Programação Web

Estilo Arquitetural REST, Model Validation, Error Handling, HATEOAS, DTO, CORS Policy, Security

Universidade Federal do Rio Grande do Norte

Criar um novo projeto Spring

- Dependências
  - Spring Web
  - Spring Data JPA
  - Spring Boot Dev Tools
  - Postgres Driver
  - Lombok

REST

- Representational State Transfer
  - Transferência de Estado Representacional
  - 2000 por Roy Fielding, Tese de Doutorado
- **Estilo Arquitetural** aplicado no desenvolvimento de serviços Web
- A maioria das aplicações Web modernas expõe APIs que os clientes podem usar para interagir com o sistema.

REST

- REST é um estilo de arquitetura para construir sistemas distribuídos baseados em hipermídia.
- O REST é independente de qualquer protocolo subjacente e não está necessariamente vinculado ao HTTP. No entanto, as implementações de API REST mais comuns usam HTTP como protocolo aplicação
- Uma vantagem principal do REST sobre HTTP é que ele usa padrões abertos e não vincula a implementação da API ou dos aplicativos cliente a nenhuma implementação específica.

REST

- Uma API da Web bem projetada deve ter como objetivo oferecer suporte para:
  - **Independência de plataforma.** Qualquer cliente deve ser capaz de chamar a API, independentemente de como a API é implementada internamente.
  - **Evolução do serviço.** A API da Web deve ser capaz de evoluir e adicionar funcionalidades independentemente dos aplicativos cliente. À medida que a API evolui, os aplicativos cliente existentes devem continuar a funcionar sem modificações.

Definição de Recursos

- Um recurso é utilizado para identificar de forma única um objeto abstrato ou físico.
- Exemplos:
  - /clientes/1
  - /produtos/1
  - /clientes/1/notificação
- Um recurso tem um identificador, que é uma URI que identifica exclusivamente esse recurso. Por exemplo, o URI para um determinado pedido pode ser:
  - /pedidos/1

Definição de Recursos

- O formato dos recursos pode variar de acordo com a biblioteca usada para implementação do serviço
- Exemplos de formatos:
  - XML
  - JSON

Definição de Recursos

```xml
<cliente>
  <id>10</id>
  <nome>Alan Turing</nome>
  <nascimento>23/06/1912</nascimento>
  <profissao>Matemático</profissao>
  <endereco>
      <cidade>Manchester</cidade>
      <pais>Inglaterra</pais>
  </endereco>
</cliente>
```

```json
{
    "id": 10,
    "nome": "Alan Turing",
    "nascimento": "23/06/1912",
    "profissao": "Matemático",
    "endereco": {
        "cidade": "Manchester",
        "pais": "Inglaterra"
    }
}
```

Interface

- As APIs REST usam uma interface uniforme, que ajuda a desacoplar as implementações de cliente e serviço.
- Para APIs REST criadas em HTTP, a interface uniforme inclui o uso de verbos HTTP padrão para executar operações em recursos.
- As operações mais comuns são GET, POST, PUT, PATCH e DELETE.

Estado

- As APIs REST usam um **modelo de solicitação sem estado**.
- As solicitações HTTP devem ser independentes e podem ocorrer em qualquer ordem, portanto, manter as informações de estado transitório entre as solicitações não é viável.
- O único local onde as informações são armazenadas é nos próprios recursos, e cada solicitação deve ser uma operação atômica.
- Essa restrição permite que os serviços da Web sejam altamente escaláveis, pois não há necessidade de manter nenhuma relação entre clientes e servidores específicos.

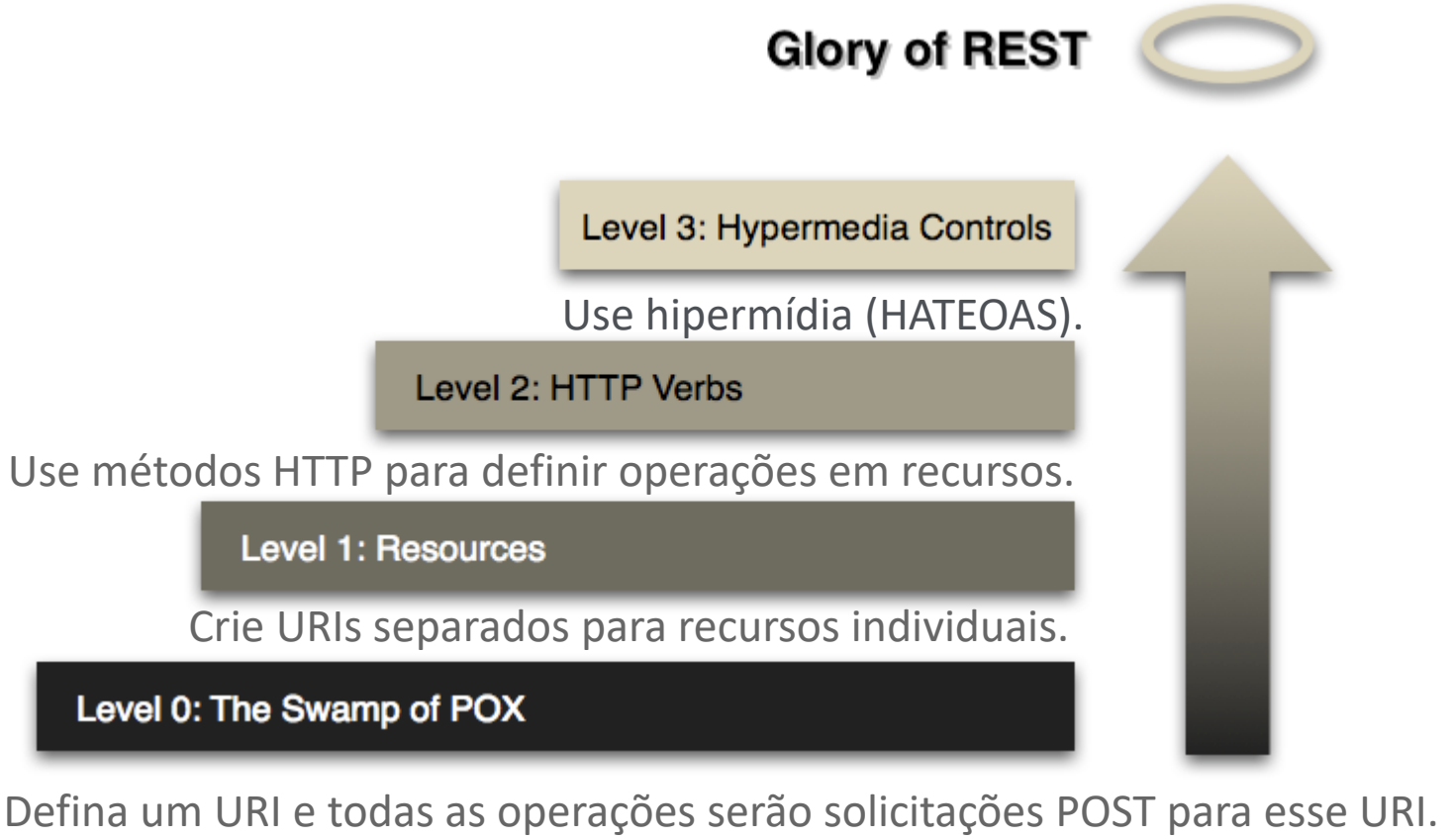
Representação

- As APIs REST são orientadas por links de hipermídia que estão contidos na representação.
- Por exemplo, o seguinte mostra uma representação JSON de um pedido. Ele contém links para obter ou atualizar o cliente associado ao pedido.

```
{
    "orderID":3,
    "productID":2,
    "quantity":4,
    "orderValue":16.60,
    "links": [
        {"rel":"product","href":"https://adventure-works.com/customers/3", "action":"GET" },
        {"rel":"product","href":"https://adventure-works.com/customers/3", "action":"PUT" }
    ]
}
```

Richardson Maturity Model

- Em 2008, Leonard Richardson propôs o seguinte modelo de maturidade para APIs web
- As categorias para serviços REST:
  - Level Zero Services
  - Level One Services
  - Level Two Services
  - Level Three Services



Richardson Maturity Model

- O nível 3 corresponde a uma API verdadeiramente RESTful de acordo com a definição de Fielding.
- Na prática, muitas APIs da Web publicadas ficam em algum lugar em torno do nível 2.

HATEOAS

- *Hypermedia As The Engine Of Application State*
- Os recursos possuem links que apontam para outros recursos.
- Nesse nível de especificação clientes podem consumir o serviço sem a necessidade de conhecimento prévio profundo da API.
- Redução no acoplamento entre View e Controllers

HATEOAS

- Não existe uma definição de quais links devem ser exibidos em um recurso. Cada API determina quais links irão aparecer. A sugestão é que sejam adicionados os links para navegação com base nos links suportados pelo controlador.
- A uma RFC (5598) define os formatos de links.

```
{
“id”: 1,
“nome”: “João”,
...
_links: {
          “self”: { “href”: “http://domain/cliente/1”,
          “update”: { “href”: “http://domain/cliente/1”,
          “delete”: {“href”: “http://domain/cliente/1”]
}
```

Estrutura Geral do Projeto

- Separação de responsabilidades é fundamental: cada pacote deve conter apenas classes relacionadas à sua função específica.
- Essa estrutura facilita a manutenção, testes, escalabilidade e a entrada de novos desenvolvedores.
- A estrutura pode ser adaptada conforme o porte e necessidades do projeto, mas seguir essas práticas é altamente recomendado para projetos profissionais e de médio/grande porte.

Estrutura Geral do Projeto

Pacote	Função Principal	Estrutura Típica
controller	Recebe e processa as requisições HTTP (REST), delegando para os serviços.	Classes anotadas com @RestController ou @Controller, métodos mapeados com @RequestMapping
dto	Objetos de transferência de dados entre camadas, sem lógica de negócio.	Classes simples, apenas atributos, construtores, getters/setters
mapper	Faz a conversão entre entidades do domínio e DTOs (e vice-versa).	Interfaces/classes, geralmente usando MapStruct ou conversão manual6
domain	Representa o núcleo do negócio: entidades (modelos) e lógicas de domínio.	Classes anotadas com @Entity, enums, value objects, agregados.
repository	Responsável pelo acesso e manipulação dos dados no banco, via Spring Data JPA.	Interfaces que estendem JpaRepository ou CrudRepository.
service	Implementa a lógica de negócio e orquestra as operações entre repositórios e outras camadas.	Classes anotadas com @Service, métodos de negócio, validações, regras de aplicação.
core	Itens gerais e utilitários da aplicação, como helpers, constantes, util classes, exceções comuns.	Classes utilitárias, helpers, constantes globais, exceções genéricas.
errorhandling	Centraliza o tratamento de erros e exceções, geralmente com ControllerAdvice.	Classes anotadas com @ControllerAdvice, métodos com @ExceptionHandler.
config	Configurações gerais da aplicação: beans, segurança, CORS, propriedades customizadas.	Classes anotadas com @Configuration, configurações de beans, datasources, profiles, etc
security	Implementa regras de autenticação e autorização, filtros, configurações de segurança.	Classes de configuração de segurança, filtros JWT, providers, detalhes de usuário, etc.
base	Classes abstratas e genéricas para reaproveitamento (ex: BaseEntity, BaseService).	Classes abstratas, interfaces genéricas, superclasses para entidades ou serviços.

Estrutura Geral do Projeto

- **controller:** Responsável por expor endpoints REST, receber requisições, validar entradas e retornar respostas (geralmente usando DTOs). Não deve conter lógica de negócio, apenas orquestração e delegação para os serviços.
- **dto:** Data Transfer Objects encapsulam apenas os dados necessários para comunicação entre camadas ou para o frontend, evitando expor entidades do domínio diretamente.
- **mapper:** Facilita a conversão entre entidades e DTOs, centralizando a lógica de mapeamento e evitando duplicidade de código. Pode usar frameworks como MapStruct para gerar código automaticamente.
- **domain:** Contém as entidades do negócio (ex: User, Order), regras de domínio, enums e value objects. É o núcleo da aplicação, refletindo o modelo de dados e regras essenciais.

Estrutura Geral do Projeto

- **repository:** Define as interfaces para acesso aos dados, geralmente estendendo JpaRepository. Não contém lógica de negócio, apenas operações de persistência.
- **service:** Implementa as regras de negócio, validações, integrações entre diferentes repositórios e lógica de aplicação. É a camada intermediária entre controller e repository.
- **core:** Agrupa utilitários, helpers, constantes e componentes que são usados em várias partes da aplicação, promovendo reuso e centralização de lógica comum.

Estrutura Geral do Projeto

- **errorhandling:** Centraliza o tratamento de exceções, melhorando a padronização das respostas de erro e desacoplando o tratamento de erros das controllers. Usa @ControllerAdvice e métodos com @ExceptionHandler.
- **config:** Armazena todas as configurações customizadas do Spring Boot, beans, configurações de datasources, CORS, profiles, etc. Facilita a manutenção e organização das configurações.
- **security:** Isola toda a lógica de segurança, como autenticação, autorização, filtros de requisição, configurações de endpoints protegidos, etc.
- **base:** Contém classes e interfaces abstratas que servem de base para outras classes do projeto, promovendo reuso e padronização (ex: BaseEntity com id, BaseService com operações CRUD genéricas).

Controlador Básico para API REST em Spring Web

```java
@RestController
@RequestMapping("/cliente")
public class ClienteController {
    private ClienteService service;

    public ClienteController(ClienteService service) {
        this.service = service;
    }

}
```

Implementação de API REST

- A RFC 7231 3 especifica um conjunto de 8 métodos os quais podemos utilizar para a criação de uma API RESTful.
- Dos 8 métodos os 5 mais usados em APIs RESTful são:
  - GET é utilizado quando existe a necessidade de se obter um recurso.
  - POST é utilizado para processamento de um recurso a partir do uso de uma representação.
  - PUT é utilizado como forma de atualizar ou inserir um determinado recurso.
  - PATCH executa uma atualização parcial de um recurso. O corpo da solicitação especifica o conjunto de alterações a serem aplicadas ao recurso.
  - DELETE tem como finalidade a remoção de um determinado recurso

GET

- Um método GET bem-sucedido geralmente retorna o código de status HTTP 200 (OK).
- Se o recurso não puder ser encontrado, o método deve retornar 404 (Not Found).
- Se a solicitação foi atendida, mas não há corpo de resposta incluído na resposta HTTP, ela deve retornar o código de status HTTP 204 (sem conteúdo);
  - Por exemplo, uma operação de pesquisa que não produz correspondências pode ser implementada com esse comportamento.

GET

```java
@GetMapping
public List<Cliente> listAll(){
    return service.getAll();
}

@GetMapping(path = {"/{id}"})
public ResponseEntity<Cliente> getOne(@PathVariable Long id){

    Optional<Cliente> clienteOptional = service.findById(id);

    if (clienteOptional.isEmpty()){
        return ResponseEntity.notFound().build();
    }else {

        Cliente cliente = clienteOptional.get();
        return ResponseEntity.ok().body(cliente);
    }
}
```

POST

- Se um método POST criar um novo recurso, ele retornará o código de status HTTP 201 (Criado). O URI do novo recurso é incluído no cabeçalho Location da resposta.
  - O corpo da resposta contém uma representação do recurso.
- Se o método fizer algum processamento, mas não criar um novo recurso, o método poderá retornar o código de status HTTP 200 e incluir o **resultado da operação no corpo da resposta**.
- Como alternativa, se não houver resultado a ser retornado, o método pode retornar o código de status HTTP 204 (sem conteúdo) sem corpo de resposta.
- Se o cliente inserir dados inválidos na solicitação, o servidor deverá retornar o código de status HTTP 400 (Solicitação inválida). O corpo da resposta pode conter informações adicionais sobre o erro ou um link para um URI que fornece mais detalhes.

POST

```java
@PostMapping
public Cliente insert(@RequestBody Cliente c){
    return service.insert(c);
}
```

```java
@PostMapping
public ResponseEntity<Cliente> create(@RequestBody Cliente pessoa) {

    Pessoa created = service.create(pessoa);

    URI location = ServletUriComponentsBuilder
            .fromCurrentRequest()
            .path("{id}")
            .buildAndExpand(created.getId())
            .toUri();

    return ResponseEntity.created(location).body(created);
}
```

```java
@PostMapping
@ResponseStatus(HttpStatus.CREATED)
public Cliente insert(@RequestBody Cliente c){

  return service.insert(c);
}
```

PUT

- Se um método PUT criar um novo recurso, ele retornará o código de status HTTP 201 (Criado), como no método POST.
- Se o método atualizar um recurso existente, ele retornará 200 (OK) ou 204 (Sem conteúdo).
- Em alguns casos, pode não ser possível atualizar um recurso existente. Nesse caso, considere retornar o código de status HTTP 409 (Conflito).
- Considere utilizar HTTP PUT para implementar atualizações em lote para vários recursos em uma coleção. A solicitação PUT deve especificar o URI da coleção e o corpo da solicitação deve especificar os detalhes dos recursos a serem modificados. Essa abordagem pode melhorar o desempenho.

PUT

```
@PutMapping("/{id}")
public ResponseEntity<?> updateJoia(@RequestBody Pessoa p, @PathVariable long id){

    Optional<Pessoa> pessoa = service.buscarPorId(id);
    if(pessoa.isPresent()){
        return ResponseEntity.ok(service.alterar(p));
    }
    return ResponseEntity.notFound().build();
}
```

```
@PutMapping("{id}")
@ResponseStatus(HttpStatus.NO_CONTENT)
public void update(@PathVariable Long id, @RequestBody Cliente cliente){
    service.update(cliente);
}
```

PATCH

- Com uma solicitação PATCH, o cliente envia um conjunto de atualizações para um recurso existente, na forma de um documento de correção.
- O servidor processa o documento de correção para realizar a atualização.
- O documento do patch não descreve todo o recurso, apenas um conjunto de alterações a serem aplicadas. A especificação do método PATCH (RFC 5789) não define um formato específico para documentos de correção. O formato deve ser inferido do tipo de mídia na solicitação.
- Uma das estratégias mais utilizadas é o JSON de mesclagem.
- Estudar JSON patch (RFC 6902)

```
{
    "name":"gizmo",
    "category":"widgets",
    "color":"blue",
    "price":10
}
```

```
{
    "price":12,
    "color":null,
    "size":"small"
}
```

DELETE

- Se a operação de exclusão for bem-sucedida, o servidor da Web deve responder com o código de status HTTP 204 (sem conteúdo), indicando que o processo foi tratado com sucesso, mas que o corpo da resposta não contém mais informações.
- Se o recurso não existir, o servidor web pode retornar HTTP 404 (não encontrado).

DELETE

```java
@DeleteMapping("{id}")
@ResponseStatus(HttpStatus.NO_CONTENT)
public void deleteById(@PathVariable("id") Long id) {
    service.deleteById(id);
}
```

```java
@DeleteMapping("/{id}")
public ResponseEntity<?> deleteJoiaById(@PathVariable long id){

    Optional<Pessoa> p = service.buscarPorId(id);
    if(p.isPresent()){
        service.remover(id);
        return ResponseEntity.noContent().build();
    }
    return ResponseEntity.notFound().build();
}
```

Operações assíncronas

- Operações POST, PUT, PATCH ou DELETE pode exigir um processamento que demora um pouco para ser concluído.
- Se você esperar a conclusão antes de enviar uma resposta ao cliente, isso poderá causar uma latência inaceitável. É importante considerar tornar a operação assíncrona.
- Retornar o código de status HTTP 202 (Accepted) para indicar que a solicitação foi aceita para processamento, mas não foi concluída.
- Expor um endpoint que retorne o status de uma solicitação assíncrona, para que o cliente possa monitorar o status pesquisando o endpoint de status. Inclua o URI do endpoint de status no cabeçalho Location da resposta 202.

```
HTTP/1.1 202 Accepted
Location: /api/status/12345
```

Operações assíncronas

- Se o cliente enviar uma solicitação GET para esse endpoint, a resposta deverá conter o status atual da solicitação. Opcionalmente, também pode incluir um tempo estimado para conclusão ou um link para cancelar a operação.

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "status":"In progress",
    "link": { "rel":"cancel", "method":"delete", "href":"/api/status/12345" }
}
```

- Se a operação assíncrona criar um novo recurso, o endpoint de status deverá retornar o código de status 303 (See Other) após a conclusão da operação. Na resposta 303, inclua um cabeçalho Location que forneça o URI do novo recurso.

```
HTTP/1.1 303 See Other
Location: /api/orders/12345
```

Filtrar e paginar dados

- A exposição de uma coleção de recursos por meio de um único URI pode fazer com que os aplicativos busquem grandes quantidades de dados quando apenas um subconjunto das informações é necessário.
  - Por exemplo, suponha que um aplicativo cliente precise encontrar todos os pedidos com um custo acima de um valor específico. Ele pode recuperar todos os pedidos do URI /pedidos e, em seguida, filtrar esses pedidos no lado do cliente. Claramente, este processo é altamente ineficiente.
- A API pode permitir a passagem de um filtro na string de consulta do URI, como /pedidos?minCusto=n. A API da Web é responsável por analisar e manipular o parâmetro minCusto na string de consulta e retornar os resultados filtrados no lado do servidor.

Filtrar e paginar dados

- Solicitações GET sobre recursos de coleção podem retornar um grande número de itens. Você deve projetar uma API da Web para limitar a quantidade de dados retornados por qualquer solicitação única.
- É importante oferecer suporte a strings de consulta que especificam o número máximo de itens a serem recuperados e um deslocamento inicial na coleção.
  - Impor um limite máximo no número de itens devolvidos, para ajudar a evitar ataques de negação de serviço. Para auxiliar os aplicativos clientes, as solicitações GET que retornam dados paginados também devem incluir alguma forma de metadados que indique o número total de recursos disponíveis na coleção.

Filtrar e paginar dados

```java
@GetMapping
public Page<Pessoa> listAll(Pageable pageable) {
    return service.listAll(pageable);
}
```

```java
@Override
public Page<T> listAll(Pageable pageable) {
    return repository.findAll(pageable);
}
```

```
GET /clientes?page=1&size=10
```


```json
{
  "content": [
    {
      "id": 1,
      "nome": "Maria",
      "idade": 30
    },
    {
      "id": 2,
      "nome": "João",
      "idade": 25
    }
  ],
  "pageable": {
    "sort": {
      "empty": true,
      "sorted": false,
      "unsorted": true
    },
    "offset": 0,
    "pageNumber": 0,
    "pageSize": 2,
    "paged": true,
    "unpaged": false
  },
  "totalPages": 5,
  "totalElements": 10,
  "last": false,
  "size": 2,
  "number": 0,
  "sort": {
    "empty": true,
    "sorted": false,
    "unsorted": true
  },
  "numberOfElements": 2,
  "first": true,
  "empty": false
}
```

Recursos

- Métodos disponibilizarem uma interface CRUD (Create, Read, Update e Delete) para manipulação de recursos
  - GET, POST, PUT, DELETE
- É importante lembrar que é possível fazer processamentos a partir de métodos GET e POST

Modelo

- URIs REST devem ser modeladas com substantivos
- Será o verbo HTTP que irá determinar a ação a ser tomada

Modelo

RPC (POX)		
Verbo HTTP	**URI**	**Ação**
GET	/buscarCliente/1	Visualizar
POST	/salvarCliente	Criar
POST	/alterarCliente/1	Alterar
GET/POST	/deletarCliente/1	Remover

Modelo

REST		
Verbo HTTP	**URI**	**Ação**
GET	/cliente/1	Visualizar
POST	/cliente	Criar
PUT	/cliente/1	Alterar
DELETE	/cliente/1	Remover

Modelo

- Um recurso não precisa ser baseado em um único item de dados físicos.
- Por exemplo, um recurso de pedido pode ser implementado internamente como várias tabelas em um banco de dados relacional, mas apresentado ao cliente como uma única entidade.
- Evite criar APIs que simplesmente espelham a estrutura interna de um banco de dados. O objetivo do REST é modelar entidades e as operações que um aplicativo pode executar nessas entidades.
- Um cliente não deve ser exposto à implementação interna.

Modelo

- É importante adotar uma convenção de nomenclatura consistente em URIs.
- Em geral, é uma boa prática usar substantivos plurais para URIs que fazem referência a coleções.
- É uma boa prática organizar URIs para coleções e itens em uma hierarquia.
  - Por exemplo, /clientes é o caminho para a coleção de clientes e /clientes/5 é o caminho para o cliente com ID igual a 5.
- Essa abordagem ajuda a manter a API da web intuitiva.
- Além disso, muitas estruturas de API da Web podem rotear solicitações com base em caminhos de URI parametrizados, portanto, você pode definir uma rota para o caminho /clientes/{id}.

Modelo

- As relações entre diferentes tipos de recursos e como expor essas associações devem ser consideradas.
  - Por exemplo, /clientes/5/pedidos pode representar todos os pedidos do cliente 5.
- Você também pode ir na outra direção e representar a associação de um pedido de volta a um cliente com um URI como /pedidos/99/cliente.
- No entanto, estender demais esse modelo pode se tornar complicado de implementar. Uma solução melhor é fornecer links navegáveis (HATEOAS) para recursos associados no corpo da mensagem de resposta HTTP.

Modelo

- Solicitações da web impõem uma carga no servidor da web.
- Quanto mais solicitações, maior a carga. Portanto, é importante evitar APIs que expõem um grande número de pequenos recursos.
- Tal API pode exigir que um aplicativo cliente envie várias solicitações para localizar todos os dados necessários.
- Em vez disso, podem ser utilizadas estratégias para desnormalizar os dados e combinar informações relacionadas em recursos maiores que podem ser recuperados com uma única solicitação.
  - É necessário equilibrar essa abordagem com a sobrecarga de buscar dados que o cliente não precisa.
  - A recuperação de objetos grandes pode aumentar a latência de uma solicitação e incorrer em custos adicionais de largura de banda.

Model Validation

- A Validação de Bean funciona definindo restrições para os campos de uma classe, anotando-os com certas anotações.
  - @NotNull: campo não deve ser nulo.
  - @NotEmpty: campo não deve ser vazio.
  - @NotBlank: campo de string não deve ser a string vazia
  - @Min e @Max: campo numérico só é válido quando seu valor está acima ou abaixo de um determinado valor.
  - @Pattern: campo de string só é válido quando corresponde a uma determinada expressão regular.
  - @Email: campo de string deve ser um endereço de e-mail válido.

Model Validation

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

Model Validation

```
public class Cliente extends AbstractEntity {

    @NotBlank(message = "Nome não pode estar em branco ou vazio.")
    String nome;

}
```

BaseEntity

- Criar uma classe abstrata base entity que **centraliza atributos comuns relacionados ao banco de dados,** como id, dataCriacao, dataAtualizacao, é uma prática recomendada em projetos Java com JPA/Spring Data, pois traz diversos benefícios para a arquitetura e manutenção do sistema.

BaseEntity

- Principais vantagens e relação com o Princípio da Responsabilidade Única
  - **Redução de repetição de código:** Ao definir atributos e comportamentos comuns (por exemplo, o campo id e timestamps) na classe base, evita-se duplicar essas definições em todas as entidades do domínio. **Isso facilita a manutenção e reduz a chance de inconsistências.**
  - **Facilita a padronização:** Todas as entidades que estendem a base entity seguem o mesmo **padrão de identificação e auditoria**, tornando o sistema mais coeso e previsível.

BaseEntity

- Principais vantagens e relação com o Princípio da Responsabilidade Única
  - **Aproveitamento de recursos do JPA:** Utilizando a anotação @MappedSuperclass, a base entity não gera uma tabela própria no banco, **mas seus atributos são herdados pelas entidades concretas, otimizando o modelo de dados e a performance**.
  - **Facilita a evolução do sistema:** Alterações em atributos comuns (como trocar o tipo do id ou adicionar campos de auditoria**) são feitas em um único local, propagando-se automaticamente para todas as entidades que herdam da base entity.**

Base Entity

```java
@MappedSuperclass
@Getter
@Setter
@RequiredArgsConstructor
public abstract class BaseEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @CreationTimestamp
    @Column(updatable = false)
    private LocalDateTime createdAt;
    private LocalDateTime deletedAt = null;
    @UpdateTimestamp
    private LocalDateTime updatedAt = null;

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || Hibernate.getClass(this) != Hibernate.getClass(o)) return false;
        BaseEntity that = (BaseEntity) o;
        return id != null && Objects.equals(id, that.id);
    }

    @Override
    public int hashCode() {
        return getClass().hashCode();
    }
}
```

Base Entity

- @MappedSuperclass:
  - Indica que esta classe não será uma entidade JPA por si só, mas suas propriedades serão herdadas por outras entidades concretas. Os campos definidos aqui serão mapeados para as tabelas das subclasses.
- @Getter, @Setter, @RequiredArgsConstructor:
  - São anotações do Lombok que geram automaticamente os métodos getters e setters para todos os campos, e um construtor para os campos finais ou @NonNull.

Base Entity

- @CreationTimestamp
  - Anotação do Hibernate. O campo createdAt recebe automaticamente a data/hora de criação do registro.
- @Column(updatable = false)
  - Indica que o campo createdAt não pode ser alterado após a inserção.
- private LocalDateTime deletedAt = null;
  - Campo para armazenar a data/hora de exclusão lógica (soft delete). Inicialmente é nulo.
- @UpdateTimestamp
  - Anotação do Hibernate. O campo updatedAt é atualizado automaticamente toda vez que o registro sofrer alteração.

Base Entity

- equals(Object o)
  - Garante que duas entidades são consideradas iguais se forem da mesma classe (usando Hibernate.getClass(this) para evitar problemas com proxies do Hibernate) e tiverem o mesmo id (e o id não for nulo).
- hashCode()
  - Gera o hash baseado na classe, o que é importante para o uso correto em coleções como Set ou como chave de Map.

Entity

```java
@EqualsAndHashCode(callSuper = true)
@Entity
@NoArgsConstructor
@AllArgsConstructor
@Data
@SQLDelete(sql = "UPDATE pessoa SET deleted_at = CURRENT_TIMESTAMP where id=?")
@SQLRestriction("deleted_at is null")
public class Pessoa extends BaseEntity {

    @NotBlank (message = "O nome não pode estar em branco.")
    String nome;
    @Min(value = 0, message = "A idade tem que ser maior que zero.")
    int idade;
    String dataNascimento;
    String sexo;
    boolean isAdmin = false;

    @OneToOne(cascade = CascadeType.ALL, orphanRemoval = true)
    @JoinColumn(name = "endereco_id")
    Endereco endereco;
}
```

Entity

- @EqualsAndHashCode(callSuper = true)
  - Gera automaticamente os métodos equals e hashCode, incluindo os campos da superclasse (AbstractEntity). Isso é importante para garantir que a comparação de objetos considere o id e outros atributos herdados.
- @SQLDelete
  - Customiza o comportamento do delete. Em vez de deletar fisicamente o registro, executa um SQL para atualizar o campo deleted_at com o timestamp atual, implementando o chamado soft delete (exclusão lógica).
- @SQLRestriction
  - Adiciona uma restrição global para todas as consultas JPA nesta entidade, filtrando apenas registros onde deleted_at é nulo (ou seja, não deletados logicamente).

Fail Fast

- Estratégia que relata imediatamente em sua interface qualquer condição que possa indicar uma falha.
- Os sistemas fail-fast geralmente são projetados para interromper a operação normal em vez de tentar continuar um processo possivelmente falho.
- Esses projetos geralmente verificam o estado do sistema em vários pontos de uma operação, para que quaisquer falhas possam ser detectadas antecipadamente.
- A responsabilidade de um módulo fail-fast é detectar erros e permitir que o próximo nível mais alto do sistema os trate.

Tratando exceções

- O Spring 3.2 apresentou suporte para um @ExceptionHandler global com a anotação @ControllerAdvice.
- @ControllerAdvice permite um mecanismo que faz uso de ResponseEntity junto com a segurança de tipo e flexibilidade de @ExceptionHandler

Tratando exceções

```
@ControllerAdvice
public class ControllerAdvisor  extends ResponseEntityExceptionHandler{

    @ExceptionHandler(ConstraintViolationException.class)
    ResponseEntity<Object> handleConstraintViolationException(ConstraintViolationException e) {
        Map<String, Object> body = new LinkedHashMap<>();
        body.put("timestamp", LocalDateTime.now());
        body.put("message", "Erro ao acessar a URI");
        body.put("error", e.getMessage());
        return new ResponseEntity<>(body, HttpStatus.BAD_REQUEST);
    }
```

Tratando exceções

```
@ExceptionHandler(EntityNotFoundException.class)
public ResponseEntity<Object> handleEntityNotFoundException(
        EntityNotFoundException ex, WebRequest request) {
    Map<String, Object> body = new LinkedHashMap<>();
    body.put("timestamp", LocalDateTime.now());
    body.put("message", "Erro ao acessar a URI");
    body.put("error", ex.getMessage());
    return new ResponseEntity<>(body, HttpStatus.NOT_FOUND);
}
```

Tratando exceções

- O Spring 5 introduziu a classe ResponseStatusException.
- É possível criar uma instância dela fornecendo um HttpStatus e opcionalmente um motivo e uma causa:

```
throw new ResponseStatusException(HttpStatus.NOT_FOUND, "Foo Not Found", exc);
```

Spring HATEOAS

- O Spring HATEOAS é uma biblioteca de APIs para criar facilmente representações REST que seguem o princípio de HATEOAS

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-hateoas</artifactId>
</dependency>
```

Spring HATEOAS

```java
import org.springframework.hateoas.RepresentationModel;

public class Cliente extends RepresentationModel<Cliente> {

    String nome;
}
```

```java
public class RepresentationModel<T extends RepresentationModel<? extends T>> {
    private final List<Link> links;

    public RepresentationModel() {
        this.links = new ArrayList();
    }

    public RepresentationModel(Link initialLink) {
        Assert.notNull(initialLink, "initialLink must not be null!");
        this.links = new ArrayList();
        this.links.add(initialLink);
    }
```

```java
public T add(Link link) {
    Assert.notNull(link, "Link must not be null!");
    this.links.add(link);
    return this;
}

public T add(Iterable<Link> links) {
    Assert.notNull(links, "Given links must not be null!");
    links.forEach(this::add);
    return this;
}

public T add(Link... links) {
    Assert.notNull(links, "Given links must not be null!");
    this.add((Iterable)Arrays.asList(links));
    return this;
}
```

Spring HATEOAS

```
import static org.springframework.hateoas.server.mvc.WebMvcLinkBuilder.linkTo;
```

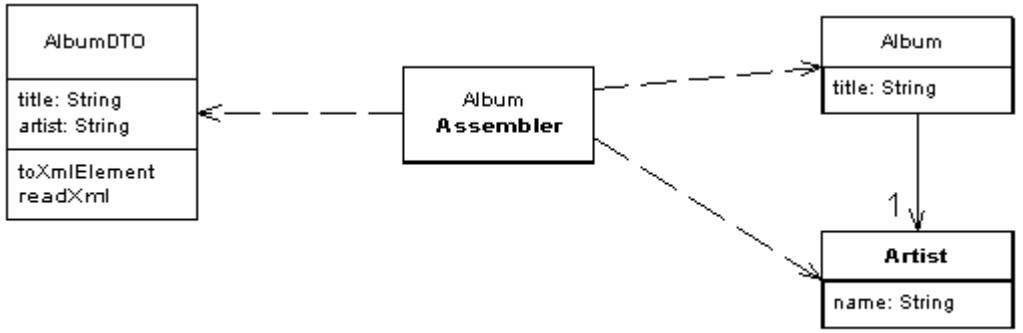
```
//mim mesmo
cliente.add(linkTo(ClienteController.class).slash(id).withSelfRel());
//para todos
cliente.add(linkTo(ClienteController.class).withRel("allClientes"));

Endereco endereco = cliente.getEndereco();

endereco.add(linkTo(EnderecoController.class).slash(endereco.getId()).withSelfRel());
endereco.add(linkTo(EnderecoController.class).withRel("allEnderecos"));
```

Data Transfer Object

- Padrão de projetos usado para o envio de dados entre diferentes componentes de um sistema via serialização.
- É possível enviar/receber dados de maneira organizada de maneira diferente da entidade que será persistida.



https://martinfowler.com/eaaCatalog/dataTransferObject.html

http://modelmapper.org/

Data Transfer Object

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class PessoaRequestDto {
    String nome;
    String dataNascimento;
    String sexo;
ı
@Data
@NoArgsConstructor
@AllArgsConstructor
public class PessoaResponseDto extends RepresentationModel<PessoaResponseDto> {
    String nome;
    int idade;
    String sexo;
    EnderecoResponseDto endereco;

    public void addLinks(Pessoa p){
        this.add(linkTo(PessoaController.class).slash(p.getId()).withSelfRel());
        this.add(linkTo(EnderecoController.class).slash(p.getEndereco().getId()).withRel("endereco"));
    }
}
```

Data Transfer Object

```
@PostMapping
public ResponseEntity<PessoaResponseDto> create(@RequestBody PessoaRequestDto pessoa) {

    Pessoa created = service.create(convertToEntity(pessoa));

    URI location = ServletUriComponentsBuilder
            .fromCurrentRequest()
            .path("{id}")
            .buildAndExpand(created.getId())
            .toUri();

    return ResponseEntity.created(location).body(convertToDto(created));
}
```

Data Transfer Object

```
@PostMapping
public ResponseEntity<PessoaResponseDto> create(@RequestBody PessoaRequestDto pessoa) {

    Pessoa created = service.create(convertToEntity(pessoa));

    URI location = ServletUriComponentsBuilder
            .fromCurrentRequest()
            .path("{id}")
            .buildAndExpand(created.getId())
            .toUri();

    return ResponseEntity.created(location).body(convertToDto(created));
}
```

Model Mapper

- O MapStruct é um framework de mapeamento de objetos Java que permite transformar automaticamente um objeto em outro, como por exemplo:
  - De uma entidade (Entity) para um DTO
  - De um DTO para uma entidade
  - De um formulário para uma entidade
  - De um objeto de camada de negócio para um objeto da camada de apresentação, e vice-versa

Map Struct

- MapStruct é um framework de "bean mapping", ou seja, um gerador de código que cria implementações de métodos que convertem um tipo de objeto em outro, com base em interfaces e anotações Java.

```java
public class User {
    private String name;
    private String email;
}

public class UserDTO {
    private String name;
    private String email;
}
```

```java
@Mapper
public interface UserMapper {
    UserDTO toDto(UserEntity entity);
    UserEntity toEntity(UserDTO dto);
}
```

Map Struct

- O MapStruct gera automaticamente uma **classe implementando esse código**, como:

```
public class UserMapperImpl implements UserMapper {
    public UserDTO toDto(User entity) {
        if (entity == null) return null;
        UserDTO dto = new UserDTO();
        dto.setName(entity.getName());
        dto.setEmail(entity.getEmail());
        return dto;
    }
    ...
}
```

Map Struct

```xml
<dependencies>
    <!-- MapStruct -->
    <dependency>
        <groupId>org.mapstruct</groupId>
        <artifactId>mapstruct</artifactId>
        <version>1.5.5.Final</version>
    </dependency>

    <!-- Apenas para projetos Spring Boot com suporte à injeção -->
    <dependency>
        <groupId>org.mapstruct</groupId>
        <artifactId>mapstruct-processor</artifactId>
        <version>1.5.5.Final</version>
        <scope>provided</scope>
    </dependency>
</dependencies>
```

```xml
<build>
    <plugins>
        <!-- Plugin para gerar os mapeamentos automaticamente -->
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.11.0</version>
            <configuration>
                <source>17</source> <!-- ou 21,ou 24 se for seu caso --
                <target>17</target>
                <annotationProcessorPaths>
                    <path>
                        <groupId>org.mapstruct</groupId>
                        <artifactId>mapstruct-processor</artifactId>
                        <version>1.5.5.Final</version>
                    </path>
                </annotationProcessorPaths>
            </configuration>
        </plugin>
    </plugins>
</build>
```

Map Struct

```java
@Mapper(componentModel = "spring") // permite injetar como @Autowired
public interface PessoaMapper {

    PessoaMapper INSTANCE = Mappers.getMapper(PessoaMapper.class);

    PessoaRequestDto toRequestDto(Pessoa entity);
    Pessoa toEntity(PessoaRequestDto dto);

    PessoaResponseDto toResponseDto(Pessoa entity);
    Pessoa toEntity(PessoaResponseDto dto);
}
```

Map Struct

```
public class PessoaController {

    final PessoaService service;
    final PessoaMapper mapper;

    public PessoaController(PessoaService service, PessoaMapper mapper) {
        this.service = service;
        this.mapper = mapper;
    }


@PostMapping
public ResponseEntity<?> createPessoa(@RequestBody PessoaRequestDto p) throws URISyntaxException {

    Pessoa pEntity = mapper.toEntity(p);

    Pessoa pessoa = service.adicionar(pEntity);
    return ResponseEntity.created(new URI("/pessoas/"+pessoa.getId())).build();
}
```

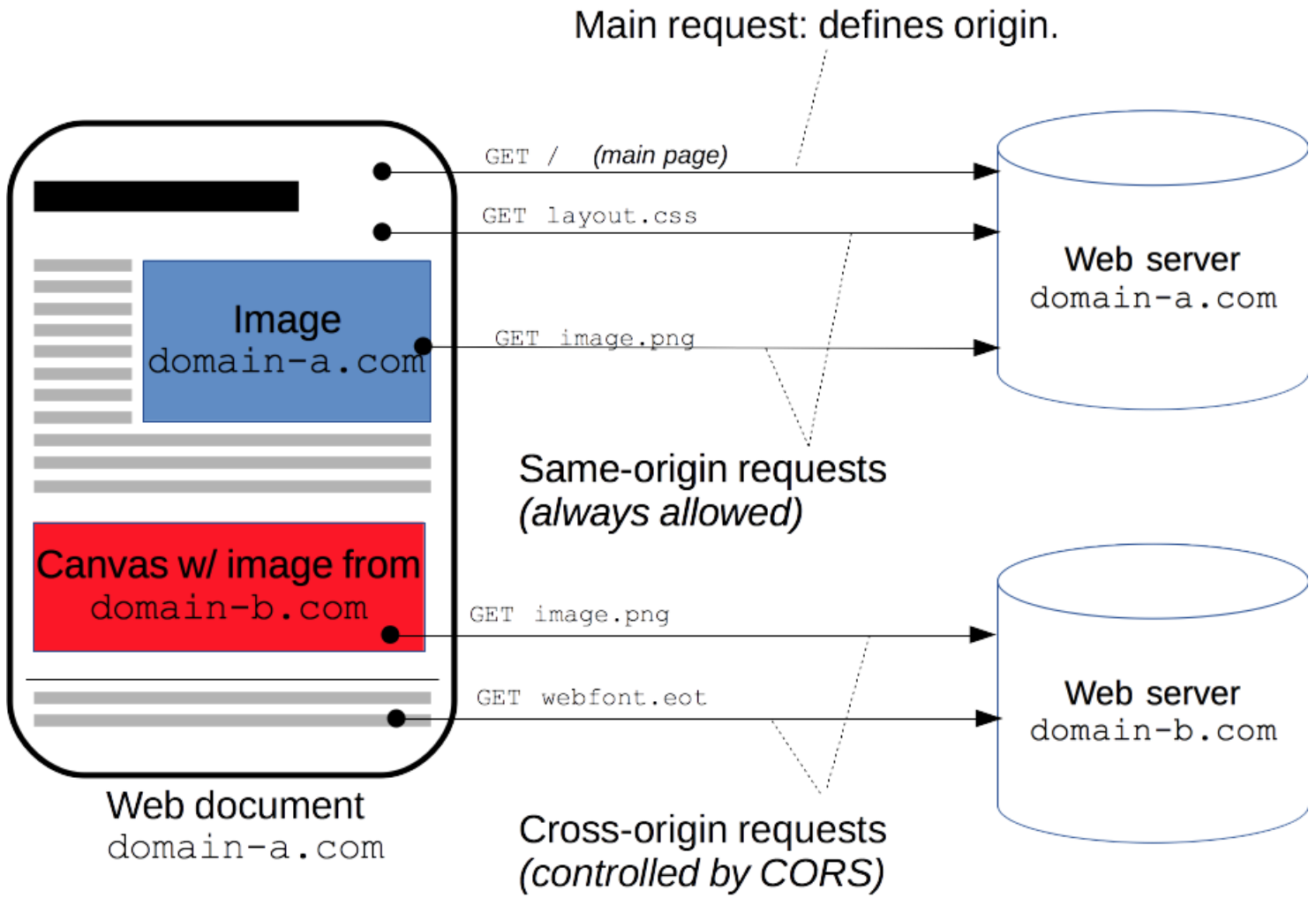
CORS Policy

- CORS - Cross-Origin Resource Sharing ou ompartilhamento de recursos com origens diferentes
- Mecanismo que usa cabeçalhos adicionais HTTP para informar a um navegador que permita que uma aplicação web seja executado em uma **origem** (domínio) com permissão para acessar recursos selecionados de um servidor em uma origem **diferente**.
- Por motivos de segurança, navegadores restringem requisições cross-origin HTTP **iniciadas por scripts**.

CORS Policy

- Uma aplicação web executa uma requisição cross-origin HTTP ao solicitar um recurso que tenha uma origem **diferente** (domínio, protocolo e porta) da sua **própria origem**.
  - Um frontend disponível em http://domain-a.com usa XMLHttpRequest para fazer uma requisição para http://api.domain-b.com/data.json.

CORS Policy



https://developer.mozilla.org/pt-BR/docs/Web/HTTP/CORS

CORS Policy

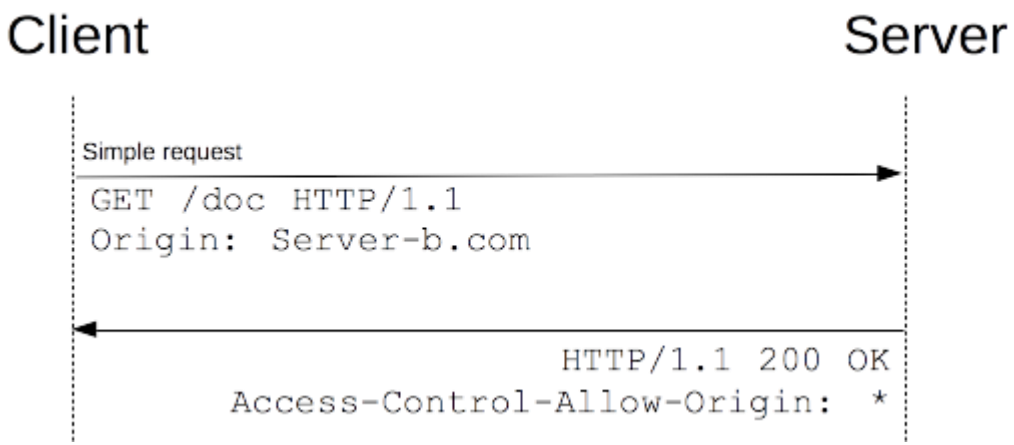
- O padrão Cross-Origin Resource Sharing trabalha adicionando novos cabeçalhos HTTP que permitem que os servidores descrevam um conjunto de origens que possuem permissão de ler uma informação usando o navegador.
- A especificação exige que navegadores "pré-enviem“ (Preflighted requests) a requisição, solicitando os métodos suportados pelo servidor com um método de requisição HTTP OPTIONS.
  - Após a "aprovação", o servidor envia a requisição verdadeira com o método de requisição HTTP correto.

Requisições Simples

`http://foo.example`

```
var invocation = new XMLHttpRequest();
var url = 'http://bar.other/resources/public-
data/';

function callOtherDomain() {
  if(invocation) {
    invocation.open('GET', url, true);
    invocation.onreadystatechange = handler;
    invocation.send();
  }
}
```



Requisições Simples

```
GET /resources/public-data/ HTTP/1.1
Host: bar.other
User-Agent: Mozilla/5.0 (Macintosh; U; Intel
Mac OS X 10.5; en-US; rv:1.9.1b3pre)
Gecko/20081130 Minefield/3.1b3pre
Accept:
text/html,application/xhtml+xml,application/xml
;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Connection: keep-alive
Referer: http://foo.example/examples/access-
control/simpleXSInvocation.html
Origin: http://foo.example
```

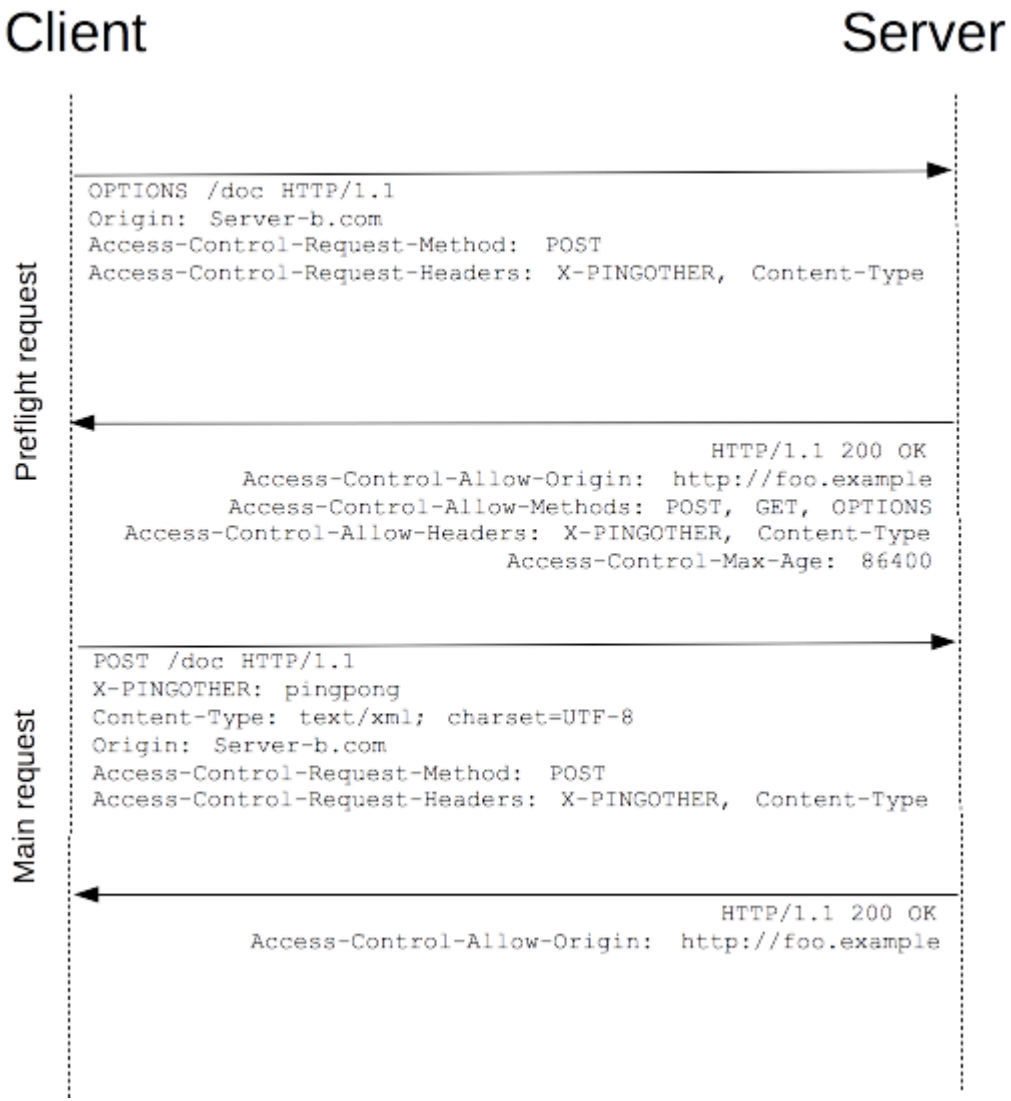
```
HTTP/1.1 200 OK
Date: Mon, 01 Dec 2008 00:23:53 GMT
Server: Apache/2.0.61
Access-Control-Allow-Origin: *
Keep-Alive: timeout=2, max=100
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: application/xml

[XML Data]
```

Requisição com pré-envio

```
var invocation = new XMLHttpRequest();
var url = 'http://bar.other/resources/post-here/';
var body = '<?xml version="1.0"?><person><name>Arun</name></person>';

function callOtherDomain(){
  if(invocation)
    {
      invocation.open('POST', url, true);
      invocation.setRequestHeader('X-PINGOTHER', 'pingpong');
      invocation.setRequestHeader('Content-Type', 'application/xml');
      invocation.onreadystatechange = handler;
      invocation.send(body);
    }
}
```



Requisição com pré-envio

```
OPTIONS /resources/post-here/ HTTP/1.1
Host: bar.other
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.5; en-US;
rv:1.9.1b3pre) Gecko/20081130 Minefield/3.1b3pre
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Connection: keep-alive
Origin: http://foo.example
Access-Control-Request-Method: POST
Access-Control-Request-Headers: X-PINGOTHER, Content-Type


HTTP/1.1 200 OK
Date: Mon, 01 Dec 2008 01:15:39 GMT
Server: Apache/2.0.61 (Unix)
Access-Control-Allow-Origin: http://foo.example
Access-Control-Allow-Methods: POST, GET, OPTIONS
Access-Control-Allow-Headers: X-PINGOTHER, Content-Type
Access-Control-Max-Age: 86400
Vary: Accept-Encoding, Origin
Content-Encoding: gzip
Content-Length: 0
Keep-Alive: timeout=2, max=100
Connection: Keep-Alive
Content-Type: text/plain
```

```
POST /resources/post-here/ HTTP/1.1
Host: bar.other
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.5; en-US;
rv:1.9.1b3pre) Gecko/20081130 Minefield/3.1b3pre
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Connection: keep-alive
X-PINGOTHER: pingpong
Content-Type: text/xml; charset=UTF-8
Referer: http://foo.example/examples/preflightInvocation.html
Content-Length: 55
Origin: http://foo.example
Pragma: no-cache
Cache-Control: no-cache

HTTP/1.1 200 OK
Date: Mon, 01 Dec 2008 01:15:40 GMT
Server: Apache/2.0.61 (Unix)
Access-Control-Allow-Origin: http://foo.example
Vary: Accept-Encoding, Origin
Content-Encoding: gzip
Content-Length: 235
Keep-Alive: timeout=2, max=99
Connection: Keep-Alive
Content-Type: text/plain
```

Configuração no Código

```
@RestController
@RequestMapping("/endereco")
@CrossOrigin(origins = "http://localhost:3000", exposedHeaders = "X-Total-Count")
public class EnderecoController {
}
```

Implementação de Autenticação e Autorização no Spring

- JWT
- Oauth 2.0

Termos que serão utilizados

- **Authentication** processo de verificação da identidade de um usuário com base nas credenciais fornecidas. “Quem é você?”
- **Authorization** processo de determinar se um usuário tem permissão adequada para realizar uma determinada ação ou ler dados específicos, supondo que o usuário já esteja autenticado. “O usuário pode fazer/ler isso?”
- **Principle** usuário autenticado no momento.
- **Granted authority** permissão do usuário autenticado.
- **Role** grupo de permissões do usuário autenticado.

Filtros

```
@WebFilter("/*")
public class ExemploFilter implements Filter {
    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
        Filter.super.init(filterConfig);
    }

    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse,
FilterChain filterChain) throws IOException, ServletException {
        System.out.println("Filtered");
    }

    @Override
    public void destroy() {
        Filter.super.destroy();
    }
}
```

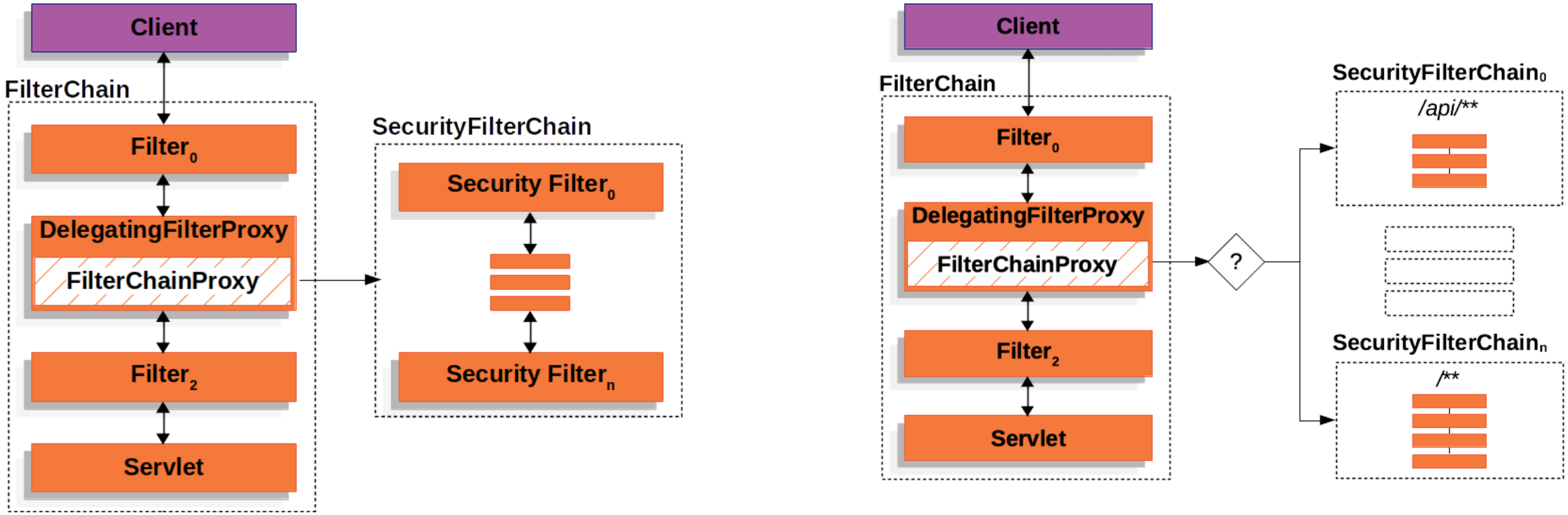
```
@SpringBootApplication
@ServletComponentScan
public class AulaSpringRestApplication implements CommandLineRunner {}
```

Spring Security

- Adicionar outras dependências Spring em geral não implica em um efeito imediato na aplicação até que a configuração adequada seja fornecida.
- No Spring Security isso é diferente e acaba por confundir novos desenvolvedores.
- Após a adição da dependência do Spring Security qualquer rota que desejarmos acessar será enviada para uma página de login.
- Este é o comportamento padrão porque o Spring Security requer autenticação para uso para todas as rotas.

```
spring.security.user.password=123
```

Estrutura Básica de uma Aplicação com Spring Security

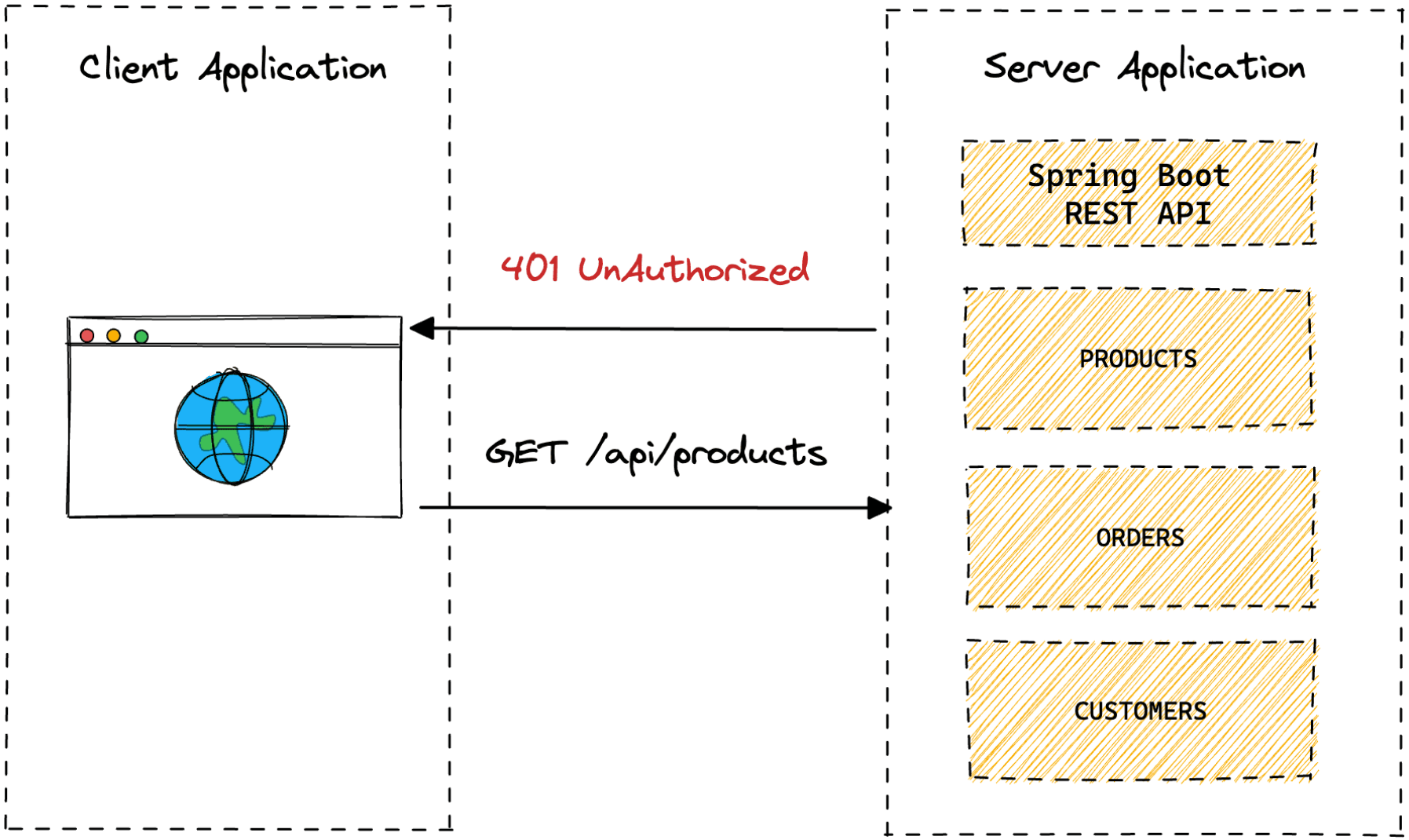


FONTE: https://docs.spring.io/spring-security/reference/servlet/architecture.html#servlet-securityfilterchain

Spring Security Filters Chain

- Quando o Spring Security é adicionado às dependência da aplicação, uma cadeia de filtros passa a interceptar todas as solicitações recebidas.
- Essa cadeia consiste em vários filtros e cada um deles lida com um caso de uso específico.
  - Verifique se a URL solicitada é acessível publicamente
  - Verifique se o usuário está autorizado a realizar a ação solicitada
  - Etc

Arquitetura de uma Aplicação



Arquitetura de uma Aplicação

- A aplicação cliente fará chamadas para um aplicação de backend Spring que expõe dados por meio da API REST.
- Existem atualmente controladores REST que expõem os recursos como clientes, produtos, etc.
- Vamos proteger todos os recursos para que, quando o cliente fizer uma chamada para a API REST, o cliente receba um 401 (não autorizado), **o que significa que a solicitação do cliente não foi concluída porque não possui credenciais de autenticação válidas para o recurso solicitado**.

JWT

- JSON Web Token (JWT) é um padrão aberto (RFC 7519) que define uma maneira compacta e independente de transmitir informações com segurança entre as partes como um objeto JSON.
- Essas informações podem ser verificadas e confiáveis porque são assinadas digitalmente.
- Os JWTs podem ser assinados usando um segredo (com o algoritmo HMAC - *Hash-based message authentication code*) ou um par de chaves pública/privada usando RSA ou ECDSA.
- Quando os tokens são assinados usando pares de chaves pública/privada, a assinatura também certifica que apenas a parte que detém a chave privada é a que a assinou.

JWT - Estrutura

- Os JSON Web Tokens consistem em três partes separadas por pontos (.), que são:
  - Cabeçalho
  - Payload
  - Assinatura
- Portanto, um JWT geralmente se parece com o seguinte.

xxxxx.aaaa.zzzzz

JWT - Estrutura

- O cabeçalho normalmente consiste em duas partes: o tipo do token, que é JWT, e o algoritmo de assinatura usado, como HMAC, SHA256 ou RSA.

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

JWT - Estrutura

- A segunda parte do token é o payload que contém as *claims*.
- As *claims* são declarações sobre uma entidade (normalmente, o usuário) e dados adicionais.
- Existem três tipos de *claims*:
  - *Claims* registradas, públicas e privadas.

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

JWT - Estrutura

- ***Claims* registradas:** são um conjunto de *claims* predefinidas que não são obrigatórias, mas recomendadas, para fornecer um conjunto de *claims* úteis e interoperáveis.
  - Alguns deles são: iss (emissor), exp (tempo de expiração), sub (assunto), aud (público) e outros.
- ***Claims* públicas:** podem ser definidas à vontade por aqueles que usam JWTs. Mas, para evitar colisões, elas devem ser definidos no IANA JSON Web Token Registry ou ser definidos como um URI que contém um namespace resistente a colisões.
- ***Claims* privadas:** são as *claims* personalizadas criadas para compartilhar informações entre as partes que concordam em usá-las e não são reivindicações registradas ou públicas.

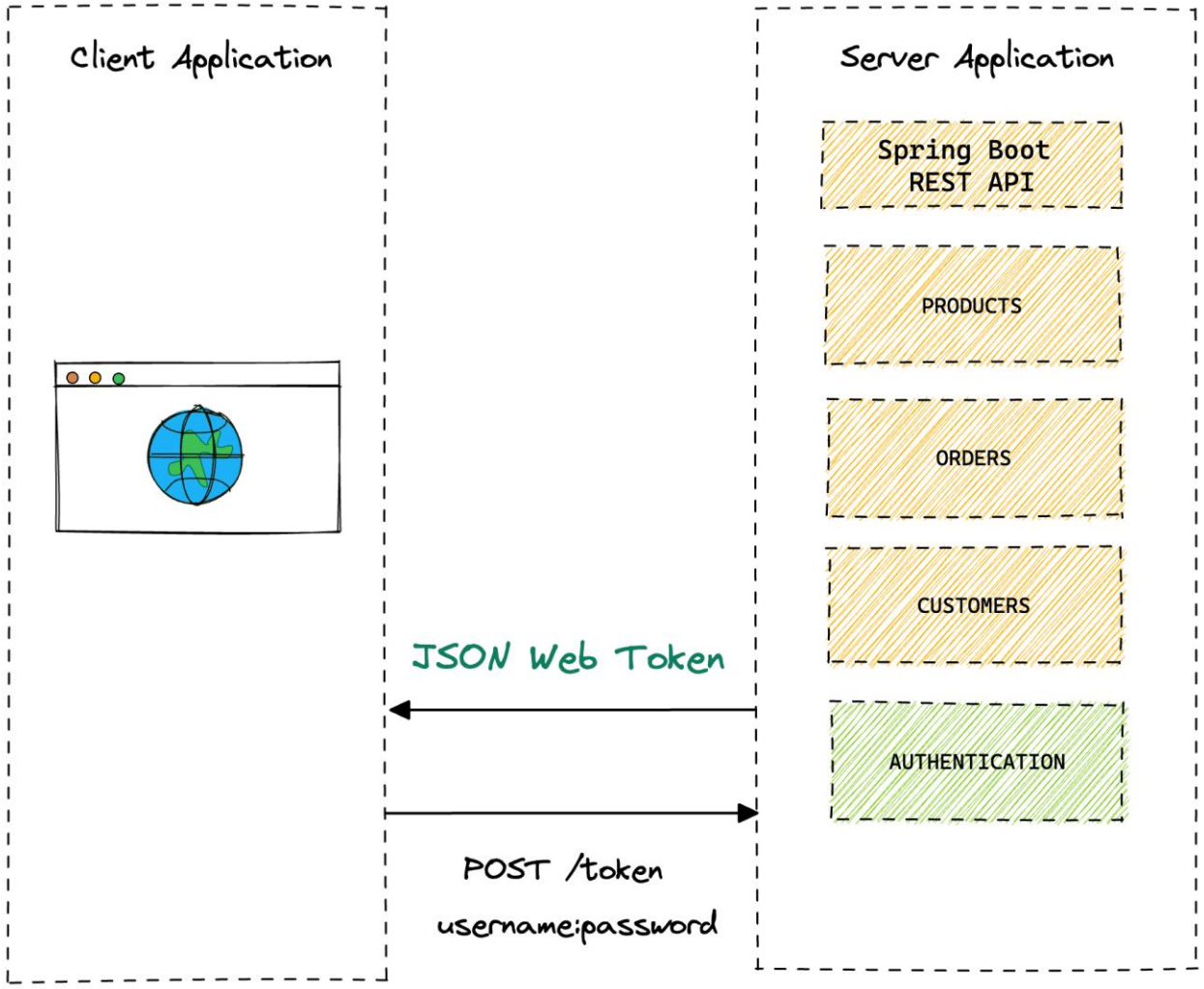
JWT - Estrutura

- Para criar a parte da assinatura é usado o cabeçalho, o payload, um segredo e o algoritmo especificado no cabeçalho e assiná-lo.
- Por exemplo, se você deseja usar o algoritmo HMAC SHA256, a assinatura será criada da seguinte maneira:

```
HMACSHA256(base64UrlEncode(cabeçalho) + "." +
    base64UrlEncode(payload), segredo)
```

- A assinatura serve para verificar se a mensagem não foi alterada no meio do caminho e, no caso de tokens assinados com chave privada, também pode verificar se o remetente do JWT é quem diz ser.

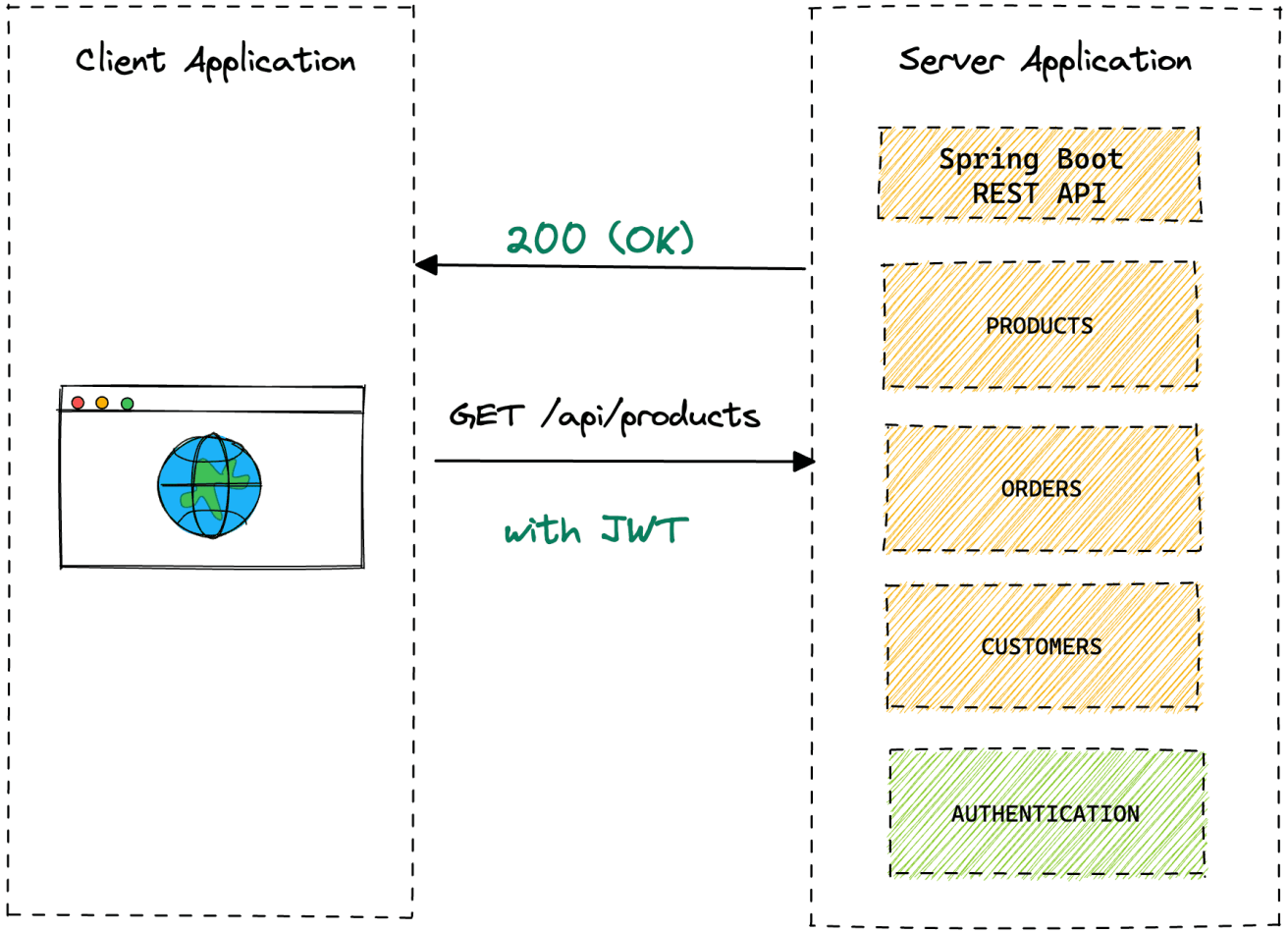
Arquitetura de uma Aplicação



Arquitetura de uma Aplicação

- Precisaremos adicionar um novo controlador de autenticação para o qual um cliente pode fazer uma solicitação com suas credenciais de autenticação (nome de usuário + senha) e, quando forem autenticados com sucesso, o serviço retornará um JWT.

Arquitetura de uma Aplicação



Arquitetura de uma Aplicação

- O cliente armazenará o JWT e cada solicitação subsequente o passará por meio do cabeçalho de autorização.
- Quando a aplicação do servidor receber a solicitação com o JWT, ele verificará se é um token válido e, se for, permitirá que a solicitação continue.

Dependências

- Vamos adicionar as dependências ao projeto.

```
<!-- Spring Security-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-configuration-processor</artifactId>
    <optional>true</optional>
</dependency>
```

Dependências

- A dependência `oauth2-resource-server` adiciona o Spring Security de maneira indireta.

https://docs.spring.io/spring-security/reference/servlet/oauth2/resource-server/index.html

Configuração do Spring Security

```
@EnableWebSecurity
@Configuration
public class SecurityConfig{

    @Bean
    public InMemoryUserDetailsManager users() {
        return new InMemoryUserDetailsManager(
                User.withUsername("taniro")
                        .password("{noop}password")
                        .authorities("read")
                        .build()
        );
    }

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        return http
                .csrf(csrf -> csrf.disable())
                .authorizeHttpRequests( auth -> auth
                        .anyRequest().authenticated()
                )
                .sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
                .httpBasic(Customizer.withDefaults())
                .build();
    }
}
```

Resource Server

- O Spring Security oferece suporte à proteção de endpoints usando duas formas Bearer Tokens do OAuth 2.0:
  - JWT
  - Opaque Tokens
- Existem circunstâncias em que a aplicação pode delegar seu gerenciamento de autoridade a um servidor de autorização
  - Okta
  - Spring Authorization Server
  - Keycloak
- O servidor de autorização pode ser consultado por servidores de recursos para autorizar requisições.

Resource Server

- Nesta aula usaremos JWTs autoassinados que eliminarão a necessidade de introduzir um servidor de autorização.

OAuth2 Resource Server JWT

- Para utilizar um ResourceServer é necessário realizar a sua configuração definindo .oauth2ResourceServer().
- Pode ser configurado a partir de um servidor de recursos personalizado ou você pode usar a classe OAuth2ResourceServerConfigurer fornecida pelo Spring.
- Por padrão isso conecta um BearerTokenAuthenticationFilter que pode ser usado para analisar a solicitação de Bearer tokens e fazer uma tentativa de autenticação.
- Esta classe de configuração tem as seguintes opções disponíveis:
  - accessDeniedHandler - Personaliza como os erros de acesso negado são tratados.
  - authenticationEntryPoint - Personaliza como as falhas de autenticação são tratadas.
  - bearerTokenResolver - Personaliza como resolver um bearer token da solicitação.
  - jwt(Customizer) - Ativa o suporte ao bearer token codificado em Jwt.
  - opaqueToken(Customizer) - Ativa o suporte de opaque bearer token.

OAuth2 Resource Server JWT

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    return http
            .csrf(csrf -> csrf.disable())
            .authorizeRequests( auth -> auth
                    .anyRequest().authenticated()
            )
            .oauth2ResourceServer( oauth2 -> oauth2.jwt(Customizer.withDefaults()))
            .sessionManagement(session ->
session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .httpBasic(Customizer.withDefaults())
            .build();
}
```

OAuth2 Resource Server JWT

- Ao usar o personalizador JWT, você precisa fornecer um dos seguintes:
  - Forneça um Jwk Set Uri via OAuth2ResourceServerConfigurer.JwtConfigurer.jwkSetUri
  - Forneça uma instância JwtDecoder via OAuth2ResourceServerConfigurer.JwtConfigurer.decoder e Exponha um bean JwtDecoder.

```
Description:

Parameter 0 of method setFilterChains in
org.springframework.security.config.annotation.web.configuration.WebSecurityConfiguration
required a bean of type 'org.springframework.security.oauth2.jwt.JwtDecoder' that could not be found.

Action:

Consider defining a bean of type 'org.springframework.security.oauth2.jwt.JwtDecoder'
in your configuration.
```

Assinatura de JSON Web Token

- A próxima etapa é criar um novo bean JwtDecoder.
- Como já vimos, existem 3 partes no JWT:
  - Cabeçalho
  - Payload
  - Assinatura.
- A assinatura é criada criptografando o cabeçalho + payload e um segredo (ou chave privada).
- Um JWT pode ser criptografado usando uma chave simétrica (segredo compartilhado) ou chaves assimétricas (a chave privada de um par público-privado).

RSA public & private KEYS

- Serão criadas chaves RSA e colocadas no diretório /src/main/rescurces/certs.
- A aplicação OpenSSL, instalada por padrão no Linux, será utilizada para gerar as chaves.

```
# create rsa key pair
openssl genrsa -out keypair.pem 2048

# extract public key
openssl rsa -in keypair.pem -pubout -out public.pem

# create private key in PKCS#8 format
openssl pkcs8 -topk8 -inform PEM -outform PEM -nocrypt -in keypair.pem -out private.pem
```

JWT Decoder

- Com as chaves pública e privada criadas será definido um bean JwtDecoder.
- Primeiro, no pacote de configuração criar o record chamado RsaKeyProperties que será usado para externalizar as chaves pública e privada.

```
@ConfigurationProperties(prefix = "rsa")
public record RsaKeyProperties(RSAPublicKey publicKey, RSAPrivateKey privateKey) {

}
```

Application.yml

- Adicione o local das chaves no arquivo application.yml
- Externalize as propriedades na classe da aplicação.

```
rsa:
  private-key: classpath:certs/private.pem
  public-key: classpath:certs/public.pem
```

Application

```java
@SpringBootApplication
@EnableConfigurationProperties(RsaKeyProperties.class)
public class Application {

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

}
```

Configuração do Spring Security

- Injetar na configuração do Spring Security as chaves RSA

```
@EnableWebSecurity
@Configuration
public class SecurityConfig{

    private final RsaKeyProperties rsaKeys;

    public SecurityConfig(RsaKeyProperties rsaKeys) {
        this.rsaKeys = rsaKeys;
    }
```

Configuração do Spring Security

- Implementar as Beans JWT Decoder e JWTEncoder.

```
@Bean
JwtDecoder jwtDecoder() {
    return NimbusJwtDecoder.withPublicKey(rsaKeys.publicKey()).build();
}

@Bean
JwtEncoder jwtEncoder() {
    JWK jwk = new RSAKey.Builder(rsaKeys.publicKey()).privateKey(rsaKeys.privateKey()).build();
    JWKSource<SecurityContext> jwks = new ImmutableJWKSet<>(new JWKSet(jwk));
    return new NimbusJwtEncoder(jwks);
}
```

Token Service

- Serviço utilizado para assinar Tokens

Token Service

```
@Service
public class TokenService {

    private final JwtEncoder encoder;

    public TokenService(JwtEncoder encoder) {
        this.encoder = encoder;
    }

    public String generateToken(Authentication authentication) {
        Instant now = Instant.now();
        String scope = authentication.getAuthorities().stream()
                .map(GrantedAuthority::getAuthority)
                .collect(Collectors.joining(" "));
        JwtClaimsSet claims = JwtClaimsSet.builder()
                .issuer("self")
                .issuedAt(now)
                .expiresAt(now.plus(1, ChronoUnit.HOURS))
                .subject(authentication.getName())
                .claim("scope", scope)
                .build();
        return this.encoder.encode(JwtEncoderParameters.from(claims)).getTokenValue();
    }

}
```

Controller para autenticação

```
@RestController
public class AuthController {

    private static final Logger LOG = LoggerFactory.getLogger(AuthController.class);

    private final TokenService tokenService;

    public AuthController(TokenService tokenService) {
        this.tokenService = tokenService;
    }

    @PostMapping("/token")
    public String token(Authentication authentication) {
        LOG.debug("Token requested for user: '{}'", authentication.getName());
        String token = tokenService.generateToken(authentication);
        LOG.debug("Token granted: {}", token);
        return token;
    }

}
```

Atualização para remoção do uso de HTTP Basic

```
@Bean
public AuthenticationManager authenticationManager(UserDetailsService
userDetailsService){
    var authProvider = new DaoAuthenticationProvider();
    authProvider.setUserDetailsService(userDetailsService);
    return new ProviderManager(authProvider);
}

@Bean
public UserDetailsService users() {
    return new InMemoryUserDetailsManager(
            User.withUsername("taniro")
                    .password("{noop}password")
                    .authorities("read")
                    .build()
    );
}
```

Atualização para remoção do uso de HTTP Basic

```
@RestController
public class AuthController {

    private final TokenService tokenService;
    private final AuthenticationManager authenticationManager;

    public AuthController(TokenService tokenService, AuthenticationManager authenticationManager) {
        this.tokenService = tokenService;
        this.authenticationManager = authenticationManager;
    }

    @PostMapping("/token")
    public String token(@RequestBody LoginDTO dto) {

        Authentication authentication = authenticationManager.authenticate(new
UsernamePasswordAuthenticationToken(dto.username(), dto.password()));
        return tokenService.generateToken(authentication);
    }
}
```

Autenticação utilizando usuário do banco de dados

```java
@Entity
public class Credenciais implements UserDetails {

    @Id
    String id;
    String username;
    String password;
    String role;

    @OneToOne
    @MapsId
    @JoinColumn(name = "pessoa_id")
    Pessoa pessoa;

    LocalDateTime deletedAt;
    @UpdateTimestamp
    LocalDateTime updatedAt;
    @CreationTimestamp
    LocalDateTime createdAt;

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return Arrays.stream(role.split(",")).map(SimpleGrantedAuthority::new).toList();
    }
}
```

Autenticação utilizando usuário do banco de dados

```
public interface CredenciaisRepository extends JpaRepository<Credenciais, String> {

    Optional<Credenciais> findByUsername(String username);
}


@Service
public class CredenciaisService implements UserDetailsService {

    private final CredenciaisRepository repository;

    public CredenciaisService(CredenciaisRepository repository) {
        this.repository = repository;
    }

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {


        return this.repository
                .findByUsername(username)
                .orElseThrow( () -> new UsernameNotFoundException("Not found user " + username));
    }
}
```

Autenticação utilizando usuário do banco de dados

```
@Bean
public AuthenticationManager authenticationManager(UserDetailsService userDetailsService, BCryptPasswordEncoder
encoder){
    var authProvider = new DaoAuthenticationProvider();
    authProvider.setUserDetailsService(userDetailsService);
    authProvider.setPasswordEncoder(encoder);
    return new ProviderManager(authProvider);
}


@Bean
public BCryptPasswordEncoder encoder(){
    return new BCryptPasswordEncoder();
}
```

Projeto Final

- Desenvolver uma aplicação web utilizando o estilo arquitetural REST
- A aplicação deve conter entidades com relacionamentos 1-1, 1-N e N-N.
- Usar o mesmo tema da prova 2.
- Desenvolver todos os endpoints do CRUD para cada recurso
  - Implementar um soft delete ou implementar um logger em nível de aplicação para qualquer procedimento relacionado ao banco de dados.
- Utilizar verbos HTTP e status code da resposta de maneira adequada
- Implementar a API em grau de maturidade REST nível 3.
- Criar Data Transfer Objects para request/ response para criação/update/listagem de objetos
- Criar endpoint para login
- Adicionar segurança em todos os demais end-points da aplicação utilizando uma estratégia STATELESS (JWT ou OAuth2.0)
- Busca paginada

Referências

- COULOURIS, George. **Sistemas distribuídos: conceitos e projeto.** 5. ed. Porto Alegre: Bookman, 2013. 1047 p. ISBN: 9788582600535.
- https://bootify.io/spring-rest/rest-api-spring-security-with-jwt.html
- https://programadev.com.br/spring-security-jwt/
- https://www.keycloak.org/
- https://www.danvega.dev/blog/2022/09/06/spring-security-jwt/

Bibliografia

https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design

DEITEL, H. M.; DEITEL, P. J. **Java: Como Programar**. Bookman, 2005.

ORACLE. **The Java EE 6 Tutorial.** Disponível em: <http://docs.oracle.com/javaee/6/tutorial/doc/bnafe.html>. Acesso em 25/08/2015.

FARIA, Thiago. Java EE 7 com JSF , PrimeF aces e CDI. 2º Edição.

Bibliografia

http://www.arquivodecodigos.net/dicas/jsf-java-server-faces-aprendendo-a-usar-a-classe-facescontext-suas-aplicacoes-jsf-3582.html

http://www.journaldev.com/6881/jsf-beans-example-tutorial-configuring-and-injecting-managed-beans

Bibliografia

- http://www.devmedia.com.br/configurando-pool-de-conexoes-c3p0-com-spring-e-hibernate/28359
- http://www.devmedia.com.br/connection-pool/5869
- http://jamacedo.com/2010/06/crud-jsf-2-0-hibernate-exemplo-gerenciando-livros-2/

Bibliografia

- https://vladmihalcea.com/2017/03/29/the-best-way-to-map-a-onetomany-association-with-jpa-and-hibernate/
- https://en.wikibooks.org/wiki/Java_Persistence
- http://www.devmedia.com.br/lazy-e-eager-loading-com-hibernate/29554