Chapter 5

Automação de Tarefas com *APIs*: Integração entre *Google Calendar* e *WhatsApp* para Otimização de Processos

Francisnilto dos Santos Nascimento, Luis Guilherme Sampaio Fontenele, Ricardo Moura Sekeff Budaruiche e Vanessa Pereira Cunha

Abstract

This chapter discusses the development of a task automation solution by integrating the Google Calendar API with the WPPConnect Server, aiming to automate the delivery of personalized notifications via WhatsApp. The proposal employs an architecture based on RESTful APIs, secure authentication through OAuth 2.0, and modular configuration using Python. The configuration of APIs, daily retrieval of events and tasks, and automated message delivery processes are described. The approach highlights security best practices and the challenges related to using unofficial APIs. The proposed solution contributes to optimizing schedule management and disseminating automation processes in personal and corporate environments.

Resumo

Este capítulo aborda o desenvolvimento de uma solução para automação de tarefas, integrando a API (Application Programming Interface) do Google Calendar ao WPPConnect Server, com o objetivo de automatizar o envio de notificações personalizadas via WhatsApp. A proposta utiliza uma arquitetura baseada em APIs RESTful, com autenticação segura via OAuth 2.0 e configuração modular em Python. São descritas as etapas de configuração das APIs, recuperação de eventos e tarefas diárias e envio automatizado de mensagens. A abordagem destaca boas práticas de segurança e os desafios associados ao uso de APIs não oficiais. A solução proposta contribui para a otimização da gestão de compromissos e para a disseminação de processos de automação em ambientes pessoais e corporativos.

5.1. Introdução

Na era da informação e da hiperconectividade, a automação de tarefas consolidou-se como uma estratégia indispensável tanto no cotidiano quanto no ambiente corporativo,

permitindo otimizar rotinas, reduzir o esforço manual e aumentar significativamente a produtividade. A adoção de arquiteturas baseadas em serviços e interfaces bem definidas facilita a integração eficiente de diferentes sistemas. Essa tendência é impulsionada pela crescente utilização de APIs, que viabilizam a comunicação entre softwares de forma prática e escalável (SOMMERVILLE, 2019).

O conceito de *APIs* refere-se a um conjunto de regras e especificações que permitem que *softwares* distintos se comuniquem entre si. A adoção de *APIs* tem transformado a maneira como aplicações são desenvolvidas e integradas, especialmente com a difusão da arquitetura *REST* (*Representational State Transfer*), proposta por Roy Fielding em sua tese de doutorado em 2000, amplamente reconhecida como um marco na evolução dos sistemas distribuídos e da *Web* moderna. A *API* do *Google Calendar*, por exemplo, é uma *API RESTful*, o que significa que segue essa arquitetura, baseada em diretrizes que visam uma comunicação simples, escalável e eficiente entre sistemas distribuídos (GOOGLE DEVELOPERS, 2025).

A arquitetura *REST* é um estilo de arquitetura para sistemas distribuídos, projetado para garantir a comunicação eficiente entre sistemas. Ela utiliza o protocolo **HTTP** (*Hypertext Transfer Protocol*) como meio de comunicação, com operações padrão como consulta (*GET*), envio de dados (*POST*), atualização (*PUT*) e exclusão (*DELETE*) (FIELDING, 2000). A principal característica do *REST* é sua simplicidade e escalabilidade, permitindo que os sistemas sejam facilmente integrados e interoperáveis.

Segundo estudos (FIELDING, 2000), um sistema que segue a arquitetura *REST* deve ser *stateless* (sem estado), o que significa que cada requisição deve ser independente e conter todas as informações necessárias para ser processada, sem depender de sessões anteriores. Além disso, *REST* facilita a troca de dados entre sistemas distribuídos por meio de representações padronizadas, como *JSON* (*JavaScript Object Notation*) ou *XML* (*Extensible Markup Language*), garantindo uma comunicação eficiente e escalável.

Automatizar significa utilizar tecnologias capazes de executar ações sem a necessidade de intervenção humana contínua, otimizando processos e eliminando atividades repetitivas (BARRETO et al., 2024). Um exemplo prático é a integração entre o *Google Calendar* e o *WhatsApp*: imagine que, ao iniciar o dia, um usuário receba automaticamente, via WhatsApp, uma mensagem contendo seus compromissos agendados, proporcionando organização pessoal e ganhos de produtividade.

Ferramentas como o *Google Calendar* são fundamentais neste cenário, pois oferecem *APIs* bem documentadas e estáveis, facilitando integrações com outros serviços, como *Gmail*, *Google Meet* e *Google Tasks* (MELO MESSIAS et al., 2022). Da mesma forma, o *WhatsApp* também disponibiliza uma *API oficial*, a *WhatsApp Business API*, que permite a automação de interações e integrações com sistemas externos. Embora a *API* oficial do *WhatsApp* seja paga e voltada para empresas, ela oferece uma solução robusta e escalável para comunicação via *WhatsApp*.

No entanto, alternativas como o *WPPConnect*, uma *API* não oficial, também são amplamente utilizadas em projetos que buscam uma solução mais acessível e simples de implementar, embora com algumas limitações e riscos em termos de estabilidade e conformidade (WPPCONNECT, 2025). A arquitetura *REST*, utilizada tanto por essas *APIs* quanto por outras, é caracterizada pela escalabilidade, simplicidade na troca de informações (FIELDING, 2000).

Neste capítulo, exploraremos como a automação de tarefas pode ser realizada através da integração entre o *Google Calendar* e o *WhatsApp*, utilizando a arquitetura *REST* e *APIs* específicas para otimizar a gestão de compromissos pessoais e profissionais.

5.1.1. Conceitos e Vantagens da Automação de Tarefas.

Na atualidade, gerenciar compromissos e tarefas seja no ambiente empresarial ou na vida pessoal pode ser desafiador e repetitivo. Em um mundo dinâmico, onde a eficiência na organização é essencial, a automação de tarefas surge como uma solução indispensável, pois é capaz de otimizar processos e liberar tempo para atividades estratégicas (SANTOS, 2023).

A automação de tarefas proporciona a **redução de erros manuais** ao seguir padrões definidos, garantindo a **padronização e a execução** uniforme dos processos. Além disso, promove **economia de tempo** ao liberar o usuário de atividades repetitivas, permitindo maior foco em questões estratégicas. Como consequência, há um **aumento da produtividade**, com a diminuição do tempo gasto em tarefas operacionais e a melhoria na agilidade dos processos. A automação também representa um passo importante rumo à **transformação digital**, favorecendo a eficiência operacional e a inovação nos setores público e privado (BRANDAO, 2021).

5.1.2. Exemplos Práticos de Automação no Cotidiano Pessoal e Corporativo.

A automação está cada vez mais presente em diferentes aspectos da vida moderna, muitas vezes de forma quase imperceptível, simplificando tarefas rotineiras e aprimorando a experiência do usuário. No cotidiano, é possível observar a aplicação de soluções automatizadas como o envio automático de mensagens de aniversário ou lembretes via *WhatsApp*, a sincronização de agendas com alertas em tempo real e a geração automática de relatórios a partir de planilhas *online*. Esses recursos tornam-se aliados importantes na organização pessoal e na economia de tempo.

No ambiente corporativo, a automação também exerce um papel fundamental ao otimizar processos e melhorar a comunicação entre equipes. Exemplos comuns incluem o envio de alertas de reuniões com base em calendários compartilhados, o processamento automatizado de pagamentos e a emissão de faturas, além da integração entre sistemas de *CRM* (*Customer Relationship Management*) e plataformas de atendimento ao cliente. Tais aplicações, frequentemente viabilizadas por meio de *APIs*, demonstram como soluções tecnológicas simples e bem projetadas podem promover ganhos expressivos em produtividade, gestão de tempo e eficiência operacional.

5.1.3. Tipos de *APIs* para Automação.

A base da automação moderna está na integração entre sistemas, viabilizada principalmente pelo uso de *APIs*. Cada tipo de *API* possui uma finalidade específica, e compreendê-las é essencial para a escolha das soluções mais adequadas a diferentes demandas. **APIs de integração**, por exemplo, permitem a troca de dados entre plataformas distintas, como ocorre com serviços como *Google Calendar*, *Trello* e *GitHub*. Já as **APIs de comunicação** possibilitam o envio automatizado de mensagens, alertas e notificações, sendo representadas por ferramentas como *Twilio*, *Telegram* e *WhatsApp* (via *WPPConnect*).

Outro grupo importante são as *APIs* voltadas a assistentes virtuais, que utilizam inteligência artificial para interpretar comandos e executar tarefas, como *Alexa Skills* e

Google Assistant. O domínio dessas interfaces confere maior autonomia no desenvolvimento de soluções automatizadas sob medida, com potencial para promover ganhos significativos em agilidade, organização e eficiência. Seja em contextos pessoais ou profissionais, o uso estratégico de *APIs* contribui diretamente para a transformação digital e a otimização de processos.

5.2. *APIs Google Calendar* e *WPPConnect Server* - Automatizando Eventos e Comunicações.

A crescente demanda por soluções tecnológicas que otimizem a comunicação e a organização de atividades impulsiona o desenvolvimento de aplicações integradas a serviços em nuvem por meio de *APIs*. Em contextos empresariais, educacionais e sociais, a automação de tarefas e o gerenciamento eficiente de eventos tornam-se diferenciais estratégicos. Nesse cenário, destacam-se duas ferramentas amplamente utilizadas: a *API* do *Google Calendar* e o *WPPConnect Server*.

A *API* do *Google Calendar* permite o gerenciamento programático de agendas, oferecendo funcionalidades como criação, edição e exclusão de eventos, além de notificações personalizadas. Por meio do protocolo *OAuth* 2.0, garante-se o acesso seguro e autorizado aos dados do usuário, respeitando os princípios de privacidade e proteção da informação. Já o *WPPConnect Server* surge como uma alternativa acessível à *API* oficial do *WhatsApp*, permitindo a automação de mensagens e interações via *WhatsApp Web*, com flexibilidade e controle sobre sessões autenticadas.

A integração dessas duas tecnologias possibilita a criação de soluções robustas que não apenas organizam compromissos e atividades, mas também notificam automaticamente os usuários por meio de canais amplamente utilizados, como o *WhatsApp*. Este capítulo apresenta os conceitos, configurações e práticas de uso dessas *APIs*, visando apoiar o desenvolvimento de aplicações funcionais, escaláveis e centradas na experiência do usuário.

A Figura 5.1 ilustra a infraestrutura da solução desenvolvida, evidenciando a interação entre os principais componentes do sistema: as *APIs* do *Google Calendar* e *Google Tasks*, o servidor *WPPConnect* e o *script* de automação. O fluxo apresenta a sequência de operações desde a configuração inicial, passando pela consulta aos serviços de agenda, até o envio automatizado das mensagens via *WhatsApp*. Este diagrama visa proporcionar uma visão geral da arquitetura proposta, facilitando a compreensão das etapas descritas nas seções subsequentes.

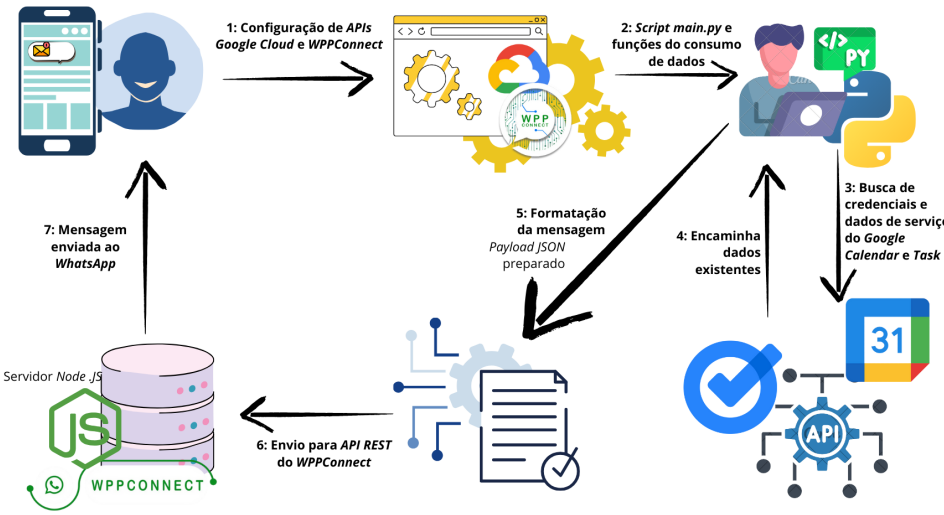

Figure 5.1: Arquitetura de integração para automatizar mensagens.

5.2.1. O Que é o *WPPConnect Server*?

O *WPPConnect Server* é uma solução de código aberto que viabiliza a integração com o *WhatsApp* por meio de uma interface *web*. Desenvolvido a partir da biblioteca *wppconnect*, esse servidor utiliza técnicas de **engenharia reversa** aplicadas ao protocolo *WebSocket* do *WhatsApp Web*, permitindo funcionalidades como envio, recebimento e gerenciamento de mensagens (WPPCONNECT, 2025).

Engenharia reversa é definida como o processo de análise de produtos ou protocolos existentes a partir de artefatos como binários, *memory dumps* ou tráfego de rede. Esse processo visa compreender a estrutura, lógica de operação e protocolos internos de sistemas, mesmo na ausência de documentação formal ou acesso ao código-fonte original (EILAM, 2011).

A utilização da engenharia reversa possibilita a construção de interfaces de comunicação com sistemas fechados, por meio de *APIs* não oficiais. Além disso, essa abordagem permite a recuperação de especificações de sistemas legados e a identificação de vulnerabilidades, contribuindo para ações de correção e auditoria.

No contexto do *WPPConnect Server*, observa-se o tráfego criptografado entre o cliente *web* do *WhatsApp* e seus servidores oficiais. Através da ferramenta *Puppeteer*, são reproduzidas as chamadas *WebSockets* e os *endpoints* utilizados pelo *WhatsApp Web*, os quais são expostos como uma *API REST*.

Contudo, por tratar-se de uma implementação não oficial da plataforma META, não há suporte oficial. Dessa forma, a cada modificação no *front-end* ou no protocolo do *WhatsApp*, torna-se necessário atualizar o processo de engenharia reversa para garantir a compatibilidade do sistema.

5.2.2. Criando um Projeto no *Google Cloud* para Uso da *API Google Calendar*.

Para utilizar a *API* do *Google Calendar* de maneira funcional e robusta, é necessário, primeiramente, criar um projeto na plataforma *Google Cloud Console*. Esse processo abrange desde a criação de um novo projeto até a geração de credenciais por meio de um arquivo *JSON* instalável.

Ao abrir a interface do *Google Cloud Console*, clique em "***Console***", localizado na parte superior da página. Posteriormente, será possível acessar o *Google Cloud Platform*,

que disponibiliza os meios de configurações e serviços de várias *APIs*, principalmente a *API* do *Google Calendar*. Acessando o *Google Cloud Platform* e depois logando uma conta google de usuário, crie um novo projeto ao clicar no botão "***My First Project***", logo, abrirá a janela de **seleção** ou **criação** de projeto.

Dessa forma, vamos criar um **novo projeto**, defina um nome de projeto, e, se necessário, selecione uma organização ou mantenha a opção padrão. Logo após a criação de seu projeto, certifique-se de que ele esteja selecionado no mesmo local que se encontrava a informação "***My First Project***", agora substituído pelo nome do seu projeto.

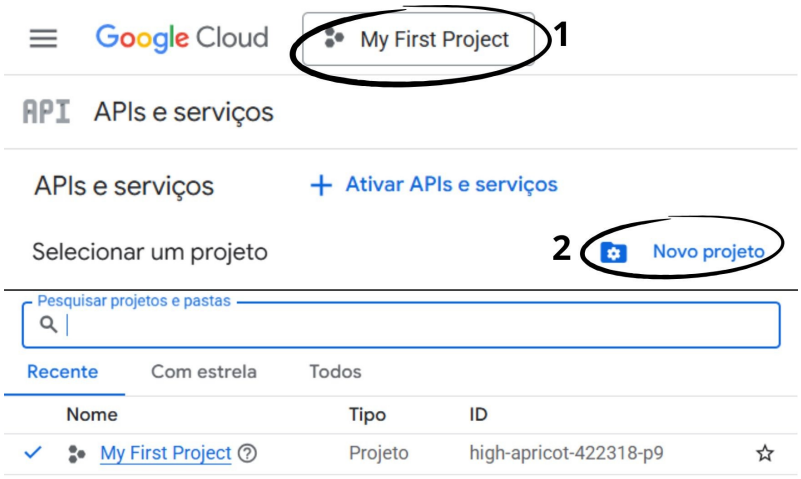

Figure 5.2: Criando um projeto no *Google Cloud*

Com o projeto selecionado, o próximo passo é acessar a seção responsável pela criação do **ID do Cliente *OAuth***. Clique nas **três barras** ao lado esquerdo superior da tela, procure por "***APIs* e Serviços**" e selecione por "**credenciais**".

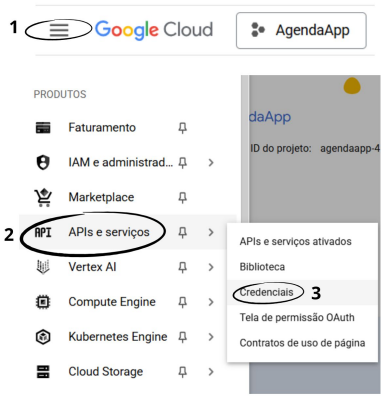

Figure 5.3: Passo a passo para acesso a tela de credenciais.

ID do Cliente é utilizado quando sua aplicação precisa acessar a *API* em nome de um usuário, permitindo a autenticação e autorização por meio da conta pessoal desse usuário. Esse tipo de credencial é essencial para aplicações que interagem diretamente com usuários e precisam de permissões específicas para acessar dados protegidos. Na parte superior da página de credenciais, clique em "**+ Criar credenciais**" e, em seguida, selecione a opção "**ID do Cliente *OAuth***".

Ao cadastrar uma conta ao *Google Cloud* pela primeira vez, a página de criação do ID do Cliente para a autenticação, antes de tudo, fará com que o usuário configure a **tela de consetimento**. Ao ser redirecionado para a página de configuração sobre permissões autenticadas, será necessário inserir dados como: **informações do *app*** - insira nome e *email* para suporte de usuário; **público** - escolha para público externo e **dados de contato** - recomenda-se inserir o mesmo *email* inserido em informações do *app*.

Figure 5.4: Passo a passo para acesso a tela de credenciais.

Após configurar a permissão de consetimento do usuário, a página será atualizada para a aba de "**visão geral de cliente *OAuth***". Logo, será possível criar e configurar um "**Cliente *OAuth***". Na mesma página de visão geral, em **métricas** clique em "**Criar um cliente *OAuth***".

Para criar um **Cliente *OAuth***, deve-se preencher os campos **tipo de aplicativo**, **nome do aplicativo** e ***URLs* de redirecionamento autorizados**. Para o "Tipo de aplicativo" deve ser **aplicação *web***, o "nome" fica ao critério coerente do usuário e para a "*URL* de redirecionamento autorizado" pode-se utilizar a chamada ***HTTP*** vinculada a um *localhost* da máquina, inicialmente apenas para testes de algoritmos trabalhados na sessão, por exemplo: `http://localhost:9090/`

Figure 5.5: Passo a passo para criar o Cliente *OAuth*.

Ressalta-se que, posteriomente, com toda a aplicação do sistema construída, a mesma passará para um deploy, subindo a aplicação para um servidor e com um domínio específico . Portanto, a *URL* de redirecionamento autorizado do *localhost* deve ser alterada para a nova *URL* do servidor, onde a aplicação estará rodando. Guarde esse arquivo de credenciais apra ser utilizado no código da aplicação.

Criando o cliente de autenticação do *Google*, será possível baixar o arquivo ***JSON*** com as respectivas credenciais do cliente. Veja ao anexo baixo a estrutura do arquivo *JSON*' gerado.

```
{"web":{
   "client_id": <"Seu ID de cliente">,
   "project_id":<"Seu ID de projeto">,
   "auth_uri":"https://accounts.google.com/auth", #Valor fictício
   "token_uri":"https://oauth.googleapis.com/token", #Valor fictício
   "auth_provider_x509_cert_url":"https://www.googleapis.com/oauth2/",
   "client_secret": <"Seu client_secret">,
   "redirect_uris":["http://localhost:9090/"]
   }
}
```

Listing 5.1: Dados de autenticação disponibilizados pelo *response body*

Ainda tratando-se de configurações para o Cliente *OAuth*, precisa-se realizar mais um passo: delimitar usuários testes que poderão usar as credenciais do ID de cliente criado para o projeto "AgendaAPP", pois o cliente não foi publicado. Logo, a autenticação vai ser bloqueada para as contas *emails* que não foram inseridas como conta teste para esse Cliente *OAuth* criado.

Para delimitar os usuários testes, procure por "***APIs* e Serviços**", selecione por "**Credenciais**", assim, é apresentada a janela que ficam registradas as "Chaves de *APIs*", "IDs do Cliente *OAuth* 2.0" e "Contas de Serviço". Em **IDs do Cliente *OAuth* 2.0**, escolha o Cliente ID criado e clique em editar - Ícone representado por um lápis na parte de ações.

Figure 5.6: Passo a passo para adicionar usuário testes.

Por fim, ative as *APIs* do *Google Calendar* e *Google Task*. Volte para a página inicial do *Google Cloud Console*, clique em "**Serviços e APIs**", logo em seguida, procure por **"Biblioteca"**. Ao acessar a interface de busca por serviços *Google*, insira na barra de pesquisa por ambas as *APIs*: "*Google Calendar*" e "**Google Task API**"

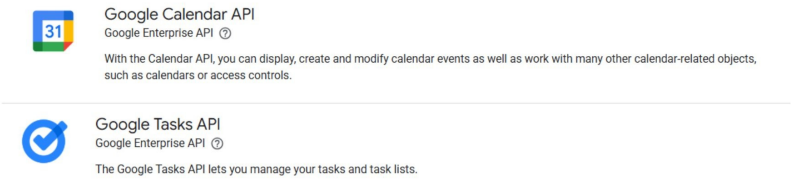

Figure 5.7: *APIs Google* necessárias para a automação de eventos e tarefas.

Como é possível ver na descrição da figura 5.7 para cada *API*, verifica-se que *Google Calendar API* é responsável por manipular os eventos agendados no *Google Calendar*; já *Google Tasks API* é um complemento funcional para o *Google Calendar*, que possibilita a manipulação de tarefas diárias agendadas para um determinado dia e horário.

5.2.3. Inicialização e Implantação Prática da *API*.

Antes de iniciar as implementações, o *WPPConnect Server* requer a instalação do Node.js, na versão 16.14 ou superior. Para verificar a versão instalada em sua máquina, utilize o comando `node -v` no *terminal*. Esse requisito se justifica pela necessidade de compatibilidade com funcionalidades modernas do *JavaScript/Node.js*, bem como com bibliotecas dependentes, além de proporcionar maior segurança, correção de *bugs*, e melhor desempenho no que se refere ao tempo de resposta de *APIs*, manipulação de grandes volumes de dados e processamento assíncrono.

Com o *Node.js* devidamente instalado, o próximo passo consiste em clonar o repositório oficial responsável pela estrutura da *API* e do servidor do *WPPConnect Server*. Após criar o diretório do projeto e acessá-lo por meio do *terminal*, *CMD* ou *PowerShell*, utilize o seguinte comando:

```
git clone https://github.com/wppconnect-team/wppconnect-server.git
```

Entre dentro da pasta *wppconnect-server* e instale todas as dependências da aplicação através do comando:

```
npm install
```

Atualmente, muitos projetos são desenvolvidos utilizando o ecossistema *JavaScript*, especialmente com o uso do *TypeScript*. O projeto *WPPConnect Server* adota essa abordagem. Nesse contexto, tanto a *API WPPConnect* quanto outros sistemas de *back-end* são construídos com recursos avançados do *Node.js*. Contudo, o *Node.js* não é capaz de interpretar código *TypeScript* de forma nativa, sendo necessário transpilar o código para *JavaScript*.

Para compilar a aplicação e inicializar o servidor, é necessário realizar o processo de *build* dentro do diretório *wppconnect-server*. Em seguida, o servidor pode ser executado por meio do comando de `start`, permitindo o acesso à interface do *WPPConnect Server* por meio do navegador.

npm run build

npm run start

Após a execução do comando `npm start`, torna-se possível acessar a interface *Swagger UI* da *API WPPConnect Server*. O ponto central para o desenvolvimento de uma automação básica de envio de mensagens via *WhatsApp* encontra-se nos *endpoints* do módulo de autenticação (*Auth*) da própria *API*.

A conexão entre o servidor e o *WhatsApp* é estabelecida por meio dos seguintes *endpoints* de autenticação:

```
POST/api/session/secretkey/generate-token
POST/api/session/start-session
```

```
GET/api/session/qrcode-session
```

Seguindo a ordem dos *endpoints* mencionados anteriormente, o primeiro a ser utilizado é o `POST/api/session/secretkey/generate-token`. Embora aspectos de segurança sejam discutidos mais detalhadamente em seção específica deste trabalho, é importante destacar que, para qualquer tentativa válida de conexão entre dados pessoais e o servidor, é necessário gerar um *token* de autenticação via ***JWT*** (*JSON Web Token*), utilizando uma *secretkey* - chave secreta vinculada à sessão a ser iniciada em um dispositivo.

Quando o servidor está configurado com autenticação baseada em *token*, esse *endpoint* deve ser invocado antes de qualquer outro, a fim de obter um *token* válido para autenticação nas requisições subsequentes.

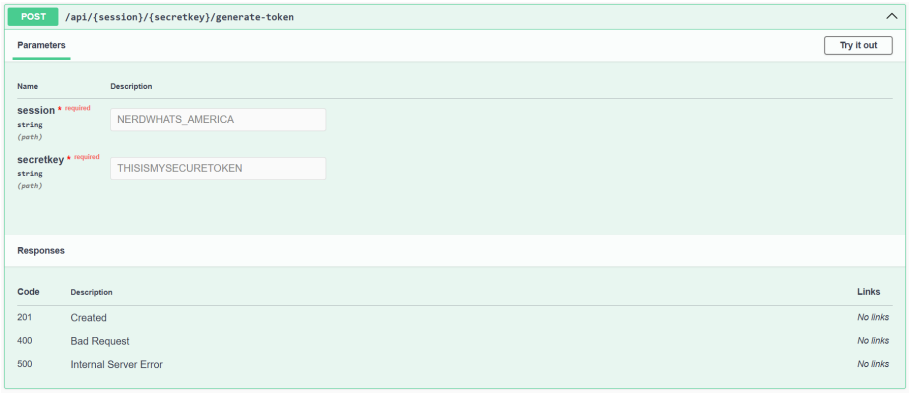

Figure 5.8: *Endpoint* para gerar *token* via *Swagger UI*.

É fundamental registrar corretamente o nome da sessão e a chave secreta, uma vez que essas informações serão utilizadas nas etapas posteriores de implementação do código. Esses dados permitem a comunicação direta com o servidor, possibilitando que requisições dos tipos *GET* e *POST* sejam devidamente autenticadas e executadas, assegurando o envio de mensagens e a execução de comandos por meio da *API*.

```
1 {
2    "status": "success",
3    "session": "NERDWHATS_AMERICA",
4    "token": <"Seu token">,
```

```
    "full": <"...">
}
```

Listing 5.2: Dados de autenticação disponibilizados pelo *response body*

Após habilitar os campos editáveis do `POST/api/{session}/{secretkey}/generate-token` por meio do botão ***Try it out***, e definir corretamente o nome da sessão e a chave secreta, ao clicar em ***execute***, o servidor retornará, no campo *response body*, o *token* de autenticação *JWT* correspondente à sessão que está prestes a ser iniciada.

Em seguida, ainda na interface do *Swagger*, deve-se clicar no botão ***Authorize*** para inserir o *token* gerado e autorizar a comunicação com o servidor por meio do *JWT*, garantindo o acesso autenticado às demais rotas da *API*. O procedimento detalhado pode ser visualizado na Figura 5.9.

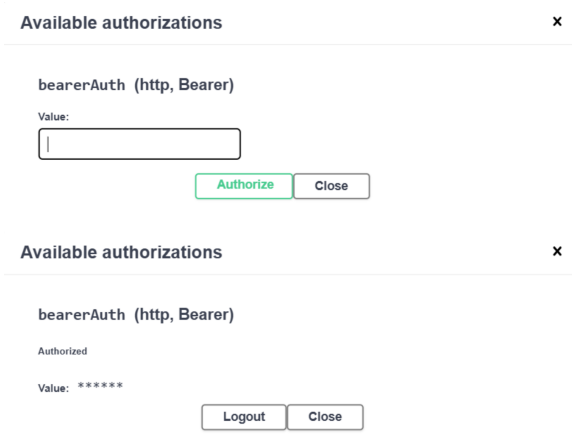

Figure 5.9: Janela de autorização do servidor via *JWT*.

Com o servidor devidamente autorizado por meio do *token JWT*, é possível iniciar uma sessão com maior confiabilidade e segurança em relação aos dados que serão instanciados ao longo da execução da aplicação.

Para iniciar uma sessão utilizando o *token* previamente autorizado, deve-se utilizar a requisição `POST/api/session/start-session`, disponível na interface do *Swagger*. Para esse processo, é necessário habilitar os campos editáveis por meio do botão ***Try it out***, assegurar que o nome da sessão coincida com o utilizado anteriormente, e, no campo ***request body***, atribuir o valor lógico `true` à propriedade ***`waitQrCode`***.

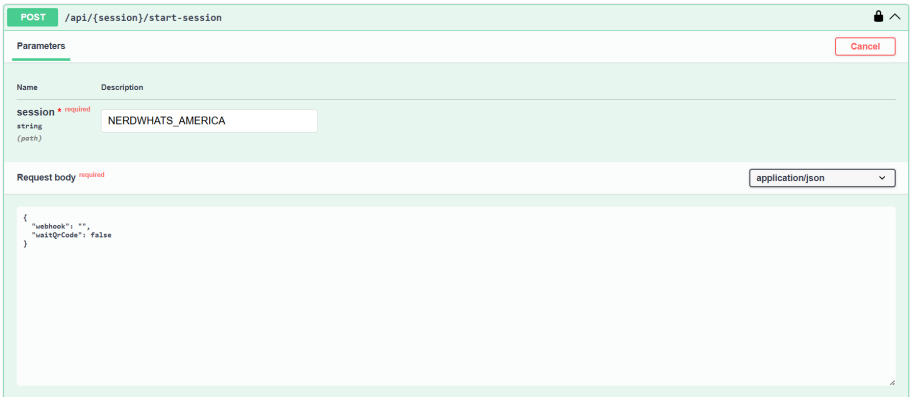

Figure 5.10: *Endpoint* para iniciar uma sessão.

Em seguida, ao executar a requisição, o próximo passo consiste na utilização do último *endpoint*, responsável por conectar o aplicativo *WhatsApp* ao servidor do *WPP-Connect Server*: `GET/api/session/qrcode-session`.

Não muito diferente dos demais *endpoints* trabalhados anteriormente, para este também é necessário clicar em ***Try it out***, definir o mesmo nome de sessão utilizado nos outros *endpoints* configurados previamente e, em seguida, acionar a opção ***execute***. Após essa execução, será gerado um ***QRCode***, o qual deve ser escaneado pelo aplicativo *WhatsApp* em um dispositivo móvel. Esse procedimento vincula a conta do usuário ao servidor do *WPPConnect*.

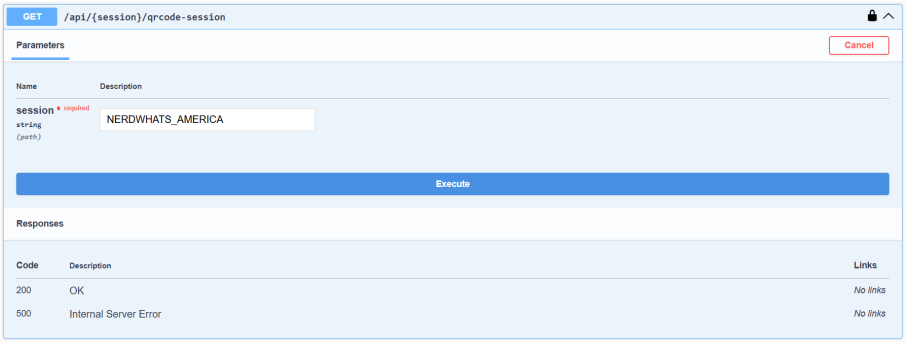

Figure 5.11: *Endpoint* para gerar o *QRCode* de vínculação.

5.3. Integração entre *Google Calendar* e *WhatsApp* para Recuperação de Eventos e Tarefas Diários.

5.3.1. Objetivo.

Neste tópico apresenta-se uma aplicação prática utilizando a *API* do *Google Calendar*, com o objetivo de recuperar, organizar e exibir os eventos agendados para o dia atual. Essa solução serve como base para sistemas de automação que visam facilitar a gestão de compromissos, notificações e produtividade pessoal ou corporativa.

5.3.2. Função de Carregar as Credenciais do *OAuth* 2.0

A aplicação é composta por um código em *Python* que utiliza bibliotecas amplamente conhecidas no ecossistema de desenvolvimento, disponibilizadas pelo arquivo ***requirements.txt***; faça o *download* e instalação dessas dependências de ambiente através do comando "`pip3 install -r requirements.txt`" dentro da pasta "*calendar-bot*".

Antes de tudo, é necessário importar bibliotecas essenciais para o funcionamento do processo. Utiliza-se módulos da biblioteca oficial do *Google*, como ***InstalledAppFlow***, que facilita o fluxo de autenticação para aplicações instaladas, e ***build***, para criar instâncias de serviços que acessam as *APIs*. A biblioteca ***pickle*** é utilizada para salvar e carregar o *token* de autenticação localmente, enquanto o pacote ***dotenv*** é responsável por carregar variáveis de ambiente definidas em um arquivo *.env*.

Listing 5.3: Importando módulos necessários para a função de carregar credenciais *Google*.

```
1 import os
2 import pickle
3 from google.auth.transport.requests import Request
```

```
from google_auth_oauthlib.flow import InstalledAppFlow
from googleapiclient.discovery import build
from dotenv import load_dotenv
load_dotenv()
]
```

Na sequência, é necessário definir os escopos de acesso, que determinam quais permissões a aplicação irá solicitar junto à conta *Google* do usuário. Os escopos funcionam como filtros de segurança, limitando rigorosamente quais recursos e tipos de dados a aplicação poderá acessar. Cada escopo está diretamente vinculado a uma funcionalidade específica, como acesso de leitura, configuração, edição ou fazer *download* de agenda.

Esses escopos são oficialmente documentados pelo *Google* e podem ser consultados no seguinte endereço https://developers.google.com/identity/protocols/oauth2/scopes?hl=pt-br. Nesse repositório oficial, encontra-se uma lista completa e constantemente atualizada dos escopos disponíveis para os diversos serviços da plataforma, como *Google Calendar*, *Tasks*, *Drive*, *Gmail*, entre outros.

Cada escopo possui uma descrição clara de sua função, permitindo que desenvolvedores escolham de forma precisa as permissões necessárias para suas aplicações. Isso garante não apenas o bom funcionamento do sistema, mas também reforça as práticas de segurança e privacidade, solicitando acesso apenas aos dados estritamente necessários para a finalidade proposta.

No contexto deste projeto, foram adotados escopos com permissões exclusivamente de leitura, tanto para o *Google Calendar* quanto para o *Google Tasks*. Essa escolha se justifica pelo fato de que a proposta da aplicação consiste apenas na consulta e exibição de informações, não havendo necessidade de criar, editar ou excluir eventos e tarefas.

Listing 5.4: Escopos necessários para permissionar a aplicação.

```
SCOPES = [
    "https://www.googleapis.com/auth/calendar.readonly",
    "https://www.googleapis.com/auth/tasks.readonly"
]
```

Em seguida, é implementada a função **carregarCredenciais()**, que tem como responsabilidade gerenciar todo o processo de autenticação e inicializar os serviços das *APIs*.

Listing 5.5: Função de carregar as credencias do *Google Calendar*.

```
def carregarCredenciais():
    creds = None
    token_path = 'token.pickle'
    client_secret_file = os.getenv("GOOGLE_CLIENT_SECRET", "
        credentials_oauth.json")
    ...
```

Listing 5.6: Verificação do *token*.

```
def carregarCredenciais():
    ...
    # Verifica se já existe um token salvo
```

```
    if os.path.exists(token_path):
        with open(token_path, 'rb') as token:
            creds = pickle.load(token)

    # Se não houver token válido, executa o fluxo de autenticação
    if not creds or not creds.valid:
        if creds and creds.expired and creds.refresh_token:
            creds.refresh(Request())
        else:
            flow = InstalledAppFlow.from_client_secrets_file(
                client_secret_file, SCOPES
            )
            creds = flow.run_local_server(port=9090,
               redirect_uri_trailing_slash=True)

        # Salva o token para reutilização futura
        with open(token_path, 'wb') as token:
            pickle.dump(creds, token)

    # Inicializa os serviços da API Calendar e Tasks
    service_calendar = build("calendar", "v3", credentials=creds)
    service_tasks = build("tasks", "v1", credentials=creds)
    return service_calendar, service_tasks
    ...
```

O processo descrito acima é essencial para qualquer integração com *APIs* do *Google*, garantindo que a aplicação acesse os dados de forma segura e autorizada. Além disso, o uso do arquivo **token.pickle** permite que o processo de autenticação não precise ser repetido toda vez que o *script* for executado, tornando-o mais eficiente.

5.3.3. Função Carregar Eventos.

O início do *script* traz as importações necessárias para o funcionamento da aplicação:

Listing 5.7: Importação de módulos necessários para carregar credenciais.

```
    from datetime import datetime, timedelta
    from dotenv import load_dotenv
    import os
    from utils import log
```

Cada uma dessas bibliotecas cumpre um papel específico. A ***datetime*** permite a manipulação de datas e horários, essencial para delimitar o intervalo de eventos a serem consultados. A biblioteca ***dotenv*** é usada para carregar variáveis de ambiente a partir de um arquivo externo *.env*, uma prática recomendada para manter dados sensíveis, como credenciais, fora do código-fonte. Já a biblioteca ***os*** permite acessar essas variáveis no ambiente de execução, enquanto o módulo ***log*** contém uma função auxiliar para registrar mensagens e erros.

Logo após as importações, o *script* carrega o ID do calendário utilizado na consulta à *API*. Esse ID é definido como uma variável de ambiente e carregado da seguinte forma:

Listing 5.8: Declarando variável de ID calendário.

```
    load_dotenv()
```

```
2     CALENDAR_ID = os.getenv("CALENDAR_ID")
```

A função principal do código chama-se carregarEventos e recebe como parâmetro o serviço autenticado da *API* do *Google Calendar*. Nela, é definido o intervalo de tempo correspondente ao dia atual, em formato *UTC* (*Coordinated Universal Time*), com base no horário do sistema. Isso garante que apenas os eventos do dia corrente sejam incluídos na busca:

Listing 5.9: Bloco de código que define intervalo de tempo ao dia atual.

```
1     def carregarEventos(service_calendar):
2     #Lista de mensagens
3     mensagens = []
4     try:
5         #Definindo intervalo de tempo com a biblioteca datetime
6         agora = datetime.utcnow()
7         inicio_do_dia = agora.replace(hour=0, minute=0, second=0,
              microsecond=0)
8         fim_do_dia = agora.replace(hour=23, minute=59, second=59,
              microsecond=999999)
9         ...
```

Com o intervalo definido, o próximo passo é fazer a requisição à *API*, informando o ***calendarId***, o intervalo de tempo e alguns parâmetros adicionais. A opção ***singleEvents=True*** garante que eventos recorrentes sejam tratados individualmente, e ***orderBy="startTime"*** organiza os resultados por horário:

Listing 5.10: Bloco de código

```
1     def carregarEventos(service_calendar):
2         ...
3         try:
4             ...
5             eventos_resultado = service_calendar.events().list(
6             calendarId=CALENDAR_ID,
7             timeMin=inicio_do_dia.isoformat() + 'Z',
8             timeMax=fim_do_dia.isoformat() + 'Z',
9             singleEvents=True,
10            orderBy="startTime"
11            ).execute()
12            ...
```

Os dados retornados pela *API* são armazenados em uma lista chamada eventos. Caso existam compromissos agendados, cada item é percorrido em um laço de repetição. A função extrai o horário de início e o título de cada evento e formata a mensagem em linguagem amigável, adequada para entrega ao usuário via interfaces como mensageiros ou assistentes virtuais.

Listing 5.11: Condicionais de existência para eventos

```
1     def carregarEventos(service_calendar):
2         ...
3         try:
4             ...
5             eventos = eventos_resultado.get('items', [])
```

```
            if eventos:
                mensagens.append("Eventos de hoje:")
                    for evento in eventos:
                        inicio = evento['start'].get('dateTime', evento
                            ['start'].get('date'))
                        titulo = evento.get('summary', 'Sem título')
                        mensagens.append(f"{inicio} - {titulo}")
            else:
                mensagens.append("Hoje você não tem eventos agendados."
                    )
            ...
```

Caso a agenda do dia esteja vazia, a função informa ao usuário com uma mensagem objetiva. Em situações de erro como falhas de autenticação, problemas de rede ou dados mal formatados o sistema captura a exceção, registra-a no terminal e retorna uma mensagem de falha controlada. Ao final da função, todas as mensagens são retornadas de forma unidas em um único texto, com quebras de linha:

Listing 5.12: Captura de exceção para controle de falhas

```
    def carregarEventos(service_calendar):
        ...
        try:
            ...
        except Exception as e:
            log(f"Erro ao carregar eventos: {e}")
            mensagens.append("Erro ao carregar eventos.")
        return "\n".join(mensagens)
```

5.3.4. Função Carregar Tarefas

A função apresentada a seguir tem como objetivo buscar e estruturar as tarefas presentes na conta do *Google Tasks*, organizando-as em uma mensagem formatada de forma clara e objetiva. Inicialmente, é realizada apenas a importação da função `log()`, responsável por exibir informações no *terminal*, especialmente em casos de erro durante a execução.

Listing 5.13: Importação da função log.

```
from utils import log
```

Em seguida, o código da função `carregarTarefas()` recebe como parâmetro o serviço da *API* do *Google Tasks* já autenticado. Dentro dela, inicia-se a criação de uma lista chamada `mensagens`, que armazenará os textos que compõem o retorno formatado da função.

Listing 5.14: Função de receber e criar a formatação das mensagens.

```
def carregarTarefas(service_tasks):
    mensagens = []
    tarefas_resultado = service_tasks.tasks().list(tasklist='@default')
        .execute()
    tarefas = tarefas_resultado.get('items', [])
```

A função também implementa um bloco de tratamento de exceções `try` e `except` para capturar e registrar possíveis erros na obtenção dos dados. Se ocorrer algum erro,

este é registrado no *terminal* através da função `log()` e uma mensagem de erro é retornada ao usuário.

Listing 5.15: Função de receber e criar a formatação das imagens.

```
def carregarTarefas(service_tasks):
    ...
    try:
        if tarefas:
            mensagens.append("\n*Tarefas de hoje:*")
            for tarefa in tarefas:
                titulo = tarefa.get('title', 'Sem título')
                mensagens.append(f"{titulo}")
        else:
            mensagens.append("Você não tem tarefas para hoje.")
    except Exception as e:
        log(f"Erro ao buscar tarefas: {e}")
        mensagens.append("Ocorreu um erro ao buscar as tarefas.")
    return "\n".join(mensagens)
```

Por fim, a lista `mensagens` é convertida em uma única *string*, separada por quebras de linha por meio do método `.join`.

5.3.5. Função de Enviar Mensagem

A seguir, o *script* apresentado tem como objetivo enviar mensagens via *WPPConnect*. Inicialmente, são importadas bibliotecas essenciais: ***requests*** para requisições *HTTP*, **os** para acesso a variáveis de ambiente e funções auxiliares importantes do sistema que são trabalhadas em modulos.

Listing 5.16: Importando módulos necessários para a função de enviar mensagem.

```
    import requests
    import os
    from utils import log

    #Funções auxilares importadas
    from credentials import carregar_credenciais
    from calendar_events import carregar_eventos
    from calendar_tasks import carregar_tarefas
```

Define-se variáveis que armazenam dados recuperados pela biblioteca **os**, sendo necessários para efetuar a chamada ao *endpoint*:

```
\textit{POST/api/{session}/send-message}
```

Listing 5.17: Definindo variáveis necessárias para a chamada de *endpoint*.

```
    # Nome da sessão do WPPConnect
    session = 'NERDWHATS_AMERICA'

    # Obter variáveis do ambiente
    CALENDAR_ID = os.getenv("CALENDAR_ID")
    WHATSAPP_NUMBER = os.getenv("WHATSAPP_NUMBER")
    WPP_API_SECRETKEY = os.getenv("WPP_API_SECRETKEY")
```

Após definir as variáveis necessários, criaremos uma função chamanda **enviarMensagem** que recebe um parâmetro chamado mensagem.

Listing 5.18: Bloco de código da requisição e postagem da mensagem.

```
    def enviarMensagem(mensagem):
        try:
            url = f'http://localhost:21465/api/{session}/send-message'
            headers = {
                'Authorization': f'Bearer {WPP_API_SECRETKEY}'
            }
            payload = {
                'phone': WHATSAPP_NUMBER,
                'message': mensagem
            }
            response = requests.post(url, json=payload, headers=headers
                )

    Restante do código nos próximos anexos...
```

No bloco ***try***, construímos a *URL* do *endpoint* utilizando o nome da sessão, declarada na variável *session* e o *endpoint* `/send-message`. Em seguida, é criado o cabeçalho de autenticação com o *token* pela linha `Authorization: bearer` e definido o corpo da requisição pelo *payload* com o número de telefone e a mensagem. A função *requests.post* é usada para enviar a requisão *HTTP* ao servidor *WPPConnect*.

Listing 5.19: Condicionais que verificam se a mensagem foi encaminhada.

```
    def enviarMensagem(mensagem):
        try:
            ...
            response = requests.post(url, json=payload, headers=headers
                )

            if response.status_code == 200:
                log("Mensagem enviada com sucesso!")
            else:
                log(f"Erro ao enviar mensagem: {response.text}")
        except Exception as e:
            log(f"Erro ao enviar mensagem: {e}")
```

Após o envio, a resposta é verificada. Caso o status seja 200 (sucesso), a mensagem de confirmação é registrada. Caso contrário, o erro retornado pelo servidor é exibido no *log*. Todavia, o bloco ***except*** captura exceções que possam ocorrer durante a execução, como falhas de conexão ou formatação incorreta da requisição, garantindo que erros sejam tratados de forma segura e registrada no sistema.

5.3.6. Função *Log*

O módulo auxiliar *utils.py* contém uma função fundamental para o diagnóstico e acompanhamento do sistema. A função ***log(msg)*** imprime mensagens no *terminal* precedidas pela data e hora, o que facilita a identificação de erros e o rastreamento do comportamento da aplicação ao longo do tempo. Seu funcionamento é simples, mas eficiente:

```
    from datetime import datetime

```

```
3     def log(msg):
4         print(f"[{datetime.now().strftime('%Y-%m-%d %H:%M:%S')}] {msg}"
              )
```

Esse recurso simples é útil na automação por permitir rastreamento de erros sem ferramentas externas, sendo leve, reutilizável e fácil de adaptar. A integração com o *Google Calendar* ilustra uma solução prática e eficiente, baseada em boas práticas como uso de variáveis de ambiente, tratamento de exceções e retorno claro ao usuário elementos que tornam o sistema automatizado confiável e aplicável em diversos contextos.

5.3.7. Função *Main*

O código a seguir representa o *script* principal responsável por executar uma tarefa agendada diariamente, integrando eventos de calendário e tarefas com o envio automatizado de mensagens via *WhatsApp*, utilizando o *WPPConnect Server*:

Antes de tudo, precisa-se importar bibliotecas como ***schedule*** e ***time*** responsáveis pelo agendamento e controle do tempo de execução. Além do mais, importa-se as funções modularizadas que, lidam com autenticação, coleta de dados e envio de mensagens, enquanto *log* exibe *logs* no *terminal*.

Listing 5.20: Importando módulos necessários para a formatação da mensagem.

```
1     import schedule
2     import time
3     from mensagem import enviar_mensagem, carregar_credenciais,
          carregar_eventos, carregar_tarefas
4     from utils import log
```

Posteriormente, trabalharemos apenas com uma função, **tarefaDiaria()**, que é responsável em realizar a ação principal do sistema: iniciar a autenticação com os serviços do *Google*, coleta os dados e formata a mensagem, que é exibida no *terminal* e enviada via *WhatsApp*.

Listing 5.21: Bloco de código para a função de formatação da mensagem.

```
1     def tarefaDiaria():
2         log("Executando tarefa diária...")
3         service_calendar, service_tasks = carregarCredenciais()
4         eventos = carregarEventos(service_calendar)
5         tarefas = carregarTarefas(service_tasks)
6         mensagem = eventos + "\n" + tarefas
7         log(mensagem)
8         enviarMensagem(mensagem)
```

Fora da função tarefaDiaria(), criamos uma estrutura condicional, afim de que garantir que o *script* seja executado diretamente. A tarefa processada pela função tarefaDiaria() é agendada para rodar todos os dias em um horário escolhido pelo usuário, parâmetros de tempo e tarefa passados para os métodos do módulo ***schedule***.

Listing 5.22: *Script* condicional que verifica a execução direta da mensagem.

```
1     if __name__ == "__main__":
2         schedule.every().day.at("10:33").do(tarefaDiaria)
3         log("Bot iniciado e aguardando o horário programado...")
4         while True:
```

```
5             schedule.run_pending()
6             time.sleep(30)
```

Em seguida, um *loop* infinito verifica se há tarefas pendentes para executar a cada 30 segundos, mantendo o *bot* ativo em segundo plano.

5.4. Segurança e Privacidade em *APIs*.

O uso de *APIs* em processos de automação e integração entre serviços impõe desafios importantes relacionados à segurança da informação e à privacidade dos dados. Esta seção aborda práticas recomendadas para o tratamento de dados sensíveis, autenticação segura, riscos relacionados ao uso de *APIs* não oficiais e ferramentas adequadas para testes.

Dados como nomes, senhas, CPF, localização e informações bancárias exigem atenção redobrada quando manipulados por sistemas automatizados. *Tokens* de autenticação, por exemplo, são recursos amplamente utilizados para acessar *APIs* de terceiros, como *Google Calendar* ou *WhatsApp*, e devem ser tratados como informações confidenciais. Seu vazamento pode resultar em acesso indevido, exposição de dados sensíveis ou comprometer a reputação da aplicação.

Para mitigar esses riscos, recomenda-se armazenar tokens e outras credenciais em arquivos *.env*, fora do controle de versão, utilizando bibliotecas como *dotenv* em *Python*. Além disso, é essencial garantir que esses arquivos estejam listados no *.gitignore*, prevenindo sua inclusão acidental em repositórios públicos. Um exemplo simples de carregamento seguro de variáveis de ambiente pode ser realizado com:

Listing 5.23: Instânciando módulo *dotenv*

```
1     from dotenv import load_dotenv
2     import os
3
4     load_dotenv()
5     API_TOKEN = os.getenv(<"Seu \textit{Token}">)
```

A **autenticação segura** é outro aspecto central. *APIs* mais robustas, como as do ecossistema *Google*, utilizam o protocolo *OAuth2*, que oferece uma arquitetura escalável e segura baseada em tokens de acesso temporários. O fluxo mais comum e seguro, conhecido como *Authorization Code Flow*, envolve a autorização explícita do usuário, emissão de um código de autorização e posterior troca por um *token* de acesso.

Tal abordagem elimina a necessidade de compartilhar senhas com o aplicativo e permite renovação de acesso por meio de *refresh tokens*. Ferramentas como o *Google OAuth 2.0 Playground* viabilizam o entendimento e testes desse processo de forma didática e segura.

Contudo, a **integração com APIs** nem sempre ocorre por vias oficiais. Em alguns projetos, como no caso abordado anteriormente, optou-se pela utilização da *WPPConnect Server* uma *API* não oficial para envio de mensagens via *WhatsApp*. Embora viabilize soluções automatizadas sem custos diretos, seu uso impõe riscos consideráveis. Dentre eles, destaca-se a possibilidade de violação dos termos de uso do *WhatsApp*, sujeitando o número utilizado a bloqueios.

Além disso, por serem mantidas por comunidades independentes, tais *APIs* não asseguram padrões consistentes de privacidade e segurança, e sua funcionalidade pode ser comprometida por alterações na estrutura do *WhatsApp Web*. Em razão disso, recomenda-

se que *APIs* não oficiais sejam utilizadas apenas em contextos experimentais, protótipos ou ambientes educativos.

Durante o **processo de desenvolvimento**, ferramentas como *Postman* e *Insomnia* são fortemente indicadas para testes seguros e organizados. Ambas permitem configurar variáveis de ambiente, simular autenticações com *Bearer Token*, inspecionar respostas e organizar coleções de chamadas *HTTP*.

Além disso, oferecem recursos de documentação e exportação de rotinas de teste, facilitando o trabalho colaborativo sem expor credenciais. Essas práticas tornam o processo de integração mais seguro, especialmente em equipes de desenvolvimento. Por fim, a Tabela 5.1 resume as recomendações fundamentais para segurança no uso de *APIs*:

Table 5.1: *Checklist* de Segurança em *APIs*

Recomendação	**Descrição**
Evitar versionamento de credenciais	Nunca incluir *tokens* ou senhas em repositórios.
Uso de `.env`	Armazenar chaves e variáveis fora do código-fonte.
HTTPS obrigatório	Garantir comunicação segura entre cliente e servidor.
Compreensão da autenticação	Entender se a *API* usa chave, *OAuth2* ou outro mecanismo.
Ferramentas de teste confiáveis	Preferir *Postman* e *Insomnia* para simulações.
APIs não oficiais apenas em protótipos	Usar com cautela, fora de ambientes de produção.

Adotar essas práticas é fundamental para garantir a integridade, a confidencialidade e a confiabilidade das aplicações que dependem de *APIs* externas. A segurança não deve ser tratada como etapa complementar, mas como parte integrante de todo o ciclo de desenvolvimento.

5.5. Considerações Finais

Este capítulo apresentou uma abordagem sistemática para a automação de tarefas a partir da integração entre a *API* do *Google Calendar* e o *WPPConnect Server*, demonstrando como a combinação dessas tecnologias viabiliza soluções eficientes para o gerenciamento de compromissos e a comunicação automatizada via *WhatsApp*. A aplicação proposta evidencia o potencial das arquiteturas baseadas em serviços de A *APIs* e o papel central das *APIs* na construção de sistemas distribuídos, escaláveis e interoperáveis.

A partir da explicação dos conceitos fundamentais de automação e das vantagens proporcionadas pela adoção de *APIs RESTful*, foi possível demonstrar o desenvolvimento de uma solução prática que automatiza a recuperação de eventos e tarefas agendados no *Google Calendar*, com envio automatizado de notificações via *WhatsApp*. Essa integração é particularmente relevante diante do cenário contemporâneo de hiperconectividade e necessidade crescente por soluções que otimizem processos e promovam a eficiência operacional.

O detalhamento das etapas de configuração do *Google Cloud*, da autenticação segura via *OAuth* 2.0 e da utilização do *WPPConnect Server* para o envio de mensagens permite compreender a complexidade técnica envolvida na construção de sistemas desse tipo. Ademais, evidencia a importância do domínio de boas práticas de desenvolvimento, como a modularização do código, o uso de variáveis de ambiente para proteção de credenciais e o tratamento adequado de exceções, fatores essenciais para garantir a confiabilidade e a manutenção da aplicação.

Ressalte-se que a escolha pelo *WPPConnect*, uma *API* não oficial para integração com o *WhatsApp*, impõe desafios adicionais relacionados à segurança, à privacidade e à conformidade com os termos de uso da plataforma. A utilização de *APIs* não oficiais deve ser criteriosamente avaliada, especialmente em ambientes de produção, em razão dos riscos inerentes à instabilidade, à ausência de suporte oficial e à possibilidade de bloqueio de contas. Nesse sentido, reforça-se a recomendação de seu uso em contextos acadêmicos, protótipos ou ambientes controlados.

Outro aspecto de destaque refere-se à segurança na manipulação de dados sensíveis, como *tokens* de autenticação e informações pessoais dos usuários. Foram discutidas práticas indispensáveis para a proteção desses dados, como o armazenamento seguro de variáveis de ambiente e a utilização de ferramentas confiáveis para testes de *APIs*, tais como *Postman* e *Insomnia*. Essas práticas são fundamentais para mitigar vulnerabilidades e assegurar a integridade e a confidencialidade das informações processadas pela aplicação.

A experiência descrita neste capítulo também aponta para oportunidades de evolução e aperfeiçoamento das soluções de automação. Entre as possibilidades futuras, destaca-se a integração com sistemas de inteligência artificial para o enriquecimento da análise e priorização de compromissos, bem como o desenvolvimento de assistentes virtuais capazes de interagir de forma proativa com os usuários. Tais avanços podem ampliar ainda mais os benefícios da automação, promovendo maior personalização e eficiência na gestão de tarefas e compromissos.

Vale destacar que todos os exemplos apresentados ao longo deste capítulo foram executados localmente (*localhost*), com o objetivo de facilitar o desenvolvimento e a depuração. Contudo, para que a aplicação seja executada de forma automática e sem interrupção, em especial para o envio diário de notificações via *WhatsApp*, torna-se essencial o seu uso em ambiente de nuvem como o *Google Cloud Run, AWS Lambda* ou outro provedor equivalente. Dessa forma, garante-se que o serviço esteja sempre ativo, sem depender de uma máquina local ligada todo o tempo, e possibilita escalabilidade, monitoramento e uma maior disponibilidade.

A integração entre o *Google Calendar* e o *WPPConnect Server* constitui um exemplo concreto do potencial transformador das *APIs* no contexto da automação de processos, demonstrando como a adoção de tecnologias bem estruturadas e a observância de boas práticas podem resultar em soluções funcionais e alinhadas às demandas atuais de produtividade e eficiência. A abordagem aqui apresentada contribui para a disseminação do conhecimento técnico sobre integrações de sistemas e reforça a importância de se considerar aspectos éticos, legais e de segurança na implementação de soluções baseadas em *APIs*.

Portanto, espera-se que os conceitos, metodologias e práticas descritos possam

orientar novos desenvolvimentos, pesquisas e aplicações no domínio da automação de tarefas, promovendo inovação e agregando valor em diferentes contextos, tanto pessoais quanto corporativos.

5.6. Referências Bibliográficas

BARRETO, Danilo Alves Paes et al. AUTOMAÇÃO E PROCESSOS ADMINISTRATIVOS: DESAFIOS E OPORTUNIDADES NA ERA DIGITAL. **Revista ft**, Revista ft Ltda, 2024.

BRANDÃO, Rodrigo. O futuro do trabalho: Entre a automação e a integração entre humanos e máquinas. **Revista Eletrônica Internacional de Economia Política da Informação da Comunicação e da Cultura**, v. 23, n. 3, p. 39–55, 2021.

EILAM, Eldad. **Reversing: Secrets of Reverse Engineering**. Indianapolis: Wiley, 2011.

FIELDING, Roy Thomas. **Architectural Styles and the Design of Network-based Software Architectures**. 2000. PhD thesis – University of California, Irvine. Disponível em: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm. Acesso em: 23 maio 2025.

GOOGLE DEVELOPERS. **Google Calendar API Overview**. [S.l.: s.n.], 2025. https://developers.google.com/calendar/api/guides/overview. Acesso em: 23 maio 2025.

MELO MESSIAS, Márcio de et al. O uso de ferramentas Google para a produção automatizada de relatórios da aprendizagem em turmas de séries iniciais para as escolas públicas do município de Fortaleza-CE. **Research, Society and Development**, v. 11, n. 17, e122111739151–e122111739151, 2022.

SANTOS, Elizabete. Gestão de processos: desafios e benefícios da implantação de uma plataforma de automação para a otimização de processos. Universidade Federal de Mato Grosso do Sul, 2023.

SOMMERVILLE, Ian. **Engenharia de Software**. 10. ed. São Paulo: Pearson Universidades, 2019. ISBN 9788543024974. Available from: <https://archive.org/details/sommerville-engenharia-de-software-10e>.

WPPCONNECT. **WPPConnect Server - API for WhatsApp integration**. [S.l.: s.n.], 2025. https://github.com/wppconnect-team/wppconnect-server. Acesso em: 23 maio 2025.