable Long id) {
        Produto produto = produtoService.buscarPorId(id);
        // Jackson automaticamente serializa o objeto 'produto' para JSON
        return produto;
    }
}
```

Desserialização (JSON para Java)

```java
@PostMapping("/produtos")
public Produto criarProduto(@RequestBody Produto produto) {
    // Jackson automaticamente desserializa o JSON da requisição para o objeto 'produto'
    return produtoService.salvar(produto);
}
```

Personalização com anotações

- `@JsonProperty`: Renomeia campos no JSON.
- `@JsonIgnore`: Ignora campos na serialização.
- `@JsonFormat`: Formata datas, números, etc.
- `@JsonInclude`: Controla a inclusão de campos nulos.

```java
public class Produto {

    @JsonProperty("nome_produto")
    private String nome;

    @JsonIgnore
    private String codigoInterno;

    @JsonFormat(pattern = "dd/MM/yyyy")
    private LocalDate dataCadastro;

    // ... outros campos e métodos
}
```

Boas práticas

- **Use anotações para personalização:** Facilita a leitura e manutenção do código.
- **Defina um `ObjectMapper` global:** Centraliza as configurações de serialização/desserialização.
- **Utilize módulos:** Para lidar com tipos de dados específicos.
- **Teste a serialização/desserialização:** Garante que os dados estão sendo formatados corretamente.

Em resumo

- Jackson é uma ferramenta poderosa para formatar dados em JSON em aplicações Spring Boot. Com suas opções de personalização, você pode controlar a forma como seus objetos são serializados e desserializados, garantindo a interoperabilidade com outras aplicações e sistemas.

Templates com Thymeleaf

Thymeleaf

- **Engine de templates Java:** Permite criar páginas HTML dinâmicas, integrando-se facilmente com o Spring Boot.
- **Sintaxe:** Utiliza atributos especiais para inserir dados do modelo, iterar sobre listas, criar condicionais, etc.
- **Segurança:** Previne ataques XSS (Cross-Site Scripting) por padrão.
- **Integração com Spring:** Facilita o uso de objetos e expressões Spring no template.

Exemplo

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Lista de Produtos</title>
</head>
<body>
    <h1>Produtos</h1>
    <ul>
        <li th:each="produto : ${produtos}">
            <span th:text="${produto.nome}"></span> -
            <span th:text="${#numbers.formatDecimal(produto.preco, 1, 2)}"></span>
        </li>
    </ul>
</body>
</html>
```

Principais atributos:

- `th:text`: Insere o valor de uma variável no template.
- `th:each`: Itera sobre uma lista de objetos.
- `th:if`, `th:unless`, `th:switch`, `th:case`: Criam estruturas condicionais.
- `th:href`, `th:src`: Inserem URLs dinâmicas.
- `th:fragment`: Define fragmentos de template reutilizáveis.

Integração com Spring

- **`@Controller`**: Retorna o nome da view e o modelo de dados.
- **`Model`**: Adiciona atributos ao modelo (ex: `model.addAttribute("produtos", produtos);`).
- **`ModelAndView`**: Combina a view e o modelo em um único objeto.

Configuração

- **Adicionar a dependência:** `spring-boot-starter-thymeleaf` no `pom.xml` (ou `build.gradle`).
- **Configurar o prefixo e sufixo das views:**

```
spring.thymeleaf.prefix=classpath:/templates/
spring.thymeleaf.suffix=.html
```

Vantagens

- **Natural Template:** Permite que o HTML seja visualizado corretamente no navegador, mesmo sem o servidor rodando.
- **Flexível:** Ampla variedade de atributos para manipular o conteúdo dinamicamente.
- **Seguro:** Previne ataques XSS por padrão.
- **Integração com Spring:** Facilita o uso de recursos do Spring nos templates.

Em resumo

- Thymeleaf é uma ótima opção para criar templates em aplicações Spring Boot, oferecendo uma sintaxe simples, segurança e integração com o framework.

Spring Data

Spring Data JPA

- **Módulo do Spring Data:** Simplifica o acesso a dados em bancos de dados relacionais usando JPA (Java Persistence API).
- **Reduz o código boilerplate:** Elimina a necessidade de escrever código repetitivo para operações CRUD (Create, Read, Update, Delete) e consultas simples.
- **Facilita a manutenção:** Abstrai a tecnologia de banco de dados, permitindo trocar de banco com facilidade.

Repositórios

- **Interfaces que estendem `JpaRepository` (ou `CrudRepository`):**
  - Fornecem métodos prontos para operações CRUD básicas.
  - Permitem criar consultas personalizadas usando métodos derivados ou `@Query`.

Repositórios

- **Métodos derivados:**
  - Nomes de métodos seguem uma convenção para gerar consultas automaticamente (ex: `findByNome`, `findByNomeAndIdade`).
- **`@Query`:**
  - Permite escrever consultas JPQL (Java Persistence Query Language) personalizadas.

Exemplo

```
@Repository
public interface ProdutoRepository extends JpaRepository<Produto, Long> {

    List<Produto> findByNome(String nome);

    List<Produto> findByNomeAndPrecoLessThan(String nome, BigDecimal preco);

    @Query("SELECT p FROM Produto p WHERE p.categoria.nome = :categoria")
    List<Produto> findByCategoria(@Param("categoria") String categoria);
}
```

Funcionamento

1. **Crie uma interface que estenda `JpaRepository`:** Especifique a entidade e o tipo de chave primária.
2. **Defina métodos derivados ou `@Query`:** Para criar consultas personalizadas.
3. **Injete o repositório no seu serviço:** Use `@Autowired` para injetar o repositório no seu serviço.
4. **Utilize os métodos do repositório:** Para realizar operações CRUD e consultas no banco de dados.

Vantagens

- **Produtividade:** Reduz significativamente a quantidade de código necessário para acessar o banco de dados.
- **Manutenção:** Facilita a manutenção e evolução do código, pois as consultas são definidas em um único lugar.
- **Flexibilidade:** Permite usar diferentes tecnologias de banco de dados com poucas alterações no código.
- **Testabilidade:** Facilita a criação de testes unitários, pois os repositórios podem ser facilmente mockados.

Em resumo

- Spring Data JPA e repositórios são ferramentas poderosas para simplificar o acesso a dados em aplicações Spring Boot, permitindo que você se concentre na lógica de negócio em vez de na implementação de código repetitivo.

Consultas personalizadas em Spring Data JPA

Métodos Derivados (Query Methods)

- **Convenção sobre configuração:** Spring Data JPA interpreta o nome do método no repositório para criar automaticamente consultas JPQL.

Métodos Derivados (Query Methods)

Padrões de nome

- `findBy<Atributo>` : Busca por um único resultado com base no atributo.
- `findAllBy<Atributo>` : Busca por todos os resultados com base no atributo.

Métodos Derivados (Query Methods)

Padrões de nome

- `existsBy<Atributo>` : Verifica se existe algum registro com o atributo.
- `countBy<Atributo>` : Conta o número de registros com o atributo.
- Combinações: `findBy<Atributo>And<Atributo>` , `findBy<Atributo>OrderBy<Atributo>Desc` , etc.

Exemplo

```
public interface ProdutoRepository extends JpaRepository<Produto, Long> {
    List<Produto> findByNome(String nome);
    List<Produto> findByNomeContainingIgnoreCase(String nome);
    List<Produto> findByPrecoLessThan(BigDecimal preco);
    Optional<Produto> findTopByOrderByPrecoDesc();
}
```

Anotação `@Query`

- **Flexibilidade:** Permite escrever consultas JPQL (Java Persistence Query Language) personalizadas.
- **Uso:** Quando os métodos derivados não são suficientes ou quando se deseja maior controle sobre a consulta.

Exemplo

```
@Query("SELECT p FROM Produto p WHERE p.nome LIKE %:nome%")
List<Produto> buscarPorNome(@Param("nome") String nome);
```

Injeção de Dependências com `@Autowired`

Injeção de Dependências (DI)

- **Princípio SOLID:** "Inversão de Controle" (Dependency Inversion Principle).
- **Objetivo:** Reduzir o acoplamento entre classes, tornando o código mais flexível e fácil de testar.
- **Funcionamento:**
  - As dependências de uma classe são fornecidas externamente, em vez de serem criadas internamente.
  - Um contêiner (framework) é responsável por criar os objetos e injetá-los nas classes que precisam deles.

`@Autowired`

- **Anotação do Spring:** Indica que um campo, construtor ou método setter deve receber uma instância de um bean gerenciado pelo Spring.
- **Injeção por tipo:** O Spring procura um bean compatível com o tipo do campo/parâmetro e o injeta automaticamente.

Exemplo

```
@Service
public class ProdutoService {

    @Autowired
    private ProdutoRepository produtoRepository;

    // ...
}
```

Boas Práticas e Dicas para Desenvolvimento com Spring Boot

Estrutura do projeto

- **Organização:** Siga uma estrutura de pacotes clara e consistente (controllers, services, repositories, models, etc.).
- **Modularização:** Divida a aplicação em módulos menores e coesos para facilitar a manutenção e reutilização de código.

Configuração

- **Externalize as configurações:** Utilize arquivos externos (`application.properties` , `application.yml`) ou variáveis de ambiente para armazenar configurações, especialmente as sensíveis (senhas, chaves de API).
- **Utilize profiles:** Crie profiles para diferentes ambientes (desenvolvimento, teste, produção) e ative-os conforme necessário.
- **Evite configurações mágicas:** Documente as configurações e explique suas finalidades para facilitar o entendimento e a manutenção do código.

Desenvolvimento

- **Siga os princípios SOLID:** Aplique os princípios de design SOLID para criar um código mais flexível, reutilizável e fácil de manter.
- **Utilize injeção de dependências:** Utilize `@Autowired` ou `@Inject` para injetar dependências em vez de criá-las manualmente, promovendo o baixo acoplamento e facilitando os testes.
- **Escreva testes unitários e de integração:** Garanta a qualidade do código e evite regressões com testes automatizados.

Desenvolvimento

- **Utilize o Spring Boot Actuator:** Monitore a saúde e as métricas da aplicação em tempo real para identificar e resolver problemas rapidamente.
- **Mantenha as dependências atualizadas:** Utilize a versão mais recente do Spring Boot e suas dependências para garantir a segurança e o acesso aos recursos mais recentes.

Dicas:

- **Leia a documentação:** A documentação do Spring Boot é abrangente e oferece informações detalhadas sobre todos os recursos e funcionalidades.
- **Explore a comunidade:** A comunidade Spring Boot é grande e ativa, com fóruns, blogs e eventos onde você pode encontrar ajuda e trocar experiências.
- **Utilize ferramentas:** IDEs como IntelliJ IDEA e Eclipse oferecem suporte e plugins para facilitar o desenvolvimento com Spring Boot.
- **Experimente:** Crie projetos simples para praticar e explorar os diferentes recursos do Spring Boot.

Em resumo

- Seguir boas práticas e dicas no desenvolvimento com Spring Boot garante um código mais organizado, manutenível, escalável e de alta qualidade, além de facilitar a colaboração em equipe e a resolução de problemas.