UNIVERSIDAD DE BUENOS AIRES

FACULTAD DE INGENIERÍA

Año 2020 - 1er Cuatrimestre

Taller de Programación II (75.52)

ChoTuve

Media Server

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 Media Server utilizado en la aplicación ChoTuve.

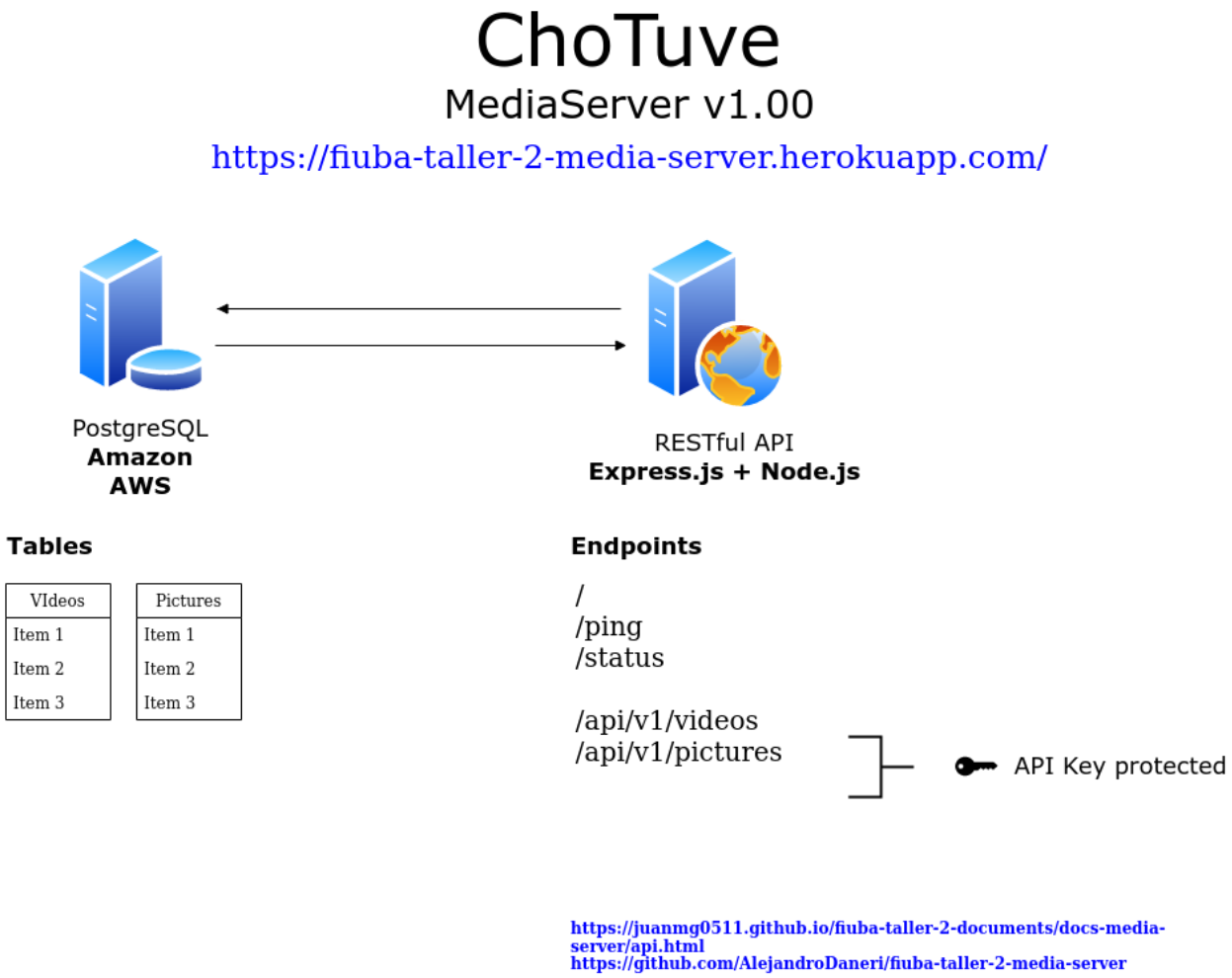
Adicionalmente, se incluye la especificación completa de la API provista por el servidor, en formato OpenAPI 3.0.

2. Arquitectura

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

2.1 Consideraciones generales

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

El Media Server está desarrollado usando **Express.js** que corre sobre **Node.js**, proveyendo una serie de servicios que, si bien se ahondará sobre estas capacidades en la sección de diseño de este documento, se puede observar el gráfico para tener un pantallazo del diseño del servidor.

La base de datos es del tipo PostgreSQL, hosteada en Amazon AWS y accedida mediante el uso del add on que está disponible en Heroku.

2.1.1 Configuración

El Media 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:

```
MediaServer.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 Deployment, mediante el uso de la herramienta **Travis CI**.

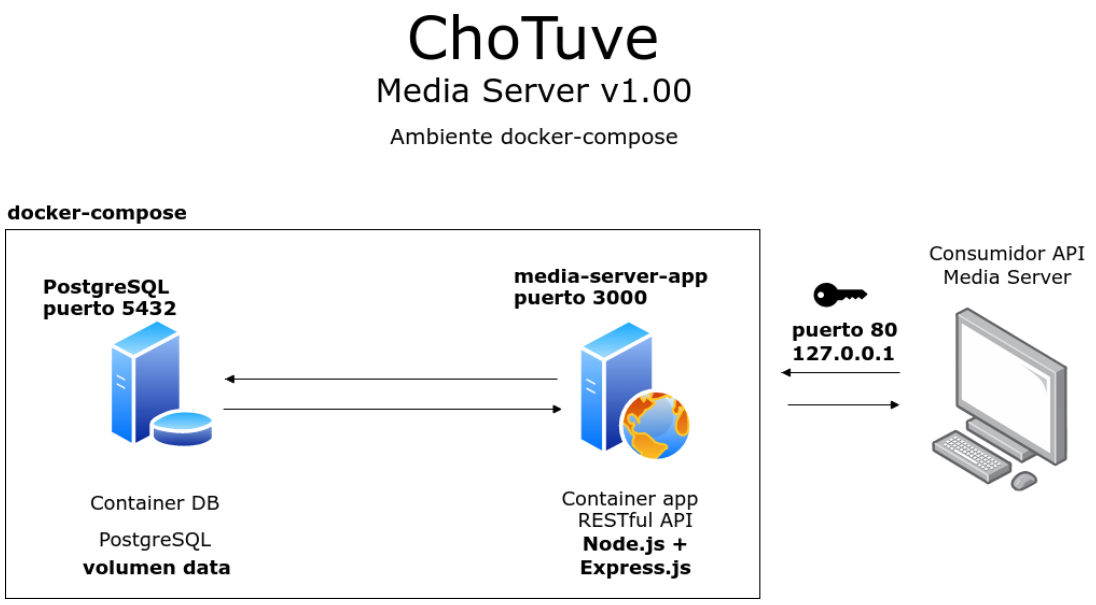
Dicha plataforma es la encargada de realizar la ejecución de pruebas, subir la imagen dockerizada del proyecto a docker.io y hacer el *deploy* a Heroku, para los dos ambientes disponibles, staging y producción.

Los resultados de las distintas corridas pueden consultarse en:

https://travis-ci.com/github/AlejandroDaneri/fiuba-taller-2-media-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:

`MediaServer.v1.00.Manual.Administrador.pdf`

2.2 Front end

El front-end del Media Server ejecuta una instancia de la imagen de docker sobre Heroku que esta hosteada en docker.io.

En términos de recursos, el Media Server 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 el framework **jest**[1]. Dichos test se pueden correr en el ambiente de desarrollo local, ejecutandolo mediante el script *test*

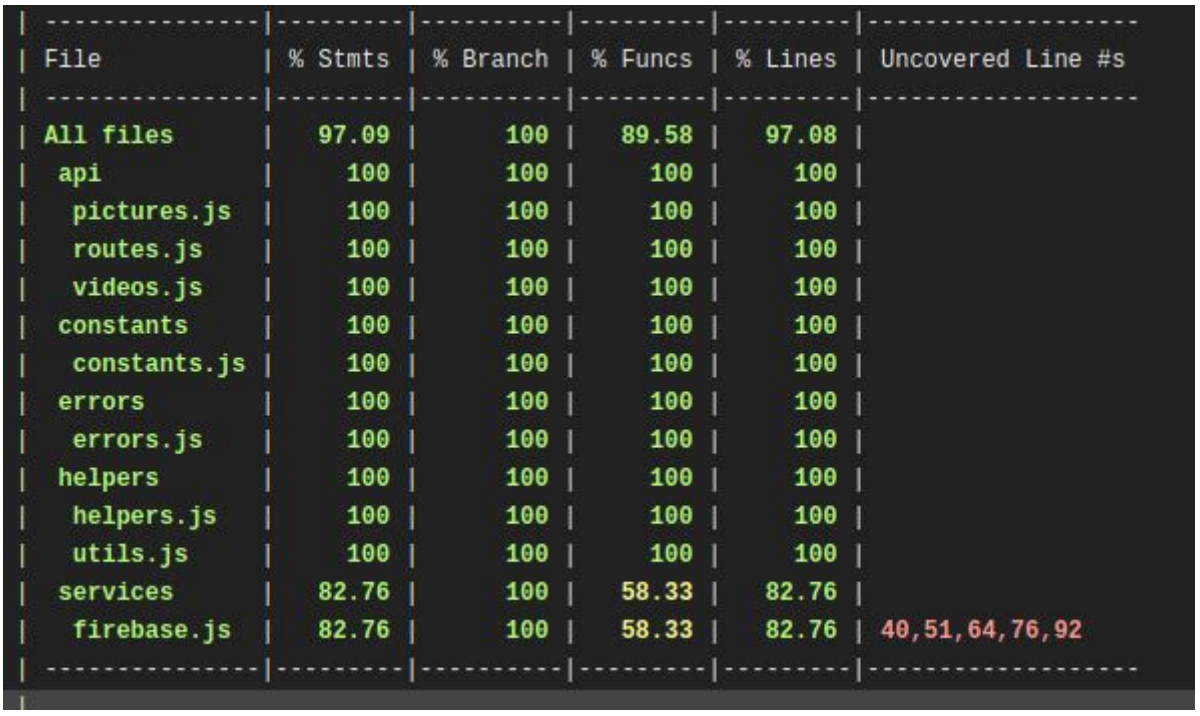
Se cubren la mayoría de la funcionalidad de todos los endpoints del servidor, considerando también algunos flujos completos.

2.2.2 Code coverage

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

Un reporte típico de coverage tiene la forma:

[1] https://jestjs.io/



File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	97.09	100	89.58	97.08	
api	100	100	100	100	
pictures.js	100	100	100	100	
routes.js	100	100	100	100	
videos.js	100	100	100	100	
constants	100	100	100	100	
constants.js	100	100	100	100	
errors	100	100	100	100	
errors.js	100	100	100	100	
helpers	100	100	100	100	
helpers.js	100	100	100	100	
utils.js	100	100	100	100	
services	82.76	100	58.33	82.76	
firebase.js	82.76	100	58.33	82.76	40,51,64,76,92

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/AlejandroDaneri/fiuba-taller-2-media-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 correspondiente a la nueva feature que se introduce, una vez finalizada se procede a pasar los cambios a develop para pruebas integrales, 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/AlejandroDaneri/fiuba-taller-2-media-server

Su estructura es:

```
├─── app.js
├─── db
│    ├─── Dockerfile
│    ├─── init.sql
│    ├─── knex.js
│    ├─── migrations
│    │    ├─── 20200514194304_videos.js
│    │    └─── 20200611120651_pictures.js
│    ├─── queries.js
│    └─── seeds
│      ├─── development
│      │    ├─── pictures_seed.js
│      │    └─── videos_seed.js
│      └─── test
│             ├─── pictures_seed.js
│             └─── videos_seed.js
├─── docker-compose.travis.yml
├─── docker-compose.yml
├─── Dockerfile
├─── firebase.json
├─── knexfile.js
├─── package.json
├─── package-lock.json
├─── README.md
├─── src
│    ├─── api
│    │    ├─── helpers
│    │    │    └─── helpers.js
│    │    ├─── pictures.js
│    │    ├─── routes.js
│    │    └─── videos.js
│    ├─── config
│    │    └─── logger.js
│    ├─── constants
│    │    └─── constants.js
│    ├─── errors
│    │    └─── errors.js
│    ├─── __mocks__
│    │    ├─── firebase.mock.js
│    │    └─── utils.mock.js
│    ├─── services
│    │    └─── firebase.js
│    └─── utils
│      ├─── authUtils.js
│      └─── pgUtils.js
└─── tests
       ├─── authUtils.spec.js
       ├─── firebase.spec.js
       ├─── pgUtils.spec.js
       ├─── pictures.spec.js
       ├─── routes.spec.js
       └─── videos.spec.js
```

2.3 Base de datos

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

Dentro de la base de datos se cuenta con 2 *relaciones*, una para cada tipo de entidad que maneja el Media Server.

3. Diseño

En esta sección se expone el diseño del Media Server. 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 Node.js

La API RESTful disponibilizada por el Media Server fue codificada en NodeJS en su totalidad, mediante la utilización del framework Express.js y algunos módulos adicionales que existen para NodeJS.

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

Nombre	Versión utilizada	Propósito
express	4.17.1	Framework base.
dotenv	8.2.0	Utilización de archivos de entorno
firebase	7.14.2	Framework base para la interacción con Firebase
firebase-tools	8.2.0	Utilizado para la creación y monitorización de recurso de Firebase
cors	2.85	Configuración de CORS (cross-origin resource sharing).
http-status-codes	1.4.0	Maneja los códigos HTTP para que queden mejor documentados
pg	8.0.3	Framework base para la interacción con la base PostgreSQL
coveralls	3.1.0	
firebase-admin	8.11.0	Utilizada para la autenticación sobre Firebase
knex	0.21.1	Simplifica la comunicación con la base de datos
jest	25.5.4	Se utilizó para la creación de los tests así como también el cálculo

		de coverage
nodemon	2.0.4	Herramienta que monitorea cambios en el código y al encontrarlos reinicia el servidor automáticamente
supertest	4.0.2	Herramienta para facilitar los tests de integración

3.1.1 API

El Media Server disponibiliza sus servicios a través de una API RESTful. Para diseñarla, se tomó como premisa hacerla lo más simple posible. 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.
/api/v1/pictures	POST	Registro nuevo avatar.
/api/v1/pictures/<userID>	GET	Consulta de avatar.
	PUT	Actualización de avatar.
	PATCH	Actualización de avatar.
	DELETE	Borrado de avatar.
/api/v1/videos	POST	Registro nuevo video.
/api/v1/videos/<videoID>	GET	Consulta de video.
	PUT	Actualización de avatar.
	DELETE	Borrado de video.

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 el header especial, X-Client-ID. 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 Videos

Para el almacenamiento de los videos que fueron subidos a ChoTuve se utilizó el siguiente esquema:

video_id	name	user_id	date_created	type	size	url	thumb
varchar	varchar	varchar	varchar	varchar	int	varchar	varchar

Video_id: id del video asociado a una publicación
Name: nombre del archivo
User_id: usuario que subio el video
Date_created: fecha de subida del video
Type: Tipo de archivo / formato
Size: Tamaño del video subido en bytes
Url: link del video en Firebase
Thumb: link del thumb del video en Firebase

3.1.3 Avatars

Para el almacenamiento de los avatares que fueron utilizados por los usuarios de ChoTuve se utilizó el siguiente esquema:

name	user_id	url
varchar	varchar	varchar

Name: nombre del archivo
User_id: usuario que subio el video
Url: link del avatar en Firebase

3.1.4 Log de aplicación

La aplicación cuenta con un sistema de log que escribe sobre la consola . Tanto los logs presentes en el código. Se soportan los niveles de log: “debug”, “info”,”log”,“warning”, “error”. Este nivel es parametrizable mediante una variable de entorno.

Para cada tipo de log se asoció un color, a fin de resaltar cada tipo de log de forma rápida. Cada uno tiene de prefijo la hora y fecha de cuándo sucedió. A su vez cuenta con tablas descriptivas (en el nivel de debug) del payload de las request para hacer más directo el seguimiento de errores en caso de ser necesario

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-media-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 PostgreSQL. Para la comunicación con la base de datos se usó el framework knex.

3.2.1 Relaciones

Las relaciones utilizadas son:

Nombre	Descripción
Videos	Almacena los videos que fueron subidos a la aplicación.
Pictures	Almacena los avatares que fueron subidos a la aplicación.

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-media-server/api.html