

BACK-END

Prof. Bruno Kurzawe



Construção e implementação de APIs

Uma API (Interface de Programação de Aplicativos) é um conjunto de regras e protocolos que permitem a comunicação entre diferentes softwares ou sistemas. Ela define os métodos e formatos de dados que os desenvolvedores podem utilizar para interagir com um serviço ou aplicativo específico.

Através de uma API, um sistema pode solicitar informações ou funcionalidades de outro sistema de forma padronizada, sem a necessidade de entender a complexidade interna do software subjacente. Isso promove a interoperabilidade entre sistemas e facilita a integração de diferentes componentes de software.

As APIs são amplamente utilizadas em desenvolvimento de software e são fundamentais para a criação de aplicações modernas. Elas podem ser encontradas em diversos contextos, como em sistemas operacionais, plataformas de nuvem, redes sociais, serviços web, bancos de dados, entre outros.

REST

REST (Representational State Transfer) é um estilo arquitetural para a construção de sistemas distribuídos na World Wide Web. Ele foi proposto por Roy Fielding em sua tese de doutorado em 2000 e se tornou uma das abordagens mais populares para o design de APIs.

Principais características do REST:

Estado Representacional: REST se baseia na ideia de que as informações são representadas como recursos identificados por URLs (Uniform Resource Locators). Cada recurso pode ter diferentes estados (representações), que podem ser acessados e manipulados através dos métodos HTTP.

Principais características do REST:

Comunicação sem Estado: Cada requisição do cliente para o servidor deve conter todas as informações necessárias para entender e atender a requisição. O servidor não mantém estado da comunicação entre requisições. Isso significa que cada requisição do cliente deve conter todas as informações necessárias para entendê-la, sem depender de requisições anteriores.

Principais características do REST:

Métodos HTTP: REST utiliza os métodos HTTP padrão (GET, POST, PUT, DELETE, etc.) para definir as operações que podem ser realizadas nos recursos. Cada método tem um significado semântico específico.

Principais características do REST:

Representações: Os recursos podem ter várias representações (como JSON, XML, HTML, etc.), e o cliente pode negociar a representação que deseja ao fazer uma requisição, utilizando os cabeçalhos HTTP como "Accept" e "Content-Type".

Uniformidade de Interface: REST segue um conjunto de regras e convenções bem definidas, o que promove a simplicidade, a consistência e a reutilização. Entre essas regras estão a identificação de recursos através de URLs, o uso de métodos HTTP e a manipulação de recursos através das representações.

Principais características do REST:

Sistema Cliente-Servidor: REST separa as preocupações entre o cliente e servidor. O servidor é responsável pelo armazenamento e gerenciamento dos recursos, enquanto o cliente é responsável por interagir com esses recursos através das requisições HTTP.

Principais características do REST:

Desempenho: REST é projetado para ser eficiente em termos de desempenho e escalabilidade, o que o torna uma escolha popular para sistemas distribuídos e APIs web.

Em resumo, REST é um estilo arquitetural que define como os sistemas distribuídos devem se comunicar na web. Ele oferece uma abordagem simples, uniforme e eficiente para a construção de APIs, promovendo a interoperabilidade entre sistemas e facilitando a integração de aplicações distribuídas.

Vamos criar nossas apis...

Antes vamos alterar nosso **ProdutoService**

```
@Service
public class ProdutoService {

    @Autowired
    private ProdutoRepository repository;

    public Produto save(Produto entity) {
        return repository.save(entity);
    }

    public List<Produto> buscaTodos() {
        return repository.findAll();
    }

    public Produto buscaPorId(Long id) {
        return repository.findById(id).orElse(null);
    }

    public Produto alterar(Produto entity) {
        return repository.save(entity);
    }

    public void remover(Long id) {
        repository.deleteById(id);
    }
}
```

Vamos criar um pacote, **resource**

Vamos criar um controller, **ProdutoController**

O uso do sufixo "Controller"

Clareza na Identificação: Ao nomear uma classe com o sufixo "Controller", imediatamente fica claro para os desenvolvedores que essa classe é responsável por lidar com as requisições e controlar o fluxo de uma API. Isso facilita a leitura e compreensão do código.

O uso do sufixo "Controller"

Organização do Código: O sufixo "Controller" ajuda na organização do código-fonte, especialmente em projetos grandes ou complexos. Ele indica explicitamente qual é o papel da classe dentro da aplicação.

O uso do sufixo "Controller"

Padrão de Projeto MVC: O sufixo "Controller" está intimamente relacionado com o padrão de projeto Model-View-Controller (MVC), que é amplamente utilizado no desenvolvimento de software. Nesse padrão, o controlador é responsável por gerenciar a interação entre o modelo (dados) e a visão (interface do usuário).

O uso do sufixo "Controller"

Facilita a Manutenção: Ao seguir convenções de nomenclatura amplamente aceitas, facilitamos a vida dos desenvolvedores que trabalham no projeto. Eles saberão exatamente onde procurar por componentes específicos.

O uso do sufixo "Controller"

Compatibilidade com Frameworks: Muitos frameworks e bibliotecas adotam essa convenção. Ao seguir essas convenções, você torna seu código mais compatível com ferramentas e frameworks populares.

O uso do sufixo "Controller"

Boa Prática Compartilhada: É uma prática bem estabelecida na comunidade de desenvolvimento de software. Ao usar essa convenção, você torna seu código mais compreensível para outros desenvolvedores, facilitando a colaboração.

O padrão REST utiliza os seguintes verbos HTTP para definir as ações que podem ser realizadas em um recurso:

GET	Usado para recuperar dados de um recurso. É uma operação segura, o que significa que não deve alterar o estado do servidor. Exemplo: obter informações de um usuário.
POST	Usado para criar um novo recurso. Pode ser usado para enviar dados que serão processados e armazenados pelo servidor. Exemplo: criar um novo usuário.
PUT	Usado para atualizar um recurso existente. Deve ser usado quando o cliente tem todos os detalhes do recurso a ser atualizado. Exemplo: atualizar informações de um usuário.
PATCH	Usado para atualizar parcialmente um recurso. Diferentemente do PUT, o PATCH é usado quando o cliente possui apenas alguns dos detalhes do recurso a ser atualizado.
DELETE	Usado para excluir um recurso. Exemplo: excluir um usuário.
OPTIONS	Pode ser usado para obter informações sobre os métodos permitidos em um recurso. É útil para a implementação do CORS (Cross-Origin Resource Sharing).
HEAD	Semelhante ao GET, mas usado para obter apenas os cabeçalhos da resposta, sem o corpo. Geralmente, é usado para verificar a existência de um recurso.
TRACE	Usado para testar a conectividade da rede. Ele faz com que o servidor retorne os cabeçalhos de requisição de volta ao cliente, o que pode ser útil para diagnósticos.

Vamos criar CRUD utilizando o **ProdutoController**

Nossa primeira ação será a de Criar alguma coisa, então dentro de **ProdutoController** escrevam o seguinte código

Exemplo POST

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

    @Autowired
    private ProdutoService service;

    @PostMapping
    public ResponseEntity create(@RequestBody Produto entity) {
        Produto save = service.salvar(entity);
        return ResponseEntity.created(URI.create("/api/produtos/" + entity.getId())).body(save);
    }
}
```

A anotação **@RestController** no Spring é uma especialização da anotação **@Controller** e é utilizada para indicar que a classe é um controlador que lida com requisições HTTP e retorna os resultados diretamente no corpo da resposta, sem a necessidade de uma visualização (view) adicional.

A anotação **@RequestMapping** é uma das anotações mais fundamentais no Spring e é usada para mapear solicitações HTTP para métodos de controlador específicos ou para um controlador em geral. Ela pode ser aplicada em nível de classe ou em nível de método.

A anotação **@PostMapping** é uma das anotações de mapeamento no Spring que é usada para mapear solicitações HTTP do tipo POST para métodos de manipulação de recursos em um controlador. Ela é frequentemente usada em conjunto com a anotação **@Controller** para definir os pontos de extremidade (endpoints) da API.

Agora podemos iniciar nossa aplicação

E vamos testar nossa aplicação, usando Postman, Talend ou Insomnia...

METHOD

POST

SCHEME :// HOST [":" PORT] [PATH ["?" QUERY]]

http://localhost:8080/api/produtos

Send

length: 34 byte(s)

QUERY PARAMETERS

+ Add query parameter

HEADERS ? 1/2

Form



Content-Type : application/json



+ Add header

Add authorization



BODY ?

Text

```
1 {  
2   "nome": "Produto A",  
3   "precoCompra": 50.0,  
4   "dataValidade": "2023-12-31",  
5   "dataPrazo": "2023-11-30",  
6   "status": "DISPONIVEL",  
7   "descricao": "Item X",  
8   "valorUnitario": 10.0,  
9   "estocavel": true  
10 }
```



▶ BODY ?

```
1 {  
2   "nome": "Produto A",  
3   "precoCompra": 50.0,  
4   "dataValidade": "2023-12-31",  
5   "dataPrazo": "2023-11-30",  
6   "status": "DISPONIVEL",  
7   "descricao": "Item X",  
8   "valorUnitario": 10.0,  
9   "estocavel": true  
10 }
```

JSON (JavaScript Object Notation) é um formato de dados leve e independente de linguagem de marcação utilizado para troca de informações entre sistemas. Ele é amplamente utilizado em aplicações web como um meio de transmitir dados entre um servidor e um cliente, e também é comumente usado em configurações de armazenamento de dados.

Ao chamarmos nossa api passando nosso JSON pelo método POST teremos o seguinte retorno.

Response

201

HEADERS ②

Location: </api/produtos/7>
Content-Type: application/json
Transfer-Encoding: chunked
Date: Sun, 08 Oct 2023 15:15:44 GMT
Keep-Alive: timeout=60
Connection: keep-alive

▶ COMPLETE REQUEST HEADERS

pretty ▼

BODY ②

```
{  
  id: 7,  
  descricao: "Item X",  
  valorUnitario: 10.0,  
  estocavel: true,  
  nome: "Produto A",  
  precoCompra: 50.0,  
  dataValidade: "2023-12-31",  
  dataPrazo: "2023-11-30",  
  status: "DISPONIVEL"  
}
```

lines nums  copy

HTTP Status Codes



1xx Informational

- 100 Continue
- 101 Switching Protocols
- 102 Processing

2xx Success

- 200 OK
- 201 Created
- 202 Accepted
- 203 Non-authoritative Information
- 204 No Content
- 205 Reset Content
- 206 Partial Content
- 207 Multi-Status
- 208 Already Reported
- 226 IM Used

3xx Redirection

- 300 Multiple Choices
- 301 Moved Permanently
- 302 Found
- 303 See Other
- 304 Not Modified
- 305 Use Proxy
- 307 Temporary Redirect
- 308 Permanent Redirect

4xx Client Error

- 400 Bad Request
- 401 Unauthorized
- 402 Payment Required
- 403 Forbidden
- 404 Not Found
- 405 Method Not Allowed
- 406 Not Acceptable
- 407 Proxy Authentication Required
- 408 Request Timeout
- 409 Conflict
- 410 Gone
- 411 Length Required
- 412 Precondition Failed
- 413 Payload Too Large
- 414 Request-URI Too Long
- 415 Unsupported Media Type
- 416 Requested Range Not Satisfiable
- 417 Expectation Failed
- 418 I'm a teapot
- 421 Misdirected Request
- 422 Unprocessable Entity
- 423 Locked
- 424 Failed Dependency
- 426 Upgrade Required
- 428 Precondition Required
- 429 Too Many Requests
- 431 Request Header Fields Too Large
- 444 Connection Closed Without Response
- 451 Unavailable For Legal Reasons
- 499 Client Closed Request

5xx Server Error

- 500 Internal Server Error
- 501 Not Implemented
- 502 Bad Gateway
- 503 Service Unavailable
- 504 Gateway Timeout
- 505 HTTP Version Not Supported
- 506 Variant Also Negotiates
- 507 Insufficient Storage
- 508 Loop Detected
- 510 Not Extended
- 511 Network Authentication Required
- 599 Network Connect Timeout Error

Vamos implementar agora uma api para **buscar todos** os nossos produtos.

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

    @Autowired
    private ProdutoService service;

    @PostMapping
    public ResponseEntity create(@RequestBody Produto entity) {
        Produto save = service.salvar(entity);
        return ResponseEntity.created(URI.create("/api/produtos/" + entity.getId())).body(save);
    }

    @GetMapping
    public ResponseEntity findAll() {
        List<Produto> produtos = service.buscaTodos();
        return ResponseEntity.ok(produtos);
    }
}
```



E vamos testar nossa aplicação, usando Postman, Talend ou Insomnia...

DRAFT

METHOD

GET

SCHEME :// HOST [":" PORT] [PATH ["?" QUERY]]

http://localhost:8080/api/produtos

▼ QUERY PARAMETERS

+ Add query parameter

HEADERS ⓘ

Form ▼



BODY ⓘ

+ Add header

🔑 Add authorization

XHR does not allow payloads for GET request.

Response

200

HEADERS ⓘ

Content-Type: application/json
Transfer-Encoding: chunked
Date: Sun, 08 Oct 2023 15:25:46 GMT
Keep-Alive: timeout=60
Connection: keep-alive

▶ COMPLETE REQUEST HEADERS

pretty ▼

BODY ⓘ

```
[
  {
    id: 4,
    descricao: "Item X",
    valorUnitario: 10.0,
    estocavel: true,
    nome: "Produto A",
    precoCompra: 50.0,
    dataValidade: "2023-12-31",
    dataPrazo: "2023-11-30",
    status: "DISPONIVEL"
  },
  {
    id: 5.
```

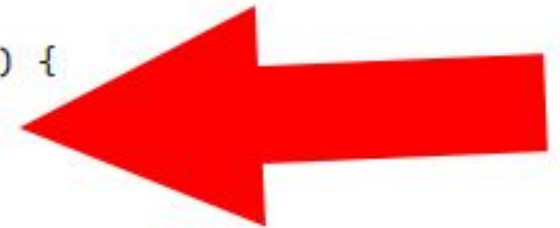
Vamos implementar agora uma api para **buscar por id** os nossos produtos.


```
@GetMapping
```

```
public ResponseEntity findAll() {  
    List<Produto> produtos = service.buscaTodos();  
    return ResponseEntity.ok(produtos);  
}
```

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

```
public ResponseEntity findById(@PathVariable("id") Long id) {  
    Produto produto = service.buscaPorId(id);  
    return ResponseEntity.ok(produto);  
}
```



DRAFT

METHOD

GET

SCHEME :// HOST [":" PORT] [PATH ["?" QUERY]]

http://localhost:8080/api/produtos/4

▼ QUERY PARAMETERS

+ Add query parameter


HEADERS ⓘ

+ Add header ⓘ Add authorization ⓘ

Form ▼ ◀ ▶

BODY ⓘ

XHR does not allow payloads for GET request.



Response

200

HEADERS ?

Content-Type: application/json
Transfer-Encoding: chunked
Date: Sun, 08 Oct 2023 15:54:43 GMT
Keep-Alive: timeout=60
Connection: keep-alive

▶ COMPLETE REQUEST HEADERS

pretty ▼

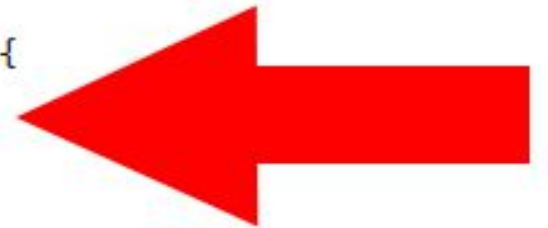
BODY ?

```
{  
  id: 4,  
  descricao: "Item X",  
  valorUnitario: 10.0,  
  estocavel: true,  
  nome: "Produto A",  
  precoCompra: 50.0,  
  dataValidade: "2023-12-31",  
  dataPrazo: "2023-11-30",  
  status: "DISPONIVEL"  
}
```

Vamos implementar agora uma api para **remover** os nossos produtos.

```
@GetMapping("/{id}")
public ResponseEntity findById(@PathVariable("id") Long id) {
    Produto produto = service.buscaPorId(id);
    return ResponseEntity.ok(produto);
}
```

```
@DeleteMapping("/{id}")
public ResponseEntity remove(@PathVariable("id") Long id) {
    service.remover(id);
    return ResponseEntity.noContent().build();
}
```



DRAFT

METHOD

DELETE

SCHEME :// HOST [":" PORT] [PATH ["?" QUERY]]

http://localhost:8080/api/produtos/4

QUERY PARAMETERS

+ Add query parameter


HEADERS ⓘ

+ Add header ⓘ Add authorization

Form

BODY ⓘ

XHR does not allow payloads for DELETE request.



Response

204

HEADERS [?]

pretty ▾



BODY [?]

Date: Sun, 08 Oct 2023 18:35:14 GMT
Keep-Alive: timeout=60
Connection: keep-alive

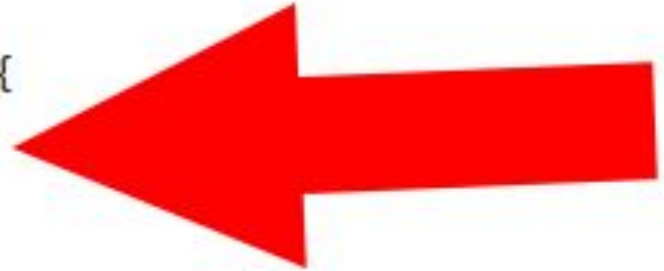
▶ COMPLETE REQUEST HEADERS

 copy

Vamos implementar agora uma api para **alterar** os nossos produtos.

Antes de criarmos a api, vamos alterar nosso service de alteração.

```
public Produto alterar(Produto entity) {  
    return repository.save(entity);  
}
```



```
public class NotFoundException extends RuntimeException {  
  
    public NotFoundException(String message) {  
        super(message);  
    }  
  
    public NotFoundException(String message, Throwable cause) {  
        super(message, cause);  
    }  
}
```

```
public Produto alterar(Long id, Produto entity) {  
    Optional<Produto> existingProdutoOptional = repository.findById(id);  
    if (existingProdutoOptional.isEmpty()) {  
        throw new NotFoundException("Produto não encontrado");  
    }  
  
    Produto existingProduto = existingProdutoOptional.get();  
    existingProduto.setNome(entity.getNome());  
    existingProduto.setEstocavel(entity.getEstocavel());  
    existingProduto.setStatus(entity.getStatus());  
    existingProduto.setPrecoCompra(entity.getPrecoCompra());  
    existingProduto.setDataValidade(entity.getDataValidade());  
    existingProduto.setDescricao(entity.getDescricao());  
    existingProduto.setValorUnitario(entity.getValorUnitario());  
  
    return repository.save(existingProduto);  
}
```

```
@PutMapping("{id}")
public ResponseEntity update(@PathVariable("id") Long id, @RequestBody Produto entity) {
    try {
        Produto alterado = service.alterar(id, entity);
        return ResponseEntity.ok().body(alterado);
    } catch (NotFoundException nfe) {
        return ResponseEntity.noContent().build();
    }
}
```



DRAFT

METHOD

PUT

SCHEME :// HOST [":" PORT] [PATH ["?" QUERY]]

http://localhost:8080/api/produtos/5

▼ QUERY PARAMETERS

+ Add query parameter

HEADERS ⓘ ⚙



Content-Type

: application/json



+ Add header

🔑 Add authorization

Form ▼

BODY ⓘ

```
1 {  
2   "id": 5,  
3   "descricao": "Item XYZW",  
4   "valorUnitario": 10.0,  
5   "estocavel": true,  
6   "nome": "Produto A",  
7   "precoCompra": 50.0,  
8   "dataValidade": "2023-12-31",  
9   "dataPrazo": "2023-11-30",  
10  "status": "DISPONIVEL"  
11 }
```

Text JSON XML HTML | Format body | ☒ Enable body evaluation

Response

200

HEADERS [?]

Content-Type: application/json
Transfer-Encoding: chunked
Date: Sun, 08 Oct 2023 19:05:01 GMT
Keep-Alive: timeout=60
Connection: keep-alive

► COMPLETE REQUEST HEADERS

pretty ▼

BODY [?]

```
{  
  id: 5,  
  descricao: "Item XYZW",  
  valorUnitario: 10.0,  
  estocavel: true,  
  nome: "Produto A",  
  precoCompra: 50.0,  
  dataValidade: "2023-12-31",  
  dataPrazo: "2023-11-30",  
  status: "DISPONIVEL"  
}
```

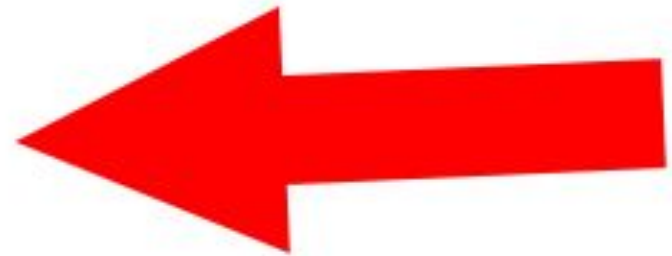
lines nums  copy

Funcionou? Sim, mas esse código do update é trabalhoso! Imagine se tivéssemos 70 campos para atualizar;

```
existingProduto.setNome(entity.g  
existingProduto.setEstocavel(ent  
existingProduto.setStatus(entity  
existingProduto.setPrecoCompra(e  
existingProduto.setDataValidade(  
existingProduto.setDescricao(ent  
existingProduto.setValorUnitario
```


Vamos modificar nossa classe de serviço para usar uma LIB para facilitar esse processo.

```
<dependency>  
  <groupId>org.modelmapper</groupId>  
  <artifactId>modelmapper</artifactId>  
  <version>2.4.4</version>  
</dependency>
```



clean + install // update e generate

Na raiz do projeto, mesmo nível Application

```
@Configuration
public class AppConfig {

    @Bean
    public ModelMapper modelMapper() {
        return new ModelMapper();
    }
}
```

Vamos injetar o ModelMapper no service

```
@Service
public class ProdutoService {

    @Autowired
    private ModelMapper modelMapper;


    @Autowired
    private ProdutoRepository repository;

    public Produto salvar(Produto entity) {
        return repository.save(entity);
    }
}
```



Vamos usar o modelMapper no lugar dos getter e setter

```
public Produto alterar(Long id, Produto entity) {  
    Optional<Produto> existingProdutoOptional = repository.findById(id);  
    if (existingProdutoOptional.isEmpty()) {  
        throw new NotFoundException("Produto não encontrado");  
    }  
  
    Produto existingProduto = existingProdutoOptional.get();  
  
    modelMapper.map(entity, existingProduto);  
  
    return repository.save(existingProduto);  
}
```



Continua funcionando...

Response

200

HEADERS ⓘ

Content-Type: application/json
Transfer-Encoding: chunked
Date: Sun, 08 Oct 2023 19:15:19 GMT
Keep-Alive: timeout=60
Connection: keep-alive

▶ COMPLETE REQUEST HEADERS

pretty ▼

BODY ⓘ

```
{  
  id: 5,  
  descricao: "Item XYZW",  
  valorUnitario: 10.0,  
  estocavel: true,  
  nome: "Produto AZZZZ",  
  precoCompra: 50.0,  
  dataValidade: "2023-02-28",  
  dataPrazo: "2023-08-15",  
  status: "ALUGADO"  
}
```

Beleza, agora conseguimos **criar, alterar, excluir** e **listar**.

Vamos criar uma **ValidationException**

```
public class ValidationException extends RuntimeException {  
  
    public ValidationException(String message) {  
        super(message);  
    }  
  
    public ValidationException(String message, Throwable cause) {  
        super(message, cause);  
    }  
}
```

Usaremos ela como base para nossas regras de negócio

Vamos fazer uns ajustes no nosso controller, vamos criar um **AbstractController**.

```
public abstract class AbstractController {

    @ResponseStatus(HttpStatus.BAD_REQUEST)
    @ExceptionHandler(MethodArgumentNotValidException.class)
    public Map<String, String> handleValidationExceptions(
        MethodArgumentNotValidException ex) {
        Map<String, String> errors = new HashMap<>();
        List<String> collect = ex.getBindingResult()
            .getAllErrors().stream()
            .map(p -> ((FieldError) p).getField() + " "
                + p.getDefaultMessage())
            .collect(Collectors.toList());
        errors.put("erro", collect.toString());
        return errors;
    }

    @ResponseStatus(HttpStatus.UNPROCESSABLE_ENTITY)
    @ExceptionHandler(ValidationException.class)
    public Map<String, String> handleValidationExceptions422(
        ValidationException ex) {
        Map<String, String> errors = new HashMap<>();
        errors.put("erro", ex.getMessage());
        return errors;
    }
}
```

Agora vamos criar uma regra de negócio qualquer

```
public Produto salvar(Produto entity) {  
  
    if (entity.getValorUnitario() < entity.getPrecoCompra()) {  
        throw new ValidationException("O valor unitário não pode ser menor que o preço de compra do produto!");  
    }  
  
    return repository.save(entity);  
}
```

DRAFT

METHOD SCHEME // HOST [":" PORT] [PATH ["?" QUERY]]

POST

▼ QUERY PARAMETERS

+ Add query parameter

HEADERS ? 1/2

☒ Content-Type : application/json

+ Add header

Form ▼ ◀ ▶ BODY ?

```
1 {
2   "nome": "Produto A",
3   "precoCompra": 50.0,
4   "dataValidade": "2023-12-31",
5   "dataPrazo": "2023-11-30",
6   "status": "DISPONIVEL",
7   "descricao": "Item X",
8   "valorUnitario": 10.0,
9   "estocavel": true
10 }
```

Por que deu 500?

Response

500

HEADERS ⓘ

Content-Type: application/json
Transfer-Encoding: chunked
Date: Sun, 08 Oct 2023 20:01:00 GMT
Connection: close

▶ COMPLETE REQUEST HEADERS

pretty ▼

BODY ⓘ

```
{  
  timestamp: "2023-10-08T20:01:00.145+00:00",  
  status: 500,  
  error: "Internal Server Error",  
  path: "/api/produtos"  
}
```

lines nums [copy](#)

```
import java.util.List;

@RestController
@RequestMapping("/api/produtos")
public class ProdutoController extends AbstractController {

    @Autowired
    private ProdutoService service;

    @PostMapping
    public ResponseEntity create(@RequestBody Produto entity) {
        Produto save = service.salvar(entity);
        return ResponseEntity.created(URI.create("/api/produtos/" + entity.getId())).body(save);
    }
}
```



Este bloco de código tratará nosso **ValidationException**

```
@ResponseStatus(HttpStatus.UNPROCESSABLE_ENTITY)
@ExceptionHandler(ValidationException.class)
public Map<String, String> handleValidationExceptions422(
    ValidationException ex) {
    Map<String, String> errors = new HashMap<>();
    errors.put("erro", ex.getMessage());
    return errors;
}
```


Response

422

HEADERS ⓘ

Content-Type: application/json
Transfer-Encoding: chunked
Date: Sun, 08 Oct 2023 20:08:39 GMT
Keep-Alive: timeout=60
Connection: keep-alive

▶ COMPLETE REQUEST HEADERS

pretty ▼


BODY ⓘ

```
{
  "erro": "O valor unitário não pode ser menor que o preço de compra do produto!"
}
```

lines nums [copy](#)

Agora vamos criar outra regra de negócio, uma de duplicidade

```
public Produto salvar(Produto entity) {  
  
    if (entity.getValorUnitario() < entity.getPrecoCompra()) {  
        throw new ValidationException("O valor unitário não pode ser menor que o preço de compra do produto!");  
    }  
  
    if (!repository.findAll(QProduto.produto.descricao.eq(entity.getDescricao())).isEmpty()) {  
        throw new ValidationException("Já existe um produto com essa descrição cadastrado!");  
    }  
  
    return repository.save(entity);  
}
```



Response

422

HEADERS ⓘ

Content-Type: application/json
Transfer-Encoding: chunked
Date: Sun, 08 Oct 2023 20:18:02 GMT
Keep-Alive: timeout=60
Connection: keep-alive

▶ COMPLETE REQUEST HEADERS

pretty ▼

BODY ⓘ

```
{  
  erro: "Já existe um produto com essa descrição cadastrado!"  
}
```

[lines](#) [nums](#) [copy](#)

javax.validation.constraints

O pacote `javax.validation.constraints` faz parte da API de validação do Java, conhecida como Bean Validation. Essas anotações são utilizadas para definir regras de validação em atributos de classes Java, permitindo que você especifique restrições nos dados que seus objetos podem conter.

@NotNull

@Size(min = 2, max = 50)

@Min(18)

@Max(100)

@Pattern(regex = "[a-zA-Z0-9]+")

@Email

@NotBlank

@NotEmpty:

Vamos adicionar as seguintes dependências

```
<dependency>
  <groupId>javax.validation</groupId>
  <artifactId>validation-api</artifactId>
</dependency>

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-validation</artifactId>
  <version>2.7.16</version>
</dependency>
```



```
@Entity
@DiscriminatorValue("produto")
public class Produto extends ItemVendavel {

    @NotNull(message = "O Nome do produto deve ser informado")
    @Column(name = "nome")
    private String nome;

    @Column(name = "preco_compra")
    private Double precoCompra;

    @Column(name = "dt_validade")
    private LocalDate dataValidade;

    @Column(name = "dt_prazo")
    private LocalDate dataPrazo;

    @Enumerated(EnumType.STRING)
    @Column(name = "status")
    private Status status;
```





```
@Autowired
private ProdutoService service;

@PostMapping
public ResponseEntity create(@RequestBody @Valid Produto entity) {
    Produto save = service.salvar(entity);
    return ResponseEntity.created(URI.create("/api/produtos/" + entity.getId())).body(save);
}
```

DRAFT

METHOD

POST

SCHEME :// HOST [":" PORT] [PATH ["?" QUERY]]

http://localhost:8080/api/produtos

▼ QUERY PARAMETERS

+ Add query parameter

HEADERS ⓘ ⚙



Content-Type

: application/json



+ Add header

🔗 Add authorization

Form ▼

BODY ⓘ

```
1 {  
2   "nome": null,  
3   "precoCompra": 50.0,  
4   "dataValidade": "2023-12-31",  
5   "dataPrazo": "2023-11-30",  
6   "status": "DISPONIVEL",  
7   "descricao": "Item 123123",  
8   "valorUnitario": 60.0,  
9   "estocavel": true  
10 }
```

Text JSON XML HTML | Format body | Enable body evaluation

Response

400

HEADERS ⓘ

Content-Type: application/json
Transfer-Encoding: chunked
Date: Sun, 08 Oct 2023 20:56:30 GMT
Connection: close

▶ COMPLETE REQUEST HEADERS

pretty ▼

BODY ⓘ

```
{  
  "erro": "[nome O Nome do produto deve ser informado]"  
}
```

lines nums  copy

Utilizando QueryDSL nas nossas APIs

Vamos criar um pacote, **enterprise**


Criem uma classe chamada **BooleanBuilderUtil** o conteúdo está no meu drive.

Agora vamos fazer um ajuste na nossa interface
CustomQuerydslPredicateExecutor


```
public interface CustomQuerydslPredicateExecutor<T> extends QuerydslPredicateExecutor<T> {  
  
    @Override  
    List<T> findAll(Predicate predicate);  
  
    default List<T> findAll(String filter, Class<T> entityType) {  
        BooleanBuilder booleanBuilder = BooleanBuilderUtil.buildPredicateFromFilter(filter, entityType);  
        return this.findAll(booleanBuilder);  
    }  
}
```

Vamos alterar nosso **ProdutoService** para utilizar nosso `findAll` customizado.

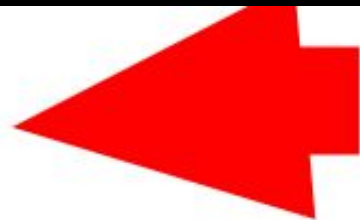
```
public List<Produto> buscaTodos(String filter) {  
    return repository.findAll(filter, Produto.class);  
}
```



Vamos alterar nosso **ProdutoController** para utilizar nosso **buscaTodos**.

@GetMapping

```
public ResponseEntity findAll(@RequestParam(required = false) String filter) {  
    List<Produto> produtos = service.buscaTodos(filter);  
    return ResponseEntity.ok(produtos);  
}
```



DRAFT

METHOD

GET

SCHEME :// HOST [":" PORT] [PATH ["?" QUERY]]

http://localhost:8080/api/produtos

length: 34 byte(s)

Send

QUERY PARAMETERS

+ Add query parameter

HEADERS

Form

BODY

+ Add header

Add authorization

XHR does not allow payloads for GET request.

Response

Cache Detected - Elapsed Time: 1ms

200

HEADERS

pretty

BODY

pretty

Content-Type: application/json

Transfer-Encoding: chunked

Date: Sun, 15 Oct 2023 13:09:13 GMT -1s

Keep-Alive: timeout=60

Connection: keep-alive

COMPLETE REQUEST HEADERS

```
[
  {
    id: 6,
    descricao: "Item X",
    valorUnitario: 10.0,
    estocavel: true,
    nome: "Produto A",
    precoCompra: 50.0,
    dataValidade: "2023-12-31",
    dataPrazo: "2023-11-30",
    status: "DISPONIVEL"
  },
]
```

DRAFT

METHOD

GET

SCHEME :// HOST [":" PORT] [PATH ["?" QUERY]]

http://localhost:8080/api/produtos?filter=descricao+like+I5 16gb

▼ QUERY PARAMETERS ↓₂

filter

=

descricao+like+I5 16gb

+ Add query parameter

HEADERS [?]

Form ▼

BODY [?]+ Add header [?]

Add authorization

XHR does not allow payloads for GET request.

▶ BODY ⓘ

pretty ▼

```
▼ [
  ▼ {
    id: 19,
    descricao: "Computador I5 16gb completo mais Gabinete",
    valorUnitario: 2000.0,
    estocavel: true,
    nome: "Computador I5 16gb",
    precoCompra: 1500.0,
    dataValidade: "2023-12-31",
    dataPrazo: "2023-11-30",
    status: "DISPONIVEL"
  }
]
```

lines nums  copy

length: 231 bytes

DRAFT

METHOD

SCHEME :// HOST [":" PORT] [PATH ["?" QUERY]]

GET

http://localhost:8080/api/produtos?filter=descricao+like+I5

▼ QUERY PARAMETERS ↓



filter

=

descricao+like+I5

+ Add query parameter

HEADERS ⓘ

Form ▼



BODY ⓘ

+ Add header ⓘ



Add authorization

XHR does not allow payloads for GET request.

BODY

pre

```
[
  {
    id: 18,
    descricao: "Computador I5 8gb completo mais Gabinete",
    valorUnitario: 2000.0,
    estocavel: true,
    nome: "Computador I5 8gb",
    precoCompra: 1500.0,
    dataValidade: "2023-12-31",
    dataPrazo: "2023-11-30",
    status: "DISPONIVEL"
  },
  {
    id: 19,
    descricao: "Computador I5 16gb completo mais Gabinete",
    valorUnitario: 2000.0,
    estocavel: true,
    nome: "Computador I5 16gb",
    precoCompra: 1500.0,
    dataValidade: "2023-12-31",
    dataPrazo: "2023-11-30",
    status: "DISPONIVEL"
  },
  {
    id: 20,
    descricao: "Computador I5 32gb completo mais Gabinete",
    valorUnitario: 2000.0,
    estocavel: true,
    nome: "Computador I5 32gb",
    precoCompra: 1500.0,
    dataValidade: "2023-12-31",
    dataPrazo: "2023-11-30",
    status: "DISPONIVEL"
  }
]
```

DRAFT

METHOD

GET

SCHEME :// HOST [":" PORT] [PATH ["?" QUERY]]

http://localhost:8080/api/produtos?filter=valorUnitario+greaterEqual+2700

▼ QUERY PARAMETERS 1/2



filter

=

valorUnitario+greaterEqual+2700

+ Add query parameter

HEADERS ?

Form ▼



BODY ?

+ Add header



Add authorization

XHR does not allow payloads for GET request.

```
▼ [
  ▼ {
    id: 23,
    descricao: "[Gamer] Computador I7 16gb completo mais Gabinete",
    valorUnitario: 2700.0,
    estocavel: true,
    nome: "Computador I7 16gb",
    precoCompra: 1500.0,
    dataValidade: "2023-12-31",
    dataPrazo: "2023-11-30",
    status: "DISPONIVEL"
  },
  ▼ {
    id: 24,
    descricao: "[Gamer] Computador I7 32gb completo mais Gabinete",
    valorUnitario: 2900.0,
    estocavel: true,
    nome: "Computador I7 32gb",
    precoCompra: 1500.0,
    dataValidade: "2023-12-31",
    dataPrazo: "2023-11-30",
    status: "DISPONIVEL"
  }
]
```

DRAFT

METHOD SCHEME :// HOST [":" PORT] [PATH ["?" QUERY]]

GET ▼

▼ QUERY PARAMETERS ⌵

☒ filter = valorUnitario+greater+2700

[+ Add query parameter](#)

HEADERS ? Form ▼ ◀ ▶ BODY ?

[+ Add header](#) [Add authorization](#)

XHR does not allow payloads for GET request.

```
▼ [
  ▼ {
    id: 24,
    descricao: "[Gamer] Computador I7 32gb completo mais Gabinete",
    valorUnitario: 2900.0,
    estocavel: true,
    nome: "Computador I7 32gb",
    precoCompra: 1500.0,
    dataValidade: "2023-12-31",
    dataPrazo: "2023-11-30",
    status: "DISPONIVEL"
  }
]
```

DRAFT

METHOD

GET

SCHEME :// HOST [":" PORT] [PATH ["?" QUERY]]

http://localhost:8080/api/produtos?filter=valorUnitario+between+2700+2800

▼ QUERY PARAMETERS ↓^A₂

filter

=

valorUnitario+between+2700+2800

+ Add query parameter

HEADERS ⓘ

Form ▼

BODY ⓘ

+ Add header

🔗 Add authorization

XHR does not allow payloads for GET request.

```
▼ [
  ▼ {
    id: 23,
    descricao: "[Gamer] Computador I7 16gb completo mais Gabinete",
    valorUnitario: 2700.0,
    estocavel: true,
    nome: "Computador I7 16gb",
    precoCompra: 1500.0,
    dataValidade: "2023-12-31",
    dataPrazo: "2023-11-30",
    status: "DISPONIVEL"
  }
]
```


DRAFT

METHOD

GET

SCHEME :// HOST [":" PORT] [PATH ["?" QUERY]]

http://localhost:8080/api/produtos?filter=dataValidade+between+2022-12-01+2022-12-31

▼ QUERY PARAMETERS ↕



filter

=

dataValidade+between+2022-12-01+2022-12-31

+ Add query parameter

HEADERS ⓘ

Form ▼



BODY ⓘ

+ Add header



Add authorization

XHR does not allow payloads for GET request.

▶ BODY ⓘ

pretty ▼

```
▼ [
  ▼ {
    id: 25,
    descricao: "[Gamer][Ajustado] Computador I7 32gb completo mais Gabi",
    valorUnitario: 2900.0,
    estocavel: true,
    nome: "Computador I7 32gb",
    precoCompra: 1500.0,
    dataValidade: "2022-12-30",
    dataPrazo: "2022-12-30",
    status: "DISPONIVEL"
  }
]
```

lines nums  copy

length: 249 bytes

DTO

DTO, ou Data Transfer Object, é um padrão de projeto de software que é usado para transferir dados entre diferentes partes de um sistema. Ele é especialmente útil em situações em que você precisa enviar dados de um local para outro, como entre camadas de uma aplicação ou entre sistemas distribuídos.

A ideia principal por trás do DTO é encapsular os dados em um objeto simples que pode ser facilmente transmitido pela rede ou passado entre diferentes partes de um programa. Geralmente, um DTO contém apenas os dados relevantes para uma operação específica, evitando o envio de informações desnecessárias.

Suponha que você tenha uma aplicação de e-commerce e deseja exibir informações sobre um produto em uma página web. Você pode ter uma classe Produto no seu backend com muitos detalhes, como nome, descrição, preço, quantidade em estoque, data de criação, etc.

Ao exibir uma lista de produtos em uma página, você não precisa enviar todos esses detalhes para o frontend, pois isso seria ineficiente e consumiria mais largura de banda. Em vez disso, você pode criar um DTO chamado **ProdutoDTO** que contém apenas os detalhes relevantes para a exibição, como nome, descrição e preço.

Criem uma classe chamada **ProdutoDTO** no pacote de resource.

ProdutoDTO

```
public class ProdutoDTO {  
  
    private Long id;  
    private String nome;  
    private String descricao;  
    private Status status;  
  
    //Criem os Getters e Setters
```

```
public static ProdutoDTO fromEntity(Produto produto) {  
    ProdutoDTO dto = new ProdutoDTO();  
    dto.setId(produto.getId());  
    dto.setNome(produto.getNome());  
    dto.setDescricao(produto.getDescricao());  
    dto.setStatus(produto.getStatus());  
    return dto;  
}  
  
public Produto toEntity() {  
    Produto produto = new Produto();  
    produto.setId(this.getId());  
    produto.setNome(this.getNome());  
    produto.setDescricao(this.getDescricao());  
    produto.setStatus(this.getStatus());  
    return produto;  
}
```

```
public static List<ProdutoDTO> fromEntity(List<Produto> produtos) {  
    return produtos.stream().map(produto -> fromEntity(produto)).collect(Collectors.toList());  
}
```

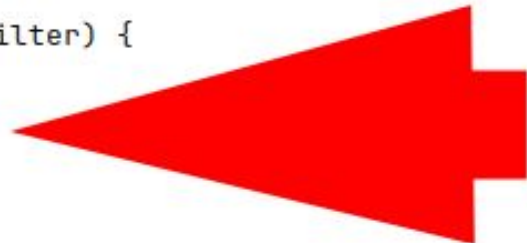
Hoje as apis retornam assim

```
▶ BODY ⓘ pretty ▼  
  
{  
  id: 25,  
  descricao: "[Gamer][Ajustado] Computador I7 32gb completo mais Gabinete",  
  valorUnitario: 2900.0,  
  estocavel: true,  
  nome: "Computador I7 32gb",  
  precoCompra: 1500.0,  
  dataValidade: "2022-12-30",  
  dataPrazo: "2022-12-30",  
  status: "DISPONIVEL"  
}
```

lines nums copy length: 247 bytes

Após uma leve modificação no ProdutoController

```
@GetMapping
public ResponseEntity findAll(@RequestParam(required = false) String filter) {
    List<Produto> produtos = service.buscaTodos(filter);
    List<ProdutoDTO> produtoDTOS = ProdutoDTO.fromEntity(produtos);
    return ResponseEntity.ok(produtoDTOS);
}
```



O retorno passa a ser esse

▶ BODY ?

```
[
  {
    id: 6,
    nome: "Produto A",
    descricao: "Item X",
    status: "DISPONIVEL"
  },
  {
    id: 7,
    nome: "Produto A",
    descricao: "Item X",
    status: "DISPONIVEL"
  },
]
```

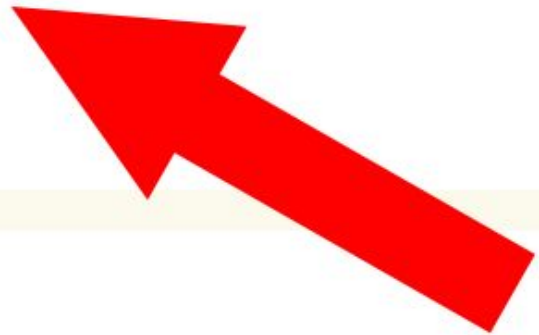
Dessa forma podemos definir, como nossos objetos serão retornados;

Paginação

A paginação é um mecanismo utilizado para dividir grandes conjuntos de dados em partes menores, chamadas de "páginas". Isso facilita a apresentação e a navegação pelos dados de forma mais eficiente. Em uma aplicação web, a paginação é comumente usada para exibir grandes listas de itens em várias páginas, evitando a sobrecarga de carregar todos os itens de uma só vez.

Agora vamos fazer um ajuste na nossa interface
CustomQuerydslPredicateExecutor

```
public interface CustomQuerydslPredicateExecutor<T> extends QuerydslPredicateExecutor<T> {  
  
    @Override  
    List<T> findAll(Predicate predicate);  
  
    default List<T> findAll(String filter, Class<T> entityType) {  
        BooleanBuilder booleanBuilder = BooleanBuilderUtil.buildPredicateFromFilter(filter, entityType);  
        return this.findAll(booleanBuilder);  
    }  
  
    default Page<T> findAll(String filter, Class<T> entityType, Pageable pageable) {  
        BooleanBuilder booleanBuilder = BooleanBuilderUtil.buildPredicateFromFilter(filter, entityType);  
        return this.findAll(booleanBuilder, pageable);  
    }  
}
```



Vamos alterar nosso **ProdutoService** para utilizar nosso `findAll` customizado.

```
public List<Produto> buscaTodos(String filter) {  
    return repository.findAll(filter, Produto.class);  
}  
  
public Page<Produto> buscaTodos(String filter, Pageable pageable) {  
    return repository.findAll(filter, Produto.class, pageable);  
}
```

Vamos alterar nosso **ProdutoController** para utilizar nosso **buscaTodos** paginado.

```
@GetMapping
public ResponseEntity findAll(@RequestParam(required = false) String filter,
                              @RequestParam(defaultValue = "0") int page,
                              @RequestParam(defaultValue = "10") int size) {
    Page<Produto> produtos = service.buscaTodos(filter, PageRequest.of(page, size));
    List<ProdutoDTO> produtoDTOS = ProdutoDTO.fromEntity(produtos.getContent());
    return ResponseEntity.ok(produtoDTOS);
}
```



DRAFT

METHOD

GET

SCHEME :// HOST [":" PORT] [PATH ["?" QUERY]]

http://localhost:8080/api/produtos?page=0&limit=10

Save as

Send

length: 50 byte(s)

QUERY PARAMETERS



page

=

0



limit

=

10



+ Add query parameter

HEADERS

Form

BODY

+ Add header

Add authorization

XHR does not allow payloads for GET request.

Response

Error Detected - Elapsed Time: 247ms

200

HEADERS

pretty

BODY

pretty

Content-Type: application/json
Transfer-Encodin... chunked
Date: Sun, 15 Oct 2023 14:51:05 GMT
Keep-Alive: timeout=60
Connection: keep-alive

COMPLETE REQUEST HEADERS

```
[
  {
    id: 6,
    nome: "Produto A",
    descricao: "Item X",
    status: "DISPONIVEL"
  },
  {
    id: 7,
    nome: "Produto A",
    descricao: "Item X",
    status: "DISPONIVEL"
  }
]
```

METHOD

GET

SCHEME // HOST [":" PORT] [PATH ["?" QUERY]]

http://localhost:8080/api/produtos?page=1&limit=10

Save as

Send

length: 50 bytes

Send request (Alt + Enter)

QUERY PARAMETERS 1

☒ page

= 1

☒ limit

= 10

+ Add query parameter

HEADERS

Form

BODY

+ Add header

Add authorization

XHR does not allow payloads for GET request.

Response

Cache Detected - Elapsed Time: 1ms

200

HEADERS

pretty

BODY

pretty

Content-Type: application/json

Transfer-Encoding: chunked

Date: Sun, 15 Oct 2023 14:51:42 GMT

Keep-Alive: timeout=60

Connection: keep-alive

▶ COMPLETE REQUEST HEADERS

```
[
  {
    id: 15,
    nome: null,
    descricao: "Item 123123",
    status: "DISPONIVEL"
  },
  {
    id: 17,
    nome: "Produto B",
    descricao: "Item XWB",
    status: "DISPONIVEL"
  },
  {
    id: 18,
    nome: "Produto C",
    descricao: "Item YWB",
    status: "DISPONIVEL"
  }
]
```


O ideal é que quando retornarmos os objetos tragamos as informações das páginas.

```
@GetMapping
public ResponseEntity findAll(@RequestParam(required = false) String filter,
                             @RequestParam(defaultValue = "0") int page,
                             @RequestParam(defaultValue = "10") int size) {
    Page<Produto> produtos = service.buscaTodos(filter, PageRequest.of(page, size));
    List<ProdutoDTO> produtoDTOS = ProdutoDTO.fromEntity(produtos.getContent());
    Page<ProdutoDTO> produtosDTOPage = new PageImpl<>(produtoDTOS, produtos.getPageable(), produtos.getTotalElements());
    return ResponseEntity.ok(produtosDTOPage);
}
```

METHOD SCHEME :// HOST [":" PORT] [PATH ["?" QUERY]]

GET http://localhost:8080/api/produtos?page=0&size=10

▼ QUERY PARAMETERS 1/2

<input checked="" type="checkbox"/>	page	=	0
<input checked="" type="checkbox"/>	size	=	10

[+ Add query parameter](#)

▶ BODY ? pretty ▼

```
{
  content: [
    {id: 15, nome: null, descricao: "Item 123123", status: "DISPO"},
    {id: 17, nome: "Produto B", descricao: "Item XWB", status: "D"},
    {id: 18, nome: "Computador I5 8gb", descricao: "Computador I5"},
    {id: 19, nome: "Computador I5 16gb", descricao: "Computador I5"},
    {id: 20, nome: "Computador I5 32gb", descricao: "Computador I5"},
    {id: 21, nome: "Computador I7 8gb", descricao: "Computador I7"},
    {id: 22, nome: "Computador I7 16gb", descricao: "Computador I7"},
    {id: 23, nome: "Computador I7 16gb", descricao: "[Gamer] Compu"},
    {id: 24, nome: "Computador I7 32gb", descricao: "[Gamer] Compu"},
    {id: 25, nome: "Computador I7 32gb", descricao: "[Gamer][Ajuste"}
  ],
  pageable: {sort: {empty: true, sorted: false, unsorted: true }, off
last: true,
totalElements: 20,
totalPages: 2,
size: 10,
number: 1,
sort: {empty: true, sorted: false, unsorted: true},
first: false,
numberOfElements: 10,
empty: false
}
```

METHOD GET SCHEME `://` HOST `["*" PORT]` [PATH `["?" QUERY]`]

`http://localhost:8080/api/produtos?page=0&size=15`

QUERY PARAMETERS 1/2

<input checked="" type="checkbox"/>	page	=	0
<input checked="" type="checkbox"/>	size	=	15

[+ Add query parameter](#)

HEADERS ? Form ▼ ▶ BODY ?

[+ Add header](#) [Add authorization](#)

▶ BODY ?

pretty ▼

```
{
  content: [
    {id: 6, nome: "Produto A", descricao: "Item X", status: "DISP"},
    {id: 7, nome: "Produto A", descricao: "Item X", status: "DISP"},
    {id: 5, nome: "Produto AZZZZ", descricao: "Item XYZW", status: "DISP"},
    {id: 8, nome: null, descricao: "Item Xdsdfdsfdfsfd", status: "DISP"},
    {id: 9, nome: null, descricao: "Item Xdsdfdsfdfsfdasd", status: "DISP"},
    {id: 10, nome: null, descricao: "Item Xdsdfdsfdfsfdasdasdd", status: "DISP"},
    {id: 11, nome: null, descricao: "Item asd", status: "DISPONIVEL"},
    {id: 12, nome: null, descricao: "Item asasdd", status: "DISPONIVEL"},
    {id: 13, nome: null, descricao: "Item asasddasd", status: "DISPONIVEL"},
    {id: 14, nome: null, descricao: "Item 123", status: "DISPONIVEL"},
    {id: 15, nome: null, descricao: "Item 123123", status: "DISPONIVEL"},
    {id: 17, nome: "Produto B", descricao: "Item XWB", status: "DISPONIVEL"},
    {id: 18, nome: "Computador I5 8gb", descricao: "Computador I5 8gb"},
    {id: 19, nome: "Computador I5 16gb", descricao: "Computador I5 16gb"},
    {id: 20, nome: "Computador I5 32gb", descricao: "Computador I5 32gb"}
  ],
  pageable: {sort: {empty: true, sorted: false, unsorted: true}, offset: 0, last: false, totalElements: 20, totalPages: 2, size: 15, number: 0, sort: {empty: true, sorted: false, unsorted: true}, first: true, numberOfElements: 15, empty: false}
}
```

Na classe ProdutoDTO

```
public static Page<ProdutoDTO> fromEntity(Page<Produto> produtos) {  
    List<ProdutoDTO> produtosFind = produtos.stream().map(produto -> fromEntity(produto)).collect(Collectors.toList());  
    Page<ProdutoDTO> produtosDTO = new PageImpl<>(produtosFind, produtos.getPageable(), produtos.getTotalElements());  
    return produtosDTO;  
}
```

Na classe **ProdutoController**

```
@GetMapping
public ResponseEntity findAll(@RequestParam(required = false) String filter,
                              @RequestParam(defaultValue = "0") int page,
                              @RequestParam(defaultValue = "10") int size) {
    Page<Produto> produtos = service.buscaTodos(filter, PageRequest.of(page, size));
    Page<ProdutoDTO> produtoDTOS = ProdutoDTO.fromEntity(produtos);
    return ResponseEntity.ok(produtoDTOS);
}
```

Documentação de API - Swagger

O Swagger é, basicamente, um conjunto de ferramentas que nos ajuda a fazer o design, ou seja, fazer a modelagem, a documentar e até gerar código para desenvolvimento de APIs.

No pom.xml adicionamos a seguinte dependência

```
<dependency>  
  <groupId>io.springfox</groupId>  
  <artifactId>springfox-boot-starter</artifactId>  
  <version>3.0.0</version>  
</dependency>
```

Criaremos a classe **SwaggerConfig**

```
@Configuration
@EnableSwagger2
public class SwaggerConfig {

    @Bean
    public Docket api() {
        return new Docket(DocumentationType.SWAGGER_2)
            .select()
            .apis(RequestHandlerSelectors.basePackage("com.satc.satcloja.resource"))
            .build()
            .pathMapping("/")
            .apiInfo(apiInfo());
    }

    private ApiInfo apiInfo() {
        return new ApiInfoBuilder()
            .title("Nome da Sua API")
            .description("Descrição da Sua API")
            .version("1.0")
            .build();
    }
}
```

Vamos habilitar o @EnableWebMvc



```
@EnableWebMvc
@SpringBootApplication
public class SatcLojaApplication {

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

}
```

Vamos reiniciar a aplicação

<http://localhost:8080/swagger-ui/index.html#/>

Nome da Sua API ^{1.0}

[Base URL: localhost:8080/]
<http://localhost:8080/v2/api-docs>

Descrição da Sua API

produto-controller Produto Controller

GET

/api/produtos

Retorna uma lista de produtos

POST

/api/produtos

create

GET

/api/produtos/{id}

findById

PUT

/api/produtos/{id}

update

DELETE

/api/produtos/{id}

remove

Models

Produto >

Loja de Informatica ^{1.0}

[Base URL: localhost:8080/]

<http://localhost:8080/v2/api-docs>

[Documentação completa](#)

produto-controller Produto Controller

GET /api/produtos Retorna uma lista de produtos

POST /api/produtos create

GET /api/produtos/{id} findById

PUT /api/produtos/{id} update

DELETE /api/produtos/{id} remove

Models

Produto >

ResponseEntity >

Models

```
Produto {
  dataPrazo          string($date)
  dataValidade       string($date)
  descricao          string
  estocavel          boolean
  id                 integer($int64)
  nome*              string
  precoCompra        number($double)
  precoVenda         number($double)
  status             string
  Enum:
  > Array [ 2 ]
  valorUnitario     number($double)
}
```

Testes de API

Fim da aula 09...