IIC3103 - Taller de Integración
Departamento Ciencia de la Computación
Escuela de Ingeniería
Pontificia Universidad Católica

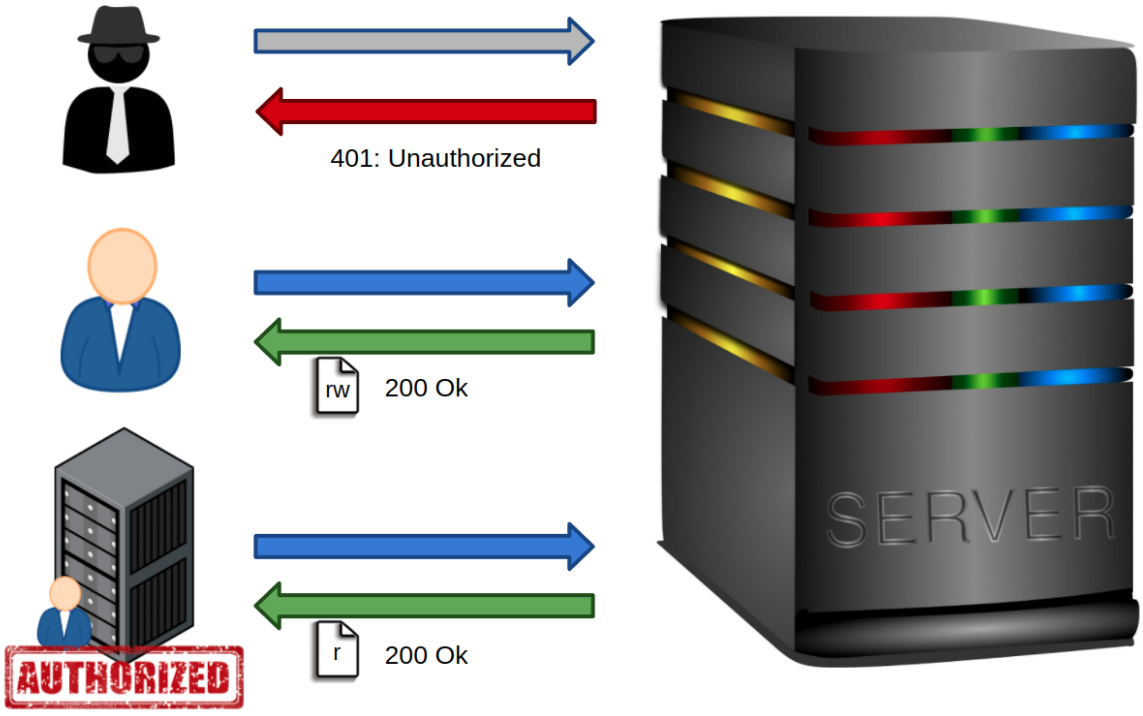
Enunciado **Tarea 1**

Objetivo

El objetivo de esta tarea es implementar un flujo de autenticación entre aplicaciones.

Trabajo a realizar

Deben exponer una API que permita **crear** usuarios, **obtener, editar y eliminarlos** usando un método de autenticación para usuarios, y permitir que otras aplicaciones obtengan sus datos usando un método de autenticación para aplicaciones. Estas acciones se llevarán a cabo a través de distintos requests HTTP a su API.



Índice

Descripción general

Para la realización de esta tarea deben implementar una aplicación que cumpla con el comportamiento esperado, la cual debe contar con una base de datos en la que deben tener los modelos suficientes para representar a los Usuarios, Aplicaciones y el estado o validez de los Tokens y solicitudes de autorización. No hay restricciones sobre la implementación de estos modelos, no obstante en el anexo 1 se adjunta un modelamiento sugerido.

Su servicio debe exponer una API REST que permita interactuar con los elementos en su base de datos según lo descrito a continuación.

Descripción detallada

Crear Usuarios

Los atributos[1] de los usuarios y el *scope* o permiso necesario para acceder a estos datos se muestran en la tabla.

Attributes	Scopes
username: str * **name**: str * **age**: int *	**basic**
psu_score: int **university**: str **gpa_score**: float	**education**
job: str **salary**: float **promotion**: bool	**work**
hospital: str **operations**: List[str] **medical_debt**: float	**medical**

Al crear un usuario también debe proporcionarse una contraseña (**password***) y crearse un identificador único para ese usuario (**user_id***).[2]

[1] *requerido
[2] No usar **username** como identificador ya que ese campo será modificable.

La creación de un usuario se hará mediante un **request HTTP** al endpoint **/users**, usando el método **POST** y entregando los datos del usuario dentro del **body en formato json**[3].

Si el usuario no existe anteriormente (username no existe), y los atributos proporcionados son válidos, deben crear el usuario en DB, y deben retornar una respuesta con **status HTTP 201 Created**, y dentro del **body** el user_id y token de usuario de la siguiente forma:

```
{
  "id": <user_id>
  "token": <token>
}
```

En caso contrario:

- Usuario ya existe: **409 Conflict, {"error": "user already exists"}**
- Atributos inválidos (requeridos, tipos): **400 Bad request, {"error": "invalid attributes"}**

Token de Usuario

Llamaremos al token anterior **Token de Usuario**. Este token permitirá acceso de **lectura y escritura** de cualquier campo del Usuario, incluso su contraseña 😱.

Por simplicidad, este token tendrá **validez ilimitada**, es decir:

- No tendrán fecha de expiración.
- No se invalida al editar el usuario.
- **Solo se invalidará cuando se elimine al usuario.**

El token se incluirá en los headers de los siguientes requests[4], de la forma:

"Authorization": <token>

Y deberá cumplir con ciertas condiciones de seguridad.

Obtener Usuarios

para obtener usuarios se hará un request GET al endpoint **/users/<user_id>.**

- Si el usuario existe, retornar **200 Ok, {**user}**

[3] En esta tarea siempre trabajaremos con body en formato json.
[4] Su API debe validar este token en los endpoints que requieran autenticación

Editar Usuarios

Las ediciones de uno o varios atributos de un usuario se hará con un **request HTTP** con el método **PATCH** al endpoint **/users/<user_id>.**

Al editar uno o varios atributos se debe validar que todos los inputs sean correctos (del tipo esperado), en cuyo caso debe retornar **200 Ok, {**user}**.

De lo contrario, debe retornar **400 Bad request, {“error”: “invalid update”}**

Si se está editando el campo username, también se debe validar que no esté siendo usado por otro usuario. Si ya está ocupado, debe retornar **409 Conflict, {“error”: “username is taken”}**

Eliminar Usuarios

Para eliminar usuarios se hará un request **DELETE** al endpoint **/users/<user_id>.**

- Si el usuario existe, eliminar y retornar **204 No Content**

Prohibido y no autorizado

Si en cualquiera de los requests anteriores el token de usuario es válido, pero se está intentando acceder a un recurso de otro usuario, debe retornar **403 Forbidden, {“error”: “you do not have access to this resource”}.**

Si el token fuera inválido, retornar **401 Unauthorized, {“error”: “invalid token”}**

Notar que usando **tokens de usuario** nunca vamos a poder consultar por un usuario que no existe, ya que antes nos encontraremos con el error **401 Unauthorized.** (Ya que el token de usuario se invalida al eliminar el usuario).

Autenticación de otras aplicaciones

Otras aplicaciones van a querer consultar los datos de los Usuarios de tu API 🤯, por lo que tendrás que implementar un **flujo de OAuth2**.

Para esto necesitaremos un endpoint para solicitar la autenticación: **/oauth/request**, y un endpoint para obtener la autenticación: **/oauth/grant**

Cuyo comportamiento (simplificado) se detalla en los siguientes apartados.

Solicitar Autenticación

El endpoint **/oauth/request** debe recibir como *query parameter* el id del usuario a cuya información se quiere acceder, los *scopes* requeridos para acceder a la información a la que se quiere acceder, y el nombre de la aplicación que está intentando acceder a esos datos. De esta forma el request de solicitud de autenticación será de la forma:

GET /oauth/request?user_id=<user_id>&scopes=<scopes>&app_id=<app_id>

Donde **<user_id>** es el id del usuario, **<scopes>** es la lista de *scopes* requeridos separados por coma y **<app_id>** es el identificador de la app, ej:

/oauth/request?user_id=123&scopes=basic,work&app_id=spotify

Frente a este request el endpoint debe responder con status **202 Accepted** y un body con la url para obtener la autorización solicitada, incluyendo en la url como *query parameters*:

- id de usuario
- scopes solicitados
- Identificador de la app
- *nonce* (string random con letras y números de 20 carácteres)[5]

Además de la url, dentro del body debe indicar la fecha de vencimiento de la solicitud en formato unix timestamp (la solicitudes tendrán validez durante 10 segundos), y un mensaje informativo.

De esta forma, una respuesta de ejemplo sería:

```
{
  “message”: “spotify está intentando acceder a basic,work, ¿desea continuar?”
  “grant_url”: “/oauth/grant?user_id=1&scopes=basic,work&app_id=spotify&nonce=1w2...1y38”,
  “expiration”: 1647490627
}
```

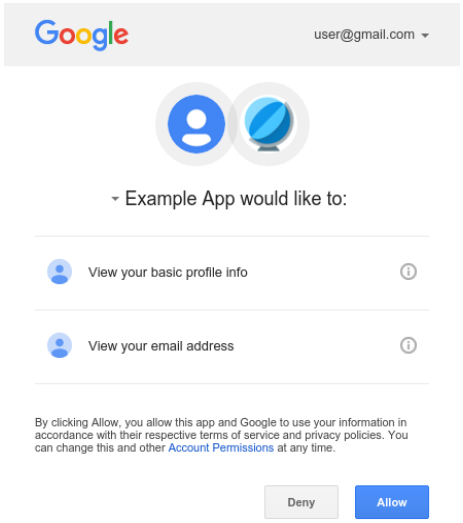
En un flujo de autenticación real, la app externa también debería indicar una **url de redirección** (tipo callback), y frente al primer request tu app debería redirigir al usuario a una ventana de ella misma (donde el usuario ya está loggeado o puede hacer login seguro), donde se le informa que es lo que se está autorizando, y finalmente se autoriza usando su token de usuario.

Luego tu app le entregaría el token de acceso a la app externa a través del callback.

[5] El nonce se usa para validar que el grant venga del mismo request iniciador de la solicitud

Conceder Autenticación

En nuestro flujo simplificado nosotros actuaremos como el usuario autorizando a la app, cómo si fuera un usuario apretando el botón **Permitir** de la ventana. Esto lo haremos haciendo un GET a la grant_url, con el token del usuario en el header.



¡Como si el usuario estuviera aceptando la solicitud loggeado desde tu app! 😲

Por su lado, ustedes deberán validar los datos (token de usuario correcto, mismo *nonce*, no ha expirado), y en caso de éxito[6] retornar **200 Ok**, y en el body un *access_token* y una fecha de expiración (tendrán validez durante 1 minuto):

```
{
   “access_token”: <access_token>
   “expiration”: 16475626781
}
```

Token de Acceso

Llamaremos al token anterior **Token de Acceso**. Este token permitirá acceso de lectura solamente de los campos cuyos scopes que fueron indicados al solicitar el token.

Este token tendrá **validez limitada** y se invalidará cuando se sobrepase la **fecha de expiración** o cuando **se elimine el usuario asociado a ese token**.

Al igual que el token de usuario, este token se incluirá en los headers de los requests que requieran autenticación.

[6] Los detalles de las posibles respuestas de error serán descritos en la documentación.

Obtener datos

Finalmente se deben disponer los siguientes endpoints para obtener los datos de cada scope haciendo un request tipo **GET** al endpoint correspondiente:

- **/users/<user_id>/basic**
- **/users/<user_id>/education**
- **/users/<user_id>/work**
- **/users/<user_id>/medical**

Naturalmente, sólo se podrá acceder a estos si se tiene un token con los *scopes* correspondientes, de lo contrario su servicio responderá con **401 Unauthorized** cuando el token es derechamente inválido o bien **403 Forbidden** cuando el token sea válido pero el solicitante no es el propietario (o no tiene permisos) sobre ese recurso.

Seguridad Tokens

Tanto los **Tokens de Usuario** como los **Tokens de Acceso** deben cumplir con las siguientes restricciones de seguridad:

- Ser generado aleatoriamente (no pueden ser todos iguales).
- Contener números, letras minúsculas y mayúsculas.
- Tener un largo de al menos 30 caracteres.

Ejemplos:

- “Dm4hFcPw9fb7q2pGn5utPNgZ6f7HUj” ✅
- “JQYCnF6XTYna6s6RwJy58YT6d3he897y8awdh8vpJV8f” ✅
- “1111111272727273889484459595998 74” ❌
- “Aaaaaaaaaaaaaaaaaaaaaa” ❌

Momento de apreciación

Tomen un minuto para apreciar lo importante que es este flujo y la forma en que mantiene nuestros datos seguros. Si ven para atrás, el único momento donde se comunicó la contraseña fue al momento de crear el usuario, y luego pudimos autorizar a otras aplicaciones sin tener que ingresar la contraseña en un sitio externo.

Esta fue una versión simplificada del flujo real, si quieren aprender más sobre el estándar **OAuth2.0** puede visitar su página oficial.

Resumen de endpoints a implementar

A continuación, se presentan los *endpoints* que su API debe disponer.

Crear Usuarios:

- **POST /users**: crea un usuario y retorna su id y token de usuario.

Obtener, Editar y Eliminar Usuarios:

- **GET /users/<user_id>**: retorna el usuario <user_id>.
- **PATCH /users/<user_id>**: edita los atributos del usuario <user_id>.
- **DELETE /users/<user_id>**: elimina al usuario <user_id>.
- **GET /users/<user_id>/<scope>**: retorna los datos del usuario <user_id> del scope <scope>.

Solicitar y Conceder Autorización:

- **GET /oauth/request**: Solicita autorización para acceder a cierta información de un usuario.
- **GET /oauth/grant**: Obtiene un token de acceso para que una aplicación externa pueda acceder a cierta información de un usuario.

Documentación endpoints

En la siguiente URL encontrará la documentación completa de la API a implementar, incluyendo *requests* y *responses*, y principales códigos de error:

> **Documentación API**
>
> La documentación de la API se encuentra publicada en el sitio https://tarea-1.2022-1.tallerdeintegracion.cl/

Arquitectura de la solución

La API podrá estar construida sobre un framework de desarrollo de API's o un framework MVC tal como Rails (Ruby), Django (Python), Sails (Nodejs) o cualquier otro framework que permita la generación de servicios REST. No hay requerimientos específicos sobre lenguajes o frameworks a utilizar.

La solución deberá tener una capa de almacenamiento de datos, con persistencia mínima de 1 hora, tiempo suficiente para soportar la ejecución de varios flujos en serie, esto quiere decir que se deberá implementar una base de datos u otro sistema de almacenamiento que permita ir almacenando los registros que se van creando en los flujos que permite la API.

Se recomienda hacer un deploy en Heroku, sistema de servidores Cloud que tiene una capa gratuita de servidor y base de datos con capacidades suficientes para el desarrollo de esta tarea.

Versionamiento del código

El sitio y todo su código fuente deberá estar versionado en un repositorio de Github Classroom proporcionado por el equipo docente.

Entregables

Cada alumno deberá entregar, mediante un formulario publicado en el sitio del curso, las siguientes *url's*:

- URL del endpoint que permite consumir la api
- URL del repositorio Github.

Adicionalmente, cada estudiante deberá inscribir su url en un formulario que se habilitará para testing de la API.

Para la tarea, cada alumno deberá crear un repositorio en Github Classrooms, en el siguiente link:

> **Creación repositorio Github Classroom**
>
> URL de Classroom: https://classroom.github.com/a/bio9OdlR
>
> Cada alumno debe linkear su usuario Github con su nombre de la lista.

Fecha de entrega

La tarea se deberá entregar el día 1 de abril antes de las 18:00 hrs.

Requisitos mínimos

Las tareas que no cumplan las siguientes condiciones no serán corregidas y serán evaluados con la nota mínima:

- La API deberá ser pública, accesible desde cualquier dispositivo conectado a internet.
- El código deberá estar versionado en su totalidad en un repositorio Git, disponible en el Github Classroom del curso.
- El sitio debe reflejar fielmente el código entregado en el repositorio
    - Se recomienda configurar un deploy automático en Heroku asociado al repositorio Github

Penalizaciones

Se descontarán 0,2 puntos de la nota de la tarea por cada hora de atraso en la entrega, contados a partir de la fecha estipulada en el punto anterior.

Cualquier intento de copia, plagio o acto deshonesto en el desarrollo de la tarea, será penalizado con nota 1,1 de acuerdo a la política de integridad académica del DCC.

Dudas

StackOverflow

Foro más popular para resolver dudas sobre código, *frameworks* y lenguajes de programación. Antes de preguntar, asegúrate que la pregunta no se ha realizado anteriormente.

> http://stackoverflow.com/

Resolución de dudas en Github

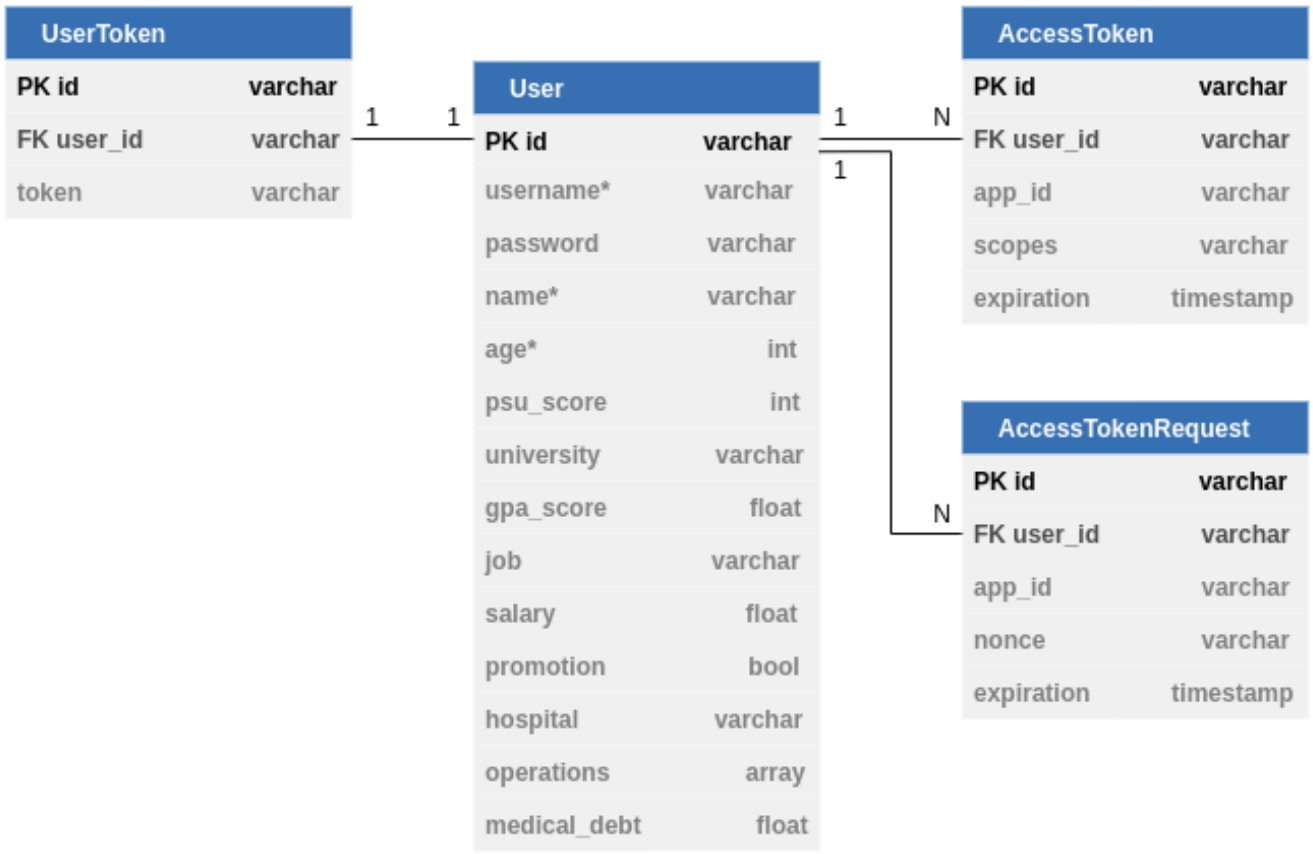
También se resolverán dudas mediante los *issues* del repositorio en github: https://github.com/IIC3103/2021-2-dudas/issues

Anexos

Modelamiento Sugerido

Existen muchas formas distintas de crear los modelos para cumplir con el comportamiento esperado, se puede hacer tanto con bases de datos relacionales como no relacionales.

A continuación se muestra un modelamiento relacional que será suficiente para cumplir con los objetivos de la tarea:



Como se ve en el diagrama, hay una relación 1 a 1 entre los Usuarios y sus tokens de usuario, ya que cada usuario puede tener 1 y solo 1 token de usuario y cada token de usuario pertenece únicamente a 1 usuario.

Por otro lado, cada usuario puede tener N solicitudes y tokens de acceso (muchas aplicaciones pueden hacer el flujo y en un momento dado muchas apps podrían tener acceso a la información de un usuario, pero cada solicitud y token de acceso en particular estará asociado a 1 y solo 1 usuario.

JSON Web Tokens

Tengan en cuenta que no es **estrictamente necesario** definir todas estas tablas y campos en la base de datos o seguir este modelamiento al pie de la letra.

De hecho, si usan JSON Web Tokens pueden guardar toda la información del token necesaria (tipo, scopes, fecha de expiración) dentro del mismo token. Básicamente es algo así:

Para generar el token se hace a partir de un json con toda la información necesaria:

```
{
   "user_id": 9873,
   "app_id": "youtube",
   "type": "access_token",
   "scopes": "basic,work,medical",
   "expiration": 16475626781
}
```

Luego el JSON se codifica usando un secreto que solo tu conoces (una llave ssh privada por ejemplo), generando un token parecido a esto:

“eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkIjo5ODczLCJhcHBfaWQiOiJ5b3V0dWJlIiwidHlwZSI6ImFjY2Vzc190b2tlbiIsInNjb3BlcyI6ImJhc2ljLHdvcmssbWVkaWNhbCIsImV4cGlyYXRpb24iOjE2NDc1NjI2NzgxfQ.5of0AFEyAONNdJCMhhmr7X_4ODmD6IgKuVBIOs_5Mcc”

Y eso☝ se usa como token. Si se fijan cumple con las condiciones de seguridad pedidas.

Quien recibe y guarda el token no tiene forma de acceder o modificar sus contenidos 🔒 (ya que no conoce la llave).

Luego cuando lo recibes en una solicitud lo puedes decodificar (usando la llave 🔑), y validar los datos. **Con esto te ahorras tener que guardar el estado de los tokens en DB** 🤩.

Existen librerías de JWT para todos los lenguajes ya que es un estándar moderno muy usado últimamente. Aquí se puede probar.

Recursos adicionales

A continuación, se presentan una serie de links que le podrán ser útiles para el desarrollo de la tarea.

	Ruby on Rails	Python Django	Node
Cursos y documentación	Instalar: https://gorails.com/setup/osx/10.15-catalina Documentación: https://guides.rubyonrails.org/getting_started.html Curso: https://www.codecademy.com/learn/learn-rails	https://docs.djangoproject.com/en/3.0/intro/tutorial01/ https://docs.docker.com/compose/django/[7] ***	Tutorial: https://itnext.io/a-new-and-better-mvc-pattern-for-node-express-478a95b09155
Despliegue de una app en Heroku	https://devcenter.heroku.com/articles/getting-started-with-rails5#store-your-app-in-git	https://devcenter.heroku.com/articles/django-app-configuration	https://devcenter.heroku.com/articles/deploying-nodejs

*** Este link es muy útil, pero está desactualizado en un detalle. En el paso 2 de “Connecting to the database”, es necesario agregar la siguiente línea: `'PASSWORD': 'postgres'`.

[7] Tutorial para crear un proyecto utilizando Django, Docker y Postgres.