

**UNIVERSIDADE REGIONAL INTEGRADA DO ALTO URUGUAI E DAS MISSÕES**  
**CAMPUS DE ERECHIM**  
**DEPARTAMENTO DE ENGENHARIAS E CIÊNCIA DA COMPUTAÇÃO**  
**CURSO DE CIÊNCIA DA COMPUTAÇÃO**

**KELWIN KOMKA**  
**JOÃO VITOR VERONESE VIEIRA**  
**VINICIUS EMANOEL ANDRADE**

**TRABALHO FINAL:**  
**REDES DE COMPUTADORES I E SISTEMAS OPERACIONAIS II**

**ERECHIM - RS**  
**2018**

## LISTA DE ILUSTRAÇÕES

Figura 1 – Histórico e versões do protocolo HTTP . . . . .	4
Figura 2 – Representação da busca de uma página com HTTP . . . . .	6
Figura 3 – Métodos disponíveis para uma requisição HTTP . . . . .	8
Figura 4 – Interconexão entre dispositivos com TCP/IP . . . . .	11
Figura 5 – Distribuição de arquiteturas . . . . .	11
Figura 6 – Camadas de enlace e física: componentes . . . . .	13
Figura 7 – Interligação de redes distintas: roteadores . . . . .	14
Figura 8 – Sessão SMTP: funcionamento . . . . .	16
Figura 9 – Exemplo de tratamento para <i>URL</i> acessada pelo usuário . . . . .	25
Figura 10 – <i>URL</i> acessada pelo usuário . . . . .	25
Figura 11 – Método <i>index</i> do arquivo <b>DashboardController.php</b> . . . . .	26
Figura 12 – Arquivo responsável pelo conteúdo da tela <i>Dashboard</i> . . . . .	27
Figura 13 – Tela resultante da interpretação do arquivo <i>dashboard.blade.php</i> . . . . .	27
Figura 14 – Passos para obter a aplicação localmente . . . . .	36
Figura 15 – Instalando dependências da aplicação . . . . .	37
Figura 16 – Editando arquivo de configuração com <i>nano</i> . . . . .	38
Figura 17 – Sequência de passos para <i>levantar</i> o servidor da aplicação . . . . .	42
Figura 18 – Tela inicial do sistema . . . . .	43
Figura 19 – Tela <i>Enviar</i> . . . . .	44
Figura 20 – Escolhendo um problema para resolver . . . . .	45
Figura 21 – Conteúdo do <i>box</i> alterado . . . . .	45
Figura 22 – Campos referentes à solução do problema . . . . .	45
Figura 23 – Template do código alterado conforme a linguagem escolhida . . . . .	46
Figura 24 – Mensagem para confirmação da submissão . . . . .	46
Figura 25 – Mensagem confirmando o envio . . . . .	47
Figura 26 – Tela <i>Submissões</i> . . . . .	47

## **LISTA DE ABREVIATURAS E SIGLAS**

ASCII	American Standard Code for Information Interchange
CSS	Cascading Style Sheets
IETF	Internet Engineering Task Force
LAN	Local Area Network
MD5	Message-Digest algorithm 5
OSI	Open System Interconnection
RAID	Redundant Array of Independent Disks
RFC	Request for Comments
SCM	Supply Chain Management
SQL	Structured Query Language
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
URL	Uniform Resource Locator

## SUMÁRIO

<b>1</b>	<b>INTRODUÇÃO . . . . .</b>	<b>1</b>
<b>2</b>	<b>BASE TEÓRICA . . . . .</b>	<b>2</b>
<b>2.1</b>	<b>Gluster . . . . .</b>	<b>2</b>
<b>2.2</b>	<b>Protocolos . . . . .</b>	<b>2</b>
2.2.1	HTTP . . . . .	3
2.2.1.1	Versões e Histórico . . . . .	4
2.2.1.2	Conceitos e funcionamento . . . . .	5
2.2.2	TCP/IP . . . . .	9
2.2.2.1	Histórico . . . . .	10
2.2.2.2	Conceitos e funcionamento . . . . .	10
2.2.2.2.1	Camada de enlace ou acesso à rede . . . . .	12
2.2.2.2.2	Camada de rede ou internet . . . . .	13
2.2.2.2.3	Camada de transporte . . . . .	14
2.2.2.2.4	Camada de aplicação . . . . .	15
2.2.3	Protocolo frontend/backend do postgresql . . . . .	16
2.2.3.1	Prefácio . . . . .	16
2.2.3.2	Mensagens . . . . .	17
2.2.3.2.1	Servidor . . . . .	17
2.2.3.2.2	Fluxo de mensagens . . . . .	17
2.2.3.3	<i>Unix Sockets</i> . . . . .	22
2.2.3.4	<i>TCP/IP vs Unix sockets</i> . . . . .	22
2.2.3.4.1	Características . . . . .	22
2.2.3.4.2	Implementação utilizando <i>python</i> . . . . .	23
<b>3</b>	<b>DOCUMENTAÇÃO DAS APLICAÇÕES DESENVOLVIDAS . . . . .</b>	<b>24</b>
<b>3.1</b>	<b><i>Frontend</i> . . . . .</b>	<b>24</b>
3.1.1	Visão Geral . . . . .	24
3.1.2	Fluxo de Trabalho . . . . .	24
3.1.3	Fechamento . . . . .	28
<b>3.2</b>	<b><i>Backend</i> . . . . .</b>	<b>28</b>
3.2.1	Visão Geral . . . . .	28
3.2.2	Fluxo de Trabalho . . . . .	28
3.2.2.1	Compiladores / Interpretadores . . . . .	28
3.2.2.2	<i>Scripts</i> . . . . .	29
3.2.2.3	Retornos de execução e compilação . . . . .	30

3.2.2.4	<i>Threads</i> . . . . .	30
3.2.2.5	Banco de dados . . . . .	30
3.2.3	Fechamento . . . . .	31
<b>4</b>	<b>REPLICAÇÃO DO SISTEMA</b> . . . . .	<b>32</b>
<b>4.1</b>	<b>GlusterFS</b> . . . . .	<b>32</b>
4.1.1	Instalando o sistema operacional . . . . .	32
4.1.2	Configurando os servidores . . . . .	32
4.1.3	Configurando os clientes . . . . .	34
<b>4.2</b>	<b>Frontend</b> . . . . .	<b>35</b>
<b>4.3</b>	<b>Backend</b> . . . . .	<b>38</b>
4.3.1	Prefácio . . . . .	38
4.3.2	Recomendações . . . . .	39
4.3.2.1	Distribuição . . . . .	39
4.3.3	Instalando os compiladores . . . . .	39
4.3.3.1	<i>G++</i> . . . . .	39
4.3.3.2	<i>Python</i> . . . . .	39
4.3.3.3	<i>Java</i> . . . . .	40
4.3.3.4	<i>Projeto</i> . . . . .	40
4.3.3.5	<i>Execução</i> . . . . .	40
<b>5</b>	<b>MANUAL DE USO</b> . . . . .	<b>42</b>
<b>5.1</b>	<b>Dashboard</b> . . . . .	<b>43</b>
<b>5.2</b>	<b>Enviar</b> . . . . .	<b>44</b>
<b>5.3</b>	<b>Submissões</b> . . . . .	<b>47</b>
<b>6</b>	<b>CONCLUSÃO</b> . . . . .	<b>48</b>
	<b>REFERÊNCIAS</b> . . . . .	<b>49</b>

## 1 INTRODUÇÃO

A exponencial transição de aplicações para plataformas *web* ocorre pelas vantagens que uma ferramenta online possibilita, pois requisita muito menos de seus usuários e dá mais controle à seus gerenciadores e donos. O sistema desenvolvido gerou-se da ideia de agregar conhecimentos, junto à utilização de tecnologias atuais dentro de um contexto familiar, onde seu desenvolvimento gerará uma plataforma para aprendizado e treino da programação, um sistema de submissão baseado ao URI Online Judge, para documentação das tecnologias que funcionam por trás da plataforma e demonstração de funcionamento.

O sistema faz uso de protocolos, que se resume à um conjunto de regras, como: o HTTP que é utilizado para requisição e transmissão de páginas web, utilizado por qualquer pessoa que utilize um navegador; o TCP/IP um conjunto de protocolos que fornece todo padrão para transmissão de dados e conversas entre máquinas ligadas à uma rede, sendo um dos mais importantes, pois garante que qualquer máquina possa se comunicar; o protocolo utilizado pelo banco Postgres, que se baseia no TCP/IP e Unix Sockets e, padroniza a conversa entre as máquinas e o banco.

A transmissão de arquivos entre as máquinas que compõem o sistema, formam um cluster, onde estão interligadas pelo *Gluster*, um *software* que possibilita a criação de um sistema de arquivos distribuído, onde as máquinas terão um volume sincronizado e poderão trabalhar com arquivos compartilhados, gerando uma redundância controlada como o RAID 1.

A composição do sistema traz um *front-end web* para realização de submissões dos exercícios de programação, criando arquivos que serão sincronizados entre as máquinas pelo *Gluster*, o banco de dados será usado como uma fila de processamento e outra máquina terá o compilador que pegará o código dos arquivos, compilará e executará este código, gerando uma saída que será enviada para o usuário, lhe informando sobre o estado e resultado de sua submissão.

## 2 BASE TEÓRICA

### 2.1 Gluster

O *cluster* é definido por: um grupo de servidores e outros recursos que trabalham em conjunto, agindo como um sistema único, de alta disponibilidade, permitindo diversas possibilidades de uso, podendo também ser utilizado para balanceamento do carregamento de dados e processamento paralelo. Um sistema em *cluster* é composto de máquinas que compartilham memória e são estreitamente conectadas por uma LAN ou InfiniBand. O modo de operação de *software* que possibilitam a formação de um cluster é:

Uma camada de *software* de cluster opera sobre os nós do cluster. Cada nó pode monitorar um ou mais dos outros nós (por meio da LAN). Se a máquina monitorada falhar, a máquina de monitoramento pode apropriar-se da sua memória e reiniciar as aplicações que estavam sendo executadas na máquina que falhou. Os usuários e clientes das aplicações percebem somente uma breve interrupção do serviço. (SILBERSCHATZ; GALVIN; GAGNE, 2015)

O *GlusterFS* é um *software* que possibilita a criação de um sistema de arquivos distribuídos, que pode ser expandido na forma de bloco para armazenamento. O bloco, definido como uma unidade básica de armazenamento, é representado como um diretório em uma máquina, onde é compartilhado por uma ferramenta de armazenamento, composto por um sistema de arquivos onde pode-se exportá-lo com um ponto de montagem, sendo o volume para a máquina cliente, pelo *gluster*, por exemplo.

Segundo Rouse (2017), o *gluster* possui dois componentes: o cliente que irá montar o volume e os servidores. Os servidores são configurados para funcionarem como blocos de armazenamento. Uma versão do sistema é executada em cada servidor, onde estas exportam o sistema de arquivos local como um volume. O cliente conecta-se com os servidores através de protocolos como o TCP/IP, criando um volume virtual de dados compostos a partir de diversos servidores remotos usando tradutores, que servem para conectar vários subvolumes. Por padrão, os arquivos são guardados como um todo, como uma informação só, mas é possível particionar entre diversos volumes remotos.

O *gluster* é utilizado frequentemente em sistemas de computação em nuvem, transmissão de dados e *streaming* de mídia, pois se adequa a dados desestruturados como arquivos de áudio, vídeo e *logs*.

### 2.2 Protocolos

Todos os protocolos, sejam específicos sobre redes de computadores ou não, podem ser definidos, de forma resumida, como regras e procedimentos de comunicação que viabilizam a troca de informações entre duas pontas (ou conexões). Portanto, atendo-se aos protocolos de

rede de computadores, pode-se dizer que um protocolo é um conjunto de regras e convenções que padronizam e definem como os computadores e equipamento de informática trocam informações através de uma rede (BARRETT; KING, 2010).

Com isso, observa-se que os protocolos determinam qual a linguagem que "*será falada*" pelos equipamentos em uma rede e, em virtude disso, parte essencial de um projeto de redes, por exemplo, é a escolha dos protocolos que comporão a mesma. Para que seja possível definir quais protocolos serão necessários em uma rede, ainda segundo os autores citados anteriormente, características como velocidade, *overhead*, eficiência e roteamento devem ser levadas em consideração. Além disso, vale ressaltar que quanto maior for o nível de complexidade do protocolo, mais alta é a camada do modelo OSI em que ele atua e, consequentemente, mais amplo será o planejamento da rede. Moraes (2014) afirma que é preciso levar em conta três premissas básicas para estar apto a iniciar os estudos acerca de protocolos:

- Muitos protocolos podem trabalhar conjuntamente, o que é conhecido como pilha de protocolo (*protocol stack*).
- Alguns protocolos trabalham em mais de uma camada OSI, por exemplo, o protocolo X.25;
- A camada em que o protocolo trabalha é quem vai descrever a sua função;

Geralmente, um protocolo implementa as tarefas que são executadas em uma camada do modelo OSI. Porém, uma camada não define um único protocolo. Pelo contrário: uma camada do modelo OSI, ainda conforme Barrett e King (2010), "define uma função de comunicação de dados que pode ser realizada por qualquer quantidade de protocolos. Como cada camada define uma função ela pode conter múltiplos protocolos, cada um oferecendo um serviço adequado à função dessa camada". Levando em conta esta afirmação, é possível deduzir que, nos dias atuais, existem diversos protocolos em uso pelos equipamentos de rede.

E sim, pode-se afirmar que esta dedução está correta, pois existem inúmeros tipos diferentes de protocolos que, quando utilizados em conjunto, formam as chamadas "pilhas de protocolos" (como descrito acima). Esses diversos protocolos acabam diferenciando-se entre si de acordo com sua velocidade, utilização de recursos, facilidade de configuração, eficiência de transmissão e capacidade de tráfego entre segmentos de LAN. Em virtude dessas diferenças, cada um deles acaba possuindo vantagens e desvantagens, objetivos de utilização e capacidade de realização de tarefas.

Consequentemente, é notório que qualquer aplicação que comunique-se na rede estará utilizando uma combinação de protocolos de redes única e, em virtude disso, o objetivo desta seção do documento é detalhar cada um dos principais protocolos usufruídos pelo grupo durante o projeto.

### 2.2.1 HTTP

Conforme é definido na própria RFC do protocolo, o Hypertext Transfer Protocol (ou Protocolo de Transferência de Hipertexto) "é um protocolo à nível de aplicação com a leveza e



velocidade necessárias para distribuir, de maneira colaborativa, sistemas de informação hipermídias. É um genérico e sem-estado protocolo orientado a objetos que pode ser usado para muitas tarefas, tais como servidores de domínio e sistemas de gerenciamento de objetos distribuídos, através da extensão de seus métodos de requisição (comandos). Uma ferramenta do protocolo HTTP é a digitação da representação de dados, permitindo que os sistemas sejam construídos independentemente dos dados que estão sendo transferidos. O HTTP vem sendo usado desde a iniciativa da Rede Mundial de Computadores (world Wide Web), de 1990." (FIELDING et al., 1999)

Após incorporar a definição técnica e original, escrita pelos criadores do protocolo, este documento vai passar a utilizar uma linguagem mais amigável e, consequentemente, mais fácil de ser compreendida. Para resumir o que foi transcrito acima, segundo Vieira (2016), "o protocolo HTTP é um mecanismo utilizado pelos navegadores de internet para requisitarem informações ao servidor web e exibir as páginas na tela do dispositivo em uso.". Como já citado anteriormente, inclusive neste capítulo, um protocolo compreende um conjunto de regras que definem como algo irá funcionar e, como HTTP é um protocolo, ele define, neste caso, como a informação será transferida de um computador para o outro.

A versão mais utilizada desta norma é a HTTP/1.1, concebida em 1999. No entanto, atualmente, o esboço da próxima versão do protocolo - HTTP/2.0 - já foi publicado pela IETF e, apesar de ser uma prévia, pode-se notar consistentes e significativas mudanças na comunicação e renderização de dados entre navegadores e servidores. (VIEIRA, 2016)

#### 2.2.1.1 Versões e Histórico

Figura 1 – Histórico e versões do protocolo HTTP



Fonte: Akamai (2017)

**Versão 1: HTTP/0.9:** esta versão era caracterizada, principalmente, por ter como objetivo a simplicidade. A meta era ser tão simples que Tim Berners-Lee, criador do protocolo, sugeriu que os dados fossem transferidos no formato de texto ASCII. Além disso, o método GET era a única maneira de requisição disponível nesta versão. Essa versão inicial está intensamente conectada com o surgimento da *World Wide Web*.

**Versão 2: HTTP/1.0:** lançada 5 anos após a criação da versão 1 do protocolo, esta versão foi criada para acompanhar o desenvolvimento constante da web e modificar o cenário de ineficiência da primeira versão quando comparada a esse desenvolvimento. Nesse ponto, já era necessário que o protocolo permitisse itens a mais do que uma simples transferência de

textos e seria necessário incluir informações mais consistentes na troca de informações, tais como metadados da requisição e resposta/negociação de conteúdo, por exemplo.

**Versão 3: HTTP/1.1:** disponibilizada 3 anos após o lançamento da versão 2, a terceira versão é, até os dias atuais, a versão mais utilizada, revolucionou a comunicação através da rede, é considerada um marco histórico na área de tecnologia e definiu o padrão da Internet. Nesta versão, o principal diferencial foi a correção de algumas ambiguidades encontradas na versão 2 e, além disso, a melhoria drástica de desempenho do protocolo, fazendo uso de itens como: mecanismo adicional de cache, pipelining de solicitação, conexões vivas e codificações de transferência.

**Versão 4: HTTP/2 (ainda não lançada oficialmente):** apesar de ter sido citado anteriormente que mudanças significativas estão em construção, juntamente com a versão 4 deste protocolo, vale ressaltar que todas as aplicações e sites continuarão funcionando normalmente com a troca de versão do protocolo. No entanto, as vantagens desta mudança só poderão ser usufruídas por quem adaptar-se às melhorias, tais como empresas de hospedagem, no que diz respeito ao armazenamento de sistemas, e desenvolvedores, que devem considerar o uso de versão mais recente em suas aplicações. Apesar de manter as sintaxes e semânticas dos métodos, status dos códigos, campos dos cabeçalhos e URIs da versão anterior, o HTTP/2 traz significativas mudanças em alguns tópicos extremamente importantes, tais como: tráfego das informações entre o cliente e o servidor; priorização de requisições; paralelização de requisições com multiplexação; compressão automática com HPACK e GZIP e maior segurança, com criptografia nos dados. (VIEIRA, 2016)

Para obter um primeiro contato com essa nova versão do protocolo e poder fazer uma rápida comparação entre essa versão e sua antecessora, acesse: **Demonstração**.

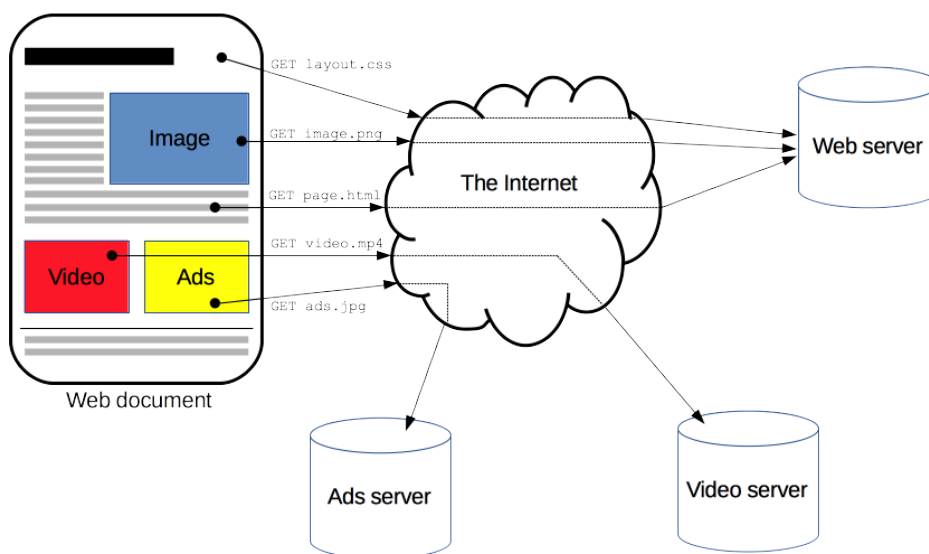
#### 2.2.1.2 Conceitos e funcionamento

Segundo Vieira (2007), "O protocolo HTTP é baseado em requisições e respostas entre clientes e servidores. O cliente — navegador ou dispositivo que fará a requisição; também é conhecido como *user agent* — solicita um determinado recurso (resource), enviando um pacote de informações contendo alguns cabeçalhos (headers) a um URI ou, mais especificamente, URL. O servidor recebe estas informações e envia uma resposta, que pode ser um recurso ou um simplesmente um outro cabeçalho."

É esse protocolo que permite a obtenção de recursos, como documentos HTML, por exemplo. Funciona como a base de qualquer transição de informação na web e possui como característica seu funcionamento *cliente-servidor*. Essa característica informa (implicitamente) que todas as requisições são iniciadas pelo destinatário, geralmente um *browser* (navegador). Tendo feito essa requisição, ao receber a resposta, um documento completo é reconstruído a partir das várias partes de um documento obtidas, tais como texto, imagens, vídeos, scripts, entre outros. (MDN, 2018)

A imagem a seguir facilita essa compreensão:

Figura 2 – Representação da busca de uma página com HTTP



Fonte: MDN (2018)

Isso quer dizer que, toda vez que uma pessoa acessa um *website* ou utiliza seu celular para navegar na web, ela está realizando requisições HTTP constantemente. Cada vez que este usuário clica sobre um *link* ou abre uma nova página, ele (internamente) está enviando algumas informações que identifica o dispositivo que está sendo usado dentro da rede naquele momento e o método da requisição (será abordado mais adiante). Ao receber essa requisição do usuário, o servidor identifica as informações contidas no cabeçalho que são úteis naquele momento e envia sua resposta.

Vale ressaltar que nesta *resposta* do servidor, um dos dados mais importantes presentes é o código de resposta (conhecido como status). Esse código de resposta é representado por números e, portanto, existem diferentes códigos de resposta com diferentes significados. Devido à quantidade de status existentes, eles são divididos em 5 categorias principais, conforme explica MDN (2018):

- **Categoria de resposta - 1xx:** a requisição foi reconhecida e o processamento continua.  
*Códigos existentes:* 100; 101 e 102.
- **Categoria de resposta - 2xx:** a requisição foi recebida, compreendida e executada com sucesso.  
*Códigos existentes:* 200; 201; 202; 203; 204; 205; 206; 207; 208 e 226.
- **Categoria de resposta - 3xx:** alguma outra ação deve ser realizada para tratar a requisição.  
*Códigos existentes:* 300; 301; 302; 303; 304; 305; 306; 307; e 308.
- **Categoria de resposta - 4xx:** informa ao usuário que sua requisição está incorreta ou incompleta.  
*Códigos existentes:* 400; 401; 402; 403; 404; 405; 406; 407; 408; 409; 410; 411;

412; 413; 414; 415; 416; 417; 418; 421; 422; 423; 424; 426; 428; 429; 431 e 451.

- **Categoria de resposta - 5xx:** avisa ao solicitante que o servidor apresentou uma falha ao tentar executar uma requisição que, possivelmente, está correta.

*Códigos existentes:* 501; 501; 502; 503; 504; 505; 506; 507; 508; 510 e 511.

Para que toda essa comunicação seja possível, ainda segundo MDN (2018), o HTTP apresenta algumas características de funcionamento, tais como:

**Simplicidade:** apesar desse item diminuir um pouco com a versão HTTP/2.0, que está em fase de homologação, o HTTP foi planejado para ser algo bastante trivial, básico e legível para qualquer um. Por isso que, mesmo uma pessoa que não tenha tanto conhecimento, consegue ler as mensagens HTTP, o que provê uma maior facilidade para desenvolvimento e testes. Aproveitando que este item aborda a legibilidade, abaixo pode ser encontrado exemplos de requisição e resposta HTTP de forma genérica e, posteriormente, com exemplos, conforme Pillou (2017):

#### Exemplo 2.1 – Pedido HTTP Genérico

```
# PEDIDO GENÉRICO
VERSÃO DO METODO DO URL <break_line>
Cabeçalho: Valor <break_line>

CABEÇALHO: Valor <break_line>
Linha vazia<break_line>
CORPO DO PEDIDO
```

#### Exemplo 2.2 – Pedido HTTP Real

```
# PEDIDO REAL
GET http://br.ccm.net HTTP/1.0
Accept : text/html
If-Modified-Since : Saturday,
    15-January-2017 14:37:11 GMT
User-Agent : Mozilla/52.0.1
    (compatível; MSIE 5.0; Windows 10)
```

#### Exemplo 2.3 – Resposta HTTP Genérica

```
# RESPOSTA GENÉRICA
EXIBIÇÃO CÓDIGO DA VERSÃO—HTTP<break_line>
Cabeçalho: Valor <break_line>
CABEÇALHO: Valor<break_line>
Linha vazia<break_line>

CORPO DA RESPOSTA
```

#### Exemplo 2.4 – Resposta HTTP Real

```
# RESPOSTA REAL
HTTP/1.0 200 OK
Date: Sat,15 Jan 2000 14:37:12 GMT
Server: Microsoft-IIS/2.0
Content-Type: text/HTML
Content-Length : 1245
Last-Modified: Fri,14 Jan 2000 08:25:13 GMT
```

**Extensibilidade:** em virtude da utilização dos cabeçalhos HTTP, este protocolo pode, facilmente, ser estendido e usado em experimentos. É possível, inclusive, introduzir novas funcionalidades entre um cliente e servidor que estejam de acordo sobre uma nova semântica de um cabeçalho.

**Sem Estado:** diz-se que o protocolo HTTP é sem estado. Isto é, o protocolo não armazena nenhum tipo de informação de sessão ou algo do gênero, ou seja, não existe uma relação entre duas requisições sendo feitas através da mesma conexão, por exemplo. No entanto, como o HTTP foi planejado para não armazenar "estados de sessões", os *cookies* HTTP existem para atender essa demanda sempre que for necessário, como em um carrinho de compras de um *e-commerce*, por exemplo.

**Conexão:** comumente, uma conexão é controlada e *trabalha* na camada de transporte. Com isso, teoricamente, essa conexão está fora do alcance do protocolo para que ele possa controlá-la. Contudo, o HTTP não exige que o protocolo de transporte utilizado seja baseado em conexões, mas exige que ele seja confiável e não perca as mensagens (ou, caso ocorra alguma perda, apresente os erros que a causaram). Consequentemente, o HTTP utiliza o padrão TCP - pois o UDP não possui a confiança como característica.

Na citação abaixo, encontra-se a demonstração de um fluxo de execução normal e completo do protocolo HTTP, com o objetivo de que fique claro e evidente todos os passos que precisam ser executados para que essa comunicação ocorra sem problemas.

1) Abre uma conexão TCP: A conexão TCP será usada para enviar uma requisição, ou várias, e receber uma resposta. O cliente pode abrir uma nova conexão, reusar uma conexão existente, ou abrir várias conexões aos servidores. 2) Envia uma mensagem HTTP: mensagens HTTP (antes do HTTP/2.0) são legíveis às pessoas. Com o HTTP/2.0, essas mensagens simples são encapsuladas dentro de quadros (frames), tornando-as impossíveis de ler diretamente, mas o princípio se mantém o mesmo. 3) Lê a resposta do servidor. 4) Fecha ou reutiliza a conexão para requisições futuras. MDN (2018)

Por fim, para que todos esses atributos sejam utilizados e o protocolo seja, realmente, implementado, é necessário que toda requisição defina um método que irá especificar o tipo de requisição daquela chamada. A tabela abaixo, criada por Forouzan e Mosharraf (2013), apresenta os 8 tipos de métodos disponíveis para uma requisição HTTP:

Figura 3 – Métodos disponíveis para uma requisição HTTP

Método	Ação
GET	Solicita um documento ao servidor
HEAD	Solicita informações sobre um documento, mas não o documento em si
PUT	Envia um documento do cliente para o servidor
POST	Envia alguma informação do cliente para o servidor
TRACE	Ecoa a solicitação recebida
DELETE	Remove a página Web
CONNECT	Reservado
OPTIONS	Consulta opções disponíveis

Fonte: Forouzan e Mosharraf (2013)

Dentre esses métodos, os dois mais importantes são o *GET* e o *POST*, que podem ser encontrados em qualquer sistema web disponível na rede hoje. Inclusive, vale a pena evidenciar que o *frontend* do sistema **Simple Judge System**, criado para complementar este projeto, faz uso destes dois métodos e, em virtude disso, os dois serão brevemente definidos nas linhas que seguem, conforme Torres (2017).

**GET:** este método disponibiliza uma capacidade máxima de 1024 caracteres e é frequentemente utilizado quando se deseja passar poucas informações para o servidor, como em

uma pesquisa ou no envio de uma informação para outra página web através da URL. O objetivo deste método é, essencialmente, recuperar um recurso disponível no servidor e, o resultado de uma requisição que utiliza este método, é *cacheável* pelo cliente.

**POST:** já o método POST, por sua vez, utiliza a URI para envio de informações ao servidor. A URI não é retornável para o dispositivo que executou a requisição e, em virtude disso, este método é considerado mais seguro, pois não expõe as informações enviadas no navegador. Cada chamada deste método abre uma nova conexão paralela, que é o meio por onde os dados serão trafegados.

### 2.2.2 TCP/IP

O TCP/IP pode ser considerado como o mais importante protocolo para o envio e recebimento de dados na internet. Seu nome deriva de dois protocolos em especial, são eles: TCP (*Transmission Control Protocol* - Protocolo de Controle de Transmissão) e o IP (*Internet Protocol* - Protocolo de Internet, ou ainda, protocolo de interconexão). Apesar de seu nome fazer referência a dois protocolos apenas, na verdade o TCP/IP é um conjunto de protocolos que são divididos em quatro distintas camadas. Inclusive, Pillou (2018) descreve de forma breve e eficiente o que significa esse elemento tão presente nesse protocolo, “O termo camada é utilizado para evocar o fato que os dados que transitam na rede atravessam vários níveis de protocolos. Assim, os dados (pacotes de informações) que circulam na rede são tratados por cada camada, sucessivamente, acrescentando um elemento de informação (chamado ‘cabeçalho’) e, em seguida, transmitindo à camada seguinte.”.

SOUSA (2013) complementa, “Devido à sua arquitetura e forma de endereçamento, o TCP/IP consegue realizar o roteamento de informação entre redes locais e externas, transferência de arquivos, emulação remota de terminais, e-mail, gerenciamento e outras funções, permitindo a interoperabilidade de diferentes tipos de rede. Vários ambientes e sistemas operacionais suportam o TCP/IP, como o *Unix*, *DOS*, *Windows*, *Linux*, entre outros, permitindo a integração de diferentes plataformas.”.

O modelo proposto pelo TCP/IP é muito semelhante ao famoso modelo OSI (*Open Systems Interconnection*), lançado em 1984 pela Organização Internacional para a Normalização (*International Organization for Standardization*). A diferença dos dois está em como são distribuídos os níveis ou camadas de suas arquiteturas, enquanto o modelo OSI possui sete camadas, o TCP/IP possui apenas quatro. Inclusive, elas dividem-se da seguinte forma, **camada de enlace de rede:** No qual tem os protocolos do nível 1 e 2 do modelo OSI, que carregam informações localmente ou entre pontos de uma rede como *Ethernet*, *Token-Ring*, PPP, X.25 e *Frame-Relay*; **camada de roteamento:** Onde ocorre o roteamento dos dados dentro da rede, executado pelo protocolo IP; **camada de transporte:** Onde atuam os protocolos TCP e UDP que transmitem os dados que são recebidos pelo protocolo IP. Inclusive, o TCP é um protocolo orientado a conexão, faz o controle entre a origem e o destino; **camada de aplicação:** Nela estão as camadas 5, 6 e 7 do modelo OSI, alguns exemplos de protocolos dela são o HTTP, FTP,

SMTP e NTP, dentre outros. No decorrer desse capítulo todas camadas serão descritas de forma detalhada.

#### 2.2.2.1 Histórico

Como citado por SOUSA (2013), “A arquitetura TCP/IP foi projeto de uma agência de pesquisas avançadas da defesa dos EUA, na década de 1960 (Darpa - *Defense Advanced Research Projects*). O objetivo era obter uma arquitetura de comunicação de dados aberta, que permitisse a interligação de redes e computadores locais ou remotos, com *hardwares* diferentes ou mesmo sistemas operacionais e aplicativos diversos. É uma arquitetura cliente-servidor, que se tornou padrão de fato na comunicação entre redes e sistemas de informação.”

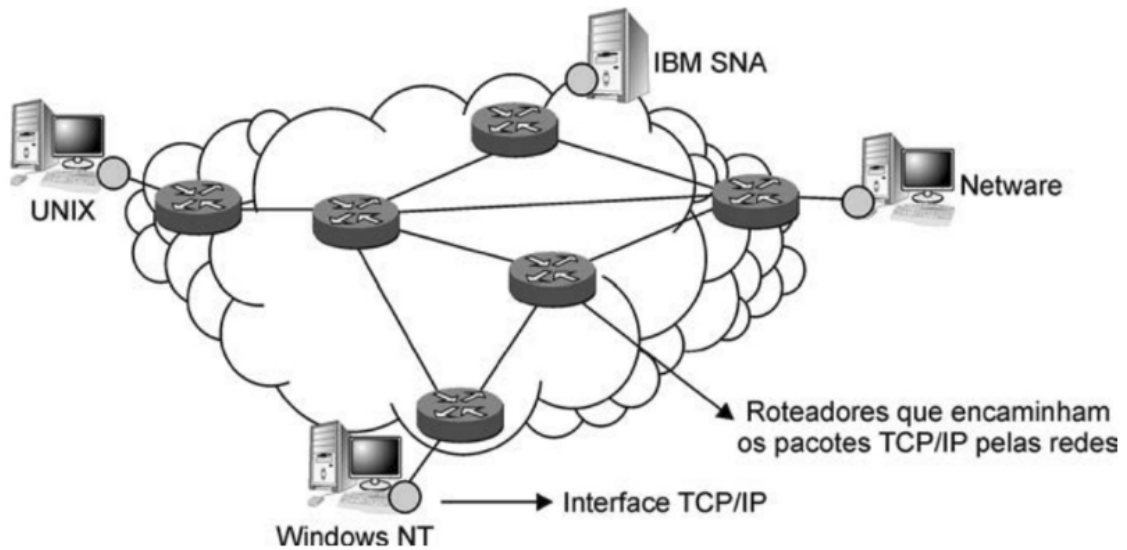
Logo, a primeira rede criada usando esses conceitos foi em 1969, conhecida como ARPANET, ela interligava as universidades americanas e utilizava inicialmente um protocolo conhecido como NCP (*Network Control Protocol*). Somente no ano de 1970 que o TCP entrou em vigor na ARPANET, o principal motivo era que ele apresentava um controle de fluxo melhor. Em 1975 o protocolo IP foi adicionado ao TCP para executar a função de conectividade e endereçamento na rede. Em 1983 ficou definido que os computadores conectados na ARPANET passariam a utilizar o TCP/IP, e em 1986, juntamente com a integração da NSF (*National Science Foundation*) à ARPANET veio a consolidação do TCP/IP como padrão em *internetworking*, ou seja, interconexão de redes.

#### 2.2.2.2 Conceitos e funcionamento

O protocolo TCP/IP tornou-se um padrão mundial para a comunicação de diferentes dispositivos de *hardware* e sistemas operacionais. Por conta disso, é inevitável que novos produtos lançados no mercado venham com interfaces para comunicação TCP/IP. Quando fala-se em protocolos de rede suportados por *hardwares* e *softwares*, no pensamento de muitas pessoas trata-se somente de computadores, *notebooks* e dispositivos móveis. Porém, a amplitude do protocolo TCP/IP vai muito além disso, por exemplo, se uma geladeira, um micro-ondas ou um ar-condicionado tiver uma interface de comunicação TCP/IP, ele poderá comunicar-se com todo e qualquer dispositivo presente na rede e que suporte também esse protocolo.

Com base no que foi descrito acima sobre o protocolo, torna-se notável a versatilidade e praticidade do TCP/IP. A imagem abaixo reforça a ideia de interconexão entre dispositivos distintos, todos comunicando-se através do protocolo em questão.

Figura 4 – Interconexão entre dispositivos com TCP/IP



Fonte: SOUSA (2013)

Como citado por SOUSA (2013), “Como o modelo TCP/IP não é aderente ao modelo de camadas OSI, devemos ter um certo cuidado ao analisá-lo. O que é denominado *data-link* no modelo TCP/IP corresponderia aproximadamente às camadas física e de enlace no modelo OSI. A camada *internetwork* corresponderia à camada de rede OSI. Além disso, as camadas de sessão e apresentação do modelo OSI, não existem no modelo TCP/IP, estando seus conceitos embutidos na camada de aplicação do TCP/IP”. Nota-se que cada camada do TCP/IP tem muito mais tarefas que as camadas do modelo OSI.

A imagem abaixo elucida como são distribuídas as arquiteturas acima mencionadas.

Figura 5 – Distribuição de arquiteturas

Modelo TCP/IP	Modelo OSI
Camada de Aplicação	Camada de Aplicação
	Camada de Apresentação
	Camada de Sessão
Camada de Transporte (TCP)	Camada de Transporte
Camada de Internet (IP)	Camada de Rede
Camada de Acesso à Rede	Camada de Ligação de Dados
	Camada Física

Fonte: Pillou (2018)



#### 2.2.2.2.1 Camada de enlace ou acesso à rede

É a primeira camada da arquitetura do TCP/IP, oferece todos os meios e recursos necessários para transmitir dados através de uma rede. Ela contém todas as especificações relativas à transmissão de dados em uma rede física, seja ela uma rede local, uma conexão com linha telefônica ou qualquer outro meio de conexão a uma rede. Todas as tarefas realizadas nessa camada são invisíveis aos usuários, quem coordena as mesmas é o sistema operacional e os *drivers* referentes ao *hardware*. Abaixo, algumas das tarefas que são executadas:

- Estabelecer e encerrar conexões;
- Notificar e corrigir falhas;
- Utilizar sinais analógicos ou digitais nas conexões;
- Utilizar meios guiados (cabos) ou não guiados (rádio, micro-ondas);
- Formato dos dados;
- Emissão de mais de um sinal em um mesmo meio físico;
- Encaminhamento dos dados na conexão;
- Mapear endereços lógicos em físicos;
- Converte endereços físicos em lógicos (endereço IP);
- Comutar pacotes dentro de um equipamento;
- Permite que o TCP/IP seja implementado em diferentes *hardwares*.

O protocolo de Enlace endereça os dados no meio físico utilizando os *MAC-Address*, eles ficam gravados na memória fixa da placa de rede na sua fabricação. Para que não haja conflitos nos endereços, cada fabricante tem sua própria faixa de endereços e deve respeitá-la. Sousa (2013) descreve algumas características dos *MAC-Address*, por exemplo, são compostos por 6 *bytes*, onde os 3 primeiros representam o código do fabricante e os outros representam o número de sequência. O endereço físico é representado no formato hexadecimal e cada quatro *bits* representa um caractere hexadecimal que varia de 0 a F.

A figura abaixo representa as camadas de enlace e física com seus respectivos componentes para LANs e WANs.

Figura 6 – Camadas de enlace e física: componentes

Enlace	Ethernet	WAN	
	IEEE 802.2	HDLC	Frame-Relay
Físico	IEEE 802.3	EIA/TIA-232 V.35	

Fonte: Bernardino (2018)

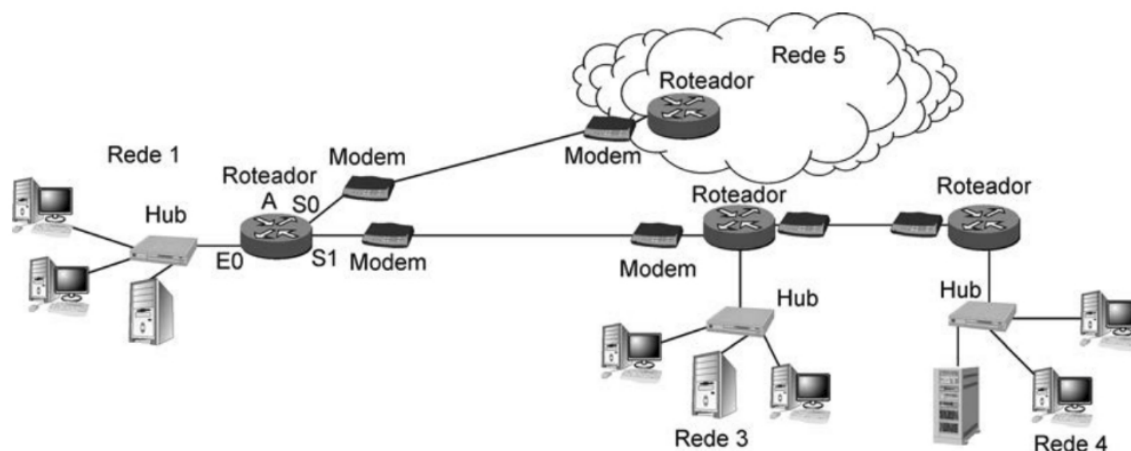
#### 2.2.2.2.2 Camada de rede ou internet

SOUSA (2013) descreve de forma precisa como funciona o transporte e roteamento de dados, “A camada de rede define como transportar dados entre dispositivos que não estão localmente conectados. Para tanto utiliza os endereços lógicos da origem e de destino, como os IPs, e escolhe os caminhos através da rede que serão utilizados para atingir o destino. Interliga dispositivos que não estão no mesmo domínio de colisão. O endereço de rede (IP, por exemplo) chama-se virtual ou lógico (*virtual address* ou *logical address*). A escolha do melhor caminho é feita pelo protocolo de roteamento que fica armazenado no roteador. Alguns protocolos de roteamento, como o RIP (*Routing Information Protocol*), escolhem a melhor rota pelo menor número de trechos ou saltos (*hops*) que deve passar para atingir o destino. Outros protocolos de roteamento, como o OSPF (*Open Shortest Path First*), escolhem a melhor rota pela melhor velocidade ou desempenho dos trechos que fazem parte dela.”

Nessa camada trafegam dois tipos de pacotes, os que levam dados efetivamente (*data packets*) e os pacotes que levam informações de detecção e atualização de rotas (*router update packets*) que vão parar nos roteadores. Os *data packets* não são puramente formados de informações ou dados brutos, eles contém cabeçalhos que transportam alguns metadados. Pacotes de dados IP possuem em seus metadados qual é o IP de origem e qual é o IP de destino. Eles também podem ter alguns *trailers* que ajudam a refinar a transmissão dos dados.

A imagem abaixo demonstra como funciona a interligação de redes distintas, os protagonistas nesse processo são os roteadores, eles encaminham os pacotes de dados baseados nas informações do *header* do pacote IP.

Figura 7 – Interligação de redes distintas: roteadores



Fonte: SOUSA (2013)

#### 2.2.2.2.3 Camada de transporte

Ela é responsável por fazer com que exista uma comunicação entre os dispositivos da rede, também conhecidos como *hosts*. Sua principal função é atribuir confiabilidade e manter os dados íntegros durante o processo de transmissão. Outro importante recurso pertencente a essa camada é o de solicitar retransmissões caso os dados não cheguem em sua origem. Nela existem dois conhecidos protocolos, inclusive um deles é responsável por parte do nome do protocolo em destaque nesse capítulo, são eles: TCP e UDP.

O TCP é constantemente utilizado em aplicações que não possuem tolerância a perda de dados. Ele estabelece conexões ponto-a-ponto entre os *hosts*, garantindo que o dado saia do destino e chega na origem, pois durante a transmissão ele confere se ocorreram perdas de pacotes ou se aparecerem erros inesperados. O TCP também ordena os dados para que chegam da maneira correta em seu destino e não ocorra congestionamentos nas transmissões.

O protocolo UDP não possui as mesmas características do TCP, ao invés de confiabilidade e integridade ele pressupõe velocidade na transmissão. Geralmente é utilizado na transmissão de áudios, vídeos e em outras ocasiões que a velocidade é mais importante que a integridade dos pacotes.

Para localizar os destinos em meio as inúmeras aplicações que estão rodando, são utilizados números identificadores, também conhecidos como portas. Bernardino (2018) cita como funcionam esses mecanismos junto à essa camada, “Essa camada utiliza portas lógicas para garantir que a aplicação (*software*) que iniciou a conversação encontrará no seu destino a aplicação desejada. Essas portas lógicas são canais virtuais aleatórios, geralmente definidos pelo Sistema Operacional, que se abrem conforme o tipo de aplicação executando, como por exemplo, o HTTP utiliza a porta 80, o FTP a porta 21, etc. Esse canal virtual garante que uma aplicação que iniciou uma chamada pela porta 80, como por exemplo, o uso de um navegador

para abrir uma página HTTP no computador A, encontre, no destino, o servidor *web* que fornecerá a página HTTP solicitada também por uma porta 80. Assim se evita que a informação seja direcionada erroneamente para outra aplicação, como por exemplo, um servidor FTP (porta 21).”.

#### 2.2.2.2.4 Camada de aplicação

Ela é a camada mais alta da arquitetura do protocolo TCP/IP, sendo responsável por transportar os dados entre as aplicações. Alguns exemplos de comunicação seriam, recebimento de e-mails (correio eletrônico), transferência de arquivos e emulação de terminais. Os protocolos que estão presentes nessa camada executam um constante gerenciamento, evitando que existam congestionamento entre os dados que trafegam nas aplicações.

Algumas técnicas utilizadas nessa camadas serão descritas abaixo, elas ajudam em quesitos como a diminuição de perda de dados e regulamento do fluxo de dados recebidos e transmitidos.

**Buffers** São memórias temporárias, ou seja, recursos utilizados para armazenar informações até que as mesmas não possam ser processadas.

**Pedido de interrupção e envio de dados** Recurso utilizado para um controle de fluxo, quando um transmissor está emitindo informações em excesso ou apenas um número de informações suficiente para que encham os *buffers*, o receptor pode pedir para que o transmissor interrompa o envio das informações até que todas aqueles dados armazenados sejam processados.

**Envio de dados por janelas** Uma janela resume-se em uma quantidade de pacotes transmitidos sem que se tenham uma confirmação de recebimento. SOUSA (2013) descreve como é o funcionamento desses envios e como são retornadas as confirmações de recebimento. Basicamente, o receptor após receber todos os pacotes de uma determinada janela (*windowing*) envia ao receptor uma confirmação de recebimento, o termo é conhecido como *acknowledgement*.

Para exemplificar de forma mais clara as tarefas executadas por essa parte do TCP/IP, serão abordados de forma superficial alguns dos principais protocolos, eles são: FTP, SMTP e NTP.

**FTP** Conhecido como Protocolo de transferência de arquivos, ele é uma forma muito versátil e rápida de transferir arquivos. Inclusive, quando trata-se de transferência de arquivos o FTP está no topo do ranking de utilizações na internet. Ele utiliza apenas duas conexões durante suas sessões, uma *half-duplex* e uma *full-duplex*, uma controla a conexão e outra gerencia a transferência de arquivos, respectivamente. Duas portas são utilizadas pelo protocolo, a porta

21 é utilizada para estabelecer e manter uma conexão entre cliente e servidor e a porta 20 é utilizada para a transferências dos arquivos.

**SMTP** É um protocolo de envio de mensagens, inclusive é o protocolo padrão utilizado pelo TCP/IP para o envio de e-mails. O SMTP não permite a recepção de mensagens, ou seja, para realizar tal tarefa um outro protocolo deve ser utilizado, exemplos de protocolo são o POP3 e o IMAP. Ele utiliza a porta 25 para sua execução e é conhecido por ter uma forma de execução muito simples, pois todo conteúdo que trafega pelo protocolo são compostos de textos simples, inclusive os arquivos. Posteriormente os mesmos são remontados. A imagem abaixo ilustra como funciona uma sessão SMTP.

Figura 8 – Sessão SMTP: funcionamento

**Passos da conexão SMTP**

```

S: 220 www.example.com ESMTP Postfix
C: HELO mydomain.com
S: 250 Hello mydomain.com
C: MAIL FROM: sender@mydomain.com
S: 250 Ok
C: RCPT TO: friend@example.com
S: 250 Ok
C: DATA
S: 354 End data with <CR><LF>.<CR><LF>
C: Subject: test message
C: From: sender@mydomain.com
C: To: friend@example.com
C:
C: Hello,
C: This is a test.
C: Goodbye.
C: .
S: 250 Ok: queued as 12345
C: quit
S: 221 Bye
  
```

Fonte: Bezerra (2008)

**NTP** Algumas aplicações precisam que todos *hosts* estejam com seus relógios sincronizados, o que naturalmente não ocorre por inúmeros fatores. Para isso é utilizado esse protocolo que tem seu funcionamento baseado no UDP. O funcionamento do NTP pode parecer simples, porém os algoritmos de sincronização de relógios (ou cálculos de deslocamento de hora) são extremamente complexos.

## 2.2.3 Protocolo frontend/backend do postgresql

### 2.2.3.1 Prefácio

O PostgreSQL tem um protocolo designado para a comunicação de frontends (clientes) e backends (servidores). O protocolo é implementado sobre TCP/IP e também por Unix Sockets,

ambos serão detalhados posteriormente no documento. A porta padrão definida no protocolo é a 5432, ela foi a porta registrada na Internet Assigned Numbers Authority (IANA), porém na prática qualquer número de porta não privilegiada pode ser utilizada.

Servidores podem suportar várias versões de protocolos, inclusive a mensagem inicial que um cliente envia quando está tentando iniciar uma conexão é a versão do protocolo que ele está utilizando no momento. O servidor pode rejeitar a conexão por não suportar o protocolo especificado pelo cliente, posteriormente o servidor verifica qual a versão secundária de protocolo que o cliente está tentando utilizar, se a versão não for suportada o servidor tem a possibilidade de encerrar a conexão ou enviar uma mensagem de *NegotiateProtocolVersion*, ela informa ao cliente qual é a versão de protocolo secundária mais alta que o servidor consegue suportar. O cliente tem a opção de prosseguir com a conexão utilizando algum dos protocolos sugeridos pelo servidor ou abortá-la.

O capítulo tem um grande percentual de seu conteúdo retirado da documentação oficial do Group (2018). No intuito de evitar redundâncias em todos os subcapítulos a seguir, optamos por deixar a devida menção aos autores na introdução do capítulo em questão. Ressaltamos, todos os conteúdos que não tiverem sua devida citação tratam-se do Group (2018), os demais autores terão suas citações normalmente.

### 2.2.3.2 Mensagens

Toda a comunicação é através de um fluxo de mensagens. O primeiro *byte* de uma mensagem identifica o tipo de mensagem e os próximos quatro *bytes* especificam o tamanho do resto da mensagem. O conteúdo restante da mensagem é determinado pelo seu tipo. Por motivos históricos, a primeira mensagem enviada pelo cliente (a mensagem de *startup*) não possui um *byte* inicial do tipo mensagem.

#### 2.2.3.2.1 Servidor

O servidor (ou *postmaster*, em versões mais antigas da documentação do PostgreSQL) será um elemento muito importante para o decorrer desse capítulo, por conta disso o mesmo será detalhado de forma breve. Ele é o servidor de banco de dados multiusuário do PostgreSQL, para que uma aplicação (cliente) se comunique com o banco de dados ela deve se conectar a um servidor em execução, além disso é ele quem gerencia a comunicação entre os processos que estão executando. Mais de um processo servidor pode ser executado em um sistema ao mesmo tempo, porém os processos devem utilizar áreas de dados e portas de comunicação distintas.

#### 2.2.3.2.2 Fluxo de mensagens

Existem cinco tipos diferentes de fluxos de mensagem, dependendo do estado atual da conexão, são eles: *Startup*, *Query*, *Function Call*, *Copy* e *Termination*. No decorrer do capítulo, todos os fluxos serão descritos de forma detalhada.

**Startup** Para que uma nova sessão seja iniciada, um cliente abre uma conexão com o servidor e envia uma mensagem de inicialização para o mesmo. A mensagem é composta pelo nome do usuário, qual banco de dados ele deseja fazer conexão e qual a versão do protocolo que ele está utilizando. Existem algumas outras informações que podem ser passadas na *Startup*, como por exemplo parâmetros de tempo de execução. Todas essas informações são utilizadas pelo servidor, juntamente com as informações presentes nos arquivos de configuração (como por exemplo o `pg_hba.conf`) para determinar se a conexão é provisoriamente aceitável e se será necessário algum tipo de autenticação.

A próxima etapa é o servidor enviar uma mensagem solicitando a autenticação do cliente, o mesmo deve responder com uma autenticação apropriada. O fluxo de autenticação pode ter inúmeras iterações entre cliente e servidor, porém nenhum dos métodos atuais usa mais que uma solicitação e resposta. Outra possibilidade é que nenhuma solicitação de autenticação ocorra, portanto nenhuma resposta do cliente é necessária.

O ciclo de autenticação chegará ao fim quando o servidor aceitar (`AuthenticationOk`) ou rejeitar (`ErrorResponse`) a conexão. Existem alguns tipos de mensagem que o servidor pode retornar na etapa de autenticação, são elas:

- **ErrorResponse**

A tentativa de conexão foi rejeitada. O servidor encerra a conexão de forma imediata.

- **AuthenticationOk**

A tentativa de autenticação foi feita com sucesso.

- **AuthenticationKerberosV5**

O *frontend* agora deve participar de um diálogo de autenticação do Kerberos V5 com o servidor. Se isso for bem sucedido, o servidor responde com um `AuthenticationOk`, caso contrário ele responde com um `ErrorResponse`.

- **AuthenticationCleartextPassword**

O *frontend* agora deve enviar uma `PasswordMessage` contendo a senha em forma de texto não criptografado. Se esta é a senha correta, o servidor responde com um `AuthenticationOk`, caso contrário ele responde com um `ErrorResponse`.

- **AuthenticationCryptPassword**

O *frontend* deve enviar uma `PasswordMessage` contendo a senha criptografada via *crypt* (3), usando o *salt* de 2 caracteres especificado na mensagem `AuthenticationCryptPassword`. Se esta é a senha correta, o servidor responde com um `AuthenticationOk`, caso contrário ele responde com um `ErrorResponse`.

- **AuthenticationMD5Password**

O *frontend* agora deve enviar uma *PasswordMessage* contendo a senha criptografada via MD5, usando o *salt* de 4 caracteres especificado na mensagem *AuthenticationMD5Password*. Se esta é a senha correta, o servidor responde com um *AuthenticationOk*, caso contrário ele responde com um *ErrorResponse*.

- **AuthenticationSCMCredential**

Essa resposta só é possível para conexões locais do domínio Unix em plataformas que suportam mensagens de credenciais do SCM. O *frontend* deve emitir uma mensagem de credencial do SCM e, em seguida, enviar um único *byte* de dados. Se a credencial for aceitável, o servidor responderá com um *AuthenticationOk*, caso contrário, responderá com um *ErrorResponse*.

Na próxima etapa um processo de *backend* está sendo iniciado e o *frontend* passa a ser apenas um espectador. Erros ainda podem acontecer nessa fase do processo (*ErrorResponse*), supondo que tudo ocorra como esperado, o *backend* enviará algumas mensagens, são elas: *ParameterStatus*, *BackendKeyData* e *ReadyForQuery*.

Outro detalhe importante de ser mencionado é que o *backend* tentará aplicar as configurações que foram informadas anteriormente na inicialização. Se bem-sucedidas, esses valores se tornarão padrões de sessão.

Abaixo estão alguns detalhes sobre as possíveis mensagens do *backend* nessa etapa do processo:

- **BackendKeyData**

Essa mensagem fornece dados de chave secreta que o *frontend* deve salvar se quiser emitir solicitações de cancelamento posteriormente. O *frontend* não deve responder a esta mensagem, mas deve continuar ouvindo uma mensagem *ReadyForQuery*.

- **ParameterStatus**

Essa mensagem informa ao *frontend* sobre a configuração atual dos parâmetros de *backend*, como *client\_encoding* ou *DateStyle*. O *frontend* pode ignorar essa mensagem ou gravar as configurações para uso futuro. O *frontend* não deve responder a esta mensagem, mas deve continuar ouvindo uma mensagem *ReadyForQuery*.

- **ReadyForQuery**

O *startup* está concluído. O *frontend* agora pode emitir comandos.

- **ErrorResponse**



*Startup* falhou. A conexão é fechada após o envio desta mensagem.

- **NoticeResponse**

Uma mensagem de aviso foi emitida. O *frontend* deve exibir a mensagem, mas continua ouvindo ReadyForQuery ou ErrorResponse.

**Queries** O ciclo de uma *query* inicia quando o *frontend* envia para o *backend* um ou mais comandos SQL. Posteriormente, o *backend* responderá conforme a quantidade de comandos enviados no início do processo. O final do procedimento é quando uma mensagem ReadyForQuery chega ao *frontend*, lhe informando que novos comandos podem ser executados.

O *frontend* tem a possibilidade de executar comandos antes da chegada do ReadyForQuery, porém o mesmo deve assumir a responsabilidade de descobrir possíveis falhas na execução.

As mensagens que o *backend* poderá retornar no decorrer dessa etapa são:

- **CommandComplete**

Comando SQL executado normalmente.

- **CopyInResponse**

O *backend* está pronto para copiar dados do *frontend* para uma tabela.

- **CopyOutResponse**

O *backend* está pronto para copiar dados de uma tabela para o *frontend*.

- **RowDescription**

Indica que as linhas estão prestes a ser retornadas em resposta a uma consulta *select*, *fetch*, etc... O conteúdo desta mensagem descreve o layout da coluna das linhas. Isso será seguido por uma mensagem DataRow para cada linha sendo retornada ao *frontend*.

- **DataRow**

Um dos conjuntos de linhas retornadas por uma consulta *select*, *fetch*, etc...

- **EmptyQueryResponse**

Uma consulta com o retorno vazio.

- **ErrorResponse**

Erro na execução do processamento.

- **ReadyForQuery**

O processamento de consulta está completo. Uma mensagem é enviada para indicar isso. O ReadyForQuery sempre será enviado, em casos de sucesso ou falha.

- **NoticeResponse**

Uma mensagem de aviso foi emitida em relação à consulta. Os avisos são adicionais a outras respostas.

O protocolo divide as *queries* em simples e estendidas, a principal diferença entre elas é que as estendidas dividem o processo das consultas simples em várias etapas no intuito de melhorar e reaproveitar alguns procedimentos/etapas já realizadas, por consequência existe um ganho de performance.

**Function call** Esse subprotocolo permite que os clientes possam chamar todas as funções disponíveis no catálogo do banco de dados, obviamente, os mesmos devem ter as permissões necessárias para executar uma determinada função.

O processo tem início quando o *frontend* envia para o *backend* uma mensagem de chamada de função. Posteriormente, uma resposta ReadForQuery retornará informando que o *frontend* poderá enviar com segurança uma nova chamada de função.

As possíveis mensagens de resposta do *backend* são:

- **ErrorResponse**

Ocorreu um erro.

- **FunctionCallResponse**

A chamada de função foi concluída e retornou o resultado fornecido na mensagem.

- **ReadyForQuery**

O processamento de consulta está completo. Uma mensagem é enviada para indicar isso. O ReadyForQuery sempre será enviado, em casos de sucesso ou falha.

- **NoticeResponse**

Uma mensagem de aviso foi emitida em relação à consulta. Os avisos são adicionais a outras respostas.

**NOTA:** A chamada de função é um recurso legado do PostgreSQL, a própria documentação recomenda a não utilização delas em implementações futuras.

**Copy** Esse comando permite que uma grande quantidade de dados seja transferida para o servidor ou parta do servidor. Como citado por Leopoldino (2007), “Este comando realiza a cópia de dados entre tabelas e arquivos. É muito utilizada para fazer cargas de dados e implementar rotinas de *backup* e migrações de bancos de dados. Não faz parte da especificação tradicional do SQL.”.

**Termination** O modo correto de se terminar uma conexão, ou seja chamar o fluxo *termination*, é o *frontend* enviando uma mensagem para que o *backend* feche-a. Existem casos em que o *backend* é interrompido inesperadamente, antes de que a conexão se interrompa por completo ele tentará enviar mensagens aos clientes com o motivo da desconexão.

### 2.2.3.3 *Unix Sockets*

Como citado por Hanson (2017), “Os soquetes de domínio UNIX são um método pelo qual os processos no mesmo host podem se comunicar. A comunicação é bidirecional com soquetes de fluxo e unidirecional com soquetes de datagrama.”. Ao contrário do que se utiliza em cima de um protocolo TCP/IP, as Unix Sockets não informam um endereço IP e uma porta, mas sim um caminho válido para que os processos consigam encontrá-lo e trafegar seus dados.

Hanson (2017) também cita uma das maiores vantagens que os Unix Sockets tem sobre os Sockets de internet, “Para soquetes de domínio UNIX, as permissões de arquivo e diretório restringem quais processos no host podem abrir o arquivo e, assim, se comunicar com o servidor. Portanto, os soquetes de domínio UNIX fornecem uma vantagem sobre os soquetes de Internet (aos quais qualquer um pode se conectar, a menos que uma lógica de autenticação extra seja implementada).”.

### 2.2.3.4 *TCP/IP vs Unix sockets*

Como já descrito acima, o protocolo do PostgreSQL tem duas formas de transitar dados entre *frontends* e *backends*, são elas: TCP/IP e Unix Sockets. No decorrer desse capítulo serão feitos alguns pequenos comparativos entre os dois recursos e será demonstrado como iniciar uma conexão com o banco de dados definindo qual será o método utilizado.

#### 2.2.3.4.1 *Características*

Unix Sockets utilizam os recursos do sistema de arquivos do Sistema Operacional para definir as permissões de seus dados. É possível limitar o acesso direto de processos rodando em um determinado *host*. Eles são comprovadamente mais velozes que o TCP/IP.

Em geral, o principal motivo para se implementar TCP/IP é que ele fornece independência de localização e portabilidade imediata, é possível trocar um cliente ou movê-lo e atualizar um endereço que ele funcionará de forma instantânea.

#### 2.2.3.4.2 Implementação utilizando python

Abaixo será demonstrado como definir qual será o método de transmissão de informações entre um *frontend* (cliente) e um *backend* (servidor). A linguagem utilizado será o Python, juntamente com a biblioteca *psycopg2*.

**TCP/IP** O que caracteriza a conexão utilizando o TCP/IP é a passagem do parâmetro de *host*.

---

```

/* Código de exemplo */
def __init__( self ):
    try :
        self .connection = psycopg2.connect(
            "dbname='banco' user='postgres' host='localhost' password='minhasenha' port='5432'"
        )
        self .connection .autocommit = True
        self .cursor = self .connection .cursor ()
    except :
        print ("Erro ao conectar no database!")

```

---

**Unix Sockets** Elas não precisam de um *host* informado, está intrínseco que os processos estarão rodando em um mesmo. Portanto, deve-se ocultar o parâmetro de *host* e o parâmetro de porta torna-se desnecessário.

---

```

/* Código de exemplo */
def __init__( self ):
    try :
        self .connection = psycopg2.connect(
            "dbname='banco' user='postgres' password='minhasenha'"
        )
        self .connection .autocommit = True
        self .cursor = self .connection .cursor ()
    except :
        print ("Erro ao conectar no database!")

```

---

### 3 DOCUMENTAÇÃO DAS APLICAÇÕES DESENVOLVIDAS

#### 3.1 *Frontend*

##### 3.1.1 Visão Geral

Toda aplicação que recebe *frontend* como adjetivo, de forma geral, tem como função ser o intermédio entre o usuário, suas intenções e o sistema. Isso porque o *frontend*, é o responsável pela parte visual de um sistema e envolve conceitos como *design* e *usabilidade*.

No caso deste projeto, o sistema *web* desenvolvido se adequa muito bem nesta definição geral, tendo em vista que ele foi criado justamente para abstrair detalhes de implementação e funcionamento e possibilitar que, de forma simples, uma pessoa que se interessar pelo assunto veja um exemplo, na prática, de como o *GlusterFS* funciona.

Portanto, esta seção do documento vai explicar, de forma breve e objetiva, os principais pontos que compõe a parte *web* do **Simple Judge System**, principalmente no que se refere a código. Além disso, o fluxo de execução ideal do sistema será resumido e listado através de passos.

##### 3.1.2 Fluxo de Trabalho

Tendo em vista que essa *visão* ou *face* do sistema foi desenvolvida utilizando **Laravel Framework**, **HTML**, **CSS**, entre outros conceitos *web* e que, nativamente, o *framework* implementa o conceito de **MVC** (*Model-View-Controller*), nos parágrafos que seguem será descrito como os arquivos que contém as estruturas *HTML* e *CSS* trabalham e interagem com os *controllers* da aplicação.

Neste ponto, vale ressaltar que as 3 partes principais do sistema - *Dashboard*; *Enviar e Submissões* - possuem a mesma estrutura e, para que esta explicação não se torne repetitiva e desnecessária, apenas uma delas será explanada, pois será suficiente para que fique claro como a estrutura do projeto *web* funciona.

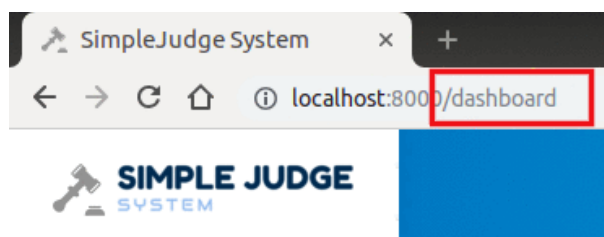
O fluxo em si inicia quando o usuário acessa uma *URL* em seu navegador. Em projetos *Laravel*, quando esta ação é executada, o arquivo **web.php** é interpretado e, caso a *URL* acessada pelo usuário possua um tratamento, ou seja, caso esse *URL* direcione o usuário para um destino esperado pela aplicação, esse destino será interpretado naquele instante. Veja abaixo o exemplo da rota que o usuário acessa ao navegar para o *dashboard* da aplicação **Simple Judge System**:

Figura 9 – Exemplo de tratamento para *URL* acessada pelo usuário

```
20
21 // Dashboard
22 Route::group(['prefix' => 'dashboard'], function() {
23     Route::get('', ['as' => 'dashboard', 'uses' => 'DashboardController@index']);
24 });
```

Fonte: Própria (2018)

Esse trecho de código é executado quando a pessoa que está utilizando o sistema acessa a *URL* exibida abaixo, por exemplo. Vale lembrar que a parte inicial - *localhost:8000* -, que seria o domínio do sistema, como *simplejudgesystem.com.br:8000*, por exemplo, deve ser desconsiderada:

Figura 10 – *URL* acessada pelo usuário

Fonte: Própria (2018)

Assim que o trecho de código é executado, em virtude da segunda chave do *array* da rota, a chave **uses**, o método *index* (localizado após o *caracter* @), do arquivo **DashboardController**, é invocado. Esse método possui em seu interior algumas linhas que executam a lógica responsável por criar as estatísticas de uso do sistema e, por fim, um simples redirecionamento para a *view* responsável por das as "boas-vindas" ao usuário e exibir todas as estatísticas organizadas no método.

Esse método geralmente é o primeiro a ser construído em um *controller*, isso porque a maioria dos arquivos controladores de um projeto estão vinculados à uma visão. Essa vinculação ocorre para que seja possível fornecer dados dinâmicos necessários para que a visão consiga exibir as informações corretas e para que o controlador possa ser invocado sempre que o usuário realizar uma ação. A imagem abaixo apresenta o conteúdo do método *index* citado anteriormente.

Figura 11 – Método *index* do arquivo **DashboardController.php**

```

10 class DashboardController extends Controller
11 {
12
13     public function index() {
14         $submissoesCriadasHoje = Submissao::whereDate('created_at', Carbon::today())->get()->count();
15         $submissoesTotais = Submissao::all()->count();
16         $linguagemMaisUtilizadaHoje = Linguagem::find(1);
17         $linguagemMaisUtilizadaTotal = Linguagem::find(1);
18
19         // Identifica qual linguagem de programação foi mais utilizada nas submissões realizadas no dia atual e em todo o histórico do sistema para popular os grá
20         $maiorNumeroSubmissoesHoje = Linguagem::find(1)->submissoes()->whereDate('created_at', Carbon::today())->get()->count();
21         $maiorNumeroSubmissoesTotal = Linguagem::find(1)->submissoes()->count();
22         $linguagensDeProgramacao = Linguagem::where('id', '>', 1)->get();
23         foreach ($linguagensDeProgramacao as $key => $linguagem) {
24             $numeroSubmissoesHojeLinguagemAtual = $linguagem->submissoes()->whereDate('created_at', Carbon::today())->get()->count();
25             $numeroSubmissoesTotalLinguagemAtual = $linguagem->submissoes()->count();
26
27             if( $numeroSubmissoesHojeLinguagemAtual > $maiorNumeroSubmissoesHoje ) {
28                 $maiorNumeroSubmissoesHoje = $numeroSubmissoesHojeLinguagemAtual;
29                 $linguagemMaisUtilizadaHoje = $linguagem;
30             }
31             if( $numeroSubmissoesTotalLinguagemAtual > $maiorNumeroSubmissoesTotal ) {
32                 $maiorNumeroSubmissoesTotal = $numeroSubmissoesTotalLinguagemAtual;
33                 $linguagemMaisUtilizadaTotal = $linguagem;
34             }
35         }
36
37         if( $maiorNumeroSubmissoesHoje <= 0 ) $linguagemMaisUtilizadaHoje = null;
38         if( $maiorNumeroSubmissoesTotal <= 0 ) $linguagemMaisUtilizadaTotal = null;
39
40         // Após obter os valores concretos de utilização nas submissões, calcula a porcentagem de uso para utilizar nos gráficos
41         $porcentagemSubmissoesHoje = $this->calculaPorcentagemDeUso($submissoesCriadasHoje, $maiorNumeroSubmissoesHoje);
42         $porcentagemSubmissoesTotal = $this->calculaPorcentagemDeUso($submissoesTotais, $maiorNumeroSubmissoesTotal);
43
44         return view('dashboard', compact('submissoesCriadasHoje', 'submissoesTotais', 'linguagemMaisUtilizadaHoje', 'porcentagemSubmissoesHoje', 'linguagemMaisUtili
45     }
46 }

```

Fonte: Própria (2018)

Após buscar os dados necessários do banco de dados e criar as informações atuais para que a tela *dashboard* possa ser preenchida com informações relevantes ao uso do sistema, note a linha **44** que **retorna** uma visão - ou *view*. Essa linha indica ao sistema que o pré-processamento necessário para que o conteúdo da página estivesse completo está pronto e que, neste momento, a página já pode ser renderizada.

Sendo assim, o *framework* se encarrega de "informar" ao interpretador para que ele carregue o arquivo **dashboard.blade.php**, que é o arquivo que contém o código fonte da página de entrada do sistema. Esse código-fonte é composto, basicamente, por elementos *HTML* e variáveis *PHP* que são exibidas conforme necessário. Veja o conteúdo completo do arquivo abaixo:

Figura 12 – Arquivo responsável pelo conteúdo da tela *Dashboard*

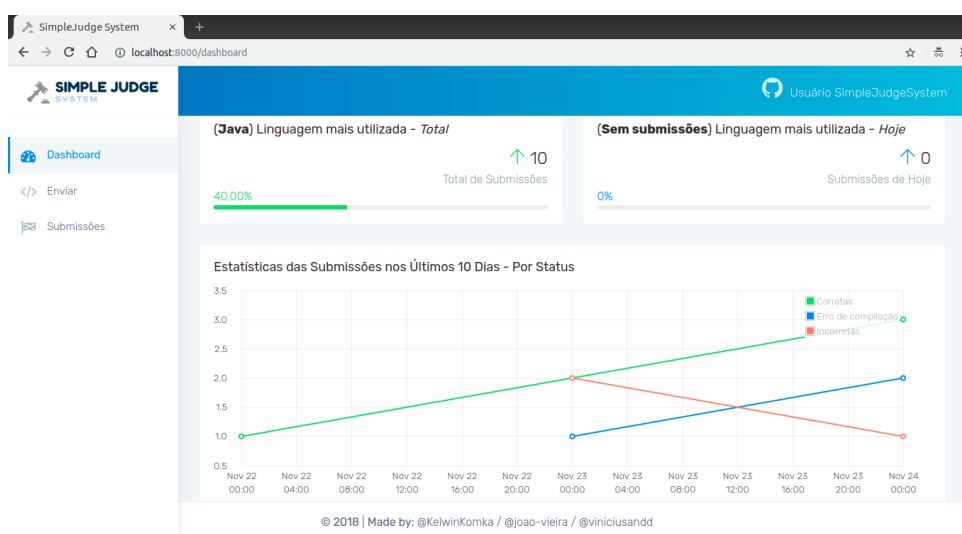
```

1 @extends('layouts.app')
2
3 @section('content')
4
5 <!-- Row -->
6 <div class="row">
7 <!-- Column -->
8 <div class="col-sm-6">
9 <div class="card">
10 <div class="card-title">(<span style="font-weight: bold; font-size: 18px;">{{ $linguagemMaisUtilizadaTotal->nome or 'Sem submissões' }}</span>) Linguagem mais utilizada - <i>Total</i> </div>
11 <div class="text-right">
12 <div class="font-light m-b-0"><i>class="ti-arrow-up text-success"</i> </div> <div class="font-weight: bold">{{ $submissoesTotais }}</div>
13 <div class="text-muted">Total de Submissões</div>
14 </div>
15 <div class="text-success">{{ $porcentagemSubmissoesTotal }}%</div>
16 <div class="progress">
17 <div class="progress-bar bg-success" role="progressbar" style="width: {{ $porcentagemSubmissoesTotal }}%; height: 6px;" aria-valuemin="0" aria-valuemax="100"></div>
18 </div>
19 </div>
20 </div>
21 <!-- Column -->
22 <div class="col-sm-6">
23 <div class="card">
24 <div class="card-title">(<span style="font-weight: bold; font-size: 18px;">{{ $linguagemMaisUtilizadaHoje->nome or 'Sem submissões' }}</span>) Linguagem mais utilizada - <i>Hoje</i> </div>
25 <div class="text-right">
26 <div class="font-light m-b-0"><i>class="ti-arrow-up text-info"</i> </div> <div class="font-weight: bold">{{ $submissoesCriadasHoje }}</div>
27 <div class="text-muted">Submissões de Hoje</div>
28 </div>
29 <div class="text-info">{{ $porcentagemSubmissoesHoje }}%</div>
30 <div class="progress">
31 <div class="progress-bar bg-info" role="progressbar" style="width: {{ $porcentagemSubmissoesHoje }}%; height: 6px;" aria-valuemin="0" aria-valuemax="100"></div>
32 </div>
33 </div>
34 </div>
35 </div>
36 </div>
37 </div>
38 </div>
39 <!-- Row -->
40 <div class="row">
41 <div class="col-sm-12">
42 <div class="card">
43 <div class="card-title">Estatísticas das Submissões nos Últimos 10 Dias - Por Status</div>
44 <div id="placeholder" style="min-width: 600px; min-height: 300px;"></div>
45 </div>
46 </div>
47 </div>
48 </div>
49 </div>
50 </div>
51 </div>
52 </div>
53 </div>
54 </div>
55 </div>
56 </div>
57 </div>
58 </div>
59 </div>
60 </div>
61 </div>
62 </div>
63 </div>
64 </div>

```

Fonte: Própria (2018)

Assim que esse arquivo for interpretado, as informações nele contidas já estarão disponíveis e visíveis para o usuário através do *browser* pelo qual ele acessou o sistema. Como mencionado anteriormente, esse fluxo *MVC* básico se repete nas demais telas do sistema. Veja abaixo a tela resultante da interpretação do arquivo referido.

Figura 13 – Tela resultante da interpretação do arquivo *dashboard.blade.php*

Fonte: Própria (2018)



### 3.1.3 Fechamento

Concluindo este item, como citado no início da seção, a parte visual desse projeto foi criada com o intuito de diminuir a curva de aprendizagem do projeto como um todo. Isso é muito benéfico para que pessoas sem conhecimento algum sobre sistemas de arquivos distribuídos, por exemplo, possam ver a demonstração de uma utilização real do conceito, de forma clara e simples, sem precisarem aprender todos os tópicos envolvidos no assunto.

## 3.2 *Backend*

### 3.2.1 Visão Geral

Para realizar a implementação do *backend* do **Simple Judge System**, foi utilizada a linguagem de programação Python. Ela demonstrou ser uma excelente ferramenta para a criação de *scripts*, permitindo que o programador execute-os e capture todos os retornos que podem partir do sistema operacional. Algumas bibliotecas foram utilizadas no decorrer do processo, todas receberão menção nesse capítulo, juntamente com trechos de código fonte retirados do projeto.

### 3.2.2 Fluxo de Trabalho

O fluxo de execução do *backend* inicia quando um registro é inserido no banco de dados. Uma função captura quais foram os registros e com base no ID dos mesmos é executada a busca dos arquivos dentro da pasta do sistema. Agora, inicia-se um fluxo de compilação, execução, comparação de resultados e envio de informações para o banco de dados. Todas essas etapas serão detalhadas em tópicos que serão compostos de uma parte teórica e exemplos práticos.

#### 3.2.2.1 Compiladores / Interpretadores

O *software* conta com três compiladores / interpretadores para funcionar, são eles: **g++**, **py\_compile** e o **javac**. Cada um deles tem seus próprios *scripts* dentro do *software*. Os *scripts* são divididos em duas categorias: compilação e execução.

Abaixo, serão mostrados os *scripts* de compilação das três linguagens:

- **C++**

```
g++ <diretorio><arquivo>.cpp -o <diretorio>compilacoes/<arquivo> 2>
<diretorio>erros/<arquivo>.txt && echo 'Compilado com sucesso'
|| cat <diretorio>erros/<arquivo>.txt
```

- **Python**

```
python -m py_compile <diretorio><arquivo>.py 2>
<diretorio>erros/erros_<arquivo>.txt && echo 'Compilado com sucesso'
|| cat <diretorio>erros/erros_<arquivo>.txt
```

- **Java**

```
javac <diretorio><arquivo>.java 2> <arquivo>erros/erros_<arquivo>.txt
&& echo 'Compilado com sucesso'
|| cat <diretorio>erros/erros_<arquivo>.txt
```

Após o arquivo ser compilado com sucesso, inicia-se o fluxo de execução das compilações que foram criadas. Todas as execuções utilizam entradas que estão definidas na pastas /arquivos/entradas/ do sistema, elas são direcionadas para o arquivo compilado dentro do *script* de execução. Abaixo, serão mostrados os *script* de execução das três linguagens:

- **C++**

```
cd <diretorio>compilacoes/ && ./<arquivo> < <entrada> >
<diretorio>compilacoes/<arquivo>.txt
```

- **Python**

```
python <diretorio><arquivo>.pyc < <entrada> >
<diretorio>compilacoes/<arquivo>.txt
```

- **Java**

```
cd <diretorio> && java <arquivo> < <entrada> >
<diretorio>compilacoes/<arquivo>.txt
```

O fluxo de execução é encerrado após a criação das saídas em formato de texto na pasta /arquivos/compilacoes/ do sistema ou se ocorrer alguma falha durante o processo.

### 3.2.2.2 *Scripts*

Para realizar a execução de todos os comandos descritos no subcapítulo anterior, é utilizada a biblioteca subprocess. Em seguida, um exemplo de como é a sintaxe esperada para utilizar esse recurso.

```
compilacao = subprocess.check_output('comandos do sistema', shell=True)
```

Como foi utilizada a chamada *check\_output*, é possível armazenar o retorno em alguma variável e depois ver o que o sistema operacional retornou. Abaixo encontra-se um exemplo do que foi dito.

```
print(compilacao.decode('utf-8'))
```

### 3.2.2.3 Retornos de execução e compilação

Todas compilações geram arquivos dentro das pastas do sistema, se executadas com sucesso, geram os ‘executáveis’. Se houver falhas na execução, geram um arquivo de texto contendo quais foram os erros encontrados no arquivo que o usuário enviou. Uma compilação que não obtém sucesso, interrompe imediatamente o fluxo e é enviado para o banco de dados ‘Erro de compilação’, juntamente com os erros do usuário. Se a compilação tiver sucesso o processo da sequência e chega aos *scripts* de execução.

Quando é executado o *script* de execução, o mesmo pode ou não gerar um arquivo no formato de texto. Se ocorrer falha na execução, o fluxo do sistema é interrompido e é enviado ao banco de dados as seguintes informações sobre a tentativa do usuário, ‘Incorreta’ e ‘Resposta incorreta: 100.0’. Se for executado com sucesso, é criada a saída na pasta /arquivos/compilacoes/ e essa saída é comparada com as saídas que estiverem na pasta /arquivos/saidas/. Para executar essas comparações é utilizada a biblioteca *diff*, que retorna um percentual baseado nas diferenças encontradas no conteúdo dos dois arquivos que foram comparados. Seu funcionamento é semelhante ao comando *diff*.

Abaixo, o trecho do código fonte onde o percentual é capturado pelo *software*. A função *SequenceMatcher* é a protagonista nessa etapa:

```
percent = SequenceMatcher(None, compilacao, saida_esperada)
resposta = percent.ratio() * 100
```

### 3.2.2.4 *Threads*

O *software* executa constantes buscas no banco de dados, para que isso não travasse a Thread principal uma outra foi criada e ela é executada logo no início do *software*. Para criar a thread apontando para a função do sistema que executa as buscas, foi utilizado o seguinte trecho de código fonte:

```
buscar_submissoes = Thread(target=BuscarSubmissoes)
buscar_submissoes.start()
```

### 3.2.2.5 Banco de dados

O banco de dados escolhido para o *software* foi o PostgreSQL, ele é executado dentro do servidor, ou seja em um máquina separada de onde é executado o *backend*. Para conectar-se ao banco de dados a biblioteca utilizada foi a *psycopg2*. O PostgreSQL como anteriormente mencionado nesse documento, permite que seu protocolo utilize duas formas distintas de transporte de dados, são elas: Unix Sockets e TCP/IP.

Por estar rodando em uma máquina diferente do banco de dados, o *backend* utiliza o TCP/IP. Foi criada uma classe exclusiva para a execução de todas as ações vinculadas ao banco de dados.

---

```
class Database():
    def __init__(self, banco, usuario, senha):
        try:
            self.connection = psycopg2.connect(
                "dbname='meu_banco' user='meu_usuario' host='localhost' password='minha_senha' port='5432'"
            )
            self.connection.autocommit = True
            self.cursor = self.connection.cursor()
            print("Conexao com o banco de dados efetuada com sucesso")
        except:
            print("Erro ao conectar no database!")

    def query(self, sql):
        self.cursor.execute(sql)
        registros = self.cursor.fetchall()
        return registros

    def execute(self, sql):
        self.cursor.execute(sql)
```

---

Como já mencionado em outros capítulos do documento, é no fluxo de *startup* que é definida a forma de transporte de dados, na classe acima é passado o parâmetro de *host* e a porta que será utilizada para comunicar-se com o banco instalado no servidor.

### 3.2.3 Fechamento

Basicamente, o *backend* tem a função de compilar, executar, comparar os resultados e enviar todas essas informações para o banco de dados. Para que isso ocorra, todos os elementos acima descritos foram unidos em um mesmo fluxo de execução. O *backend* funciona de forma independente, ou seja ele não é acoplado ao *frontend*.

Para ele funcionar corretamente é necessário que o *frontend* siga alguns padrões esperados como, por exemplo, enviar registros ao banco de dados com o status = 'Processando', com o *id* da linguagem de programação utilizada no arquivo e referente a qual exercício é, por fim criar arquivos cujo nome é 'file<id do banco de dados>.<extensão da linguagem>' na pasta que o *backend* tem acesso.

## 4 REPLICAÇÃO DO SISTEMA

### 4.1 GlusterFS

A implementação do sistema pode ser feita com máquinas físicas ou utilizando máquinas virtuais, como será exemplificado nesta seção. A instalação de alguns componentes pode necessitar de conhecimento não descrito neste documento, junto a programas e o sistema operacional que será usado.

O sistema foi implementado em quatro máquinas virtuais: dois servidores (srv1 e srv2) que servirão para criar o bloco de armazenamento do *gluster* e para a utilização do banco de dados; duas máquinas cliente (cli1 e cli2), uma para a utilização do sistema de submissão de códigos e uma para executar o processo de compilação dos códigos já enviados.

#### 4.1.1 Instalando o sistema operacional

Para ambas as máquinas, cliente e servidor, foi utilizada a distribuição Ubuntu do sistema operacional Linux, nas versões 16.04 e 18.04, respectivamente. Será necessário que todas possuam 2 placas de rede, uma para conexão à internet e uma para a rede interna.

A instalação do sistema operacional nas máquinas clientes necessita apenas da configuração e instalação básica e deve ser configurada a placa de rede que será utilizada na rede interna. A máquina cli1 deve receber o Ipv4 estático 192.168.5.10. A máquina cli2 deve receber o Ipv4 estático 192.168.5.11.

As máquinas que serão os servidores necessitam possuir: uma partição para o sistema operacional e outra para o *gluster*, para evitar erros de sincronização, ou dois discos rígidos que terão uma partição cada. Nesta reprodução do sistema, será utilizado dois discos rígidos, um com 20GB de espaço para o sistema operacional e um disco de 500MB para o bloco do *gluster*. A instalação necessita apenas das configurações básicas e apenas deverá ser atribuído um ip estático para as placas da rede interna. A máquina srv1 deve receber o Ipv4 192.168.5.2 e a máquina srv2 deve receber o Ipv4 192.168.5.3.

#### 4.1.2 Configurando os servidores

A configuração dos servidores será feita por linha de comando, pois a versão *server* da distribuição Ubuntu não possui interface gráfica. Todos os comandos realizados devem ser feitos pelo usuário *root(su)* ou devem iniciar com a palavra **sudo**, que fará o usuário *root* executar o comando.

**Passo 1.** Deve-se atribuir nomes às máquinas na rede para serem encontradas pelo *gluster*. O comando abaixo deve ser executado em ambas máquinas servidor:

```
nano /etc/hosts
```

As seguintes linhas devem ser adicionadas ao final do arquivo. Após isso, deve-se salvá-lo e fechá-lo. Para testar se a modificação foi concluída, é necessário utilizar o comando **ping** e trocar o ip pelo nome atribuído.

```
192.168.5.2 srv1
192.168.5.3 srv2
192.168.5.10 cli1
192.168.5.11 cli2
```

**Passo 2.** Para que não ocorram problemas na transmissão de dados entre as máquinas servidor, é necessário digitar o comando abaixo, que liberará o tráfego de dados a partir do ip digitado.

- Para a máquina srv1:

```
iptables -I INPUT -p all -s 192.168.5.3 -j ACCEPT
```

- Para a máquina srv2:

```
iptables -I INPUT -p all -s 192.168.5.2 -j ACCEPT
```

**Passo 3.** Deve-se adicionar o repositório do *GlusterFS* para, posteriormente, instalá-lo. O banco de dados *PostgreSQL* também deve ser acrescentado. Os comandos abaixo devem ser realizados nas duas máquinas, exceto pela instalação do *PostgreSQL*, que deve ser executado apenas na máquina srv1.

```
add-apt-repository ppa:gluster/glusterfs-3.12 (Apertar ENTER para aceitar)
apt update
apt install glusterfs-server (Apertar 'y' para aceitar)
apt install postgresql postgresql-contrib
```

**Passo 4.** Para montar os volumes do *GlusterFS* deverá existir outra partição no sistema. Para isso, deverá ser montado um volume em outro disco. Os comandos abaixo devem ser executados em ambas máquinas, considerando que há dois discos disponíveis e o segundo seja referenciado por */dev/sdb*. Isso fará com que um novo volume seja criado no caminho *'/gluster'*.

```
fdisk /dev/sdb
mkfs.ext4 /dev/sdb1
mkdir -p /gluster/
mount /dev/sdb1 /gluster
mkdir /gluster/shared
```

**Passo 5.** Deve-se criar uma ligação entre ambas máquinas servidor para a criação do volume replicado do *gluster*. O comando abaixo deve ser apenas executado na máquina srv1 para criar a ligação.

```
gluster peer probe srv2
```

Para verificar se o comando foi executado com sucesso, os comandos abaixo podem ser executados em ambas as máquinas para ver a situação da ligação e a lista de máquinas.

```
gluster peer status
gluster pool list
```

**Passo 6.** Assim que todos os passos forem realizados, será possível criar o volume distribuído entre os servidores. Os comandos abaixo devem ser realizados apenas na máquina *srv1*, onde criará o volume e o iniciará.

```
gluster volume create gv1 replica 2 srv1:/gluster/shared
srv2:/gluster/shared
gluster volume start gv1
```

- Para verificar a situação do volume é possível executar o comando abaixo:

```
gluster volume info
```

Após a execução de todos os passos acima, o volume deve estar pronto para uso, é necessário que um cliente conecte e monte um acesso ao volume para que a sincronização de arquivos ocorra. Há a possibilidade de montar um acesso ao volume no próprio servidor caso o administrador queira testar as configurações.

**Passo 7.** Com a instalação do banco de dados realizada, é necessário que você defina uma nova senha para o usuário *postgres* e crie a base de dados que será utilizada pelo sistema. Portanto, ainda com o terminal aberto, execute estes comandos:

```
sudo -i -u postgres
psql
ALTER USER postgres PASSWORD 'nova-senha';
\q
createdb -T template0 simple_judge_system
exit
```

#### 4.1.3 Configurando os clientes

A configuração dos clientes será feita por linha de comando, apesar da versão *client* da distribuição Ubuntu possuir interface gráfica. Todos os comandos realizados devem ser feitos pelo usuário *root(su)* ou devem iniciar com a palavra **sudo**, que fará o usuário *root* executar o comando.

**Passo 1.** Deve-se adicionar o repositório do *GlusterFS* e instalá-lo. Os comandos abaixo devem ser realizados nas duas máquinas cliente.

```
add-apt-repository ppa:gluster/glusterfs-3.12 (Apertar ENTER para aceitar)
```

```
apt update
```

```
apt install glusterfs-client (Apertar 'y' para aceitar)
```

**Passo 2.** Deve-se montar o volume que está localizado na máquina `srv1`. O comando abaixo deve ser realizado para montar um volume na pasta designada para que a sincronização entre as máquinas funcione.

- Executar na máquina `cli1`:

```
mount -t glusterfs srv1:/gv1 /Documentos/ (?)
```

- Executar na máquina `cli2`:

```
mount -t glusterfs srv1:/gv1 /opt/simplejudgesystem
```

Para testar se o volume foi montado é possível executar o comando abaixo, se houver um volume apontado para o caminho `srv1:/gv1`, então não houve problemas nesta etapa.

```
df -h
```

Para realizar testes e verificar o funcionamento do *gluster*, arquivos podem ser criados ou enviados para os diretórios onde os volumes foram criados. Caso um arquivo for criado ou enviado para este diretório do volume, todas as outras máquinas com acesso ao volume devem possuir o mesmo arquivo. Caso uma máquina seja religada, os arquivos devem ser sincronizados após a inicialização do servidor do *GlusterFS*, se sua inicialização não estiver definida como automática.

## 4.2 Frontend

Esta seção do documento é destinada especificamente para que, mesmo uma pessoa que não conheça este projeto e não conheça profundamente as tecnologias utilizadas, consiga seguir os passos descritos aqui e ter a parte visual (*frontend*) do sistema **Simple Judge System** funcionando localmente em sua máquina.

Vale ressaltar que este tutorial considera que o usuário esteja fazendo uso da distribuição **Linux: Ubuntu 16.04.5 LTS (Xenial Xerus)**. Como a aplicação armazena informações criadas durante o uso do sistema, também é preciso que você possua o banco de dados **PostgreSQL** instalado.

Além disso, é necessário possuir o **PHP** instalado na versão *5.6 ou maior*, o **Laravel Framework** sendo executado na versão *5.4.36 ou maior* e o **Composer** rodando na versão *1.5.1 ou maior*. Caso algum problema ou dificuldade seja encontrado, por favor, acesse o link que possa lhe ajudar: **Laravel**, **PHP** ou **Composer**.

Por fim, para que seja dado início aos poucos passos necessários para executar uma aplicação *PHP + Laravel*, é necessário ter o **GIT** instalado (pois o projeto já existente será



clonado para a máquina do usuário). Então, caso o *GIT* não esteja instalado na sua máquina, considere executar o seguinte comando no terminal de seu equipamento:

```
sudo apt-get install git
```

Apenas para confirmar, assim que o comando digitado acima encerrar seu processamento, execute:

```
git --version
```

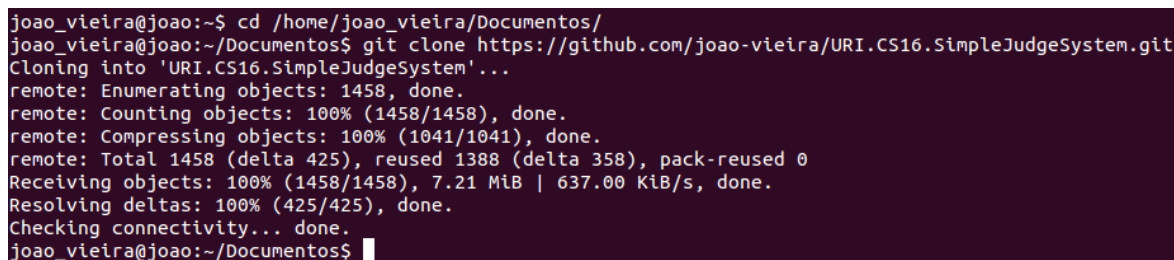
**obs:** note que antes da palavra *version* existem **dois** sinais de menos, sem espaço

Esse comando irá informar a versão deste sistema de controle de versão de arquivos que foi instalado na sua máquina. Feito isso, está na hora de obter a parte *frontend* deste projeto. Para fazer isso, navegue até a pasta na qual você deseja armazenar o novo sistema e execute o comando que segue abaixo, pois ele irá "copiar" o projeto de um repositório disponível no *GitHub* para o diretório que você se encontra neste momento.

```
cd /<caminho-de-sua-preferencia>
git clone https://github.com/joao-vieira/URI.CS16.SimpleJudgeSystem.git
```

Para facilitar a conferência, veja a imagem abaixo, que representa os passos descritos anteriormente. Confira se o resultado obtido após os **dois** comandos digitados é o mesmo.

Figura 14 – Passos para obter a aplicação localmente



```
joao_vieira@joao:~$ cd /home/joao_vieira/Documentos/
joao_vieira@joao:~/Documentos$ git clone https://github.com/joao-vieira/URI.CS16.SimpleJudgeSystem.git
Cloning into 'URI.CS16.SimpleJudgeSystem'...
remote: Enumerating objects: 1458, done.
remote: Counting objects: 100% (1458/1458), done.
remote: Compressing objects: 100% (1041/1041), done.
remote: Total 1458 (delta 425), reused 1388 (delta 358), pack-reused 0
Receiving objects: 100% (1458/1458), 7.21 MiB | 637.00 KiB/s, done.
Resolving deltas: 100% (425/425), done.
Checking connectivity... done.
joao_vieira@joao:~/Documentos$
```

Fonte: Própria (2018)

Se tudo ocorreu bem, a aplicação já está armazenada na sua máquina. Agora vamos instalar as dependências do projeto para que ele possa ser executado. Acesse a pasta que acabou de ser *clonada* do *GitHub* com este comando:

```
cd URI.CS16.SimpleJudgeSystem/
```

Agora você está dentro da pasta do projeto **Simple Judge System**. Para instalar as dependências citadas anteriormente, execute o comando exibido abaixo. Atenção: este comando pode demorar alguns segundos para acabar de ser executado!

```
composer install
```

Assim que este comando for digitado e iniciar seu processamento, você deve visualizar uma tela similar à ilustração retratada abaixo:

Figura 15 – Instalando dependências da aplicação

```
joao_vieira@joao:~/Documentos$ cd URI.CS16.SimpleJudgeSystem/
joao_vieira@joao:~/Documentos/URI.CS16.SimpleJudgeSystem$ composer install
Loading composer repositories with package information
Installing dependencies (including require-dev) from lock file
Package operations: 61 installs, 0 updates, 0 removals
- Installing doctrine/inflector (v1.1.0): Loading from cache
- Installing erusev/parsedown (1.7.1): Downloading (100%)
- Installing jakub-onderka/php-console-color (v0.2): Downloading (100%)
- Installing symfony/polyfill-mbstring (v1.9.0): Downloading (100%)
- Installing symfony/var-dumper (v3.4.17): Downloading (100%)
- Installing psr/log (1.0.2): Loading from cache
- Installing symfony/debug (v3.4.17): Downloading (100%)
- Installing symfony/console (v3.4.17): Downloading (100%)
- Installing nikic/php-parser (v3.1.5): Downloading (100%)
- Installing jakub-onderka/php-console-highlighter (v0.3.2): Loading from cache
- Installing dnoegel/php-xdg-base-dir (0.1): Loading from cache
- Installing psy/psysh (v0.9.8): Downloading (100%)
- Installing vlucas/phpdotenv (v2.5.1): Downloading (100%)
- Installing symfony/css-selector (v3.4.17): Downloading (100%)
- Installing tijsverkoyen/css-to-inline-styles (2.2.1): Downloading (100%)
- Installing symfony/routing (v3.4.17): Downloading (100%)
- Installing symfony/process (v3.4.17): Downloading (100%)
- Installing symfony/polyfill-ctype (v1.9.0): Downloading (100%)
- Installing paragonie/random_compat (v2.0.17): Downloading (100%)
- Installing symfony/polyfill-php70 (v1.9.0): Downloading (100%)
- Installing symfony/http-foundation (v3.4.17): Downloading (100%)
- Installing symfony/event-dispatcher (v3.4.17): Downloading (100%)
- Installing symfony/http-kernel (v3.4.17): Downloading (100%)
- Installing symfony/finder (v3.4.17): Downloading (100%)
- Installing swiftmailer/swiftmailer (v5.4.12): Downloading (100%)
- Installing ramsey/uuid (3.8.0): Downloading (100%)
- Installing symfony/translation (v3.4.17): Downloading (100%)
- Installing nesbot/carbon (1.34.0): Downloading (100%)
- Installing mtdowling/cron-expression (v1.2.1): Loading from cache
- Installing monolog/monolog (1.23.0): Loading from cache
- Installing league/flysystem (1.0.47): Downloading (100%)
- Installing laravel/framework (v5.4.36): Loading from cache
- Installing laravel/tinker (v1.0.8): Downloading (100%)
- Installing fzaninotto/faker (v1.8.0): Downloading (100%)
- Installing hamcrest/hamcrest-php (v1.2.2): Loading from cache
- Installing mockery/mockery (0.9.9): Loading from cache
```

Fonte: Própria (2018)

Aguarde até que o processamento termine. Assim que o *composer* acabar de instalar as dependências, seu terminal pode voltar a ser utilizado normalmente. Note que as últimas mensagens exibidas no terminal, em virtude da instalação das dependências, devem ser semelhantes a esse trecho:

```
[...]
Generating optimized autoload files
> Illuminate\Foundation\ComposerScripts::postInstall
> php artisan optimize
Generating optimized class loader
The compiled services file has been removed.
```

Estamos quase acabando a configuração! Agora, é necessário informar o usuário e senha do *PostgreSQL* para que a aplicação consiga se comunicar com a base de dados. No *Laravel Framework*, existe um arquivo de configuração que é utilizado para este tipo de situação e, por isso, é dentro deste arquivo que deve ser informado o usuário e senha do banco de dados local. Se você ainda estiver com o terminal aberto dentro do projeto clonado, basta executar (considera-se que o aplicativo *nano* está instalado):

```
nano .env
```

Quando este comando for digitado, essa tela será exibida:

Figura 16 – Editando arquivo de configuração com *nano*

```

joao_vieira@joao: ~/Documentos/URI.CS16.SimpleJudgeSystem
GNU nano 2.2.6 Arquivo: .env

APP_NAME="Simple Judge System"
APP_ENV=production
APP_KEY=base64:viRvG7xTt/zhrbkaXqZn4MZLdyrdJz/AtzQnF0rMqgo=
APP_DEBUG=false
APP_LOG_LEVEL=debug
APP_URL=http://localhost

DB_CONNECTION=pgsql
DB_HOST=localhost
DB_PORT=5432
DB_DATABASE=simple_judge_system
DB_USERNAME=usuario-do-banco-de-dados-aqui
DB_PASSWORD=senha-do-banco-de-dados-aqui

BROADCAST_DRIVER=log
CACHE_DRIVER=file
SESSION_DRIVER=file
QUEUE_DRIVER=sync

REDIS_HOST=127.0.0.1
REDIS_PASSWORD=null
REDIS_PORT=6379

MAIL_DRIVER=smt
MAIL_HOST=smt
MAIL_PORT=2525
MAIL_USERNAME=null
MAIL_PASSWORD=null
MAIL_ENCRYPTION=null

PUSHER_APP_ID=
PUSHER_APP_KEY=
PUSHER_APP_SECRET=

33 linhas lidas
Obter Ajuda  Gravar  Ler o arq  Pág anter  Recort txt  Pos atual
Sair  Justificar  Onde está?  Próx pag  Colar Txt  VerFórtog

```

Fonte: Própria (2018)

Altere as seguintes linhas:

- **DB\_USERNAME** => coloque postgres
- **DB\_PASSWORD** => coloque a senha escolhida no início deste capítulo, quando o *PostgreSQL* foi instalado

Feito isso, resta apenas gerar uma nova *chave* para a aplicação e criar as tabelas necessárias para as ações executadas no sistema. Além disso, vamos dar permissão para a pasta do projeto (isso vai evitar que alguns arquivos fiquem com permissão de somente leitura). Todas essas ações podem ser executadas com estes comandos:

```

sudo chmod -R 777 ../URI.CS16.SimpleJudgeSystem/
sudo php artisan migrate:refresh --seed
sudo php artisan key:generate

```

**Parabéns! A sua aplicação está pronta para ser executada.** Vá até o capítulo 5 para encontrar um tutorial de como usar a aplicação.

## 4.3 Backend

### 4.3.1 Prefácio

Neste capítulo estará descrito o passo a passo para executar o *backend* do *software Simple Judge System*. Ele resume-se em instalar alguns componentes auxiliares para que o

Python consiga ler e gravar de um banco de dados e instalar alguns compiladores que serão executados posteriormente através de *scripts*.

### 4.3.2 Recomendações

O *backend* foi desenvolvido exclusivamente para Linux e é executado em uma máquina virtual, por conta disso abaixo estão algumas recomendações que podem ser úteis aos usuários que queiram replicar o projeto com mais facilidade.

#### 4.3.2.1 Distribuição

A distribuição utilizada para executar o *backend* é o Ubuntu 16.04.5 LTS (Xenial Xerus), nessa distribuição damos a garantia de que já virá dois importantes recursos utilizados no projeto, são eles:

- **G++ Compilador**
- **Python (versão 2.7.12)**

Neste **link** pode-se encontrar a ISO dessa distribuição disponível para *download*.

### 4.3.3 Instalando os compiladores

Vale ressaltar que, caso você esteja **utilizando a distribuição recomendada**, as duas primeiras seções a seguir devem ser ignoradas.

#### 4.3.3.1 **G++**

1. Abra o terminal do Ubuntu (CTRL + ALT + T) ou localize-o pela barra de busca.
2. Digite o seguinte comando comando:
3. Informe sua senha de super usuário.
4. Verifique se o g++ foi instalado com sucesso utilizando o seguinte comando:

```
g++ -version
```

5. O resultado deve ser semelhante a isso:

```
g++ (Ubuntu 5.4.0-6ubuntu1~16.04.10) 5.4.0 20160609
Copyright (C) 2015 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

#### 4.3.3.2 **Python**

1. Abra o terminal do Ubuntu (CTRL + ALT + T) ou localize-o pela barra de busca.
2. Digite o seguinte comando comando:

```
sudo apt-get install python2.7
```

3. Informe sua senha de super usuário.
4. Verifique se o python2.7 foi instalado com sucesso utilizando o seguinte comando:  

```
python -version
```
5. O resultado deve ser semelhante a isso:  

```
Python 2.7.12
```
6. Precisamos instalar o gerenciador de pacotes pip, digite no terminal:  

```
sudo apt-get install python-pip
```
7. Verifique se o pip foi instalado com sucesso utilizando o seguinte comando:  

```
pip -version
```
8. O resultado deve ser semelhante a isso:  

```
pip 8.1.1 from /usr/lib/python2.7/dist-packages (python 2.7)
```
9. Para instalar a biblioteca de conexão com o banco de dados, digite no terminal:  

```
sudo pip install psycopg2
```
10. **Nota:** Esse último passo poderá ser verificado se obteve sucesso quando executarmos o *software*.

#### 4.3.3.3 *Java*

1. Abra o terminal do Ubuntu (CTRL + ALT + T) ou localize-o pela barra de busca.
2. Digite o seguinte comando comando:  

```
sudo add-apt-repository ppa:webupd8team/java
```
3. Informe sua senha de super usuário.
4. Atualize o APT com esse comando:  

```
sudo apt-get update
```
5. Execute o comando de instalação do java:  

```
sudo apt-get install oracle-java8-installer
```
6. Verifique se o Java foi instalado, digite o seguinte comando:  

```
javac -version
```
7. O resultado deve ser semelhante a isso:  

```
javac 1.8.0_181
```

#### 4.3.3.4 *Projeto*

Você pode fazer o download do projeto no seguinte repositório do GitHub:

**<https://github.com/viniciusandd/compilador-multilinguagens>**

#### 4.3.3.5 *Execução*

Se todos os passos acima foram feitos com sucesso, você pode executar o arquivo que está no seguinte caminho:

```
cd Documentos/Python/compilador-multilinguagens/app/
```

```
sudo python compilador_redes_so.py
```

Informe as entradas necessárias ao *software* e ele retornará em primeira mão se a conexão com o banco de dados foi estabelecida com sucesso. Caso ocorrer problemas, revise o passo a passo de configuração do servidor. Após executar a primeira vez com sucesso você pode salvar em um arquivo suas configurações, supondo que seu arquivo seja config.txt, execute na próxima vez o projeto da seguinte maneira.

```
sudo python compilador_redes_so.py < config.txt
```

Logo em seguida, a aplicação já retornará se ocorreu sucesso ao conectar com o banco de dados e começará a compilar todos arquivos que forem criados pelo projeto do *frontend*.

## 5 MANUAL DE USO

Levando em conta que o usuário seguiu os passos e instruções listados no capítulo anterior, este capítulo irá demonstrar, através de imagens e explicações, como utilizar o *frontend* do sistema, ou seja, as telas feitas especificamente para que o usuário pudesse interagir, de maneira amigável, com o projeto desenvolvido.

Basicamente, o sistema *web* que será acessado é um *sistema simples de avaliação*, que possibilita que o usuário envie suas soluções para os problemas inseridos dentro da plataforma em várias linguagens diferentes. Essa ideia de uma aplicação que avalia códigos escritos com base em uma saída esperada é bastante interessante para indivíduos que desejam aprimorar ou desafiar suas capacidades em uma linguagem, pois resolver problemas de programação é uma das formas mais indicadas para tal.

Nas seções a seguir, será demonstrado um tutorial de uso completo do sistema, com explicações detalhadas para cada uma das ações, com objetivo de que, caso alguma pessoa que tenha acesso a esse documento deseje reutilizar esse projeto, ela consiga utilizá-lo sem nenhuma grande dificuldade.

Após ter replicado o sistema, consequentemente, este tutorial considera que você gostaria, pelo menos, de ter acesso ao código ou, quem sabe, até alterá-lo. Por isso, todas as instruções que serão explanadas aqui consideram que você esteja em um ambiente, *Linux*, de *desenvolvimento* do sistema. Isso tornará a experiência mais prática, tendo em vista que não será preciso nenhum tipo de etapa de *deploy*.

- **Passo 1:** Abra seu terminal digitando as teclas: *Ctrl + Alt + T* (ou procure na barra de pesquisa do sistema).
- **Passo 2:** Navegue até a pasta que você clonou do projeto utilizando o comando *cd*:  
`cd /caminho-completo-para-a-pasta-clonada-do-sistema`
- **Passo 3:** Após entrar na pasta, digite o seguinte comando:  
`sudo php artisan serve`
- **Passo 4:** Sua senha de usuário será solicitada. Coloque sua senha corretamente e, então, uma mensagem idêntica à imagem abaixo deve ter sido exibida:

Figura 17 – Sequência de passos para *levantar* o servidor da aplicação

```
joao_vieira@joao:~$ cd /var/www/FACULDADE/SimpleJudgeSystem/
joao_vieira@joao:/var/www/FACULDADE/SimpleJudgeSystem$ sudo php artisan serve
[sudo] password for joao_vieira:
Laravel development server started: <http://127.0.0.1:8000>
```

Fonte: Própria (2018)

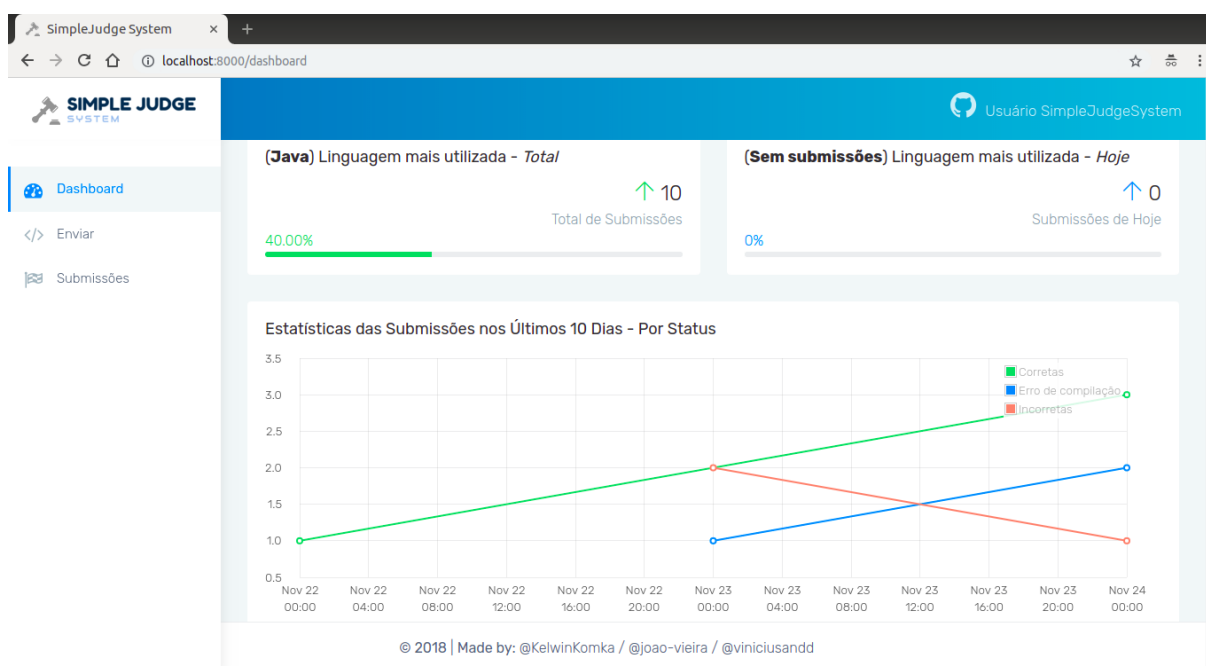
## 5.1 Dashboard

Feito isso, já é possível acessar o sistema em um *browser* de sua preferência. Ao abrir o navegador, digite este endereço:

`http://localhost:8000/dashboard`

Com isso, a seguinte tela deverá ser exibida:

Figura 18 – Tela inicial do sistema



Fonte: Própria (2018)

Esta é a tela inicial do sistema, utilizada para dar *boas-vindas* ao usuário. Nesta tela existem três componentes principais:

- **Canto superior esquerdo:** este painel irá apresentar a quantia de submissões realizadas dentro da plataforma desde a sua instalação. Além disso, antes da frase *Linguagem mais utilizada - Total*, entre parênteses, pode ser visualizada a linguagem que mais foi utilizada dentre todas as submissões já feitas no sistema e, na barra de porcentagem abaixo, quanto as submissões feitas nesta linguagem representam perante os 100% de submissões totais do sistema.
- **Canto superior direito:** este painel tem uma função bem semelhante ao descrito no item anterior. No entanto, os dados que, no painel da esquerda, se aplicam a todas as submissões do sistema, no painel da direita correspondem às submissões do dia atual. Isso quer dizer que, se no dia de hoje nenhuma submissão tiver sido realizada, ao invés de mostrar a porcentagem de uma linguagem específica e seu nome, o painel exibirá *Sem submissões* e a barra de contagem estará em 0%.

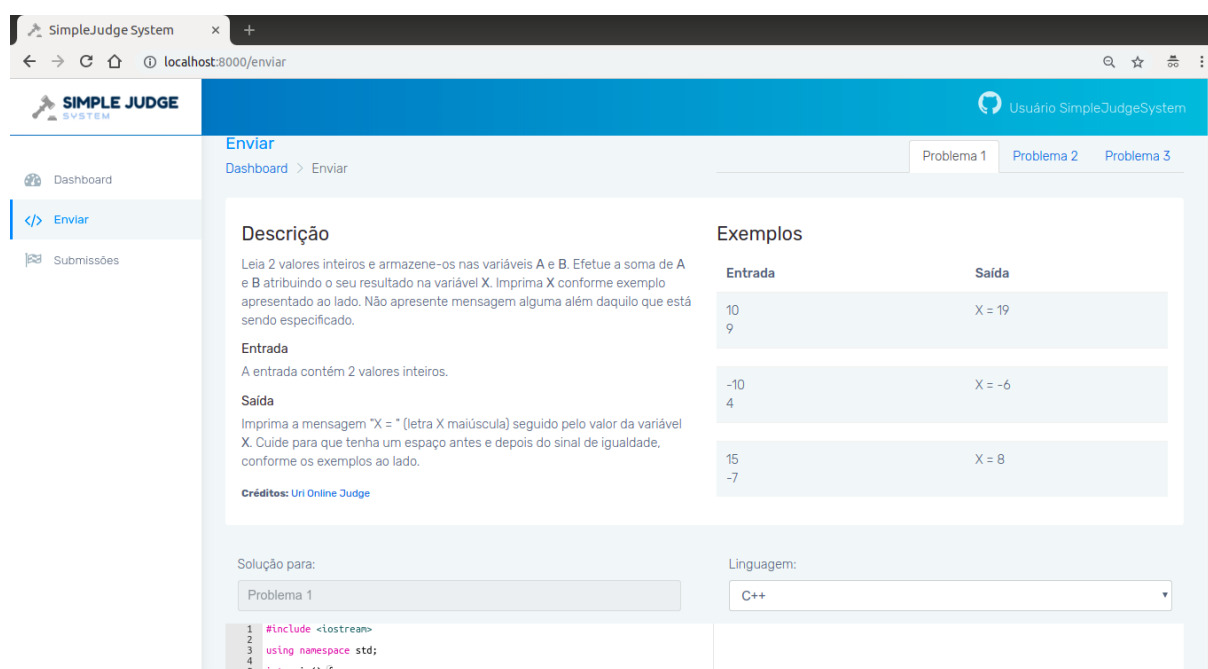


- **Centro:** logo abaixo dos dois painéis, estará sendo exibido um gráfico que apresenta um breve relatório das submissões realizadas nos últimos 10 dias quanto a seu *status* após processamento. Vale lembrar que os *status* disponíveis dentro do sistema, após a submissão ter sido processada, são: *Correta*, *Incorreta* e *Erro de compilação*.

## 5.2 Enviar

Ao clicar no **menu *Enviar***, disponível na barra lateral esquerda, você irá se deparar com esta tela:

Figura 19 – Tela *Enviar*



Esta tela está dividida horizontalmente em duas partes: a parte superior, na qual pode-se escolher um dos 3 problemas disponíveis no sistema (**Problema 1**, **Problema 2** e **Problema 3**), ler sua descrição, ver exemplos de entrada e saída e o local do qual este problema foi retirado (créditos); e a parte inferior, onde é possível escolher entre as 3 linguagens disponíveis no sistema (**C++**, **Python** e **Java**) para criar uma solução para o problema selecionado e enviar a submissão, clicando no botão **Enviar**, que pode ser encontrado logo após ao campo onde o código foi digitado.

Para alternar entre os problemas disponíveis até encontrar o que mais lhe agrade, você deve usar as *tabs* disponíveis na parte superior da página. Veja na imagem abaixo:

Figura 20 – Escolhendo um problema para resolver



Fonte: Própria (2018)

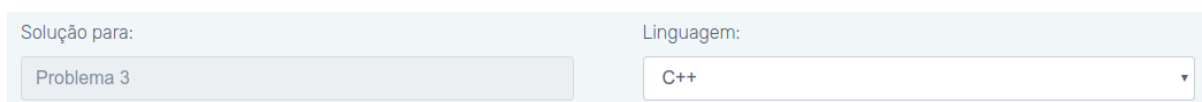
Conforme a *tab* escolhida, o *box* localizado logo abaixo, que contém informações como a descrição do problema e alguns exemplos de entrada e saída esperados, terá seu conteúdo alterado. Na imagem abaixo, a *tab*: **Problema 3** foi escolhida. Veja como ficou o *box*:

Figura 21 – Conteúdo do *box* alterado

Fonte: Própria (2018)

Agora, resta apenas demonstrar como escolher entre as linguagens de programação disponíveis para que você esteja apto a criar sua solução para o problema selecionado. Logo abaixo ao *box* de informações do problema escolhido, dois campos são exibidos: **Solução para**, que informa instantaneamente qual é o problema selecionado atualmente; e **Linguagem**, que possibilita que você altere a linguagem na qual a solução será escrita. Estes dois campos são exibidos na figura a seguir:

Figura 22 – Campos referentes à solução do problema



Fonte: Própria (2018)

Por fim, para facilitar o uso do sistema por parte de pessoas que estão iniciando em programação, toda vez que o campo *Linguagem* for alterado, o sistema preencherá o campo disponível para que o código seja escrito com o **template básico** da linguagem escolhida. Isso permite que o usuário se preocupe apenas com a lógica para resolver o problema definido. Essa situação está demonstrada na imagem abaixo:

Figura 23 – Template do código alterado conforme a linguagem escolhida

Solução para: Problema 3

Linguagem: Java

```
1 public class Main {
2 public static void main(String[] args) {
3 /** Digite seu código aqui */
4 }
5 }
```

Fonte: Própria (2018)

Após escrever seu código, basta clicar no botão **Enviar** para que sua solução seja submetida. Se tudo ocorreu bem, a mensagem ilustrada na imagem a seguir será exibida, questionando se você deseja realmente submeter seu código neste momento. Ao aceitar, uma mensagem de confirmação será mostrada.

Figura 24 – Mensagem para confirmação da submissão

?

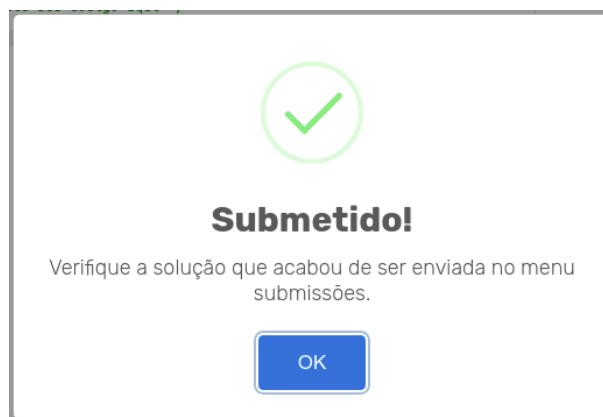
**Submeter código?**

Certifique-se de ter revisado e não ter deixado freopen no arquivo! =)

Sim, submeter! Cancelar

Fonte: Própria (2018)

Figura 25 – Mensagem confirmando o envio



Fonte: Própria (2018)

### 5.3 Submissões

Quando você clicar em *OK* na mensagem ilustrada na imagem anterior, você será redirecionado para a tela de **Submissões**, que é uma tela para simples conferência das submissões já realizadas. Nesta tela, haverá uma tabela listando todas as submissões feitas, juntamente com algumas informações específicas, tais como: **Status**, **Resposta** e **Linguagem Utilizada**. A imagem a seguir demonstra esta tela:

Figura 26 – Tela *Submissões*

**Submissões Enviadas**  
Todas as submissões já realizadas podem ser encontradas na tabela abaixo.

#	Status	Resposta	Linguagem Utilizada
1	Correta	Solução correta.	C++
2	Correta	Solução correta.	Python
3	Correta	Solução correta.	Java
4	Correta	Solução correta.	Java
5	Erro de compilação	Solução apresentou problemas ao ser compilada. Verifique sua sintaxe!	C++
6	Erro de compilação	Solução apresentou problemas ao ser compilada. Verifique sua sintaxe!	Python
7	Erro de compilação	Solução apresentou problemas ao ser compilada. Verifique sua sintaxe!	Java
8	Incorreta	Sua sintaxe está correta. Porém, a solução não apresentou os resultados esperados.	C++
9	Incorreta	Sua sintaxe está correta. Porém, a solução não apresentou os resultados esperados.	Python
10	Incorreta	Sua sintaxe está correta. Porém, a solução não apresentou os resultados esperados.	Java

Fonte: Própria (2018)

## 6 CONCLUSÃO

O desenvolvimento de *software* sempre foi e sempre será uma tarefa complexa, ela exige muito dos profissionais que à exercem. Um dos fatores para tal afirmação é a constante evolução da tecnologia, sempre inovando e permitindo que os desenvolvedores reinventem seus antigos produtos e criem novos. Ou seja, temos uma área que evolui constantemente e que é extremamente favorável para que as pessoas utilizem o máximo de seu potencial criativo, o que por consequência torna ela um ambiente perfeito para um completo caos de informações e recursos. Para que isso não ocorra existem os protocolos, eles são uma série de regras e padrões que fazem com que as aplicações comuniquem-se ou ‘falem a mesma língua’ independentemente da tecnologia utilizada por trás da aplicação ou da plataforma em que ela está sendo executada.

Outro grande amigo dos desenvolvedores é o sistema operacional, ele fornece vários recursos e acaba isentando os desenvolvedores de inúmeras tarefas complexas, ou seja, saber extrair o máximo de recursos do sistema operacional é muito importante para todos profissionais da área de tecnologia. Bom, agora que os dois protagonistas do projeto (protocolo e sistema operacional) em questão foram apresentados, vamos aos resultados que obtivemos.

A implementação do projeto pode nos aproximar de recursos que são extremamente importantes em nosso dia a dia e que geralmente não nos preocupamos com eles, por exemplo, em nosso servidor selecionamos o *PostgreSQL* para ser nosso banco de dados, o mesmo utiliza um protocolo próprio para a comunicação de clientes e servidores, conhecido como protocolo *frontend/backend*. Por padrão esse protocolo utiliza a comunicação utilizando o TCP/IP, porém ele permite que os clientes e servidores comuniquem-se de outra maneira, utilizando *Unix Sockets*. Elas são muito mais velozes quando estamos utilizando apenas um host e por pertencerem ao sistema operacional fornecem uma série de recursos novos que o protocolo TCP/IP não consegue disponibilizar.

Nossa aplicação foi fragmentada em duas partes, o *frontend web* e o *backend*, que inclusive estão hospedados em máquinas distintas. Para a comunicação das duas, além do banco de dados, foi utilizado um *Gluster*. Ele isentou ambas as partes de implementarem dentro de seus projetos formas de comunicação direta entre elas, ou seja, um recurso disponibilizado pelo sistema operacional que roda em uma máquina distinta da do *frontend* e do *backend* fez com que ambos se comunicassem como se estivessem rodando em uma mesma máquina.

O projeto fez com que pensássemos de uma forma diferente para desenvolvê-lo, o que enriqueceu muito nossa capacidade de resolver problemas e buscar recursos já existentes no mercado. Outra importante menção é a de que ele fez com que nos aprofundássemos em todos os recursos que cogitamos utilizar, ou seja, não só utilizamos excelentes recursos como os utilizamos da melhor forma possível para que nosso projeto obtivesse sucesso em sua execução e tudo isso méritos a forma em que esse projeto foi desenhado, comprovando que a teoria aliada a prática é sempre a melhor forma de aprendizado.

## REFERÊNCIAS

- AKAMAI, W. **Turn-on HTTP/2 today!** 2017. Disponível em: <<https://http2.akamai.com/>>. Acesso em: 15 de novembro de 2018. Citado na página 4.
- BARRETT, D.; KING, T. **Redes de Computadores**. 1. ed. Rio de Janeiro: LTC, 2010. Citado na página 3.
- BERNARDINO, M. **O que é o TCP/IP e como funciona**. 2018. Disponível em: <<https://www.infonova.com.br/artigo/o-que-e-tcp-ip-e-como-funciona/>>. Acesso em: 20 de novembro de 2018. Citado 2 vezes nas páginas 13 e 14.
- BEZERRA, R. M. **A Camada de aplicação**. 2008. Disponível em: <<http://www2.ufba.br/~romildo/downloads/ifba/aplicacao.pdf>>. Acesso em: 14 de novembro de 2018. Citado na página 16.
- FIELDING, R. T. et al. **Hypertext Transfer Protocol – HTTP/1.1**. [S.l.], 1999. <<http://www.rfc-editor.org/rfc/rfc2616.txt>>. Disponível em: <<http://www.rfc-editor.org/rfc/rfc2616.txt>>. Acesso em: 23 de novembro de 2018. Citado na página 4.
- FOROUZAN, B. A.; MOSHARRAF, F. **Redes de computadores: uma abordagem top-down**. 1. ed. Porto Alegre: AMGH, 2013. Citado na página 8.
- GROUP, T. P. G. D. **Chapter 50. Frontend/Backend Protocol**. 2018. Disponível em: <<https://www.postgresql.org/docs/9.5/protocol.html>>. Acesso em: 16 de novembro de 2018. Citado na página 17.
- HANSON, T. D. **UNIX domain sockets**. 2017. Disponível em: <[https://troydhanson.github.io/network/Unix\\_domain\\_sockets.html](https://troydhanson.github.io/network/Unix_domain_sockets.html)>. Acesso em: 10 de novembro de 2018. Citado na página 22.
- LEOPOLDINO, C. **O Comando COPY**. 2007. Disponível em: <<http://postgresqlbr.blogspot.com/2007/07/o-comando-copy.html>>. Acesso em: 16 de novembro de 2018. Citado na página 22.
- MDN. **Uma visão geral do HTTP**. 2018. Disponível em: <<https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Overview>>. Acesso em: 25 de novembro de 2018. Citado 4 vezes nas páginas 5, 6, 7 e 8.
- MORAES, A. F. de. **Redes de computadores**. 1. ed. São Paulo: SARAIVA, 2014. Citado na página 3.
- PILLOU, J.-F. **O protocolo HTTP**. 2017. Disponível em: <<https://br.ccm.net/contents/266-o-protocolo-http>>. Acesso em: 24 de outubro de 2018. Citado na página 7.
- PILLOU, J.-F. **O que é o protocolo TCP/IP**. 2018. Disponível em: <<https://br.ccm.net/contents/285-o-que-e-o-protocolo-tcp-ip#o-que-significa-tcp-ip>>. Acesso em: 20 de novembro de 2018. Citado 2 vezes nas páginas 9 e 11.
- ROUSE, M. **What is GlusterFS (Gluster File System)?** 2017. Disponível em: <<https://searchstorage.techtarget.com/definition/GlusterFS-Gluster-File-System>>. Acesso em: 17 de novembro de 2018. Citado na página 2.

SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G. **Fundamentos de Sistemas Operacionais**. 9. ed. Rio de Janeiro: LTC, 2015. Citado na página 2.

SOUSA, L. B. de. **Redes de Computadores: Guia Total**. 1. ed. São Paulo: Editora Érica Ltda, 2013. Citado 6 vezes nas páginas 9, 10, 11, 13, 14 e 15.

TORRES, M. **Diferença entre os métodos GET e POST**. 2017. Disponível em: <<http://professortorres.com.br/diferenca-entre-os-metodos-get-e-post/>>. Acesso em: 29 de outubro de 2018. Citado na página 8.

VIEIRA, L. **HTTP/2: a história, detalhes, vantagens e benefícios do protocolo**. 2016. Disponível em: <<https://blog.apiki.com/2016/12/20/http2/>>. Acesso em: 10 de novembro de 2018. Citado 2 vezes nas páginas 4 e 5.

VIEIRA, N. **Entendendo um pouco mais sobre o protocolo HTTP**. 2007. Disponível em: <<https://nandovieira.com.br/entendendo-um-pouco-mais-sobre-o-protocolo-http>>. Acesso em: 25 de outubro de 2018. Citado na página 5.