Documentación SICC

1. Tecnologías Utilizadas

1.1. Frontend

1.1.1. Angular

Angular 1.5 fue utilizado como framework para el desarrollo de la aplicación web.

1.1.2. SASS

SASS es una extensión de css más robusta y completa. Se utilizó la versión 3.2.0.

1.1.3. Android

1.2. Backend

1.2.1. Python / Django

El servidor intermedio se encuentra implementado en dichas tecnologías, en sus versiones 3.4 y 1.10 respectivamente.

1.2.2. PostgreSQL

Utilizado para la implementación de la base de datos, en su versión 9.3.

1.2.3. Celery

Celery es una cola de tareas asíncrona basada en el pasaje de mensajes distribuidos, está escrito en Python. A la entrada de la cola de tareas se le llama *task,* se tiene un proceso *worker* el cual se encuentra todo el tiempo monitoreando la cola con el fin de ver si tiene nuevas tareas a realizar.
Celery se comunica vía mensajes utilizando un *broker* para conectar los clientes con los *workers*, el *broker* que elegimos fue RabbitMQ.

Para iniciar una *task* un cliente pone un mensaje en la cola y el *broker* entrega el mensaje al *worker.*

Celery se utilizó en su versión 3.1 tanto para el envío de mail como en la implementación de las push notifications.

1.2.4. Firebase

Firebase es un servicio ofrecido por Google cuya función es servir como plataforma de almacenamiento y sincronización. A su vez también funciona como back-end como servicio.

Fue utilizado en la implementación de las push notifications.

1.2.5. Docker

Docker sirve para la automatización en el despliegue de aplicaciones dentro de contenedores de software, proporcionando una capa adicional de abstracción y automatización de Virtualización a nivel de sistema operativo en Linux.

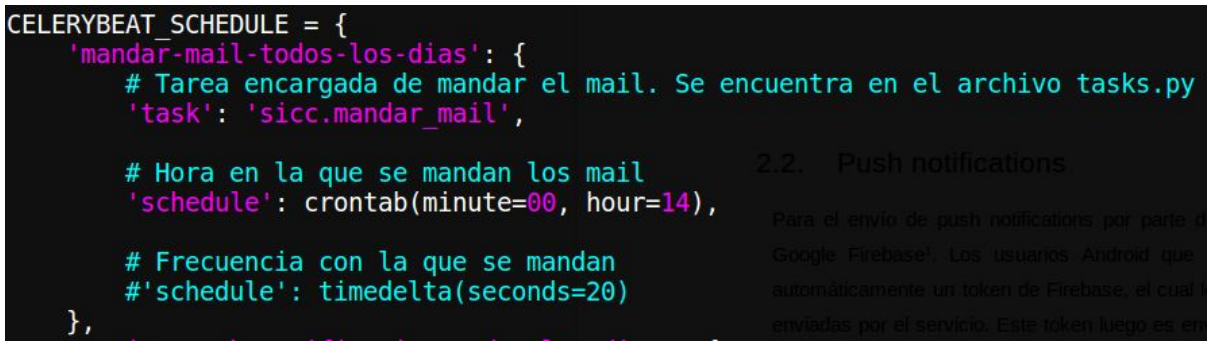
Fue utilizado para la implantación de nuestro sistema: servidor intermedio, base de datos, aplicación web.

2. Tareas Automatizadas

2.1. Envío de mails

El envío de mails se realiza todos los días a las 14 horas, enviando un mail a todos aquellos estudiantes que tuvieran planificada notificaciones para dicho día. La frecuencia con la que se realiza esta tarea es modificable, esta configuración se encuentra en el archivo settings.py dentro del directorio Servidor/sicc/sicc/ .

Para enviar los mail se utilizó el servidor de gmail, se creó la cuenta *sicc.notificaciones@gmail.com* con contraseña *siccpis2016,* desde la cual se realiza el envío. En el archivo settings.py dentro del directorio Servidor/sicc/sicc/ se encuentran los parámetros definidos para el correcto envío. A continuación se adjunta una imagen donde se pueden observar los parámetros del archivo settings.py que se deben modifcar en caso de querer variar la hora y/o la frecuencia con la que se envian los mails.



2.2. Push notifications

Para el envío de push notifications por parte del servidor se utilizó el servicio de Google Firebase[1]. Los usuarios Android que inicien sesión en la app obtienen automáticamente un token de Firebase, el cual les permite recibir las notificaciones enviadas por el servicio. Este token luego es enviado y almacenado en una tabla de la base de datos del servidor (TokenDispositivo). En caso de que el servicio Firebase actualice el token para un dispositivo dado, el mismo es notificado y procede a actualizar el token en la base de datos del servidor.

Una vez que un usuario de la aplicación Android se suscribe a una notificación de examen, queda guardado en la base de datos toda la información asociada a ese recordatorio: usuario, fecha del recordatorio, etc.

Por otro lado, en el servidor corre el gestor de tareas Celery, el cual todos los días a la hora configurada revisa si tiene alguna notificación a enviar. De ser así, se comunica con el servicio Firebase a través de una API REST, indicando que desea enviar una notificación con un contenido X al dispositivo con token Y. Esto es lo que permite al usuario recibir la push notification en su celular el día que configuró para que se lo notifique.

Las notificaciones a los celulares se envían con la misma frecuencia que los mails. Para cambiar la frecuencia se debe modificar el archivo settings.py dentro del directorio Servidor/sicc/sicc/. A continuación se adjunta una imagen donde se pueden observar los parámetros del archivo settings.py que se deben modifcar en caso de querer variar la hora y/o la frecuencia con la que se envian los mails.

[1] Firebase: https://firebase.google.com/?hl=es

```
'enviar-push-notification-todos-los-dias': {
    # Tarea encargada de mandar las push. Se encuentra en el archivo tasks.py
    'task': 'sicc.mandar_push',

    # Hora en la que se mandan las push notification
    'schedule': crontab(minute=00, hour=14),

    # Frecuencia con la que se mandan
    #'schedule': timedelta(seconds=40)
},
```

3. Configuración Web

3.1. IP servidor intermedio

La IP del servidor intermedio se configura en el archivo env.js en el directorio Web/app/scripts. La variable window.__env.apiUrl define el protocolo (http ó https) a utilizar, en qué IP y puerto se van a realizar los pedidos al servidor intermedio.

Por ejemplo:

```
// URL base del servidor central. Quien ofrece los servicios
window.__env.apiUrl = 'http://127.0.0.1:8000';
```

3.2. Estilos

Los estilos utilizados se encuentran dentro de los archivos .scss, donde estos a sus vez están todos importados dentro del main.scss. Esto implica que si se desea realizar algún cambio sobre los mismos, se debe compilar el main.scss para generar un nuevo archivo .css con todos los estilos. Dicho archivo se encuentra incluido únicamente en el index.html.

4. Configuración Android

4.1. IP servidor

La IP del servidor se configura en el archivo ManejadorConexion.java, el cual se encuentra en el directorio App/app/src/main/java/grupo5/com/sicc/conexion. La variable SERVER_IP de la clase ManejadorConexion define a qué IP y puerto se van a realizar los pedidos al servidor intermedio.

Por cuestiones de practicidad, al no tener prevista una IP fija para el servidor intermedio, se dejará en la pantalla de login de la app el campo IP junto con el botón OK.

Esto permite que el usuario pueda setear la IP del servidor intermedio sin necesidad de modificar manualmente el código y re-generar el apk.

Una vez que se disponga de un hosting definitivo para el servidor intermedio las modificaciones a realizar en la app son las siguientes:

- En el archivo **App/app/src/main/res/activity_login.xml** eliminar el código:

```
<LinearLayout
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
  android:layout_gravity="center|fill_vertical"
  android:layout_weight="0.2"
  android:gravity="center">

  <EditText
    android:id="@+id/campoIP"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:width="200dp"
    android:backgroundTint="@color/colorTextoMenu"
    android:hint="IP"
    android:singleLine="true"
    android:textColor="@color/colorTextoMenu"
```

```
        android:textColorHint="@color/colorHint" />

    <Button
        android:text="OK"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/button"
        android:layout_alignTop="@+id/campoIP"
        android:layout_toRightOf="@+id/checkboxRecordarme"
        android:layout_toEndOf="@+id/checkboxRecordarme"
        android:onClick="guardarIP" />

</LinearLayout>
```

- En el archivo **App/app/src/main/java/grupo5/com/sicc/Login.java** eliminar el método ***guardarIP*** y también eliminar el código:

```
File file = getApplicationContext().getFileStreamPath("ip");
if (file.exists()) {
  try {
    FileInputStream fis = context.openFileInput("ip");
    InputStreamReader isr = new InputStreamReader(fis);
    BufferedReader bufferedReader = new BufferedReader(isr);
    String line = bufferedReader.readLine();
    con.setIP(line);
  } catch (FileNotFoundException e) {
    e.printStackTrace();
  } catch (IOException e) {
    e.printStackTrace();
  }
}
```

El cual se encuentra dentro del método:

```
protected Pair<String, String> doInBackground(String... args)
```

- Por último, en el archivo

App/app/src/main/java/grupo5/com/sicc/conexion/ManejadorConexion.java

se debe eliminar el método ***setIP*** y posteriormente cambiar la definición de la variable **SERVER_IP** de manera de que quede de la forma:

```
public static String SERVER_IP = "XXX.XXX.XXX.XXX:54321";
```

Donde XXX.XXX.XXX.XXX representa la IP del servidor intermedio y 54321 el puerto del mismo.

4.2. Push notifications

En el producto desarrollado en Android, la clase encargada de recibir las push notifications es la clase ListenerNotificaciones.java dentro del paquete *utils*. Cuando llega una push notification al celular la clase chequea que el usuario haya marcado la opción de ser recordado. Si es así, muestra la push notification con la información pertinente. Si tocamos esta push notification, nos llevará a la vista de Notificaciones de nuestra aplicación. Estas push notification, como se explicó anteriormente, son recibidas a través del servicio de Google Firebase, enviadas desde nuestro servidor intermedio.

Para identificar a cada uno de los dispositivos del usuario, con el fin de enviar las notificaciones a los dispositivos correctos, también empleamos el servicio Firebase. Para esto utilizamos la clase ServicioID.java, encontrada también en el paquete *utils*. Esta clase se encarga de recibir un identificador del móvil proporcionado por Firebase y enviarlo a nuestro servidor intermedio.

5. Configuración Servidor Intermedio

5.1. Credenciales servidor central

El usuario y contraseña para realizar los pedidos al servidor provisto por el cliente se definen en el archivo settings.py (*Servidor/sicc/sicc/)* en las variables USUARIO_SICC y CONTRASEÑA_SICC respectivamente. Dicho archivo se encuentra en el directorio sicc/sicc.

5.2. IP servidor central

En el archivo settings.py (*Servidor/sicc/sicc/)* se define la variable "IP_SERVIDOR_CENTRAL" en la cual se especifica el protocolo, la IP y el puerto en que se puede acceder a los servicios provistos por el servidor central. Por ejemplo:

```
# IP del servidor central.
IP_SERVIDOR_CENTRAL = 'http://0.0.0.0:12345'
```

5.3. Autenticación

5.3.1. Actual

Actualmente la autenticación a la aplicación (Android y Web) se realiza consumiendo un servicio publicado en el servidor intermedio. Este recibe un par (ci_usuario, contraseña_usuario) que valida contra la base de datos del servidor, particularmente en la tabla sicc_server_estudiante. En caso de que exista un registro con cédula "ci_usuario" y contraseña "contraseña_usuario" el servicio retorna True. En caso contrario retorna False.

5.3.2. A futuro

Dada la privacidad que requiere toda la información asociada a la actividad del estudiante y su escolaridad, es indispensable que solamente el usuario correctamente autenticado pueda consultar por su información. De esta forma surge la necesidad de que a futuro, el manejo de la autenticación del sistema sea implementado por otro servidor de autenticación que sea capaz de verificar que un usuario que se intenta registrar sea realmente quien dice ser.

Una posibilidad sería que el sistema disponga, así como la plataforma EVA, de todos los usuarios pre-cargados con sus respectivas contraseñas. Otra posibilidad es que el servicio de autenticación provisto por el framework Django (utilizado en el servidor del sistema desarrollado) verifique contra otro servidor antes de dar el una respuesta afirmativa. Para implementar esto

puede observarse, en la clase *CustomAuthToken* en el archivo `views.py`, como se sobre escribió el método de Django que verifica el token de autenticación del usuario para que devuelva adicionalmente los datos del usuario:

```
class CustomObtainAuthToken(ObtainAuthToken):
    def post(self, request, *args, **kwargs):
        response = super(CustomObtainAuthToken, self).post(request, *args,
        **kwargs)
        token = Token.objects.get(key=response.data['token'])
        print(request.data)
        estudiante = Estudiante.objects.get(ci=request.data['username'])
        return Response({'token': token.key, 'user' :
        EstudianteSerializer(estudiante).data })
```

Como se puede apreciar, esta clase posee un método *post* el cual es el encargado de procesar el pedido de login del frontend.

Lo que resulta de particular interés a los efectos del presente documento, es que en este método puede ejecutarse la acción que se desee antes de devolverle la respuesta al usuario. A modo de ejemplo, se podría enviar los datos de autenticación a otro servidor y re-enviar su respuesta al frontend.

La solución presentada anteriormente resulta la más sencilla de implementar y de menor impacto para el sistema existente, aunque no tan elegante.

Otra forma de resolver el problema de autenticación sería utilizar los llamados *backends de autenticación*[2] de Django. Estos backends de autenticación proveen una extensión al sistema de autenticación provisto por Django por defecto, permitiendo autenticar contra diferentes servicios, tales como LDAP, permitiendo implementarlos uno mismo.

6. Configuración HTTPS

6.1. Servidor WEB

Para la configuración https se usó un servidor nginx; para poder descargarlo se puede correr el siguiente comando en linux: *`sudo apt-get -y install nginx.`*

[2] Django authentication backends: https://docs.djangoproject.com/en/1.10/topics/auth/customizing/

El archivo de configuración de nginx se encuentra dentro del docker bajo el nombre nginx.conf, en él se especifica el puerto 443 para escuchar los pedidos al igual que la ruta donde se encuentra la app:

```
root /path/to/web/app/;
```

También en este archivo se especifica el certificado a utilizar por el servidor y la dirección de este, por ejemplo:

```
ssl_certificate /etc/nginx/ssl/nginx.crt;
ssl_certificate_key /etc/nginx/ssl/nginx.key;
```

Para crear este certificado se usó la herramienta OpenSSL, con la siguiente línea:

sudo openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout /etc/nginx/ssl/nginx.key -out /etc/nginx/ssl/nginx.crt

Al utilizar un certificado autofirmado se debe de aceptar este en el navegador, por lo cual cuando se desee ingresar a la web app por primera vez aparecerá una página en la que el navegador nos dirá que estamos intentando ingresar a una página no segura. Al aceptar dicho certificado, se podrá ingresar a la web app.

6.2. Servidor Intermedio

Los certificados en el servidor intermedio se generan de la misma manera que en la aplicación web, utilizando OpenSSL. Además se utiliza la herramienta uwsgi para poder ejecutar el servidor utilizando los certificados creados, y por lo tanto, que se cambie el protocolo a https.
Para instalar uwsgi se puede utilizar pip (pip install uwsgi).

6.3. Android

Las clases encargadas de la conexión HTTPS en la aplicación Android son *CustomSSLSocketFactory.java* y *CustomX509TrustManager.java* dentro del paquete *conexion.* No se realiza un encriptado TLS, pero los datos en la conexión entre la

aplicación y el servidor intermedio si se encuentran encriptados. Se debería utilizar TLS en un futuro.

7. Docker

7.1. Composición

El docker entregado contiene:

- Un servicio con una base postgresql publicada en el puerto 5433.
- Un servicio para un servidor nginx (publicado en el puerto 443 u 80, dependiendo del tipo de conexión) para la aplicación web.
- Un servicio para el servidor intermedio, publicado en el puerto 54321.
- Un servicio para Celery.

En el directorio ./servidor/ se encuentra un script (“run_celery.sh”) que se encarga de ejecutar Celery y Rabbit. El mismo es ejecutado por el servicio Celery al levantar el docker.

7.2. Configuración inicial

Antes de levantar el docker se deben realizar una serie de configuraciones iniciales para lograr la comunicación entre web - servidor intermedio y servidor intermedio - servidor central.

1. En el archivo *./servidor/sicc/sicc/settings.py* asignar a la variable IP_SERVIDOR_CENTRAL el siguiente string: *(http\https):///(IP):(PUERTO)*. Donde IP y PUERTO son la ip y puerto correspondiente a donde está publicado el servidor central. Por lo general IP = *ip_máquina* donde corre el docker del servidor central y PURTO = 12345.
2. En el archivo */web/app/scripts/env.js* asignar a la variable “window.__env.apiUrl" el string *(http|https):///(IP):(PUERTO)* donde IP es la ip de la computadora en donde se levanta el docker que contiene con el servidor y la web y PUERTO = 54321.

7.3. BUILD and UP

Para crear las imágenes de los servicios ejecutar y luego levantar el docker ejecutar:

1. sudo docker-compose build
2. sudo docker-compose up

7.4. Ejecución

7.4.1. Web

Previo a la ejecución de la aplicación web se deben validar los certificados del servidor en el navegador. Para ello acceder a la URL

https://IP_MAQUINA_DONDE _CORRE_DOCKER:54321/

Luego puede acceder a la aplicación web accediendo desde la siguiente URL:

https://IP_MAQUINA_DONDE _CORRE_DOCKER/#/login

7.4.2. Android

Lo primero que debemos realizar es instalar la aplicación a través del APK proporcionado al cliente. Al abrir la aplicación veremos un un campo en la parte inferior de la pantalla con el hint ***IP***, donde deberemos ingresar

IP_MAQUINA_DONDE _CORRE_DOCKER:54321

Luego presionar el botón que se encuentra a la derecha de este campo. Tras hacer esto podremos tener conexión con el servidor. Si no se ingresa la dirección IP correcta o el servidor no está levantado, la aplicación mostrará un popUp donde explicará que no pudo realizar una conexión con el mismo.

7.5. Usuarios

Los usuarios posibles para ingresar a la aplicación se encuentran especificados con id y contraseña en el archivo README ubicado en la carpeta raíz del docker.

7.6. Puertos utilizados

- 54321 - Servidor intermedio.
- 12345 - Servidor del cliente.
- 443/80 - Web.
- 5432 - Base de datos del cliente.
- 5433 - Base de datos del servidor intermedio.
- 5672 - Rabbit.

7.7. Posibles problemas

- Puertos ya utilizados.
  Asegúrese de que no hayan instancias de postgres corriendo especialmente en los puertos 5433 y 5432. Para corroborar si hay procesos corriendo pueden ejecutar *ps -A | grep postg*.
- No poder iniciar sesión en la web.
  Realizar la validación de los certificados a través del navegador. Para ello ingresar a: *https://IP_MAQUINA_DONDE _CORRE_DOCKER:54321/*

Observaciónes.

- La estructura de la base de datos se crea utilizando la herramienta “migrate” de Django.
- Los datos iniciales se cargan utilizando la herramienta loaddata de Django y el archivo “init_carreras.json” e “init_estudiante.json”
- Como los certificados no son certificados válidos (incluyendo los del servidor central) tanto el servidor intermedio como la aplicación Android deben especificar en el código que al hacer los request no se validen los certificados. En la web cuando se añade los certificados a las excepciones en el navegador hace que no se validen.
  Esto no hace que no se encripten los paquetes.

8. Diferencias entre Web y Android

Existen algunas diferencias de implementación entre la aplicación web y la aplicación Android. Las mismas se encuentran detalladas a continuación:

- En Android es posible indicar que ya no se desea ser notificado del vencimiento de cierto examen, mientras que en la web no. Sin embargo, al eliminar la notificación en Android, también se elimina de la Web.
- Si se cambia el mail en la web para alguna notificación, el cambio no se ve reflejado en la app hasta que se ingresa a la vista *Exámenes Pendientes* nuevamente. Si se cambia en Android el mail del usuario, se verá en la web al confirmar una notificación luego de recargar la página.
- Pop up del planificador de Android se encuentra ordenado por defecto por materias, mientras que en la aplicación web no sigue un orden en particular.
- En Android no se persiste la selección de la carrera. En la aplicación web si.

9. Errores conocidos

Se encuentran descritos en el documento *Informe final de Verificación*, en la sección 2.1.

Se agrega como posible mejora el hecho que al planificar, o modificar la planificación ya existente, de un semestre en la web, el cambio no se verá reflejado en la aplicación móvil hasta que no se ingrese a la vista *Planificar Carrera* nuevamente. Viceversa sucede lo mismo.

10. Estructura de la base de datos

A continuación se adjunta un esquema de las principales tablas de la base de datos.