UNIVERSIDAD DE BUENOS AIRES

FACULTAD DE INGENIERÍA

Año 2020 - 1er Cuatrimestre

Taller de Programación II (75.52)

ChoTuve

AuthServer

Documento de arquitectura / diseño

Fecha de entrega: 30/07/2020

Grupo 8

79979 – Gonzalez, Juan Manuel (juanmg0511@gmail.com)
82449 – Laghi, Guido (guido321@gmail.com)
86429 – Casal, Romina (casal.romina@gmail.com)
96453 – Ripari, Sebastian (sebastiandripari@gmail.com)
97839 – Daneri, Alejandro (alejandrodaneri07@gmail.com)

Contenido

1. Objetivo

El objetivo de este documento es presentar la arquitectura utilizada, así como también comentar los aspectos de diseño más importantes del AuthServer utilizado en la aplicación ChoTuve.

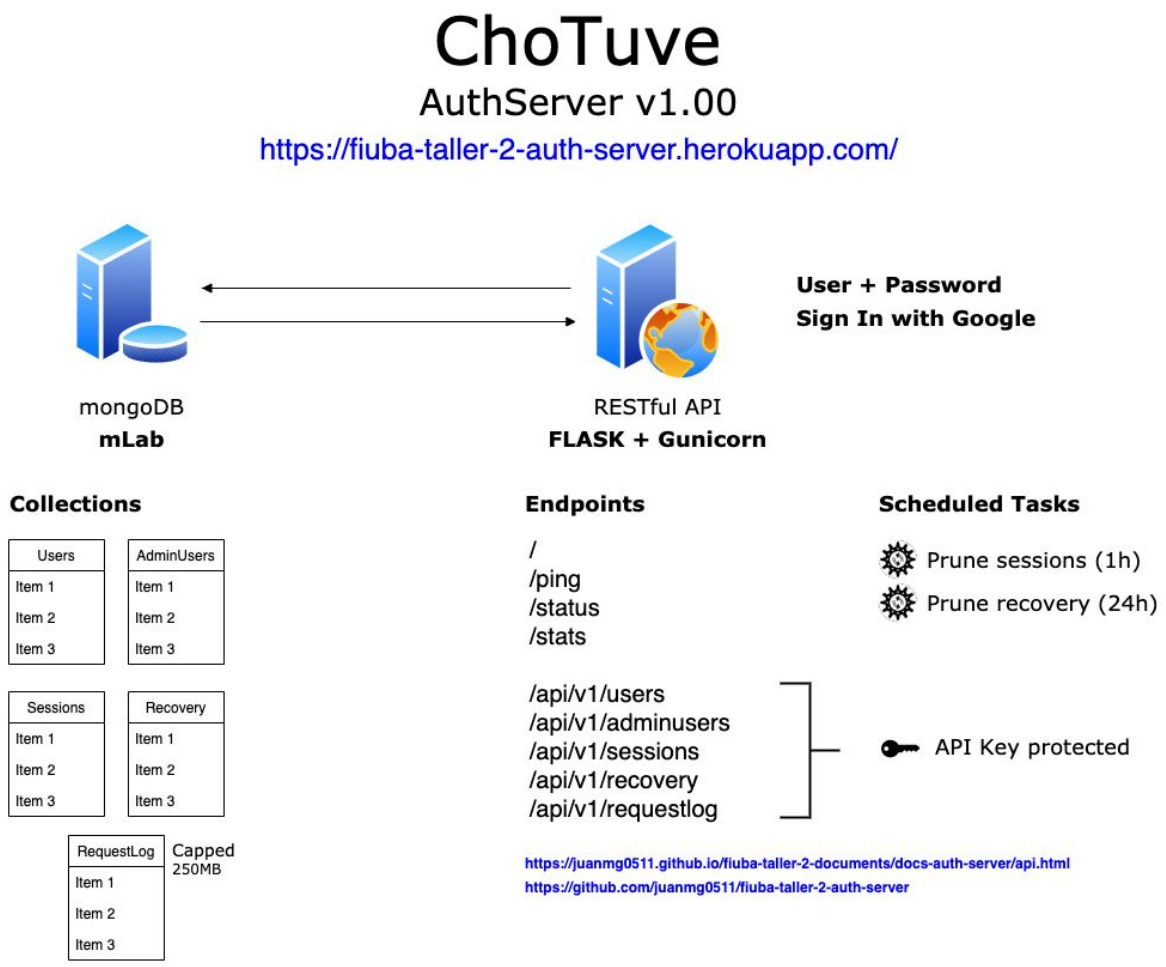
Adicionalmente, se incluye la especificación completa de la API provista por el servidor, en formato OpenAPI 3.0.

2. Arquitectura

En esta sección se describe la arquitectura del AuthServer, tomando como base el *deploy* en el ambiente productivo, hosteado en Heroku.

2.1 Consideraciones generales

A grandes rasgos, la arquitectura del AuthServer puede resumirse mediante el siguiente esquema:



Se cuenta con 2 *deploy* en la nube de Heroku, uno para staging y otro con fines productivos. Las URL de ambos son:

PRODUCCIÓN: https://fiuba-taller-2-auth-server-st.herokuapp.com/
STAGING: https://fiuba-taller-2-auth-server-st.herokuapp.com/

El AuthServer está compuesto por un front-end que consiste en una aplicación **Flask** corriendo sobre **Gunicorn**, proveyendo una serie de servicios de manejo de perfiles y autenticación, así como también como algunos adicionales. Si bien se ahondará sobre estas capacidades en la

sección de diseño de este documento, el gráfico sirve para tener un pantallazo del diseño del servidor.

La base de datos es del tipo Mongo DB, y se encuentra hosteada en mLab, mediante el uso del plugin que dicha organización tiene disponible en Heroku.

2.1.1 Configuración

El Auth server se diseñó para ser altamente configurable, a fin de poder realizar migraciones o pasajes entre ambientes en una forma sencilla. Basta con definir una serie de variables de entorno para setear todos los valores configurables. Asimismo, esto permite setear distintos valores para cada ambiente, lo que facilita tareas de desarrollo y *debug*.

Para evitar errores innecesarios, la mayoría de estos parámetros tienen definidos valores default en el código. Si se utiliza algunos de estos valores, como por ejemplo la **API Key**, el log mostrará un mensaje WARNING al momento de iniciar el servidor, a fin de alertar al administrador.

Puede consultarse el listado completo de configuraciones posibles en el manual del administrador incluido con el proyecto:

```
AuthServer.v1.00.Manual.Administrador.pdf
```

2.1.2 Continuous integration/deploy

En su despliegue en la nube, el proyecto está configurado con Continuous Integration y Continuous Deploy, mediante el uso de la herramienta **Travis CI**.

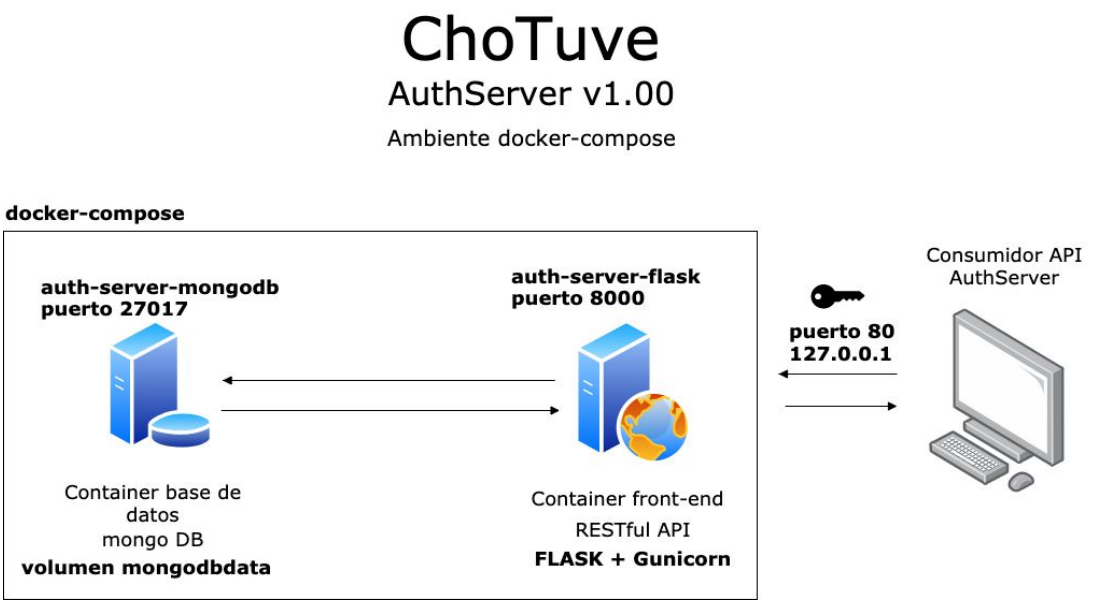
Dicha plataforma es la encargada de realizar la ejecución de pruebas, subir la imagen dockerizada del proyecto a docker.io y hacer el *deploy* a Heroku, para los dos ambientes disponibles, staging y producción.

Los resultados de las distintas corridas pueden consultarse en:

https://travis-ci.com/github/juanmg0511/fiuba-taller-2-auth-server

2.1.3 Ambiente de desarrollo

Para desarrollo local, se cuenta con un ambiente implementado mediante docker-compose, imitando la arquitectura descrita en esta sección. En forma esquemática puede representarse como se muestra a continuación:



Los pasos para instalar, configurar y ejecutar este ambiente pueden encontrarse en el manual del administrador:

`AuthServer.v1.00.Manual.Administrador.pdf`

2.2 Front end

El front end del AuthServer ejecuta *dockerizado,* utilizando en ambos ambientes 2 worker processes de Gunicorn, en el plan **Free** de Heroku. La cantidad de workers es parametrizable mediante una variable de entorno, y se configuró de acuerdo a lo recomendado por la documentación de la plataforma. Se puede consultar en:

https://devcenter.heroku.com/articles/python-gunicorn#basic-configuration

Para evitar que la aplicación “duerma” luego de 30 minutos de inactividad, se utiliza la aplicación Kaffeine, que pinguea el servidor en forma sincrónica, con una frecuencia de 1 hora:

https://kaffeine.herokuapp.com

A fin de cumplir con el requisito de que la aplicación duerma 6hs por día, se configuró en dicha aplicación que ese período ocurra durante la madrugada.

En términos de recursos, el AuthServer cuenta en ambas instancias con 512MB de ram y 1 dyno, tal como especifica el *tier* gratuito de Heroku.

2.2.1 Testing

Para encarar el testing, se realizaron test unitarios y de integración mediante la biblioteca **unittest** de Python. Dichos test se pueden correr en el ambiente de desarrollo local, ya que están incluidos en el source del proyecto.

En el caso del AuthServer, se cuenta con 86 tests, que cubren la mayoría de la funcionalidad de todos los endpoints del servidor, considerando también algunos flujos completos. A modo de ejemplo de esto último, para testear la funcionalidad de cierre de sesión o *logout*, se siguen los siguientes pasos:

1. Alta de usuario.
2. Creación de sesión o *login.*
3. Logout del usuario.

2.2.2 Code coverage

La cobertura de código alcanzada por los tests es monitoreada mediante el uso de la herramienta **coverage**. Para el AuthServer, se apunta a tener una cobertura mayor al 90%.

Un reporte típico de coverage tiene la forma:

```
Name                                      Stmts   Miss  Cover
-----------------------------------------------------------
/home/ubuntu/auth_server.py                 111     19    83%
/home/ubuntu/src/__init__.py                  0      0   100%
/home/ubuntu/src/adminusers.py              158     11    93%
/home/ubuntu/src/helpers.py                 211     35    83%
/home/ubuntu/src/home.py                     62      3    95%
/home/ubuntu/src/recovery.py                117      7    94%
/home/ubuntu/src/requestlog.py              128      9    93%
/home/ubuntu/src/sessions.py                149     23    85%
/home/ubuntu/src/users.py                   214     24    89%
/home/ubuntu/tests/__init__.py                0      0   100%
/home/ubuntu/tests/aux_functions.py          30      0   100%
/home/ubuntu/tests/test_adminusers.py       114      1    99%
/home/ubuntu/tests/test_helpers.py           28      1    96%
/home/ubuntu/tests/test_home.py              42      1    98%
/home/ubuntu/tests/test_recovery.py         111      4    96%
/home/ubuntu/tests/test_requestlog.py        46      1    98%
/home/ubuntu/tests/test_sessions.py          83      1    99%
/home/ubuntu/tests/test_users.py            130      1    99%
-----------------------------------------------------------
TOTAL                                      1734    141    92%
```

Dicha herramienta es utilizada tanto en forma local como en Travis. En este último caso, además, el informe de cobertura es posteado a **Coveralls**, con el propósito de tener un seguimiento más detallado, para ambos ambientes *deployados* en la nube. Puede consultarse el estado de los últimos *builds* en la siguiente URL:

https://coveralls.io/github/juanmg0511/fiuba-taller-2-auth-server

2.2.3 Estructura del proyecto

El proyecto se encuentra hosteado en GitHub. Se cuenta con dos *branches*, una para staging (develop) y otra para producción (master). Durante la fase de desarrollo, se *pushean* los updates al branch de desarrollo, y luego de cada checkpoint se actualiza master con la entrega realizada.

Los cambios menores propuestos por algún integrante del equipo, producto del uso de la API o testing es codificada en un *branch* nuevo e integrada a develop mediante *pull requests*.

La home del repositorio es:

https://github.com/juanmg0511/fiuba-taller-2-auth-server

Su estructura es:

```
.
├── Dockerfile
├── LICENSE
├── Procfile
├── README.md
├── auth_server.pid
├── auth_server.py
├── docker-compose.yml
├── docs
│   ├── AuthServer.v1.00.Manual.Administrador.gdoc
│   ├── AuthServer.v1.00.Manual.Administrador.pdf
│   ├── AuthServer.v1.00.Diseño.Diseño.gdoc
│   ├── AuthServer.v1.00.Diseño.Diseño.pdf
│   ├── AuthServer.v1.00.Diseño.Diseño.API.pdf
│   ├── AuthServer.v1.00.Diseño.Diseño.API.yaml
│   ├── AuthServer.v1.00.diagram
│   ├── AuthServer.v1.00.diagram.docker
│   ├── AuthServer.v1.00.diagram.docker.png
│   └── AuthServer.v1.00.diagram.png
├── gunicorn_config.py
├── icons
│   ├── chotuve-icon-square.png
│   ├── chotuve-icon.png
```

```
│   ├── chotuve-icon.psd
│   └── chotuve-icon.svg
├── mongoinit
│   └── mongoinit.sh
├── requirements.txt
├── runtime.txt
├── setup.py
├── src
│   ├── __init__.py
│   ├── adminusers.py
│   ├── helpers.py
│   ├── home.py
│   ├── recovery.py
│   ├── requestlog.py
│   ├── sessions.py
│   └── users.py
├── templates
│   └── mailTemplate.html
├── tests
│   ├── __init__.py
│   ├── aux_functions.py
│   ├── test_adminusers.py
│   ├── test_helpers.py
│   ├── test_home.py
│   ├── test_recovery.py
│   ├── test_requestlog.py
│   ├── test_sessions.py
│   └── test_users.py
└── tests_google_login
    ├── requirements.txt
    ├── tests_google_login.py
    └── working_example.png

7 directories, 47 files
```

2.3 Base de datos

Como se mencionó, la base de datos utilizada es MongoDB. El AuthServer utiliza el plugin de Heroku de mLab para resolver el *hosting* de este componente. En ambos ambientes se utiliza la versión gratuita, que provee 512MB de hosting sin limitaciones.

A pesar de que se trata de un motor NoSQL, se cuenta con 5 *collections*, una para cada tipo de entidad que maneja el AuthServer.

Como se verá en profundidad en la sección de diseño, la única collection con documentos de distintos tipos es “RequestLog”, donde son guardados los eventos del servidor. Dado la limitación de *storage* impuesta por mLab, dicha *Collection* tiene un *cap* de 250MB, ya que al tratarse de un log su tamaño puede ser muy grande.

Las *Collections* con *cap* funcionan como un buffer circular, por lo que de esta forma se asegura que el tamaño se mantenga acotado y los registros almacenados sean los más recientes.

3. Diseño

En esta sección se expone el diseño del AuthServer. Se utilizará una sub-sección para hacer comentarios sobre cada componente de la solución.

La especificación detallada de la API puede consultarse en la sección 3.3.

3.1 Servidor Flask

La API RESTful disponibilizada por el AuthServer fue codificada en Python 3 en su totalidad, por lo que también utiliza las facilidades de Flask y muchos de los módulos adicionales que existen para dicho *micro framework*.

En la siguiente tabla se presenta una lista de los paquetes más importantes utilizados por el AuthServer:

Nombre	Versión utilizada	Propósito
Flask	1.1.2	Framework base.
Flask-RESTful	0.3.8	Facilita la implementación de una API RESTful sobre Flask.
Flask-PyMongo	2.3.0	Conexión a la base de datos.
Flask-APScheduler	1.11.0	Ejecución de las tareas programadas lanzadas por el servidor.
Flask-Cors	0.3.8	Configuración de CORS (cross-origin resource sharing).
Flask-Log-Request-ID	0.10.1	Captura de los request ID enviados en los *headears*, de existir.
Flask-Mail	0.9.1	Envío de notificaciones de correo electrónico, para la funcionalidad de recupero de contraseña.

Asimismo, se destaca el uso los siguientes paquetes, no pertenecientes a Flask:

Nombre	Versión utilizada	Propósito
coverage	5.1	Permite calcular el porcentaje de cobertura de código de los test unitarios y de integración.
coveralls	2.0.0	Integración con la plataforma coveralls.
passlib	1.7.2	*Hashing* de contraseñas.
google	2.0.3	Integración de Sign In with Google.
google-api-core	1.17.0	Integración de Sign In with Google.
google-api-python-client	1.8.4	Integración de Sign In with Google.
google-auth	1.16.9	Integración de Sign In with Google.
google-auth-httplib2	0.0.3	Integración de Sign In with Google.
googleapis-common-protos	1.51.0	Integración de Sign In with Google.

3.1.1 API

El AuthServer disponibiliza sus servicios a través de una API RESTful. Para diseñarla, se tomó como premisa hacerla lo más simple posible. De esta forma, por ejemplo, no existe un endpoint /logout para realizar un cierre de sesión, sino que debe hacerse un DELETE de /session/<token>.

De esta forma, se logra condensar prácticamente todas las operaciones vinculadas a una entidad en un endpoint del mismo nombre.

Por otro lado, se diseñó una respuesta JSON unificada y consistente para errores y mensajes entre todos los endpoints, a fin de facilitar su interpretación. Respuestas que devuelven el mismo código HTTP (como por ejemplo 401) diferencian su origen utilizando códigos de error distintos, a fin de poder identificarlo fácilmente. Una operación exitosa tendrá código 0, mientras que los códigos menores a 0 indicarán una situación anómala o un error.

Los endpoints disponibles que proveen los servicios básicos son:

Path	Verbo	Servicio
/ping	GET	Ping.
/status	GET	Estado del front-end y base de datos.
/stats	GET	Estadísticas de uso del servidor.
/api/v1/users	POST	Registro.
/api/v1/users/<username>	GET	Consulta de perfil.
	PUT	Actualización de perfil.
	PATCH	Actualización de avatar y de contraseña.
	DELETE	Cierre de cuenta.
/api/v1/users/<username>/sessions	GET	Estado de conexión.
/api/v1/adminusers	POST	Registro.
/api/v1/adminusers/<username>	GET	Consulta de perfil.
	PUT	Actualización de perfil.
	PATCH	Actualización de contraseña.
	DELETE	Cierre de cuenta.
/api/v1/adminusers/<username>/sessions	GET	Estado de conexión.
/api/v1/sessions	POST	Inicio de sesión.
/api/v1/sessions/<session-token>	GET	Chequeo de sesión.
	DELETE	Cierre de sesión.
/api/v1/recovery	POST	Pedido de recuperación de contraseña.
/api/v1/recovery/<username>	POST	Reseteo de contraseña.

Finalmente, los path que incluyen ‘api’ se encuentran protegidos por una API key, es decir que para operar con ellos debe proporcionarse dicha clave en un header especial, X-ClientID. Si la

clave es erronea o no está presente, el servidor devuelve un error HTTP 401, es decir NOT_AUTHORIZED.

La especificación detallada de la API puede consultarse en la sección 3.3.

3.1.2 Usuarios

El AuthServer debe mantener un registro de los usuarios del sistema, así como también de su información de perfil. Un objeto del tipo usuario tiene la forma:

```
{
	"username": "JuanManuel",
	"first_name": "Juan Manuel",
	"last_name": "Gonzalez",
	"password": "*",
	"contact": {
		"email": "juan@example.com",
		"phone": "+54 11 5555 5555"
	},
	"avatar": {
		"url": "http://www.google.com/"
	},
	"login_service": false,
	"account_closed": false,
	"date_created": "2020-05-09T22:16:05.458187",
	"date_updated": null
}
```

Las contraseñas de los usuarios se almacenan en la base de datos como un *hash*, generado mediante el uso de la biblioteca *passlib* de Python.

Se destacan dos campos de uso del sistema, “login_service” y “account_closed”. El primero se define al registrar al usuario, y determina si debe utilizarse la combinación usuario/password o Sign In with Google para iniciar sesión. El valor de este campo no puede ser cambiado una vez registrado el usuario. En cuanto al segundo, los usuarios que dan de baja su cuenta pasarán a tener el “account_closed” en True, pero no son eliminados de la *collection*. Esta última acción no se puede deshacer.

3.1.3 Administradores

Así como se guarda la información de los usuarios, lo mismo se hace en forma análoga para los administradores. El objeto de un usuario administrador es de la forma:

```
{
	"username": "JuanManuel",
	"first_name": "Juan Manuel",
	"last_name": "Gonzalez",
```

```
        "password": "*",
        "email": "juan@example.com",
        "account_closed": false,
        "date_created": "2020-05-09T22:16:05.458187",
        "date_updated": null
}
```


Todas las aclaraciones hechas para los usuarios aplican para los administradores de la plataforma. Estos usuarios no pueden utilizar Sign In with Google ni pedir recuperación de contraseña.

3.1.4 Sesiones

Además de mantener una base del registro de los usuarios y administradores, otra de las funcionalidades esenciales del AuthServer es el manejo de sesiones. De esta manera, el servidor se encarga de la creación, chequeo y eliminación de las sesiones.

Un objeto de sesión tiene la forma:


```
{
        "username": "chotuvegod",
        "session_token": "d9f6c4b4-933b-11ea-9566-72ba225d89f0",
        "expires": "2020-05-11T05:59:45.740404",
        "date_created": "2020-05-11T03:59:45.740445"
}
```


Como se puede observar, la información de sesión comprende el nombre de usuario, un *token* y las fechas de expiración y creación. Al momento de crear una sesión (*login*), esta información es pasada al cliente.

Se eligió para los *token* de sesión utilizar un UUID, dada la cantidad de usuarios y sesiones esperadas. Este número puede cambiarse otro de otro tipo, o por un objeto más complejo si así se requiere.

El tiempo de validez de la sesión está definido por un delta parametrizable, según el tipo de usuario. Al hacer *login*, se asigna a la sesión un tiempo de validez de un delta, que es refrescado en un delta cada vez que la sesión es chequeada. Si el usuario no registra actividad y el tiempo de expiración vence, la sesión queda inhabilitada y se debe hacer login nuevamente, generando una nueva sesión.

Para el ambiente productivo, estos valores están fijados en 10 minutos para usuarios administradores y 1 semana para los usuarios de la APP. De esta manera, se puede crear la ilusión de una sesión “infinita” sin implementarla realmente, ya que si el usuario no usa la aplicación por más de 1 semana deberá iniciar sesión nuevamente, pero si la usa regularmente no deberá volver a hacerlo.

3.1.5 Sign In with Google

Para el login de usuarios sin contraseña, el AuthServer permite el inicio de sesión utilizando una cuenta de Google. Para ello, debe pasarse un *payload* especial al registrar un usuario, especificando que utiliza este servicio.

A la hora de iniciar una sesión, el AuthServer debe recibir para este tipo de usuarios el **id_token** proporcionado por Google luego de un login exitoso en la aplicación cliente. Recibido, el AuthServer lo decodifica utilizando la API de Google y compara que se trate del mismo usuario, y que el usuario sea válido y no tenga la cuenta cerrada. Si todo es validado correctamente, se crea una sesión en el AuthServer y se entrega un session-token, caso contrario el cliente recibe un código 401.

Esta funcionalidad solo es soportada para usuarios registrados de la aplicación, no para los administradores.

3.1.6 Recovery

La funcionalidad de recupero de contraseña funciona de manera análoga a las sesiones. Se entrega un token (en este caso llamado recovery_key) que tiene un vencimiento parametrizable (3 días por default) y se le envía un mail a un usuario. A diferencia de las sesiones, si se hace esto dos veces seguidas sin resetear la contraseña, el primer pedido queda invalidado.

El mail que recibe el usuario contiene un link que apunta a una página especial de la WebAdmin, donde el usuario puede resetear su contraseña. El link proporciona la clave y el nombre de usuario en forma *querystring*. Al resetear la contraseña exitosamente, registro del pedido es dado de baja.

El envío de los correos se resolvió con la biblioteca Flask-Mail, y utilizando el servicio de SMTP de *Mailjet* (https://www.mailjet.com), en su tier gratuito. Si bien se trata de una compañía focalizada en la generación y envío de campañas publicitarias por correo electrónico, al registrarse proporciona en forma gratuita acceso al su servidor de correo saliente sin necesidad de crear o lanzar una campaña.

Esta funcionalidad solo es soportada para usuarios registrados de la aplicación, no para los administradores.

3.1.7 Scheduled Tasks

Dado que las sesiones y los pedidos de recovery expiran luego de un tiempo dado, se programaron 2 tareas en el servidor para limpiar las *collections* correspondientes de los pedidos que no se encuentren activos. Para ello se utilizó el paquete Flask-APScheduler:

- La tarea que limpia sesiones vencidas corre 1 vez por hora.
- La tarea que limpia pedidos de recuperación de contraseña corre 1 vez por día.

Estas tareas son definidas y lanzadas desde el *init hook* de Gunicorn, para evitar que se genere una instancia por worker.

Para consultar el historial de los documentos eliminados se puede consultar el log de eventos, descrito en la sección 3.1.9, es decir que no se pierde información.

3.1.8 Log de aplicación

La aplicación cuenta con un sistema de log unificado que escribe a la consola y al archivo de log interno de Gunicorn. Tanto los logs presentes en el código, los generados por Flask y los generados por Gunicorn son dirigidos al mismo *stream*. Se soportan los niveles de log: “debug”, “info”, “warning”, “error”, “critical”. Este nivel es parametrizable mediante una variable de entorno, por lo que difiere en los distintos ambientes.

Se intentó hacer los logs lo más descriptivos posible, incluyendo abundante información de *debug*, a fin de facilitar las tareas de desarrollo. Todos los mensajes de log de la aplicación incluyen el valor del header X-Request-ID, con el propósito de identificarlos rápida e inequívocamente.

Se pueden pedir los logs a Heroku mediante el comando (el ejemplo es para el ambiente productivo, su uso en staging es análogo):

```
heroku logs -a fiuba-taller-2-auth-server
```

3.1.9 Log de eventos

Así como se cuenta con un completo log de aplicación, se creó además un log de eventos, que guarda información de 3 tipos de eventos en la base de datos:

- Requests a la API.
- Envío de mails.
- Ejecución de tareas programadas.

Estos registros permiten hacer un seguimiento detallado de los eventos que sucedieron en el servidor. Se dispone asimismo de un endpoint especial para su consumo:

```
/api/v1/requestlog
```

Este log debe solicitarse en un rango de fechas (el mínimo es un día), y se incluye como extra un parámetro para filtrar los resultados mediante un *query*, que permite acotar los resultados a un valor determinado. La idea de esta funcionalidad es hacer búsquedas muy específicas, no proveer una API de búsqueda de registros, por lo que su implementación es muy acotada.

Los detalles de su funcionamiento se encuentran descritos en la sección 3.3.

3.1.10 Estadísticas

Tomando como base el log de eventos del punto anterior, se construyó un servicio que recopila esta información y provee estadísticas de uso y actividad del AuthServer.

La información entregada está compuesta de un *snapshot* de la situación de las cuentas del servidor a la hora de hacer el pedido, más un array de estadísticas, compuesto de un registro por día para el rango de fechas solicitado a la hora de hacer la consulta.

Estos datos pueden utilizarse, por ejemplo, para consultar la cantidad de usuarios registrados, o graficar la evolución de los parámetros reportados en un intervalo dado de tiempo.

Los detalles de su funcionamiento se encuentran descritos en la sección 3.3.

3.2 Base de datos

Se presenta en esta sección el diseño de la base de datos utilizada, como se mencionó en las secciones anteriores, se trata de una base MongoDB.

3.2.1 Collections

Las collections utilizadas son:

Nombre	Descripción
Users	Almacena documentos que representan los usuarios registrados de la aplicación.
Adminusers	Almacena documentos que representan los usuarios administradores de la aplicación.
Sessions	Almacena documentos que representan las sesiones de ambos tipos de usuarios.
Recovery	Almacena documentos que representan los pedidos de recuperación de contraseña de los usuarios de la aplicación.
RequestLog	Almacena tres tipos de documentos: **requests**, **task** y **mail**, que representan los eventos procesados por el AuthServer.

3.2.2 Tipos de documentos

Se presenta a continuación ejemplos para cada tipo de documento utilizado, se puede ver su definición en mayor profundidad consultando la sección 3.3:

User


```
{
	"id": "5eb72ba5d2824353b87e5e94",
	"username": "JuanManuel",
	"first_name": "Juan Manuel",
	"last_name": "Gonzalez",
	"password": "*",
	"contact": {
		"email": "juan@example.com",
		"phone": "+54 11 5555 5555"
	},
	"avatar": {
		"url": "http://www.google.com/"
	},
	"login_service": false,
	"account_closed": false,
	"date_created": "2020-05-09T22:16:05.458187",
	"date_updated": null
}
```


Los usuarios que poseen “**login_service**” en true no utilizan el campo “**password**”.

Adminuser


```
{
	"id": "5eb72ba5d2824353b87e5e94",
	"username": "JuanManuel",
	"first_name": "Juan Manuel",
	"last_name": "Gonzalez",
	"password": "*",
	"email": "juan@example.com",
	"account_closed": false,
	"date_created": "2020-05-09T22:16:05.458187",
	"date_updated": null
}
```

Session


```
{
	"id": "5eb8cdb1ec54d8c9b812c074",
	"username": "chotuvegod",
	"session_token": "d9f6c4b4-933b-11ea-9566-72ba225d89f0",
	"expires": "2020-05-11T05:59:45.740404",
	"date_created": "2020-05-11T03:59:45.740445"
}
```


Recovery


```
{
	"id": "5eb8cdb1ec54d8c9b812c074",
	"username": "chotuvegod",
	"recovery_key": "d9f6c4b4-933b-11ea-9566-72ba225d89f0",
	"expires": "2020-05-11T05:59:45.740404",
	"date_created": "2020-05-11T03:59:45.740445"
}
```


RequestLog:request


```
{
	"_id": "5edbc4544cf6a8f05a97055d",
	"log_type": "request",
	"request_date": "2020-06-06T16:29:08.668802",
	"request_id": "6b3020de-271c-4fc7-98ac-18e06c2bf54a",
	"remote_ip": "181.99.163.199",
	"host": "fiuba-taller-2-auth-server-st.herokuapp.com",
	"api_version": "v1",
	"method": "GET",
	"path": "/api/v1/users/JuanManuel",
	"status": "200 OK",
	"duration": "0.211577",
	"headers": {
		"Host": "fiuba-taller-2-auth-server-st.herokuapp.com",
		"Connection": "close",
		"X-Client-Id": "407f7dbf-57d5-4aea-bfbd-d317ae872428",
		"User-Agent": "advanced-rest-client",
		"Accept": "*/*",
		"X-Request-Id": "6b3020de-271c-4fc7-98ac-18e06c2bf54a",
		"X-Forwarded-For": "181.99.163.199",
		"X-Forwarded-Proto": "https",
		"X-Forwarded-Port": "443",
```

```
            "Via": "1.1 vegur",
            "Connect-Time": "0",
            "X-Request-Start": "1591460948436",
            "Total-Route-Time": "0"
      },
      "args": {},
      "data": {},
      "response": {
            "id": "5eb72ba5d2824353b87e5e94",
            "username": "JuanManuel",
            "first_name": "Juan Manuel",
            "last_name": "Gonzalez",
            "contact": {
                  "email": "juan0511@icloud.com",
                  "phone": "+54 11 5555 5555"
            },
            "avatar": {
                  "url": null
            },
            "login_service": false,
            "account_closed": false,
            "date_created": "2020-05-09T22:16:05.458187",
            "date_updated": "2020-06-04T03:01:06.100869"
      }
}
```


RequestLog:mail


```
{
      "_id": "5ed9341f3531938e30928e32",
      "log_type": "mail",
      "request_date": "2020-06-04T17:49:19.289279",
      "request_id": "2702e2c2-dcea-4b01-b50b-3992f1e1930b",
      "api_version": "v1",
      "username": "guido321@gmail.com",
      "email": "guido321@gmail.com",
      "recovery_key": "b6609e04-a68b-11ea-98ba-4627c31bbba5",
      "mail_status": "Mail dispatched successfully.",
      "exception_message": null
}
```

RequestLog:task


```
{
	"_id": "5edad3171bb9b9aef97f9c3f",
	"log_type": "task",
	"request_date": "2020-06-05T23:19:51.353960",
	"task_type": "prune expired sessions",
	"api_version": "v1",
	"pruned_sessions": 0
}
```


3.3 Especificación OpenAPI 3.0

La API del servidor fue documentada en su totalidad utilizando la especificación OpenAPI 3.0, trabajando con el editor de Swagger. La misma puede consultarse en la ubicación:

https://juanmg0511.github.io/fiuba-taller-2-documents/docs-auth-server/api.html

Adicionalmente, se incluye una copia de dicha especificación en formato PDF junto al presente documento, pero dadas las características interactivas de este tipo de documentación se recomienda su consulta en la URL anteriormente expuesta.