ULBRA CENTRO UNIVERSITÁRIO LUTERANO DE PALMAS
Recredenciado pela Portaria Ministerial nº 3.607, de 17/10/05, D.O.U. nº 202, de 20/10/2005
ASSOCIAÇÃO EDUCACIONAL LUTERANA DO BRASIL

Jesiel Souza Padilha

CodeLive: Interatividade com o usuário em tempo-real e Interpretação e execução do código do usuário

Palmas - TO
2016

Jesiel Souza Padilha

CodeLive: Interatividade com o usuário em tempo-real e Interpretação e execução do código do usuário

Trabalho de Conclusão de Curso (TCC) elaborado e apresentado como requisito parcial para obtenção do título de bacharel em Ciência da Computação pelo Centro Universitário Luterano de Palmas (CEULP/ULBRA).

Orientador: Prof. Msc. Jackson Gomes de Souza.

Palmas - TO
2016

Jesiel Souza Padilha

CodeLive: Interatividade com o usuário em tempo-real e Interpretação e execução do código do usuário

Trabalho de Conclusão de Curso (TCC) elaborado e apresentado como requisito parcial para obtenção do título de bacharel em Ciência da Computação pelo Centro Universitário Luterano de Palmas (CEULP/ULBRA).

Orientador: Prof. Msc. Jackson Gomes de Souza.

Aprovada em: ____/____/____

BANCA EXAMINADORA

_____

Prof. M.Sc. Jackson Gomes de Souza

Centro Universitário Luterano de Palmas

_____

Prof. M.Sc. Fabiano Fagundes

Centro Universitário Luterano de Palmas

_____

Prof(a). M.Sc. Madianita Bogo Marioti

Centro Universitário Luterano de Palmas

Palmas – TO

2016

AGRADECIMENTOS

Primeiramente gostaria de agradecer aos meus familiares que mesmo distantes sempre estiveram me apoiando desde de a minha infância até agora, na vida acadêmica, sem eles sei que não chegaria até aqui.

No período de desenvolvimento deste trabalho recebi muitos incentivos, especialmente da minha amiga e namorada Carla Silva, esta sempre presente nos momentos de dificuldade e também vibrando a cada conquista feita por mim.

Dentro da universidade foram muitas as contribuições, visto que desde que cheguei na cidade e Palmas conheci e fiz muitos amigos, dentre estes gostaria de agradecer aos sempre presentes: Leomar Camargo, Ranyelson Carvalho, Marcus Vinicius Germano, Jhonatan Mota e João Neto, os conselhos, dicas e parcerias ao longo dessa jornada contribuíram não só na minha formação como estudante, mas também como pessoa.

Finalmente gostaria de agradecer a todos os professores do CEULP/ULBRA, em especial ao meu orientador Jackson Gomes que sempre deu dicas incríveis tanto nos trabalhos de estágio e TCC, quanto nas disciplinas ministradas por ele, estendo esse agradecimento também aos professores presentes na minha banca, Fabiano Fagundes e Madianita Bogo, muito da qualidade do presente trabalho e da minha evolução como estudante se deu por conta dos conselhos, apoio e da forma simples, porém bastante eficiente de ensinar destes dois.

RESUMO

PADILHA, Jesiel Souza. **CodeLive: Interatividade com o usuário em tempo-real e Interpretação e execução do código do usuário**. 2016. 59 f. TCC (Graduação) - Curso de Ciência da Computação, Centro Universitário Luterano de Palmas, Palmas, 2016.

Afim de usufruir dos benefícios disponibilizados pelo protocolo *WebSocket*, como por exemplo, o fornecimento de um canal de comunicação bidirecional em tempo-real, o presente trabalho visa atualizar a aplicação CodeLive - um ambiente gamificado para aprendizado de programação, e com isso, proporcionar um ambiente mais interativo para o usuário. Também faz parte do trabalho a integração de um interpretador Python para executar o código do usuário diretamente no browser, que tornará a tarefa de validar um desafio muito mais simples. Espera-se que com a atualização da ferramenta, o usuário se sinta mais estimulado a praticar programação por um tempo maior levando o mesmo a um conhecimento mais solido.

PALAVRAS-CHAVE: Comunicação em tempo real, WebSocket, Execução de código, Programação.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

API - Application Programming Interface

HTML - HyperText Markup Language

HTTP - Hypertext Transfer Protocol

TCP - Transmission Control Protocol

UDP - User Datagram Protocol

TSL – Transport Layer Security

SSL – Security Socket Layer

DOM – Document Object Model

SUMÁRIO

1 INTRODUÇÃO

Utilizando os elementos da gamificação e do universo da saga *Star Wars* como metáfora, Cardoso (2015) concebeu um *software* web que tem como proposta motivar o usuário a aprender e praticar programação utilizando o conceito de gamificação. Tal objetivo foi alcançado por meio de recompensas que o usuário ganha a cada desafio vencido. Os desafios são exercícios de programação, logo, resolvê-los aumenta o conhecimento do usuário e faz com que o mesmo avance dentro do jogo (por exemplo, subindo de nível).

Assim como a outros *softwares* web, o CodeLive faz uso do protocolo HTTP para realizar a comunicação entre cliente e servidor. O HTTP é um protocolo de requisição-resposta (*request/response*) para o modelo cliente-servidor. Porém, o protocolo HTTP não permite que servidores iniciem a comunicação com os clientes e, portanto, aplicações que exigem essa característica precisam de alternativas como o modelo de comunicação *push server*. O *push server* é uma técnica criada com base no modelo “comet”, apresentada em 2006 por Alex Russell. O “comet*”* descreve um modelo de comunicação entre cliente e servidor em que a cada requisição do cliente a conexão HTTP é persistida por um certo tempo, permitindo, que nesse período o servidor envie dados para o cliente sem uma requisição HTTP explícita (ALVARENGA, 2013).

A criação de tecnologias que seguem o modelo *push server* possibilitaram uma evolução nas aplicações web, que antes dependiam de uma requisição ao servidor para retornar um conteúdo, resultando em aplicações estáticas e com pouca interatividade com o usuário. Um dos responsáveis por tonar as páginas web mais interativas é a tecnologia *Ajax,* que permite atualizar fragmentos da página dinamicamente manipulando os elementos do DOM.

O *Ajax* (*Asynchronous JavaScript and XML*) é um grupo de tecnologias que permite enviar e receber dados do servidor através de requisições assíncronas em segundo plano, com o objeto *XMLHttpRequest* (disponível pela maioria dos browsers conhecidos do público geral, como *Google Chrome* e *Mozilla Firefox*). Isso evita que clientes fiquem ociosos à espera da resposta do servidor, aumentando, portanto, a interatividade. Esse fator ocorre porque, com a utilização do *Ajax*, não é necessário atualizar toda a página, apenas manipular o DOM para modificar a parte da página desejada.

Outra tecnologia que ajudou na evolução das aplicações web foi o *WebSocket*. *WebSocket* é uma tecnologia que permite a comunicação bidirecional entre clientes e servidores através de um canal *full-duplex* sobre um único *socket* (ALVARENGA 2013). Seu foco é a comunicação em tempo real e, diferente do *Ajax* e outras tecnologias como as baseadas no “comet”, o *WebSocket* geralmente realiza sua comunicação de acordo com as mudanças ocorridas no servidor, não sendo necessário uma requisição do usuário para que algo aconteça. O *WebSocket* é utilizado em *chats,* portais de notícias e aplicações que necessitam atualizar seus dados constantemente.

Em versões anteriores do CodeLive foi utilizado a tecnologia REST (*Represetational State Transfer*) sendo a mesma baseado em requisições HTTP, ou seja, a ferramenta utilizava o modelo de comunicação tradicional. Entretanto, entende-se que este ambiente poderia proporcionar uma comunicação com o usuário que fosse além da tradicional. Por exemplo, anteriormente o usuário precisava acessar a página de desafios para saber quais novos itens estavam disponíveis, porém seria mais interessante uma funcionalidade em que o próprio CodeLive notificasse o usuário quando um novo desafio estivesse disponível, sem a necessidade de uma ação do usuário para tomar conhecimento deste fato.

A deficiência na interação da ferramenta com o usuário é apenas um dos problemas existentes. Outro problema está na execução do código do usuário, outra funcionalidade de extrema importância para o funcionamento da aplicação e cumprimento do objetivo proposto. Antes da aplicação do presente trabalho, o CodeLive se propunha a executar código em linguagem de programação Java, entretanto, nos trabalhos anteriores não houve a criação de uma ferramenta capaz de executar códigos Java de maneira eficiente.

Diante das tecnologias apresentadas e da necessidade de evolução do CodeLive para afim de sanar os problemas citados, o presente trabalho se propôs a dar continuidade ao desenvolvimento do CodeLive, provendo maior interatividade com o usuário através do uso da tecnologia *WebSocket* (implementada pelo *framework Socket.io*), e criando um mecanismo de execução do código do usuário mais eficiente, com o uso do *framework Skulpt* que executa códigos na linguagem de programação *Python*.

As ferramentas citadas (*Socket.io* e *Skulpt*) foram fundamentais no desenvolvimento deste trabalho, sendo utilizadas em áreas diferentes da aplicação.

O *Socket.io* implementa o protocolo *WebSocket*, logo sua utilização permite a comunicação em tempo-real. Esse recurso foi usado para atualizar o *ranking* dos melhores usuários, notificar sobre mudanças no jogo, tais como: novo desafio criado e usuário que acaba de entrar no jogo. Também sendo utilizado para transportar dados entre *back-end* e *front-end* quando necessário.

O *Skulpt* é um *framework* para execução de código *Python* diretamente no navegador e foi utilizado para sanar o problema na execução do código do usuário. É importante deixar claro que a utilização do *Skulpt*, além de sanar o problema de execução do código do usuário, também é uma adequação ao contexto dos cursos de Ciência da Computação e Sistemas de Informação do CEULP/ULBRA (que utilizavam a linguagem Java, mas que agora utilizam a linguagem Python no ensino da programação).

Ao fazer uso dos *frameworks* citados o projeto também teve sua arquitetura modificada, principalmente no que tange ao *back-end*. Inicialmente a solução foi desenvolvida utilizando a linguagem de programação PHP, porém, tanto o *Socket.io* quando o *Skulpt,* trabalham com a linguagem de programação *JavaScript*, logo modificar o *back-end* da aplicação foi um dos requisitos deste trabalho. Como consequência, a alteração trouxe o benéfico de um código mais homogêneo, já que o *front-end* utiliza a mesma tecnologia (o *front-end* utiliza o *AngularJs* que é um *framework JavaScript*).

O presente texto está organizado da seguinte forma: o capítulo 2 apresenta a base teórica do trabalho, contendo a evolução das tecnologias e dos principais protocolos de comunicação da internet; o capítulo 3 contém os materiais e métodos utilizados ao logo do trabalho, ou seja, *frameworks,* tecnologias de *software* e as técnicas adotadas em cada etapa do desenvolvimento; o capítulo 4 ilustra o processo de atualização da aplicação e os resultados obtidos; o capítulo 5 faz um apanhado geral do trabalho, a pontando os problemas existes, as soluções e resultados encontrados e as melhorias futuras que podem ser aplicadas a ferramenta.

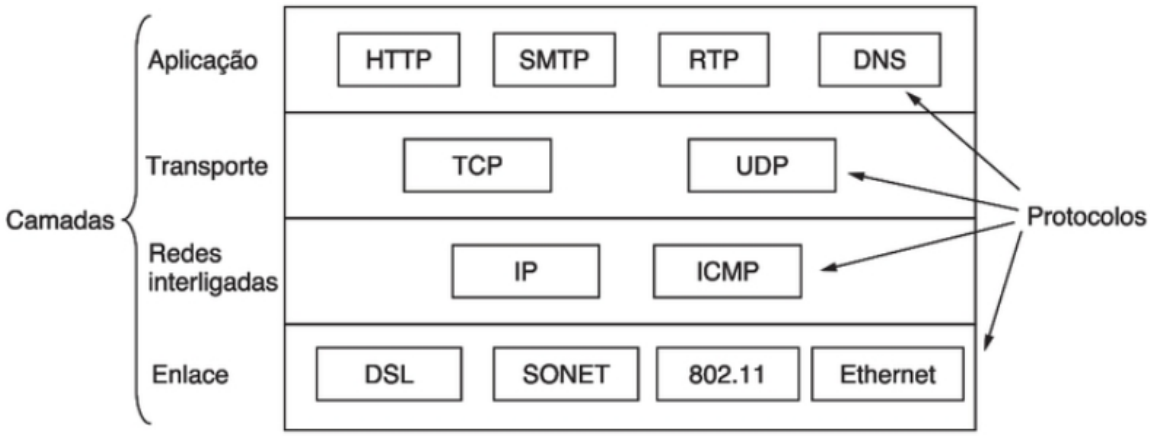
2 REFERENCIAL TEÓRICO

Nesta seção são apresentados os conceitos relacionados aos protocolos de transporte de rede e sua utilização em outros protocolos como o HTTP. Também são apresentadas algumas tecnologias voltadas para a troca de dados em tempo real na Web, sendo algumas delas: *Polling*, *Streaming* e *WebSocket*. Por fim, como o trabalho se propõe a explorar a tecnologia *WebSocket*, esta será apresentada em detalhes.

2.1. Protocolos de transporte de rede

Muitas aplicações distribuídas exigem a transferência de dados entre *hosts* remotos, ou seja, precisam de um recurso de comunicação em rede. Essa troca de dados é realizada por meio de um protocolo de “camada de transporte”, tal como o TCP ou UDP, bem como de um protocolo de “camada de aplicação” (IDOL, 2013).

Figura 1 - Camadas TCP/IP.



Fonte: TANENBAUM; WETHERALL, 2011, p. 29

A Figura 1 ilustra as camadas que compõem o protocolo TCP/IP. Posicionada entre as camadas de aplicação e de rede, a **camada de transporte** é uma peça central da arquitetura de rede em camadas (ROSS; KUROSE, 2013). Para Ribeiro e Mendes (2005) a camada de transporte é responsável por receber os dados vindos da camada de aplicação e transformá-los em pacotes que posteriormente serão entregues à camada de Rede. Ela desempenha o papel fundamental de fornecer serviços de comunicação diretamente aos processos de aplicação que executam em *host*s diferentes, além de oferecer um serviço confiável (no caso do TCP), eficiente e

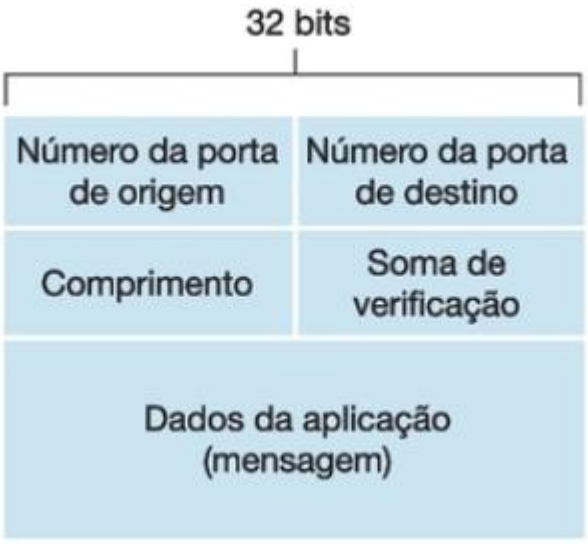
econômico a seus usuários, que, em geral, são processos presentes na camada de aplicação.

Na internet há dois protocolos principais que estão na camada de transporte, um é orientado a conexões (TCP) e o outro não (UDP). As subseções 2.1.1 e 2.1.2, explanam as características e particularidade de cada um deles.

2.1.1. UDP

Segundo Tanenbaum e Wetherall (2011), o UDP oferece um meio para as aplicações enviarem datagramas IP encapsulados sem que seja necessário estabelecer uma conexão, transmitindo datagramas UDP que consistem em um cabeçalho de 8 bytes, seguido pela carga útil (Dados da aplicação). O protocolo é orientado a transação e não garante a entrega nem a proteção dos dados, logo, aplicações que exigem uma entrega ordenada de fluxo de dados com segurança devem usar o TCP (POSTEL, 1980). Na Figura 2 são apresentados os segmentos que compõem a estrutura do UDP.

Figura 2 - Estrutura do segmento UDP.



Fonte: ROSS; CUROSE, 2013, p. 148

O cabeçalho UDP tem apenas quatro campos, cada um deles consistindo em 2 bytes (ROSS; CUROSE, 2013). Os tópicos a seguir descrevem cada um deles:

- **Dados da aplicação:** Presentes na estrutura de segmento UDP (Figura 2), ocupam o campo de dados do segmento UDP. Para uma aplicação de

recepção de áudio, por exemplo, amostras de áudio preenchem o campo de dados;

- **Portas de Origem/Destino:** As duas portas servem para identificar os pontos extremos nas máquinas de origem e destino. Quando um pacote UDP chega, sua carga útil é entregue ao processo associado à porta de destino (TANENBAUM; WETHERALL, 2011). As portas são como caixas de correios que as aplicações podem utilizar para receber pacotes;
- **Comprimento:** Este campo representa o comprimento em octetos dessa mensagem, incluindo o cabeçalho e os dados. (Isso significa que o valor mínimo do comprimento é 8) (POSTEL, 1980).
- ***Checksum:*** O *Checksum* (Soma de verificação) serve para detectar erros. Em outras palavras, é usando para determinar se bits dentro do segmento UDP foram alterados durante sua movimentação da origem até o destino (ROSS; CUROSE, 2013). Ele faz o c*hecksum* do cabeçalho, dos dados e de um pseudocabeçalho conceitual do IP.

O pseudocabeçalho, prefixado ao cabeçalho UDP, contém os endereços de origem e destino, o protocolo, e o comprimento do UDP (POSTEL, 1980). A inclusão do pseudocabeçalho no cálculo do *checksum* do UDP ajuda a detectar pacotes não entregues, mas incluí-lo também infringe a hierarquia de protocolos, pois os endereços IP contidos nele pertencem à camada do IP e não à camada do UDP (TANENBAUM; WETHERALL, 2011).

Para Ross e Kurose (2013) o UDP possui algumas vantagens que fazem com que algumas aplicações optem por utilizá-lo ao invés do TCP, mesmo este sendo mais confiável. Algumas das vantagens são:

- melhor controle no nível da aplicação sobre quais dados são enviados e quando. Quando um processo passa dados, o UDP empacota e repassa para a camada de rede. No caso do TCP, o remetente é limitado pelo controle de congestionamento;
- não há estabelecimento de conexão, ou seja, o UDP simplesmente envia mensagens sem nenhuma preliminar formal, assim não introduz nenhuma atraso para estabelecer uma conexão;

- não há estados de conexão. Por essa razão, um servidor devotado a uma aplicação específica pode suportar um úmero muito maior de clientes ativos quando a aplicação roda sobre UDP e não sobre TCP;
- pequena sobrecarga de cabeçalho de pacote. O segmento TCP tem 20 bytes de sobrecarga de cabeçalho, enquanto o UDP tem somente 8 bytes.

Morimoto (2005) diz que o protocolo UDP se concentra em transmitir dados com a maior eficiência possível, tendo sua utilização voltada para serviços que admitam alguma perda de dados como no caso do *streaming* (tanto de áudio, quanto de vídeo) e do *VoIP*. Outras utilizações são em serviços com fluxo de dados em tempo real, *multicasting* e *broadcasting*. O *YouTube*[1] e *Skype*[2] são exemplos de ferramentas que utilizam o UDP.

2.1.2. TCP

O TCP foi projeto especificamente para oferecer um fluxo de bytes fim a fim confiável em uma rede interligada (como a internet) que é diferente de uma única rede, porque suas diversas partes podem ter topologias, larguras de banda, atrasos, tamanhos de pacote e outros parâmetros diferentes (TANENBAUM; WETHERALL, 2011).

Ross e Kurose (2013) denominam o TCP como um protocolo orientado a conexão, porque antes que um processo comece a enviar dados a outro, os dois processos enviam segmentos preliminares para estabelecer os parâmetros da transferência de dados, ou seja, estabelecem uma conexão. Um detalhe importante da conexão TCP é que este provê um serviço full-duplex. Isso significa que se há uma conexão do processo A em um hospedeiro e do processo B em outro hospedeiro, então os dados da camada de aplicação poderão fluir de A para B ao mesmo tempo que de B para A. Outro detalhe é que a conexão é sempre fim a fim, isto é, entre um único remetente e um único destinatário, não possibilitando o multicast (transferência de dados de um remetente para vários destinatários) como o UDP.

---

[1] Site de compartilhamento de vídeos enviados pelos usuários através da internet;

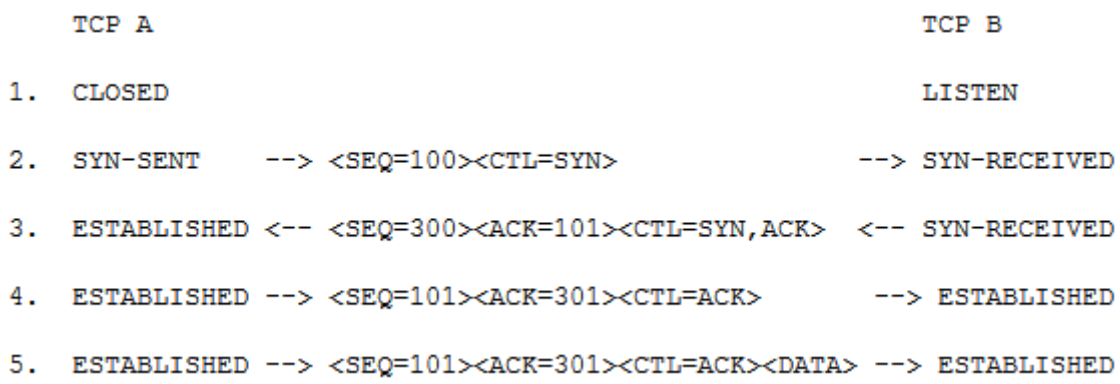
[2] Software que possibilita comunicações de voz e vídeo via Internet, permitindo a chamada gratuita entre usuários em qualquer parte do mundo.

O processo para se estabelecer uma conexão TCP é denominado *3-way handshake* (ou cumprimento de três vias). Este procedimento é normalmente iniciado por um *host* que responde O TCP foi projeto especificamente para oferecer um fluxo de bytes fim a fim confiável em uma rede interligada (como a internet) que é diferente de uma única rede, porque suas diversas partes podem ter topologias, larguras de banda, atrasos, tamanhos de pacote e outros parâmetros diferentes (TANENBAUM; WETHERALL, 2011).

Ross e Kurose (2013) denominam o TCP como um protocolo orientado a conexão, porque antes que um processo comece a enviar dados a outro, os dois processos enviam segmentos preliminares para estabelecer os parâmetros da transferência de dados, ou seja, estabelecem uma conexão. Um detalhe importante da conexão TCP é que este provê um serviço full-duplex. Isso significa que se há uma conexão do processo A em um hospedeiro e do processo B em outro hospedeiro, então os dados da camada de aplicação poderão fluir de A para B ao mesmo tempo que de B para A. Outro detalhe é que a conexão é sempre fim a fim, isto é, entre um único remetente e um único destinatário, não possibilitando o multicast (transferência de dados de um remetente para vários destinatários) como o UDP.

O processo para se estabelecer uma conexão TCP é denominado *3-way handshake* (ou cumprimento de três vias). Este procedimento é normalmente iniciado por um *host* que deseja se comunicar com outro *host*. Ele também funciona se dois processos iniciarem o procedimento simultaneamente (POSTEL, 1981).

Figura 3 - 3-way handshake para sincronização de conexão.

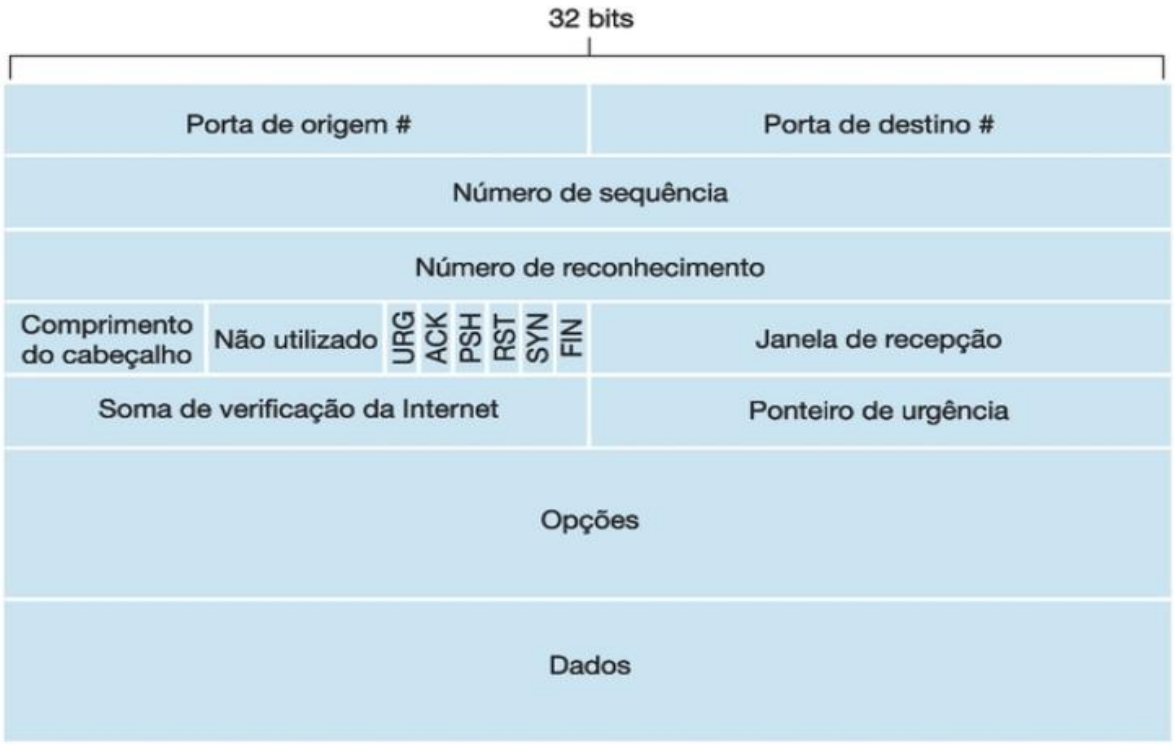


Fonte: POSTEL, 1981, p. 31

A Figura 3 ilustra o estabelecimento de uma conexão entre os processos A (cliente) e B (Servidor). O processo A inicia a conexão enviando o segmento SYN ao processo B, o processo B por sua vez, informa que o segmento foi recebido e reconhecido, e envia uma resposta ao processo A. Nesse momento, A aceita o primeiro termo, ou seja, está de acordo com os parâmetros de B, A envia mais dois segmentos sendo que o último possui uma carga útil. Após o término desse processo, A e B podem iniciar o intercâmbio de dados.

Um dos processos que acontecem durante o *handshake* é a reserva de um *buffer* de envio, que armazena todos os dados que serão enviados. De tempos em tempos o TCP retira pedaços dos dados do *buffer* e passa à camada de rede. Esses dados são passados no formato de segmentos. A quantidade máxima de dados que pode ser retirada e colocada em um segmento é limitada pelo tamanho máximo do segmento (*maximum segment size* - MSS). O MSS é estabelecido com base na unidade máxima de transmissão (*maximum transmission unit* - MTU), que é o maior quadro de camada de enlace que pode ser enviado pelo remetente. Os protocolos da camada e enlace Ethernet e PPP possuem um MSS de 1.500 bytes (ROSS; KUROSE, 2013).

Figura 4 - Estrutura do segmento TCP.



Fonte: ROSS; KUROSE, 2013, p. 172.

A Figura 4 ilustra a estrutura de um segmento TCP, alguns campos como: **Porta de origem**, **Porta de destino**, **Soma de verificação da Internet** (ou *checksum*) e **Dados**, também estão presentes no UDP e possuem a mesma finalidade, os demais campos são específicos do TCP. Os tópicos abaixo descrevem cada um deles (ROSS; KUROSE, 2013).

- **Comprimento do cabeçalho** – possui 4 bits, especifica o comprimento do cabeçalho TCP em palavras de 32 bits. Geralmente o tamanho do cabeçalho TCP é de 20 bytes, mas pode variar de acordo com o tamanho do campo Opções;
- **Opções** – campo opcional e de comprimento variável, é usado quando um remetente e um destinatário negociam o MSS, ou como um fator de aumento de escala da janela para utilização em redes de alta velocidade;
- **Campo de flag** – contém 6 bits, sendo eles:
    - **ACK** – é usado para indicar se o valor carregado no campo de reconhecimento é válido, isto é, se o segmento contém um reconhecimento para um segmento que foi recebido com sucesso;
    - **RST**, **SYN** e **FIN** - são usados para estabelecer e encerrar a conexão;

    - **PSH** – indica que o destinatário deve passar os dados para a camada superior imediatamente;
    - **URG** – é usado para mostrar que há dados nesse segmento que a entidade da camada superior do lado remente marcou como "urgente";
- **Ponteiro de urgência** – tem 16 bits, ele indica a localização do último byte dos dados urgentes;
- **Número de sequência** – é o número do primeiro byte do segmento. Exemplo: Uma cadeia de dados consiste em um arquivo composto de 500 mil bytes, que o MSS seja 1.000 bytes e que seja atribuído o número 0 ao primeiro byte da cadeia de dados. Para transportar esses dados, o TCP irá construir 500 segmentos a partir da cadeia de dados. O primeiro recebe o número de sequência 0; o segundo 1.000, o terceiro 2.000 e assim por diante;
- **Número de reconhecimento** – em uma troca de dados entre o hospedeiro A e o hospedeiro B, o número de reconhecimento que o hospedeiro A atribui a seu segmento é o número de sequência do próximo byte que ele está aguardando do hospedeiro B. Suponha que o hospedeiro A tenha recebido do hospedeiro B todos os bytes numerados de 0 a 535 e também que esteja preste a enviar um segmento a B. O hospedeiro A está esperando pelo byte 536 e por todos os bytes subsequentes da cadeia de dados do hospedeiro B. Assim, ele coloca o número 536 no campo de número de reconhecimento do segmento que envia para o hospedeiro B;
- **Janela de recepção** – se sabe que os *host*s de cada lada de uma conexão TCP reservam um buffer de recepção para a conexão, assim, quando uma conexão TCP recebe bytes, estes são colocados no buffer de recepção. Logo, a janela de recepção é usada para dar ao remente uma ideia do espaço disponível no buffer de recepção do destinatário. Como o TCP é full-duplex, o remetente de cada lado da conexão mantém uma janela de recepção distinta.

Diante dos diversos aspectos sobre o TCP, o mais relevante sem dúvida é a transferência de dados de forma confiável, requisito indispensável para aplicações que precisam garantir a entrega da informação.

Esta seção apresentou os diversos aspectos dos protocolos de transporte de rede, dando ênfase ao UPD e ao TCP, já que estes são amplamente utilizados na internet e de grande relevância para o presente trabalho. A próxima seção explana

os detalhes de umas das tecnologias de comunicação mais utilizadas da Web: o protocolo HTTP, que utiliza o TCP.

2.2. Protocolo HTTP

O desenvolvimento do protocolo HTTP iniciou-se por volta de 1990 para facilitar a requisição e transferência de documentos HTML entre navegadores e servidores web. Ele é um protocolo genérico, orientado a objetos e stateless (sem estado), isso significa que o servidor envia ao cliente os arquivos solicitados sem armazenar nenhuma informação de estado sobre este, ou seja, se um cliente solicitar um mesmo objeto várias vezes em um curto período de tempo, o mesmo será enviando também várias vezes, já que não há um estado que indique o envio daquele objeto anteriormente (ROSS; KUROSE, 2013).

O cliente envia a requisição (com os verbos GET, POST, PUT, DELETE etc.) para um servidor que então envia a resposta de volta. Todas as mensagens HTTP são precedidas por um cabeçalho de informações codificadas com pares de chave/valor de texto, ou seja, arbitrariamente em formato ASCII. No entanto, o próprio conteúdo da mensagem pode ser de qualquer tipo, dados binários ou de texto (IDOL, 2013).

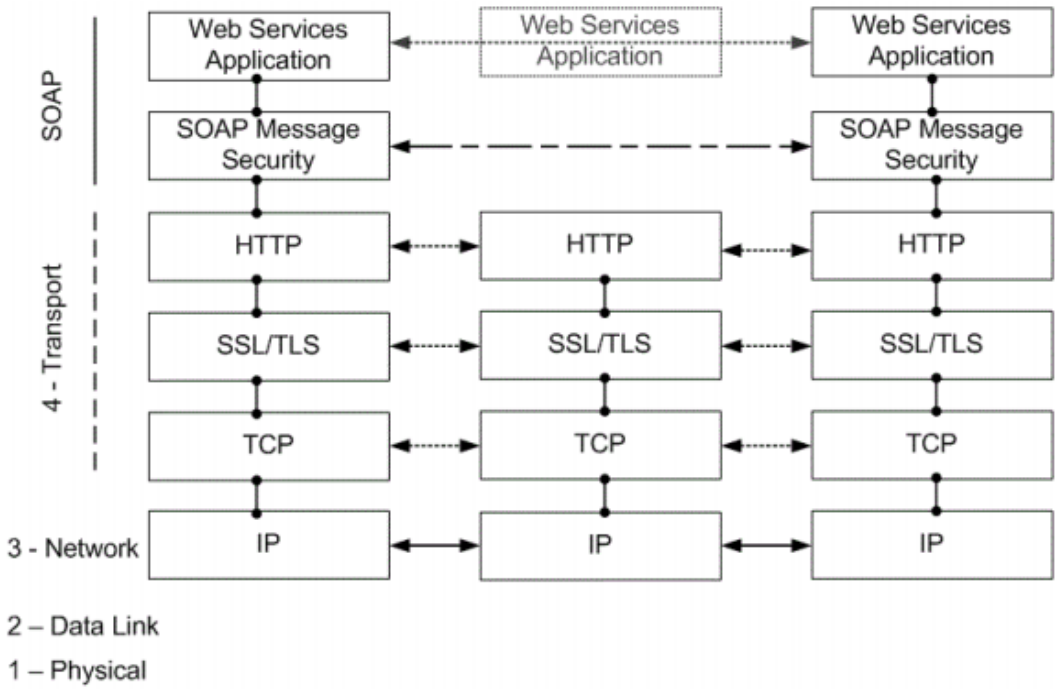
O HTTP define como clientes web requisitam páginas web aos servidores, e como estes as transferem de volta aos clientes. Essas requisições são transportadas pelo protocolo de transporte TCP, assim, o cliente primeiramente inicia uma conexão TCP com o servidor, e após estabelecer a conexão, os processos do browser e do servidor acessam o TCP por meio de suas interfaces *sockets*. No lado do cliente a interface socket é a porta entre o processo cliente e a conexão TCP, no lado do servidor, a interface *socket* é a porta entre o processo servidor e a conexão TCP. O TCP é utilizado pois provê ao HTTP um serviço confiável de transferência de dados, logo toda mensagem de requisição ou resposta chegará intacta ao seu destino.

“Originalmente o HTTP foi projetado para transferir apenas documentos HTML (e somente suportava o verbo GET) mas, com o lançamento da versão 1.0 em 1996, o suporte para outros tipos foi adicionado na especificação” (IDOL, 2013, p. 6). Naquela época as páginas web eram documentos estáticos que não exigiam qualquer tipo de comunicação entre cliente e servidor após a requisição ser atendida, ou seja, utilizava-se um modelo de conexão TCP não persistente. Porém, mesmo com esse modelo de páginas estática, múltiplas requisições HTTP são feitas

uma para cada elemento (ou objeto) diferente da página (tal como arquivos de: imagens, CSS, JavaScript, etc.) e isso consequentemente exige uma conexão TCP preparada para ser estabelecida e destruída imediatamente antes e depois de cada requisição. Para solucionar esse problema, a versão 1.1 adotou como padrão o modelo de conexão persistente (IDOL, 2013). Em conexões persistentes o servidor deixa a conexão TCP aberta após enviar a resposta e quando o servidor recebe requisições consecutivas, os objetos são enviados de forma ininterrupta de uma única vez (ROSS; KUROSE, 2013).

O HTTP também é usado como um protocolo genérico para comunicação entre agentes de usuários e proxies/gateways por outros protocolos web, tais com: SMTP, NNTP, FTP, entre outros, permitindo acesso básico a hipermídia e recursos disponíveis em diversas aplicações que simplificam a implementação de agentes de usuário.

Figura 5 - Exemplo de WebService construído sob o HTTP.



Fonte – IDOL, 2013, p. 7

Web Service (WS) é um tipo de serviço que utiliza o HTTP. Ele foi criado para construir aplicações que são serviços na internet, tendo a função de “chamar métodos” (remotamente) usando XML (GARBO; DIAS, 2015). Com WS é possível enviar ou receber informações de outros *softwares*, não importando a linguagem de programação em que estes foram desenvolvidos, o sistema operacional em que

rodam e o hardware que é utilizado (Gomes, 2014). “Existem dois padrões de desenvolvimento de *web services*, um é o SOAP (*Simple Object Acces Protocol*) que é baseado no protocolo XML e o outro é o REST (*Representational State Transfer*) ou *RESTfull* que utiliza unicamente o protocolo HTTP” (GARBO; DIAS, 2015).

As tecnologias REST e SOAP (Figura 5) são exemplos de arquiteturas *Web Service* que não estão relacionados por uma página web. O REST e SOAP fornecem uma maneira dos servidores exporem uma API para que os clientes acessem usando requisições HTTP, e tenham acesso aos recursos disponibilizados por este serviço.

2.2.1. Comunicação

O protocolo HTTP é baseado no paradigma *request/response* (ou requisição-resposta). Nesse modelo, um cliente estabelece uma conexão com um servidor enviando uma requisição na forma de um método de solicitação com uma URI, a versão do protocolo seguido por uma mensagem MIME contendo os modificadores da requisição, as informações do cliente e no corpo, o possível conteúdo. Após a requisição, o servidor responde, envia uma linha de status incluindo a versão do protocolo e um código de sucesso ou de erro seguido por uma mensagem MIME contendo as informações do servidor, e o corpo o possível conteúdo (BERNERS-LEE; FIELDING; NIELSEN, 1996).

O cliente geralmente solicita algum do servidor. Em um caso mais simples isso pode ser conseguido através de uma única conexão entre o cliente e o servidor.

Figura 6 - Comportamento de requisição-resposta do HTTP.

Fonte – ROSS; CUROSE, 2013, p. 73

A Figura 6 ilustra a comunicação de usuários com sistemas operacionais distintos a um servidor Web. O caso ilustrado na Figura 6 é o mais simples, pois a comunicação entre o cliente e o servidor é direta sem a presença de intermediários. Uma situação mais complicada ocorre quando um ou mais intermediários estão presentes na cadeia de *request/response.* Existem três formas comuns de intermediário: de proxy, de gateway e de túnel.

Como dito anteriormente o HTTP é baseado no paradigma *request/response,* isso significa que existe um comportamento e formato de mensagem especifico para uma requisição e para uma resposta.

Figura 7 - Mensagem de requisição HTTP.

```
GET /somedir/page.html HTTP/1.1
Host: www.someschool.edu
Connection: close
User-agent: Mozilla/4.0
Accept-language: fr
```

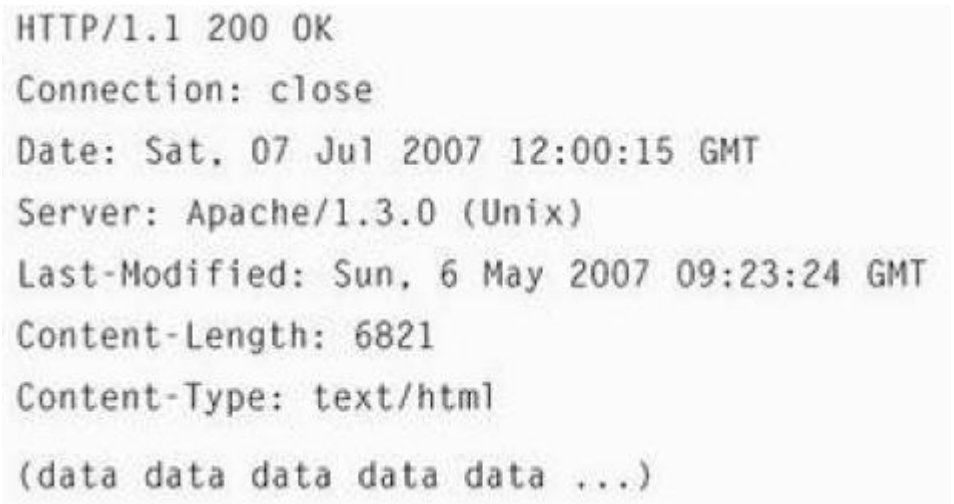
Fonte – ROSS; CUROSE, 2013, p. 76

O exemplo de requisição HTTP mostrado na Figura 7 possui somente cinco linhas, mas uma requisição HTTP pode ser muito maior do que o apresentado ou possuir apenas uma linha. Neste exemplo, a primeira linha possui o método (nesse caso o GET) a URL onde se encontra o recurso (nesse caso /somedir/page.html) e a versão do protocolo (nesse caso a 1.1). Essa primeira linha é chamada de **linha de requisição**, as linhas subsequentes são chamadas de **linhas de cabeçalho**. As linhas de cabeçalhos da Figura 7 possuem os seguintes atributos:

- **Host** – especifica o hospedeiro no qual o objeto que se está solicitando reside;
- **Connection** – especifica se a conexão é persistente ou não;
- **User-agent** – especifica o agende de usuário, ou seja, o navegador que está fazendo a solicitação. Com essa informação é possível enviar versões diferentes de um mesmo objeto;
- **Accept-language** – especifica a linguagem na qual o usuário prefere receber o objeto, caso não exista uma versão na linguagem solicitada, o servidor envia uma versão *default*.

Vale ressaltar que ainda existe um espaço após as linhas de cabeçalho, esse é o corpo da mensagem, que geralmente é preenchido quando se utiliza o método POST ou o método PUT, usados quando se deseja enviar dados (de um formulário HTML, por exemplo) para o servidor.

Figura 8 - Mensagem de resposta HTTP.



Fonte – ROSS; CUROSE, 2013, p. 78

A Figura 8 ilustra um exemplo de mensagem de resposta. A mensagem está dividida em três seções: **linha de estado**, **linha de cabeçalho** e, por último, o **corpo da entidade**. A primeira linha é formada de três atributos: a versão do protocolo (nesse caso a 1.1), o código de estado da resposta (nesse caso o 200) e uma mensagem de estado correspondente (nesse caso OK). Este exemplo possui seis linhas de cabeçalho:

- **Connection** – especifica se vai manter a conexão aberta ou não após enviar a mensagem;
- **Date** – mostra a data e hora em que a resposta foi enviada pelo servidor;
- **Server** – indica de qual servidor e a requisição foi enviada. Também pode mostra o sistema operacional e a versão do servidor;
- **Last-Modified** – mostra a data e hora em que o objeto foi criado ou modificado pela última vez;
- **Content-Length** – indica a número de bytes do objeto que está sendo enviado;
- **Content-Type** – mostra qual o tipo do objeto presente no corpo da mensagem.

A última seção é a linha que possui o valor “data...”. Este é o corpo da resposta e possui o objeto solicitado (KOSS; KUROSE, 2013). Analisando a URL da solicitação (contido Figura 7), fica claro que este campo está preenchido por documento HTML.

Esta seção apresentou detalhes sobre protocolos de rede e sobre o principal protocolo utilizado na internet, o HTTP. A seção seguinte apresenta outras tecnologias para troca de dados na internet, estas baseiam no HTTP, porém, o objetivo é simular uma comunicação em tempo real.

2.3. Tecnologias alternativas ao HTTP

Segundo Muller (2014) desde o surgimento do HTTP, por volta de 1990, a comunicação entre cliente e servidor tem passado por muitas atualizações. No início dos anos 90, a maior parte da web era estática. Como consequência disso, a comunicação entre cliente e servidor era bastante limitada. Geralmente, o cliente faz uma requisição ao servidor e este então a responde, porém, toda a comunicação se

mantem “parada” até que uma nova ação seja feita pelo cliente. A noção de Web dinâmica apareceu em 2005 com a introdução de tecnologias com o *Comet*.

O *Comet* é constituído por um conjunto de tecnologias que é também conhecido como *Reverse Ajax*, por utilizar o Ajax, dentre outras tecnologias, para permitir que o servidor inicie a comunicação com o cliente. O termo foi criado em 2006 pelo Engenheiro de Software Alex Russell que descreve um modelo de comunicação entre cliente e servidor onde a cada requisição do cliente. A conexão HTTP persiste por um certo tempo, permitindo que nesse período o servidor envie dados para o cliente sem uma requisição explicita, essa técnica é conhecida como *push server* (ALVERENGA, 2013).

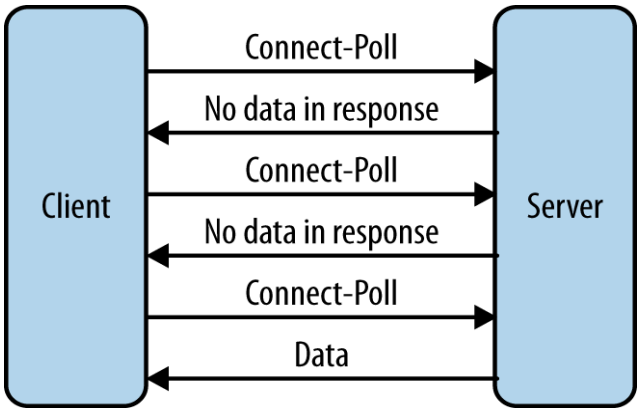
Tentativas atuais para fornecer aplicações web em tempo real giram em torno principalmente do *Polling* e outras tecnologias de envio no lado do servidor, sendo a maioria baseada no *Comet*. Tecnologias de envio baseado no *Comet* são quase sempre implementadas em *JavaScript* e usam conexões estratégicas tais como o *Long-Polling* ou o *Streaming* (LUBBERS; GRECO, 2010).

As subseções: 2.3.1, 2.3.2 e 2.3.3 explanam o funcionamento de algumas tecnologias baseadas no *Comet* que buscam realizar a comunicação em tempo real iniciando uma conexão do servidor para o cliente.

2.3.1. Polling

No *Polling* o navegador envia requisições HTTP em um intervalo regular pré-configurado (por exemplo, utilizando a função `setInterval()`, do JavaScript) e recebe uma resposta logo em seguida (FAIN; RASPUTNIS; TARTAKOVSKY; GAMOV, 2014).

Figura 9 - Esquema de funcionamento do Polling.



Fonte – http://enterprisewebbook.com/ch8_websockets.html

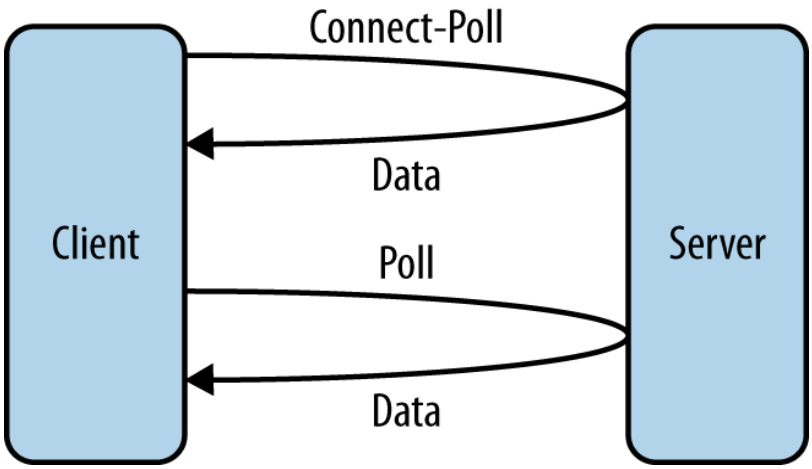
Essa técnica foi a primeira tentativa de fazer com que o navegador entregasse informações em tempo real. Obviamente, essa é a melhor solução se o intervalo entre as requisições for conhecido e for suficiente para “sincronizar” o cliente com o servidor. Uma aplicação que tenha novos dados no intervalo de 15 minutos por exemplo, se beneficiaria deste recurso.

Entretanto, frequentemente não é possível prever o intervalo correto das requisições, o que tornará algumas delas inevitavelmente irrelevantes pois nem todas as requisições trarão dados, assim como muitas conexões serão abertas e fechadas desnecessariamente. A Figura 9 ilustra bem esse problema, pois em três requisições apenas em uma (a terceira) obteve-se uma resposta que contivesse dados.

2.3.2. Long-Polling

O *Long-Polling* é uma tecnologia baseado no *Comet*, sendo uma evolução do *Pooling* no que diz respeito a eventos enviados do servidor para o cliente na comunicação em tempo real. Nele o navegador envia uma requisição ao servidor, que a mantém aberta por um período de tempo (Figura 10). Se a solicitação é recebida dentro do prazo, uma resposta contendo a mensagem é enviada para o cliente e a conexão é encerrada, caso contrário, o servidor irá responder com uma notificação para finalizar a conexão. Após o navegador do cliente receber a resposta ele cria um novo pedido para lidar com com o próximo evento, mantendo sempre o cliente atualizado com os novos dados assim que o servidor os disponibiliza (MULLER, 2014).

Figura 10 - Esquema de funcionamento do Long-Polling.



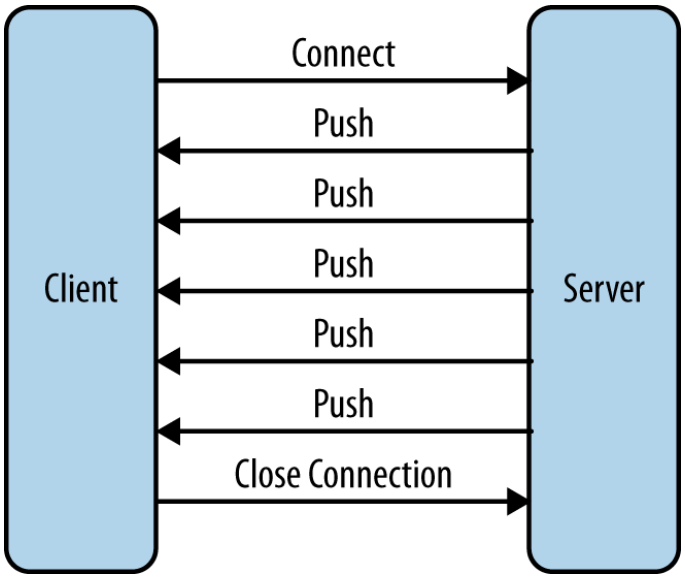
Fonte – http://enterprisewebbook.com/ch8_websockets.html

É importante entender que, quando se tem um volume alto de mensagens, o *Long-Polling* não fornece uma melhoria de desempenho substancial sobre o *Polling* tradicional, pois o *Long-Polling* pode sair do controle em um loop contínuo de requisições imediatas. Isso acontece justamente pelo fato do *Long-Polling* possuir um tempo maior para encerrar sua conexão. Assim, uma aplicação como um bate-papo, que possui atualizações constantes em um intervalo pequeno de tempo, poderia abrir diversas conexões e sobrecarregar o sistema.

2.3.3. Streaming

O *Streaming* é baseado em uma conexão HTTP persistente que se inicia com o navegador, a diferença está na resposta. O servidor nunca sinaliza ao navegador que a mensagem está concluída, dessa maneira a conexão se mantém aberta e pronta para entregar mais dados (MULLER, 2014).

Figura 11 - Esquema de funcionamento do Streaming.



Fonte – http://enterprisewebbook.com/ch8_websockets.html

A Figura 11 ilustra o funcionamento do *Streaming*, na qual é possível ver que após a conexão ser aberta o servidor consegue enviar os dados sem a necessidade de fazer várias requisições. A conexão é mantida por um tempo indeterminado (ou por um período definido) até que o cliente solicite o encerramento. Entretanto, uma vez que o *Streaming* ainda é encapsulado no HTTP e passa por *firewalls,* os servidores de *proxy* podem escolher armazenar a resposta acarretando no aumento

da latência na entrega das mensagens. De forma alternativa, conexões TLS[3] (SSL) podem ser usadas para proteger a resposta de ser armazenada, mas nesse caso, a configuração destrói cada taxa de conexão disponível e os recursos mais pesados do servidor (LUBBERS; GRECO, 2010).

Ao longo da seção 2.3 foram apresentados alguns conceitos relacionados a comunicação na internet, as dificuldades e limitações que o HTTP possui quando se deseja transportar dados do servidor para o cliente em tempo real e algumas das tecnologias que se propõem a solucionar este problema. A seção a seguir apresenta o protocolo *WebSocket*, detalha seus componentes, funcionamento e como este permite a comunicação em tempo real de forma eficiente.

2.4. WebSocket

Para Alvarenga (2013), *WebSocket* é uma tecnologia que permite a comunicação bidirecional entre clientes e servidores através de um canal full-duplex sobre um único socket. Já para Wang, Salim e Moskovits (2006), *WebSocket* é um protocolo de rede que define como servidores e clientes se comunicam através da Web. Os protocolos são regras acordadas que definem como estabelecer uma comunicação. Essas especificações são descritas na RFC 6455 (O Protocolo *WebSocket*) criada em 2011 pelo IETF (*Internet Engineering Task Force*) e contém as regras exatas (especificações) que devem ser seguidas na implementação de um cliente ou servidor *WebSocket* (ALVARENGA, 2013).

A motivação para criação do *WebSocket* vem de alguns problemas existentes na comunicação tradicional utilizando o protocolo HTTP e da necessidade das aplicações como, por exemplo, se comunicar de forma bidirecional entre o cliente e servidor. Na comunicação tradicional diversas conexões TCP são feitas, além da alta taxa de sobrecarga devido à presença do cabeçalho HTTP nas mensagens entre cliente-servidor. Para Fette e Melnikov(2011) uma solução simples seria a utilização de uma única conexão TCP para trafegar as informações em ambas as direções (cliente-servidor). O *WebSocket* fornece justamente esse serviço,

[3] TLS é um protocolo criptográfico cuja a função é conferir segurança para a comunicação na Internet para serviços como e-mail (STPM), navegação na Internet (HTTP) e outros tipos de transferência de dados.

combinado com o *WebSocket* API (WSAPI), ele fornece uma alternativa ao HTTP para a comunicação de duas vias a partir de uma página web para um servidor remoto. Neste sentido, tem-se que:

> Após estabelecida a conexão *WebSocket*, clientes e servidores podem se comunicar de forma assíncrona, permitindo então que servidores enviem notificações aos clientes a qualquer momento, possibilitando a comunicação em tempo real (ALVARENGA, 2013, p. 31).

O *WebSocket* API (WSAPI) permite as aplicações controlarem o protocolo *WebSocket* e responderem a eventos disparados pelo servidor. Atualmente, o WSAPI é suportado pela maioria dos navegadores modernos e incluem métodos e atributos necessários para conexões *WebSocket* full-duplex bidirecional. A API permite a performance necessária em ações como abrir e fechar a conexão, enviar e receber mensagens, e escutar eventos disparados pelo servidor.

Há uma variedade de implementações *WebSocket* do lado do servidor disponíveis, sendo algumas delas: *Apache mod, pywebsocket, Jetty, Socket.IO, Tornado, SuperWebSocket, EventMachine e Kaazing's WebSocket Gateway* (WANG; SALIM; MOSKOBITS, 2013).

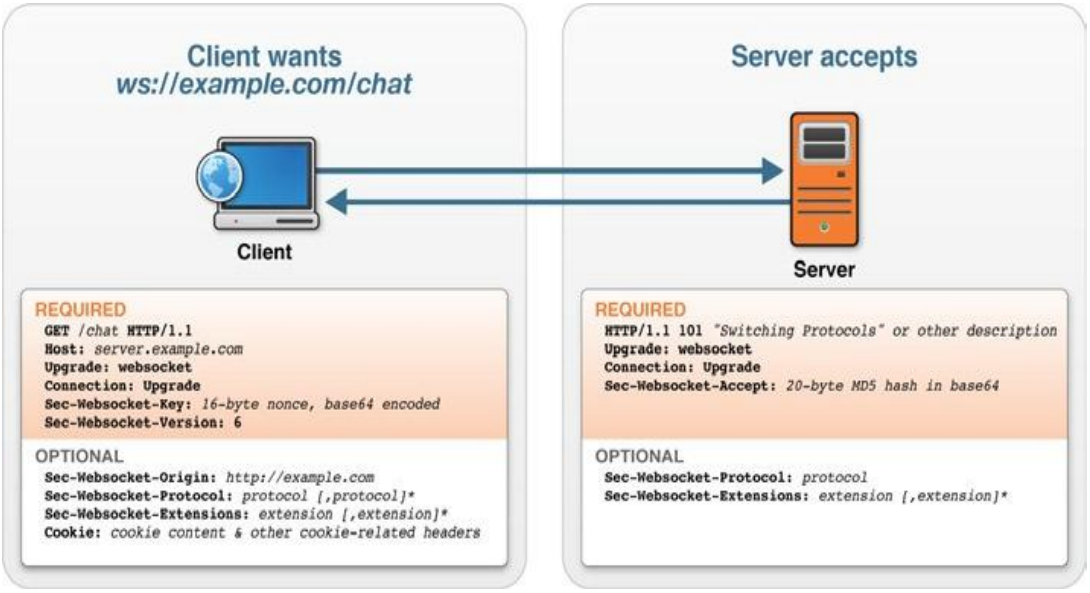
Na comunicação entre cliente e servidor utilizando o *WebSocket* diversas ações são feitas, sendo necessária uma série de processos para abrir a conexão, transmitir os dados, e encerrá-la após a conclusão do objetivo. Os processos presentes no *WebSocket* e os demais detalhes sobre a tecnologia são apresentados nas subseções a seguir.

2.4.1. Opening Handshake

O protocolo *WebSocket* possui basicamente duas partes: uma é o *handshake* (ou “aberto de mão”) e a outra e a transferência de dados. No *handshake* ainda é possível realizar duas operações: o O*pening Handshake* e o C*losing Handshake*, sendo que o primeiro é realizado para estabelecer a conexão entre cliente e servidor, o segundo é realizado para encerrar essa conexão.

Algumas características do HTTP ainda são mantidas, como as URLs e mensagens do tipo texto. A conexão ainda precisa ser iniciada pelo cliente, porém, o protocolo possui o estilo de comunicação do TCP, ou seja, uma vez estabelecida a conexão, as mensagens são enviadas de forma bidirecional (ALVARENGA 2013).

Figura 12 - Exemplo de um Opening handshake.



Fonte – WANG; SALIM; MOSKOBITS, 2013, p. 41

A Figura 12 ilustra o estabelecimento de uma conexão (Opening handshake) entre cliente e servidor, esta operação pretende ser compatível com o *software* do lado do servidor baseado no HTTP, logo o *Opening handshake* é realizado pelo cliente sendo um pedido de atualização do protocolo HTTP para o protocolo *WebSocket* (indicado pelo campo ***Upgrade***) (FETTE; MELNIKOV, 2011).

> Após o envio da requisição do cliente para o servidor, é preciso que o servidor comprove o recebimento. Isso garante que o servidor entende o protocolo *WebSocket* e também funciona como uma medida de segurança, onde somente conexões *WebSocket* serão aceitas. Para comprovar o recebimento, o servidor utiliza o valor do campo Sec-WebSocket-Key do cabeçalho enviado pelo cliente e concatena com uma constante GUID (*Globally Unique Identifier*) que todo servidor *WebSocket* deve conhecer. Após a concatenação, é preciso aplicar o hash SHA-1 (160 bits) e finalmente no resultado do *hash*, a codificação Base64. O resultado desse valor é o valor do campo `Sec-WebSocket-Accept` no cabeçalho do servidor (ALVARENGA, 2013).

Se o *Opening handshake* for realizado com sucesso, ou seja, o servidor retornar o código 101 e o campo `Sec-WebSocket-Accept` retornar o valor esperado, a conexão está estabelecida, porém, se um desses requisitos não for atendido a conexão não foi estabelecida.

2.4.2. Closing Handshake

Após completar a comunicação entre cliente e servidor, a conexão *WebSocket* pode ser finalizada, operação que é chamada de *Closing handshake*. Ela pode ser feita por uma das partes (cliente ou servidor) ou pelas duas ao mesmo tempo, para isso,

deve-se enviar um *frame* de controle com os dados contendo uma especificada sequência de controle para iniciar o encerramento da conexão.

Esse cenário representa que o propósito da conexão foi realizado com sucesso, porém, outros motivos podem levar ao encerramento da conexão. O *frame* é composto por um campo *Opcode* 8 e uma mensagem onde são especificados o código de *status* (um inteiro de 16-bits sem sinal) e a razão (uma string codificada em UTF-8) pelo término da conexão. Isso permite a aplicação diferenciar se uma conexão foi finalizada intencionalmente ou por alguma falha (ALVARENGA, 2013).

Depois da realização do *Closing handshake* nenhuma das partes envia dados, descartando qualquer dado recebido após a operação de encerramento.

2.4.3. Mensagem WebSocket

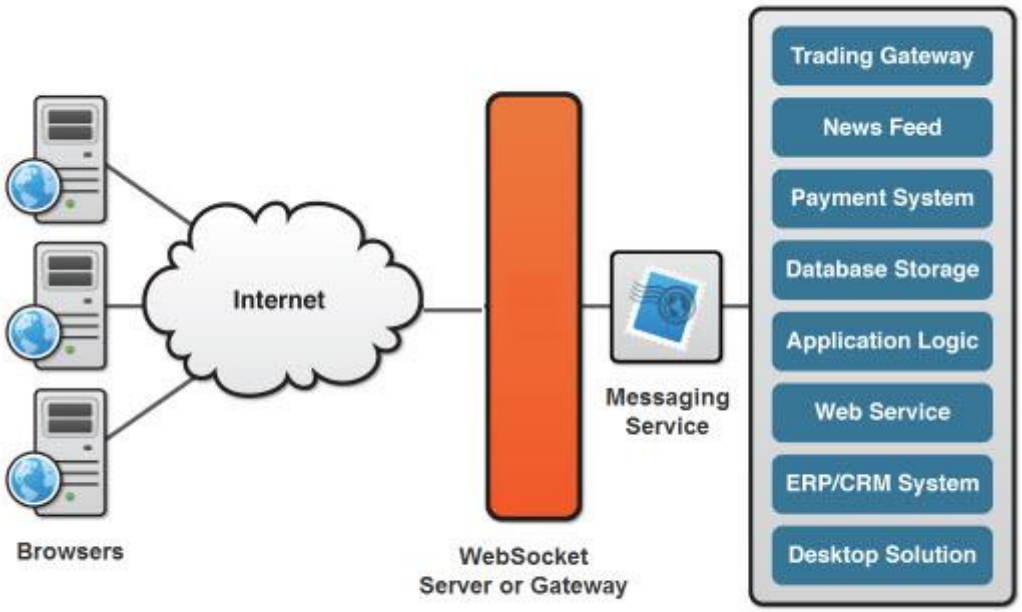
Após a realização do *handshake* com sucesso, a conexão *WebSocket* é estabelecida e mensagens podem ser trocadas entre cliente e servidor. O *WebSocket* usa um formato de mensagem binária de frames para a comunicação pela rede. Cada mensagem é dividida em um ou mais *frames*, onde são reagrupadas ao chegar no destinatário.

2.5. HTML5 WebSocket

A especificação HTML 5 *WebSocket* define uma API que permite que páginas Web utilizem o protocolo *WebSocket* para comunicação bidirecional com um *host* remoto. Ele introduz a interface *WebSocket* e define um canal de comunicação *full-duplex* que opera por meio de um único *socket* na Web (WEBSOCKET.ORG, 2016).

Por não necessitar de duas conexões para a comunicação bidirecional como o *Polling* e o *Long-Polling* (que usam uma conexão para enviar e outra para receber os dados), a API gera uma redução na latência e no tráfego desnecessário da rede, logo, as aplicações baseadas no HTML 5 *WebSocket* geram menos carga aos servidores e permitem o suporte há um número maior de conexões concorrentes. A Figura 13 mostra uma arquitetura básica baseada no *WebSocket* na qual os navegadores usam uma conexão *WebSocket full-duplex* que se comunica diretamente com um *host* remoto.

Figura 13 - Arquitetura básica de comunicação WebSocket entre navegador e host remoto.



Fonte – WEBSOCKET.ORG, 2016

A arquitetura ilustrada na Figura 13 mostra os diversos navegadores Web conectados de forma remota as várias aplicações existentes na Web, sendo que essa conexão se dá pela internet. Essas aplicações por sua vez, utilizam o serviço de troca de mensagens do *WebSocket* que está sempre escutando as mudanças que ocorrem no servidor e levando essas mudanças para os navegadores.

Uma das características principais do *WebSocket* é a habilidade de passar por *Firewalls* e *Proxies*, um problema para muitas aplicações baseadas em outras tecnologias como o *Long-Polling*. Isso se dá porque o *WebSocket* detecta a presença de um servidor de *proxy* e automaticamente define um túnel para passar através do *proxy*. O túnel é estabelecido através da emissão de uma declaração CONNECT HTTP para o servidor *proxy*, que então solicita a abertura de uma conexão TCP/IP para o *host* e porta específica. Uma vez que o túnel está configurado, a comunicação flui livremente através do *proxy* (WEBSOCKET.ORG, 2016).

O HTML 5 *WebSocket* torna a tarefa de se conectar a um *host* mais simples. Utilizando a interface da API para esse fim, apenas é necessário instanciar um novo objeto *WebSocket* com a URL que representa o *host* ao qual se deseja conectar.

```
EX: var myWebSocket = new WebSocket("ws://www.websockets.org");
```

O trecho de código acima ilustra uma instância *WebSocket*. As letras: “`ws://`” no início da URL, são um prefixo para *WebSocket*, também poderiam ser escritas da seguinte maneira: “`wss://`”, sendo que a segunda forma é uma indicação ao *Secure WebSocket* (assim como acontece com o HTTP e o HTTPS).

Como dito na subseção 2.4.1, uma conexão *WebSocket* é estabelecida durante o *handshake*, quando ocorre a atualização do protocolo HTTP para o protocolo *WebSocket*. De forma mais pratica, isso corre através das funções “`onmessage`” e “`send`” definidas pela interface da API. Antes de se conectar a um *host* e enviar uma mensagem, é possível associar uma série de eventos que escutam as mudanças no servidor para lidar com cada fase do ciclo de vida de conexão (WEBSOCKET.ORG, 2016). O trecho de código a seguir ilustra isso:

```
myWebSocket.onopen = function(evt) { alert("Connection open ..."); };
myWebSocket.onmessage = function(evt) { alert( "Received Message: " + evt.data);
};
myWebSocket.onclose = function(evt) { alert("Connection closed."); };
```

Após estabelecer uma conexão o envio de mensagens e o encerramento da conexão podem ser feitas da seguinte maneira:

```
myWebSocket.send("Hello WebSockets!");
myWebSocket.close();
```

Ao se instanciar um objeto *WebSocket* uma série de atributos, eventos e métodos ficam disponíveis. Eles podem ser descritos da seguinte maneira (TUTORIALS POINT, 2016):

Tabela 1 - Atributos associados ao objeto *WebSocket*.

Atributos	**Descrição**
Socket.readyState	É um tribute apenas de leitura, ele representa o estado da conexão podendo assumir os seguintes valores: • O valor **0** indica que a conexão ainda não foi estabelecida; • O valor **1** indica que a conexão foi estabelecida e que é possível se comunicar;

	• O valor **2** indica que a conexão está sendo encerrada através do *handshake*; • O valor **3** indica que a conexão foi encerrada e não pode ser aberta
Socket.readyState	É um atributo apenas de leitura, ele representa o número de bytes de um texto UTF-8 que foram colocados em fila usando o método `send()`

Fonte – TUTORIALS POINT, 2016

Tabela 2 - Eventos associados ao objeto *WebSocket*.

Evento	**Manipulador dos eventos**	**Descrição**
Open	Socket.onopen	Esse evento ocorre quando a conexão é estabelecida.
Message	Socket.onmenssage	Esse evento ocorre quando o cliente recebe dados do servidor
Error	Socket.onerror	Esse evento ocorre quando há algum erro na comunicação.
Close	Socket.onclose	Esse evento ocorre quando a conexão é encerrada.

Fonte – TUTORIALS POINT, 2016

Tabela 3 - Métodos associados ao objeto *WebSocket*.

Método	**Descrição**
Socket.send()	O método `send(data)` transmite dados (tanto texto quanto dados binários) usando a conexão.
Socket.close()	O método `close()` pode ser usado para terminar qualquer conexão existente.

Fonte – TUTORIALS POINT, 2016

Esse conjunto de funcionalidades fornecida sutilmente pela API do *WebSocket* permite aos desenvolvedores implementarem aplicações mais eficientes e que usam pouco código, o que contribui para a redução da latência no tráfego de dados na rede, como dito no início desta seção. A Figura 14 apresenta o código de

uma página HTML que exemplifica de maneira simples a utilização do HTML 5 *WebSocket.*

Figura 14 - Exemplo de implementação do HTML 5 WebSocket.

```
<!DOCTYPE HTML>
<html>
   <head>

      <script type="text/javascript">
         function WebSocketTest()
         {
            if ("WebSocket" in window)
            {
               alert("WebSocket is supported by your Browser!");

               // Let us open a web socket
               var ws = new WebSocket("ws://localhost:9998/echo");

               ws.onopen = function()
               {
                  // Web Socket is connected, send data using send()
                  ws.send("Message to send");
                  alert("Message is sent...");
               };

               ws.onmessage = function (evt)
               {
                  var received_msg = evt.data;
                  alert("Message is received...");
               };

               ws.onclose = function()
               {
                  // websocket is closed.
                  alert("Connection is closed...");
               };
            }

            else
            {
               // The browser doesn't support WebSocket
               alert("WebSocket NOT supported by your Browser!");
            }
         }
      </script>

   </head>
   <body>

      <div id="sse">
         <a href="javascript:WebSocketTest()">Run WebSocket</a>
      </div>

   </body>
</html>
```

Fonte – TUTORIAL POINT, 2016.

Como visto na Figura 14, utilizar os recursos do *WebSocket* através de sua API não é uma tarefa árdua, isso porque a API encapsula códigos extensos em métodos pequenos, sendo necessário ao desenvolvedor apenas utilizar esses

métodos para usufruir dos recursos do *WebSocket*, logo para demonstrar o funcionamento básico da tecnologia é necessário apenas algumas linhas de código. Atualmente, a versão mais recente de *browsers* como o Google Chrome, o Mozilla FireFox, o Internet Explore, o Opera e o Safari, já suportam as especificações da API. Para Ubl e Kitamura (2010) utilizar *WebSocket* é uma ótica forma de aumentar o desempenho, porém, é necessário repensar o modo como se constrói as aplicações no lado do servidor, mantendo o foco em tecnologias que gerenciem as filas de eventos, por exemplo. Alguns casos de uso para o *WebSocket* são:

- Jogos online com múltiplos jogadores;
- Aplicações de bate-papo; e
- Atualização em tempo real de redes sociais;

A presente seção apresentou os diversos aspectos da tecnologia *WebSocket,* demonstrando através de exemplos reais, os diversos recursos presentes na sua API. A seção a seguir apresenta a Metodologia adotado para execução do presente trabalho.

3 METODOLOGIA

O presente trabalho buscou dar continuidade ao desenvolvimento da ferramenta CodeLive. Para isso, foi feita uma atualização da aplicação a fim de fornecer um ambiente mais interativo ao usuário, através do uso da tecnologia *WebSocket* (que permite interações em tempo real) e da inclusão de um interpretador *Python*, para facilitar a execução e análise do código do usuário. As seções a seguir apresentam as tecnologias de softwares que foram utilizadas, bem como os métodos aplicados para o desenvolvimento do trabalho.

3.1. Linguagem de programação e software

A seguir são listadas as linguagens de programação, os *frameworks* e os *softwares* que foram utilizados no desenvolvimento dos módulos do sistema.

Javascript é uma linguagem de programação interpretada pelo navegador, realiza comunicação assíncrona e manipula os elementos do DOM. Apesar de ser uma linguagem regularmente utilizada no lado do cliente (navegador), atualmente, também começa a ser usada no lado do servidor através de ambientes como o *Node.js*. No trabalho, o *JavaScript* foi utilizado tanto no *back-end* (para construir as funções que se comunicam com o banco de dados e processar as informações vindas do *front-end*) quanto no *front-end* (pelo *framework* AngularJs para gerenciar as informações na tela do usuário).

Python é uma linguagem de programação criada por Guido Van Rossum em 1991. Ela suporta múltiplos paradigmas de programação e possui inúmeras bibliotecas, que contém classes, métodos e funções. É bastante utilizada na área cientifica devido a sua legibilidade que possibilita uma tradução fácil do raciocínio em algoritmo. Ela é multiplataforma e permite executar os códigos em um terminal de comandos do sistema operacional. No trabalho, *Python* é a linguagem utilizada pelos usuários no momento de responder um desafio;

Node.js é uma plataforma construída sobre o motor *JavaScript* do navegador Google Chrome e seu propósito é facilitar a construção de aplicações de rede rápidas e escaláveis. O *Node.js* usa um modelo de I/O direcionada a evento não bloqueante que o torna leve e eficiente, ideal para aplicações em tempo real com troca intensa de dados através de dispositivos distribuídos (NODEJS, 2016). Com o *Node.js* é possível utilizar *JavaScript* no lado do servidor (MOREIRA, 2013). No

trabalho, o Node.js é utilizado para executar a aplicação, ou seja, é o servidor de aplicação;

Socket.io o Socket.io oferece uma API *JavaScript* simples, baseada em eventos que te permite a comunicação entre o servidor e o cliente em tempo real. Por padrão, o mecanismo de comunicação do *Socket.io* é o *WebSocket*, porém, se não houver suporte ao *WebSocket* no navegador do usuário, ele recorrerá à *fallbacks*, como *Flash* e *Ajax*, tornando possível sua utilização em um número maior de navegadores. Ele está presente tanto no *back-end* quanto no *front-end*, sendo projetado para rodar sobre a plataforma *Node.js* (ZIM, 2013). No trabalho, o *Socket.io* é utilizado para "escutar" as mudanças e eventos que ocorrem no servidor, e expor essas para o usuário (sendo usado também para emitir notificações, atualizar o ranking de usuários entre outras funções);

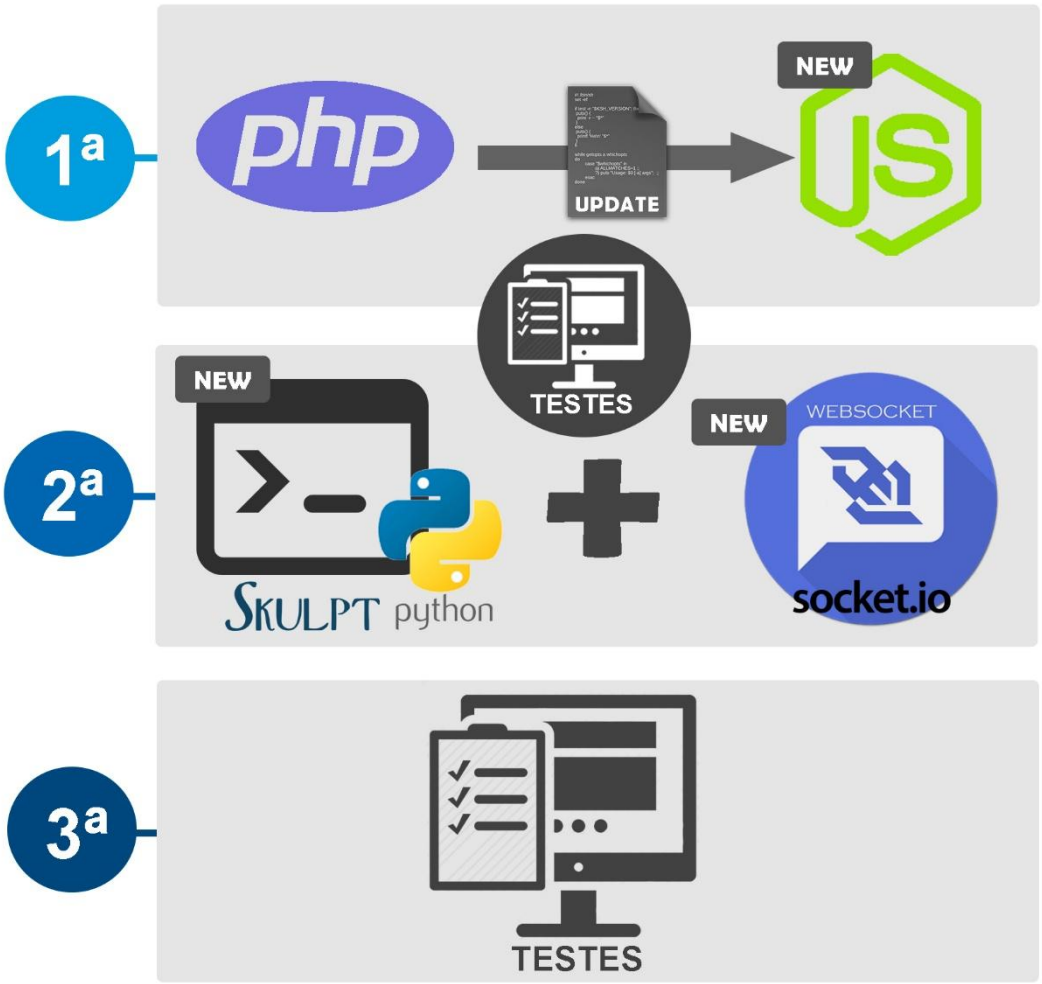
Skulpt é um *framework* gratuito, que executa códigos *Python* no navegador do usuário sem a necessidade de *plug-ins* ou suporte no lado do servidor (GRAHAM, 2016). É utilizado no trabalho para executar e interpretar os códigos dos usuários;

CodeMirror é um editor de texto versátil implementado em *JavaScript* para o navegador. Ele é especializado em edição de código e conta com inúmeros modos de linguagens e *plug-ins* que implementam funcionalidades de edição avançada. Ele também conta com uma API rica em temas para personalizar o ambiente e com isso, proporcionar uma experiência mais próxima ao utilizado nas ferramentas de desenvolvimento especializadas (CODEMIRROR, 2016). O presente trabalho faz uso dessa ferramenta no módulo de responder desafio (seção 4.2.6).

3.2. O desenvolvimento da aplicação

O desenvolvimento do trabalho ocorreu em três em etapas principais, apresentadas na Figura 15, bem como os elementos envolvidos.

Figura 15 - Etapas de desenvolvimento do trabalho.



A primeira etapa do projeto consistiu na atualização do código já existente no *back-end*. Inicialmente, a aplicação foi desenvolvida utilizando a linguagem de programação PHP no *back-end*, portanto, a primeira etapa do projeto consistiu em atualizar o *back-end* da aplicação, que passou a ser desenvolvida em J*avaScript* e agora é executada sobre a plataforma *Node.js* e não mais no *Apache* (como era antes com o PHP).

Após atualizar o *back-end* da aplicação, foram feitos alguns testes para comprovar o funcionamento dos módulos já existentes. Comprovou-se que nem todos os módulos estavam funcionando corretamente, logo, foram aplicadas correções para corrigir as falhas encontradas. A metodologia adotada para os testes, tanto após a atualização do *back-end*, quanto após a implementação das novas funcionalidades, consistiu em utilizar a ferramenta e verificar se as ações feitas produziam o resultado esperado.

A segunda etapa foi de desenvolvimento das novas funcionalidades. Primeiro, foi incluído o interpretador Python (através da ferramenta *Skulpt*) pois a aplicação já possuía o módulo de **responder desafio**, criado para receber, executar e validar o código do usuário. Após a inclusão do interpretador *Python*, implementou-se os métodos que fazem as notificações e atualizações em tempo real de algumas partes da aplicação.

Na etapa final do desenvolvimento, foram feitos os testes (testes funcionais) para validar o funcionamento das novas funcionalidades, além de comprovar a eficiência da aplicação como um todo.

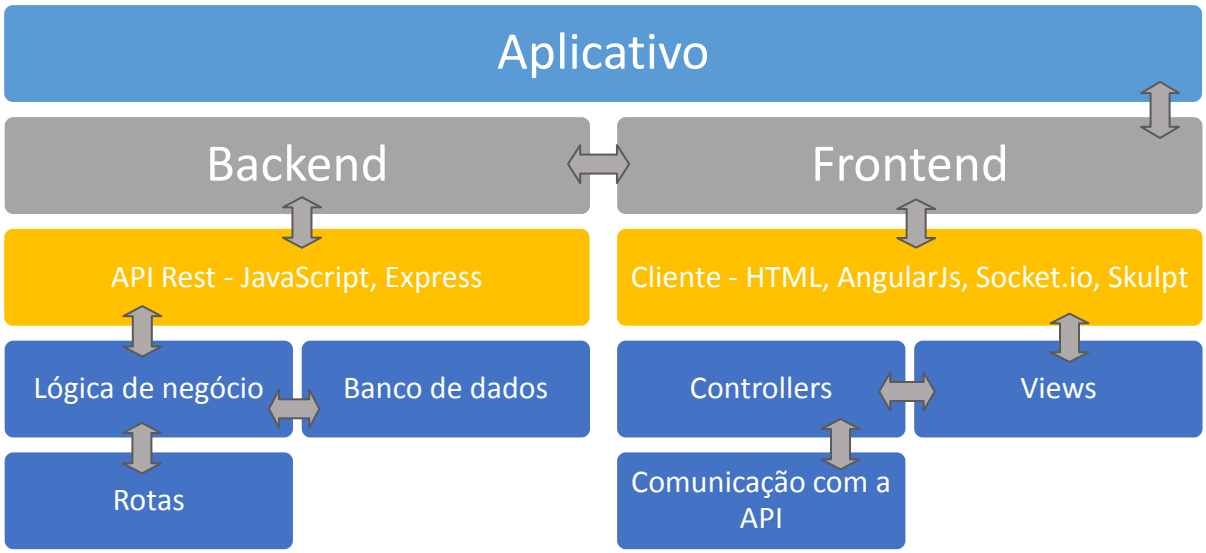
4 RESULTADOS E DISCUSSÃO

Nesta seção são apresentados os resultados obtidos após a atualização do CodeLive, o processo necessário para adaptar a ferramenta ao novo ambiente (desenvolvido em *JavaScript* sobre a plataforma *Node.js*), a arquitetura da ferramenta, o processo de inclusão das novas funcionalidades e a integração (comunicação entre os módulos).

4.1. Arquitetura da Ferramenta

A arquitetura da ferramenta descreve todos os módulos que compõem o CodeLive, as tecnologias utilizadas e a interação entre elas. A arquitetura, como ilustrada na Figura 16, é organizada em camadas, desde as camadas mais externas (camadas de alto nível, às qual o usuário tem contato) até as camadas de baixo nível (compostas pelo *backend*, encontrado no servidor).

Figura 16 - Arquitetura atualizada do CodeLive.



A Figura 16 representa a arquitetura do CodeLive, a qual é dividida em camadas (níveis) que contém diversas tecnologias. A comunicação entre os módulos é representada pelas setas bidirecionais, isso indica que a informação vai de um módulo ao outro e permite que uma informação de mais alto nível (módulo **Aplicativo**, por exemplo) seja processada em um módulo de baixo nível (módulo ***Controllers***, por exemplo) e retorne ao módulo inicial que proveu a informação.

O primeiro módulo (**Aplicativo**) representa a aplicação como um todo e pode ser visto como um agrupador dos demais módulos, é a aplicação de fato.

O segundo nível possui dois módulos (***Backend*** e ***Frontend***) que dividem todo o restante da aplicação. Para um melhor entendimento destes módulos é necessário saber que existem dois tipos principais de linguagem para desenvolvimento na internet, as linguagens: *client-side* e as linguagens *server-side*. Geralmente, essas também são respectivamente conhecidas como: *front-end* e *back-end* ou, ainda, podem ser associadas às camadas de alto nível e baixo nível. As linguagens *front-end* dizem respeito às linguagens interpretadas pelo navegador do usuário, como: HTML, CSS e *JavaScript*. Enquanto que as linguagens do *back-end* se referem as linguagens interpretadas pelo servidor, alguns exemplos são: PHP, ASP.NET e *Python* (TABLELESS, 2016).

Assim, há um fluxo inicial da informação, que inicia no primeiro nível e segue para o segundo nível pela camada de *front-end*, sendo possível uma interação com o *back-end,* já que esse pode trazer informações relevantes para o usuário (uma informação vinda do banco de dados que é exibida na tela do usuário, por exemplo).

O terceiro nível (**API Rest** e **Cliente**) é composto pelas ferramentas que operam dentro das camadas de alto nível e baixo nível. O *front-end* acopla *frameworks* como o *AngularJs* e o *bootstrap* que manipulam o DOM e usam HTML, *JavaScript* e CSS para construir a interface do usuário. Com a atualização do CodeLive, também estão presentes nesse módulo os *frameworks Socket.io* e *Skulpt*, sendo que o primeiro está presente tanto no *back-end* quanto no *front-end*. O *back-end* é composto por uma API *Rest* que fornece os serviços necessários para o *front-end* e que também passou por uma atualização. Antes era utilizada a linguagem PHP e o *Slim framework*, agora, todo o ambiente foi alterado e passa a utilizar a linguagem *JavaScript* e o *framework Express* (sobre o NodeJS) para implementar e distribuir os serviços ao *front-end*.

O quarto nível é formado por dois módulos no *front-end* (***Controllers*** e ***Views***) e dois módulos no *back-end* (**Lógica de negócio** e **Banco de dados**). Eles servem para amplificar a visão das áreas de atuação dos elementos do terceiro nível. O módulo de ***Controllers*** implementa a lógica de negócio necessária para gerenciar a interface do usuário, que por sua vez, é representada pelo módulo ***Views***. Ambos os módulos utilizam o *AngularJs* para realizar suas funções, sendo ele o responsável pela integração com os demais *frameworks front-end*. No *back-end* o módulo de **Lógica de negócio** é o responsável por implementar as funções/serviços que serão utilizados/consumidos pelo *front-end*, além de também

conter as funções *WebSocket* que são responsáveis pelas interações em tempo-real. Esse módulo também é responsável pelo acesso aos dados, e realiza tarefas como: inserir, recuperar, alterar e deletar informações do **Banco de dados**, módulo que também está presente nesse nível da arquitetura e tem o papel de armazenar os dados gerados pela aplicação.

Os módulos **Comunicação com API e Rotas**, presentes respectivamente no *front-end* e *back-end,* constituem a última camada da aplicação, sendo os responsáveis pela comunicação e a interação entre as partes: *front-end* e *back-end*. O primeiro está contido no módulo de ***Controllers*** e utiliza a diretiva *$http* do *AngularJs* para enviar/receber as informações vindas do *back-end*, já o módulo de **Rotas** constrói rotas de acesso as funções *javascript* presentes no módulo **Lógica de negócio**, sendo possível a identificação de um serviço através de uma rota.

Através da análise da arquitetura da aplicação, nota-se que a comunicação entre os módulos ocorre de forma bidirecional, logo essas informações não trafegam apenas do cliente para o servidor, mas também podem ser providas do banco de dados para um *View* do usuário, por exemplo.

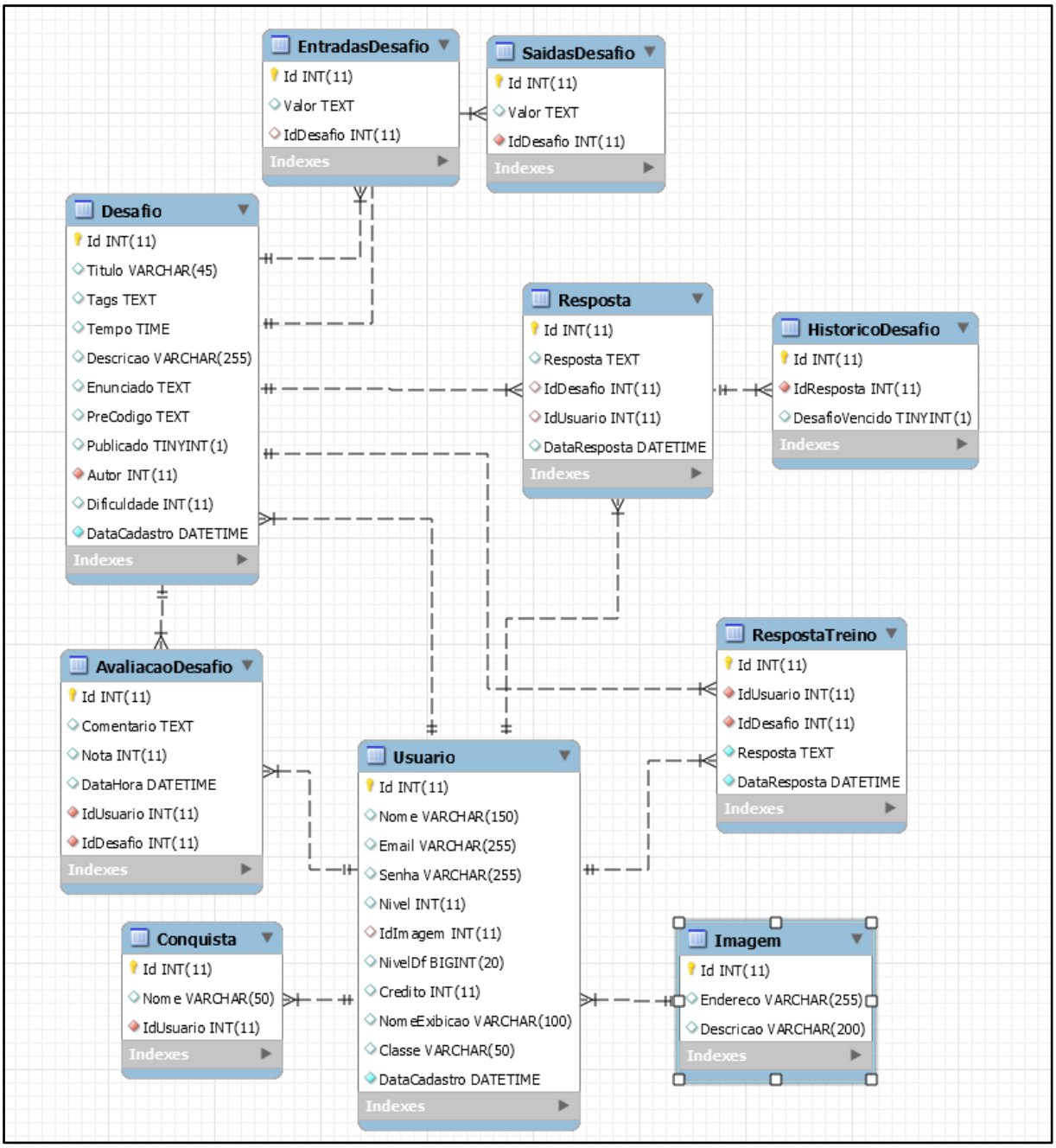
4.2. Back-end

Esta seção tem como objetivo complementar a seção anterior e detalhar os artefatos gerados no que tange ao *back-end*, ou seja, a parte da aplicação existente no lado do servidor. Ela também aborda a estrutura de dados da aplicação, os serviços existentes, a forma como o servidor distribui esses serviços, as ferramentas (*frameworks*) utilizas pelo servidor para realizar as diversas tarefas da aplicação e a composição/configuração do próprio servidor.

4.2.1. Banco de Dados

A camada de bando de dados é a parte da aplicação na qual são armazenadas todas as informações não voláteis (ou seja, informações que não são temporárias), ela se encontra no quarto nível da arquitetura da aplicação e está em constante conexão com o módulo de **Lógica de negócio**, sua estrutura é ilustrada pela Figura 17.

Figura 17 - Diagrama de banco de dados da aplicação.



A Figura 17 ilustra o diagrama de banco de dados da aplicação, essa estrutura é composta por dez tabelas que contêm diversos campos para armazenar os muitos dados da aplicação. O diagrama é composto pelas tabelas “Usuario”, “Imagem”, “Conquistas”, “AvaliacaoDesafio”, ”RespostaTreino”, “Desafio”, “EntradasDesafio”, “SaidasDesafio”, “Resposta” e “HistoricoDesafio”. Os tópicos a seguir descrevem a função de cada uma delas.

- **Usuario** – Essa tabela contém todos os dados do jogador, as principais informações são: “NomeExibicao”, “Senha”, “Classe”, “Nivel”, “NivelDf” e

"Credito", é através desses atributos que o usuário acessa a aplicação, compra os desafios é classificado no *Ranking* dos melhores jogadores e avança no jogo;

- **Imagem** – Essa tabela contém as imagens dos avatares utilizados pelos jogadores, ela também armazena as imagens enviadas (*upload*) pelos usuários no processo de atualização da imagem do perfil;
- **Conquistas** – Essa tabela contém as conquistas que os usuários adquirem ao longo do jogo;
- **AvaliacaoDesafio** – Essa tabela armazena as avaliações que os usuários fazem para os desafios, os campos mais relevantes são: "Nota" (nota atribuída aquele desafio), "Comentario" e IdUsuario (valor que identifica qual usuário realizou a avaliação). É importante destacar que a funcionalidade de avaliar um desafio não estava nos requisitos do presente trabalho e que a mesma não foi implementada, sendo que a tabela foi criada em trabalhos anteriores e poderá ser usada em trabalho futuros;
- **RespostaTreino** – Essa tabela armazena as respostas geradas pelos usuários ao realizar um treino, os dados contidos nela são: "IdUsuario" (valor que identifica qual usuário realizou o treino), "IdDesafio" (valor que identifica qual desafio estava sendo usado para o treino), "Resposta" (código produzido pelo usuário) e a "DataResposta" (refere-se a data e hora em que foi realizado o treino).
- **Desafio** – Essa tabela armazena os dados referente aos desafios disponíveis na aplicação, ela é uma das tabelas mais importantes da aplicação e como pode ser visto na Figura 17, se relaciona com a maioria das demais tabelas. Quase todos os atributos contidos nela são de suma importância para o funcionamento da aplicação, os mais relevantes (sendo até mesmo de preenchimento obrigatório na aplicação) são: "Titulo" (é o nome dado ao desafio, nome esse que deve ser único afim de identificar um desafio), "Tags" (série e palavras-chaves que categorizam o desafio), "Descrição" (texto resumido da tarefa contida no desafio), "Enunciado" (texto completo com a tarefa contida no desafio), Publicado (informa se o desafio está disponível ou não para os usuário, essa informação é preenchida pela aplicação no momento do cadastro e pode ser alterado depois pelo autor do desafio),

"Autor" (usuário que criou o desafio) e a "Dificuldade" (é o nível de dificuldade para resolver o desafio). Os campos "Tempo" e "DataCadastro" são respectivamente de preenchimento opcional e preenchido pela aplicação.

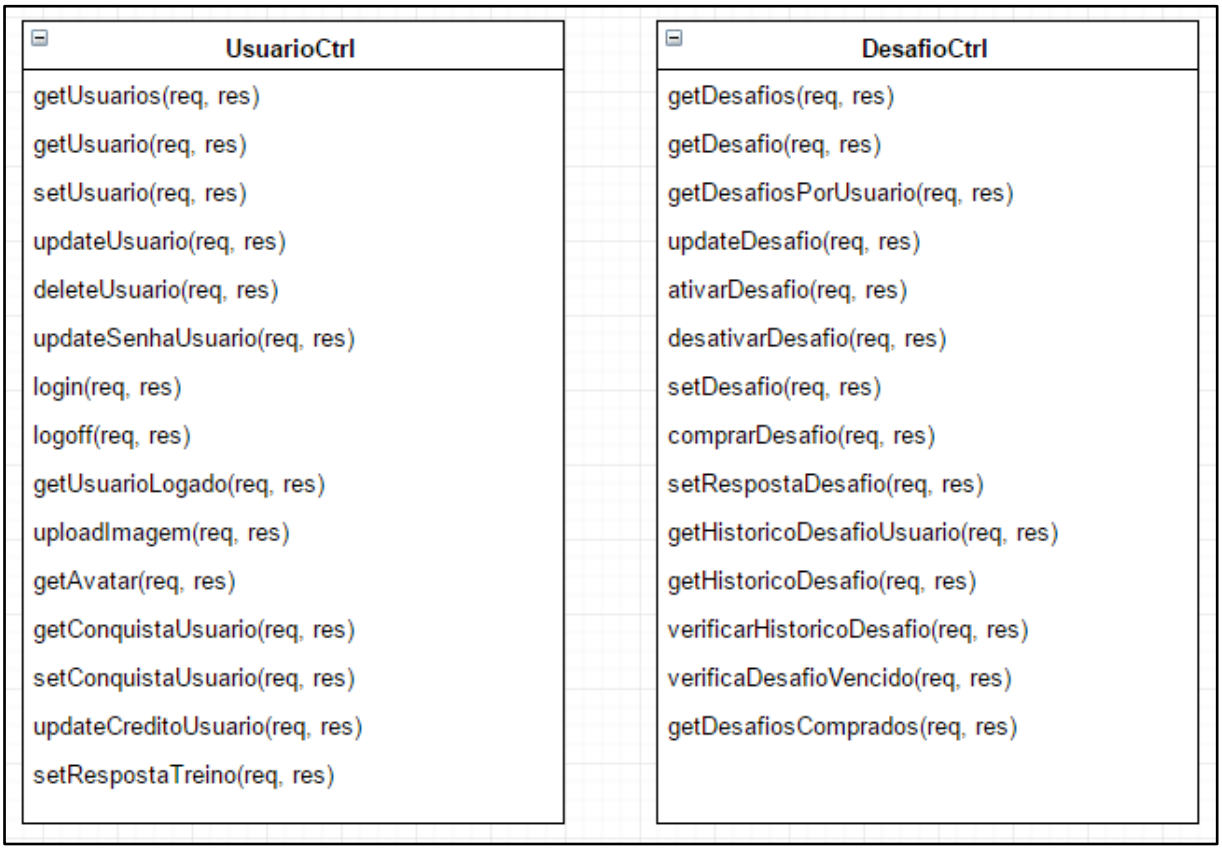
- **EntradasDesafio** – Essa tabela armazena as entradas que o desafio utilizará no momento de validar uma resposta, os atributos mais relevantes são: "Valor" (que armazena um texto que representa a entrada em si) e o "IdDesafio" (que identifica a qual desafio pertence uma determinada entrada).
- **SaidasDesafio** – Essa tabela armazena as saídas que o desafio utilizará no momento de validar uma resposta, ela possui os mesmos atributos da tabela "EntradasDesafio" e ambas as tabelas recebem seus dados no cadastro de um desafio, ou seja, ao mesmo tempo que a tabela "Desafio";
- **Resposta** – Essa tabela é similar a tabela "RespostaTreino", logo contém os mesmos atributos, porém, armazena as respostas feitas pelos usuários após comprarem um desafio, tendo com diferencial o relacionamento com a tabela "HistoricoDesafio" que permite saber a resposta enviada pelo usuário está correta ou não;
- **HistoricoDesafio** – Essa tabela tem o objetivo de fornecer um histórico dos desafios após serem comprados e respondidos, os dados armazenados são: "IdResposta" (valor que identifica a resposta usada para um desafio) e "DesafioVencido" (valor booleano que enforma se uma determinada resposta está correta ou não). É importante deixar claro que um desafio pode ser respondido diversas vezes por diversos usuários, logo, o mesmo tem diversas respostas e com isso um histórico com diversas linhas.

Dentre as tabelas apresentadas, as com maior relevância são as de "Usuario", "Desafio" juntamente com as tabelas de "EntradasDesafio" e "SaidasDesafio" (lembrando que esse conjunto de informações quase sempre estão associadas na mesma tela), "Resposta" e "HistoricoDesafio". A importância das tabelas citadas se dá pelo fato das mesmas fornecerem as informações mais importantes para aplicação, já estas armazenam os dados dos usuários e desafios, os elementos chaves no contexto do CodeLive.

4.2.2. Camada de Lógica de negócio e Controllers

A camada de Lógica de negócio está presente no quarto nível da arquitetura da aplicação e contém diversas funções *javascript*, essas funções são blocos de código e é por meio delas que ações como o acesso a dados e o processamento das informações providas pelo *front-end* acontece. Essa camada é composta por dois c*ontrollers,* "UsuarioCtrl" (presente no arquivo /api/usuário.js) e "DesadioCtrl" (presente no arquivo /api/desafio.js), que possuem *actions*, as funções que de fato processam os dados.

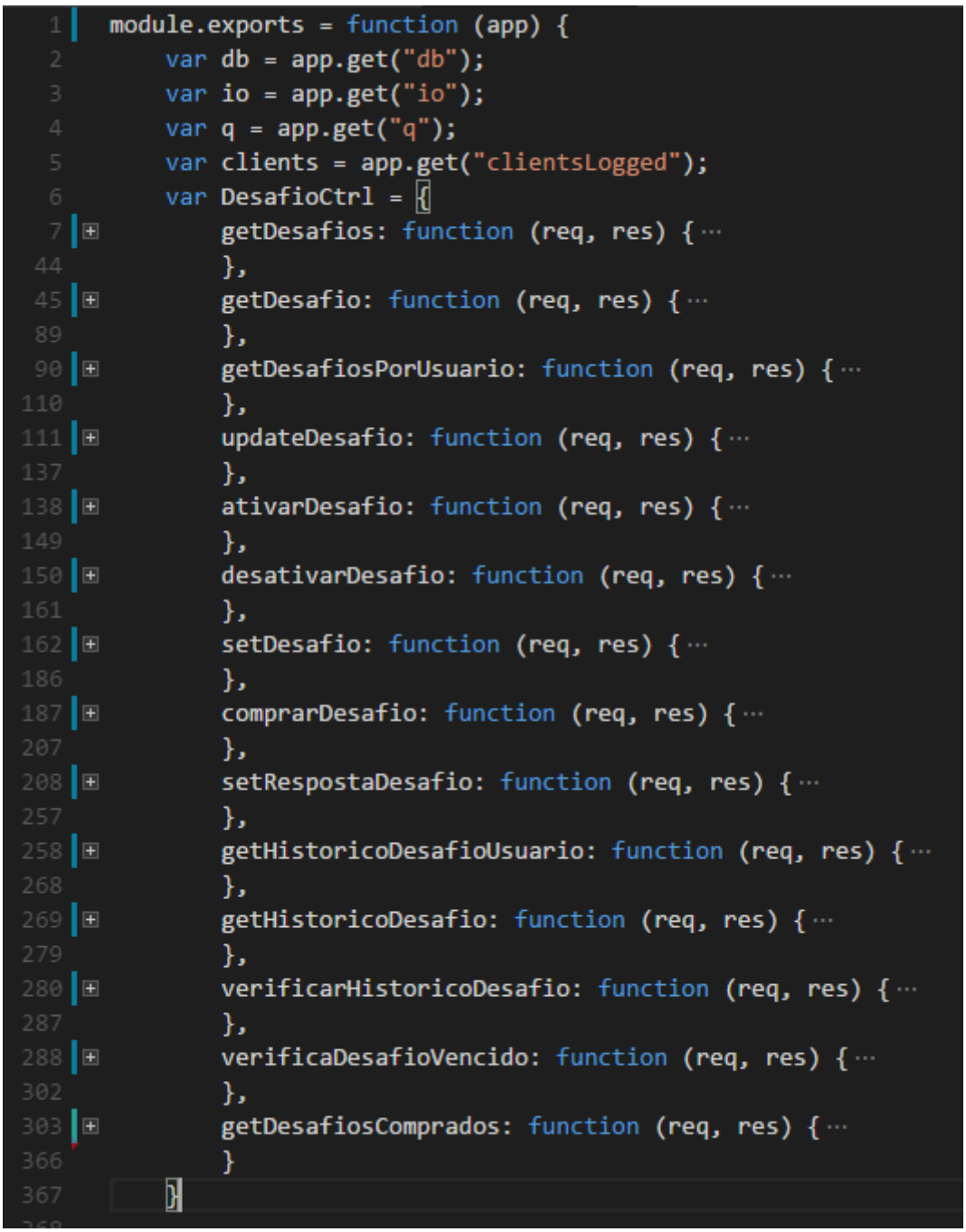
Figura 18 - Representação dos *controllers* e *actions* da API.



A Figura 18 apresenta os *controllers* e suas *actions*, o *controller* à esquerda da imagem trata na sua maioria do processar os dados referentes a entidade do banco de dados "Usuario", enquanto que o *controller* à direita da imagem se preocupa em tratar os dados referente a entidade "Desafio". É importante destacar que por existir uma ligação no banco de dados e na lógica de negócio entre as entidades "Usuario" e "Desafio", os *controllers* citados não processam somente os dados destas entidades em particular, mesmo porque essas entidades necessitam das demais entidades do banco para realizar as funções esperadas pela aplicação.

Dentro do contexto de aplicações *Node.js,* os *controllers* seguem um padrão, isso implica em como eles utilizam os *frameworks* (bibliotecas necessárias para acessar o banco de dados, por exemplo) e em como são construídos.

Figura 19 - *Controller* DesafioCtrl.



A Figura 19 ilustra o *controller* “DesafioCtrl” e as *actions* que o compõe, o código presente na linha 1 é utilizado pelo *Node.js* para identificar um módulo da aplicação, assim o servidor pode injetar (enviar) dados nesses módulos, sendo também uma

forma de reconhecer um determinado arquivo como parte da aplicação. O atributo “app” é provido pelo *framework Express* sendo que este é importado no *Entering Point* da aplicação (explanado com mais detalhes na seção 4.2.3) e injetado em todos os módulos presentes no servidor. Através desse atributo o *Express* possibilita o acesso às bibliotecas/atributos que foram importadas/criados no momento em que o servidor for iniciado. Ele utiliza o método “set()” (para inserir um atributo) e o método “get()” para acessar um atributo.

Os códigos das linhas 2 a 4 instanciam as variáveis “db”, “io” e “q”, fornecendo a estas variáveis os recursos contidos respectivamente nas abibliotecas: “m*ysql”* (para comunicação e manipulação em bando de dados *MySql*), “socket.io” (que prove recursos de comunicação em tempo-real) e “q” (que utilizar o recurso de *promise* do *javascript* a fim de garantir o retorno do resultado do processamento de uma função em uma execução assíncrona). O Código da linha 5 também recebe uma instancia, porém, este se trata de uma *array* dos usuários que estão logados na aplicação.

A partir da linha 6 tem-se o corpo do *controller*, este também segue um padrão, sendo composto pela palavra reservada “var”, um nome (nesse caso “DesadioCtrl”), um sinal de atribuição (“=”) e os sinais que determinam o início de fim do *controller* (“{};”). Seguindo o formato da linguagem *javascript*, um *controller* nada mais é do que um objeto, a onde cada *action* é um atributo que realiza uma ação.

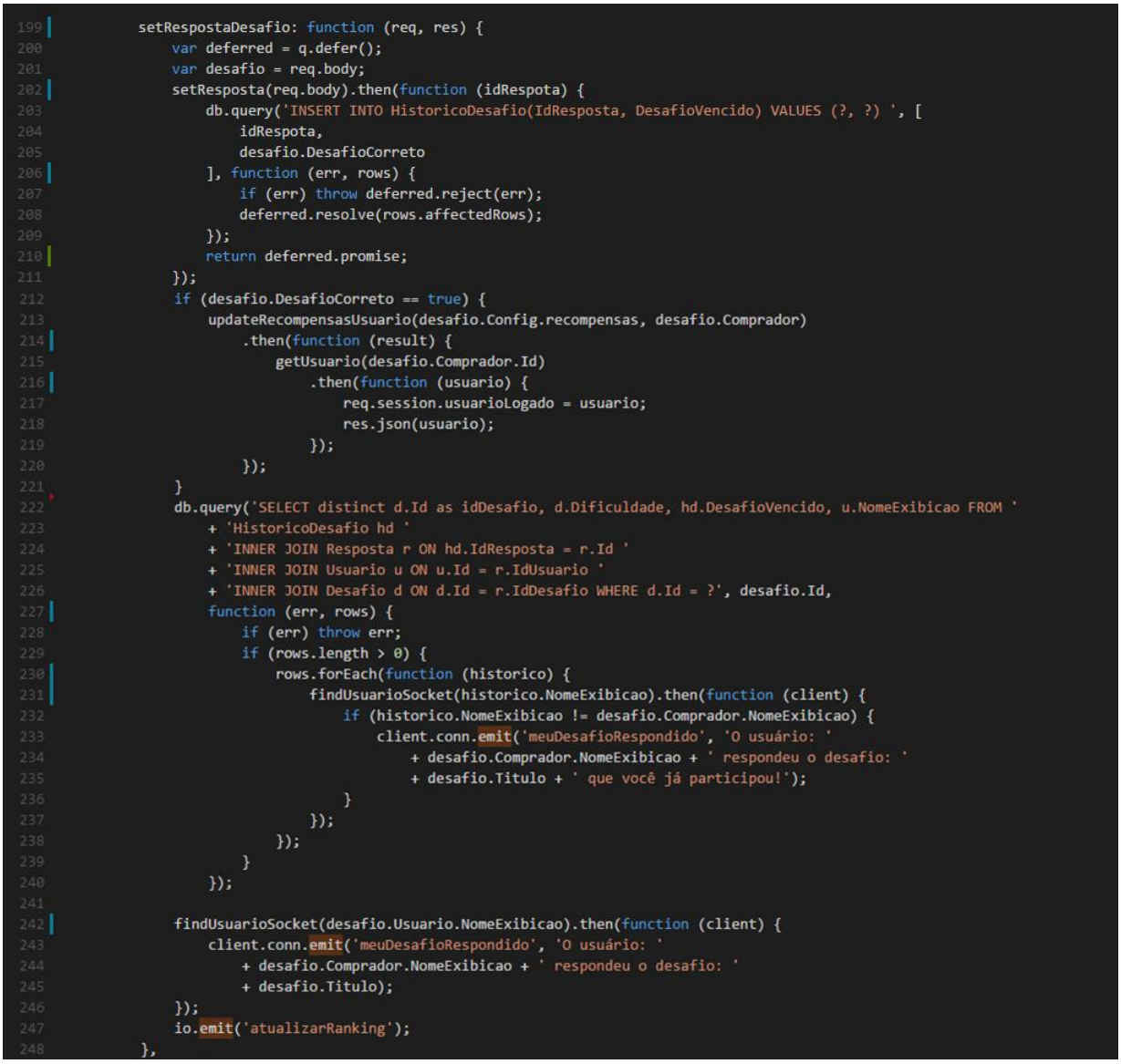
O *controller* “DesafioCtrl” é composto por 14 *actions* sendo que todos têm os mesmos atributos (req e res), estes servem respectivamente para receber um dado externo, afim de processa-lo, e enviar uma resposta. Suas *actions*:

- **getDesafios** – Retorna todos os desafios cadastrados que estão com status de “publicado”;
- **getDesafio** – Retorna um desafio especifico de acordo com o seu id de identificação;
- **getDesafiosPorUsuario** – Retorna todos os desafios de um autor especifico;
- **updateDesafio** – Atualiza os valores das entidades de banco de dados “Desafio”, “EntradasDesafio” e “SaidasDesafio”;
- **ativarDesafio** – Altera o status de publicação de um desafio para “não publicado”;

- **desativarDesafio** – Altera o status de publicação de um desafio para “publicado”;
- **setDesafio** – Insere um novo desafio (incluindo as entradas e saídas que compõem o mesmo) no banco de dados;
- **comprarDesafio** – Registra a aquisição de um desafio por um usuário, além de atualizar os créditos do usuário comprador e atribuir o valor (custo do desafio) ao usuário que criou do desafio;
- **setRespostaDesafio** – Registra a resposta enviada por um usuário a um desafio, além de atualizar dados como: “Credito”, “Nivel”, “Classe”, “NivelDf” e Conquistas do usuário que respondeu o desafio, caso a resposta esteja correta. A *action* também se encarrega de notificar aos usuários que compraram esse desafio, outra pessoa o respondeu, além de também notificar o autor do desafio que alguém respondeu o desafio criado por ele;
- **getHistoricoDesafioUsuario** – Retorna o histórico dos desafios adquiridos por um usuário especifico;
- **getHistoricoDesafio** – Retorna o histórico de um desafio específico;
- **verificarHistoricoDesafio** – Retorna a informação de participação de um usuário em um determinado desafio;
- **verificaDesafioVencido** – Retorna a confirmação de vitória um desafio um por usuário especifico;
- **getDesafiosComprados** – Retorna a lista dos desafios comprados por um usuário, incluindo informações de histórico e respostas dos treinos.

Para complementar o entendimento sobre as *actions* apresentadas, é necessário analisar seu conteúdo, com esse intuito segue a Figura 20 que contém o código da *action* “setRespostaDesafio”, escolhida por utilizar todas as bibliotecas citadas nessa seção.

Figura 20 - Código da *action* setRespostaDesafio.



```
199         setRespostaDesafio: function (req, res) {
200             var deferred = q.defer();
201             var desafio = req.body;
202             setResposta(req.body).then(function (idRespota) {
203                 db.query('INSERT INTO HistoricoDesafio(IdResposta, DesafioVencido) VALUES (?, ?) ', [
204                     idRespota,
205                     desafio.DesafioCorreto
206                 ], function (err, rows) {
207                     if (err) throw deferred.reject(err);
208                     deferred.resolve(rows.affectedRows);
209                 });
210                 return deferred.promise;
211             });
212             if (desafio.DesafioCorreto == true) {
213                 updateRecompensasUsuario(desafio.Config.recompensas, desafio.Comprador)
214                     .then(function (result) {
215                         getUsuario(desafio.Comprador.Id)
216                             .then(function (usuario) {
217                                 req.session.usuarioLogado = usuario;
218                                 res.json(usuario);
219                             });
220                     });
221             }
222             db.query('SELECT distinct d.Id as idDesafio, d.Dificuldade, hd.DesafioVencido, u.NomeExibicao FROM '
223                 + 'HistoricoDesafio hd '
224                 + 'INNER JOIN Resposta r ON hd.IdResposta = r.Id '
225                 + 'INNER JOIN Usuario u ON u.Id = r.IdUsuario '
226                 + 'INNER JOIN Desafio d ON d.Id = r.IdDesafio WHERE d.Id = ?', desafio.Id,
227                 function (err, rows) {
228                     if (err) throw err;
229                     if (rows.length > 0) {
230                         rows.forEach(function (historico) {
231                             findUsuarioSocket(historico.NomeExibicao).then(function (client) {
232                                 if (historico.NomeExibicao != desafio.Comprador.NomeExibicao) {
233                                     client.conn.emit('meuDesafioRespondido', 'O usuário: '
234                                         + desafio.Comprador.NomeExibicao + ' respondeu o desafio: '
235                                         + desafio.Titulo + ' que você já participou!');
236                                 }
237                             });
238                         });
239                     }
240                 });
241 
242             findUsuarioSocket(desafio.Usuario.NomeExibicao).then(function (client) {
243                 client.conn.emit('meuDesafioRespondido', 'O usuário: '
244                     + desafio.Comprador.NomeExibicao + ' respondeu o desafio: '
245                     + desafio.Titulo);
246             });
247             io.emit('atualizarRanking');
248         },
```

O código apresentado pela Figura 20 é bastante extenso, isso se deve ao fato da *action* realizar diversas ações. A primeira ação é realizada pelo método “setResposta” (linha 202), esse método está declarado abaixo do código do *controller*, e assim como outros métodos que se encontram na mesma localidade, o intuito é reduzir o código das *actions* e reaproveitar as funções *javascript*, visto que podem ser usadas por mais de uma *action*.

O método “setResposta” foi construído usando o recurso de *promise* do *javascript*, isso significa que após armazenar a resposta no banco de dados, o dado esperado, nesse caso o id de identificação da resposta cadastra, é retornado pela função. Na linha 203 um acesso a dados é feito utilizando a instância da biblioteca

“mysql” contida na variável “db”. Através do atributo “query” presente nessa variável, é possível escrever um código SQL que insira, recupere, atualize e remova uma certa informação do banco de dados. No caso da linha 203, o código SQL, trata-se de uma inserção na tabela “HistoricoDesafio”, ele armazena o id da resposta (provido no retorno da função “setResposta”) e o dado que informa se a resposta está correta ou não.

Nota-se que as linhas 203 a 205 são destinadas a instrução SQL e seus parâmetros, enquanto que as demais linhas são partes de função *call-back* que contém os atributos “err” e “rows”, sendo estes respectivamente um erro na operação (caso aconteça) e o resultado da operação, ambos são armazenados pela variável “deferred” que na linha 200 recebeu uma instancia da biblioteca “q” e do método “defer()”. É através da variável “deferred” que os valores são armazenados em uma execução assíncrona, o mesmo contém o método “reject” (linha 207), criado para armazenar o erro quando este correr e o método “resolve” (linha 208), que armazena o resultado da instrução SQL, antes de encerrar a função “setResposta” o resultado de todo o processamento é finalmente retornado pelo atributo “promise” (linha 210), contendo este os dados capturados na linha 207 ou na linha 208.

A próxima ação feita pela *action* “setRespostaDesafio” é a de atualizar os dados do usuário que respondeu o desafio, isso ocorre entre as linhas 212 e 221. Vale dizer que a execução esse bloco de código só ocorre mediante a confirmação de que a resposta está correta (esse dado está contido no atributo “desafio.DesafioCorreto”, linha 212). Caso a resposta esteja correta, a função “updateRecompensasUsuario” é chamada e as recompensas são atribuídas ao usuário.

As demais ações desta *action* são voltadas a notificação, a primeira está contida entre as linhas 221 e 239 e realiza um acesso a dados afim de saber quais outros usuários já responderam o desafio que acaba de ser respondido. De posse dessa informação, utiliza-se a função “findUsuarioSocket” para localizar os usuários que estão logados na aplicação e que já responderam o desafio que acabe de ser respondido, e assim notifica-los. A notificação ocorre na linha 232, e é feita utilizando a tecnologia *WebSocket*, por médio da variável “client.conn.emit”, variável essa, que é fruto do retorno da função “findUsuarioSocket” que percorre o *array* “clients” (instanciada na linha 5, Figura 19) e retorna um objeto contendo o nome de exibição

e o endereço de *socket* dos usuários logados. Algo similar ocorre entre as linhas 242 e 246, o diferencial é que nenhuma consulta SQL é realizada e foco da notificação é o Autor do desafio que acaba de ser respondido. Caso o mesmo esteja logado, recebe a notificação de quem um determinando usuário respondeu o desafio criado por ele. A última ação feita na *action* “setRespostaDesafio” também é feita com o uso do *WebSocket*, porém nenhuma mensagem é enviada, apenas ocorre uma chamada para o evento “atualizarRanking”, que atualizar o painel presente no *front-end* com o *Ranking* dos melhores jogadores. Vale destacar que o método usado na última ação dessa *action*, utiliza a variável “io” (instanciada na linha 3, Figura 19) e realiza uma chamada ao evento em todos os *sockets* que estiverem conectados a aplicação.

O *controller* “UsuarioCtrl” é composto por 15 *actions* e segue o mesmo padrão na sua construção e na utilização das bibliotecas que o *controller* “DesafioCtrl”, a principal diferença está no conteúdo e na função que cada *action* realiza. As *actions* do “UsuarioCtrl” são melhores detalhas nos tópicos a seguir.

- **getUsuarios** – Retorna uma lista de todos os usuários cadastrados no banco de dados;
- **getUsuario** – Retorna um usuário especifico de acordo com o seu id de identificação;
- **setUsuario** – Insere um novo usuário no banco de dados, permitindo assim que o mesmo possa acessar a aplicação;
- **updateUsuario** – Atualiza os dados “Nome”, “NomeExibicao”, e “Email do de um usuário especifico;
- **deleteUsuario** – Remove um determinado usuário do bando de dados;
- **updateSenhaUsuario** – Faz o mesmo processo da *action* **updateUsuario**, porém, o único atributo afetado é o de “Senha”;
- **login** – Autentica o usuário na aplicação através dos atributos “NomeExibicao” e “Senha”, caso esses dados estejam corretos, os armazena em uma variável seção controlado pelo *framework Express*;
- **logoff** – Atribui o valor “*null*” a variável que guarda a informação de qual usuário está logad,o de forma a invalidar a seção atual, e forçar o usuário a sair do sistema;

- **getUsuarioLogado** – Retorna as informações do usuário logado através da variável de seção criada na *action* **login**;
- **uploadImagem** - Realiza a ação de *upload* da foto de perfil do usuário, salvando a mesma no servidor e armazenado o caminho para acessa-la no banco de dados. Também atualiza o valor do atributo “IdImagem” (presente na tabela de “Usuario”), afim de relacionar o caminho da imagem que acabe de ser armazenada com o usuário que realizou a ação;
- **getAvatar** – Retorna uma lista das imagens utilizadas por padrão como avatares no sistema, essas são as mesmas imagens apresentadas na tela de cadastro do usuário;
- **getConquistaUsuario** – Retorna uma lista com as conquistas de um determinado usuário;
- **setConquistaUsuario** – Insere uma nova conquista para um determinado usuário;
- **updateCreditoUsuario** – Atualiza o valor do atributo “Credito” de um determinado usuário. Caso a operação seja realizada com sucesso, a *action* também realiza uma nova consulta no banco de dados pelo usuário que acaba de ter os créditos atualizados, de posse dessa informação, ocorre uma atualização nos dados da variável de seção e está por sua vez é retorna pela *action*;
- **setRespostaTreino** – Insere uma reposta referente ao desafio respondido no modulo de treino.

Assim como no *controller de* “DesafioCtrl”, o *controller* “UsuarioCtrl” também usa funções externas as suas *actions*, a principal delas é a função “getUsuario(idUsuario)”, ilustrada na Figura 21.

Figura 21 - Função getUsuario.

```
217     function getUsuario(idUsuario) {
218         var deferred = q.defer();
219         db.query('SELECT * FROM Imagem i INNER JOIN Usuario u ON i.Id = u.IdImagem WHERE u.Id = ?', idUsuario,
220             function (err, rows) {
221                 if (err) throw err;
222                 deferred.resolve(rows[0]);
223             });
224         return deferred.promise;
225     }
```

Essa função é utilizada sempre que há a necessidade de obter os dados de um usuário especifico, assim como as diversas funções da Figura 20, a função

“getUsuario” também utiliza o recurso de “promise”, que como dito, garante o retorno dos dados esperados após o processamento feito pela mesma. O único atributo necessário para fazer uso dessa função é o “IdUsuario”, valor que identifica um usuário no bando de dados.

4.2.3. Camada de serviços (API REST) e web

Para que os serviços que são gerados nos *controllers* e nas demais partes do *back-end* funcionem, é necessário a criação de um arquivo que importe as bibliotecas que o projeto precisará, que configure alguns parâmetros (acesso a banco de dados, por exemplo) e que defina a localização dos arquivos de rotas, arquivos estáticos, dentro outros arquivos. A definição desse arquivo deve ser seguida em qualquer aplicação *Node.js,* sendo a partir da sua execução que a aplicação inicia de fato.

Essas definições se encontram no arquivo “/app.js”, que é chamado no presente trabalho de ***Entering Point*** (ponto de entrada). Para facilitar a entendimento, o conteúdo do *Entering Point* será mostrado em partes e estas serão então detalhadas.

Figura 22 - Definição das bibliotecas do *Entering Point*.

```
var express = require('express'),
  load = require('express-load'),
  app = express(),
  mysql = require('mysql'),
  session = require('express-session'),
  bodyParser = require('body-parser'),
  http = require('http').Server(app),
  io = require('socket.io').listen(http),
  q = require('q'),
  path = require('path'),
  multer = require('multer'),
  port = process.env.PORT || 3000,
  storage = multer.diskStorage({
    destination: './app/assets/img/fotosUpload/',
    filename: function (req, file, cb) {
      cb(null, file.originalname.replace(path.extname(file.originalname), "") + '-' + Date.now() + path.extname(file.originalname))
    }
  });
var clients = [];
var upload = multer({ storage: storage })
//Local
var connection = mysql.createConnection("mysql://root@localhost/CodeLive2");

app.set('io', io);
app.set("db", connection);
app.set("q", q);
app.set("upload", upload);
app.set("clientsLogged", clients);
```

A Figura 22 ilustra parte do conteúdo do *Entering Point*, o código exibido trata na sua maioria da importação das bibliotecas e da definição de parâmetros, afim de utiliza-los nos módulos da ferramenta. Sobre a importação dos *frameworks,* é interessante destacar que os mesmos não necessitam da declaração do caminho

absoluto (endereço físico em que os arquivos se encontram no servidor) para que o *Node.js* os encontre. Isso ocorre por conta do gerenciador de pacotes do *Node.js*, este por sua vez se chama NPM. Sempre que uma nova biblioteca é incluída no projeto através do NPM, a mesma é salva na pasta "node_modules", pasta tida como padrão para *Nojde.js*, logo qualquer módulo (ou *framework*) contidos nessa pasta não necessitam da declaração do caminho absoluto. Os tópicos a seguir detalham as bibliotecas e variáveis definidos entre as linhas 1 e 22 da Figura 22.

- **express** – É uma variável que recebe a instancia do "express", sendo este o principal *framework* pois fornece métodos para a criação de servidores HTTP, tornando possível a criação da *API Rest* e do acesso a ela. Ele também é utilizado para mapear arquivos estáticos (HTML, CSS e *JavaScript*), e encapsular as instancias das bibliotecas e variáveis declaradas no *Entering Point*, para que estas sejam utilizadas nos módulos da aplicação;
- **load** – Variável que recebe a instancia da biblioteca "express-load", logo, pode ser considerada como um modulo do "e*xpress"*, sendo utilizado para mapear os arquivos de rotas e de *controllers*;
- **app** – Variável que recebe uma instancia do método "Express()", possibilitando a esta, utilizar todas as funcionalidades do *framework*;
- **mysql** – Variável que recebe a instancia da biblioteca *"mysql"*, que fornece os métodos necessário para criar uma conexão, e manipular um banco de dados MySql;
- **session** – Variável que recebe a instancia da biblioteca "express-session", assim como na variável "load", essa biblioteca também é um módulo do "*express*" e seu papel é de possibilitar a configuração e criação de variáveis de sessão;
- **bodyParser** – Variável que recebe a instancia da biblioteca "body-parser", e permite que os dados trafegados pela API sejam convertidos para o formato *Json*;
- **http** – Variável que recebe a instancia da biblioteca "http", sendo esta nativa do *Node.js* e criada para permitir diversas interações com o protocolo de internet HTTP. A mesma é utilizada pelo *socket.io* (para que o mesmo saiba em qual porta irá executar) e para iniciar o servidor;

- **io** – Variável que recebe a instancia da biblioteca “*socket.io*”, utilizada para permitir as interações em tempo-real realizadas pela aplicação;
- **q** – Variável que recebe a instancia da biblioteca “q”, construída para facilitar a utilização do recurso de *promise* do *javascript*;
- **path** – Variável que recebe a instancia da biblioteca “path”, tendo essa métodos que permitem extrair informações como nome e extensão de um arquivo;
- **multer** – Variável que recebe a instancia da biblioteca “multer”, sendo esta um *middleware Node.js* para manipulação de *multipart/form-data*, que é usado principalmente para o *upload* de arquivos. A ferramenta permite ainda manipular o arquivo, podendo alterar seu nome e armazena-lo em alguma pasta do servidor;
- **port** – Variável que define em qual porta o servidor irá rodar. Foi definida como padrão a porta 3000, porem caso essa esteja sendo utilizada, o método “process.env.PORT” fornecida pelo *Node.js* localiza uma outra porta e inicia o serviço;
- **storage** – Variável que contém um *string* com a localização da pasta em que foi armazenada da foto de perfil enviada pelo usuário no processo de *upload*;
- **clientes** – *Array* que armazenas os usuário que estão logados na aplicação;
- **upload** – Variável que contém um objeto fornecido pelo *framework* “multer” com a localização da pasta em que foi armazenada a foto de perfil enviada pelo usuário no processo de *upload*;
- **connection** – Guarda a instancia da conexão criada com o banco de dados da aplicação.

O restante do código da Figura 21 (linhas 24 a 28) contém algumas das variáveis declaradas no início do código, essas variáveis estão utilizando o método “set()” do *Express* para armazenar seus valores e permitir que estes sejam utilizados nos demais módulos da aplicação. Um exemplo de utilização foi demonstrado na Figura 19 da seção 4.2.2 (pelo código do *controller* “DesafioCtrl”).

Figura 23 – Definição dos arquivos estáticos no *Entering Point*.

```
76  //front-end da aplicação
77  app.use(express.static(__dirname + "/app"));
78  //frameworks js e css
79  app.use(express.static(path.join(__dirname, '/node_modules')));
80  //converte as requisições para json
81  app.use(bodyParser.json());
82  //configuração da sessão
83  app.use(session({
84    secret: 'keyboard cat',
85    resave: true,
86    saveUninitialized: true,
87    cookie: { secure: false }
88  }));
89
90  //mapea o arquivo de rotas e os controllers do back-end
91  load('models').then('api').then('rotas').into(app);
92
93  http.listen(port, function () {
94    console.log("Servidor rodando na porta: " + port);
95  });
```

A Figura 23 apresenta em sua maioria o código destinado a definição dos documentos estáticos da aplicação e a configuração de alguns serviços. O código da linha 77 é o responsável por informar ao servidor em qual pasta do projeto se encontra os arquivos de *front-end*, assim quando o servidor é iniciado a página HTML apresentada está contida nesta pasta (pasta “/app”). O código da linha 79 faz algo similar, porém, seu objetivo é mapear a pasta que contém os *frameworks* baixados pelo gerenciador de pacotes NPM, e com isso permiti que os arquivos do *front-end* os acessem. O último código destinado ao mapeamento de pastas é do da linha 91, ele utiliza o *framework express-load* para mapear as pastas “/api” e “/rotas” afim de que o *Node.js* reconheça os arquivos contidos essas pastas como módulos da aplicação.

Os demais códigos são de configuração, sendo que o primeiro (linha 81) é usado para permitir que o *body-parser* intercepte as requisições da API e converta seus dados para o formato *Json*. Os códigos das linhas 83 a 88 realizam a configuração da seção, afim permitir a criação variáveis de seção que armazenam dados por um determinado tempo. Ao fim do documento há o código que inicia o servidor, este está entre as linhas 93 e 95 e utiliza a variável “http” seguida do método “listen”. O método “listen” recebe a variável que contém a porta em que o serviço irá ser executado e uma função *call-back, apenas* para emitir uma

mensagem no console do *Node.js*, informado o número da porta no qual o serviço está sendo executado.

Com a definição do *Entering Point* os serviços estão finalmente disponíveis para acesso, entretanto, o acesso a eles deve ser feito seguindo algumas regras, estas regras estão definidas no arquivo “/rotas/rotas.js”. Este arquivo como o próprio nome deixa claro, contém as rotas, ou seja, o endereço de acesso dos serviços disponibilizados pela API.

Figura 24 - Arquivo com as rotas de acesso aos serviços da API.

```
module.exports = function (app) {
    var upload = app.get('upload');
    var usuario = app.api.usuario;
    app.post('/login', usuario.login);
    app.get('/logoff', usuario.logoff);
    app.get('/get-usuario-logado', usuario.getUsuarioLogado);
    app.post('/usuario', usuario.setUsuario);
    app.get('/usuario', usuario.getUsuarios);
    app.get('/usuario/:id', usuario.getUsuario);
    app.put('/usuario/:id', usuario.updateUsuario);
    app.delete('/usuario/:id', usuario.deleteUsuario);
    app.put('/update-senha-usuario/:id', usuario.updateSenhaUsuario);
    app.post('/upload-imagem', upload.single('file'), usuario.uploadImagem);
    app.get('/get-avatares', usuario.getAvatar);
    app.get('/get-conquistas-usuario/:id', usuario.getConquistaUsuario);
    app.post('/set-conquista-usuario', usuario.setConquistaUsuario);
    app.post('/update-credito-usuario', usuario.updateCreditoUsuario);
    app.post('/update-credito-usuario', usuario.updateCreditoUsuario);
    app.post('/responder-treino', usuario.setRespostaTreino);

    var desafio = app.api.desafio;
    app.get('/desafio', desafio.getDesafios);
    app.post('/desafio', desafio.setDesafio);
    app.get('/desafio/:id', desafio.getDesafio);
    app.put('/desafio/:id', desafio.updateDesafio);
    app.get('/desafios-por-usuario/:id', desafio.getDesafiosPorUsuario);
    app.put('/ativar-desafio/:id', desafio.ativarDesafio);
    app.put('/desativar-desafio/:id', desafio.desativarDesafio);
    app.post('/comprar-desafio', desafio.comprarDesafio);
    app.post('/responder-desafio', desafio.setRespostaDesafio);
    app.get('/historico-desafio-usuario/:id', desafio.getHistoricoDesafioUsuario);
    app.get('/historico-desafio/:id', desafio.getHistoricoDesafio);
    app.get('/verifica-historico-desafio/:id', desafio.verificaHistoricoDesafio);
    app.post('/verifica-desafio-vencido', desafio.verificaDesafioVencido);
    app.get('/get-desafios-comprados/:id', desafio.getDesafiosComprados);
}
```

A Figura 24 contém o código usado para definir as rotas de acesso aos serviços disponibilizados pela API, note-se que os serviços se tratam das *actions* que compõem dos dois *controllers* da aplicação, e rotas por tanto, são um meio para utiliza-los. Como dito, ao acessar os serviços algumas regras devem ser seguidas,

nesse caso, o formato da rota. As rotas possuem um verbo HTTP, sendo os mais populares o POST (para enviar dados no corpo da requisição), o GET (para receber ou, enviar dados pela URL), o PUT (para atualizar dados) e o DELETE (para remover algum dado). Além dos verbos HTTP, uma rota ainda tem um nome que a identifica (“/desafio”, por exemplo) e pode conter alguns parâmetros (se o utilizar o verbo GET, PUT ou DELETE). No arquivo de rotas da Figura 24 temos a utilização de todos os verbos HTTP citados, também há utilização dos códigos “app.api.usuario” e “app.api.desafio”, estes são utilizados para ter acesso as *actions* dos *controllers*. Todas as rotas da imagem são melhores detalhas nos tópicos a seguir.

Rotas referentes aos serviços do *controller* UsuarioCtrl:

- **/login** – Dá acesso ao serviço de login para autenticar um usuário;
    - **Verbo:** POST;
    - **Parâmetros:** NomeExbição e Senha;
- **/logoff** – Dá acesso ao serviço de logoff que invalida a seção do usuário e força o mesmo a sair do sistema;
    - **Verbo:** GET;
    - **Parâmetros:** Id;
- **/get-usuario-logado** – Retorna um objeto *Json* com os dados do usuário que está logado;
    - **Verbo:** GET;
    - **Parâmetros:** sem parametros;
- **/usuario**
    - **Verbo:** POST;
    - **Parâmetros:** Nome, NomeExibicao, Email, Senha e IdImagem;
- **/usuario**
    - **Verbo:** GET;
    - **Parâmetros:** sem parâmetros;
- **/usuario/:id** – Retorna um objeto *Json* com os dados de um usuário especifico;
    - **Verbo:** GET;
    - **Parâmetros:** Id;

- **/usuario/:id** – Realiza a atualização de alguns dos dados de um usuário especifico**,**
    - **Verbo:** PUT;
    - **Parâmetros:** Id (na URL), Nome, NomeExbição e Email (no corpo da requisição);
- **/usuario/:id** – Dá acesso ao serviço que deleta um usuário especifico do banco de dados;
    - **Verbo:** DELETE;
    - **Parâmetros:** Id;
- **/update-senha-usuario/:id** – Dá acesso ao serviço que atualiza a senha de um usuário especifico**;**
    - **Verbo:** PUT;
    - **Parâmetros:** Id (na URL) e Senha (no corpo da requisição);
- **/upload-imagem, upload.single('file')** – Dá acesso ao serviço que faz o *upload* da foto de perfil do usuário;
    - **Verbo:** POST;
    - **Parâmetros:** file;
- **/get-avatares** – Retorna a lista das imagens padrões (imagens pré-cadastradas) utilizadas pela aplicação no cadastro do usuário;
    - **Verbo:** GET;
    - **Parâmetros:** sem parâmetros;
- **/get-conquistas-usuario/:id** – Retorna a lista de conquistas de um usuário especifico;
    - **Verbo:** GET;
    - **Parâmetros:** sem parametros;
- **/set-conquista-usuario'** – Dá acesso ao serviço que armazena um conquista de um determinado usuário no banco de dados;
    - **Verbo:** POST;
    - **Parâmetros:** Conquista e Id;
- **/update-credito-usuario** – Dá acesso ao serviço que atualiza o crédito de um usuário especifico;
    - **Verbo:** POST;
    - **Parâmetros:** Credito e Id;

- **/responder-treino** – Dá acesso ao serviço que armazena a resposta de um determinado usuário feito no ambiente de treino;
    - **Verbo:** POST;
    - **Parâmetros:** IdUsuario, IdDesafio e Resposta.

Rotas referentes aos serviços do *controller* DesafioCtrl:

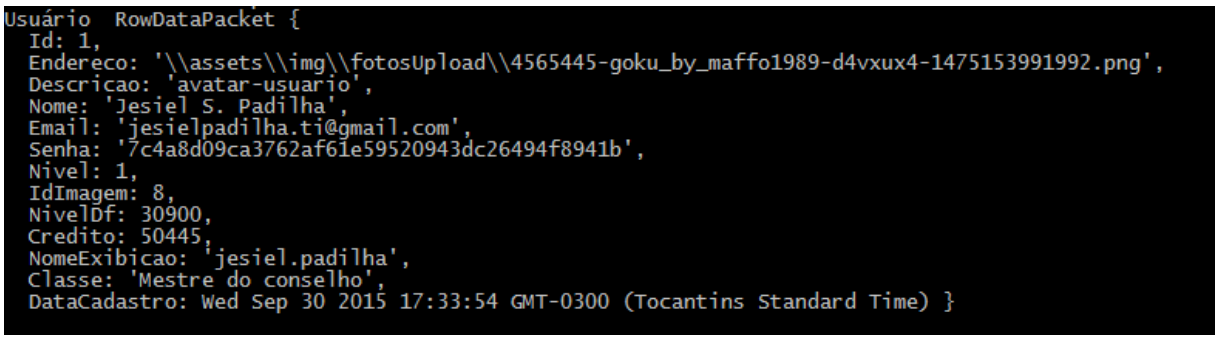
- **/desafio** – Retorna uma lista de objetos de desafios no formato *Json*;
    - **Verbo:** GET;
    - **Parâmetros:** sem parâmetro;
- **/desafio** – Dá acesso ao serviço que armazena um desafio no banco de dados;
    - **Verbo:** POST;
    - **Parâmetros:** Titulo, Tags, Tempo, Descricao, Enunciados, PreCodigo, Dificuldade, Autor, Entradas e Saidas;
- **/desafio/:id** – Retorna um objeto *Json* com os dados de um desafio especifico;
    - **Verbo:** GET;
    - **Parâmetros:**
- **/desafio/:id** – Dá acesso ao serviço que atualiza os dados de um determinado desafio;
    - **Verbo:** PUT;
    - **Parâmetros:** Titulo, Tags, Tempo, Descricao, Enunciados, PreCodigo, Dificuldade, Entradas e Saidas;
- **/desafios-por-usuario/:id** – Retorna uma lista de objetos de desafio no formato *Json* que pertençam a um determinado autor;
    - **Verbo:** GET;
    - **Parâmetros:** Id
- **/ativar-desafio/:id** – Dá acesso ao serviço que altera o status de um determinado desafio para “publicado”;
    - **Verbo:** PUT;
    - **Parâmetros:** Id;
- **/desativar-desafio/:id'** – Dá acesso ao serviço que altera o status de um determinado desafio para “publicado”;

    - **Verbo:** PUT;
    - **Parâmetros:** Id;
- **/comprar-desafio'** – Dá acesso ao serviço que registra a compra de um determinado desafio por um usuário;
    - **Verbo:** POST;
    - **Parâmetros:** Comprador (objeto), Autor, Desafio (objeto);
- **/responder-desafio** – Registra uma resposta enviada pelo usuário no banco de dados;
    - **Verbo:** POST;
    - **Parâmetros:** Resposta, Id, Comprador (objeto) e DesafioCorreto;
- **/historico-desafio-usuario/:id** – Retorna alguns dados do histórico dos desafios de um determinado usuário;
    - **Verbo:** GET;
    - **Parâmetros:** Id;
- **/historico-desafio/:id** – Retorna alguns dados do histórico de um determinado desafio;
    - **Verbo:** GET;
    - **Parâmetros:** Id;
- **/verifica-historico-desafio/:id** – Dá acesso ao serviço que verifica se um determinado desafio já foi respondido;
    - **Verbo:** GET;
    - **Parâmetros:** Id;
- **/verifica-desafio-vencido** – Dá acesso ao serviço que informa se um usuário já venceu um determinou desafio;
    - **Verbo:** POST;
    - **Parâmetros:** idDesafio, idUsuario;
- **/get-desafios-comprados/:id** – Retorna uma lista com os desafios que o usuário já comprou;
    - **Verbo:** GET;
    - **Parâmetros**; Id.

De posse da informação sobre cada rota contida no *back-end*, pode-se fazer uma análise mais aprofundada, explorando outros aspectos que as envolve. Sabe-

se que essas rotas são acessadas através de requisições HTTP, portanto existe códigos que informam a situação da requisição, indicando que ela foi bem-sucedida (200), que um determinado arquivo não foi encontrado (404) e até mesmo se ocorreu algum erro no servidor (500). Analisando a rota “/usuário/:id” (Figura 25) tem-se a seguinte situação.

Figura 25 - Resultado da requisição a rota "/usuário/:id"



A Figura 25 ilustra o resultado da requisição a rota “/usuário:id”, foi informado o valor “1” (referente id do usuário que se deseja obter as informações) e o código HTTP nesse caso foi o 200, e o resultado obtido foi um objeto no formato *Json*. É importante destacar que a requisição foi feita diretamente no navegador (acessando o endereço “http://localhost:3000/usuario/1) e que o tratamento do resultado obtido é feito pelos *controllers* no *front-end* da aplicação, logo se for informado um valor de “id” que não exista o código continuará sendo o 200, porém o resultado será um objeto vazio, sendo este devidamente tratado no *front-end*. Caso a API receba uma requisição para uma URL não existe (que não retorne um serviço descrito nas rotas, por exemplo) o código retornado é o 404, tendo a seguinte mensagem no corpo da requisição: “Cannot GET /{url-informada}”, sendo o parâmetro “{url-informada}” o valor invalido informado pelo usuário na URL.

4.2.4. Comunicação via WebSocket

Apesar da comunicação (ou acesso aos recursos) entre o *front-end* e o *back-end* ocorrer através das rotas citadas na seção anterior (Figura 24), o acesso aos recursos de tempo-real construídos com a tecnologia *WebSocket* utiliza outro meio para troca de dados. Essa comunicação ocorre a partir de eventos, sendo estes identificados por um nome e disparados quando algo acontece no lado do cliente ou do servidor. Um pouco do funcionamento dos recursos implementados utilizando o *Socket.io* foi mostrado na seção 4.2.2 (Figura 20), porém, para um entendimento

completo acerca do funcionamento da tecnologia, é necessário analisar não só o *back-end* da aplicação, mas também o *front-end* e como estes interagem si.

Figura 26 - Função que troca mensagens via WebSocket.

Front-end:

```
14      $scope.login = function (usuario) {
15          if ($scope.usuario.Senha != undefined && $scope.usuario.NomeExibicao != undefined) {
16              $http.post(host + "login", JSON.stringify({ usuario: usuario })).success(function (data) {
17                  if (data != "") {
18                      socket.emit("novoUsuarioLogado", data);
```

Back-end:

```
23  io.on('connection', function (socket) {
24    socket.on('novoUsuarioLogado', function (data) {
25      socket.broadcast.emit('novoUsuarioLogado', data.NomeExibicao + " entrou no jogo!");
26    });
27  });
```

Front-end:

```
106      socket.on('novoUsuarioLogado', function (msg) {
107          notification('info', 'AVISO', msg);
108       });
```

A Figura 26 ilustra uma série de funções *WebSocket* utilizadas para notificar um usuário sobre a entrada de um outro usuário na aplicação. Essas funções utilizam o *framework Socket.io* tanto no lado do cliente quanto no lado do servidor, e o código na Figura 26 demonstra o que é necessário para que essa troca de dados seja possível. Nesse caso, o *front-end* possui um método que envia um dado, e outro que aguarda um dado (respectivamente, linha 18 da primeira imagem e linha 106 a terceira imagem). No *back-end* o processo é similar, salvo o fato da mensagem ser enviada a todos os *sockets* conectados, exceto o *socket* que enviou o dado (faz isso usando o método "*broadcast*", linha 25), em outras palavras o usuário que entrou no jogo não é notificado, somente os demais usuários que já estavam conectados.

Como foi dito, o *WebSocket* trabalha a partir de eventos, um exemplo dos eventos fornecidos pela API do *WebSocket* se encontrado na seção 2.5 (Tabela 2). O *framework Socket.io,* no entanto utiliza outro formato, ele se vale de métodos. O primeiro é o "on", utilizando para escutar um evento, o segundo é o "emit", utilizado para enviar uma mensagem ou disparar um evento. Esses métodos são precedidos de uma instância da própria biblioteca (variável "io", da linha 8, Figura 22, por

exemplo) ou de um *socket* especifico que deseja se comunicação (linhas 24 e 25, Figura 26, por exemplo). Para escutar/disparar um evento especifico deve se utilizar um dos métodos citados seguido pelo nome do evento, na Figura 26 o evento é identificado pelo nome “novoUsuarioLogado”. Após o nome do evento usa-se uma função para receber um dado e assim processa-lo. Quando o objetivo é enviar um dado, este é anexado sem a necessidade de uma função.

Vale ressaltar que o *WebSocket* permite o trafego de dados binários, tornando possível o envio não somente de mensagens de texto, mas também de dados mais complexos como objetos e listas de objetos. Outro destaque está nos eventos nativos do *Socket.io*, por padrão o *framework* possui o evento “*connection*”, disparado quando um novo *socket* é conectado ao servidor, e o evento *“disconection”* disparado quando um *socket* se desconecta do servidor.

4.3. Front-end

Esta seção tem como objetivo complementar a seção 4.1 e detalhar os artefatos gerados no que tange o *front-end*, ou seja, a parte da aplicação existente no lado do cliente. Também aborda o acesso aos serviços providos pelo *back-end*, o funcionamento da lógica de negócio que trata os dados do usuário e controla as telas, como cada tela é associada a um *controller* e como ocorre o processamento do código *Python* no módulo de responder desafio.

4.3.1. Controllers e lógica de negócio

Na seção 4.2.2 foi apresentado o conceito de *controllers* e de lógica de negócio voltado ao *back-end* e a presente seção também se propõe a explanar estes conceitos, porém, voltado a utilização destes recursos no *front-end*. Os *controllers* no lado do cliente desempenham o papel de administrar as telas que o usuário acessa bem como a informação que é gerada nessa área. Algumas ações como comunicação com os serviços providos pelo *back-end* e configuração dos *frameworks* utilizados nas *Views* também são de responsabilidade dessa parte da aplicação. No total há 13 *controllers* que desempenha as mais diversas funções, estes são apresentados pelos tópicos a seguir.

- **UsuarioCtrl** – Gerencia principalmente as telas de perfil do usuário e as informações que são apresentadas nelas;

- **MenuCtrl** – Gerencia o acesso a aplicação sempre verificando se o usuário está realmente logado, além de gerenciar as informações do usuário logado que são apresentadas no componente menu;
- **DesafioCtrl** – Gerencia a tela de “desafios” fornecendo os serviços que obtém a lista dos desafios disponíveis (com o status de “publicado”) e que registram a comprado de um desafio. Esse *controller* também é o responsável por notificar sobre a criação de um novo desafio e de atualizar a lista de desafios quando um desafio mudar de status (de “publicado” para “não publicado”, por exemplo).
- **ModalController** – Gerencia as informações apresentada nos *modais* da tela de desafios;
- **GerenciarMeusDesafiosCtrl** – Gerencia a tela de desafios criados por um usuário;
- **LoginCtrl** – Gerencia a tela de login, além de se comunicar com o serviço de autenticação de usuário no *back-end*;
- **CadastroUsuarioCtrl** – Gerencia a tela de cadastro de usuário, sendo o responsável por enviar as informações do usuário para serem cadastradas no banco de dados;
- **ResponderDesafioCtrl** – Gerencia a tela de responder desafio, sendo o responsável por carregar as informações de um desafio após ele ser comprado. Ele também é o responsável por processar o código do usuário e assim autenticar uma resposta dada pelo mesmo, além de registrar essa resposta no banco de dados;
- **CriarDesafioCtrl** – Gerencia a tela de criação de desafio, sendo o responsável por enviar as informações de um desafio para serem registradas no banco de dados;
- **EditarDesafioCtrl** – Gerencia a tela de editar um desafio, sendo o responsável por atualizar as informações de um desafio no banco de dados;
- **AutorCtrl** – Gerencia a tela que mostra ao usuário logado os detalhes de um autor, ou seja, de um outro usuário já cadastrou algum desafio;
- **DesafiosCompradosCtrl** – Gerencia a tela com a lista e dos desafios comprados e a tela que apresenta os detalhes destes desafios;

- **TreinoCtrl** – Gerencia a tela que permite o usuário treinar, ou seja, responder um desafio que este já adquiriu sem a necessidade de compra-lo novamente, e depois, caso queira, salvar a resposta desenvolvida naquele treino.

Nota-se a partir da apresentação dos *controllers* da aplicação, que estes na sua maioria são criados para o gerenciamento das telas que o usuário tem acesso. Isso pode ser visto como um mecanismo do *AngularJs* para facilitar o gerenciamento da informação gerada, delegando informações mais especificas de uma entidade (ou serviço) a um determinado *controller.*

Figura 27 - Código do *controller* MenuCtrl.

```
app.controller("MenuCtrl", function($localStorage, $rootScope, $scope, $http) {

    $scope.logout = function() {
        $http.get(host + "logoff").success(function(data) {
            $localStorage.usuarioLogado = null;
            $localStorage.idUsuarioLogado = null;
            localStorage.setItem('logged', false);
            socket.emit('logout');
            location.assign("index.html#/login");
        });
    }

    var verificaUsuarioLogado = function() {
        $http.get(host + "get-usuario-logado").success(function(data) {
            if (data != "") {
                localStorage.setItem('logged', true);
                $rootScope.usuarioLogado = data;
            } else {
                location.assign("index.html#/login");
            }
        }).error(function(erro) {
            location.assign("index.html#/login");
        });
    }

    verificaUsuarioLogado();
});
```

A Figura 27 ilustra o corpo do *controller* “MenuCtrl”, este é apresentando com o intuito de fortalecer o conhecimento acerta do formato utilizado e do conteúdo contido em um *controller* que gerencia a lógica de negócio no *front-end*. Este é um *controller* que contém apenas as funções “logout” e “verificaUsuarioLogado”, como foi dito um dos papeis desse *controller* é de gerenciar o acesso a aplicação, sempre

verificando se um usuário está realmente logado. Isso é feito através da função “verificaUsuarioLogado” que é chamada a cada alteração de tela que tenha o componente menu, essa função utiliza a diretiva do *AngularJs* “$http” para acessar o serviço que informa se há um usuário registrado em uma seção (essa informação é armazenada em uma variável como a da linha 217, da Figura 20, seção 4.2.2), caso exista uma seção ativa, os dados do usuário logado são armazenados na variável “usuarioLogado” (linha 17 da Figura 27) que por meio da diretiva “rootScope” permite a apresentação dos dados desse usuário fora do *scope* do MenuCtrl, ou seja , em qualquer tela que o usuário esteja acessando. Se não houver uma seção ativa o usuário e redirecionado a tela de login.

A função “logout” tem o papel de invalidar a seção do usuário logado e redireciona-lo para a tela de login. Nota-se que diferente da função “verificaUsuarioLogado”, está possui a diretiva “$scope” antes do seu nome, essa diretiva está presente em diversos *controllers* e é utilizada para permitir que informações/funções seja apresentadas/acessas em uma *View*. Nota-se também que as diretivas utilizadas ($localStorage, $rootScope, $scope, $http), são declaradas após o nome do *controller* como um parâmetro da função, isso é conhecido como Injeção de Dependência, e é a forma utilizada pelo *AngularJs* para utilizar recursos que são de módulos externos. No que tange o acesso aos serviços do API, eles são feitos utilizando a diretiva “$http” e essa por sua vez possui os métodos necessários para troca de informação entre front-end e back-end via HTTP e seus respectivos verbos.

4.3.2. Views

As *Views* são as responsáveis por apresentar as informações contidas nos *controllers* para o usuário, são as telas de fato. É importante destacar que aplicação utiliza o *framework AngularJs* e a arquitetura MVC (*model view controller*), isso implica diretamente em como as telas são criadas e organizadas. Diante da arquitetura utilizada e do modo de trabalho do *AngularJs*, é utilizado o padrão SPA (*single page application*). Nesse padrão existe somente um arquivo HTML no qual são definidas as bibliotecas e os *controllers* da aplicação, as demais páginas (as *Views*) são renderizadas sobre esse arquivo HTML. A relação entre um componente HTML de um *View* e um *controller* é definida com a diretiva “ng-controller” declarada diretamente no componente (como acontece com o componente menu), porém, a

relação entre *Views* (ou todo uma página) e *Controller*s é feita pelo arquivo de Rotas, apresentado na próxima seção.

4.3.3. Rotas

Para que uma determinada *View* seja apresentada é necessário que essa seja associada a um caminho, uma Rota. O conceito de Rotas no *front-end* é similar ao *back-end*, no entanto, as Rotas definidas no *front-end* retornam a uma página HTML e não a um serviço. As rotas na aplicação se encontram no documento “/app/app.js” e são apresentadas pela Figura 28.

Figura 28 - Arquivo de Rotas do *front-end*.

```
app.config(function ($routeProvider) {
    $routeProvider.when("/perfil", {
        templateUrl: "views/usuario/perfil.html",
        controller: "UsuarioCtrl",
    });
    $routeProvider.when("/editar-perfil", {
        templateUrl: "views/usuario/editarPerfil.html",
        controller: "UsuarioCtrl",
    });
    $routeProvider.when("/desafios", {
        templateUrl: "views/desafio/desafios.html",
        controller: "DesafioCtrl",
    });
    $routeProvider.when("/novo-desafio", {
        templateUrl: "views/desafio/cadastrarDesafio.html",
        controller: "CriarDesafioCtrl"
    });
    $routeProvider.when("/editar-desafio/:id", {
        templateUrl: "views/desafio/editarDesafio.html",
        controller: "EditarDesafioCtrl",
        resolve: {
            desafio: function ($rootScope, $route) {
                return $rootScope.idDesafio = $route.current.params.id;
            }
        }
    });
    $routeProvider.when("/login", {
        templateUrl: "views/login.html",
        controller: "LoginCtrl"
    });
    $routeProvider.when("/cadastro", {
        templateUrl: "views/usuario/cadastroUsuario.html",
        controller: "CadastroUsuarioCtrl"
    });
    $routeProvider.when("/gerenciar-meus-desafios", {
        templateUrl: "views/usuario/gerenciarMeusDesafios.html",
        controller: "GerenciarMeusDesafiosCtrl"
    });
    $routeProvider.when("/responder-desafio/:id", {
        templateUrl: "views/desafio/responderDesafio.html",
        controller: "ResponderDesafioCtrl",
        resolve: {
            desafio: function ($rootScope, $route) {
                return $rootScope.idDesafio = $route.current.params.id;
            }
        }
    });
    $routeProvider.when("/perfil-autor/:id", {
        templateUrl: "views/usuario/perfilAutor.html",
        controller: "AutorCtrl",
        resolve: {
            desafio: function ($rootScope, $route) {
                return $rootScope.idUsuario = $route.current.params.id;
            }
        }
    });
    $routeProvider.when("/desafios-comprados", {
        templateUrl: "views/usuario/desafiosComprados.html",
        controller: "DesafiosCompradosCtrl",
    });
    $routeProvider.when("/detalhes-desafio-comprado/:id", {
        templateUrl: "views/usuario/detalhesDesafioComprado.html",
        controller: "DesafiosCompradosCtrl",
        resolve: {
            desafio: function ($rootScope, $route) {
                return $rootScope.idDesafio = $route.current.params.id;
            }
        }
    });
    $routeProvider.when("/treino-com-o-desafio/:id", {
        templateUrl: "views/usuario/treino.html",
        controller: "TreinoCtrl",
        resolve: {
            desafio: function ($rootScope, $route, $localStorage) {
                return $rootScope.idDesafio = $route.current.params.id;
            }
        }
    });
    $routeProvider.otherwise({ redirectTo: "/login" });
});
```

As rotas são definidas como uma configuração da aplicação, logo estão contidas dentro do método “config” (linha 4, Figura 28), as mesmas são possíveis graças a diretiva “$routeProvider” fornecida pela biblioteca “angular-route”. Sua construção é relativamente simples, sendo composta na sua maioria por quatro atributos. O primeiro atributo (“when”) é o nome da rota, o segundo atributo

(templateUrl) refere-se ao arquivo HTML que se deseja apresentar, nesse caso deve-se informar qual o caminho até o arquivo, o terceiro parâmetro é o nome do *controller* responsável por gerencia a *View* a qual a rota está ligada, e o último parâmetro (“resolve”) é usado para executar uma ação no momento em que uma rota for acessada. Na aplicação o parâmetro “resolve” é utilizado para capturar o id de identificação de uma entidade em especifico, este é passado como parâmetro nas rotas e utilizado nos *controllers* para requisitar os dados daquela entidade ao *back-end.*

Vale ressaltar que apenas os dois primeiros atributos são obrigatórios, uma vez que uma *View* não precisa de um *Controller* para existir e nem executar uma ação quando acessada. Outro destaque presente no arquivo de rotas é o atributo “otherwise” (linha 82, Figura 28), ele permite a definição de uma rota padrão caso o usuário tente acessa uma rota não existe, nesse caso a rota padrão é a de “/login”.

4.3.4. Execução do código Python

A execução do código *Python* é um dos principais recursos do CodeLive e um requisito importantíssimo para o presente trabalho, este é feito pelo *framework Skulpt* e realizado no *controller* “ResponderDesfioCtrl”.

Figura 29 - Função de processamento do código Python.

```
225     var executaCodigoModificado = function (entrada) {
226         mypre.innerHTML = '';
227         Sk.pre = "output";
228         Sk.configure({ output: outf, read: builtinRead });
229         (Sk.TurtleGraphics || (Sk.TurtleGraphics = {})).target = 'mycanvas';
230         var myPromise = Sk.misceval.asyncToPromise(function () {
231             return Sk.importMainWithBody("", false, entrada, true);
232         });
233         myPromise.then(function (mod) {
234             console.log("sucesso!");
235         }, function (err) {
236             console.log(err.toString());
237         });
238     }
```

A Figura 29 apresenta o código utilizado para realizar o processamento do código *Python*. O código consiste primeiramente em mapear a entrada do código, nesse caso o mesmo é passado por parâmetro (parâmetro “entrada”, linha 255). A saída, ou seja, o local da *View* no qual o resultado será apresentado também é mapeado, essa referência é armazenada na variável “Sk.pre” (linha 227). Essas informações são então passadas ao método “Sk.configure” (linha 228) e utilizadas no

processamento do código, sendo que este é feito pelo método “Sk.importMainWithBody” (linha 231) em uma execução assíncrona do método “Sk.misceval.asyncToPromise”.

Vale destacar que a definição da configuração é realizada no momento em que a função “executarCodigoModificado” (linha 225) é chamada, porém o processamento do código só ocorre de fato quando a variável “myPromise” (linha 233) é chamada com o método “then”, essa variável contém a instância da função “Sk.misceval.asyncToPromise” e utiliza o recurso *promise* do *javascript* para retornar o resultado o processamento.

Após processar o código do usuário o resultado será emitido no console do navegador e também em um elemento *canvas* presente na *View*. É importante deixar claro que o código apresentado pela Figura 29 é utilizado pelo método que valida a resposta do usuário, e que há dois modos de execução do código *Python*, ambos os métodos são melhor explanados na seção 4.4.7.

O processo de validação da resposta do usuário ocorre em três partes. A primeira delas consistem em gerar um novo código a partir do código inicialmente informado pelo usuário, para gerar o novo código é necessário substituir a função *Python* “`input()`” pelas entradas pré-cadastradas do desafio. Na segunda parte do processo o novo código é executado e o resultado do processamento é exibido em console contido na *View*. O último passo consiste em analisar o resultado do processamento feito no segundo passo, e comparar o resultado gerado com as saídas pré-cadastras do desafio. Se houver algum erro de sintaxe ou a saída produzida não for a esperada (igual as saídas pré-cadastras), a resposta é dada como invalida, caso contrário, é dada como correta e o usuário é então recompensado.

4.4. Conhecendo o CodeLive

O CodeLive é composto por diversos módulos com as mais diversas funcionalidades. Esses módulos são: o módulo de **login**, o módulo de **cadastro de usuário**, o módulo de **perfil**, o módulo de **visualização** e **criação de desafio,** o módulo de **responder desafios** e o módulo dos **desafios comprados**. Esta seção tem como objetivo apresentar os módulos citados, explanando de forma detalhada seu funcionamento, tecnologias envolvidas e processo de desenvolvimento.

Também, é apresentada uma visão geral da aplicação, funcionamento do jogo e os objetivos.

4.4.1. Visão geral

O CodeLive utiliza os recursos da gamificação, isso implica em um ambiente de aprendizado diferente do tradicional. Para um melhor entendimento sobre o funcionamento da ferramenta é importante conhecer qual o funcionamento geral da mesma, as regras utilizadas, o que o usuário consegue ou não fazer e quais os objetivos a serem atingidos para que o usuário avance no jogo. Os tópicos a seguir apresentam a visão geral da ferramenta, sendo importante ressaltar que as regras descritas foram definidas por Cardoso (2015) e podem ser encontradas com mais detalhes no trabalho produzido por ele.

Público e acesso ao CodeLive: o CodeLive é uma ferramenta de utilização livre voltada para o ensino e pratica de programação, logo, seu público alvo são estudantes e professores. Porém, ela não se limita somente a essas pessoas, também sendo destinada a qualquer um que se interesse por aprender e praticar programação. Para ter acesso a ferramenta basta acessa-la e criar uma conta.

Funcionamento do jogo: o jogo consiste em resolver desafios, esses são exercícios de programação constituídos de um título, enunciado entre outros dados. A resolução dos desafios é necessária para que o jogar avance no jogo, assim, esse avanço é medido com três dados principais: DF (domínio da força), Classe (podem ser: Novato, Padawan, Cavaleiro Jedi, Mestre Jedi e Mestre do Conselho), Nível (nível em que o jogador se encontra dentro de uma classe, “Mestre Jedi nível 3”, por exemplo). Essas informações são vistas pelos outros usuários e mostram o quão avançado um usuário está no jogo. Para que um jogador evolua, ou seja, aumente sua Classe e Nível, é necessário vencer os desafios e com isso ganhar pontos de DF.

Além dos pontos de DF, ao vencer um desafio, o usuário também ganha créditos e eventualmente pode vir a ganhar Recompensas. Os créditos são utilizados para comprar os desafios, já as Recompensas são prêmios que o jogador ganhar por realizar alguma ação, como por exemplo, chegar ao nível dois da classe Padawan.

Algumas regras: para o bom funcionamento da ferramenta existem algumas regras, uma delas é ao criar o desafio. Somente alguns usuários podem criar os desafios, isso é determinado de acordo com o Nível de Dificuldade do desafio e a Classe e Nível em que o usuário se encontra. Isso é feito para limitar a criação dos desafios aos usuários com mais experiência, resultado em desafios mais bem elaborados. Para comprar desafios também existem regras, não é permitido ao um usuário comprar desafios de sua própria autoria, e um usuário somente pode comprar desafios que ainda não foram vencidos por ele, isso induz o usuário a buscar novos desafios, e com isso, ganhar mais conhecimento.

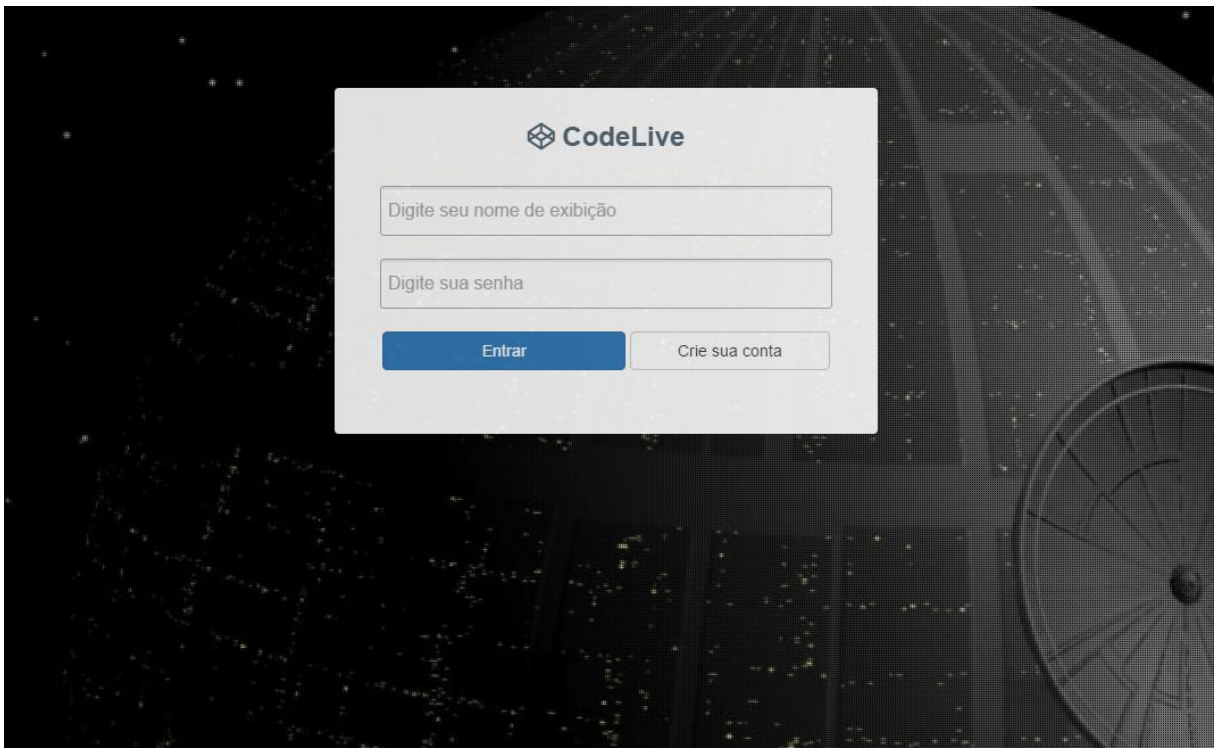
Acompanhamento dos dados: por ser uma aplicação gamificada, saber a colocação do usuário bem como o seu desempenho no jogo é muito importante. Para isso, existe o módulo de **Perfil do usuário** (seção 4.3.4), neste módulo o usuário encontra o Ranking dos melhores jogadores, o quantitativo de desafios ganhos e perdidos, o número de desafios ganhos por nível de dificuldade, as recompensas existentes e as que já foram ganhas, entre outras informações. Ainda neste módulo, o usuário dispõe da lista dos desafios criados por ele e uma tela com os dados cadastrais, sendo possível alterar qualquer um dos dados como: nome, nome de exibição, e-mail, senha e foto do perfil quando o usuário achar conveniente.

As subseções a seguir detalham os módulos e principais funcionalidades do CodeLive, além disso, é apresentada a interação entre o usuário e a ferramenta e entre os próprios módulos.

4.4.2. Módulo de login

O módulo de login é a primeira tela que o usuário tem contato ao acessar o sistema. Sua principal função é fazer a autenticação do usuário, solicitando as informações “Nome de exibição” e “Senha”, como ilustra a Figura 30.

Figura 30 - Tela de Login do sistema.

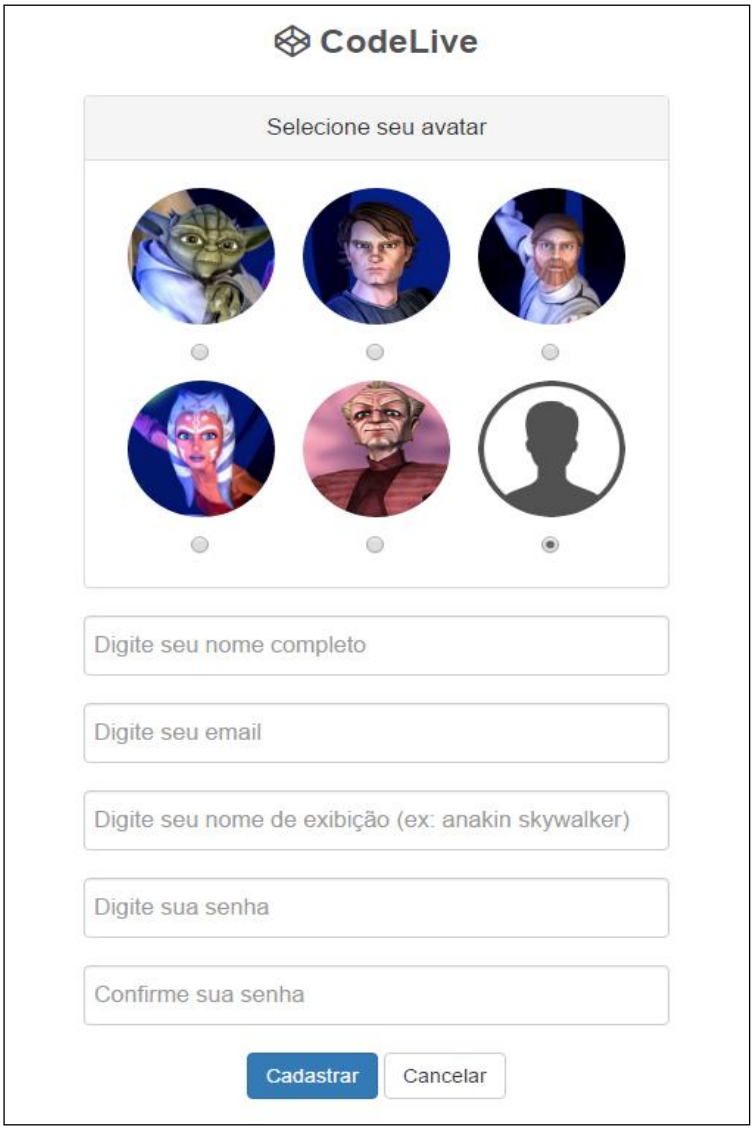


Após preencher o formulário com os dados requeridos, o usuário seleciona a opção “Entrar” para submeter as informações e aguarda o resultado. Se os dados estiverem corretos o usuário e redirecionado para o módulo de **Perfil**, caso contrário, uma mensagem é exibida informando um erro nos dados ou até mesmo uma falha no sistema, caso haja uma falha de conexão ou problema similar.

4.4.3. Módulo de cadastro de usuário

O acesso a esse módulo se encontra na tela de Login, através da opção: “Crie sua conta” (Figura 30). A função desse módulo é fazer um registro do usuário, para que o mesmo possa acessar o sistema. Os dados exigidos para o cadastro do usuário, são: “Nome completo”, “E-mail”, “Nome de exibição”, “Senha” e uma confirmação da senha, como mostra a Figura 31.

Figura 31 - Tela de cadastro de usuário.



Além dos dados citados, o usuário pode selecionar um avatar. Os avatares são personagens da saga *Star Wars* que representam o usuário no jogo. Caso o usuário não opte por um dos personagens de *Star Wars,* o sistema adota um avatar padrão que pode ser alterado a qualquer momento no módulo de **Perfil**, junto com as demais informações do usuário. Vale ressaltar que o “Nome de Exibição” deve ser único, portanto, o sistema valida essa informação ao registrar o usuário e ao alterar seus dados.

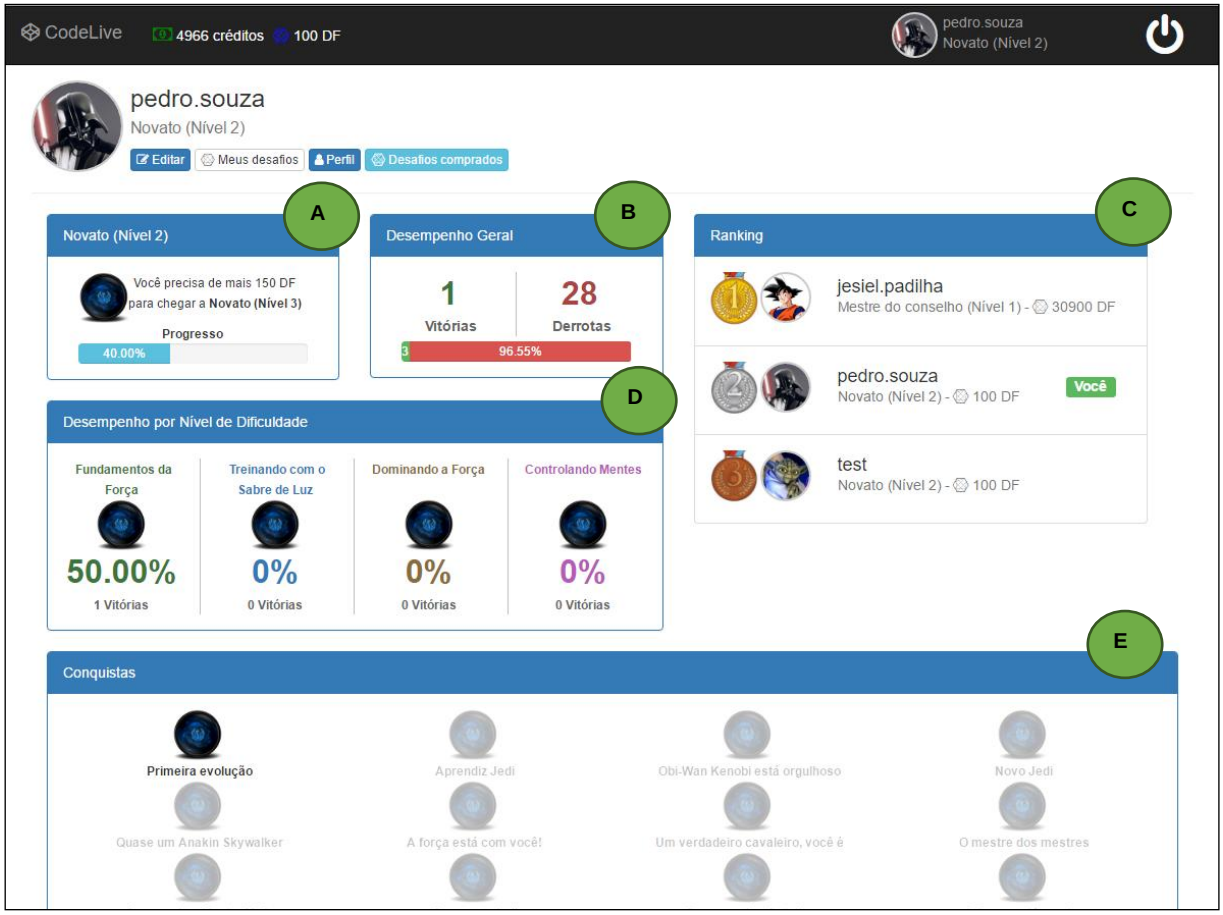
4.4.4. Módulo de perfil do usuário

O módulo de perfil do usuário é a primeira tela exibida após a autenticação do usuário, sendo necessário ao usuário criar uma conta (caso não tenha) e se

autenticar no módulo de **Login**. Esse módulo é dividido em três partes (módulos), descritas nas subseções a seguir:

Informações gerais do usuário: A primeira parte do perfil é responsável por mostrar as informações mais pertinentes ao desempenho do usuário dentro do jogo, bem com uma relação dos melhores jogadores (Figura 32).

Figura 32 - Tela de perfil com as informações gerais do usuário.



Ao analisar a Figura 32 é possível identificar diversos painéis, cada um deles exibe uma informação do usuário. Os tópicos a seguir detalham a função de cada um dos painéis organizados e identificados na Figura 22 por um círculo verde e uma letra do alfabeto:

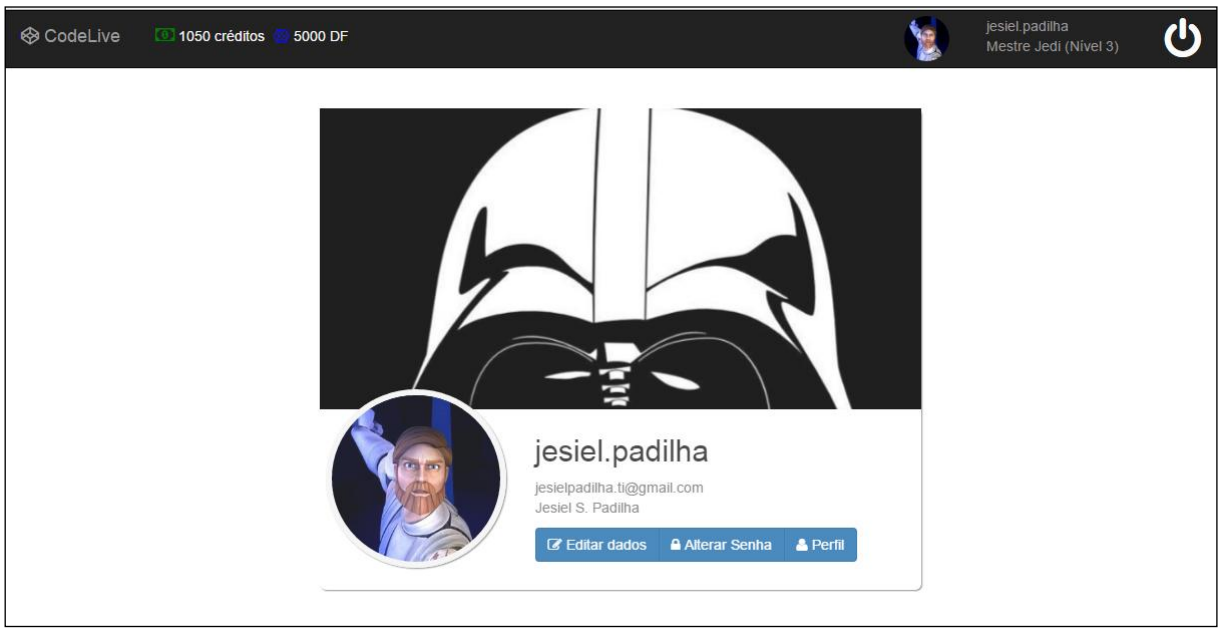
- **A** – Esse painel contém a classe e nível atual do usuário, além de informar qual a pontuação/porcentagem necessária para alcançar a próxima classe ou nível no jogo;

- **B** – Esse é o painel de **desempenho geral**, ele mostra a pontuação/porcentagem de vitórias e derrotas que o usuário teve no jogo;
- **C** – Esse é painel de **Ranking**, ele contém uma lista dos cinco melhores jogadores, sendo que essa classificação é feita de acordo com o nível de DF (domínio da força) do usuário;
- **D** – Esse é o painel de **Desempenho por nível de dificuldade**, ele exibe a quantidade de vitórias que um usuário teve em cada nível de dificuldade, ele também exibe a porcentagem dessas vitórias em comparação com a quantidade de desafios criados daquele nível de dificuldade;
- **E** – Esse é o painel de **Conquistas**, ele mostra as diversas conquistas que o usuário pode ganhar no decorrer do jogo, sendo que as conquistas que o usuário já obteve são destacadas, possuindo uma cor mais nítida.

As opções de gerenciamento para as demais partes do módulo de **Perfil** também se encontrar nesta tela, através dos botões: Editar, Meus desafios e Desafios Comprados.

Edição das informações do usuário: a edição dos dados do usuário é a segunda parte do módulo de perfil, sendo criada para que o usuário possa alterar os dados informados no momento do cadastro.

Figura 33 - Tela para editar os dados do usuário.



Como mostra a Figura 33, essa parte do módulo de perfil dá acesso as principais informações do usuário e permite que o mesmo as altere. Para tal, o usuário pode selecionar as seguintes opções: Editar dados – Mostra um formulário com os dados atuais para que sejam atualizadas; Alterar Senha – Mostra um formulário com os campos: “nova senha” e “confirmar nova senha”; Perfil – Redireciona o usuário a primeira parte do módulo (subseção **Informações gerais do usuário**). Um quarto recurso é acionado ao clicar na imagem do avatar. Ao fazer essa ação uma “janela” é exibida, esta possui um campo que permite selecionar uma foto do computador do usuário para ser utilizar como avatar.

Gerenciar meus desafios: Esse módulo foi criado para permitir que o usuário gerencie os desafios criados por ele. Nele há uma lista dos desafios criados pelo usuário, sendo possível realizar uma busca para localizar um desafio específico, como mostra a Figura 34.

Figura 34 - Tela de gerenciamento dos desafios do usuário logado.

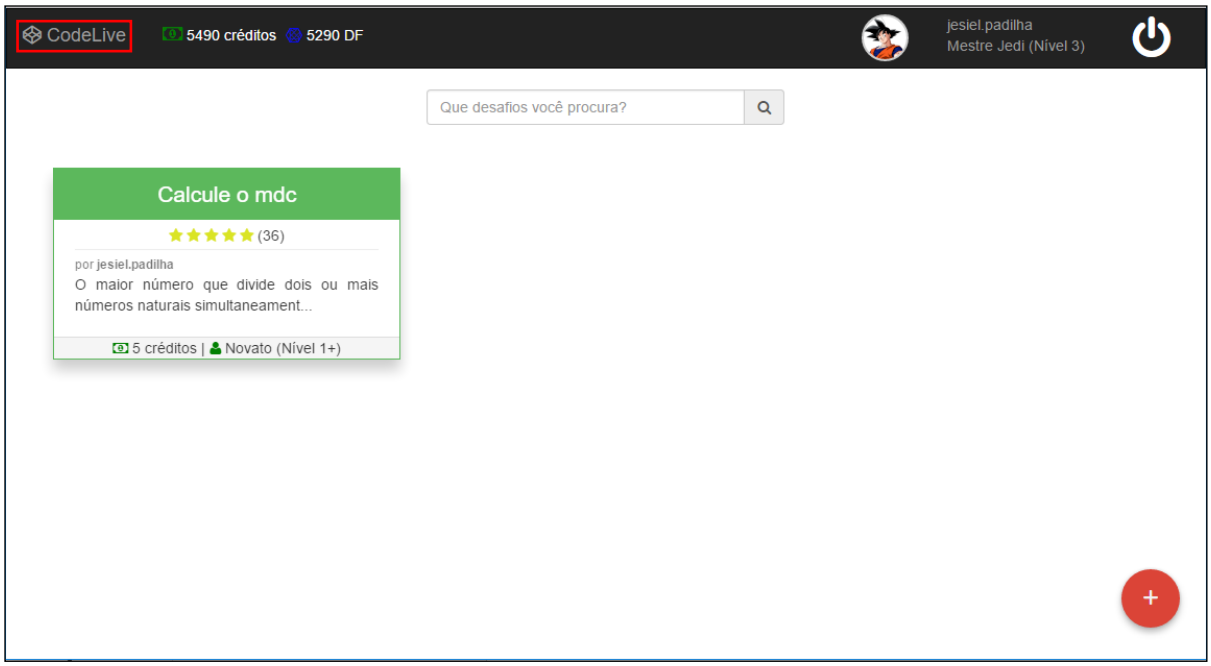
Vale ressaltar que os desafios criados não podem ser excluídos, uma vez que isso causa impacto no histórico gerado com a interação entre os usuários e os desafios. Sendo assim, as opções disponíveis são: alterar o status do desafio para “publicado” ou “não publicado”, editar os dados de um desafio (válido somente para os desafios não publicados) e ver os detalhes de um desafio.

4.4.5. Módulo de visualização e compra de desafios

O módulo de visualização e compra de desafio lista todos os desafios criados e publicados. Um desafio é exibido no formato de um painel que possui diferentes cores, sendo que cada cor representa o grau de dificuldade para resolução daquele

desafio. O painel também contém outras informações como: título, usuário que criou o desafio, valor do desafio etc.

Figura 35 - Tela de desafios.



O acesso a essa área é feito clicando no nome “CodeLive” (quadro em vermelho na Figura 35). Um menu lateral será exibido, então, basta que o usuário selecione a opção “Desafios” e o mesmo é redirecionado para tela da Figura 35. Após acessar esse módulo há uma segunda funcionalidade, a de comprar um desafio. Para tal, basta o usuário clicar em um desafio e será exibida uma janela, como detalhada pela Figura 36.

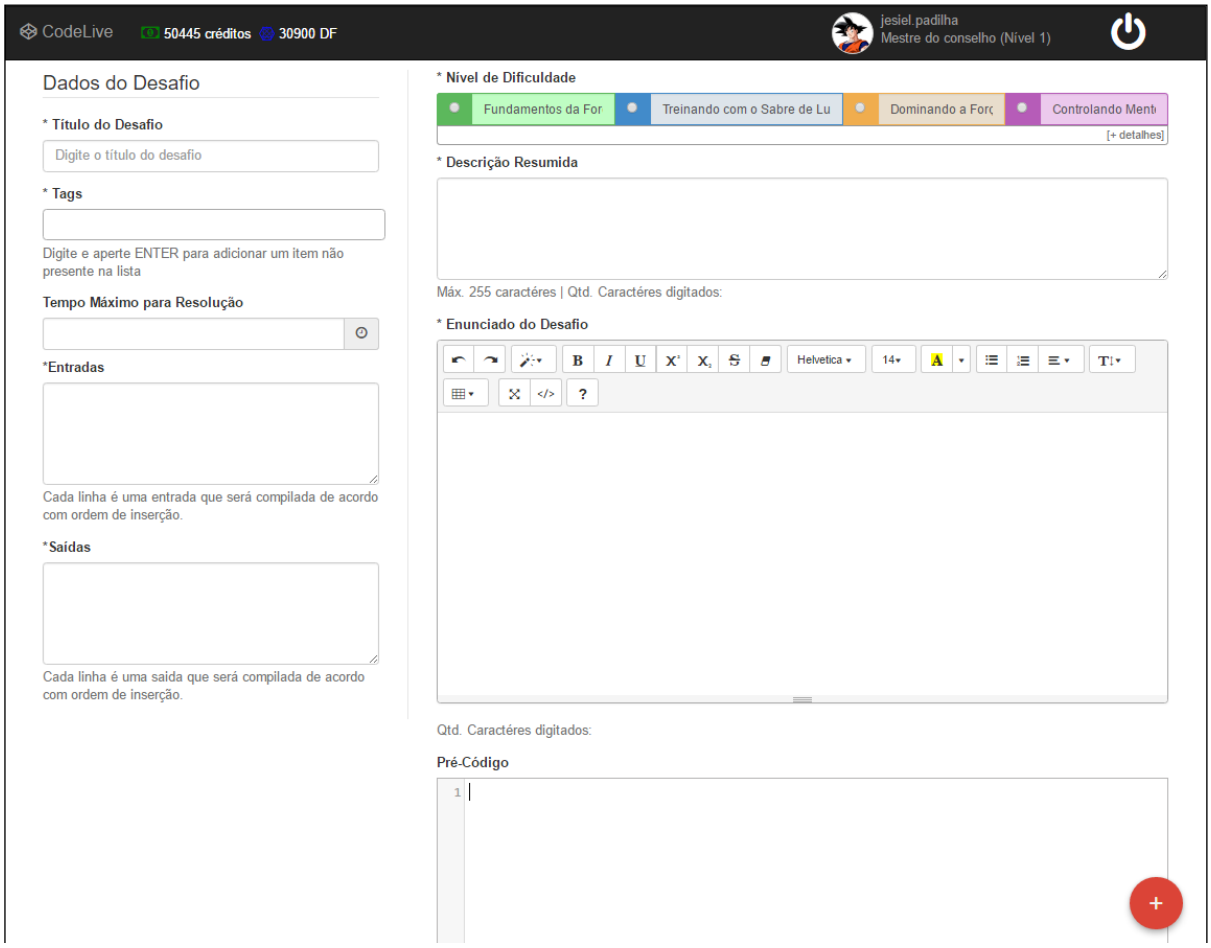
Figura 36 - Tela para comprar um desafio.

A janela apresentada pela Figura 36 exibe mais detalhes sobre o desafio, os mais importantes são o enunciado do problema a ser resolvido e as recompensas que o usuário ganhará caso vença o desafio. Para comprar o desafio exibido na janela da Figura 26, basta que o usuário selecione a opção “Comprar Desafio” e após isso confirmar a operação. Serão debitados os créditos referentes ao preço do desafio e o usuário será redirecionando para o módulo de **responder desafios.**

4.4.6. Módulo de cadastro de desafio

Para que haja desafios disponíveis no módulo anterior (seção 4.2.4) é necessário que alguém os cadastre. Para isso existe o módulo de castro de desafio. Ele é composto por um formulário com diversas opções, cujas principais e obrigatórias são: Título do desafio, Tags, Entradas, Saídas, Nível de dificuldade, Descrição resumida e Enunciado do desafio. Essas opções possuem o símbolo: *, como mostrado na Figura 37.

Figura 37 - Tela de cadastro de desafio.



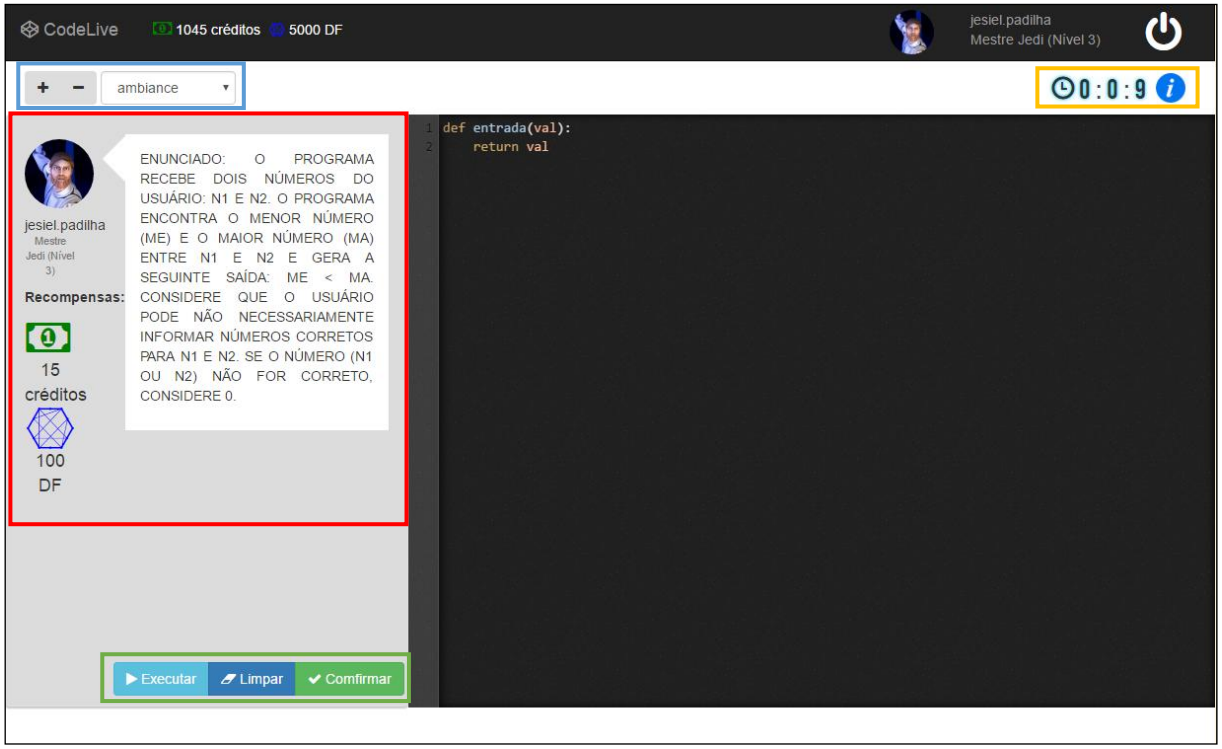
Se alguma das opções citadas não for devidamente preenchida, uma notificação e uma indicação do campo requerido são exibidos. As demais informações do formulário servem para auxiliar o usuário no momento de responder o desafio, por essa razão são opcionais. Após preencher o formulário, basta que o usuário coloque o cursor do mouse sobre o botão vermelho (canto inferior direto da imagem) e selecione a opção “Cadastrar”, para concluir a operação.

É importante ressaltar que esse módulo é de grande valia para o funcionamento da aplicação, não só por fornecer os desafios, mas também pelo modo como estes são criados. Para que seja possível validar um desafio as entradas e saídas são de suma importância, estas devem ser inseridas na ordem da execução em que serão executadas pelo *Skulpt* no módulo de **responder desafio**, esse cuidado deve ser tomado para que seja possível validar um desafio e assim atribuir ou não pontuação ao usuário.

4.4.7. Módulo de responder desafios

O módulo de responder desafios é um dos módulos mais importantes do sistema, pois é a área disponível para responder os desafios adquiridos pelos usuários, e assim permiti que mesmo avance no jogo (subir de classe/nível) quando responde um desafio corretamente. A Figura 38 ilustra os elementos que compõem esse módulo.

Figura 38 - Tela para responder os desafios.



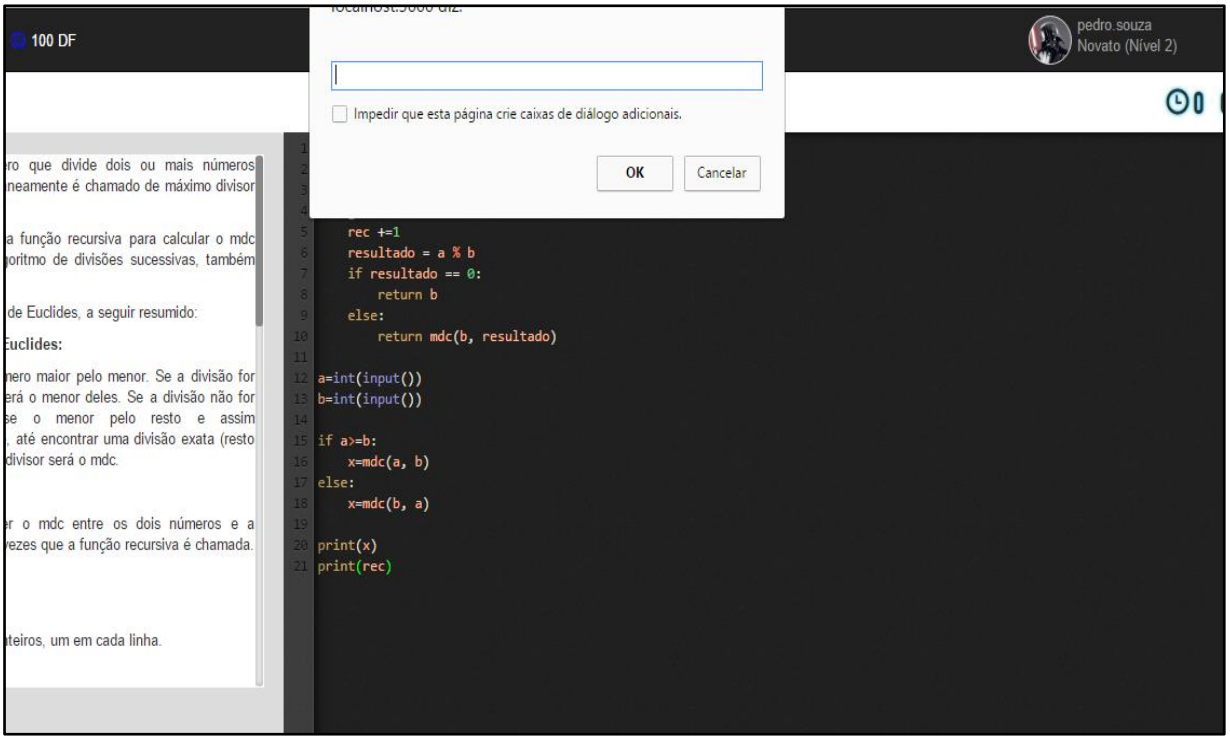
Analisando a Figura 38 nota-se que o principal elemento (elemento que ocupa a maior parte da tela) é o editor de texto. Essa área é destinada ao código do usuário e utiliza o framework *CodeMirror* para fornecer um ambiente mais amigável. Vale ressaltar que a ferramenta *Skulpt*, que interpreta o código *Python* do usuário, atua justamente nesse módulo, processando o código inserido no editor de texto.

Os demais elementos deste módulo são as informações do desafio (quadro vermelho), os botões que permitem ao usuário: executar o código, limpar o código do editor e confirma a resposta do desafio (quadro verde). Além do relógio que marca o tempo restante para responder o desafio e um botão de “ajuda” que informa ao usuário o modo correto de responder o desafio (quadro amarelo). Por último, há os controles do editor (quadro azul), eles permitem maximizar/minimizar o editor e alterar o tema do mesmo.

Quanto a execução do código inserido pelo o usuário, é importante deixar claro que pode ocorrer de duas maneiras: com interação do usuário e sem interação com o usuário.

A execução com interação do usuário trata-se de executar o código que usuário inseriu e permitir que o usuário interaja com ele, como exemplificado pela Figura 39.

Figura 39 - Execução do código do usuário.



A Figura 39 ilustra a execução do código do usuário no módulo de **responder desafio**. Essa é a primeira forma de executar o código, nota-se que uma caixa de diálogo *JavaScript* contendo um campo para inserção de dados é exibida, assim o usuário pode interagir com a mesma e, ao fim da execução, o resultado é exibido abaixo do editor. Vale ressaltar que a caixa de diálogo será exibida sempre que houver a função *Python* “`input()`”, permitindo que o usuário insira dados nela.

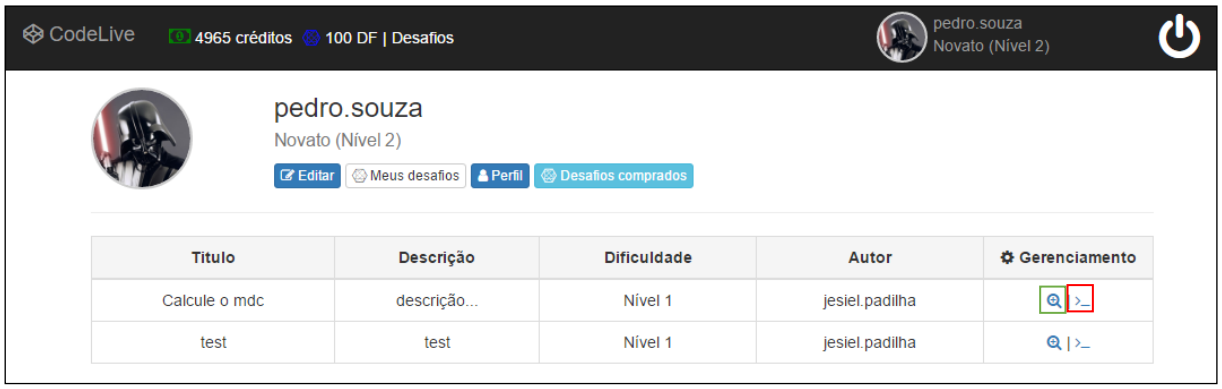
Outro ponto da interação nesse tipo de execução, é o *feedback* que o usuário tem ao executar o código, caso ocorra algum erro este é apresentado pelo console a baixo do editor de texto na cor velha, informando por exemplo, que uma variável não existe. Com isso o usuário pode retornar ao editor de texto e realizar a correção necessária.

A segunda forma de executar o código, não permite intervenção/interação do usuário e ocorre ao selecionar a opção “Confirmar” (botão verde da Figura 38), ela é realizada internamente no *controller* “ResponderDesafioCtrl” e consiste em substituir a função *Python* “`input()`” (linhas 12 e 13 da Figura 29) pelas entradas pré-cadastradas do desafio, gerando um novo código, em seguida, este código é executado e a saída gerada é comparada com as saídas pré-cadastradas do desafio, é neste momento que ocorre a validação do código do usuário. Após a validação do código, ocorre um registro da resposta no banco de dados e o usuário é redirecionado para a página de desafios, sendo notificado sobre o resultado.

4.4.8. Módulo dos desafios comprados

Este módulo foi criado para que o usuário pudesse ver os desafios dos quais ele já participou e, com isso, ter a possibilidade de melhorar suas habilidades, já que este módulo oferece uma área de treino com os desafios adquiridos. Ele é composto por três partes, a primeira delas é a de exibição dos desafios, esta exibe uma lista de todos os desafios comprados, como mostra a Figura 40.

Figura 40 - Lista dos desafios comprados.



O acesso às demais áreas do **módulo dos desafios comprados** se dá a partir da tela mostrada pela Figura 40. Tal acesso se encontra na aba “Gerenciamento” da tabela que contém os desafios. Para cada desafio da tabela há a opção “Detalhes do desafio” (quadro verde da Figura 40) e “Treinar” (quadro vermelho da Figura 40). A opção de “Detalhes do desafio” exibe os detalhes de cadastramento do desafio, além de outras informações como mostra as Figuras 41 e 42.

Figura 41 - Área de detalhes de um desafio comprado.

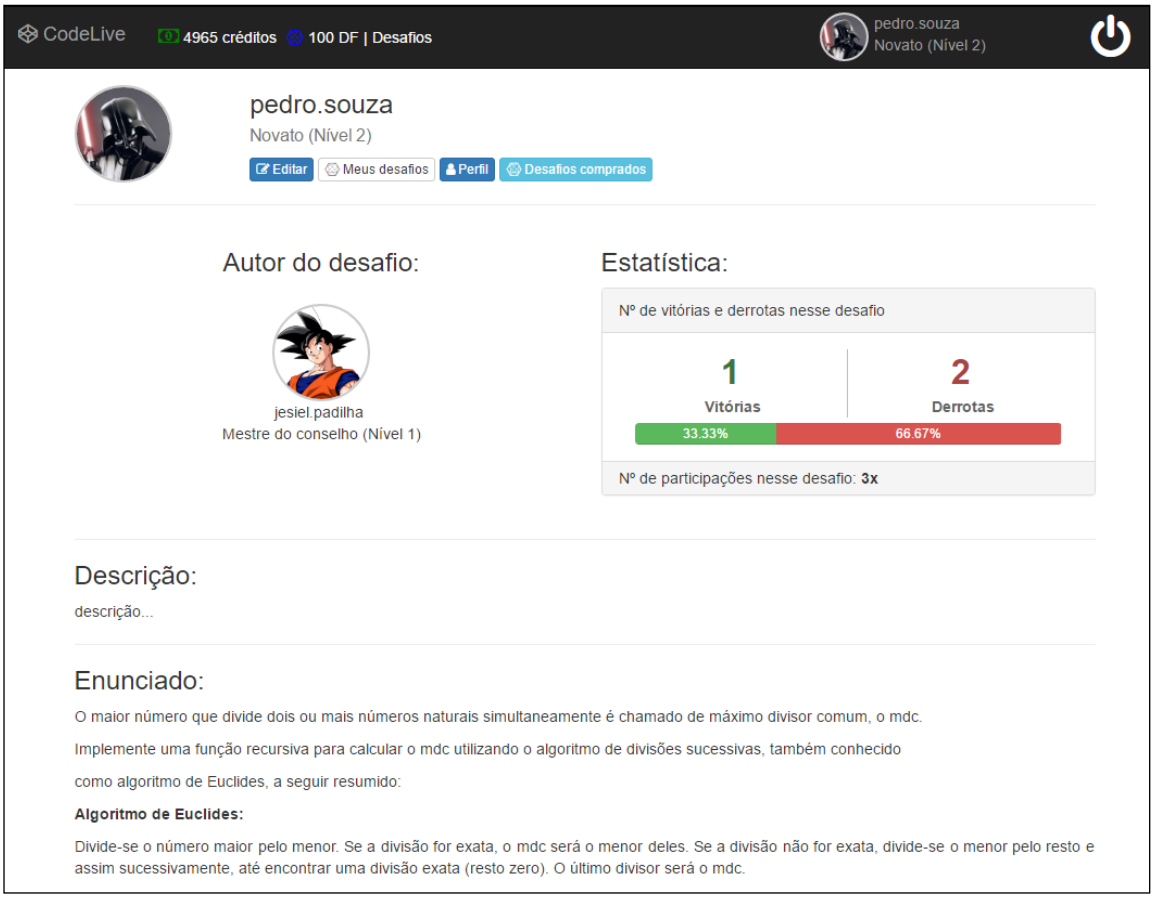
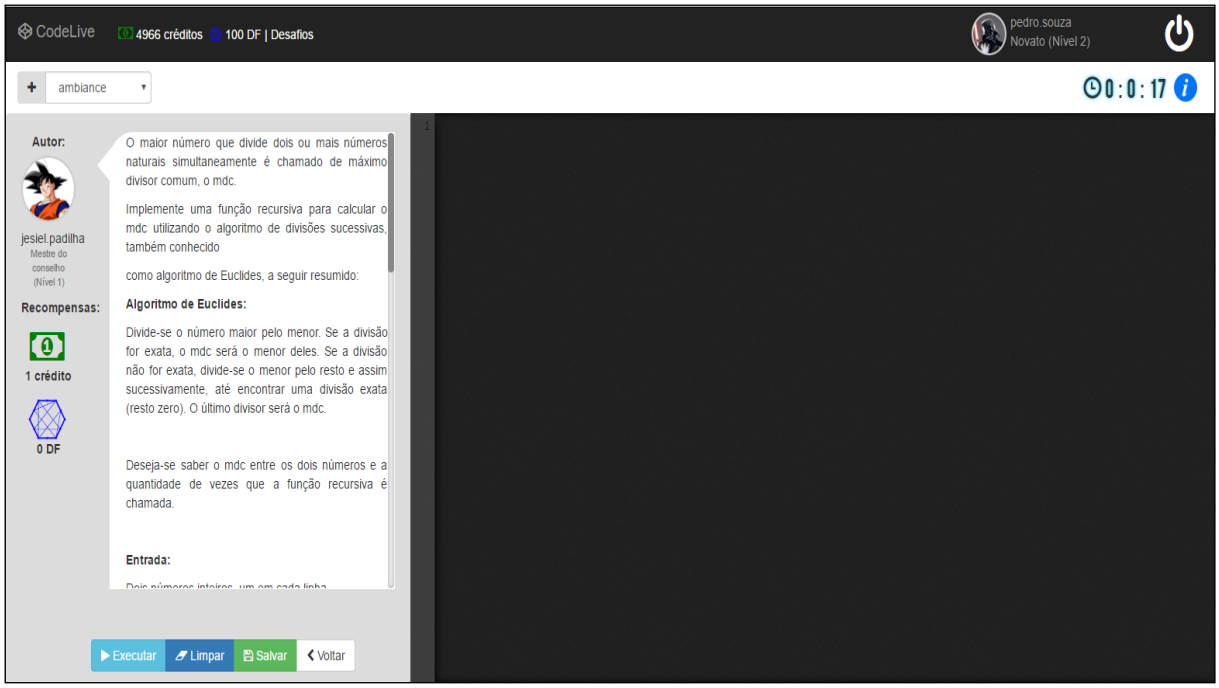


Figura 42 - Área de detalhes de um desafio comprado.

Analisando a Figura 41, nota-se que são exibidas as principais informações de um desafio, como por exemplo: Autor do desafio (com o avatar, classe e nível em que o mesmo se encontra), Descrição do desafio, Enunciado, Tempo para responder o desafio e o Pré-código, caso exista. Outra informação de grande relevância são as Estatísticas, ou seja, o desempenho do usuário no desafio, com esses dados é possível saber o número de vezes que o usuário participou daquele desafio além da quantidade de vitórias e derrotas.

A Figura 42 é uma continuação da Figura 41 e contém uma lista das respostas feitas pelo usuário. Essas respostas são divididas em tópicos, o primeiro contém as respostas produzidas pelo o usuário quando participou do desafio, nessa parte, além de ter o código, o usuário também tem a data de participação do desafio e o resultado, ou seja, se a resposta está correta ou não. O segundo tópico contém as respostas produzidas pelo usuário nos treinos, essa contém a data do treino e o código salvo pelo usuário. A última área do módulo dos desafios comprados é a área de treino, mostrado na Figura 43.

Figura 43 - Área de treino dos desafios comprados.



A área ilustrada na Figura 43 é bastante similar a área ilustrada na Figura 38 (módulo de **responder o desafio**). Tal similaridade foi proposta com o intuito de ter um ambiente que se assemelhasse ao ambiente tradicional de responder um desafio. As diferenças principais são o Tempo, que no “modo treino” é sempre ilimitado, e a Recompensa que o usuário recebe, que sempre será de 1 crédito para cada participação neste modo. É importante destacar que o código produzido pelo usuário pode ou não ser salvo. Para salvar um código, o usuário deve selecionar a opção “Salvar” (botão verde da Figura 43), caso o usuário queira somente praticar pode selecionar a opção “Voltar” (botão branco da Figura 43) e o código produzido não será salvo.

5 CONSIDERAÇÕES FINAIS

Com o propósito de motivar o usuário a aprender e praticar programação nasceu o CodeLive, este visa alcançar seus objetivos com a utilização dos elementos da gamificação e do universo da saga *Star Wars*, proporcionando um ambiente divertido e ao mesmo tempo estimulante para realizar as atividades propostas. O presente trabalho teve a tarefa de complementar os recursos da ferramenta, afim torna-la mais eficiente, no que tange a execução do código do usuário, e mais interativa, fornecendo informações em tempo-real para o usuário, sem que houvesse a necessidade do mesmo realizar alguma ação.

Diante do objetivo proposto, foram utilizadas as ferramentas *Socket.io, Skulpt, Node.js, AngularJs* e *CodeMirror.* Com os recursos disponibilizados por essas ferramentas foi possível utilizar a linguagem de programação *JavaScript* não só no *front-end,* mas também no *back-end* da aplicação, construir as funções de notificação e atualização em tempo-real e executar o código *Python* inserido pelo usuário. Elas também contribuíram para a criação da interface do usuário e dos serviços presentes na aplicação.

Dentre as diversas contribuições feitas pelo presente trabalho estão as mensagens de notificação quando um usuário entra na aplicação, quando um novo desafio é criado, quando um usuário responde um desafio que outro usuário também respondeu, sendo que o autor do desafio também é notificado e, quando um desafio tem seu status alterado, indo de “não publicado” para “publicado”, por exemplo. Todas essas mensagens são feitas em tempo-real, não importando o módulo, ou seja, a parte da aplicação que o usuário esteja acessando.

Vale ressaltar que não há apenas mensagens em tempo-real, mas também atualizações na interface, como no Ranking dos melhores jogadores (atualizado quando um usuário sobe no Ranking, por exemplo) e na página dos desafios, acrescentando ou removendo um desafio de acordo as ações feitas no sistema. Outro importante recursos provido pelo presente trabalho foi o de execução e processamento do código *Python* inserido pelo usuário. Este recurso permite que o usuário possa interagir com o código criado por ele, sendo possível, por exemplo, digitar valores e acompanhar o resultado do processamento do mesmo. Além disso, o mecanismo criado possibilita ao próprio CodeLive validar uma resposta produzida

por um usuário, fazendo com que este seja devidamente recompensado e avance no jogo.

As demais contribuições foram relacionadas a criação e atualização de alguns módulos. Um dos módulos criados foi o de “desafios comprados”, que permite ao usuário ver os desafios adquiridos por ele, além de possibilitar que este treine utilizando um desses desafios e como isso aumente sua experiência para responder desafios futuros. Já no módulo de perfil houve uma atualização, e agora há painéis que fornecem dados sobre o desempenho do usuário no jogo, informando por exemplo, quantas vitórias e quantas derrotas o usuário teve.

Diante dos diversos recursos acrescentados pelo presente trabalho, alguns pontos não foram explorados, porém eles podem ser de grande valia para a aplicação. Alguns dos pontos que podem ser levados em consideração em trabalhos futuros, são os de interação com os demais usuários. Logo, o primeiro passo seria implementar um mecanismo de chat, este possibilitaria aos usuários trocarem informações entre si e assim alcançar níveis mais elevados em menor tempo. Outro recurso bastante interessante é o de transferir créditos, que possibilitaria a transferência de crédito de um usuário para outro. Esta seria uma maneira de ajudar usuários inexperientes a comprarem desafios.

No que tange os desafios há duas alterações que podem ser muito relevantes, a primeira é a de “avaliação de um desafio”. Este recurso permitiria que os usuários classificassem os desafios através de uma nota e um comentário, sendo também uma maneira de ajudar os demais usuários a comprarem desafios mais compatíveis com o seu nível de conhecimento. O segundo recurso, descrito também por Cardoso (2015), é o de duelo entre os usuários: nesse módulo dois usuários iriam competir entre si, sendo que o mais rápido a responder um desafio de forma correta seria dado como vencedor.

6 REFERÊNCIAS BIBLIOGRÁFICAS

ALVARENGA, Sean Carlisto de. **Tecnologias Push e uma arquitetura de notificação para cidades inteligentes.** 2013. 62 f. TCC (Graduação) - Curso de Ciência da Computação, Universidade Estadual de Londrina, Londrina, 2013. Disponível em: <http://www.uel.br/cce/dc/wp-content/uploads/TCC-SeanAlvarenga-BCC-UEL-2013.pdf>. Acesso em: 19 jan. 2016.

CARDOSO, Djonathas Carneiro. **CODE LIVE:** Gamificação de um ambiente virtual de programação**.** 2015. 85 f. TCC (Graduação) - Curso de Sistemas de Informação, Centro Universitário Luterano de Palmas, Palmas, 2015.

MULLER, Gabriel L. **HTML5 WebSocket protocol and its application to distributed computing.** 2013. 63 f. Dissertação (Mestrado) - Curso de Ciência da Computação, School Of Engineering, Computational Software Techniques In Engineering, Cranfield University, Swindon, 2014. Disponível em: <http://arxiv.org/pdf/1409.3367v1.pd>. Acesso em: 15 jan. 201

WANG, Vanessa; SALIM, Frank; MOSKOBITS, Peer. **The Definitive Guide to HTML5 WebSocket:** BUILD REAL-TIME APPLICATION WITH HTML5. New York: Apress, 2013. 208 p.

LUBBERS, Peter; GRECO, Frank. **HTML5 WebSocket:** A Quantum Leap in Scalability for the Web. Disponível em: <http://www.websocket.org/quantum.html>. Acesso em: 04 abr. 2016.

RIBEIRO, Nilson José; MENDES, Luís Augusto Mattos. **VoIP – Tecnologia de Voz sobre IP.** 2005. 42 f. TCC (Graduação) - Curso de Ciência da Computação, Departamento de Ciência da Computação, Universidade Presidente Antônio Carlos - Unipac, Barbacena, 2005. Disponível em: <http://ftp.unipac.br/site/bb/tcc/tcc-6930aa4e21db90a985797092d43a775c.pdf>. Acesso em: 28 mar. 2016.

TANENBAUM, Andrew S.; WETHERALL, David J. **Redes de Computadores.** 5. ed. São Paulo: Pearson Education - Br, 2011. 586 p.

ROSS, Keith W.; KUROSE, James F. **Redes de Computadores e a Internet:** Uma Abordagem Top-Down. 6. ed. São Paulo: Pearson Education - Br, 2013. 634 p.

POSTEL, Jon. **USER DATAGRAM PROTOCOL.** 1980. Disponível em: <https://tools.ietf.org/html/rfc768>. Acesso em: 04 abr. 2016

MORIMOTO, Carlos E.. **UDP.** 2005. Disponível em: <http://www.hardware.com.br/termos/udp>. Acesso em: 16 maio 2016.

IDOL, Robert David. **Performance Differences between HTTP Long Polling and Websockets.** 2013. 20 f. Dissertação (Mestrado) - Curso de Ciência da Computação, University Of North Carolina At Chapel Hill, Chapel Hill, 2013. Disponível em: <http://cs.unc.edu/~mxrider/papers/idol-looking-beyond-http.pdf>. Acesso em: 15 jan. 2016.

POSTEL, Jon. **TRANSMISSION CONTROL PROTOCOL.** 1981. Disponível em: <https://www.ietf.org/rfc/rfc793.txt>. Acesso em: 09 abr. 2016.

BERNERS-LEE, Tim; FIELDING, Roy; NIELSEN, Henrik Frystyk. **Hypertext Transfer Protocol -- HTTP/1.0.** 1996. Disponível em: <https://tools.ietf.org/html/rfc1945>. Acesso em: 01 jun. 2016.

FAIN, Yakov; RASPUTNIS, Victor; TARTAKOVSKY, Anatole; GAMOV Viktor. **Enterprise Web Development:** Building HTML5 Applications: From Desktop to Mobile. Sebastopol: O'reilly Media, 2014. 642p.

WEBSOCKET.ORG. **About HTML5 WebSocket.** Disponível em: <http://www.websocket.org/aboutwebsocket.html>. Acesso em: 15 jan. 2016.

GARBO, Jhonatan Wilson Aparecido; DIAS, Jaime Willian. **Integração de sistemas utilizando Web Services do tipo REST.** 2015. 5 f. TCC (Graduação) - Curso de Ciência da Computação, Unipar, Paranavaí, 2015.

FETTE, Ian; MELNIKOV, Alexey. **RFC 6455:** The WebSocket Protocol. 2011. Disponível em: <https://tools.ietf.org/html/rfc6455>. Acesso em: 13 jan. 2016.

TUTORIALS POINT. **HTML5 - WebSockets.** 2016. Disponível em: <http://www.tutorialspoint.com/html5/html5_websocket.htm>. Acesso em: 14 jun. 2016.

UBL, Malte; KITAMURA, Eiji. **Introducing WebSockets:** Bringing Sockets to the Web. 2010. Disponível em: <http://www.html5rocks.com/en/tutorials/websockets/basics/>. Acesso em: 14 jun. 2016.

MOREIRA, Rafael Henrique. **O que é Node.js?** 2013. Disponível em: <http://nodebr.com/o-que-e-node-js/>. Acesso em: 12 jul. 2016.

ZIM, Joe. **Conectando no Socket.IO:** o básico. 2013. Disponível em: <http://imasters.com.br/tecnologia/redes-e-servidores/conectando-no-socket-io-o-basico/?trace=1519021197&source=single>. Acesso em: 12 jul. 2016.

TABLELESS. Comunidade Brasileira de Desenvolvimento Web. **O que é client-side e server-side?:** Diferenças entre linguagem client-side e linguagem server-side. 2016. Disponível em: <http://tableless.github.io/iniciantes/manual/obasico/o-que-front-back.html>. Acesso em: 25 set. 2016.

CODEMIRROR. **CodeMirror.** Disponível em: <http://codemirror.net/>. Acesso em: 28 set. 2016.

GRAHAM, Scott. **Skulpt:** Python. Client side. Disponível em: <http://www.skulpt.org/>. Acesso em: 23 nov. 2016.

NODEJS. **About node.js.** Disponível em: <https://nodejs.org/en/about/>. Acesso em: 23 nov. 2016.