UNIVERSIDAD DE BUENOS AIRES

FACULTAD DE INGENIERÍA

Año 2020 - 1er Cuatrimestre

Taller de Programación II (75.52)

ChoTuve

AppServer

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 AppServer utilizado en la aplicación ChoTuve.

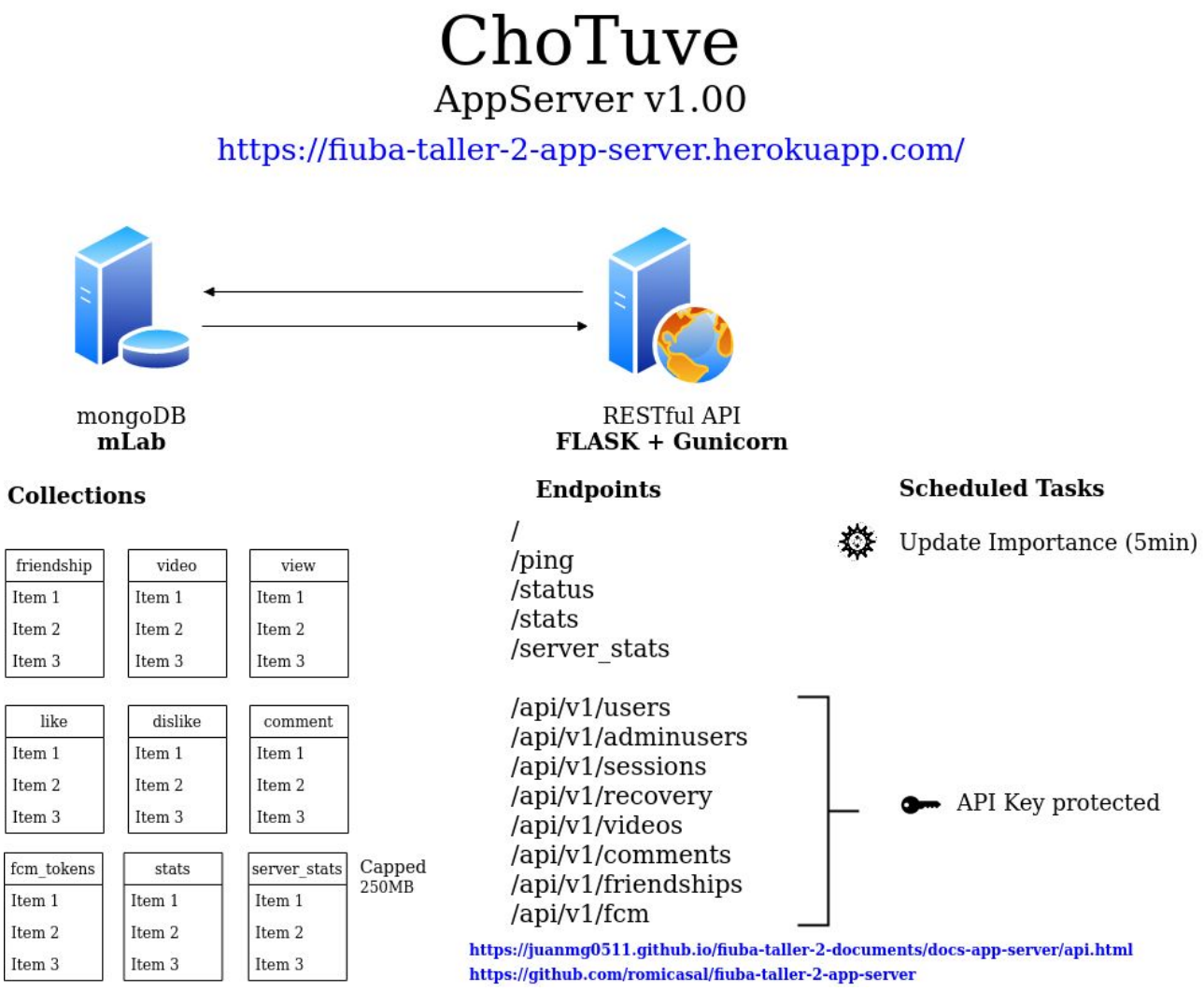
Adicionalmente, se incluye la especificación completa de la API provista por el servidor, en formato OpenAPI 3.0.

2. Arquitectura

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

2.1 Consideraciones generales

A grandes rasgos, la arquitectura del AppServer 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-app-server-st.herokuapp.com/
STAGING: https://fiuba-taller-2-app-server-st.herokuapp.com/

El AppServer está compuesto por un front-end que consiste en una aplicación **Flask** corriendo sobre **Gunicorn**, proveyendo una serie de servicios de metadata sobre videos y usuarios, y

haciendo de nexo entre los componentes de la solución. 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 App 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*.

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

```
AppServer.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**.

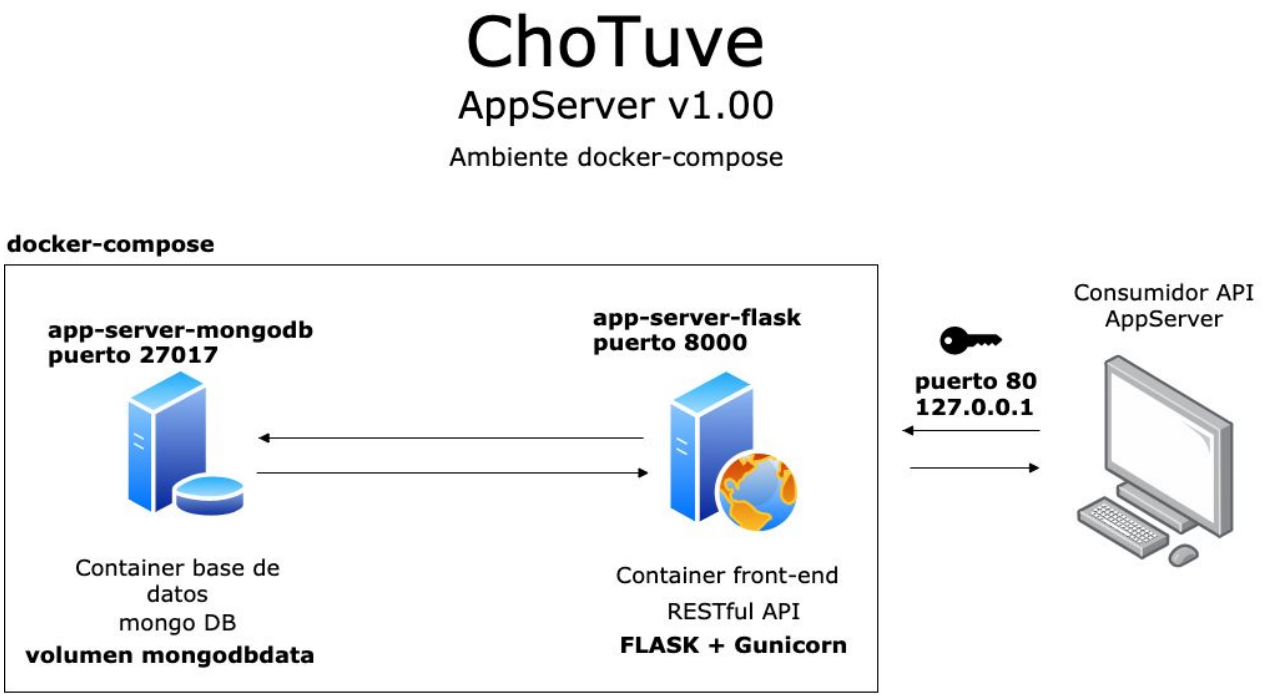
Dicha plataforma es la encargada de realizar la ejecución de pruebas, subir la imagen dockerizada del proyecto a docker.io y hacer el *deploy* a Heroku, para los dos ambientes disponibles, staging y producción.

Los resultados de las distintas corridas pueden consultarse en:

https://travis-ci.com/github/romicasal/fiuba-taller-2-app-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:

`AppServer.v1.00.Manual.Administrador.pdf`

2.2 Front end

El front end del AppServer 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 AppServer 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 AppServer, se cuenta con 139 tests, que cubren la mayoría de la funcionalidad de todos los endpoints del servidor, considerando también algunos flujos completos.

2.2.2 Code coverage

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

```
Name                                  Stmts   Miss  Cover
-------------------------------------------------------
app_server.py                            11      1    91%
src/__init__.py                          35      0   100%
src/clients/__init__.py                   0      0   100%
src/clients/auth_api.py                  59      9    85%
src/clients/fcm_api.py                   49     18    63%
src/clients/media_api.py                 21      9    57%
src/conf/__init__.py                     21      0   100%
src/conf/database.py                      6      0   100%
src/conf/jobs.py                         10      0   100%
src/conf/log.py                           1      0   100%
src/conf/routes.py                       39      0   100%
src/jobs/__init__.py                      0      0   100%
src/jobs/update_video_importance.py      12      6    50%
src/misc/__init__.py                      0      0   100%
src/misc/authorization.py                19      1    95%
src/misc/importance.py                   39      0   100%
src/misc/requests.py                     19      0   100%
src/misc/responses.py                     4      0   100%
src/misc/validators.py                   25      2    92%
src/models/__init__.py                    0      0   100%
src/models/comment.py                    11      0   100%
src/models/fcm_token.py                   5      0   100%
src/models/friendship.py                 27      0   100%
src/models/reaction.py                   24      0   100%
src/models/stat.py                       13      0   100%
```

```
src/models/video.py                          41    4     90%
src/resources/__init__.py                    0     0   100%
src/resources/adminusers.py                  28    2     93%
src/resources/comments.py                    62    0   100%
src/resources/fcm_tokens.py                  32    3     91%
src/resources/friendships.py                 92    0   100%
src/resources/reactions.py                   113   10    91%
src/resources/recovery.py                    18    2     89%
src/resources/sessions.py                    27    8     70%
src/resources/status.py                      23    0   100%
src/resources/users.py                       73    26    64%
src/resources/videos.py                      131   17    87%
src/schemas/__init__.py                      0     0   100%
src/schemas/avatar.py                        6     0   100%
src/schemas/comment.py                       20    0   100%
src/schemas/friendship.py                    19    0   100%
src/schemas/location.py                      6     0   100%
src/schemas/pagination.py                    10    0   100%
src/schemas/reaction.py                      9     0   100%
src/schemas/stat.py                          27    0   100%
src/schemas/video.py                         34    0   100%
src/services/__init__.py                     0     0   100%
src/services/fcm.py                          61    11    82%
src/services/stats.py                        58    2     97%
src/services/user.py                         25    3     88%
src/services/video.py                        31    2     94%
tests/__init__.py                            0     0   100%
tests/test_adminusers.py                     78    1     99%
tests/test_comments.py                       177   0   100%
tests/test_fcm.py                            65    0   100%
tests/test_friendships.py                    231   0   100%
tests/test_importance.py                     22    0   100%
tests/test_reactions.py                      251   0   100%
tests/test_recovery.py                       57    7     88%
tests/test_schemas/__init__.py               0     0   100%
tests/test_schemas/test_comment.py           38    0   100%
tests/test_schemas/test_stat.py              32    0   100%
tests/test_schemas/test_video.py             34    0   100%
tests/test_sessions.py                       51    7     86%
tests/test_stats.py                          88    1     99%
tests/test_users.py                          137   1     99%
tests/test_utils/__init__.py                 0     0   100%
tests/test_utils/utils.py                    118   1     99%
tests/test_videos.py                         310   0   100%
------------------------------------------------------------
TOTAL                                3085  154   95%
```

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/romicasal/fiuba-taller-2-app-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/romicasal/fiuba-taller-2-app-server

Su estructura es:

```
./
├── app_server.pid
├── app_server.py
├── docker-compose.yml
├── Dockerfile
├── docs
│   └── placeholder.txt
├── LICENSE
├── mongoinit
│   └── mongoinit.sh
├── Procfile
├── README.md
├── requirements.txt
├── runtime.txt
├── setup.py
├── src
│   ├── clients
│   │   ├── auth_api.py
│   │   ├── fcm_api.py
│   │   ├── __init__.py
│   │   └── media_api.py
│   ├── conf
```

```
│   │   ├── database.py
│   │   ├── gunicorn.py
│   │   ├── __init__.py
│   │   ├── jobs.py
│   │   ├── log.py
│   │   └── routes.py
│   ├── __init__.py
│   ├── jobs
│   │   ├── __init__.py
│   │   └── update_video_importance.py
│   ├── misc
│   │   ├── authorization.py
│   │   ├── importance.py
│   │   ├── __init__.py
│   │   ├── requests.py
│   │   ├── responses.py
│   │   └── validators.py
│   ├── models
│   │   ├── comment.py
│   │   ├── fcm_token.py
│   │   ├── friendship.py
│   │   ├── __init__.py
│   │   ├── reaction.py
│   │   ├── stat.py
│   │   └── video.py
│   ├── resources
│   │   ├── adminusers.py
│   │   ├── comments.py
│   │   ├── fcm_tokens.py
│   │   ├── friendships.py
│   │   ├── __init__.py
│   │   ├── reactions.py
│   │   ├── recovery.py
│   │   ├── sessions.py
│   │   ├── status.py
│   │   ├── users.py
│   │   └── videos.py
│   ├── schemas
│   │   ├── avatar.py
│   │   ├── comment.py
│   │   ├── friendship.py
│   │   ├── __init__.py
│   │   ├── location.py
│   │   ├── pagination.py
│   │   ├── reaction.py
│   │   ├── stat.py
│   │   └── video.py
```

```
│   └── services
│       ├── fcm.py
│       ├── __init__.py
│       ├── stats.py
│       ├── user.py
│       └── video.py
├── tests
│   ├── __init__.py
│   ├── test_adminusers.py
│   ├── test_comments.py
│   ├── test_fcm.py
│   ├── test_friendships.py
│   ├── test_importance.py
│   ├── test_reactions.py
│   ├── test_recovery.py
│   ├── test_schemas
│   │   ├── __init__.py
│   │   ├── test_comment.py
│   │   ├── test_stat.py
│   │   └── test_video.py
│   ├── test_sessions.py
│   ├── test_stats.py
│   ├── test_users.py
│   ├── test_utils
│   │   ├── __init__.py
│   │   └── utils.py
│   └── test_videos.py
└── tree.html

14 directories, 82 files
```

2.3 Base de datos

Como se mencionó, la base de datos utilizada es MongoDB. El AppServer 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.

Se cuenta con 8 *collections*, una para cada tipo de entidad que maneja el AppServer.

Dada la limitación de *storage* impuesta por mLab, la *Collection* server_stats 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 AppServer. 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 AppServer 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 AppServer:

Nombre	Versión	Propósito
APScheduler	3.6.3	Ejecución de las tareas programadas lanzadas por el servidor
Flask	1.1.2	Framework base
Flask-Cors	0.3.8	Configuración de CORS (cross-origin resource sharing)
Flask-PyMongo	2.3.0	Conexión a la base de datos
Flask-RESTful	0.3.8	Facilita la implementación de una API RESTful sobre Flask
business-rules	1.0.1	Motor de reglas para calcular importancia de un video
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
flask-mongoengine	0.9.5	Extensión que facilita la integración con Mongoengine
gunicorn	20.0.4	Servidor HTTP
marshmallow	3.6.0	Serialización y validación de

		requests
mongomock	3.19.1.dev19	Probar el código Python que interactúa con MongoDB a través de Pymongo sin usar una base real
pip	9.0.1	Instalador de paquetes
pylint	2.5.2	Cumplimiento de los estándares de código y evitar code smells
pymongo	3.10.1	Driver para MongoDB
requests	2.23.0	Realizar requests a otras apis
setuptools	39.0.1	Empaquetamiento

3.1.1 API

El AppServer disponibiliza sus servicios a través de una API RESTful. Para diseñarla, se tomó como premisa hacerla lo más simple posible. 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 son:

Path	Verbo	Servicio
/api/v1/ o /	GET	Home devuelve un mensaje de bienvenida en texto plano
/api/v1/ping o /api	GET	Ping.
/api/v1/status o /status	GET	Estado del server y base de datos.
/api/v1/server_stats o /server_stats	GET	Estadísticas de uso del server
/api/v1/stats o /stats	GET	Estadísticas de uso del sitio
/api/v1/sessions	POST	Inicio de sesión
	GET	Chequeo de la sesión del

		usuario {X-Auth-Token}
	DELETE	Cerrar la sesión del usuario {X-Auth-Token}
/api/v1/sessions/:session-token	GET	Chequeo de sesión
	DELETE	Cierre de sesión
/api/v1/users	POST	Registro de usuario
	GET	Consulta de perfil del usuario {X-Auth-Token}
/api/v1/users/:username>	GET	Consulta de perfil de un usuario
	PUT	Actualización de perfil.
	PATCH	Actualización de avatar y de contraseña.
	DELETE	Cierre de cuenta.
/api/v1/users/:username/sessions	GET	Todas las sesiones del usuario
/api/v1/users/:username/avatars	POST	Guarda o modifica el avatar
	DELETE	Borra el avatar
/api/v1/users/:username/friends	GET	Listado paginado de amigos
/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/fcm	POST	Alta de un dispositivo (token) para push notifications
	DELETE	Borra un token asociado del listado
/api/v1/recovery	POST	Pedido de recuperación de contraseña
/api/v1/recovery/:username	POST	Reseteo de contraseña
/api/v1/videos	POST	Alta de un video
	GET	Search de videos paginado

/api/v1/videos/:video_id	GET	Obtiene los datos del video
	PUT	Modificación de datos de video
	DELETE	Borrado de un video
/api/v1/videos/:video_id/likes	GET	Listado de likes paginado
	POST	Agregar like a un video
	DELETE	Borrar el like
/api/v1/videos/:video_id/likes	GET	Listado de “no me gusta” paginado
	POST	Agregar un dislike
	DELETE	Borrar un dislike
/api/v1/videos/:video_id/views	GET	Listado de views en un video
	POST	Registrar que el usuario vio el video
/api/v1/comments	POST	Comentar o responder un comentario en un video
	GET	Search de comentarios paginado
/api/v1/:comments	GET	Obtener un comentario
	DELETE	Borrar un comentario
/api/v1/friendship	POST	Solicitar una amistad
	GET	Search de solicitudes paginado
/api/v1/friendship/:friendship_id	PUT	Modificar el estado de una solicitud (aprobar)
	DELETE	Borrar la solicitud

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

3.1.2 Flujos e Interacción con los distintos componentes

A continuación se detallan algunos de los flujos más importantes para dar una idea general del funcionamiento de esta api.

Se supone que en cada flujo quién hace la petición es alguno de los clientes Mobile o WebAdmin.

3.1.2.1 Usuarios

Alta de usuario

1. El AppServer se comunica con el AuthServer y hace un passthrough de la respuesta

Obtener datos de un usuario

1. El AppServer se comunica con el AuthServer para validar el token
2. Valida que el usuario tenga permisos para acceder al registro
3. Se vuelve a comunicar con el AuthServer para solicitar el perfil del usuario solicitado
4. Obtiene el avatar del MediaServer
5. Responde con todos los datos

Borrar un usuario

1. El AppServer se comunica con el AuthServer para validar el token
2. Valida que el usuario tenga permisos para acceder al registro
3. Se vuelve a comunicar con el AuthServer para borrar al usuario
4. Borra el avatar del MediaServer
5. Responde al cliente

3.1.2.2 Videos

Alta de un video

1. El AppServer se comunica con el AuthServer para validar el token
    a. Si no es válido retorna Unauthorized
2. Valida que el usuario tenga permisos para dar de alta un video (los usuarios admines no pueden)
    a. Si no es un usuario válido retorna Forbidden
3. Valida los datos de la petición
    a. En caso de no ser válido retorna BadRequest
4. Guarda el video en la base
5. Se comunica con el MediaServer para guardar la metadata del video y obtener la url
    a. En caso de error retorna InternalServerError
6. Responde al cliente con todos los datos del video guardado

Modificación de un video

1. El AppServer se comunica con el AuthServer para validar el token
    a. Si no es válido retorna Unauthorized
2. Valida que el usuario tenga permisos para modificar un video
    a. Si no es un usuario válido retorna Forbidden

3. Valida los datos de la petición
    a. En caso de no ser válido retorna BadRequest
4. Guarda el video en la base
5. Se comunica con el MediaServer para obtener la metadata del video junto con la url
    a. En caso de error retorna InternalServerError
6. Responde al cliente con todos los datos del video guardado

Borrado de un video

1. El AppServer se comunica con el AuthServer para validar el token
2. Valida que el usuario tenga permisos para borrar el video
    a. Si no es el dueño del video o un usuario admin retorna Forbidden
3. Valida los datos de la petición
    a. En caso de no ser válido retorna BadRequest
4. Guarda el video en la base
5. Se comunica con el MediaServer para guardar la metadata del video y obtener la url
    a. En caso de error retorna InternalServerError
6. Responde OK

3.1.2.3 Comentarios

Alta de un comentario

1. El AppServer se comunica con el AuthServer para validar el token
    a. Si no es válido retorna Unauthorized
2. Valida que el usuario tenga permisos para dar de alta un video
    a. Si es un usuario admin retorna Forbidden
3. Valida los datos de la petición
    a. En caso de no ser válido retorna BadRequest
4. Verifica que exista el video (y el comentario en caso de que sea una respuesta a otro comentario)
    a. En caso de que no exista el recurso devuelve NotFound
5. Guarda el comentario en la base
6. Responde al cliente con todos los datos del comentario guardado

Borrar un comentario

1. El AppServer se comunica con el AuthServer para validar el token
    a. Si no es válido retorna Unauthorized
2. Valida que el usuario tenga permisos para dar de alta un video
    a. Si no es el propietario del comentario ni es usuario admin retorna Forbidden
3. Valida los datos de la petición
    a. En caso de no ser válido retorna BadRequest
4. Verifica que exista el comentario
    a. En caso de que no exista el recurso devuelve NotFound
5. Borra el comentario de la base
6. Responde OK

3.1.2.3 Amistades

Solicitar amistad

1. El AppServer se comunica con el AuthServer para validar el token
    a. Si no es válido retorna Unauthorized
2. Valida que el usuario tenga permisos para solicitar amistad
    a. Si es un usuario admin retorna Forbidden
3. Valida los datos de la petición
    a. En caso de no ser válida retorna BadRequest
4. Verifica que exista el usuario al cual se desea enviar la solicitud
    a. En caso de que no exista el usuario devuelve NotFound
5. Guarda la solicitud de amistad en la base con estado pendiente
6. Envía una push notification a través de FCM al destinatario de la solicitud
7. Responde al cliente con todos los datos de la solicitud guardada

Aceptar amistad

1. El AppServer se comunica con el AuthServer para validar el token
    a. Si no es válido retorna Unauthorized
2. Valida que exista la solicitud
    a. Si no existe devuelve NotFound
3. Valida que el usuario tenga permisos para aceptar la amistad
    a. Si no es el destinatario de la solicitud devuelve Forbidden
4. Valida los datos de la petición
    a. En caso de no ser válida retorna BadRequest
5. Modifica el estado de la solicitud a “approved”
6. Envía una push notification a través de FCM al usuario que solicitó la amistad
7. Responde al cliente con todos los datos de la solicitud modificada

3.1.3 Otras herramientas importantes

3.1.3.1 Push notifications

Para las push notifications utilizamos el servicio FCM de Firebase

Frente a los eventos que se detallan a continuación, el AppServer envía, a través de Firebase una notificación a todos los dispositivos registrados por el usuario.

Los eventos que disparan estos mensajes son:

- Solicitud de amistad
- Solicitud de amistad aprobada
- Nuevo comentario en tu video
- Nuevo like en tu video

Como se mencionó anteriormente en la descripción de los distintos flujos, el AppServer disponibiliza un endpoint para el registro de un token de un usuario. Cada vez que el usuario se loguea en un nuevo dispositivo, la aplicación mobile envía el token para que se persista en el AppServer.

3.1.3.2 Motor de reglas

Para calcular la importancia de un video se utiliza un motor de reglas (python business-rules)

Actualmente la forma de calcular la importancia es tomar ciertos valores estadísticos del video y el usuario, como pueden ser la cantidad de amigos del propietario o la cantidad de likes. A cada una de estas variables se le asigna un peso y con esto se calcula el valor de la importancia.

La utilización de business rules nos permite fácilmente cambiar los parámetros de ajuste según distintos criterios.

3.1.3.3 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-app-server
```

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 del Documento

Videos	Datos de un video (no incluye la metadata del archivo, sólo el nombre)
Friendships	Datos de una solicitud de amistad entre dos usuarios y su estado
Reactions (Likes, Dislikes, Views)	Reacción de un usuario en un video
FCMToken	Tokens de un usuario para el envío de push notifications, cada token representa un dispositivo
Comment	Almacena los comentarios y también la referencia al comentario padre para soportar respuestas
Stat	Datos de una request, tiempos de respuesta, status, etc.

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-app-server/api.html