Desenvolvimento de API

Programação para dispositivos móveis

Prof. Allan Rodrigo Leite

Definição de API

- Application Programming Interface (API)
  - Técnica que permite a comunicação entre componentes de software
    - Os componentes de software podem distintos e remotos
    - Necessário um conjunto de definições e protocolos

- Funcionamento básico
  - Aplicação refere-se a qualquer software com uma função distinta
  - Interface determina um contrato de serviço entre duas aplicações
    - Estabelece um padrão de comunicação para solicitações e respostas
  - Arquitetura geralmente é considerada em termos de cliente e servidor

Definição de API

- Exemplos de API
  - API SOAP (Simple Object Access Protocol)
    - Usam um protocolo de acesso a objetos simples
    - Cliente e servidor trocam mensagens usando XML
    - API menos flexível mais popular no passado
  - API RPC (Remote Procedure Call)
    - Cliente invoca uma função ou um procedimento no servidor
    - Servidor envia o resultado da função ao cliente
    - Versão mais popular do RPC é o gRPC (Google Remote Procedure Call)

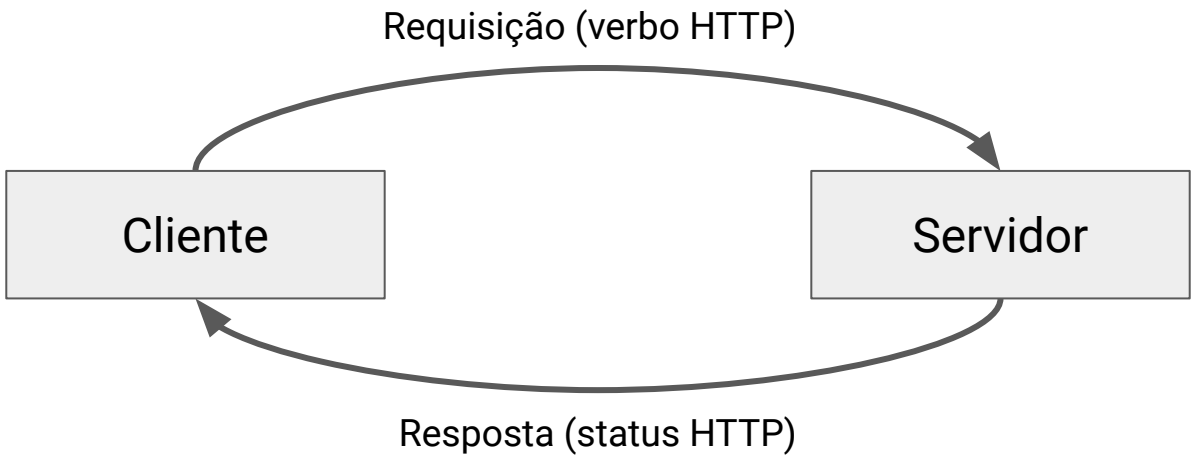
Definição de API

- Exemplos de API (cont.)
  - API WebSocket
    - Usa objetos JSON (JavaScript Object Notation) para transmitir dados
    - Suporta comunicação bidirecional entre aplicações cliente e servidor
    - Recurso moderno para aplicações web que requer dados em tempo real

  - API REST (Representational State Transfer)
    - API mais popular e flexível atualmente
    - Cliente envia solicitações ao servidor em forma de dados
    - Servidor lê a entrada, iniciar funções internas e retorna os dados ao cliente
    - Toda a comunicação é realizada sobre o protocolo HTTP

HTTP Básico

- Baseia-se no método requisição e resposta (request/response)
  - Cliente pode fazer requisições usando verbos HTTP
  - Servidor retorna uma resposta com um código de status HTTP



HTTP Básico

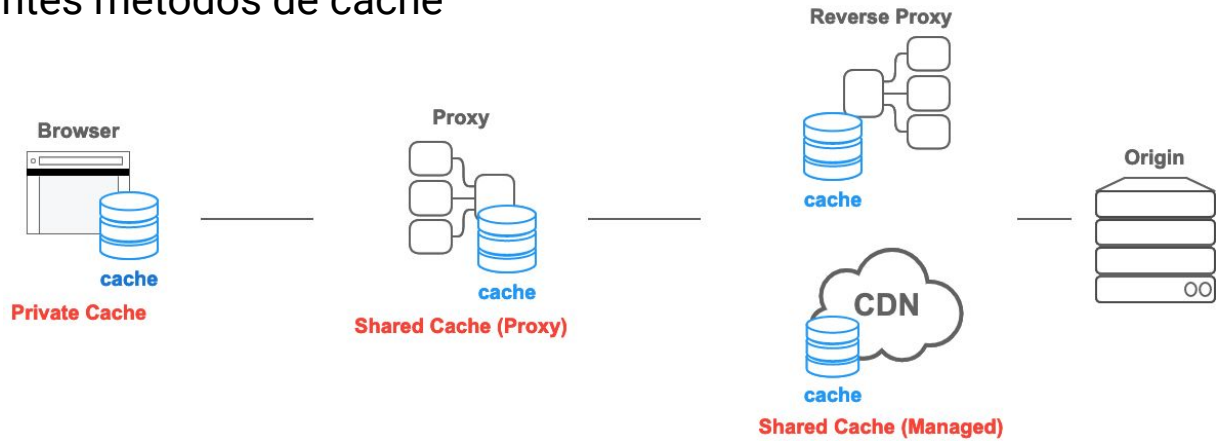
- Uma requisição contém:
  - Verbo HTTP (os mais comuns)
    - `GET`: retorna um recurso
    - `POST`: cria um novo recurso
    - `PUT`: edita um recurso
    - `PATCH`: edita parte de um recurso
    - `DELETE`: exclui um recurso
  - Cabeçalho HTTP de requisição (header)
    - Fornece informações adicionais
    - Altera ou melhora a precisão da semântica da requisição
  - Corpo de requisição (body)
    - Conteúdo da mensagem da requisição

HTTP Básico

- Uma resposta contém:
  - Status HTTP (os mais comuns)
    - `200` (`OK`): A requisição foi bem sucedida
    - `202` (`ACCEPTED`): A requisição foi recebida mas nenhuma ação foi tomada
    - `403` (`FORBIDDEN`): O cliente não tem direitos de acesso ao recurso
    - `404` (`NOT FOUND`): O servidor não pode encontrar o recurso solicitado
    - `500` (`ERROR`): O servidor encontrou uma situação com a qual não sabe lidar
  - Cabeçalho HTTP de resposta (header)
    - Fornece informações adicionais
    - Altera ou melhora a precisão da semântica da resposta
  - Corpo de resposta (body)
    - Conteúdo da mensagem da resposta

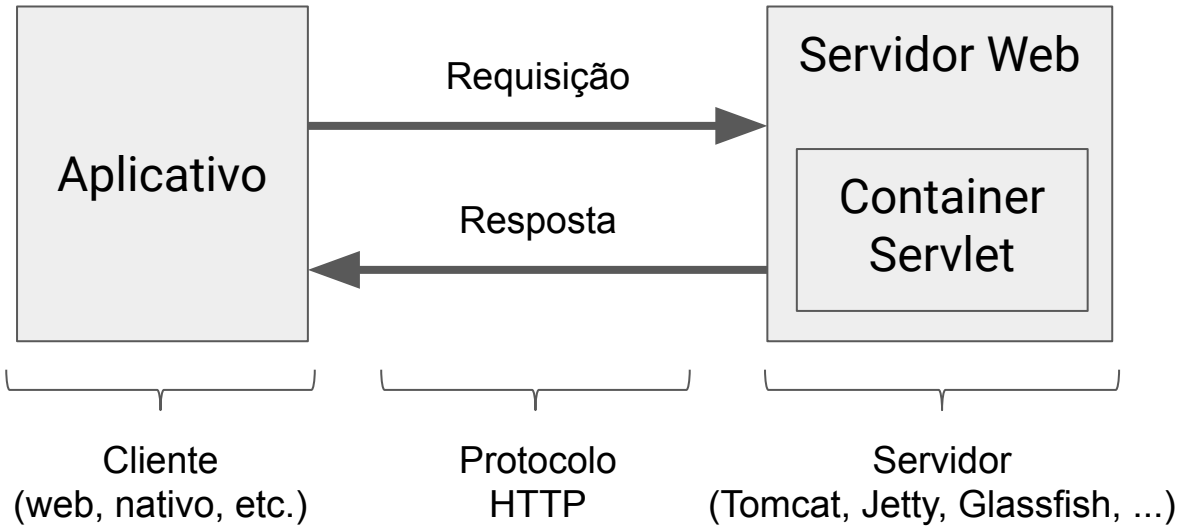
HTTP Básico

- Cache de dados
  - Cliente armazena uma resposta associada a uma solicitação
  - Resposta é reutilizada em requisições subsequentes
  - Quanto mais próximo cliente e cache estiver, mais rápida é a resposta
    - Suporta dois tipos de cache: privados e compartilhados
    - Suporta diferentes métodos de cache



Desenvolvimento de API com Java

- Especificação Servlet



Desenvolvimento de API com Java

- Especificação Servlet (cont.)
  - Setup básico
    - Download do Jetty: https://www.eclipse.org/jetty/download.php
    - Gerando um projeto web

```
mvn archetype:generate \
  -DgroupId=org.udesc \
  -DartifactId=otes06-webapi \
  -DarchetypeArtifactId=maven-archetype-webapp \
  -DinteractiveMode=false
```

    - Adicionar
    - Empacotando e gerando o artefato war

```
mvn package
```

    - Habilitando os módulos deploy e http

```
java -jar <caminho do jetty home>/start.jar --add-modules=deploy,http
```

Desenvolvimento de API com Java

- Especificação Servlet (cont.)
  - Setup básico
    - Download do Jetty: https://www.eclipse.org/jetty/download.php
    - Gerando um projeto web

```
mvn archetype:generate \
  -DgroupId=org.udesc \
  -DartifactId=otes06-webapi \
  -DarchetypeArtifactId=maven-archetype-webapp \
  -DinteractiveMode=false
```

    - Empacotando e gerando o artefato war

```
mvn package
```

    - Habilitando os módulos deploy e http

```
java -jar <caminho do jetty home>/start.jar --add-modules=deploy,http
```

Especificação Servlet

- Classe `HttpServlet` do pacote `javax.servlet`
  - Criar uma nova classe que estende de `HttpServlet`
  - Sobrescrever o método `service`

```
@WebServlet(urlPatterns = "/ola")
public class OlaMundoServlet extends HttpServlet {
    protected void service (HttpServletRequest request,
                            HttpServletResponse response)
                            throws ServletException, IOException {
        PrintWriter out = response.getWriter();
        out.println(String.format("Olá mundo!"));
    }
}
```

Criando um Servlet

- Classe `HttpServletRequest`
  - Informações sobre a requisição HTTP realizada para o servlet

- Classe `HttpServletResponse`
  - Informações sobre a resposta HTTP a ser entregue para o cliente

- Anotação `WebServlet`
  - Realiza o mapeamento entre a URL e o servlet que a controlará
  - Em especificações anteriores isto era realizado em arquivos XML
    - Arquivo WebContent > WEB-Inf > web.xml

Enviando dados para um Servlet

- Parâmetros da requisição
  - Estes dados podem ser informados por
    - URL pattern da requisição (usualmente com GET)
    - Payload da requisição (usualmente com POST)

- Recebendo parâmetros no Servlet
  - Os dados da requisição são recebidos por `HttpServletRequest`
  - Usa-se o método `getParameter`

Recebendo dados em um Servlet

- Parâmetros com o método getParameter

```
@WebServlet(urlPatterns = "/ola")
public class OlaServlet extends HttpServlet {
    protected void service(HttpServletRequest request,
                           HttpServletResponse response)
                               throws IOException {

        String nome = req.getParameter("nome");

        PrintWriter out = resp.getWriter();
        out.println(String.format("Olá %s", nome));
    }
}
```

Requisições GET e POST em um Servlet

- Suporte aos verbos HTTP em `HttpServlet`
  - O método `service` atende todos os verbos HTTP
  - Já os métodos `doGet` e `doPost` atendem apenas GET ou POST

```
@WebServlet(urlPatterns = "/ola")
public class OlaServlet extends HttpServlet {
    protected void doGet(HttpServletRequest req, HttpServletResponse res)
                                throws ServletException, IOException { ... }

    protected void doPost(HttpServletRequest req,HttpServletResponse res)
                                throws ServletException, IOException { ... }
}
```

Informações entre requisições distintas (sessão)

- Toda comunicação HTTP possui uma identificação
  - Esta identificação é chamada de Session ID (`JSESSIONID`)
  - O Session ID é criado e gerenciado pelo servidor de aplicação
    - Usa cookies para manter esta informação

- É possível associar dados temporários ao Session ID
  - Estes dados são armazenados em um objeto do tipo `HttpSession`
  - Exemplos de uso:
    - Autenticação
    - Carrinho de compras
    - Configuração de produto

Informações entre requisições distintas (sessão)

- Objeto `HttpSession`
  - Obtido pelo método `getSession` da classe `HttpServletRequest`
  - A classe `HttpSession` tem interface similar ao `HttpServletRequest`
    - `setAttribute`: adicionar um dado à sessão
    - `getAttribute`: recuperar um dado da sessão
    - `removeAttribute`: remover um dado da sessão
  - É possível eliminar todos os dados da sessão, incluindo o Session ID
    - Método `invalidate` da classe `HttpSession`

Informações entre requisições distintas (sessão)

- Objeto `HttpSession`

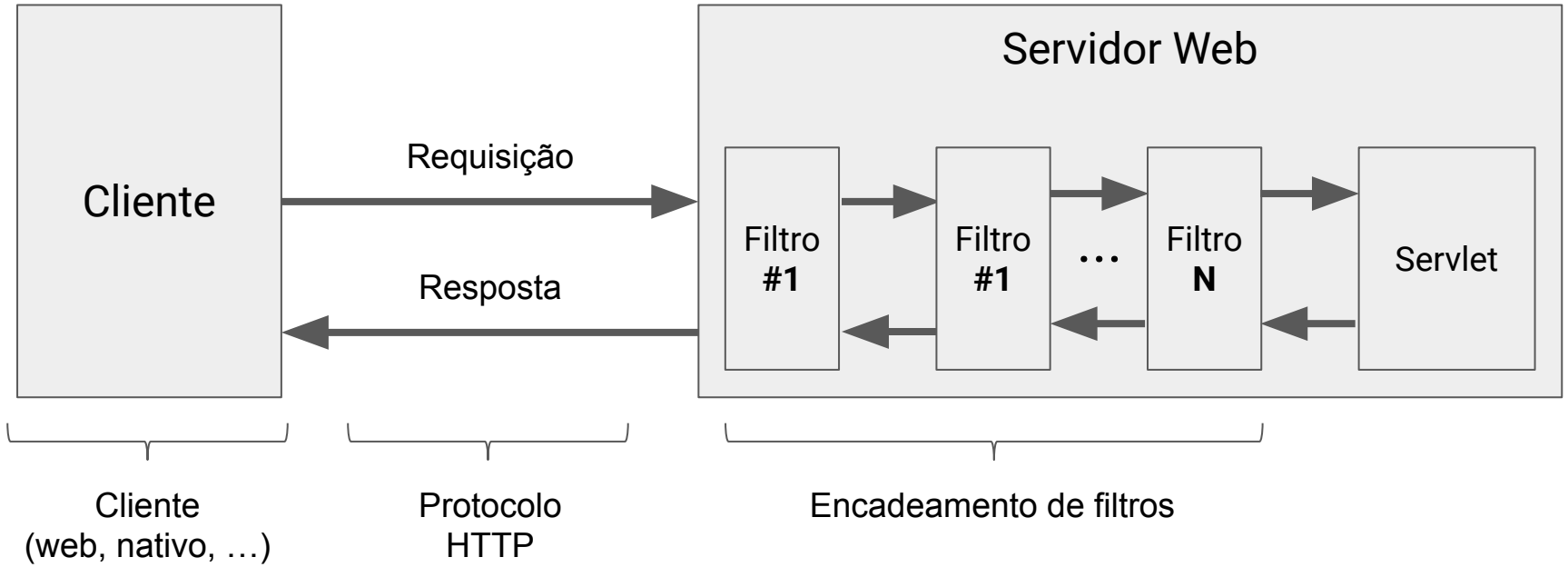
```
@WebServlet(urlPatterns = "/produto")
public class AdicionaProdutoServlet extends HttpServlet {
  protected void service(HttpServletRequest req, HttpServletResponse res)
    HttpSession session = req.getSession();
    List<String> carrinho = (List<String>) session.getAttribute("carrinho");

    if (carrinho == null) {
      carrinho = new ArrayList<>();
      session.setAttribute("carrinho",carrinho);
    }

    String produto = req.getParameter("produto");
    carrinho.add(produto);
  }
}
```

Aplicando filtros sobre requisições HTTP

- Possibilita realizar tratamentos antes de uma requisição
  - O filtro atua de forma similar ao Servlet
    - Também requer mapeamento de URL
    - Também recebe objetos que representam a requisição e resposta
  - Contudo, o filtro recebe também um objeto do tipo `FilterChain`
    - Permite interromper uma requisição ou repassa-la para frente
    - Pode existir vários filtros para uma única URL
    - O `FilterChain` controla este encadeamento de filtros

Aplicando filtros sobre requisições HTTP

- Encadeamento de filtros



Aplicando filtros sobre requisições HTTP

- Interface `Filter`

```
@WebFilter(urlPatterns = "/produto")
public class AdicionaProdutoFilter implements Filter {
  protected void doFilter (ServletRequest request,
                           ServletResponse response,
                           FilterChain filter) throws Exception {
    HttpSession session = request.getSession();
    String logado = (String) session.getAttribute("usuario");

    if (logado != null) {
      filter.doFilter(request, response);
    } else {
      System.out.println("Não autenticado, requisição cancelada");
    }
  }
}
```

Criando Web Services com Servlets

- Web Services é uma forma de integração entre sistemas
  - Permite integrar aplicativos desenvolvidos em diferentes tecnologias
  - Utiliza como base o protocolo HTTP para transferência de dados
  - Em geral o padrão utilizado para transferência de dados é XML ou JSON

- É possível criar Web Services a partir de Servlets
  - Por padrão a resposta de um Servlet é um conteúdo do tipo `text/html`
  - O tipo do conteúdo pode ser alterado pelo método `setContentType`
    - Este método pertence a classe `HttpServletRequest`

Criando Web Services com Servlets

- Método `setContentType`

```
@WebServlet(urlPatterns = "/produtos")
public class ProdutoServlet extends Filter {
  protected void service(HttpServletRequest request,
                         HttpServletResponse response) throws Exception {

    response.setContentType("application/json");

    response.getWriter().print("{ " +
                               "\"id\": 1," +
                               "\"nome\": \"Joao\"," +
                               "\"email\": \"joao@gmail.com\"" +
                             "}");
  }
}
```

Criando Web Services com Servlets

- Método `setContentType`
  - Permite definir o formato de transferência de dados
  - Os formatos são independentes de linguagem de programação
  - Os principais formatos são XML e JSON

- JSON
  - Formato leve para transferência de dados
  - Baseado na representação de objetos literais em JavaScript

- XML
  - Linguagem de marcação com regras de formatação de dados
  - Recomendação da W3C derivada da SGML

Convertendo objetos para JSON

- Uso de bibliotecas externas como Jackson e GSON
  - Converte um objeto Java para uma string em formato JSON

- Requer a adição de uma nova biblioteca ao projeto web
  - https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-databind
  - https://mvnrepository.com/artifact/com.google.code.gson/gson
  - Adicionar em WebContent > lib

Criando Web Services com Servlets

- Biblioteca GSON

```
@WebServlet(urlPatterns = "/produtos")
public class ProdutoServlet extends Filter {
  protected void service(HttpServletRequest request,
                         HttpServletResponse response) throws Exception {
    Gson gson = new Gson();
    Contato contato = new Contato();
    contato.setId(1);
    contato.setNome("Joao");
    contato.setEmail("joao@gmail.com");

    response.setContentType("application/json");
    response.getWriter().print(gson.toJson(contato));
  }
}
```

Criando Web Services com Servlets

- Biblioteca GSON

```
@WebServlet(urlPatterns = "/produtos")
public class ProdutoServlet extends Filter {
  protected void service(HttpServletRequest request,
                         HttpServletResponse response) throws Exception {
    Gson gson = new Gson();
    String json = "{\"id\":1,\"nome\":\"Joao\",\"email\":\"jao@gmail.com\" }";

    Contato contato = gson.fromJson(json, Contato.class);

    response.setContentType("application/json");
    response.getWriter().print(gson.toJson(contato));
  }
}
```

API REST

- Conceitos básicos
  - Arquitetura cliente-servidor
    - Baseada em clientes, servidores e recursos
    - Solicitações e respostas são feitas via protocolo HTTP
    - Independência entre cliente e servidor em termos de tecnologias usadas
  - Comunicação stateless
    - Não deve armazenar nenhuma informação entre as requisições
    - A requisição precisa ter todos os dados necessários para ser atendida
    - Não deve depender de informações já armazenadas em outras sessões

API REST

- Conceitos básicos (cont.)
  - Cache
    - Deve ser desenvolvida de modo que consiga armazenar dados em cache
    - As requisições e respostas entre cliente e servidor são otimizadas
    - Reduz comunicação desnecessária

  - Interface uniforme
    - Oferece uma comunicação padronizada entre o cliente e o servidor
    - Manipulação de recursos através de representações (como JSON ou XML)
    - Uso adequado de verbos HTTP e URI para manipulação dos recursos

API REST

- Conceitos básicos (cont.)
  - Sistema de camadas (layered architecture API)
    - Cada camada deve possuir uma funcionalidade específica
    - Cada camada é responsável por uma etapa da requisição e resposta
    - As camadas são ordenadas hierarquicamente, interagindo entre si

Spring Framework

- Framework para facilitar o desenvolvimento de aplicações em Java
  - Usa como base os conceitos:
    - Inversão de Controle
    - Injeção de Dependências

- Atualmente é um ecossistema de frameworks e bibliotecas de código
  - Componentes de software reusável para diferentes tipos de aplicações
  - Os principais são
    - Spring boot
    - Spring data
    - Spring cloud
    - Spring batch

Spring Boot

- Possibilita desenvolver rapidamente aplicações web ou standalone
  - Geralmente utilizado para criar aplicações baseadas em microsserviço
    - Cada projeto é um microsserviço responsável por uma parte da aplicação
    - Oferece simplicidade e robustez para a criação de API REST
    - Possui um servidor de aplicação embarcado
    - Usa um padrão conhecido como convenção sobre configuração

- Criar um novo projeto com Spring initializr
  - https://start.spring.io
  - Informar a dependência web

Spring Boot

- Iniciando a aplicação

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

API REST com Spring Boot

- Principais classes e anotações
  - `@RestController`
    - Define uma classe como controlador
    - Responsável por receber e atender requisições um conjunto de API REST
  - `@RequestMapping`
    - Define a rota a ser configurada para o controlador
    - A rota é o endereço do recurso
  - `@GetMapping`, `@PostMapping`, `@PutMapping` e `@DeleteMapping`
    - Define um método para receber requisições GET
    - Pode ser especificado um sufixo na rota e parâmetros

API REST com Spring Boot

- Principais classes e anotações (cont.)
  - `@PathVariable`
    - Parâmetros da URL capturados por parâmetros do método do controlador
  - `@RequestBody`
    - Corpo da requisição capturado por parâmetro do método do controlador
  - `ResponseEntity`
    - Classe de define o retorno da API REST
    - O objeto de resposta será serializado conforme definido (padrão é JSON)

API REST com Spring Boot

- Principais classes e anotações (cont.)

```
@RestController
@RequestMapping("/ola")
public class OlaMundoController {
    @GetMapping
    public ResponseEntity<String> get() {
        return ResponseEntity.ok("Olá mundo!");
    }
}
```

API REST com Spring Boot

- Principais classes e anotações (cont.)

```
public class Produto {
    private Integer codigo;
    private String nome;
    private Double preco;

    public void setCodigo(Integer codigo) { this.codigo = codigo; };
    public void setNome(String nome) { this.nome = nome; };
    public void setPreco(Double preco) { this.preco = preco; };

    public void getCodigo() { return codigo; };
    public String getNome() { return nome; };
    public Double getPreco() { return preco; };
}
```

API REST com Spring Boot

- Principais classes e anotações (cont.)

```
@RestController
@RequestMapping("/produtos")
public class ProdutoController {
    @GetMapping
    public ResponseEntity<Produto> get(@PathVariable Integer codigo) {
        var produto = new Produto();
        produto.setCodigo(codigo);
        produto.setNome("Nome do produto");
        produto.setPreco(150.99);

        return ResponseEntity.ok(produto);
    }
}
```

API REST com Spring Boot

- Principais classes e anotações (cont.)

```
@RestController
@RequestMapping("/produtos")
public class ProdutoController {
    @PostMapping
    public ResponseEntity<Produto> post(@RequestBody Produto produto) {
        //cadastrar produto
        ...

        return ResponseEntity.ok(produto);
    }
}
```

Exercícios

- Implemente os serviços para um aplicativo de agenda de contatos. Este aplicativo deve consumir APIs REST para manipular um contato.

  O recurso `Contato` deve possuir os atributos `nome`, `telefone` e `e-mail`.

  As APIs necessárias serão:
  - `GET /contatos/{id}`: retornar dados de um contato com o identificador informado
  - `GET /contatos`: retornar a lista de contatos cadastrados
  - `POST /contatos`: criar um novo contato, os dados são enviados no corpo da requisição
  - `PUT /contatos/{id}`: editar um contato, os dados são enviados no corpo da requisição

Desenvolvimento de API

Programação para dispositivos móveis

Prof. Allan Rodrigo Leite