DESENVOLVIMENTO *WEB* – *BACK END*

AULA 6

Prof.ª Luciane Yanase Hirabara Kanashiro

CONVERSA INICIAL

Vamos compreender as distinções entre APIs e *Web Services*. Em seguida, ao explorarmos a arquitetura de *Web Services*, trataremos de protocolos como SOAP e REST, delineando a estrutura que possibilita a troca de dados e o acesso a recursos. O formato JSON, ou JavaScript Object Notation, surge como uma ferramenta versátil e leve na representação de dados em serviços *web*. Enquanto isso, os métodos associados às APIs REST introduzem abordagens pragmáticas para manipular recursos, destacando-se pela simplicidade e eficiência. Além disso, enfocamos a importância da documentação de API, essencial para orientar desenvolvedores na utilização adequada dessas interfaces, visto que assegura a clareza e consistência necessárias para uma integração bem-sucedida. Esses tópicos formam um panorama essencial para explorar as complexidades e oportunidades no desenvolvimento de sistemas modernos.

TEMA 1 – API E *WEB SERVICES*

API e *Web Services* são dois conceitos que ouvimos com frequência quando se fala em desenvolvimento de *software*. Provavelmente já devemos ter nos deparado com inúmeras definições sobre esses dois conceitos. Mas realmente sabemos a diferença entre API e *Web Service*?

Uma **API**, ou **Interface de Programação de Aplicações**, é um conjunto de rotinas e padrões de programação que proporcionam uma ponte eficiente para a interação entre diferentes *softwares*. Seu principal objetivo é facilitar o acesso a aplicativos de *software*, o que possibilita que programadores se comuniquem com essas aplicações de maneira padronizada e eficaz.

As APIs desempenham um papel fundamental em plataformas baseadas na *web*, em que a comunicação entre sistemas distribuídos é essencial. Utilizada por programas e aplicações, a API define regras e protocolos que governam a interação entre diferentes componentes de *software*, o que possibilita a transferência de dados e a execução de operações específicas.

Ao possibilitar a comunicação entre duas ou mais aplicações, as APIs tornam-se a espinha dorsal da integração de sistemas, viabilizando a interoperabilidade e a colaboração eficiente entre *softwares* distintos. Sua

abordagem padronizada simplifica a complexidade da comunicação entre aplicações, promovendo a eficiência no desenvolvimento de *software* e a criação de ecossistemas digitais interconectados.

No contexto de APIs, a palavra *aplicação* refere-se a qualquer *software* com uma função distinta. A interface pode ser pensada como um contrato de serviço entre duas aplicações. Esse contrato define como as duas se comunicam usando solicitações e respostas. A documentação de suas respectivas APIs contém informações sobre como os desenvolvedores devem estruturar essas solicitações e respostas.

Agora, abordaremos o *Web Services*. Segundo Ferreira (2021), eles são conjuntos de programas que podem ser publicados, buscados e chamados por meio da internet. Esses programas podem realizar um simples processo de troca de mensagens, como também complexas transações comerciais ou industriais, por exemplo, um processo de compra de produtos. Toda vez que alguém publica o *back-end* de uma aplicação em um servidor *web* para que ele possa ser acessado por outros sistemas por intermédio de protocolos, ele se torna um *Web Service*.

Podemos ainda definir um ***Web Service*** como um **tipo específico de API** que opera na *web*, o que possibilita que sistemas diferentes se comuniquem e compartilhem dados pela internet. Um *Web Service* utiliza padrões como HTTP para realizar suas operações e, geralmente, segue protocolos como REST (*Representational State Transfer*) ou SOAP (*Simple Object Access Protocol*). Portanto, todos os ***Web Services*** são, de fato, **APIs**, pois fornecem uma interface para a interação entre sistemas. Em contrapartida, o termo API (Interface de Programação de Aplicações) é mais amplo e não se limita aos serviços *web*. Uma API pode ser um conjunto de rotinas e padrões de programação que possibilita a comunicação entre diferentes *softwares*, independentemente de estar na *web* ou não. APIs podem ser utilizadas localmente, em um mesmo sistema, ou em redes internas, não necessariamente na internet. Portanto, nem toda API é um *Web Service*, pois as APIs podem se manifestar de várias formas e não estão exclusivamente ligadas à comunicação pela *web*.

A seguir, são dados três exemplos de utilização de API, os quais demonstram que as APIs desempenham um papel fundamental na integração

de diferentes sistemas, facilitando a comunicação e a execução de operações específicas.

- **API de banco de dados local**: muitos sistemas de gerenciamento de banco de dados (DBMS) oferecem APIs para possibilitar a comunicação programática com o banco de dados. Por exemplo, a Java Database Connectivity (JDBC) é uma API Java que possibilita que aplicativos Java interajam com bancos de dados relacionais. As operações, como consultas, inserções, atualizações e exclusões, podem ser realizadas por meio dessa API, sem a necessidade de ser uma aplicação *web*.
- **Formas de pagamento (PagSeguro, Cielo, PayPal)**: quando realizamos uma compra *on-line* e efetuamos o pagamento, as plataformas de pagamento, como PagSeguro, Cielo ou PayPal, disponibilizam APIs para que o *site* de comércio eletrônico se comunique diretamente com o serviço de pagamento. A API desses provedores de pagamento possibilita que o *site* envie informações sobre a compra e receba confirmações de pagamento de maneira segura e eficiente. Isso simplifica a integração e oferece uma experiência de compra tranquila para o usuário, enquanto a segurança das transações é mantida.
- **Atendimento via WhatsApp Business**: no caso do atendimento ao cliente via WhatsApp, o WhatsApp Business disponibiliza uma API que integra o sistema da empresa e a plataforma de mensagens. Essa API possibilita a automação de respostas, o envio de notificações e até mesmo a realização de transações dentro da plataforma de mensagens. Dessa forma, as empresas podem oferecer suporte ao cliente de maneira eficiente, automatizada e acessível por meio do aplicativo de mensagens.

Nos dois últimos exemplos, fica bem evidente que as APIs são essenciais para a criação de uma experiência integrada e eficaz para os usuários da *web*. Elas facilitam a interação entre sistemas distintos, de modo que informações sejam compartilhadas de maneira segura e eficiente, contribuindo para a funcionalidade suave e aprimorada das operações *on-line*.

TEMA 2 – ARQUITETURA E TECNOLOGIAS DE *WEB SERVICES*

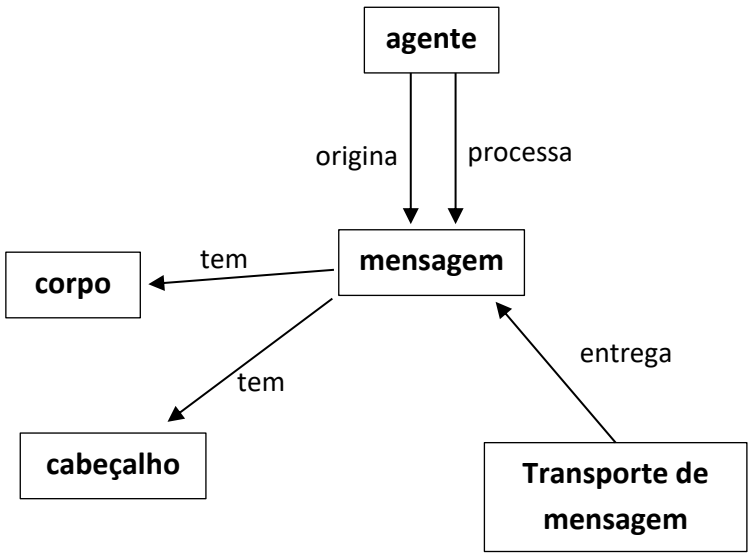
Existem diferentes modelos arquiteturais dentro da arquitetura geral dos *Web Services*. Abordaremos três modelos: modelo orientado a mensagens,

modelo orientado a serviço, modelo orientado a recursos (ROA). A seguir, veremos as características de cada modelo segundo Booth (2004).

2.1 Modelo orientado a mensagens

O modelo orientado a mensagens centra-se nas mensagens, em sua estrutura, seu transporte e assim por diante, sem referência específica às razões das mensagens, nem ao seu significado. É nesse modelo que são implementados os serviços baseados em SOAP. A essência do modelo de mensagem gira em torno de alguns conceitos-chave: o **agente** que envia e recebe **mensagens**, a estrutura da mensagem em termos de **cabeçalhos** e **corpos** de mensagens e os mecanismos usados para **entregar mensagens** (**Transporte de mensagem**). No diagrama da Figura 1 a seguir, mostramos resumidamente os conceitos-chave.

Figura 1 – Modelo Simplificado Orientado a Mensagens



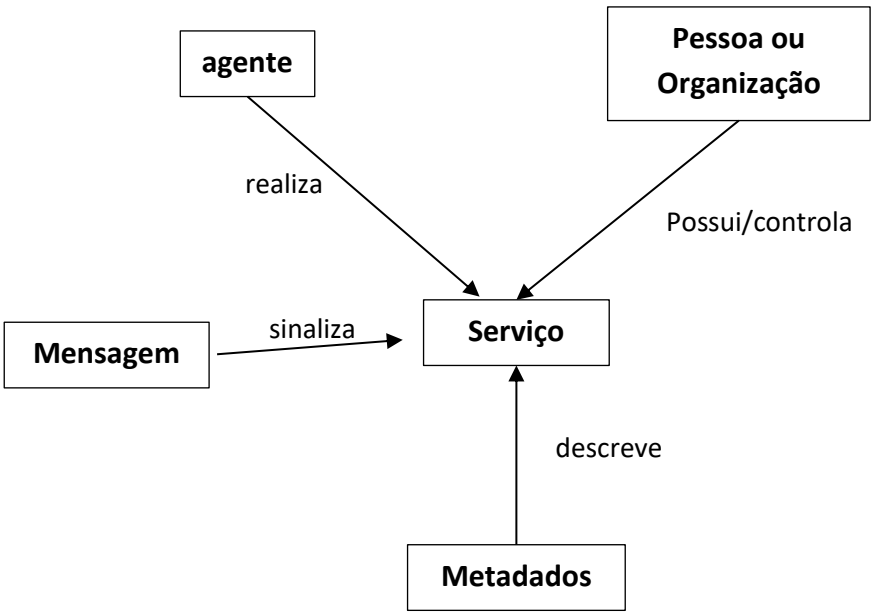
Fonte: Booth, 2004.

2.2 Modelo orientado a serviços

Considerado o mais complexo de todos os modelos da arquitetura. No entanto, também gira em torno de algumas ideias-chave. Um **serviço** é realizado por um **agente** e utilizado por outro agente. Os serviços são mediados por meio de **mensagens** trocadas entre **agentes solicitantes** e **agentes provedores**. O modelo orientado a serviços faz uso de **metadados**, que é uma propriedade-chave das arquiteturas orientadas a serviços. Modelo orientado a serviços (SOA)

concentra-se em aspectos de serviço e suas ações. No diagrama da Figura 2, mostramos resumidamente os conceitos-chave.

Figura 2 – Modelo simplificado orientado a serviços

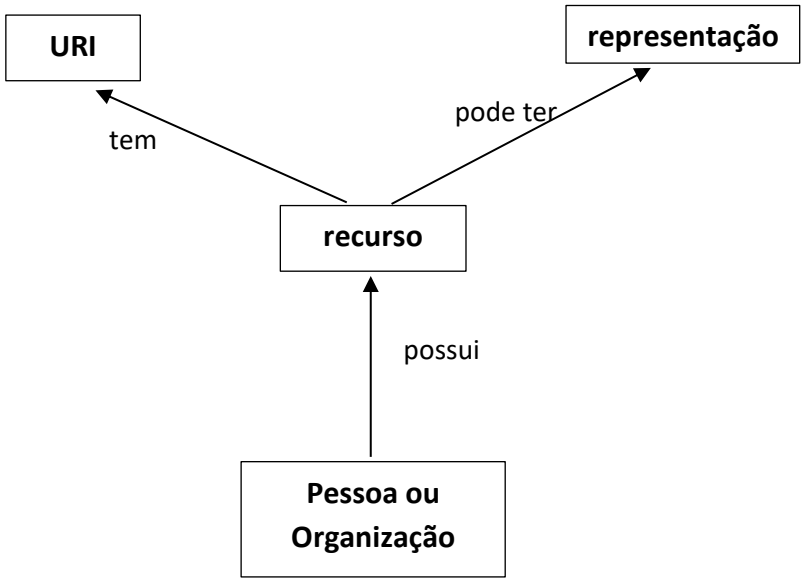


Fonte: Booth, 2004.

2.3 Modelo orientado a recursos

O modelo orientado a recursos (ROA) concentra-se nos **recursos** (documentos ou serviços que podem ser acessados por **URI**s) que existem e têm **proprietários** e suas **representações** (objetos de dados que refletem o estado de um recurso). Modelo de recursos é adotado por meio do conceito de recurso da Arquitetura da *Web*. *Web Services* RESTful são implementados por esse modelo arquitetural e serão discutidos com mais detalhes posteriormente. No diagrama da Figura 3, mostramos resumidamente os conceitos-chave.

Figura 3 – Modelo simplificado orientado a recursos



Fonte: Booth, 2004.

2.1. Tecnologias dos *Web Services*

As tecnologias de *Web Services* são conjuntos de protocolos e padrões (ou, ainda, restrições de arquitetura) que possibilitam a comunicação e a interoperabilidade entre sistemas distribuídos na *web*. Elas são fundamentais para o desenvolvimento de aplicações que precisam trocar dados e serviços pela internet de maneira padronizada e eficiente. A seguir, estão descritos alguns dos principais componentes e tecnologias associados a *Web Services*.

- ***Simple Object Access Protocol* (SOAP)**: representa uma abordagem altamente estruturada para a comunicação entre aplicações, baseando-se no formato de dados XML para a troca de informações. Funcionando como um Protocolo Simples de Acesso a Objetos, o SOAP é um padrão para a comunicação entre sistemas distribuídos, que oferece uma estrutura rigorosa e predefinida para a troca de mensagens. Sua utilização é dedicada principalmente à troca de dados no formato XML, que proporciona uma linguagem comum para sistemas heterogêneos se entenderem mutuamente. O SOAP é particularmente útil em ambientes nos quais a padronização e a estrutura são fundamentais, tornando-o adequado para aplicações legadas e APIs privadas, em que a integração eficiente entre sistemas com diferentes tecnologias é essencial. Embora seu uso tenha diminuído em algumas áreas em favor de abordagens mais leves como REST, o SOAP continua desempenhando um papel crucial

em cenários que demandam uma troca de dados altamente estruturada e confiável.

- ***Representational State Transfer* (REST) ou RESTful**: é um estilo de arquitetura para projetar interfaces de comunicação em sistemas distribuídos. De acordo com REDHAT (2024), REST não é um protocolo ou padrão, mas sim um conjunto de restrições de arquitetura. Os desenvolvedores de API podem implementar a arquitetura REST de maneiras variadas. Diferentemente de abordagens mais pesadas, REST é conhecido por sua simplicidade e escalabilidade. Um dos princípios fundamentais do REST é a Transferência Representacional de Estado, que sugere que as representações dos recursos, em vez do estado do próprio sistema, devem ser transferidas entre os componentes. O REST opera exclusivamente com *Hypertext Transfer Protocol Secure* (HTTPS), garantindo a segurança das comunicações. Uma característica flexível do REST é sua capacidade de suportar vários formatos de dados, incluindo XML, JSON, texto simples e HTML. A abordagem RESTful tem se tornado cada vez mais predominante, especialmente em serviços *web* que buscam uma comunicação eficiente, leve e orientada a recursos.
- **HTTP**: Protocolo de Transferência de Hipertexto (em inglês, *Hypertext Transfer Protocol*), é um protocolo de comunicação utilizado para transferência de dados na *World Wide Web*. O HTTP define a maneira como as mensagens são formatadas e transmitidas, bem como servidores e navegadores devem responder a diversos comandos.
- ***Extensible Markup Language* (XML)**: é uma linguagem de marcação projetada para armazenar e transportar dados. Ela fornece uma maneira flexível e padrão de representar informações estruturadas de forma legível por máquina e, ao mesmo tempo, compreensível por humanos. Assim como o HTML, o XML é uma linguagem de marcação que utiliza *tags* para definir elementos e estruturar dados. Segundo Amazon Web Services – AWS (2024), embora os arquivos HTML e XML sejam muito semelhantes, existem algumas diferenças importantes citadas a seguir:
  - **finalidade** – o objetivo da HTML é apresentar e exibir dados; por sua vez, a XML armazena e transporta dado;
  - **etiquetas** – a HTML tem etiquetas predefinidas, mas os usuários podem criar e definir suas próprias etiquetas em XML;

- **regras de sintaxe** – existem algumas diferenças secundárias, porém importantes, entre as sintaxes HTML e XML. Por exemplo, a XML diferencia maiúsculas de minúsculas, mas a HTML não. Os analisadores XML apresentarão erros se escrevermos uma etiqueta como <Book> em vez de <book>.

- ***JavaScript Object Notation* (JSON)**: é um formato leve de troca de dados. Está baseado em um subconjunto da linguagem de programação JavaScript. JSON utiliza uma estrutura de pares chave-valor, em que os dados são organizados em pares formados por uma chave (uma *string*) e um valor associado. Os valores podem ser *strings*, números, objetos, *arrays*, booleanos ou nulos. Abordaremos o JSON com mais detalhes em um tópico específico.
- **Web Services Description Language (WSDL)**: é uma linguagem que descreve os serviços *web*. WSDL é frequentemente usado em conjunto com o protocolo SOAP. É baseada em XML e, além de descrever o serviço, especifica como acessá-lo e quais as operações ou métodos disponíveis.

TEMA 3 – *JAVASCRIPT OBJECT NOTATION* (JSON)

Já citamos *JavaScript Object Notation* (JSON) anteriormente. Neste tópico, vamos abordá-lo com mais detalhes. As especificações sobre JSON foram extraídas da página oficial do JSON (2024).

JSON está baseado em um subconjunto da linguagem de programação JavaScript. JSON é em formato texto e completamente independente de linguagem, pois usa convenções que são familiares às linguagens C e familiares, incluindo C++, C#, Java, JavaScript, Perl, Python, entre muitas outras. Essas propriedades fazem com que JSON seja um formato ideal de troca de dados.

JSON está constituído em duas estruturas:

- uma coleção de pares nome/valor – em várias linguagens, isso é caracterizado como um *object*, *record*, *struct*, dicionário, *hash table*, *keyed list*, ou *arrays* associativas;
- uma lista ordenada de valores – na maioria das linguagens, isso é caracterizado como uma *array*, vetor, lista ou sequência.

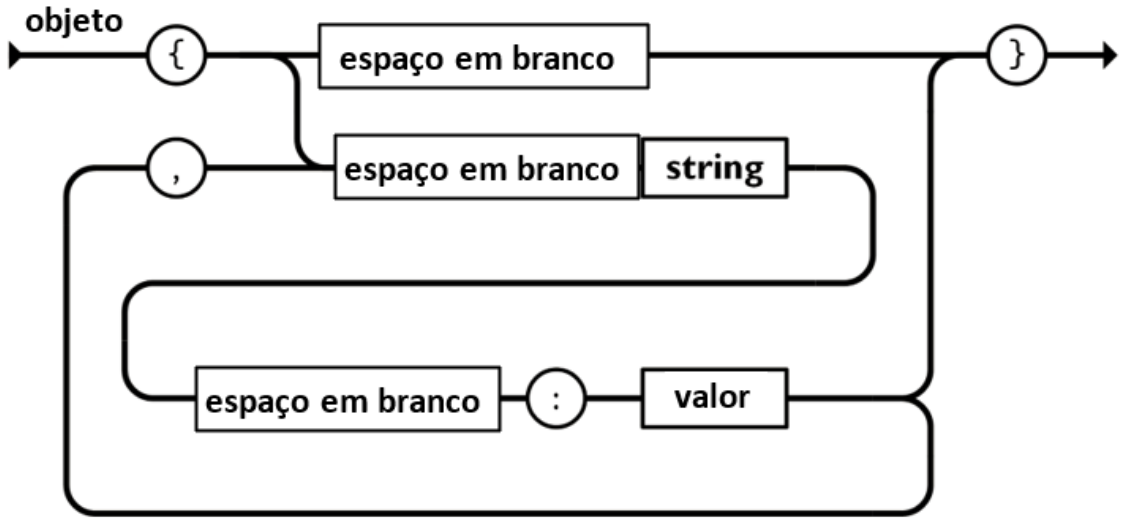
Essas são estruturas de dados universais. Virtualmente, todas as linguagens de programação modernas as suportam, de uma forma ou de outra. É aceitável que um formato de troca de dados que seja independente de linguagem de programação se baseie nessas estruturas.

3.1. Representação dos dados

Em JSON, os dados são apresentados da forma descrita a seguir.

Um objeto é um conjunto desordenado de pares nome/valor. Um objeto começa com { (chave de abertura) e termina com } (chave de fechamento). Cada nome é seguido por dois-pontos (:) e os pares nome/valor são seguidos por vírgula (,). No diagrama da Figura 4, ilustramos a representação de um objeto em JSON.

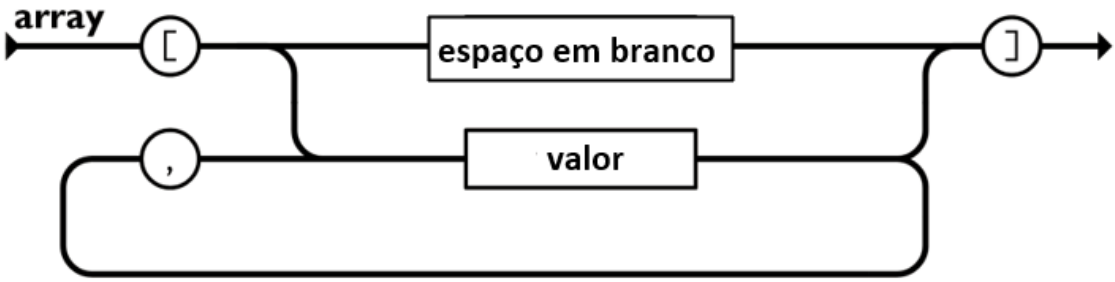
Figura 4 – Diagrama de Representação de objeto JSON



Fonte: JSON, 2024.

Uma *array* é uma coleção de valores ordenados. O *array* começa com um colchete de abertura ([) e termina com um colchete de fechamento (]). Os valores são separados por vírgula (,). No diagrama da Figura 5, ilustramos a representação de um *array* em JSON.

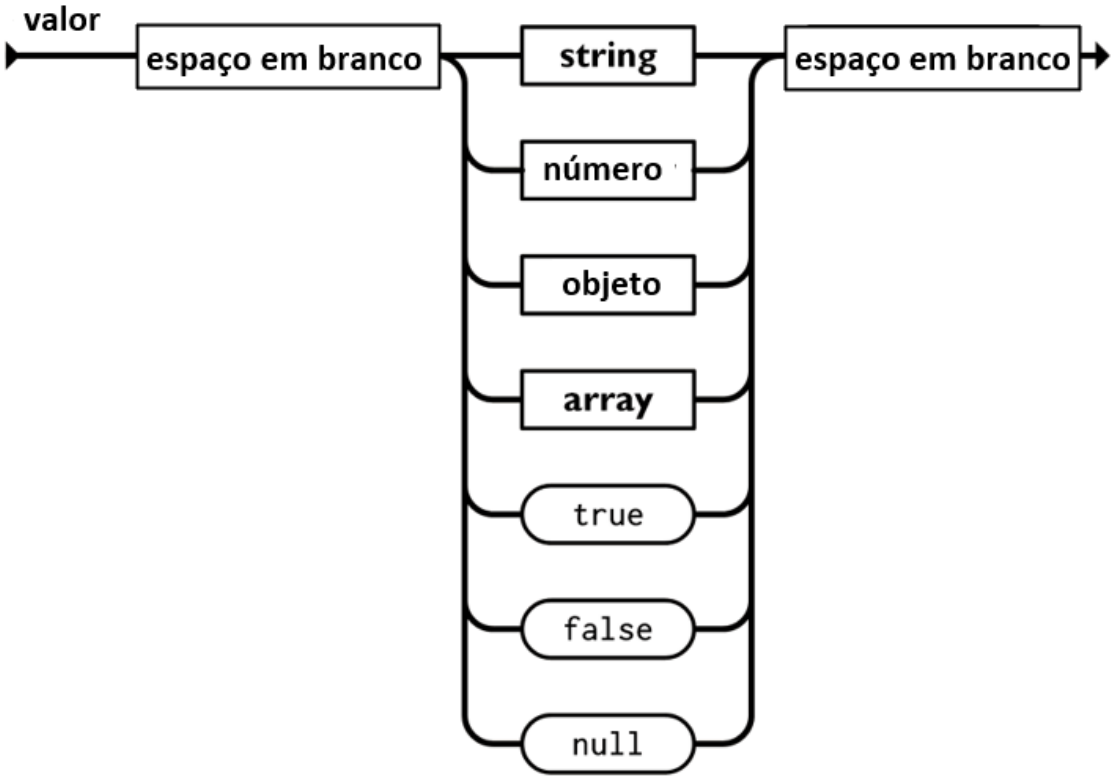
Figura 5 – Diagrama de Representação de *array* no JSON



Fonte: JSON, 2024.

Um valor (*value*, na Figura 5) pode ser uma cadeia de caracteres (*string*), ou um número, ou *true* ou *false*, ou *null*, ou um objeto ou uma *array*. Essas estruturas podem estar aninhadas. No diagrama da Figura 6, ilustramos o que pode ser um valor.

Figura 6 – Diagrama de Representação de valor em JSON



Fonte: JSON, 2024.

O conceito de *string* é semelhante ao que temos na linguagem C ou Java. Uma *string* é uma coleção de nenhum ou mais caracteres Unicode, envolvido entre aspas duplas usando barras invertidas como caractere de escape. Um caractere é representado como um simples caractere de *string*.

Como podemos observar, o JSON dá suporte a vários tipos de dados. O suporte a tipos de dados comum que o JSON fornece, o torna adequado para representar uma variedade de dados estruturados.

A seguir, temos um exemplo de objeto JSON. O objeto em questão representa informações sobre uma pessoa, com **chaves** como *nome*, *idade*, *cidade*, *ativo* e *interesses*, e **valores** correspondentes associados a essas chaves (respectivamente: `"João"`, `25`, `"Curitiba"`, `true` e `["leitura", "esportes", "programação"]`).

```
{
 "nome": "João",
 "idade": 25,
 "cidade": "Curitiba",
 "ativo": true,
 "interesses": ["leitura", "esportes", "programação"]
}
```

Vamos observar o código e compará-lo com os diagramas de representação. É interessante reparar que o objeto do exemplo começa com abertura de chaves ({) e termina com fechamento de chaves (}), e que o *array* "interesses" começa por abertura de colchete ([) e termina com fechamento de colchete (]). Observemos ainda que os valores podem ser de vários tipos em um mesmo objeto – no caso do exemplo anterior, temos *string*, número, *true* e *array*.

O JSON é uma escolha popular para a troca de dados em razão de sua simplicidade, legibilidade e suporte em várias linguagens de programação.

TEMA 4 – MÉTODOS DE API REST

Ao longo deste estudo, vimos conceitos sobre *Web Services* e APIs, como também conhecemos tecnologia de *Web Services*. Aprendemos, ainda, que *Web Services* são APIs, e a definição de REST. Mas realmente sabemos o que é uma API REST? O que diferencia uma API REST de uma API normal?

Nesta seção, veremos as características e métodos de uma API REST e entenderemos o que é API REST e as características que a fazem ser uma API REST.

Para uma API ser considerada do tipo RESTful (popularmente conhecida como *REST*), ela precisa estar em conformidade com os seguintes critérios (REDHAT):

- ter uma arquitetura cliente/servidor formada por clientes, servidores e recursos, com solicitações gerenciadas por HTTP;
- estabelecer uma comunicação *stateless* entre cliente e servidor. Isso significa que nenhuma informação do cliente é armazenada entre solicitações GET e toda as solicitações são separadas e desconectadas;
- armazenar dados em cache para otimizar as interações entre cliente e servidor;
- ter uma interface uniforme entre os componentes para as informações serem transferidas em um formato padronizado. Para tanto, é necessário que:
    - os recursos solicitados sejam identificáveis e estejam separados das representações enviadas ao cliente;
    - os recursos possam ser manipulados pelo cliente por meio da representação recebida com informações suficientes para essas ações;
    - as mensagens autodescritivas retornadas ao cliente contenham informações suficientes para descrever como processá-las;
    - hipertexto e hipermídia estejam disponíveis. Isso significa que, após acessar um recurso, o cliente pode usar *hiperlinks* para encontrar as demais ações disponíveis para ele no momento;
- ter um sistema em camadas que organiza os tipos de servidores (responsáveis pela segurança, pelo carregamento de carga, e assim por diante) envolvidos na recuperação das informações solicitadas em hierarquias que o cliente não pode ver;
- possibilitar código sob demanda (opcional) – a capacidade de enviar um código executável do servidor para o cliente quando solicitado para ampliar a funcionalidade disponível ao cliente.

Ao seguir esses princípios, **uma API é considerada RESTful**, o que geralmente leva a sistemas mais escaláveis, flexíveis e fáceis de manter. Vale ressaltar que a adesão estrita a todos esses princípios pode variar, e algumas

APIs podem implementar apenas uma parte deles e ainda serem consideradas RESTful.

Os métodos mais comuns incluem GET para recuperar um recurso, POST para criar um recurso, PUT para atualizar um recurso, PATCH para aplicar alterações parciais em um recurso e DELETE para excluir um recurso. O quadro a seguir descreve cada um desses métodos.

Quadro 1 – Métodos

Método	Descrição
GET	O método GET é usado para recuperar dados de um recurso específico. É uma operação de leitura e não deve ter efeitos colaterais no servidor.
POST	O método POST é utilizado para enviar dados para o servidor a fim de criar um novo recurso. Ele geralmente é usado para submeter formulários ou enviar dados complexos para processamento.
PUT	O método PUT é usado para atualizar um recurso existente no servidor. O cliente fornece todo o novo estado do recurso.
PATCH	O método PATCH é semelhante ao PUT, mas é utilizado para aplicar modificações parciais a um recurso. Em vez de enviar o recurso completo, apenas as alterações necessárias são enviadas.
DELETE	O método DELETE é usado para excluir um recurso específico no servidor.
OPTIONS	O método OPTIONS é utilizado para obter informações sobre as opções de comunicação disponíveis para um recurso ou servidor.
HEAD	O método HEAD é semelhante ao GET, mas é usado para obter apenas os cabeçalhos de resposta, sem o corpo da resposta. Isso é útil para verificar metadados de recursos sem baixar o conteúdo completo.

Esses métodos HTTP representam as operações básicas que podem ser realizadas em recursos em uma API REST. O uso apropriado desses métodos ajuda a garantir que a API seja consistente, escalável e siga os princípios do REST. As APIs REST também geralmente utilizam *Uniform Resource Identifiers* (URIs) para identificar recursos e respondem com dados no formato JSON ou XML.

A seguir, mostramos um exemplo de GET para uma API REST.

```
import org.springframework.web.bind.annotation.GetMapping;
import
org.springframework.web.bind.annotation.PathVariable;
import
org.springframework.web.bind.annotation.RestController;

@RestController
public class ExemploController {
```

```
  @GetMapping("/api/exemplo/{id}")
  public String getExemplo(@PathVariable Long id) {
    // Aqui você poderia acessar um serviço ou banco de
dados para obter dados com base no ID fornecido
    return "Você chamou o método GET com o ID: " + id;
  }
}
```

TEMA 5 – DOCUMENTAÇÃO

No desenvolvimento de *software*, não existe uma fase específica para a documentação. Muitas equipes de desenvolvimento adotam uma abordagem contínua para ela, integrando-a em todo o processo de desenvolvimento. A documentação não é uma tarefa a ser realizada apenas no final do ciclo de desenvolvimento. Em vez disso, deve ser incorporada ao longo de todo o processo, desde a concepção até a manutenção do *software*. Isso promove uma abordagem contínua e iterativa, garantindo que a documentação esteja sempre alinhada com o código.

Em ambientes ágeis, em que as mudanças são frequentes, a documentação deve ser flexível e capaz de se adaptar às alterações no *software*. Isso significa que os desenvolvedores e os documentadores devem trabalhar em estreita colaboração para manter a documentação atualizada em tempo real. Também podem ser utilizadas geradores de documentação e comentários no código para agilizar o processo de documentação. Isso ajuda a garantir que a documentação esteja sempre em sincronia com o código-fonte, reduzindo o risco de desatualizações.

Comentários claros e significativos no código são uma forma de documentação em si. Além disso, práticas como a documentação de contratos de API podem ser incorporadas diretamente nos testes automatizados.

A documentação não é exclusividade de uma equipe específica. Desenvolvedores, testadores, gerentes de projeto e outros membros da equipe contribuem para a documentação. Isso promove uma compreensão mais holística do *software* e uma colaboração mais eficaz. A documentação deve ser orientada para o usuário, focando nas necessidades reais dos desenvolvedores, operadores e outros *stakeholders*. Isso implica uma linguagem clara, exemplos práticos e uma estrutura que facilite a busca e a compreensão. A documentação beneficia-se de *feedback* contínuo, assim como o código. Os usuários,

desenvolvedores e outros *stakeholders* fornecem *insights* valiosos sobre a eficácia da documentação, colaborando com a melhoria contínua.

Apesar da documentação de *software* ser uma prática vital para garantir a compreensão, manutenção e evolução de sistemas, diversos entraves podem surgir durante o processo esse processo, especialmente quando há uma pressão para entregar o produto no prazo ao cliente e quando o código-fonte é considerado a única documentação disponível. A seguir, apresentamos alguns dos entraves associados a esses fatores.

- **Prazo de entrega priorizado**: em muitos projetos, a pressão para entregar o produto no prazo é intensa. Como resultado, a equipe de desenvolvimento pode priorizar a escrita de código em detrimento da documentação. Isso pode levar a uma documentação insuficiente ou, em alguns casos, à ausência total de documentação.
- **Percepção de prioridade menor para a documentação**: em alguns ambientes, a documentação é percebida como uma atividade secundária em comparação com a codificação. Isso pode ser exacerbado quando há uma ênfase excessiva na entrega rápida de funcionalidades, deixando a documentação em segundo plano.
- **Falta de recursos ou pessoal especializado**: a alocação insuficiente de recursos ou a ausência de pessoal especializado em documentação pode resultar em esforços de documentação inadequados. Às vezes, os desenvolvedores podem não ter habilidades específicas para produzir uma documentação clara e compreensível.
- **Visão do código como documentação suficiente**: em alguns cenários, os desenvolvedores podem acreditar que o código-fonte por si só é suficiente como documentação. Embora um código bem escrito seja valioso, ele nem sempre fornece informações completas sobre decisões de *design*, considerações de segurança, requisitos e casos de uso.
- **Mudanças rápidas e iterativas**: em ambientes ágeis, em que as mudanças são frequentes e a iteração é a norma, a documentação pode se tornar obsoleta rapidamente. Isso pode desencorajar os desenvolvedores a investir tempo na documentação, já que ela pode exigir atualizações constantes.
- **Falta de incentivos para documentar**: em alguns casos, a cultura organizacional ou a falta de incentivos para a documentação pode levar

os desenvolvedores a considerá-la como uma tarefa menos importante. A ausência de reconhecimento ou recompensas pode diminuir a motivação para documentar adequadamente.

- **Mantenedores dependendo exclusivamente do código-fonte**: quando a documentação é escassa e os mantenedores precisam depender exclusivamente do código-fonte, pode haver uma curva de aprendizado significativa. Isso pode levar a atrasos na solução de problemas, a uma difícil manutenção e ao aumento do risco de erros.

Para superar esses entraves, é importante reconhecer o valor da documentação desde o início do desenvolvimento, integrando-a ao processo de forma contínua, incentivando sua prática e proporcionando recursos e treinamento adequados aos desenvolvedores. A documentação eficaz é extremamente importante para a sustentabilidade, colaboração e evolução bem-sucedida de um *software* ao longo do tempo.

O caso mais clássico de transtorno que poderia ser evitado com documentação é o do acelerador médico Therac-25, que, em razão de falhas de *software*, ocasionou a superdosagem de radiação em pacientes com câncer. Uma documentação clara dos requisitos do sistema poderia ter ajudado a garantir que os desenvolvedores entendessem corretamente as funcionalidades e os limites do sistema, evitando assim a implementação de comportamentos indesejados. Ainda, a documentação detalhada da arquitetura do *software* poderia ter destacado os pontos críticos do sistema, como a interface entre o *software* e o *hardware*, em que ocorreram as falhas que levaram às superdosagens.

Um manual sobre como usar o sistema corretamente e quais precauções de segurança devem ser tomadas poderia ter ajudado a evitar erros humanos que contribuíram para a crise. Ainda na fase de testes e validação, uma documentação detalhada dos testes realizados no *software*, incluindo os casos de teste e os resultados esperados, poderia ter identificado as falhas de *software* antes que elas causassem danos aos pacientes.

Esse desastre da engenharia de *software* mostra como uma documentação mais completa e detalhada poderia ter ajudado o Therac-25 a fornecer informações claras e precisas sobre o funcionamento e os riscos do sistema, ajudando os desenvolvedores, operadores e outros envolvidos a entender e usar o sistema corretamente.

5.1 Documentação de API

A documentação de API serve como um guia completo sobre como utilizar uma API. Ela descreve detalhadamente os pontos de entrada, operações disponíveis, métodos HTTP suportados, bem como realizar solicitações e interpretar as respostas.

A documentação é uma fonte fundamental para esclarecer dúvidas relacionadas à sintaxe e funcionalidade da API. Ela fornece exemplos claros, explicações detalhadas e até mesmo casos de uso práticos para orientar os desenvolvedores no uso eficaz da API. Muitas APIs não têm uma interface gráfica do usuário (GUI) de *front-end*. Isso significa que os desenvolvedores não têm uma aplicação visual ou interativa para explorar e entender a API. Assim, a documentação se torna a principal fonte de informações sobre como interagir com a API. A documentação detalha cada campo e parâmetro associado à API, fornecendo informações sobre o tipo de dados esperado, possíveis valores e o propósito de cada elemento. Essa clareza é crucial para garantir a correta construção das solicitações e interpretação das respostas. Para que desenvolvedores possam integrar com sucesso suas aplicações à API, é necessário um recurso que forneça informações claras e estruturadas. A documentação desempenha esse papel, servindo como um guia passo a passo para facilitar a integração eficiente.

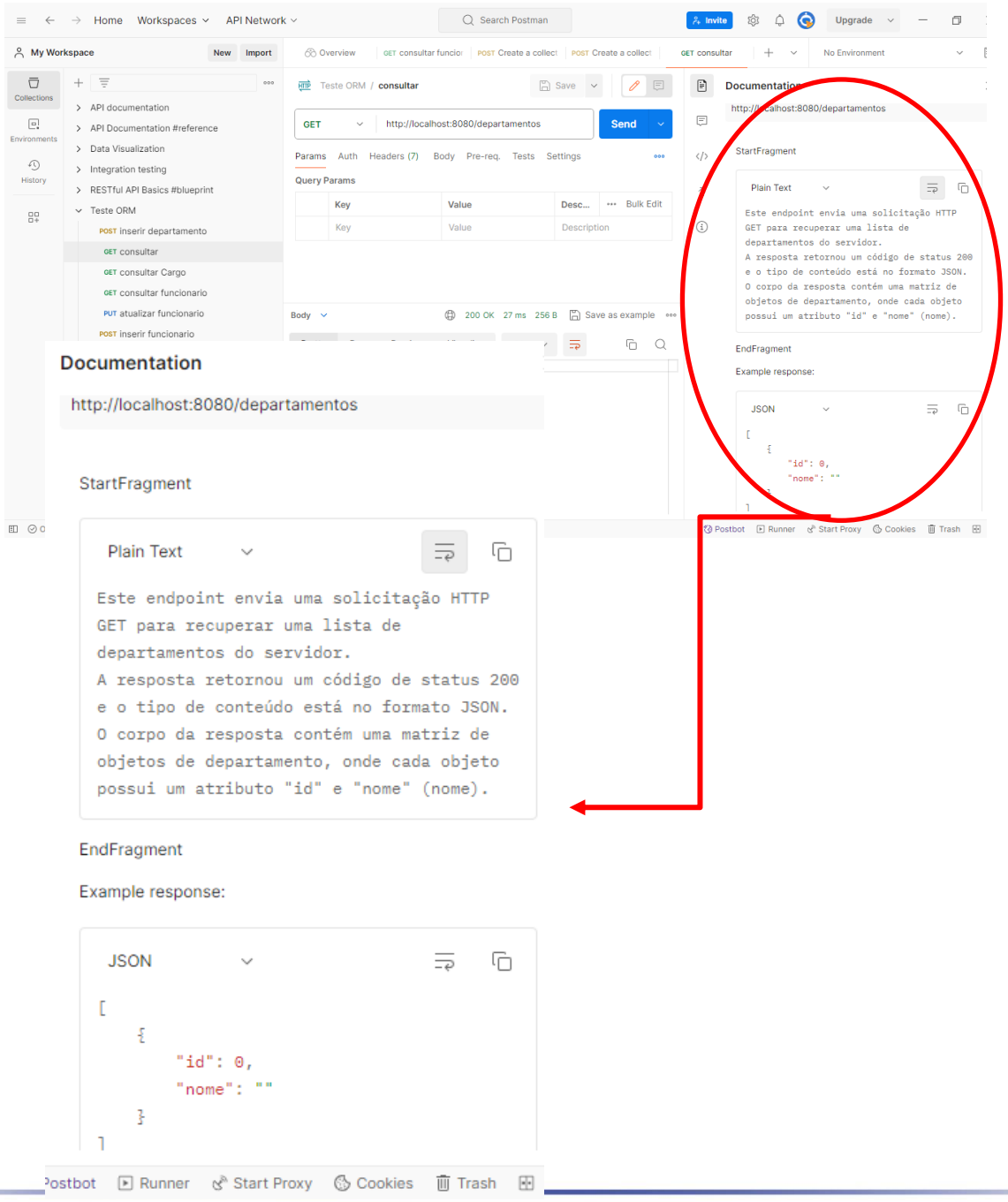
A documentação, quando mantida atualizada, garante consistência entre o que a API oferece e o que os desenvolvedores esperam. Isso ajuda a evitar discrepâncias e a assegurar que todos os envolvidos estejam na mesma página. A documentação pode incluir diretrizes sobre boas práticas de uso da API, considerações de segurança, autenticação e autorização. Essas informações são importantes para garantir que os desenvolvedores usem a API de maneira segura e eficaz. Uma documentação bem elaborada contribui para a transparência, possibilitando que desenvolvedores entendam o propósito de cada recurso, como ele deve ser usado e quais resultados esperar. Isso facilita o processo de desenvolvimento e solução de problemas.

A documentação de API é um componente fundamental para garantir uma integração eficiente, promover boas práticas de desenvolvimento, proporcionar clareza e oferecer suporte aos desenvolvedores que utilizam uma API, especialmente quando uma GUI de *front-end* não está disponível.

Já vimos que o Postman pode ser utilizado para testes de APIs, no entanto, podemos ainda utilizá-lo para documentar APIs, salvando as requisições de teste e transformando-as progressivamente na documentação da API.

No exemplo a seguir, ilustramos uma requisição GET de uma API. À direita, podemos observar uma descrição em português do que esse *endpoint* fez e um exemplo da resposta em JSON. Essa descrição pode ser ainda mais detalhada dependendo do tipo de requisição, podendo ser modificada, e, posteriormente, servirá como uma documentação para os desenvolvedores que utilizarão a API.

Figura 7 – Requisição GET de uma API



FINALIZANDO

Abordamos características e conceitos de API e *Web Service* que nos possibilitam diferenciá-los, bem como nos aprofundarmos na arquitetura que sustenta essas interfaces, explorando o versátil formato JSON, desvendando os métodos eficientes das APIs REST e ressaltando a importância da documentação.

É evidente que construir e integrar sistemas no atual cenário tecnológico exige uma compreensão abrangente desses elementos fundamentais. As APIs, como pontes digitais, oferecem conexões cruciais entre sistemas, enquanto os *Web Services*, com sua arquitetura específica, ampliam o alcance da interoperabilidade. O JSON, ágil e legível, emerge como um aliado na troca de dados, e os métodos REST, com sua simplicidade, proporcionam uma abordagem pragmática para operar sobre recursos. Não menos importante, a documentação de API, nossa bússola no vasto território de desenvolvimento, garante a clareza e a compreensão necessárias para uma implementação bem-sucedida.

Reconhecemos, ainda, a complexidade e a riqueza desses conceitos, fundamentais para quem busca desbravar os desafios e as oportunidades na construção de sistemas modernos e interoperáveis.

REFERÊNCIAS

AMAZON WEB SERVICES – AWS. **O que é XML?** Disponível em: <https://aws.amazon.com/pt/what-is/xml/>. Acesso em: 30 abr. 2024.

BOOTH, D. et al. (Ed.). **Web Service Architecture**: W3C Working Group Note. Wakefield, 11 Feb. 2004. Disponível em: <http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/#introduction>. Acesso em: 30 abr. 2024.

FERREIRA, A. G. **Interface de programação de aplicações (API) e *Web Services***. São Paulo: Saraiva, 2021.

JSON. **Introdução ao JSON**. Disponível em: <https://www.json.org/json-pt.html>. Acesso em: 30 abr. 2024.

POSTMAN. **What is Postman?** Disponível em: <https://www.postman.com/product/what-is-postman/>. Acesso em: 30 abr. 2024.