FACULDADE DE TECNOLOGIA DE SÃO JOSÉ DOS CAMPOS
FATEC PROFESSOR JESSEN VIDAL

GEOVANNY DE AVELAR CARNEIRO

SOCIALCRON: UMA PLATAFORMA PARA AUTOMAÇÃO DE POSTAGENS EM SITES DE MÍDIA SOCIAL

São José dos Campos
2017

GEOVANNY DE AVELAR CARNEIRO

SOCIALCRON: UMA PLATAFORMA PARA AUTOMAÇÃO DE POSTAGENS EM SITES DE MÍDIA SOCIAL

Trabalho de Graduação apresentado à Faculdade de Tecnologia São José dos Campos, como parte dos requisitos necessários para a obtenção do título de Tecnólogo em Análise e desenvolvimento de sistemas.

Orientador: Me. Antônio Egydio São Tiago Graça

São José dos Campos
2017 (ano)

Dados Internacionais de Catalogação-na-Publicação (CIP)
Divisão de Informação e Documentação

AVELAR, Geovanny
SocialCRON: Uma plataforma para automação de postagens em sites de mídia social.
São José dos Campos, 2017.
52f. (número total de folhas do TG)

Trabalho de Graduação – Curso de Tecnologia em Análise e desenvolvimento de sistemas
FATEC de São José dos Campos: Professor Jessen Vidal, 2017.
Orientador: Me. Antônio Egydio São Tiago Graça.

1. Áreas de conhecimento. I. Faculdade de Tecnologia. FATEC de São José dos Campos: Professor Jessen Vidal. Divisão de Informação e Documentação. II. Título

REFERÊNCIA BIBLIOGRÁFICA –

AVELAR, Geovanny. **SocialCRON: Uma plataforma para automação de postagens em sites de mídia social.** 2017. 52f. Trabalho de Graduação - FATEC de São José dos Campos: Professor Jessen Vidal.

CESSÃO DE DIREITOS –

NOME DO AUTOR: Geovanny de Avelar Carneiro
TÍTULO DO TRABALHO: SocialCRON: Uma plataforma para automação de postagens em sites de mídia social
TIPO DO TRABALHO/ANO: Trabalho de Graduação / 2017.

_____
Geovanny de Avelar Carneiro
52.132.571-7

GEOVANNY DE AVELAR CARNEIRO

SOCIALCRON: UMA PLATAFORMA PARA AUTOMAÇÃO DE POSTAGENS EM SITES DE MÍDIA SOCIAL

Trabalho de Graduação apresentado à Faculdade de Tecnologia São José dos Campos, como parte dos requisitos necessários para a obtenção do título de Tecnólogo em Análise e desenvolvimento de sistemas.

Composição da Banca

_____

Nome do Componente da Banca, titulação e Instituição

_____

Nome do Componente da Banca, titulação e Instituição

_____

Nome do Orientador, titulação e Instituição

_____/_____/_____

DATA DA APROVAÇÃO

Em memória de Sérgio Carneiro
23/12/1953 - 18/11/2014

AGRADECIMENTOS

“O verdadeiro progresso é pôr a tecnologia ao alcance de todos”

Henry Ford

RESUMO

Palavras-Chave: mídia social, rede social, automação de postagens, web, java

ABSTRACT

Keywords: social media, social network, post schedule, web, java

LISTA DE ABREVIATURAS E SIGLAS

ORM	*Object relational mapping*
RDBMS	*Relational Database Management System*
HTTP	*Hypertext transfer protocol*
REST	*Representational state transfer*
JVM	Java *virtual machine*
JDK	Java *development kit*
XML	*Extensible markup language*
DI	*Dependency injection*
AOP	*Aspect-oriented programming*
MVC	*Model-view-controller*
API	*Application programming interface*
URI	*Uniform resource identifier*
JSON	*Javascript object notation*
HTML	*Hypertext markup language*
CSS	*Cascading style sheets*
GUI	*Graphical user interface*
SPA	*Single page application*
AJAX	*Asynchronous Javascript and XML*
NoSQL	*Not only SQL*
SQL	*Structured Query Language*

SUMÁRIO

1- INTRODUÇÃO

Tendo seu uso deflagrado a partir do início da década passada, os serviços de mídias sociais são onipresentes na vida de qualquer indivíduo envolvido com tecnologia. A definição de mídia social parece bastante mal compreendida, porém pode ser entendida como um conjunto de aplicações onde o usuário interage e atua na geração de conteúdo, baseadas no conceito de *Web* 2.0 e de *User Generated Content* (UGC). É um consenso que sites como a Wikipédia, Youtube e o Facebook se enquadrem dentro desse escopo, tais ideias de interação do usuário têm sido tão aplicadas a ponto de se tornarem ubíquas, de forma que se torna estranho utilizar um site e se deparar apenas com páginas estáticas e sem algum aspecto social atrelado. (KAPLAN, [2010]).

O Facebook, hoje maior plataforma de mídia social do mundo, tinha 175 milhões de usuários ativos em 2009 (Kaplan, 2010), quase o dobro da população da Alemanha na mesma época. Tal situação foi notada dentro do ambiente corporativo, motivado pela gigantesca quantidade de informações que podem ser obtidas nesse meio. Um cliente descontente poderia se queixar em um post do Facebook antes mesmo de recorrer a um canal oficial da empresa, essa informação poderia ser útil tanto na resolução do problema quanto para ser aproveitada pela área de marketing, que teria uma maneira de entender qual imagem sua marca possui diante dos consumidores. A despeito desse exemplo, qualquer departamento dentro da organização pode se beneficiar do uso de mídias sociais, situação essa que obriga todos essas áreas a possuírem conhecimento a seu respeito. (MONTEIRO, [2012]).

A vantagem do relacionamento com o cliente pelas mídias sociais é a de que são uma via de mão dupla, uma vez que as informações são coletadas, medidas podem ser tomadas para facilitar a relação. Ao estar frente a uma reclamação, a empresa poderia prestar um serviço de SAC por meio da própria ferramenta, colher feedbacks e principalmente aumentar sua percepção junto ao mercado. (MONTEIRO, [2012]).

Esse estudo foi conduzido com o objetivo de desenvolver um protótipo de uma plataforma que automatize postagens em sites de mídia social, chamada SocialCRON. Foi um trabalho executado a pedido da Agência Code Plus, uma empresa de desenvolvimento de software récem-formada. Esse documento

descreve uma implementação dessa plataforma, que é um projeto de código aberto, licenciado sob a MIT *license*.

1.1 Objetivo geral

A intenção é dar início à construção de uma aplicação de fácil uso e voltada a produtores de conteúdo que necessitem da postagem automática em sites de mídia social.

1.2 Objetivos específicos

Esses foram os objetivos específicos definidos para o projeto:

- Desenvolver um protótipo de uma aplicação que torne automática a postagem nos horários inseridos pelo usuário, inicialmente apenas no Facebook;
- Desenhar uma interface de usuário aplicando conceitos de UX;
- Fornecer um meio para que produtores de conteúdo possam administrar suas páginas/perfis em apenas uma ferramenta;
- Utilizar durante o desenvolvimento ferramentas de código aberto na medida do possível;
- Gerar uma aplicação de código aberto, sob os termos da licença MIT.

2- CONTEXTUALIZAÇÃO TECNOLÓGICA

Esse capítulo descreve as principais tecnologias utilizadas no desenvolvimento, bem como uma comparação com soluções semelhantes e os requisitos funcionais e não funcionais da plataforma.

2.1 Tecnologias utilizadas

A tabela um contém uma relação de todas as principais tecnologias utilizadas no desenvolvimento da plataforma.

TABELA 1 – Tecnologias utilizadas

NOME	VERSÃO	LICENÇA
Java	7	BCL
Maven	3.3.9	Apache
Spring Boot	1.4.1	Apache
Hibernate	5.0.1	LGPL
Python	2.7	PSFL
MySQL	5.6	GPL
Redis	3.2.8	BSD
AngularJS	1.6.0	MIT
JQuery	3.1.0	Apache
Materialize	0.97.8	MIT
Bower	1.8.0	MIT
Gulp	3.9.1	MIT
Git	2.1.4	GPL
OAuth	2.0	-
Flask	0.12.2	BSD

Fonte: AUTOR (2017).

2.1.1 Java

Java é uma linguagem de programação de propósito geral, orientada a objetos e baseada em C++ criada em 1991 por James Gosling, da Sun Microsystems. A linguagem foi formalmente anunciada em 1995, chamando a atenção da comunidade interessada no desenvolvimento de aplicações corporativas e mais tarde na programação para dispositivos móveis. (DEITEL, [2009])

2.1.2 Maven

Ferramenta de gerenciamento de projetos na linguagem Java baseada no conceito de *project model object* (POM), um arquivo XML contendo informações do projeto, como dados para o *build*. O Maven permite o gerenciamento do *build* e das dependências. (MAVEN, [2016])

2.1.3 Spring Boot

Spring é um *framework* para desenvolvimento de aplicações corporativas em Java. O Spring Boot é uma versão do mesmo *framework*, mas com configuração muito mais simplificada. Provê funcionalidades como DI, AOP e programação *web* no padrão MVC. (WALLS, [2015])

2.1.4 Hibernate

Ferramenta para ORM, sendo uma implementação do padrão JPA. Em resumo, é responsável por transformar objetos Java em tabelas de um banco de dados relacional e vice e versa. (HIBERNATE, [2016])

2.1.5 Python

Python é uma linguagem de programação interpretada, de alto nível e multiparadigma. Foi criada em 1990, por Guido Von Rossum, do Instituto Nacional de Pesquisa para Matemática e Ciência da Computação da Holanda. Seu foco era

em usuários que não eram programadores, e sim físicos e engenheiros. Por esse motivo, possui uma sintaxe clara. (BORGES, [2010])

2.1.6 MySQL

MySQL é um sistema de gerenciamento de banco de dados relacional de código aberto, mantido hoje pela Oracle. É um RDBMS escalável, rápido e de fácil uso. (MYSQL, [2016])

2.1.7 Redis

Redis é um banco de dados não relacional criado por Salvatore Sanfilippo e liberado como software livre em 2009. Sua principal característica é funcionar em memória, sem persistir os dados de maneira física. Esse detalhe o torna extremamente rápido para leitura e gravação. (LAZOTI, [2014])

2.1.8 AngularJS

É um *framework* escrito em Javascript e desenvolvido pela Google que torna simples o desenvolvimento rápido de aplicações *front-end* que utilizam *web services* REST como *back-end*. (WILLIAMSON, [2015])

2.1.9 JQuery

JQuery é uma biblioteca Javascript criada em 2006 por John Resig, com o intuito de simplificar o desenvolvimento *client-side*. Com ela é possível manipular eventos, fazer animações e requisições AJAX que funcionem em qualquer navegador. (BIBEAULT, [2015])

2.1.10 Materialize

É um *framework* de CSS para desenho de interfaces de usuário utilizando o *Material Design*, uma linguagem de *design* desenvolvida pela Google com a intenção de unificar a experiência de usuário em todos os seus produtos. Com ele é

possível desenvolver uma interface com boa usabilidade sem muito trabalho. (MATERIALIZE, [2016])

2.1.11 Bower

Bower é um gerenciador de pacotes para projetos *web*. Gerencia componentes que contém HTML, CSS e Javascript, bem como fontes e imagens. Ele garante que a versão correta de um componente esteja instalada de maneira simples por meio de um arquivo de configuração, dispensando a procura e instalação manual das dependências. (BOWER, [2017])

2.1.12 Gulp

Conjunto de ferramentas para automação de tarefas repetitivas no desenvolvimento, como mover arquivos de dependências para os diretórios corretos depois de baixados ou concatenar e minificar arquivos. (GULP, [2017])

2.1.13 Git

Git é um software que detecta e registra mudanças em um conjunto de arquivos com o decorrer do tempo, atividade conhecida como controle de versão. O controle de versão é especialmente importante em projetos de código aberto, pois é o que permite o controle das modificações feitas pelos mais diversos contribuintes. (SILVERMAN, [2013])

2.1.14 OAuth

OAuth é uma especificação aberta de autorização que se tornou padrão na indústria. Construído com o objetivo de prover um meio seguro de autorização para aplicações *web, mobile e desktop*. Ele permite que aplicações de terceiros se conectem para obter acesso limitado a um serviço HTTP. (OAUTH, [2012])

2.1.15 Flask

Um *microframework* da linguagem Python simples e extensível. (FLASK, [2017])

2.2 Conceituação

Aqui são definidos alguns conceitos importantes para o entendimento do sistema, em especial da API.

2.2.1 Application programming interface

Application programming interface (API) é uma interface que torna possível o acesso de um sistema por meio de rotinas e/ou funções sem se envolver diretamente com a implementação. O SocialCRON possui uma API que permite que qualquer cliente manipule dados da plataforma sem conhecer suas complexidades, bastando para isso, respeitar a interface.

2.2.2 Representational State Transfer

REST é um padrão arquitetural para desenvolvimento de *web services*. Foi descrito em uma tese de doutorado por Roy Fielding, em 2000. Fielding foi um dos principais autores da especificação do HTTP, motivo pelo qual o padrão é guiado pelo o que seriam boas práticas de uso desse protocolo. O REST dita princípios de uso como:

- Aplicação correta dos métodos do HTTP;
- Uso dos códigos de status para representar sucesso ou falha nas operações;
- Uso correto dos cabeçalhos.

(SAUDATE. [2014])

2.3 Soluções existentes

A tabela dois oferece uma comparação com outras soluções.

TABELA 2. Comparativo com soluções existentes

Característica	SocialCRON	Buffer	HootSuite
Código aberto	X		
Suporte a trabalho em equipes		X	X
Analytics		X	X
Aplicativo mobile		X	X
Sugestão de conteúdo			X

Fonte: AUTOR (2017).

2.3.1 Buffer

Buffer é uma ferramenta para agendamento de postagens com suporte a seis sites de mídias sociais, possuindo um foco maior no usuário corporativo. Tem funcionalidades limitadas na conta gratuita. É o maior serviço do gênero hoje, atendendo dois milhões de usuários.

2.3.2 Hootsuite

Hootsuite é um serviço de agendamento de postagens em mídias sociais com um número de funcionalidades razoáveis mesmo na versão gratuita. Seu diferencial aos concorrentes fica por conta das sugestões de postagens.

2.3.3 SocialCRON

O SocialCRON se diferencia por ser a primeira plataforma de agendamento de postagens em mídias sociais de código aberto, permitindo que o usuário interessado possa utilizar o software em sua própria infraestrutura, sem depender de terceiros, no mesmo modelo de *softwares* consagrados, como o Wordpress.

Também é possível modificar e distribuir a aplicação sem qualquer tipo de impedimento, mesmo objetivando uso comercial, pois possui licença MIT, uma das menos restritivas. O SocialCRON é trabalha com um *web service* REST, o que permite que se desenvolva *front-ends* em qualquer plataforma.

2.4 Requisitos

Aqui são explicitados os requisitos e a metodologia utilizada na coleta.

2.4.1 Stakeholders

O único envolvido na utilização do sistema é o usuário, que consome o serviço de automação de postagens, podendo cadastrar, remover, editar postagens e definir horários onde serão executadas.

2.4.2 Metodologia

Para a coleta de requisitos foram feitas reuniões junto aos membros da Agência Code Plus, que detém a autoria do conceito da plataforma SociaICRON.

2.4.3 Requisitos funcionais

As tabelas a seguir explicitam os requisitos funcionais.

2.4.3.1 Requisito 1

O requisito um fornece uma descrição do processo de autenticação da aplicação.

RF-01	
Nome	Login
Descrição	Permite que o usuário se autentique e utilize o sistema
Atores	Usuário
Prioridade	Alta
Entradas e pré-condições	Nome de usuário (ou *e-mail*), senha e conta validada.
Saídas e pós-condições	Usuário autenticado no sistema.
Fluxo de eventos	
Fluxo principal	O sistema verifica inicialmente se existe um usuário com o nome informado, caso exista, transforma a senha digitada em uma *hash* e verifica se ela é igual à cadastrada no banco de dados, é também verificado se a conta está confirmada. Satisfeitas as condições, autoriza o login.
Fluxo secundário	Caso o nome de usuário não corresponda a nenhuma conta cadastrada, a *hash* não seja igual à cadastrada no banco e/ou conta não esteja validada não autoriza o login e informa o usuário que a senha/nome de usuário estão incorretos ou que a conta não está confirmada.

2.4.3.2 Requisito 2

O requisito dois mostra o processo de cadastro das postagens.

RF-02	
Nome	Cadastro de postagem
Descrição	Cadastra uma postagem no sistema
Atores	Usuário
Prioridade	Alta
Entradas e pré-condições	Usuário autenticado, título, contéudo.
Saídas e pós-condições	Postagem cadastrada.
Fluxo de eventos	
Fluxo principal	Usuário autenticado entra com um título, conteúdo da postagem, a postagem então é cadastrada.
Fluxo secundário	Se título e conteúdo não forem válidos, o usuário é informado e operação é cancelada.

2.4.3.3 Requisito 3

O requisito três relata o cadastro de eventos.

RF-03	
Nome	Cadastro evento
Descrição	Define os horários onde uma postagem irá se repetir
Atores	Usuário
Prioridade	Alta
Entradas e pré-condições	Postagem cadastrada, horário inicial, intervalo de repetição (caso exista), horário de término e perfil onde os posts serão feitos.
Saídas e pós-condições	Horários onde os posts serão executados cadastrados no sistema.
Fluxo de eventos	
Fluxo principal	Usuário seleciona um post, define um horário de início e se o post acontecer mais de uma vez, caso positivo, define um intervalo de repetição, um horário de término e informa o perfil (ou perfis) onde a postagem será feita.
Fluxo secundário	Caso o horário de início esteja no passado, seja depois do horário de término, se o intervalo de repetição for muito curto (de 10 em 10 minutos, por exemplo) ou se não houver perfis cadastrados, informa o usuário e não conclui o cadastro do evento.

2.4.3.4 Requisito 4

O requisito quatro exibe o cadastro de perfis do Facebook.

RF-04	
Nome	Cadastro de perfil
Descrição	Autentica junto à API do Facebook e obtém um *token* de acesso para executar as postagens.
Atores	Usuário
Prioridade	Alta
Entradas e pré-condições	Conta ativa no Facebook
Saídas e pós-condições	Perfil cadastrado dentro da plataforma
Fluxo de eventos	
Fluxo principal	Usuário é redirecionado para o login no Facebook, caso o login seja bem sucedido, o perfil é cadastrado no sistema juntamente com o *token* retornado pelo site de mídia social, o que permitirá as postagens.
Fluxo secundário	Caso o login no Facebook não seja bem sucedido o perfil não é cadastrado.

2.4.3.5 Requisito 5

O requisito cinco delimita o funcionamento do *upload* de imagens.

RF-05	
Nome	Upload de imagem
Descrição	Usuário entra com uma imagem e o arquivo é transferido para o servidor.
Atores	Usuário
Prioridade	Média
Entradas e pré-condições	Postagem cadastrada
Saídas e pós-condições	Imagem salva no servidor
Fluxo de eventos	
Fluxo principal	Usuário entra com uma imagem de até 5 megabytes nos formatos JPEG, PNG ou GIF. Satisfeitas as condições, é efetuado o upload e a imagem fica atrelada ao post informado.
Fluxo secundário	Caso a imagem não possua as características necessárias, informa o usuário e não faz o upload.

2.4.4 Requisitos não-funcionais

Nessa seção são apresentados todos os requisitos não-funcionais.

2.4.4.1 Desempenho

- Qualquer tela da aplicação deve ser carregada em três segundos.
- Qualquer postagem deve ser selecionada e postada nos sites de mídia social definidos pelo usuário em no mínimo um minuto.

2.4.4.2 Disponibilidade

- A aplicação front-end e o *web service* devem ter uma disponibilidade de no mínimo 95%
- A camada de *software* responsável pela execução das postagens deve possuir 99% de disponibilidade.

2.4.4.3 Flexibilidade

- Qualquer indivíduo com conhecimentos básicos de administração de servidores Linux deve ser capaz de fazer o *deploy* da aplicação apenas lendo a documentação disponível junto ao código-fonte.
- Qualquer programador com conhecimento das ferramentas utilizadas tanto no *front-end* quanto no *back-end* deve ser capaz de realizar alterações na plataforma apenas com consulta aos repositórios de código aberto e à documentação que os acompanha.

2.4.4.4 Integridade

- O usuário deve estar logado antes de executar qualquer cadastro, edição ou exclusão de dados relacionados à posts, eventos e perfis no sistema.
- Apenas usuários do tipo ADMIN podem listar todos os agendamentos.
- Um usuário só pode visualizar os dados relativos à sua conta.
- O login só pode ser efetuado com uma conta confirmada.

2.4.4.5 Usabilidade

- Qualquer usuário final deve ser capaz de utilizar a aplicação sem treinamento algum.

2.4.4.6 Hardware e software

- Por ser desenvolvido utilizando a plataforma Java, o *web service* pode ser executado em qualquer sistema operacional que possua implementação da JVM. Recomenda-se sistemas Linux, Tomcat 7 como servidor de aplicação e JDK 7.
- O *front-end* pode rodar sobre qualquer servidor *web*, recomenda-se Apache 2.
- O *front-end* deve ser renderizado por meio dos navegadores Mozzila Firefox e Google Chrome, tanto em suas versões *desktop* quanto *mobile*.
- Os navegadores devem aceitar *cookies*.
- O serviço REST e o *front-end* da aplicação devem utilizar o protocolo HTTPS.

2.4.4.7 Suporte

- Considerando-se que todos os artefatos de software estão sob os termos da licença MIT, que prevê o uso irrestrito por parte de qualquer interessado sem nenhum tipo de garantia, não há suporte por parte do desenvolvedor.

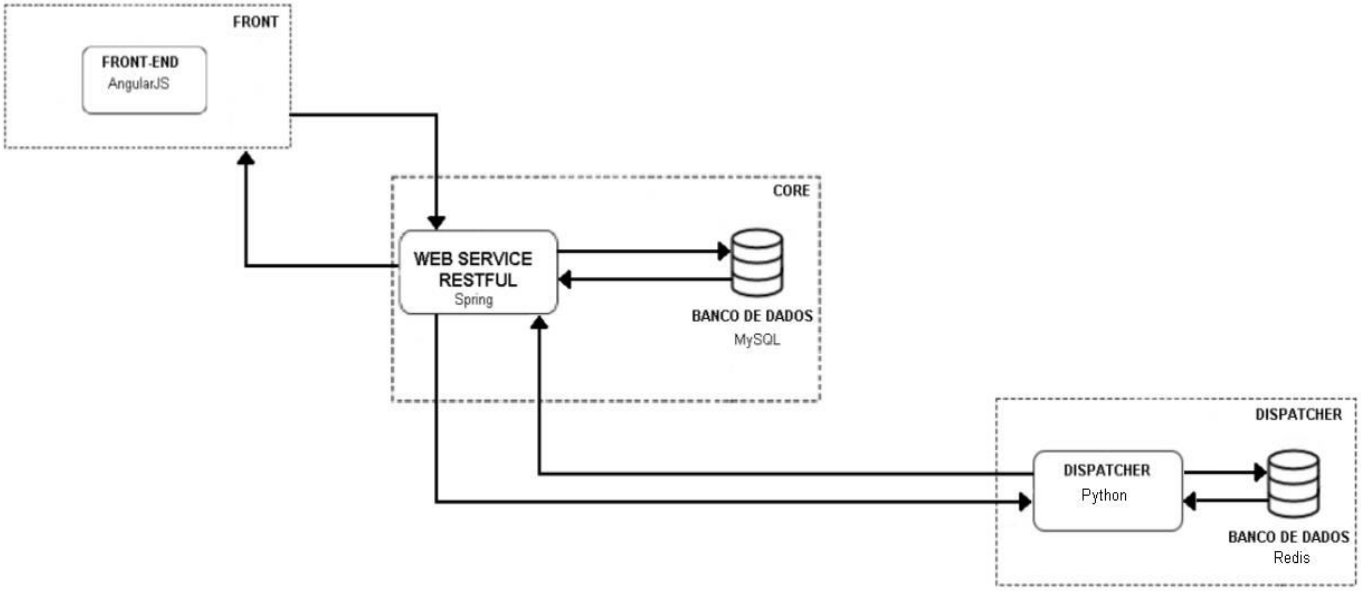
3- DESENVOLVIMENTO

Nesse capítulo são apresentadas e solucionadas questões a respeito do desenvolvimento e arquitetura da plataforma.

3.1 Arquitetura

De início, foi adotada uma arquitetura monolítica, com todas as camadas da aplicação unidas em um único artefato. Essa solução não se mostrou satisfatória conforme a complexidade aumentava, tornando dificultosa a manutenção. A estrutura foi então retrabalhada e o sistema inteiro ficou dividido em três partes: um *front-end* escrito em Javascript, uma API RESTful desenvolvida em Java e um serviço de execução das postagens, em Python. A figura um contém um diagrama descrevendo a solução.

FIGURA 1. Arquitetura da plataforma

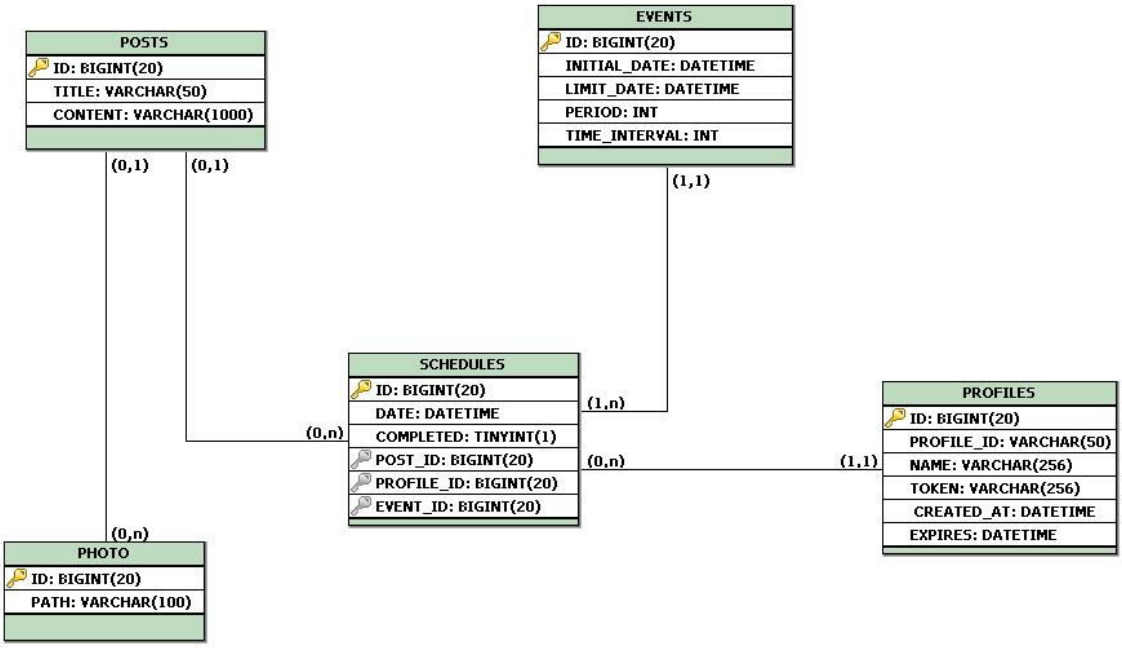


Fonte: AUTOR (2017).

3.2 Banco de dados

Na figura dois, é mostrado o diagrama de entidade-relacionamento que define o esquema de banco de dados.

FIGURA 2. Diagrama de entidade-relacionamento



Fonte: AUTOR (2017).

3.3 A API do SocialCRON

Todo o tráfego de dados é feito por meio da API. Foi implementada em Java utilizando o Spring Boot. É a única responsável por se comunicar diretamente com o banco de dados relacional MySQL. A conexão com o banco de dados e a transformação dos registros da base de dados relacional em objetos Java é feita com o Hibernate. Os objetos são serializados no formato JSON.

3.3.2 Funcionamento

Aqui são descritos detalhes a respeito do comportamento e uso da API.

3.3.2.1 Autenticação

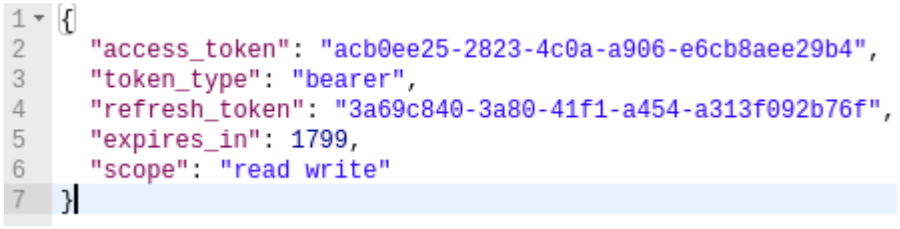
Para prover autenticação foi utilizado o Spring Security. OAuth 2.0 é o protocolo de autorização. Antes de executar qualquer ação, o cliente se autentica

por meio do par usuário/senha, recebendo como resposta um *token*. O *token* é uma longa cadeia de caracteres gerada aleatoriamente e que é armazenada no banco de dados por um período de trinta minutos, perdendo sua validade após isso. A cada chamada da API, o cliente deve passar o seu *token* por meio de um cabeçalho HTTP, do contrário, receberá um *status* 401 *UNAUTHORIZED*. Um exemplo de requisição de um *token* seria:

- POST /oauth/token?username=user&password=pass&grant_type=password HTTP/1.1

Essa requisição deve possuir um cabeçalho *Authorization* com uma *string* contendo o *id* da API e seu *secret* codificados em Base64. O que geraria algo como *Authorization:* '*Basic* c29jaWFsY3Jvbjpzb2NpYWxjcm9u'. O retorno seria um conteúdo serializado em JSON, como na figura três.

FIGURA 3. JSON retornado durante a autenticação



Fonte: AUTOR (2017).

Contém o *access token*, que deve ser passado em cada requisição. O tempo de expiração é de 1799 segundos (trinta minutos), podendo ser reiniciado por meio do *refresh token*. O escopo sempre será o mesmo, permitindo leitura e escrita.

3.3.2.2 Acesso aos recursos

Uma vez obtido o *bearer token*, o cliente pode criar, acessar, editar e excluir qualquer recurso desde que esse tenha sido criado por ele. A exceção fica por conta dos usuários do tipo ADMIN, que podem ler e efetuar qualquer modificação em qualquer recurso.

O envio do *token* é executado por meio do cabeçalho *Authentication*, que deve ser enviado em todas as chamadas. Sendo feito da seguinte maneira:

- *Authentication: Bearer* 76d54255-2fec-40a5-95fb-0b1943ca1e1b

Uma tentativa de acesso a um recurso sem esse envio ou utilizando um *token* que expirou retornaria um status 401 *UNAUTHORIZED.*

3.3.3 Elementos

Essa seção visa descrever como a API é organizada.

3.3.3.2 Resources

Tal definição dentro do padrão REST é bastante abstrata, o que torna difícil conceitua-la. Usualmente, um *resource* é qualquer coisa que possa ser armazenada em um computador e representada como *bits* (RICHARDSON, [2007]). No caso em específico, os seguintes itens dentro da API são resources:

- Um perfil de mídia social (*Profile*)
- Um rascunho de postagem (*Post*)
- Parâmetros que demarcam quando uma postagem deve acontecer (*Event*)
- Um agendamento de postagem (*Schedule*)

Todos são representados internamente como objetos Java, convertidos do banco de dados por uma ferramenta de mapeamento objeto-relacional, serializados em JSON e entregues ao cliente. As ações de criação, recuperação, edição e exclusão são executadas por meio dos *endpoints*.

3.3.3.3 Endpoints

Um *endpoint* é uma URI onde um recurso pode ser acessado ou manipulado. É a conexão onde um resource é exposto ao cliente. Alguns exemplos dentro do SocialCRON são:

- POST /posts HTTP/1.1
- GET /posts/1 HTTP/1.1
- PUT /posts/1 HTTP/1.1
- DELETE /posts/1 HTTP/1.1

As URIs não contêm verbos, cada ação é definida pelo método HTTP, de maneira que um mesmo *endpoint* deflagra diferentes comportamentos dependendo do método utilizado.

3.3.3.4 Métodos e códigos de status do HTTP

Para se adequar ao REST, é feito o uso correto dos métodos, códigos de status e cabeçalhos do protocolo HTTP. A seguir são descritas essas diretrizes e como cada uma é utilizada. Desconsideram-se detalhes relacionados à autenticação, que foram tratados na seção 3.3.2.1.

3.3.3.4.1 POST

É usado quando da criação de um recurso. Recebe um cabeçalho *Content-Type* com o valor *application/json*. O *body* contêm os recursos a ser criado no formato JSON. Se os dados contiverem um id de um recurso que já existe, lança o código 409 *CONFLICT* e adiciona uma chave *Location* ao cabeçalho com a URI do recurso já existente. Retorna o código 400 *BAD REQUEST* caso os dados enviados pelo cliente sejam inválidos ou não possam ser transformados (diz-se parseados) como um objeto de modelo Java. Retorna 201 *CREATED* quando a operação é executada com sucesso, com um cabeçalho *Location* contendo a URI para o *resource* recém-criado.

3.3.3.4.2 GET

GET é acionado na recuperação de um recurso. Não recebe *headers* e nem *body*. A resposta é o recurso serializado em JSON, caso esse exista, retornando 200 OK. Na situação do *resource* não existir, lança 404 *NOT FOUND*.

3.3.3.4.3 PUT

PUT faz a edição de um recurso. Deve possuir o *header Content-Type* com o valor *application/json*. O *body* deve conter um JSON com o recurso a ser editado, identificado por seu ID. 400 *BAD REQUEST* será recebido quando o *body* for

inválido. 204 *NO CONTENT* é código de retorno no caso a edição tenha sido executada com sucesso.

3.3.3.4.4 DELETE

Responsável por excluir um recurso. Retorna 200 OK quando a operação executa sem problemas e 404 *NOT FOUND* quando o recurso a ser excluído não existe.

3.4 Front-end

É a camada que permite o uso da plataforma pelo usuário final, abstraindo toda a sua complexidade. Foi inteiramente escrita com o AngularJS, *framework* Javascript responsável por tratar da conexão com a API, validar entradas de usuário, criar/manter a sessão e tratar das rotas.

3.4.1 Integração com a API

O AngularJS faz a integração por meio de chamadas HTTP. Para cada *endpoint* da API foi escrito um método Javascript. Esses métodos estão contidos dentro de estruturas chamadas *services*, que são objetos com escopo limitado ao *framework*. Esses objetos são depois injetados dentro dos *controllers*, de forma que possam ser utilizados. Na figura quatro é apresentado um trecho de código de um *service* contendo métodos para manipulação de *posts.*

FIGURA 4. *Service* de manipulação de *posts*

```
app.service('PostService', function ($http, AuthService, Upload, BASE_URL) {
  var postService = {};

  postService.saveDraft = function (draft) {
    return $http({
      method: "POST",
      url: BASE_URL + "/v2/posts",
      headers: {
        "Authorization": "Bearer " + AuthService.getToken()
      },
      data: draft
    }).then(function success(response) {
      return response;
    }, function error(response) {
      return {};
    });
  };

  postService.findAll = function() {
    return $http({
      method: "GET",
      url: BASE_URL + "/v2/posts/all",
      headers: {
        "Authorization": "Bearer " + AuthService.getToken()
      }
    }).then(function success(response) {
      return response.data;
    }, function error(response) {
      return [];
    });
  };
```

Fonte: AUTOR (2017).

Pode se visualizar dois métodos. A função *saveDraft* efetua uma requisição POST para salvar uma postagem, passando um JSON representando o objeto a ser salvo. *findAll* requisita por meio de um GET todos os *posts* do usuário autenticado. Ambas passam o *bearer token* por meio de um *header*.

3.4.1.1 Tratamento de sessão

No padrão REST não existe o conceito de sessão, todas as chamadas são *stateless*, portanto, todos os detalhes envolvendo a geração e manutenção dos *tokens* são tratados pelo cliente. No momento do login, o Angular chama o *endpoint* de autenticação e recebe o *token*. O *token* é então armazenado em um *cookie* do navegador, para que possa ser recuperado e passado em cada requisição em que seja necessário. A validade também é tratada, já que a aplicação renova o tempo de expiração do *token* após aproximadamente trinta minutos.

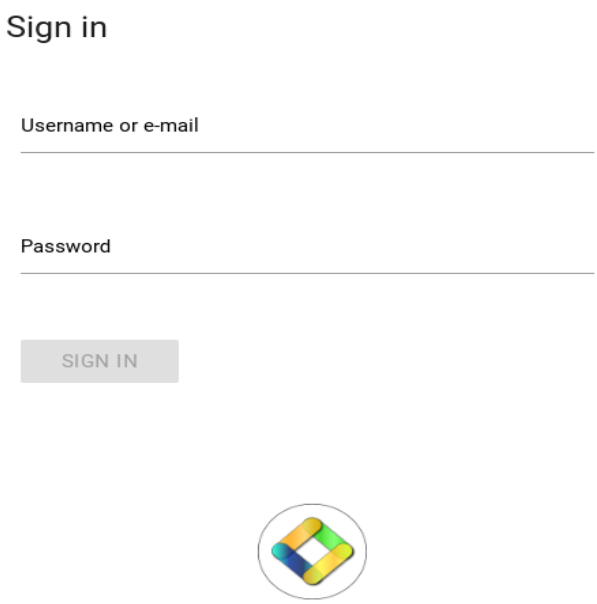
3.4.2 Interface de usuário

A GUI foi desenvolvida em HTML e CSS com o *framework* Materialize. Segue o *Material design*. A ideia foi gerar uma interface com o mínimo de elementos possíveis, facilitando seu uso por usuários sem treinamento. O resultado foi uma aplicação SPA.

3.4.2.1 Interface de login

É a tela inicial da aplicação, onde o usuário se autentica. Consiste apenas de um formulário com os campos para inserção do nome de usuário/e-mail, senha e de um botão. O botão só se torna ativo quando todas as informações são digitadas, apresentando a cor verde. Ele desaparece quando clicado, dando lugar a um *spinner* para denotar carregamento. Na parte inferior, há um rodapé transparente, exibindo apenas o logotipo da Agência Code, levando ao seu site quando clicado. Essa interface apresenta a mesma aparência tanto em *desktops* quanto em dispositivos móveis, apenas se adequando ao tamanho de tela. Na figura cinco é apresentada a tela de login.

FIGURA 5. Tela de login

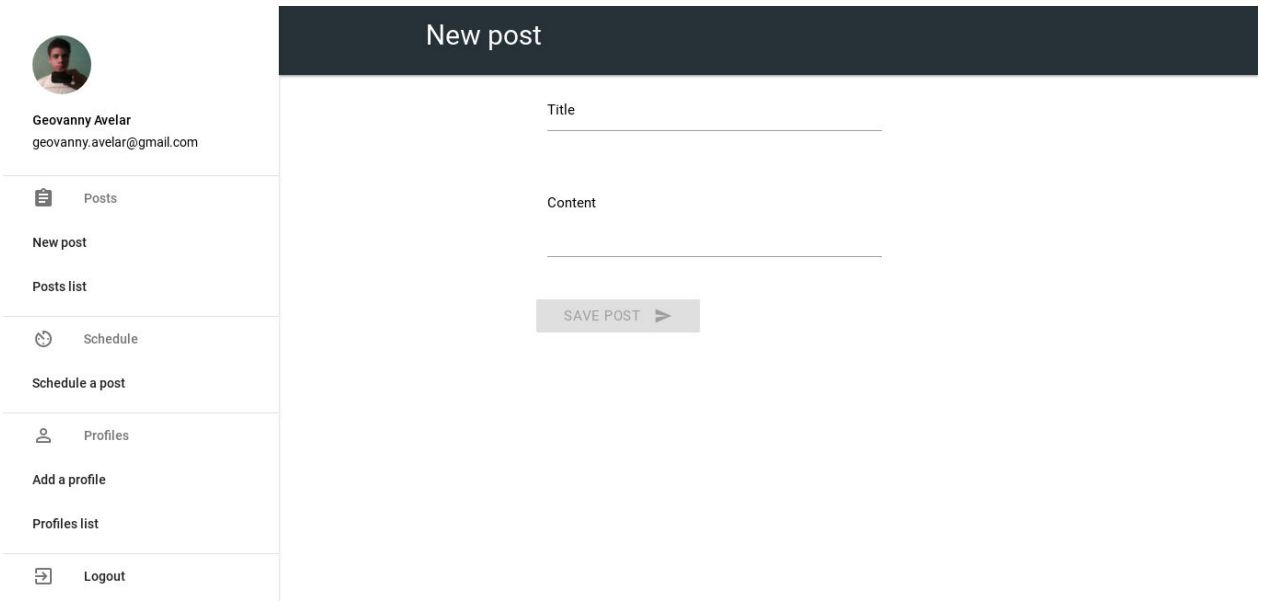


Fonte: AUTOR (2017).

3.4.2.2 Interface de cadastro e consulta

É para onde o usuário é redirecionado após o login. É uma interface simples onde o usuário pode inserir e consultar postagens, agendamentos e perfis. Em *desktops,* é composta por uma barra lateral fixa, chamada de *navbar,* que sempre se mantem à esquerda, independente da ação executada. A *navbar* mostra dados do usuário autenticado em sua parte superior, como um foto, o nome e e-mail. Logo abaixo existe um menu com três categorias: *Posts* (postagens), *Schedule* (agendamentos) e *Profiles* (perfis). Por fim, há um botão de *logout*. No lado direito, ocupando a maior parte da tela, é exibido o conteúdo requerido pelo usuário por meio da *navbar.* Por se tratar de uma *single page application*, não acontece o recarregamento quando se aciona um item do menu ou quando se interage com os componentes. O conteúdo é exibido de forma assíncrona, utilizando AJAX. A figura seis apresenta essa tela em *desktops*.

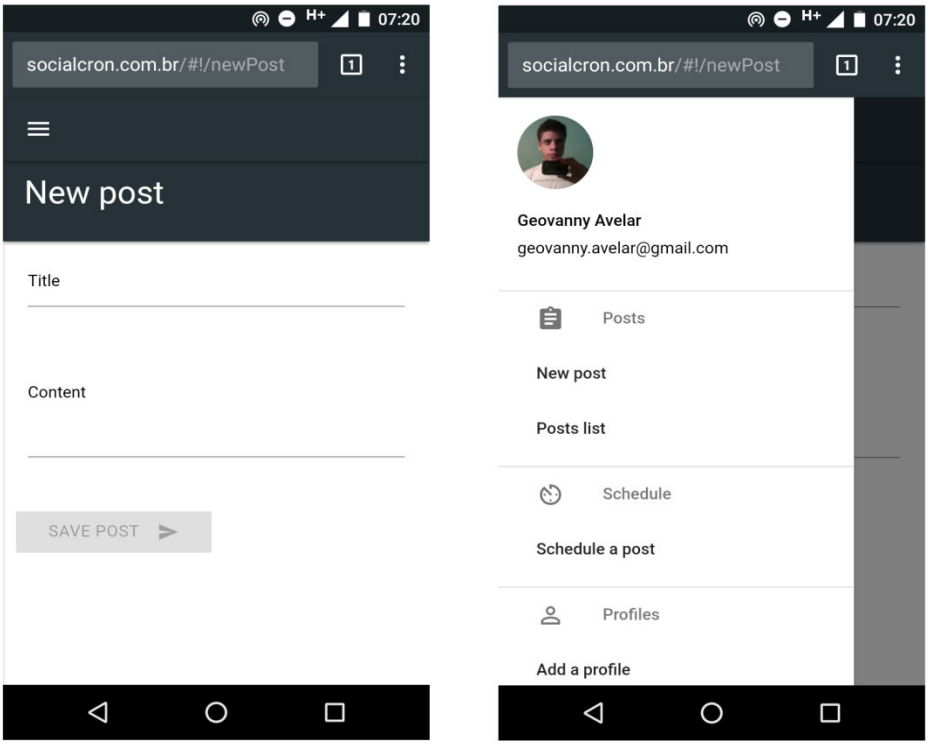
FIGURA 6. Interface em *desktops*



Fonte: AUTOR (2017).

Em *tablets* e *smartphones*, devido à falta de espaço lateral, a *navbar* fica oculta, sendo apenas acionada quando o usuário clica no ícone dos três segmentos na barra superior. A *navbar* desaparece quando um item do menu é clicado. A barra superior é apenas exibida no *mobile*. O conteúdo ocupa a tela inteira. A figura sete exibe a interface em um aparelho celular.

FIGURA 7. Interface em dispositivos móveis



Fonte: AUTOR (2017).

3.5 Agendador de postagens

O agendador é a peça de *software* responsável por selecionar as postagens que devem ser executadas em um determinado dia e enviá-las para a API do Facebook nos horários corretos. Ele foi escrito em Python. São apenas dois pequenos *scripts*, um que determina as postagens a executar naquele dia e outro que executa as postagens nos horários corretos. Esses *scripts* são chamados periodicamente utilizando-se do CRON.

3.5.1 Autenticação com a API

Da mesma forma que qualquer cliente, o agendador deve se autenticar antes de executar uma ação. Isso é feito obrigatoriamente com uma conta do tipo ADMIN, pois o *endpoint* usado para recuperar todos os agendamentos só pode ser acessado por esse tipo de usuário. A figura sete mostra o trecho de código responsável por enviar a requisição de autenticação.

FIGURA 8. Autenticação do agendador junto à API

```
auth_request = Request(BASE_URL + "oauth/token?username=geovanny.avelar@gmail.com&password=root&grant_type=password")
auth_request.add_header("Authorization", "Basic c29jaWFsY3Jvbjpzb2NpYWxjcm9u")
auth_response = json.loads(urlopen(auth_request, data="").read())

if auth_response['expires_in'] <= 120:
    refresh_request = Request(BASE_URL + "oauth/token?grant_type=refresh_token&refresh_token=" + auth_response['refresh_token'])
    refresh_request.add_header("Authorization", "Basic c29jaWFsY3Jvbjpzb2NpYWxjcm9u")
    refresh_request = json.loads(urlopen(refresh_request, data="").read())
    auth_response = refresh_request
```

Fonte: AUTOR (2017).

O código da figura oito inicia fazendo uma requisição do tipo POST com o *endpoint* de autenticação da API. Findada essa requisição, é recebida como resposta um JSON contendo o *bearer token*, *refresh token* e o tempo de expiração destes. Caso o *token* tenha menos de 120 (cento e vinte) segundos de “vida”, é feita outra requisição, dessa vez para efetuar o *refresh*, reiniciando o tempo para 1799 (mil setecentos e noventa e nove) segundos.

3.5.2 Sincronização dos agendamentos

A sincronização dos agendamentos entre a API e o agendador representou um problema. O funcionamento básico seria de recuperar os posts a cada um minuto, porém essa abordagem não é satisfatória em termos de desempenho. Se desenvolvendo dessa maneira, haveriam 1440 (Mil quatrocentas e quarenta) requisições por dia sendo disparadas contra o serviço. A solução encontrada foi armazenar os dados relativos aos agendamentos localmente e sincronizá-los com uma periodicidade menor. Determinou-se que de quinze em quinze minutos o agendador recuperaria os dados e os armazenaria em um banco de dados NoSQL, o Redis. Por se tratar de um banco em memória, há a inexistência de gargalos de desempenho relacionados ao armazenamento em mídia secundária.

Existem casos onde um agendamento é criado para ocorrer dentro dos próximos quinze minutos, dessa maneira o agendador não o sincronizaria e o post jamais seria executado. Para resolver essa questão, o agendador possui a funcionalidade de inserção dos agendamentos exposta por meio de uma URI. Essa funcionalidade foi escrita utilizando o *microframework* Flask.

3.5.3 Execução das postagens

O *script* que faz a execução das postagens é invocado a cada um minuto. Ele faz a leitura dos dados presentes no Redis e verifica um por um se o agendamento deve ser executado no minuto corrente, e, caso positivo, envia para a API do Facebook uma requisição para que a postagem seja feita. A figura nove contém o trecho responsável por executar essa ação.

FIGURA 9. O envio das postagens para a API do Facebook

```
for key in schedules_keys:
    schedule = json.loads(redis.get(key))
    schedule_date = datetime.strptime(schedule['date'][0:-5], "%Y-%m-%dT%H:%M")
    token = schedule['profile']['token']

    if schedule_date > datetime_now and schedule_date < one_minute_after:

        if 'photos' in  schedule['post']:
            photos_ids = facebook.savePhotos(schedule['post']['photos'], token)

        content = schedule['post']['content']
        thread = Thread(target=facebook.sendPost, args=(content, photos_ids, token))
        thread.start()
```

Fonte: AUTOR (2017).

O código da figura nove inicia procurando todas as chaves armazenadas no Redis, então busca o valor de cada uma, que é um JSON com os dados do agendamento. Feito isso, é verificado um por um se tal agendamento tem sua data dentro do minuto atual. Caso esteja, o *post* deve ser feito naquele momento. É verificado então se existem fotos a serem postadas, e via API do Facebook, cada uma delas é salva e um *id* é retornado. Essas imagens ainda não são públicas, mas estão disponíveis para serem atreladas ao *post* que será feito. Com todas as fotos salvas, a postagem é em seguida executada, com seu conteúdo em texto e as imagens, que agora estão ligadas a esse *post*. Para evitar que a execução do programa fique parada esperando por uma resposta da API, a requisição é feita dentro de uma *thread*. A *thread* é um recurso que permite a divisão de tarefas de maneira que possam ser executadas de maneira concorrente, ou seja, ao mesmo tempo. (NATALI MARIZ, [2016])

4- RESULTADOS

Este capítulo visa apresentar os resultados obtidos durante o desenvolvimento da plataforma, documentado no capítulo três.

4.1 Repositórios de código aberto

Sendo um projeto *open source*, três repositórios de código públicos foram gerados durante o desenvolvimento. Um contendo a API, outro o *front-end* e o último os fontes do agendador. Todos foram hospedados no GitHub. Existem espelhos no GitLab e no BitBucket, porém os repositórios ditos oficiais são os do GitHub.

4.1.1 SocialCRON-API

Contém os fontes da API do SocialCRON, sendo o maior repositório em volume de código. Formado por 2702 (duas mil setecentos e duas) linhas em Java, equivalendo a aproximadamente 57.06% do total de código da plataforma (em 28 de Maio de 2017). Está disponível em https://github.com/geovannyAvelar/SocialCRON-API.

4.1.2 SocialCRON-FRONT

Armazena o código da camada *front-end*. Contém 926 (novecentas e vinte seis) linhas em Javascript, 768 (setecentas e sessenta e oito) em HTML e 143 (cento e quarenta e três) em CSS, totalizando 1837 (mil oitocentas e trinta e sete). Corresponde a aproximadamente 39.80% do total de código plataforma (em 28 de Maio de 2017). Está disponível em https://github.com/geovannyAvelar/SocialCRON-FRONT.

4.1.3 SocialCRON-DISPATCHER

Guarda o código do agendador de postagens. Constituído por 143 (cento e quarenta e três) linhas em Python, representando aproximadamente 3.02% da plataforma (em 28 de Maio de 2017). Está disponível em https://github.com/geovannyAvelar/SocialCRON-DISPATCHER.

4.2 Processo de build

Essa seção visa explicitar as etapas de geração dos artefatos de software da plataforma a partir dos repositórios do GitHub. Tal processo é descrito na documentação que acompanha cada repositório e é reproduzido nas seções seguintes.

4.2.1 API

O *build* da API envolve a compilação dos códigos Java e a geração das tabelas do banco de dados relacional. O repositório pode ser transferido com apenas um comando do Git, como mostra a figura dez.

FIGURA 10. Transferência do repositório da API

```
geovanny@GEOVANNY-NOTE:~$ git clone https://github.com/geovannyAvelar/SocialCRON-API.git
```

Fonte: AUTOR (2017).

4.2.1.1 Banco de dados

O repositório contém na sua raiz um arquivo chamado socialcron-ddl.sql. Ele contém comandos SQL para a geração das tabelas do SocialCRON. Esse *script* não cria o banco de dados, deve ser criado um antes da sua execução. Se utilizou apenas MySQL durante o desenvolvimento, portanto não existe certeza de seu funcionamento em outros sistemas de banco de dados.

4.2.1.2 Configuração

A configuração da API é feita por meio do arquivo src/main/resources/application.properties. A figura onze mostra um trecho desse arquivo onde são definidas configurações de conexão com o banco de dados, como a URL (contendo o *host* e o nome da base de dados), usuário e senha.

FIGURA 11. Configuração de banco de dados

```
# Set here configurations for the database connection
# Connection url for the database
spring.datasource.url = jdbc:mysql://localhost:3306/socialcron

# Username and password
spring.datasource.username = root
spring.datasource.password = root
```

Fonte: AUTOR (2017).

A figura doze exibe os parâmetros de configuração do OAuth, como os dados para autenticação e o tempo de expiração dos *tokens.*

FIGURA 12. Configuração do OAuth

```
# ==============================
# = OAUTH 2
# ==============================

authentication.oauth.clientid = socialcron
authentication.oauth.secret = socialcron
authentication.oauth.tokenValidityInSeconds = 1800
```

Fonte: AUTOR (2017).

4.2.1.3 Compilação

Existem duas alternativas quando se trata do procedimento de compilação. É possível empacotar o artefato em um arquivo do tipo war ou executar a aplicação localmente, normalmente utilizada apenas para teste. Ambas são feitos com comandos do Maven. As figuras treze e quatorze mostram esses comandos.

FIGURA 13. Comando para geração do arquivo .WAR

```
geovanny@GEOVANNY-NOTE:~/SocialCRON-API$ mvn install -DskipTests
```

Fonte: AUTOR (2017).

A opção da figura treze gera um arquivo chamado socialcron-API-x.war dentro da pasta *target*, na raiz do projeto. O ‘x’ é o número da versão compilada. Pode ser feito *deploy* desse arquivo em um servidor de aplicação.

FIGURA 14. Comando para execução local

Fonte: AUTOR (2017).

O comando descrito na figura quatorze faz a compilação e roda um servidor embarcado na própria máquina. A aplicação pode ser acessada em http://localhost:8080. É exibido um *log* em tempo real.

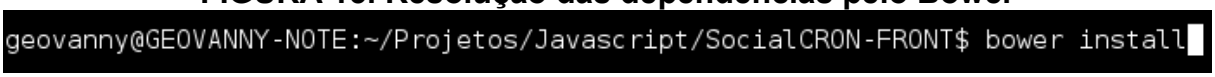
4.2.2 Front-end

A instalação do *front-end* trata da resolução de dependências, da concatenação e minificação dos arquivos Javascript.

4.2.2.1 Dependências

A obtenção das dependências é feita pelo Bower, com o comando da figura quinze. É feita a leitura do arquivo bower.json, contendo a relação dos artefatos a serem baixados.

FIGURA 15. Resolução das dependências pelo Bower



Fonte: AUTOR (2017).

Todos os arquivos necessários são salvos no diretório *bower_components*.

4.2.2.2 Organização do artefato

Visando uma melhor organização do projeto, as dependências devem ser movidas para outro diretório. Há também a necessidade de se executar dois processos. O primeiro deles é a concatenação, tarefa onde todos os arquivos Javascript são unidos em apenas um, evitando assim que o navegador precise fazer

inúmeras requisições. O segundo é a minificação, ação que retira todos os caracteres não essenciais do código Javascript, como espaços e tabulações, objetivando arquivos com menos conteúdo, e, por consequência, menor uso de rede durante as transferências. Esses procedimentos são repetitivos, propensos a erros e tomam tempo considerável quando executados manualmente, portanto são automatizados. O Gulp é utilizado como ferramenta de automação, sendo configurado por um arquivo chamado Gulpfile.js, presente na raiz do projeto. A figura dezesseis mostra um trecho desse arquivo, onde são executados os comandos para mover as dependências, concatená-las e minificá-las em um único arquivo chamado socialcron.min.js.

FIGURA 16. Arquivo Gulpfile.js

```
gulp.task('assets-dist', function() {
    gulp.src(jsFilesToMove)
        .pipe(gulp.dest('assets/components/js'));
    gulp.src(cssFilesToMove)
        .pipe(gulp.dest('assets/components/css'));
    gulp.src(fontsFilesToMove)
        .pipe(gulp.dest('assets/components/fonts'));
});

gulp.task('minify-js', function () {
  gulp.src(jsToMinify)
    .pipe(concat('socialcron.min.js'))
    .pipe(uglify({ mangle: false }))
    .pipe(gulp.dest('app/js'));
});

gulp.task('default', [ 'assets-dist', 'minify-js' ]);

gulp.task('watch', function() {
  gulp.watch(css, ['minify-js', 'minify-css']);
});
```

Fonte: AUTOR (2017).

Para executar o *script*, basta digitar o comando Gulp na raiz do projeto. Com tais procedimentos feitos, o artefato pode ser movido para um servidor *web*.

4.2.3 Agendador

O *build* do agendador envolve apenas a aquisição das dependências e a configuração das *crontabs*.

4.2.3.1 Dependências

As dependências são obtidas por meio do PIP. A listagem delas se encontra no arquivo requirements.txt, na raiz do projeto. A figura dezessete mostra o comando para a obtenção.

FIGURA 17. Obtenção das dependências com o PIP

Fonte: AUTOR (2017).

4.2.3.2 Crontabs

Os *scripts* do agendador são acionados periodicamente por *crontabs.* O *cron* é um *software* presente nos sistemas Linux que permite o agendamento de tarefas. As chamadas são definidas no arquivo de configuração do *cron*, com o comando da figura dezoito.

FIGURA 18. Edição do arquivo *crontab*

Fonte: AUTOR (2017).

Esse comando abre o arquivo *crontab.* No fim desse arquivo devem ser adicionadas duas linhas. A primeira para chamar o *script* de sincronização e a segunda invocando o de execução das postagens. A figura dezenove mostra as linhas.

FIGURA 19. Definição das *crontabs*

```
* * * * * root /usr/bin/python /root/d/exec
*/15 * * * * root /usr/bin/python /root/d/sync
```

Fonte: AUTOR (2017).

4.3 Licença

Todo o *software* está coberto pela licença MIT. Entende-se por *software* todo o conjunto de código-fonte e documentação, incluindo esse trabalho. Todo artefato acompanha uma cópia do texto da licença. O texto pode ser consultado na seção anexos deste documento.

A MIT *license* prevê, como permissões:

- Uso comercial;
- Modificação;
- Distribuição;
- Uso privado

Sob as condições de:

- Inclusão da licença em porções substanciais do código.

Com limitações de:

- Responsabilidade;
- Garantia.

(OPEN SOURCE INITIATIVE, [2017]).

4.4 Conclusão

Esse estudo intencionou mostrar a possibilidade técnica da implementação de um *software* para automação de postagens em sites de mídias sociais. Ficou explícito, em especial nos capítulos três e quatro, que o SocialCRON engloba determinadas características que o tornam adequado às funções e limites de desempenho definidos durante o levantamento de requisitos e desenvolvimento, que são os seguintes:

- Desempenho satisfatório;
- Cumprimento dos requisitos funcionais, em especial a postagem nos horários requisitados;

- Simplicidade da interface de usuário;
- Uso quase total de ferramentas de código aberto no desenvolvimento;
- Abertura do código da plataforma sob licença MIT.

Porém, ao se considerar que as intenções da Agência Code Plus, mentora do projeto, são de tornar a aplicação viável economicamente, nota-se que o SocialCRON falhou completamente nesse aspecto, o que por si só invalida qualquer qualidade técnica que possa existir. A presença no mercado de aplicações com uma gama dezenas de vezes maior de funcionalidades, desempenho superior, já consolidadas e com grande aporte financeiro cria barreiras muito difíceis de serem transpostas. Levando em conta as inferioridades técnicas e econômicas, o único uso possível da aplicação é para fins acadêmicos, como mostra do uso das tecnologias e conceitos aplicados no seu desenvolvimento, sem chance alguma de se tornar uma solução madura e que se equipare aos seus concorrentes.

5- TRABALHOS FUTUROS

Este capítulo apresenta possibilidades de trabalhos futuros passíveis de serem executados utilizando como base esse estudo. Por se tratar de um projeto de *open source*, há a liberdade de modificação dos artefatos e reaproveitamento do código.

Essas são as opções:

a) Desenvolvimento de um aplicativo para dispositivos móveis com as mesmas funcionalidades do *front-end* nas plataformas *Android* e iOS;
b) Reescrita do agendador, com o objetivo de fazê-lo trabalhar em um *cluster*, de maneira que as postagens sejam distribuídas para a execução em diferentes nós;
c) Adicionar compatibilidade com os seguintes serviços de mídia social: Twitter, Linkedin, Instagram e Google Plus;
d) Escrita de *wrappers* para acesso da API do SocialCRON nas seguintes linguagens de programação: Python, Java, C# e PHP;
e) Localização (l10n) da interface de usuário em português brasileiro;
f) Implementar o suporte a *analytics*.

Referências

KAPLAN, Andreas. **Users of the world, unite!**: The challenges and opportunities of Social Media. Bloomington: Elsevier, Janeiro–Fevereiro 2010.

MONTEIRO, Diego. AZARITE, Ricardo. **Monitoramento e métricas de Mídias Sociais**: do estagiário ao CEO: um modelo prático para toda empresa usar mídias sociais com eficiência e de forma estratégica. São Paulo: DVS Editora, 2012.

DEITEL, H. M.; DEITEL, P. J. **Java: Como Programar**. Oitava edição. São Paulo: Pearson Education, 2010.

Maven. Welcome to Apache Maven. Disponível em: https://maven.apache.org/. Acessado em: 28/05/2017.

WALLS, Craig. **Spring in action**. Quarta edição. Manning, 2015.

Hibernate ORM. Disponível em: http://hibernate.org/orm/. Acessado em: 28/05/2017.

BORGES, Luiz Eduardo. **Python para desenvolvedores**. Segunda edição. Edição do autor, 2010.

MySQL Community Edition. Disponível em: https://www.mysql.com/products/community/. Acessado: 28/05/2017.

LAZOTI, Rodrigo. **Armazenando dados com Redis**. Edição única. Casa do código. 2014.

WILLIAMSON, Ken. **Learning AngularJS**: A guide to AngularJS development. Primeira edição. O'Reilly Media. 2015.

BIBEAULT, Bear. **JQuery in Action**. Terceira edição. Manning. 2015.

MySQL Community Edition. Disponível em: https://www.mysql.com/products/community/. Acessado em: 28/05/2017.

About - Materialize CSS. Disponível em: http://materializecss.com/about.html. Acessado em: 28/05/2017.

Bower – a package manager for the web. Disponível em: https://bower.io/. Acessado em: 28/05/2017.

Gulp.js. Disponível em: http://gulpjs.com/. Acessado em: 28/05/2017.

SILVERMAN, Richard. **Git pocket guide**. Primeira edição. O'Reilly Media. 2013.

OAuth 2.0. Disponível em: https://oauth.net/2/. Acessado em: 28/05/2017.

Flask. Disponível em: http://flask.pocoo.org/. Acessado em: 28/05/2017.

SAUDATE, Alexandre. **REST: Construa API's inteligentes de maneira simples**. Edição única. Casa do código. 2014.

RICHARDSON, Leonard. **RESTFul Web Services**. Primeira edição. O'Reilly Media. 2015.

NATALI MARIZ, Raquel. **SafeHome 1.0**. 2016. 46f. Trabalho de Graduação - FATEC de São José dos Campos: Professor Jessen Vidal.

Open Source Initiative. Disponível em: https://opensource.org/licenses/MIT. Acessado em: 30/05/2017.

ANEXOS

MIT License