# N_PROG SIST II_A5 - Texto de apoio

Site: EAD Mackenzie
Tema: PROGRAMAÇÃO DE SISTEMAS II {TURMA 03B} 2023/1
Livro: N_PROG SIST II_A5 - Texto de apoio

Impresso por: FELIPE BALDIM GUERRA .
Data: sexta, 5 mai 2023, 00:45

# Descrição

# Índice

# 1. INTRODUÇÃO AOS WEB SERVICES

**Evolução no desenvolvimento de aplicações Web**

Quando a Web surgiu, seu objetivo era a troca de conteúdos por meio de, principalmente, páginas HTML estáticas. Eram arquivos escritos no formato HTML e disponibilizados em servidores para serem acessados nos navegadores. Além disso, também eram disponibilizados imagens, animações e outros conteúdos.

Mas logo se viu que a Web tinha um enorme potencial de comunicação e interação além da exibição de simples conteúdos. No entanto, para atingir esse novo objetivo, páginas estáticas não seriam suficientes. Era preciso produzir páginas HTML geradas dinamicamente baseadas nas requisições dos usuários.

Hoje, boa parte do que se acessa na Web (portais, blogs, home bankings etc.) é baseado em conteúdo dinâmico. O usuário requisita algo ao servidor que, por sua vez, processa essa requisição e devolve uma resposta nova para o usuário.

Uma das primeiras ideias para esses "geradores dinâmicos" de páginas HTML foi fazer o servidor **Web invocar um outro programa externo em cada requisição para gerar o HTML de resposta**. Era o famoso **CGI** (*Common Gateway Interface*, ou **Interface Comum de Ligação**) que permitia escrever pequenos programas para apresentar páginas dinâmicas. Para escrever os programas, eram usadas as linguagens **Perl**, **PHP**, **ASP** e até **C** ou **C++**, mas essa tecnologia apresentava problemas de portabilidade e escalabilidade, além de mesclar as regras de negócio com a visualização. Vale salientar que um servidor que usa esse tipo de tecnologia pode ter seu desempenho comprometido, uma vez que **cada solicitação** recebida de uma **CGI** requer a **criação de um novo processo**.

Em seguida, vieram os **Servlets** desenvolvidos na plataforma Java. Os **Servlets** surgiram no ano de 1997, e o nome "**Servlet**" vem da ideia de um pequeno servidor ("**servidorzinho**", em inglês) cujo objetivo é receber chamadas, processá-las e devolver uma resposta ao cliente. Uma grande vantagem do uso de **Servlet**, em relação à tecnologia **CGI**, é que o **Servlet** pode aceitar diversas requisições ao mesmo tempo em um único servidor, pois o processamento de uma requisição é executado como uma *thread* (linha de execução) do processo do servidor. Por isso, ele é mais rápido que um programa **CGI** que não permitia essa ação.

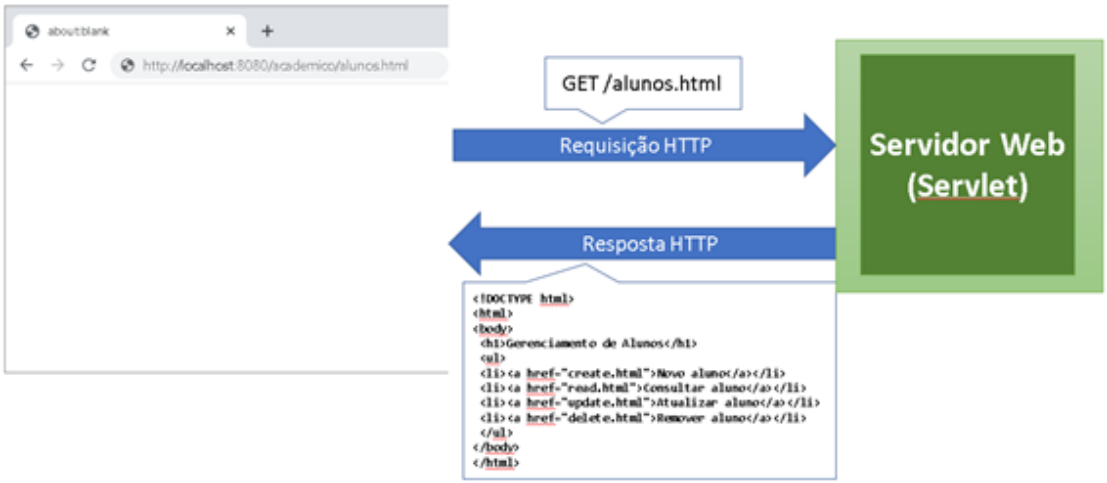
Os **Servlets** usam o **protocolo HTTP** (*Hyper Text Transport Protocol*) para realizar a comunicação entre os **clientes** (*browsers*) e o **servidor** (**Servlet**). O **Servlet** recebe uma requisição (**request**) e produz algo, uma resposta (**response**), como uma página HTML dinamicamente gerada. Como exemplo, considere que temos um **Servlet** no endereço de Internet <http://localhost:8080/academico/alunos.html> e gostaríamos de

consultar os alunos armazenados em um sistema acadêmico. As figuras abaixo descrevem como seriam os passos para a comunicação entre um **cliente** e o **Servlet**.

**Figura 1 – Cliente (browser) faz uma requisição (request) HTTP ao servidor Web (Servlet)**


Fonte: Elaborada pelo autor.

**Figura 2 – O Servlet responsável trata da requisição e produz a resposta (response) para o cliente, ou seja, uma página HTML, de forma dinâmica**



Fonte: Elaborada pelo autor.

**Figura 3 – O cliente (browser) recebe a página HTML gerada e a exibe**


Fonte: Elaborada pelo autor.

No exemplo apresentado, o **browser** fez uma requisição para um **servidor Web** localizado em localhost:8080/. Dizemos, então, que acessamos o **servidor Web** por meio de uma **URL** (*Uniform Resource Locator*, ou **Localizador de Recurso Uniforme**) e, como o próprio nome já diz, se refere ao local, ou host (servidor), que você quer acessar de determinado recurso. O objetivo da **URL** é associar um endereço remoto (endereço IP) com um nome de recurso na Internet, podendo ser imagens, páginas, arquivos e vídeos. A **URL** é parte da **URI** (*Uniform Resource Identifier*, ou **Identificador de Recursos Universal**). A **URI** une o **Protocolo** (http://) à localização do recurso (**URL** – localhost:8080/) e ao nome do recurso (academico/alunos.html).

Como dito antes, os **Servlets** são a principal tecnologia para o desenvolvimento de páginas dinâmicas na plataforma Java, mas, se tivéssemos em uma empresa, ou em empresas diferentes, sistemas que não são todos desenvolvidos em Java, como poderíamos fazer esses sistemas trabalharem juntos?

Para ilustrar o problema, imagine um cenário no qual **duas empresas** precisam trocar dados entre seus sistemas. Uma delas é uma **livraria** e a outra é uma **transportadora**. Um cliente da livraria entra no site da livraria e quer saber quanto fica para comprar e receber um livro em casa.

Os analistas do site da livraria acharam que seria melhor não dispor de um banco com os valores do transporte, acharam que atualizá-lo daria muito trabalho e necessitaria de muita atenção, então preferiram, sempre que fossem fazer a entrega de um livro, consultar o sistema da empresa que faz o transporte.

Neste cenário, quando um cliente fizer uma compra, o site da livraria envia uma mensagem com os dados de destino e outros atributos sobre a entrega a ser feita para um servidor da transportadora e o servidor da transportadora responde com o preço do frete.

Assim, tanto faz qual linguagem foi desenvolvida nos sistemas da livraria e da transportadora, os dois sistemas terão que se entender e devem usar um mesmo protocolo na troca de informação, ou seja, o protocolo HTTP. Nesse ponto é que o protocolo HTTP se mostra falho, pois com o HTTP não é possível definir, de forma clara e detalhada, um padrão para especificar o formato das informações que seriam trocadas entre os dois servidores, ou seja, se faz necessário um formato de troca de informações para que ambos os sistemas se entendam.

Diversos padrões foram propostos, mas nenhum obteve êxito suficiente, tanto pela independência de plataforma como pelo modelo proposto. Por exemplo, a utilização de **RPC** (*Remote Procedure Call*) representa tanto um problema de compatibilidade como de segurança, pois o *Firewall* e os servidores *proxy* normalmente bloquearão esse tipo de tráfego.

A solução foi criar uma padronização para especificar a comunicação de transferência de dados. Essa padronização foi desenvolvida pela **W3C** (*World Wide Web Consortium*), um consórcio de empresas de tecnologia que tem como o objetivo criar padrões comuns para conteúdo da Web, apoiada por grandes empresas como a **Microsoft**, **Samsung**, **Apple**, **Google**, **Mozilla** entre outras. A padronização foi proposta em 1999 para ser usada no desenvolvimento de **serviços na Web** (**Web Services**) e define um protocolo para a transmissão de dados remotos usando o formato **XML** (**eXtensible Markup Language**), esse protocolo recebeu o nome de **SOAP** (**Simple Object Access Protocol**).

O protocolo SOAP define o formato de mensagens de requisições (**Request**) e de mensagens de resposta (**Response**) e tem a seguinte aparência:

**Figura 4 – SOAP Request**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Header/>
  <S:Body>
    <ns2:print xmlns:ns2="http://hello.me.org/">
    <nome>Fulano</nome>
  </ns2:print>
  </S:Body>

</S:Envelope>
```

Fonte: Elaborada pelo autor.

**Figura 5 – SOAP Response**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <ns2:printResponse xmlns:ns2="http://hello.me.org/">
      <return>Ola, Fulano</return>
    </ns2:printResponse>
  </S:Body>
</S:Envelope>
```

Fonte: Elaborada pelo autor.

A partir do protocolo SOAP, surgiram os primeiros Web Services para a comunicação entre as diferentes plataformas. É importante entender que um **Web Service não é uma tecnologia ou uma biblioteca, mas sim um tipo de arquitetura de desenvolvimento cujo objetivo é a troca de informações entre duas entidades de software por meio da Internet**, utilizando os protocolos de comunicação disponíveis.

As entidades de software envolvidas nesse processo de comunicação são aplicações consideradas como **servidor** quando seu objetivo é oferecer um serviço ou mecanismo de acesso a um conjunto de dados. Já as aplicações consideradas como **cliente** têm como objetivo consumir um ou mais serviços disponibilizados pelos servidores.

Para desenvolver Web Services utilizando a linguagem Java existem duas soluções que, na documentação da Oracle, são denominadas como **Web Services SOAP** e **RESTful**. Nesta aula, não detalharemos a implementação e o funcionamento dos Web Services SOAP, mas é importante você saber que a comunicação nesse tipo de Web Service ocorre com o envio de mensagens no formato XML, seguindo as regras do protocolo SOAP.

# 2. WEB SERVICES RESTFUL

Nos anos 2000, Roy Fielding, um dos principais criadores do protocolo HTTP, apresentou em sua tese de doutorado o padrão **REST** (*Representational State Transfer*, ou **Transferência Representacional de Estado**) que foi adotado como o **modelo a ser utilizado na evolução da arquitetura do protocolo HTTP**. Muitos desenvolvedores perceberam que também poderiam utilizar o modelo REST para a implementação de Web Services, com o objetivo de se integrar aplicações pela Web, e passaram a utilizá-lo como uma alternativa aos Web Services SOAP.

Assim, surgiu os **Web Services RESTful**, que são mais leves em relação aos Web Services SOAP, o que significa que podem ser desenvolvidos com menor esforço, tornando-os mais fáceis de serem adotados como parte da implementação de um sistema. Não há **limite superior no número de clientes** que podem ser atendidos por um **único servidor**, podendo chegar ao número de milhões de clientes atendidos por um servidor. Essa escalabilidade só é possível de ser alcançada porque os **Web Service RESTful** utilizam um princípio chamado **Stateless** (sem estado), no qual o **servidor** não precisa saber em qual estado o **cliente** está e vice-versa. Dessa forma, nenhum registro das interações anteriores é salva e cada interação é tratada com base nas informações disponíveis na própria interação.

Para que as interações aconteçam, o **servidor** precisa que um **cliente** faça uma requisição de um **recurso,** solicitando que o **servidor** envie ou modifique seus dados. Um **recurso** é uma abstração sobre o determinado tipo de informação que um sistema gerencia.

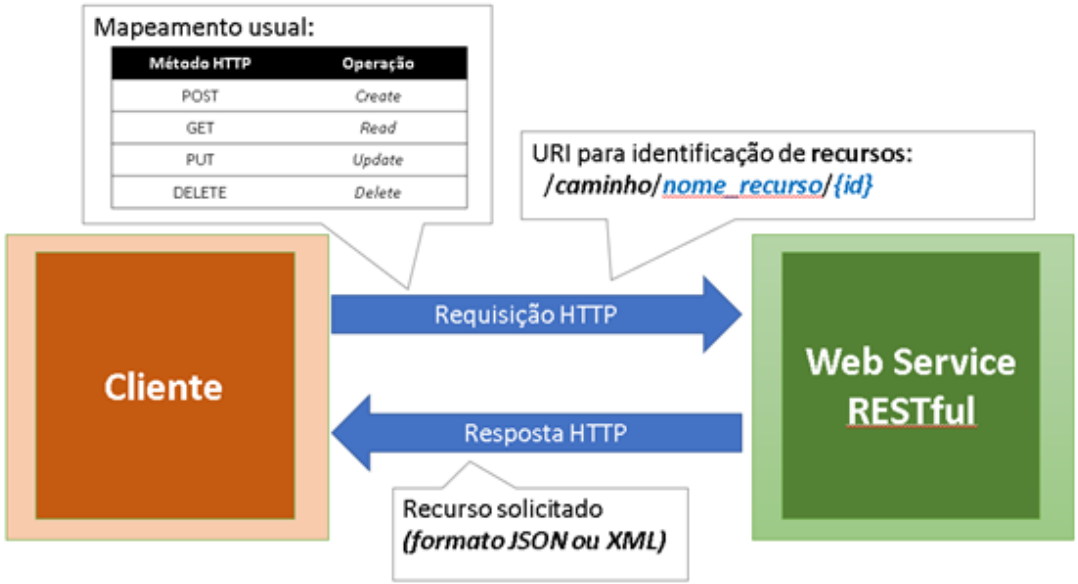
Para exemplificar o que é um **recurso**, imagine o exemplo de um Web Service RESTful de um sistema acadêmico que gerencia os dados de **alunos**, **professores**, **disciplinas** etc. Essas entidades que o sistema gerencia são chamadas de **recursos** no modelo REST. Quando chega uma requisição ao Web Service, ele precisa identificar qual dos recursos deve ser manipulado, por isso os recursos devem possuir uma identificação única. Uma boa prática para definir uma URI é utilizar nomes legíveis por humanos, que sejam de fácil dedução e que estejam relacionados ao domínio da aplicação. Isso facilita a vida dos clientes que utilizarão o serviço, além de reduzir a necessidade de documentações extensas. No exemplo do Web Service do sistema acadêmico poderíamos ter as seguintes URI para os recursos alunos, professores e disciplinas:

- http://localhost:8080/academico/alunos;
- http://localhost:8080/academico/professores;
- http://localhost:8080/academico/alunos/333;
- http://localhost:8080/academico/disciplinas.

Os recursos gerenciados pelo Web Service podem ser manipulados de diversas maneiras: é possível criá-los, atualizá-los, excluí-los, dentre outras operações. Quando um cliente faz uma requisição HTTP para um servidor, além da URI que identifica qual recurso ele pretende manipular, é necessário que ele também informe o tipo de manipulação que deseja realizar no recurso. Isso é feito utilizando os métodos (verbos) do protocolo HTTP, os métodos mais utilizados do HTTP são:

- **GET**: obter os dados de um recurso.
- **POST**: criar um novo recurso.
- **PUT**: substituir os dados de um determinado recurso.
- **DELETE**: excluir um determinado recurso.

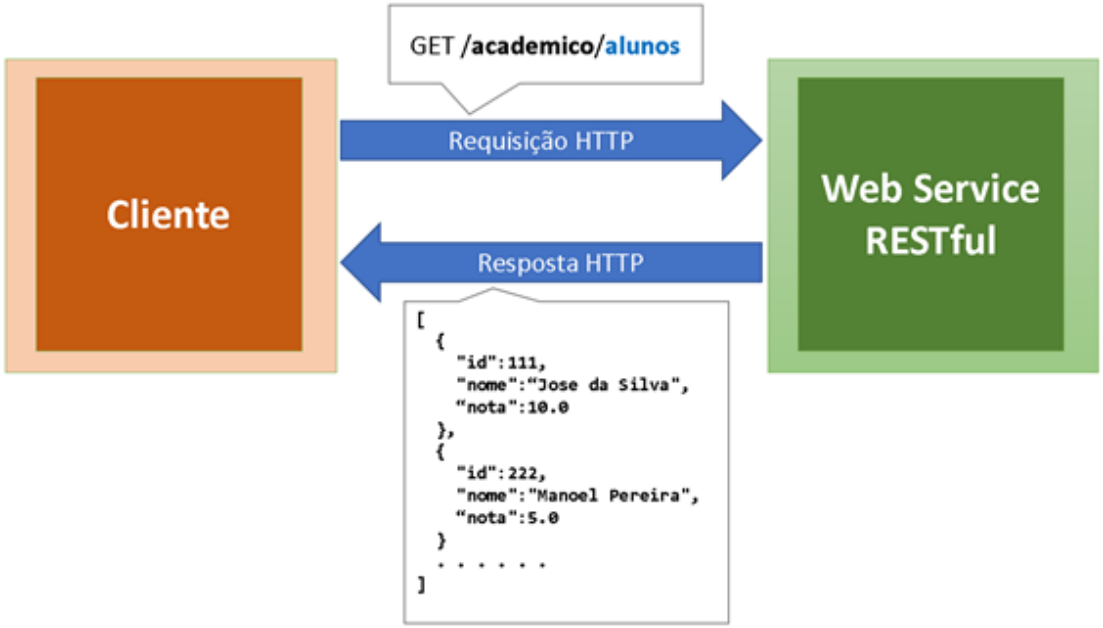
**Figura 6 – Comunicação entre um cliente e um Web Service RESTful**




Fonte: Elaborada pelo autor.

Na figura acima, é ilustrado todo o processo de comunicação entre um **cliente** e um **Web Service RESTful**. Note que os recursos ficam armazenados no **servidor** que os gerencia, e quando são solicitados pelas aplicações **clientes** (por exemplo, em uma requisição do tipo **GET**) não "abandonam" o servidor, como se tivessem sido transferidos para os **clientes**. Na verdade, o que é transferido para a aplicação **cliente** é apenas uma representação do recurso. Um recurso pode ser representado de diversas maneiras, utilizando-se formatos específicos, tais como XML, JSON, HTML e até texto puro, entre outros. A figura abaixo ilustra como seria a comunicação quando um **cliente** faz uma requisição do recurso alunos (http://localhost:8080/academico/alunos) ao WebService RESTful que gerencia as informações de um sistema acadêmico:

**Figura 7 – Requisição do recurso alunos ao WebService RESTful**

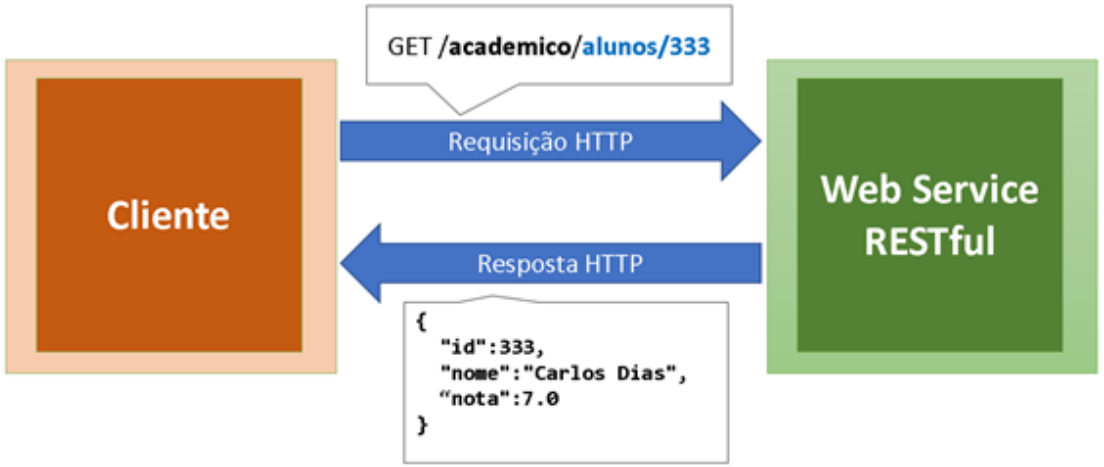



Fonte: Elaborada pelo autor.

Na requisição acima, é utilizado o **método GET** do protocolo HTPP para recuperar todos os alunos armazenados no sistema acadêmico. Em seguida, temos a resposta da requisição devolvida no formato **JSON** para a representação dos recursos obtidos. O **JSON** (*JavaScript Object Notation*) é um formato para transmissão de informações no formato texto. O Web Service RESTful também poderia responder à requisição **XML**, mas **JSON** é mais simples que o **XML** e tem sido bastante utilizado por aplicações Web devido a sua capacidade de estruturar informações de

uma forma mais compacta do que a obtida pelo modelo XML, tornando mais rápido o *parsing* (análise) dessas informações. Se quiséssemos recuperar somente um aluno, podemos acessar o recurso aluno da seguinte forma: http://localhost:8080/academico/alunos/333. Assim, teríamos a seguinte resposta do Web Service:

**Figura 8 – Resposta do Web Service**




Fonte: Elaborada pelo autor.

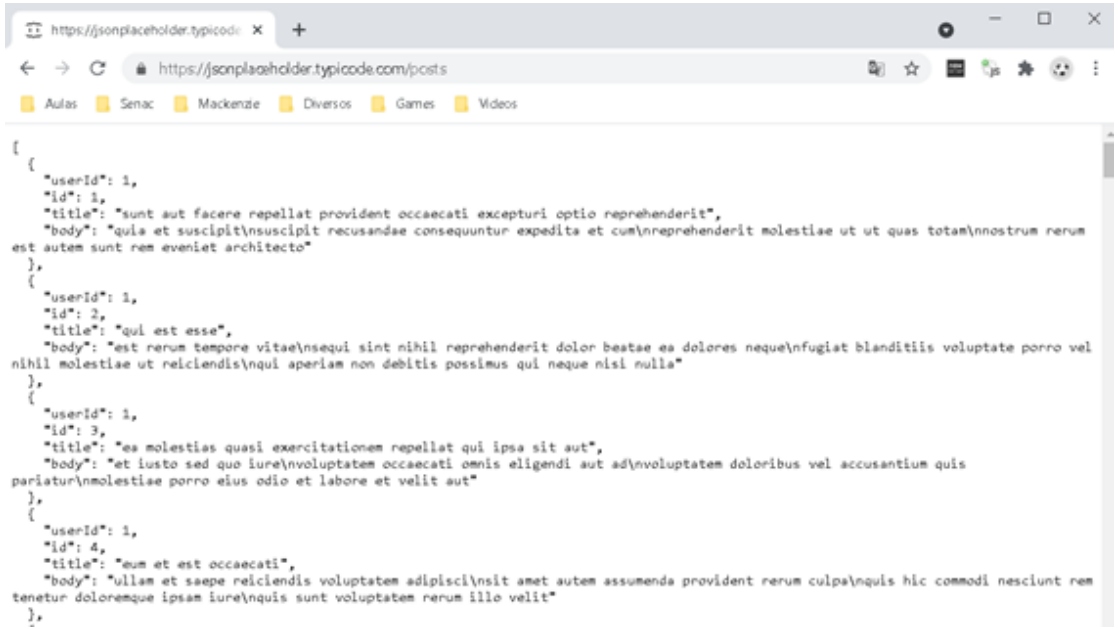
**Testando um Web Service RESTful**

Entendidos os conceitos envolvidos no desenvolvimento de um Web Service RESTful, podemos agora usar o site **JSONPlaceholder** para testar as requisições de recursos a um Web Service. O site possui um Web Service RESTful implementado com uma base de dados "fake" para teste com vários recursos disponíveis com as seguintes URIs. Abaixo, alguns exemplos de recursos no site:

- Postagens: https://jsonplaceholder.typicode.com/posts.
- Comentários: https://jsonplaceholder.typicode.com/comments.
- Usuários: https://jsonplaceholder.typicode.com/users.

Assim, se quiséssemos recuperar a lista de todas as postagens (**post**) do Web Service, basta usar a URI <https://jsonplaceholder.typicode.com/posts> na barra de endereços do browser. Em seguida, o **browser** faz uma requisição ao Web Service, utilizando o método **GET** do protocolo **HTTP**. Depois, temos a resposta da requisição, utilizando o formato **JSON** para a representação dos recursos obtidos. Veja a janela a seguir:

**Figura 9 – Resposta da requisição**




Fonte: Elaborada pelo autor.

Na próxima aula, você verá como podemos implementar nosso próprio Web Service RESTful utilizando a linguagem Java com a IDE **NetBeans 12.0 LTS** e o servidor de aplicação **GlassFish 5.1.0**.