UNIVERSIDAD
DE SANTIAGO
DE CHILE

DOCUMENTACIÓN
DASHBOARD-BACKEND

Índice

1. DESCRIPCIÓN GENERAL

El presente documento entregará detalles sobre el funcionamiento del backend construido para el dashboard del sistema centralizado de autorización. El sistema ofrece 11 endpoints para ser consumidos, los cuales permiten, entre otros, modificar el mensaje de bienvenida, añadir un rol a una aplicación, actualizar los nombres o urls de aplicaciones, obtener una lista de roles disponibles y obtener una lista de aplicaciones disponibles.

A lo largo de este documento se busca plasmar todo lo correspondiente al funcionamiento del sistema. En primer lugar se encuentra una visión general de la arquitectura, donde se describen los componentes que componen el sistema y cómo estos están conectados. Posterior, se encuentra una descripción de los 11 endpoints disponibles, indicando que necesitan para operar y qué respuesta pueden dar. Posteriormente, se presenta el diagrama de secuencia de los principales endpoints.

2. Arquitectura

2.1 Características

El backend del dashboard fue programado utilizando Django 3.0.5 y Python 3.7.7. Para encapsular los métodos programados y poder ofrecerlos mediante API para ser consumidos por el frontend, se utilizó Django Rest Framework 3.11.0.

Además de lo anterior, se utiliza como base de datos mongoDB. Para que esta pueda ser utilizada haciendo uso de la ORM de Django, se utilizó Djongo 1.3.3, el cual es una librería que permite conexiones entre Django y bases de datos no relacionales, pero manteniendo la ORM relacional con la que Django viene equipado.

Finalmente, para poder ofrecer el sistema completo, se utilizó Docker 20.10.2. Docker permite crear contenedores que mantendrán funcionando la aplicación y la base de datos. Para una mayor simpleza a la hora de levantar el sistema, se utilizó Docker Compose 1.27.4, creando un único archivo docker-compose.yml (a explicar más adelante) el cual posee los contenedores de Django y Mongodb para el funcionamiento interno, así como el contenedor de Nginx para que este revisa las peticiones de los usuarios.

Finalmente, se incluye un archivo requirements.txt en donde se encuentran todas las librerías necesarias para la correcta ejecución del sistema.

2.2 Diagrama

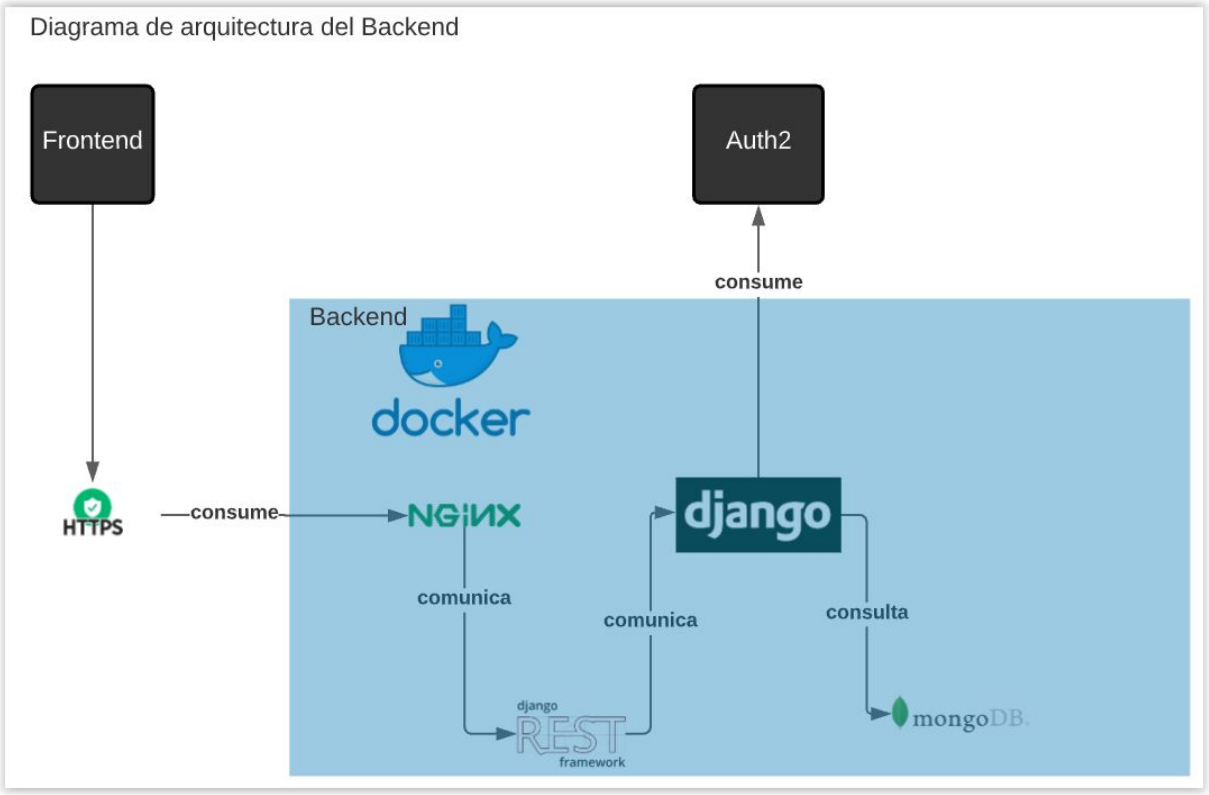

Figura 2.2.1 – Diagrama de arquitectura de backend.

En el diagrama anterior se pueden apreciar dos cosas importantes: Primero, para efectos del backend, el frontend y Auth2 son considerados cajas negras, pues este no se ve afectado por su funcionamiento interno (aún así, son importantes debido a que Auth2 es la fuente desde donde el backend verifica al usuario).

En segundo lugar, se puede ver cómo interactúan las tecnologías mencionadas anteriormente: llega una petición al servidor la cual es procesada por nginx y, si corresponde, redirigida a Django Rest Framework. Este, quien tiene encapsulados los métodos del backend, recibe la petición e invoca al endpoint correspondiente. Posteriormente, se ejecuta el código del endpoint, y se hacen las consultas necesarias: Si se está utilizando el endpoint de verificación, por ejemplo, se consumirá LDAP para determinar el rol y las aplicaciones del usuario. Finalmente, se consulta a la base de datos para acceder a los datos de la aplicación (url e imagen), lo cual es finalmente retornado al frontend. Tal como se aprecia en la imagen, todo esto se encuentra encapsulado por docker, el cual contiene los contenedores del sistema.

3. Endpoints

El backend ofrece 11 endpoints, los cuales pueden ser consumidos por el frontend según sea necesario. A continuación, se detalla el funcionamiento de cada uno de estos endpoints

3.1 /list-apps

Descripción: El endpoint /list-apps permite ofrecer la lista completa de aplicaciones que tiene el sistema. Es particularmente útil para que los administradores tengan acceso a ellas cuando se quieren asignar a un nuevo usuario.

Tipo método: GET

Parámetros: -

Retorno: Lista de aplicaciones del sistema

3.2 /list-roles

Descripción: El endpoint /list-roles permite ofrecer la lista completa de roles que tiene el sistema. Es particularmente útil para que los administradores tengan acceso a ellos cuando se quieren asignar a un nuevo usuario.

Tipo método: GET

Parámetros: -

Retorno: Lista de roles del sistema

3.3 /user-apps

Descripción: El endpoint /user-apps permite al sistema reconocer el rol de un usuario y retornar sus aplicaciones. Para esto, accede a la cookie 'DIINFAUTH2USERTOKEN', y consume el endpoint /authorize de auth² para obtener el rol. Posteriormente, se ejecuta una consulta a la base de datos para obtener las aplicaciones ligadas al rol obtenido, el cual es retornado al frontend.

Tipo método: GET

Parámetros: Sin parámetros, pero se requiere la existencia de la cookie 'DIINFAUTH2USERTOKEN'

Retorno: Rol de un usuario

3.4 /create-app

Descripción: Endpoint utilizado para crear una nueva aplicación. Recibe como parámetro el nombre, la imagen y la url de la nueva aplicación.

Tipo método: POST

Parámetros: Nombre de la nueva app ('app_name'), url ('app_url'), y link de la imagen ('img').

Retorno: status 200 si la creación fue correcta, status 500 si hubieron errores.

3.5 /create-role

Descripción: Endpoint utilizado para crear un nuevo rol. Recibe como parámetro el nombre del nuevo rol.

Tipo método: POST

Parámetros: Nombre del nuevo rol ('role_name').

Retorno: status 200 si la creación fue correcta, status 500 si hubieron errores.

3.6 /add-app

Descripción: Añade una aplicación existente a un rol existente. Recibe como parámetros la id de la aplicación y la id del rol.

Tipo método: POST

Parámetros: Id de la aplicación ('app_id'), id del rol ('role_id').

Retorno: status 200 se pudo añadir correctamente la aplicación, status 500 si hubieron errores.

3.7 /delete-app

Descripción: Elimina una aplicación existente del sistema, quedando inaccesible desde cualquier rol que la tuviese.

Tipo método: POST

Parámetros: Id de la aplicación (‘id’)

Retorno: status 200 se pudo eliminar la aplicación, status 500 si hubieron errores.

3.8 /delete-role

Descripción: Elimina un rol existente del sistema. A diferencia de la eliminación de las aplicaciones, borrar un rol NO borrará ninguna aplicación asociada.

Tipo método: POST

Parámetros: Id del rol a eliminar (‘id’).

Retorno: status 200 se pudo eliminar el rol, status 500 si hubieron errores.

3.9 /update-app

Descripción: Actualiza el nombre, la url o la imagen de una aplicación.

Tipo método: POST

Parámetros: Id de la aplicación (‘id’), nuevo nombre de la aplicación (‘app_name’)*, nueva url de la aplicación (‘app_url’)*, nueva imagen de la aplicación (‘img’)*.

*** Al menos un parámetro es requerido. Los otros pueden no incluirse.**

Retorno: status 200 se pudo actualizar la aplicación, status 500 si hubieron errores.

3.10 /create-message

Descripción: Endpoint para crear el mensaje de bienvenida.

Tipo método: POST

Parámetros: Nuevo mensaje de bienvenida (‘message’)

Retorno: status 200 s pudo crear correctamente el mensaje, status 500 si hubieron errores.

3.11 /get-message

Descripción: Retorna el último mensaje de bienvenida creado por el sistema.

Tipo método: GET

Parámetros: -

Retorno: Último mensaje de bienvenida de la base de datos.

4. Pruebas

El backend cuenta con pruebas que permiten determinar si es que el funcionamiento interno es el esperado o no. Estas pruebas fueron programadas usando TestCase y RequestFactory, y prueban 9 de los 11 endpoints.

A continuación se ejemplifica la prueba del endpoint /add-app:

```python
# Add app to role Test
# Its successful if the system can add an app to a role
class AddAppToRole(TestCase):

    def test(self):
        # New App
        new_app = RequestFactory().post('/create-app', {'app_name': 'App 1', 'app_url': 'url 1', 'app_img': 'url_1
        # New Role
        new_role = RequestFactory().post('/create-role', {'role_name': 'Rol 1'})
        # Create the new elements
        views.create_app(new_app)
        views.create_role(new_role)
        # Get the role using a query
        role = Role.objects.get(name='Rol 1')
        # get the app using a query
        app = App.objects.get(name='App 1', url='url 1')
        # Testing the endpoint
        new_role = RequestFactory().post('/create-role', {'app_id': app.pk, 'role_id': role.pk})
        final = views.add_app_to_role(new_role)
        self.assertEqual(final.status_code, 200)
```

Figura 4.1 – Docker-compose: contenedor de base de datos.

Tal como se muestra en la imagen 4.1, se hacen varios pasos para realizar la prueba. En primera instancia, se preparan dos peticiones POST a los endpoints /create-app y /create-role para crear una aplicación de pruebas y un rol de pruebas respectivamente. Luego, se ejecutan dichas peticiones utilizando las funciones create_app y create_role. Posteriormente, se obtienen ambos elementos directo desde la base de datos, para confirmar que el proceso de creación fue exitoso. Finalmente, se prepara una request con la id de ambos elementos, y se invoca al endpoint /add-app, el cual intentará añadir la aplicación al rol. La prueba se considera exitosa si el código de retorno es 200, indicando que se pudo añadir satisfactoriamente la aplicación al rol.

Para ejecutar todos los tests de la aplicación se puede usar el siguiente comando

```
$ python manage.py test
```

Obteniendo una respuesta parecida a la imagen 4.2:

```
.........
----------------------------------------------------------------------
Ran 9 tests in 4.350s

OK
Destroying test database for alias 'default'...
```

Figura 4.2 – Retorno esperado para tests.

5. Producción

Para hacer deploy del backend, este cuenta con un archivo docker-compose.yml que contiene los detalles de los contenedores a crear:

```yaml
db:
  image: mongo:latest
  container_name: db_pingeso
  environment:
    - MONGO_INITDB_DATABASE=dashbork
    - MONGO_INITDB_ROOT_USERNAME=root
    - MONGO_INITDB_ROOT_PASSWORD=pingeso
  ports:
    - "27017:27017"
  volumes:
    - ./mongo-entrypoint:/docker-entrypoint-initdb.d
      # named volumes
    - mongodb:/data/db
    - mongoconfig:/data/configdb
```

Figura 5.1 – Docker-compose: contenedor de base de datos.

```yaml
webapp:
  build:
    context: .
    dockerfile: Dockerfile
  restart: always
  command: gunicorn server.wsgi:application --bind 0.0.0.0:8000
  volumes:
    - static_volume:/home/app/web/static
    - certs:/etc/certs/
  expose:
    - 8000
  depends_on:
    - db
```

Figura 5.2 – Docker-compose: Contenedor de django.

```
nginx:
  build:
    context: ./nginx/
  ports:
    - 80:80
    - 443:443
  volumes:
    - static_volume:/home/app/web/static
    - config/:/etc/nginx/conf.d
    - certs:/etc/certs/
  depends_on:
    - webapp
```

Figura 5.3 – Docker-compose: Contenedor de nginx

En las imágenes, se puede ver cuales son los tres contenedores que creará docker: DB, que contiene la base de datos del sistema. Este usa la imagen de mongo como base, y crea una base de datos en el puerto 27017 con 'root' como usuario, y 'pingeso' como contraseña.

También está webapp, que es el contenedor que tiene el código de django. Se expone el puerto 8000 para que este sea accesible internamente. Finalmente, está el contenedor 'nginx', el cual tiene dentro la configuración y el funcionamiento de nginx. Tal como se puede apreciar, nginx escucha en los puertos 80 y 443 (SSL), y posee su propio *context* de creación en donde se encuentra su archivo de configuración y su Dockerfile.

Tal como se explicó anteriormente, junto al archivo docker-compose y Dockerfile, se incluye el archivo requirements.txt, el cual tiene una lista detallada de las dependencias necesarias para ejecutar el programa. En caso de usar docker-compose, el Dockerfile de la aplicación se encarga de ejecutar la instalación de estas dependencias. En caso de querer instalar estas dependencias a nivel local para desarrollo, se puede utilizar el siguiente comando*:

```
$ pip install -r requirements.txt
```

**Testeado en Windows 10 Enterprise 64 bits.*

Finalmente, para levantar la aplicación en producción solo se necesita ejecutar el siguiente comando:

```
$ docker-compose up -d --build
```

Lo cual creará los contenedores en modo detached (-d), y los construirá en caso de ser necesario (--build)