Universidade São Judas Tadeu

Sistemas Distribuídos e Mobile

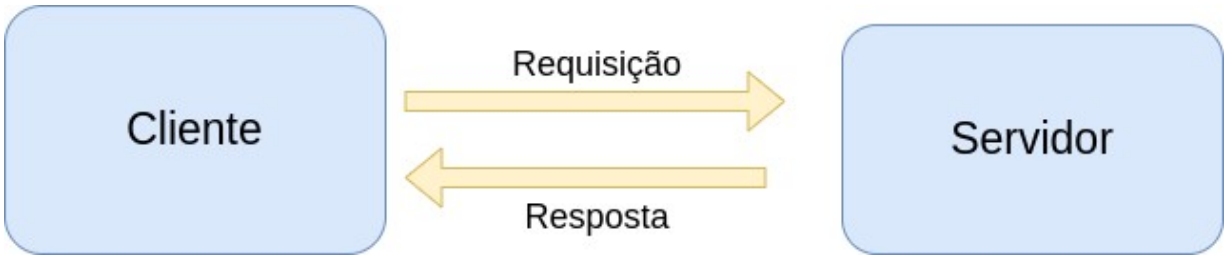
1 Introdução

O protocolo HTTP é um dos protocolos da camada de aplicação mais utilizados. É muito comum que a troca de mensagens entre clientes e servidores implementados com as mais diferentes tecnologias e linguagens se dê por meio de seu uso. Neste material iremos estudar as suas principais características.

2 Passo a passo

2.1 (Arquitetura cliente/servidor) A ideia geral da arquitetura cliente/servidor é exibida na Figura 2.1.1. Ela é utilizada para a comunicação entre processos.

Figura 2.1.1



- O modelo é elementar mas fundamental para que possamos entender o funcionamento dos protocolos. Em particular, é preciso definir o que é **cliente** e o que é **servidor**. Neste contexto de estudo, **ambos são processos**. Ou seja, programas em execução. Ocorre que um mesmo computador pode exercer o papel de cliente e servidor por executar programas que têm essas características, inclusive simultaneamente. Se em um determinado instante um computador está executando um aplicativo cliente e outro servidor, o computador propriamente dito é cliente ou servidor? Entendeu a razão pela qual não faz sentido definir servidor como hardware neste contexto?

Nota: É claro que em outros contextos de estudo pode ser conveniente utilizar definições diferentes para o que é um servidor. Por exemplo, é comum definir um servidor como um computador de alto poder computacional que executa diversos aplicativos de interesse para pessoas que o acessam remotamente. Essa definição pode ser apropriada para alguns cenários.

(Protocolo HTTP): Em geral, protocolos são especificados por meio de RFCs (Request For Comment). O protocolo HTTP é um dois mais utizados nos dias atuais. A RFC que especifica a versão 1.1 do protocolo HTTP pode ser encontrada no Link 2.1.1.

Link 2.1.1
https://tools.ietf.org/html/rfc2616

Note que, em geral, uma RFC torna obsoletas RFCs mais antigas, atualiza RFCs e pode ser atualizada ou tornada obsoleta.

2.2 (**Métodos HTTP**) Quando um processo cliente deseja interagir com um processo servidor, ele precisa especificar o tipo de operação que deseja realizar por meio da especificação de um "**Método**". Os métodos HTTP disponíveis são

- **OPTIONS**: Representa uma requisição por informações sobre as "opções" que o servidor disponibiliza para um determinado recurso. É comum que o navegador envie uma requisição dessas quando o desenvolvedor tenta utilizar requisições assíncronas como Ajax, por exemplo. Caso a aplicação cliente esteja sendo executada em um host diferente daquele em que o servidor está executando – hosts são caracterizados pela tripla protocolo, endereço e porta – é possível que a requisição seja bloqueada pelo mecanismo CORS (Cross Origin Resource Sharing).

- **GET**: Serve para obter informações. Quando o cliente deseja obter recursos como figuras, páginas HTML etc do servidor, ele utiliza esse método. Em geral, requisições que utilizam GET como método não devem causar efeitos colaterais (inserção de dados em uma base, por exemplo) no servidor, embora tecnicamente isso seja possível.

- **HEAD**: Similar ao GET. Em geral, ele é utilizado para obter meta informações sobre um determinado recurso. Por exemplo, o cliente pode querer perguntar se uma página HTML está disponível no momento, quando ela foi modificada pela última vez etc. Ou seja, informações sobre o recurso. O método HEAD retorna somente as meta informações sobre o recurso consultado. Ele não retorna o recurso propriamente dito.

- **POST**: Quando o cliente deseja fazer algum tipo de inserção de dados no servidor, ele envia uma requisição com método POST.

- **PUT**: Em geral, o cliente utiliza o método PUT quando deseja atualizar algum recurso no servidor (fazer um UPDATE em uma base relacional, por exemplo).

- **DELETE**: Como o nome sugere, o método DELETE do protocolo HTTP deve ser utilizado quando o cliente deseja apagar algum recurso no servidor.

- **TRACE**: O cliente utiliza esse método para simplesmente receber de volta o que tiver enviado para o servidor. Em geral, usado para testes e diagnósticos.

- **CONNECT**: Pode ser utilizado para acessar sites que utilizam SSL, por exemplo. Neste caso, o cliente pede ao proxy HTTP que faça um “túnel” para a conexão TCP.

- **PATCH**: Utilizado para aplicar alterações parciais em um determinado recurso.

2.3 (Códigos de status HTTP) Uma vez que um servidor receba uma requisição de um cliente, ele deve eventualmente responder a ela. Cada resposta tem um código associado que permite ao cliente descobrir o que aconteceu com sua requisição (se o que ele pediu funcionou, se o recurso solicitado não existe mais etc). Os códigos de status são

- **(Classe 1xx)** Códigos desta classe representam respostas provisórias.

- **100 Continue**: Uma resposta intermediária indicando que o cliente pode continuar com a sua requisição pois ela ainda não foi rejeitada pelo servidor.

- **101 Switching Protocols**: O servidor entrega essa resposta ao cliente indicando que ele está de acordo com a proposta de troca de protocolo. Por exemplo, um cliente pode enviar uma requisição inicial solicitando por um recurso que deve ser entregue em tempo real. Na requisição, ele inclui um pedido para que o servidor troque o protocolo. Caso o servidor possua o protocolo sugerido implementado, esse é o código de status enviado para o cliente.

- **(Classe 2xx)** Códigos desta classe indicam que a requisição feita pelo cliente foi recebida, entendida e aceita.

- **200 OK**: A requisição foi atendida com sucesso. Trata-se de um código de sucesso genérico.

- **201 Created**: A requisição foi atendida com sucesso e isso teve como resultado a criação de um recurso.

- **202 Accepted**: A requisição foi aceita para algum tipo de processamento, porém, esse processamento ainda não terminou. Por exemplo, o cliente pode solicitar que o servidor execute uma sequência de tarefas que somente é executada uma vez por dia. O servidor entrega esse código de status para o cliente indicando que ele entendeu o pedido e que, na hora certa, o processamento irá acontecer.

- **203 Non-Authoritative Information**: A meta informação devolvida não é a original e definitiva do servidor. Ela pode ter sido obtida de uma cópia local ou de terceiros. Se

informações locais sobre o recurso envolvido na requisição forem adicionadas, por exemplo, isso dá origem a um superconjunto da informação originalmente conhecida.

- **204 No Content**: A requisição foi atendida com sucesso e o servidor deseja informar ao cliente que não há nenhuma informação a ser entregue a ele.

- **205 Reset Content:** A requisição foi atendida com sucesso e o cliente deve redefinir o documento que causou a requisição. Por exemplo, um form com dados preenchidos por um usuário foi submetido ao servidor e esse, por sua vez, deseja informar ao cliente que o form por ser redefinido.

- **206 Partial Content**: O servidor devolve uma resposta com esse código de status quando a requisição feita pelo cliente indicava que ela era parcial. Por exemplo, o cliente pode ter especificado que ele desejava aplicar algum tipo de operação sobre elementos de uma coleção, porém somente até um limite especificado.

- **(Classe 3xx)** Quando o servidor devolve algum código de status dessa classe ele quer dizer ao cliente que ele deve realizar mais ações para que a sua requisição possa ser atendida por completo. Os principais códigos são:

- **301 Moved Permanently**: Quando o cliente faz uma requisição por um recurso que tem um novo e permanente identificador.

- **302 Found**: O recurso solicitado está, temporariamente, associado a um identificador diferente. Em geral, o servidor pode devolver o identificador atual para o cliente também.

- **Classe (4xx)**: Códigos pertencentes a essa classe indicam que o cliente fez uma requisição que resultou em algum tipo de erro. Os principais códigos são:

- **400 Bad Request:** A requisição não pôde ser entendida pelo servidor pois ela possui algum erro de sintaxe.

- **401 Unauthorized**: O recurso solicitado requer autenticação por parte do cliente.

- **403 Forbidden**: A requisição foi aceita mas o servidor se recusa a atendê-la, por algum motivo diferente de falta de autenticação. A requisição não deve ser realizada novamente pelo cliente.

- **404 Not Found**: O recurso solicitado pelo cliente não pôde ser encontrado.

- **405 Method Not Allowed**: O cliente solicitou algum recurso utilizando um método HTTP para o qual o servidor não dá suporte (para aquele recurso).

- **(Classe 5xx)** Essa classe possui códigos de status que o servidor utiliza para informar que detectou um erro internou e que é incapaz de atender à requisição. Os principais são:

- **500 Internal Server Error** Código genérico. Indica somente que o servidor encontrou um problema interno e não pode atender à requisição. Não inclui mais detalhes sobre o erro.

- **505 HTTP Version Not Supported** O servidor não tem suporte à versão do protocolo HTTP especificada pelo cliente em sua requisição.

2.4 (Cabeçalhos) Requisições e respostas podem incluir diferentes cabeçalhos que, em geral, contém informações sobre os recursos sendo solicitados e entregues. Vejamos os principais:

- **Allow** O servidor pode usar esse cabeçalho para informar ao cliente quais métodos HTTP ele pode usar.
Por exemplo: **Allow: GET, HEAD, PUT.**

- **Content-Language**: Descreve o idioma principal do recurso envolvido.

- **Content-Length:** Indica o tamanho do corpo da requisição ou da resposta em número de bytes.

Por exemplo: **Content-Length**: 3495

- **Content-Type**: Usado para indicar o tipo do recurso envolvido.
Por exemplo: **Content-Type**: **text/html; charset=ISO-8859-4**

Valores comuns que podem ser associados a Content-Type são:

- **image/bmp**
- **text/css**
- **text/html**
- **text/javascript**
- **application/json**

Veja uma lista com mais exemplos no Link 2.4.1.

Link 2.4.1
https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types/Common_types

2.5 (Testes com o Postman) O Postman é um aplicativo que pode ser utilizado para enviar requisições HTTP. Ele pode ser obtido no Link 2.5.1.

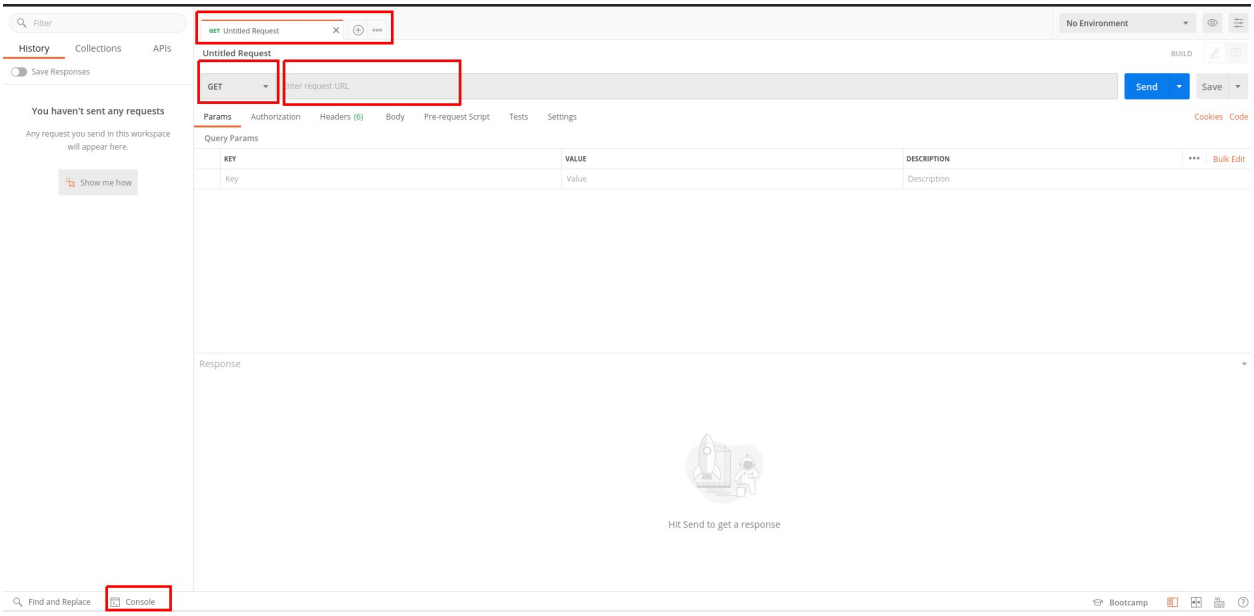
Link 2.5.1
https://www.postman.com/

- Na tela inicial do Postman, repare que você pode, entre outras coisas

- criar uma nova aba para fazer requisições
- escolher o método HTTP a ser utilizado na requisição
- digitar a URL para a qual a requisição será enviada
- abrir o console para ver a requisição completa gerada pelo Postman

Veja a Figura 2.5.1.

Figura 2.5.1



- Clique em console. A seguir, faça as seguintes consultas e verifique a resposta obtida bem como o texto gerado no console.

GET www.google.com.br

POST www.google.com.br

GET www.google.com.br/temAlgoAqui

GET www.google.com.br/search?q=teste

GET https://www.google.com/images/branding/googlelogo/1x/googlelogo_color_272x92dp.png

2.6 (Implementando um endpoint com NodeJS e Express) Nesta seção, iremos implementar uma aplicação simples que desempenhará o papel “servidor”. A seguir, faremos requisições usando o Postman.

- Há diferentes opções para a implementação de aplicações Web. Neste exemplo, usaremos o NodeJS. Ele pode ser obtido a partir do Link 2.6.1.

Link 2.6.1
https://nodejs.org

Nota: A instalação do Node acompanha o NPM (Node Package Manager) que é um aplicativo que nos permite fazer o download de pacotes para aplicações Node.

- Comece criando um diretório para abrigar os seus projetos. Chame ele de algo como **workspace**, **desenvolvimento** ou algo parecido.

- A seguir, abra um terminal (um prompt de comando) e navegue até o diretório que você criou com

cd diretorio

- Crie um novo diretório para abrigar este projeto. Ele pode se chamar algo como **teste-http**.

- Para criar um projeto Node, use

npm init -y

A opção -y indica que você aceita (dizendo **y**es) os valores padrão para os campos que serão criados, que incluem a versão, o nome do autor, uma descrição etc.

- Em seguida podemos instalar o Express, que é um framework para Node que simplifica a manipulação de requisições HTTP. Isso pode ser feito com

npm install --save express

Nota: A opção --save do npm indica que, além de baixar a dependência e armazenar os arquivos na pasta chamada **node_modules**, ela precisa ficar registrada textualmente no arquivo **package.json** (ele é análogo ao arquivo pom.xml do Maven). Assim, caso seu projeto seja compartilhado, as suas dependências estão registradas e podem ser obtidas facilmente com um **npm install**. Entretanto, desde a versão 5 do npm este é o comportamento padrão e a opção --save já não é necessária.

- Implementaremos nosso servidor de teste com Javascript. Para isso, crie um arquivo chamado **server.js**. Seu conteúdo inicial é dado na Listagem 2.6.1.

Listagem 2.6.1

```
const http = require('http');
const express = require('express');

const app = express();
const porta = 3000;
app.set('port', porta);
const server = http.createServer(app);
server.listen(3000);
```

- Você pode executar o servidor com

node server.js

- Para especificar um recurso que pode ser acessado por um cliente, utilizados métodos do objeto app. Ele define métodos que implementam os principais métodos do protocolo HTTP. Veja um teste na Listagem 2.6.2.

Listagem 2.6.2

```
const http = require('http');
const express = require('express');
const app = express();
const porta = 3000;
app.set('port', porta);
app.get("/teste", (req, res, next) => {
  res.send("Olá!");
});
const server = http.createServer(app);
server.listen(3000);
```

- Para os próximos exemplos, vamos definir uma coleção com dados de clientes. Permitiremos que o cliente execute os principais comando CRUD sobre ela. Comece com a sua definição, como mostra a Listagem 2.6.3.

Listagem 2.6.3

```
const clientes = [
  {
    id: 1,
    nome: 'Joao',
    email: 'joao@email.com'
  },
  {
    id: 2,
    nome: 'Cristina',
    email: 'cristina@email.com'
  }
]
```

- A seguir, vamos especificar um método que permite que a coleção seja obtida. Idealmente, usamos o metódo GET do protocolo HTTP. Veja a Listagem 2.6.4.

Listagem 2.6.4

```
app.get('/clientes', (req, res, next) => {
  res.json(clientes);
});
```

- A seguir, vamos especificar um método que permite a inserção de novos clientes. O cliente irá enviar os dados em formato JSON. Para simplificar a sua manipulação, vamos instalar o pacote **body-parser**. Isso pode ser feito com

npm install --save body-parser

A seguir, faça o ajuste que a Listagem 2.6.5 mostra.

Listagem 2.6.5

```
const express = require ('express');
const app = express();
const bodyParser = require ('body-parser');
app.use (bodyParser.json());
...
```

- A função que faz a inserção de um cliente recebido é exibida na Listagem 2.6.6. Note que a geração de id ficará a cargo do servidor.

Listagem 2.6.6

```
let contador = 3; // logo após as instruções require
…
app.post('/clientes', (req, res, next) => {
  const cliente = req.body;
  clientes.push({id: contador += 1, nome: cliente.nome, email: cliente.email});
  console.log(clientes);
  res.end();
})
```

- Note que também podemos devolver a coleção atualizada, além de alterar o código de status da resposta, como na Listagem 2.6.7.

Listagem 2.6.7

```
app.post('/clientes', (req, res, next) => {
  const cliente = req.body;
  clientes.push({id: contador += 1, nome: cliente.nome, email: cliente.email});
  console.log(clientes);
  res.status(201).json(clientes);
})
```

Exercícios

1. Adicione métodos para atualizar e remover clientes da base. Utilize métodos de requisição e códigos de resposta apropriados do protocolo HTTP.

Referências

MDN Web Docs. 2020. Disponível em <https://developer.mozilla.org/en-US/>. Acesso em agosto de 2020.

Node.js. 2020. Disponível em <https://nodejs.org>. Acesso em agosto de 2020.

Postman | The Collaboration Platform for API Development. 2020. Disponível em <https://www.postman.com/>. Acesso em agosto de 2020.

RFC 2616 - Hypertext Transfer Protocol – HTTP/1.1. 2020. Disponível em <https://tools.ietf.org/html/rfc2616>. Acesso em agosto de 2020.