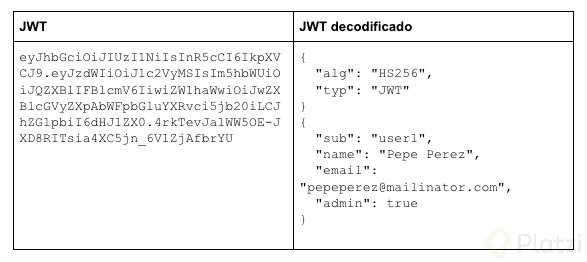
<https://platzi.com/clases/1440-ror/16662-autenticacion-con-auth0-y-jwt/>

## JWT

JWT o Json Web Token es un estándar definido por la el IETF (Internet Engineering Task Force) para transmitir digitalmente información relacionada a autenticación y autorización entre dos partes usando un token (que es simplemente un string) resultado de codificar un objeto JSON con la información que se desea transmitir. Según el estándar, este token fue pensado para transmitir “poca” información y adicionalmente para ser compatible con URLs lo que quiere decir que un JWT puede ser incluido en una petición HTTP o en un link. Esto facilita por ejemplo enviar links que dan acceso a un recurso.

Un JWT se ve así:



<https://jwt.io/#debugger-io?token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJ1c2VyMSIsIm5hbWUiOiJQZXBlIFBlcmV6IiwiZW1haWwiOiJwZXBlcGVyZXpAbWFpbGluYXRvci5jb20iLCJhZG1pbiI6dHJ1ZX0.4rkTevJa1WW5OE-JXD8RITsia4XC5jn_6V1ZjAfbrYU>

Un JWT está dividido en 3 partes separadas por un punto:

* Header (Encabezado): eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
* Payload (Contenido): eyJzdWIiOiJ1c2VyMSIsIm5hbWUiOiJQZXBlIFBlcmV6IiwiZW1haWwiOiJwZXBlcGVyZXpAbWFpbGluYXRvci5jb20iLCJhZG1pbiI6dHJ1ZX0
* Signature (Firma): 4rkTevJa1WW5OE-JXD8RITsia4XC5jn\_6V1ZjAfbrYU

El Header incluye información sobre cómo está construido el token. En el ejemplo anterior, el header dice “alg”: “HS256” haciendo referencia al algoritmo que se utiliza para generar la firma o signature.

El Payload incluye la información que se quiere transmitir. En el ejemplo se ve que se hace referencia a un usuario, su nombre, su email y su rol como admin.

Finalmente, la firma o signature, es la parte que se utiliza para verificar que la información contenida en el payload y en el header no ha sido modificada.

Los 2 tipos de token JWT más comunes son:

* tokens firmados pero no encriptados.
* tokens firmados y encriptados.

Ambos tipos proveen garantías diferentes. Por un lado, el contenido de los tokens firmados puede ser decodificado por cualquiera, pues no está encriptado. Sin embargo, al contener una firma, esta firma se utiliza por la parte que recibe el token para verificar que el contenido no ha sido modificado. Por otro lado, los token que están encriptados, no pueden ser decodificados a menos de que la parte que reciba el token tenga la llave para poder desencriptarlo.

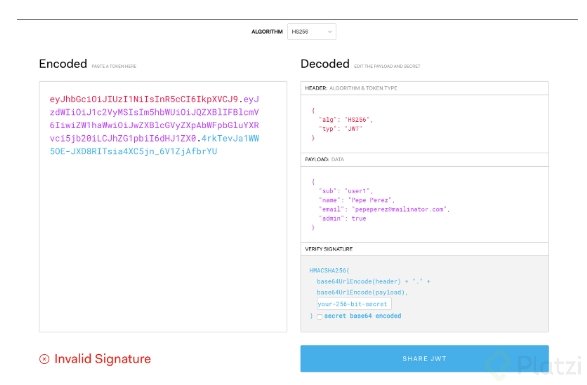
## Auth0 y [jwt.io](http://jwt.io/)

Auth0 es una empresa que provee una plataforma de autenticación y autorización para cualquier tipo de aplicación. Auth0 ha asumido JWT como el método principal para transmitir información de autenticación y autorización y han creado el sitio [jwt.io](http://jwt.io/) en donde podemos encontrar documentación sobre librerías en diferentes lenguajes para utilizar este estándar y adicionalmente se puede encontrar una herramienta para hacer debug de tokens. Por ejemplo, podemos hacer debug del token del ejemplo anterior usando la URL <https://jwt.io/#debugger-io?token=>

Por ejemplo:

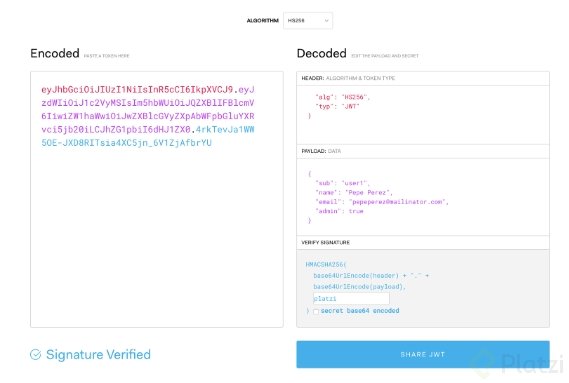
<https://jwt.io/#debugger-io?token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJ1c2VyMSIsIm5hbWUiOiJQZXBlIFBlcmV6IiwiZW1haWwiOiJwZXBlcGVyZXpAbWFpbGluYXRvci5jb20iLCJhZG1pbiI6dHJ1ZX0.4rkTevJa1WW5OE-JXD8RITsia4XC5jn_6V1ZjAfbrYU>

Ahí vamos a poder ver el contenido del token e inclusive crear un token desde cero pues el editor que nos aparece es interactivo y podemos editar su contenido.



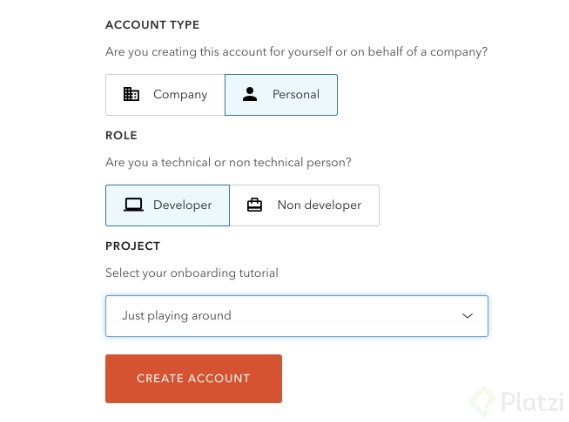
En un principio, vas a ver un mensaje diciendo que la firma no es válida. Esto sucede porque el token de ejemplo usa una firma que para ser verificada necesita de un secret. El secret debe ser conocido por la parte que genera el token para así generar la firma y por la parte que recibe el token para verificar la firma. Puedes pensar en la firma como si fuera un sello de una entidad oficial de gobierno. El sello solamente lo tiene la entidad oficial y así garantiza que los documentos que emita son oficiales. La parte que recibe el token, sabe cómo verificar la firma pues conoce cómo se debe ver el sello de esta entidad oficial.

En este caso, el secret es “platzi”. Una vez lo coloques en donde dice “your-256-bit-secret”, el sitio va a verificar que el token fue firmado con ese secret y va a desaparecer el mensaje de firma inválida.



## Configurar Auth0 para autenticación con Github

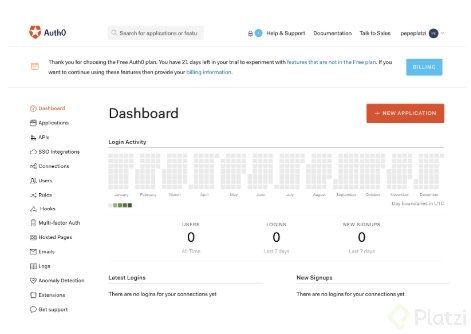
1. Abre una cuenta gratuita en [auth0.com](http://auth0.com/)



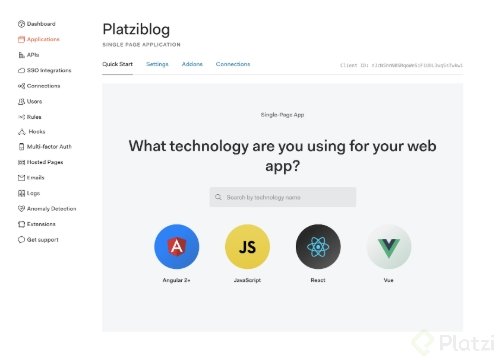
Te sugiero poner que es una cuenta personal, que eres desarrollador y que simplemente estás “playing around”.

1. Crear una aplicación:

Ve al dashboard en <https://manage.auth0.com/#/applications>, haz click en “New Application” y selecciona la opción “Single Page Web App”. Llama la aplicación PlatziBlog (o como quieras.)

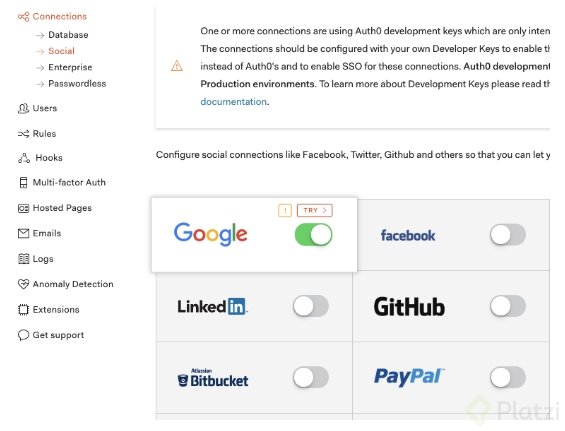


En nuestro caso no estamos implementando el frontend de nuestra aplicación así que puedes omitir el siguiente paso en el que te piden seleccionar un framework de Javascript.

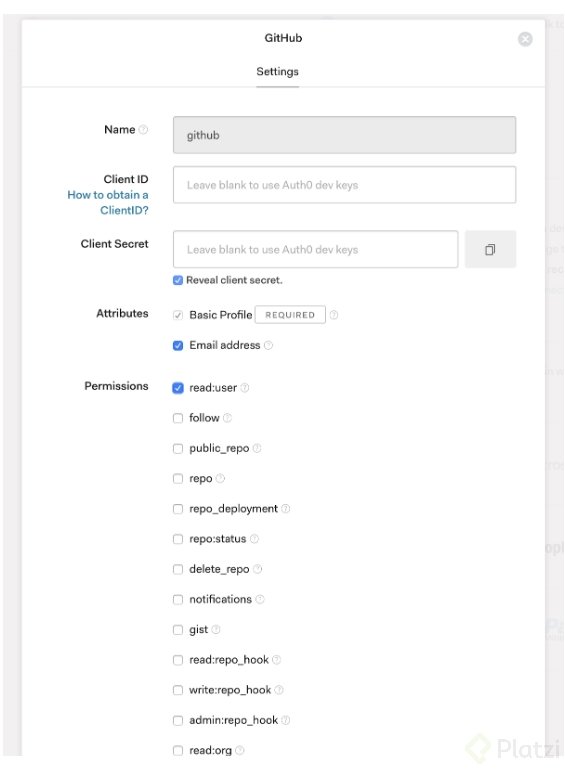


1. Ve a la pestaña “Settings” y copia el “Client ID” que vamos a utilizar más adelante. Luego, en el campo “Allowed Callback URLs” agrega [http://localhost:3000](http://localhost:3000/) y [https://jwt.io](https://jwt.io/) separados por coma. Asegúrate de incluir http y https respectivamente y de guardar los cambios.
2. Activar integración con Github:

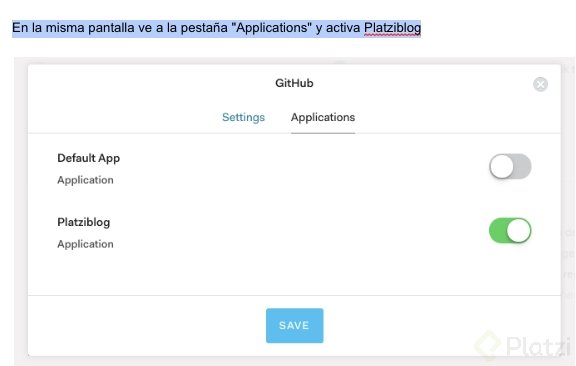
Ve a <https://manage.auth0.com/#/connections/social>. Allí, encuentras las alternativas para integrarte con proveedores de autenticación. Para este proyecto vamos a usar Github así que selecciona la opción Github.



Para hacer pruebas no es necesario que crees un client ID y un client secret. Por el momento puedes dejarlos en blanco y Auth0 va a utilizar su propio client ID y client secret de pruebas. Asegúrate de seleccionar “email address” y “read:user” para que podamos usar esta integración para poder obtener el email y el nombre del usuario.



En la misma pantalla ve a la pestaña “Applications” y activa PlatziBlog



Hasta este punto tienes configurada una aplicación the Auth0 que te permite hacer login con Github. Ahora debemos hacer la integración en la aplicación para poder obtener un JWT.

## Crear pantalla de login para obtener JWT

<https://github.com/simon0191/platzi-curso-rails-apis/commit/5bcb7c9438ce4784d402f40bc7455a263d67833d>

1. Crea un controlador llamado AuthController con el siguiente contenido:

**app/controllers/auth\_controller.rb**

**class** **AuthController** < ActionController::Base

**def** **login**

**end**

**end**

1. Crea una vista para el login:

**app/views/auth/login.html.erb**

<html>

<head>

<title>Platzi Blog - APItitle>

head>

<body>

<script src="https://cdn.auth0.com/js/auth0/9.5.1/auth0.min.js">script>

<script>

// Cuando la ventana carge ejecuta la funcion

window.addEventListener('load', **function**() {

// Inicializar integracion con auth0

**var** webAuth = **new** auth0.WebAuth({

// Apunta a tu dominio de auth0

domain: 'pepeplatzi.auth0.com',

// Usa el client id que encuentras en la configuracion de tu aplicacion en Auth0

clientID: '< TU CLIENT ID DE TU APP DE AUTH0>',

// Con esto especificamos la clase de token que queremos

// token para recibir un JWT y id\_token para que se incluya informacion del usuario

responseType: 'token id\_token',

// Con esto especificamos la informacion que deseamos incluir en el token. Para nuestro caso

// email y profile para poder obtener el nombre

scope: 'openid profile email',

// Esta es la URL a la que Auth0 va a redirigir despues de hacer login. Normalmente esto

// seria una URL al single page application de nuestra aplicacion. En nuestro caso vamos

// a usar jwt.io para ver el token y asi poder copiar y pegarlo en postman para poder

// probar el API.

redirectUri: 'https://jwt.io'

});

// Con esto se redireciona a la pagina de autenticacion de Auth0.

// En este caso estamos redirecionando inmediatamente. Normalmente, esto se ejcutaria despues

// de que el usuario haga click en un boton de login.

webAuth.authorize();

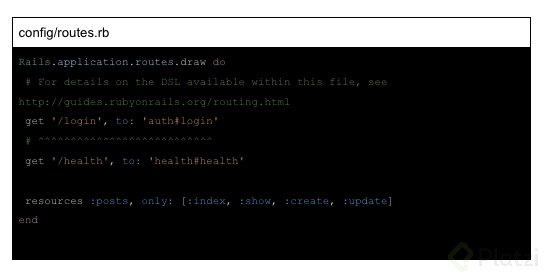
});

script>

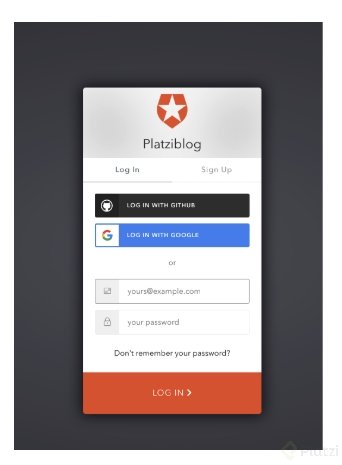
body>

html>

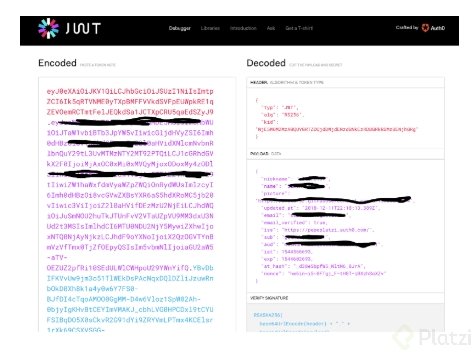
1. Finalmente agrega la ruta a routes.rb



1. Para probar la integración inicia el servidor de rails con rails s y ve a localhost:3000/login. Esto te va a redirigir a la pantalla de login de Auth0 que se debe ver parecido a esto:



Haz login con tu cuenta de GitHub y Auth0 te debe redirigir a [jwt.io](http://jwt.io/) tal como lo configuraste en donde vas a ver tu JWT que debe ser parecido a esto:



## Validar JWT y guardar usuario en caso de no existir

Ahora que podemos generar un JWT con Auth0 debemos validarlo en el backend del API

<https://github.com/simon0191/platzi-curso-rails-apis/commit/f6aa00aecb7c4b5ec423d970531cb5e155909d77>

* Incluye la gema jwt en el Gemfile y ejecuta bundle install
* Crea el archivo lib/json\_web\_token.rb

**app/lib/json\_web\_token.rb**

require '**net**/http'

require 'uri'

**class** JsonWebToken

# Metodo **que** va a verificar el **token** **que** le pasemos y **en** caso **de** ser valido va a retornar el

# contenido del mismo. **En** caso **de** ser invalido va a aroojar una excepcion.

def self.verify(**token**)

JWT.**decode**(**token**, nil,

# Muy importante! Verify the signature of this **token**

true,

# El algortimo usado para **la** firma. RS256 usa una llave privada para generar **la** firma

# Esta llave privada esta custodiada por Auth0 asi **que** **no** debemos preocuparnos por esto.

# Para validar **la** firma **se** utiliza una llave publica **que** es literalmente publica y **se**

# puede encontrar **en** el caso **de** este ejemplo **en** https://pepeplatzi.auth0.com/.well-known/jwks.json.

# Para ver **la** llave publica **de** tu dominio Auth0 debes reemplazar "pepeplatzi" con tu

# usuario **de** Auth0. Cualquiera puede ver esta llave asi **que** es seguro publicarla y compartirla.

# El objetivo es permitirle a otras partes verificar firmas generadas por tu aplicacion.

algorithm: 'RS256',

# Quien emite este **token**. Usa tu dominio **de** Auth0

iss: "https://pepeplatzi.auth0.com/",

# Verificar **que** el **token** fue emitido por lo **que** pusimos **en** "iss"

verify\_iss: true,

# Para quien fue emitido este **token**. Aqui debes colocar tu Client ID **de** Auth0

aud: "",

# Verificar **que** el **token** fue emitido para lo **que** pusimos **en** "aud"

verify\_aud: true) **do** |header|

# Dentro **de** este bloque **se** especifica como obtener **la** llave publica para verificar **la** firma

# **de** este **token**. **En** este caso, delegamos esa tarea al metodo jwks\_hash

jwks\_hash[header['kid']]

end

end

def self.jwks\_hash

# Obtenemos **la** llave publica del dominio **de** Auth0

jwks\_raw = **Net**::HTTP.get URI("https://pepeplatzi.auth0.com/.well-known/jwks.json")

# Decodificamos **la** llave publica y **la** retornamos

jwks\_keys = Array(JSON.**parse**(jwks\_raw)['keys'])

Hash[

jwks\_keys

.map **do** |k|

[

k['kid'],

OpenSSL::X509::Certificate.new(

Base64.decode64(k['x5c'].first)

).public\_key

]

end

]

end

end

1. Modificar app/controllers/concerns/secured.rb

\*\*app/controllers/concerns/secured.rb \*\*

**module** **Secured**

**def** **authenticate\_user!**

# Obtener el current user del metodo user\_from\_token y retornar en caso de que la operacion sea

# exitosa

**if**(Current.user = user\_from\_token)

**return**

**end**

# En caso de que no se obtenga un usuario, retornar 401

render json: {error: 'Unauthorized'}, status: :unauthorized

**rescue** JWT::VerificationError, JWT::DecodeError

# En caso de que haya un error de validacion del token, retornar 401

render json: {error: 'Unauthorized'}, status: :unauthorized

**end**

**def** **user\_from\_token**

# Obtener el token del header Authorization

token = get\_token\_from\_auth\_header

# Utilizar JsonWebToken para verificar y validar el token

payload = JsonWebToken.verify(token).first.with\_indifferent\_access

**if** payload.present?

# Si existe un token buscamos un usuario con el email contenido en el token.

# Si todavia no existe un usuario con este email, se crea usando el metodo `find\_or\_create\_by`

# de ActiveRecord.

User.find\_or\_create\_by(email: payload[:email]) **do** |user|

user.name = payload[:name]

**end**

**end**

**end**

**def** **get\_token\_from\_auth\_header**

# IMPORTANTE! cambiar el regex que teniamos antes. Como JWT incluye puntos, el regex anterior

# no tenia encuenta puntos.

token\_regex = /Bearer (.+)/

# leer HEADER de auth

headers = request.headers

**if** headers['Authorization'].present? && headers['Authorization'].match(token\_regex)

headers['Authorization'].match(token\_regex)[1]

**end**

**end**

**end**

1. Hacer que rails cargue el contenido de la carpeta “app/lib/”

**config/application.rb**

require\_relative 'boot'

**require** "rails"

# requires ...

**module** **Blogapi**

**class** **Application** < Rails::Application

# otras cosas ...

config.eager\_load\_paths << Rails.root.join('app/lib')

# ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

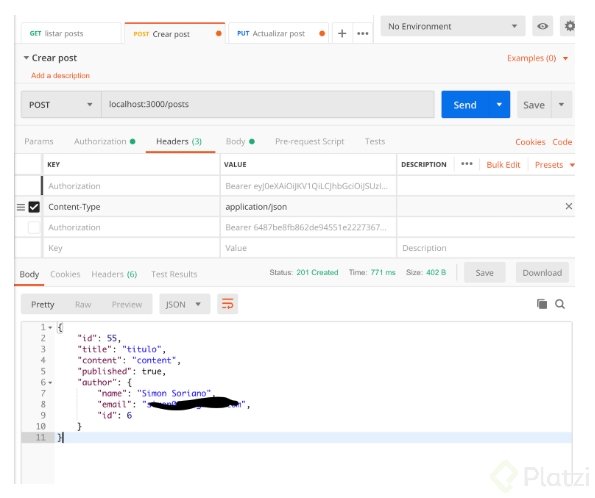
**end**

**end**

1. Opcionalmente actualizar el modelo User y sus pruebas para remover la autenticación que implementamos previamente

## Probando con Postman

1. Inicia el servidor con rails s.
2. Ve a localhost:3000/login, haz login y copia el token que aparece el [jwt.io](http://jwt.io/).
3. Abre postman y usando los mismos requests que creamos previamente modifica el header “Authorization” para que quede con el formato “Bearer ”
4. Ejecuta el request y debes ver algo parecido a esto.



Felicitaciones! Ya tienes un API que hace autenticación con un JWT creado por Auth0. Ahora solo falta pulir un par de detalles.

## Arreglar los tests

<https://github.com/simon0191/platzi-curso-rails-apis/commit/cfad808cc1b37074370c4eace2bf9a6bf3490bae>

Si removiste la generación de tokens inicial, las pruebas deben estar fallando. Debemos modificarlas para hacer que utilicen la librería JsonWebToken que implementamos.  
En nuestro caso debemos hacer un mock de JsonWebToken y hacer que no se ejecute la lógica de validación del token sino que se retorne siempre los datos del usuario del que estamos interesados en la prueba. Para esto usamos las funcionalidades de mocking de RSpec.

**mocks en RSpec**

**before** { **allow**(JsonWebToken).to **receive**(:verify).and\_return([{email: user.email}]) }

Con esta línea estamos instruyendo a RSpec para que intercepte todos los llamados al método “verify” de la clase JsonWebToken y retorne “[{email: user.email}]”.

Una vez hecho esto en las pruebas necesarias ejecuta bundle exec rspec y todas las pruebas deben pasar.

## Usar Secrets de rails 5

Finalmente vamos a utilizar una funcionalidad nueva de rails 5 para incluir secretos o información sensible en el código de nuestra aplicación sin necesidad de exponerlos en el repositorio.

En versiones anteriores de rails, se solía incluir un archivo llamado secrets.rb en donde se incluían cosas como API keys y contraseñas que se necesitaban para integración con otros servicios. Estaba a discreción del equipo de desarrollo omitir este archivo del repositorio de código para que no todos los desarrolladores tuvieran acceso a esta información. Para solucionar esto normalmente se compartian estos secrets usando manejadores de contraseñas como LastPassword o simplemente se compartia en un archivo de google drive. En cualquier caso, no era práctico porque los desarrolladores debían actualizar este archivo manualmente en su entorno de desarrollo.

En rails 5 se introdujo la posibilidad de incluir un archivo encriptado con todos los secrets para que así, esta información sensible pueda ser parte del repositorio de manera segura. Para ello se usa el comando “rails credentials:edit” que abre un editor de texto con un archivo en donde puedes incluir los secrets en formato YAML. Opcionalmente se puede especificar el editor de código que prefieras. Por ejemplo:

* Para usar VScode: EDITOR="code --wait" rails credentials:edit
* Para usar Sublime: EDITOR="subl --wait" rails credentials:edit
* Para usar vim: EDITOR="vim" rails credentials:edit

Este comando va a crear una llave de encripción en el archivo config/master.key y un archivo temporal que vas a poder editar y guardar y que se va a abrir con el editor que especificaste. Una vez guardes el archivo y cierres el editor, rails va a encriptarlo usando master.key en el archivo config/credentials.yml.enc. El archivo master.key debe ser tratado con cautela y no debe ser incluido en el repositorio. Si usas git, rails por defecto genera el archivo gitignore necesario para ignorar este archivo. Sin embargo debes guardar esta llave en un manejador de contraseñas u otro servicio de almacenamiento seguro en donde puedas compartirlo con tu equipo. Si pierdes este archivo no vas a poder desencriptar el contenidod e credentials.yml.enc. Por otro lado, el archivo credentials.yml.enc puede ser incluido en el repositorio pues está encriptado. Para que la aplicación lo pueda desencriptar, debe estar presente la llave en config/master.key o en la variable de entorno RAILS\_MASTER\_KEY. Por ejemplo:

RAILS\_MASTER\_KEY=xxxxxxx rails server

De esta manera puedes hacer uso de esta funcionalidad en servicios de despliegue como Heroku sin necesidad de incluir la llave en el repositorio.

Ahora, en nuestro caso la información que podemos incluir en este archivo es la información relacionada a Auth0: client\_id y auth0\_domain.

Ejecuta EDITOR=“code --wait” rails credentials:edit y agrega esta información:

**credentials.yaml**

# Otras cosas ...

auth0:

domain: .auth0.com

client\_id:

Guarda y cierra el archivo.

Ahora sigue los cambios en <https://github.com/simon0191/platzi-curso-rails-apis/commit/68e421bb7eb6fd0b11836523a22fa11520d96f6d> ignorando los cambios a config/credentials.yml.enc