Escuela Técnica Superior de Ingeniería Informática

Grado de Ingeniería Informática
Mención en Tecnologías de la Información

Desarrollo de una aplicación web para la gestión de un laboratorio remoto para prácticas de control automático

Developing a Web App for managing a remote laboratory for practical works of Automatic Control

Realizado por
Mauro D'Agostino

Tutorizado por
D. Vicente Manuel Arévalo Espejo

Departamento
Ingeniería de Sistemas y Automática

Universidad de Málaga
Málaga, Septiembre de 2018

Fecha defensa: Octubre de 2018
El Secretario del Tribunal

Resumen

Español

En este proyecto se describe el desarrollo de una aplicación web que servirá como una herramienta docente para alumnos de diversas carreras. Se trata de una manera de poder realizar ciertas prácticas de control automático a distancia, a través de la aplicación web, eliminando con ello cualquier uso de circuitos o aparatos de medida o generación de señales.

Se propone una infraestructura de bajo coste utilizando las famosas Raspberry Pi y placas Arduinos, así como software gratuito y open-source. El backend está desarrollado con Python, un lenguaje que está de moda, y los frameworks Flask y Jinja. El frontend se ha realizado con HTML, CSS, Bootstrap, JavaScript y jQuery. Resalta la utilización de una librería llamada Spacebrew que sirve para comunicarse con las placas Arduino, encargadas de simular los circuitos y devolver los resultados.

La aplicación ha sido desarrollada de la manera más robusta posible y la experiencia usuario se ha intentado que sea intuitiva. Además, se ha dado importancia al comportamiento responsive de la aplicación web, asegurándose de que se pueda utilizar correctamente en móviles y tablets.

Se ha añadido también una gestión administrativa de los usuarios. Los administradores serán capaces de marcar a los usuarios como activos o inactivos, así como asignarles permisos a unas prácticas u otras.

Todo esto desemboca en una manera elegante y moderna de desplegar la aplicación: Docker. La aplicación se ejecuta a través de un script que lanza la imagen Docker como un contenedor. Se incluye, además, un script para correr MySQL como un contenedor.

El proyecto está explicado con el máximo detalle posible y los pasos para su ejecución han sido simplificados.

Palabras claves Laboratorio remoto, control automático, aplicación web, Flask, Spacebrew, Docker, Nginx

English

In this project is described how to develop a web application that will serve as a teaching tool for students in various degrees. It is a way of being able to perform remotely a set of automatic control practices, via a web app, therefore removing any use of circuits or devices for measuring or generating signals.

We propose a low cost infrastructure using the famous Rapsberry Pi and Arduino boards, as well as free and open-source software. The backend has been developed with Python, a trendy language, and the Flask and Jinja frameworks. The frontend is made up of HTML, CSS, Boostrap, JavaScript and jQuery. We highlight the use of a library called Spacebrew for communicating with the Arduino boards, which are responsible for simulating the circuits and returning the results.

The app has been developed the most robust way possible and we have tried to make the user experience intuitive. Also, we have given importance to the web app's responsive behavior, making sure that it is usable in smartphones and tablets.

An administrative tool for managing users has also been added. The administrators will be able to mark users as active or inactive, as well as giving them permissions for selected practices.

All of this leads to an elegant and modern way of deploying the application: Docker. The app is executed through a script that launches the Docker image as a container. We include, also, a script for running MySQL as a container.

The project is explained with the most possible level of detail and the steps for executing it have been simplified.

Keywords Remote laboratory, automatic control, web application, Flask, Spacebrew, Docker, Nginx

Índice

Capítulo 1

Introducción y visión general

Contenido

1.1. Motivaciones

Es habitual que las carreras de ingeniería tengan al menos una asignatura relacionada con la electrónica. En mi carrera, Ingeniería Informática, tuve que cursarlas en mi primer año. No es de extrañar que sea este año cuando más cantidad de alumnos hay y, por desgracia, estas asignaturas se ven afectadas. Pongo como ejemplo la asignatura Fundamentos de la Electrónica, una de las asignaturas de formación básica que se cursan durante el primer año. Aquí, tenemos que realizar varias prácticas con diversos materiales especiales que la mayoría de los alumnos no tienen en sus casas. Por esto, estamos obligados a tener que asistir al laboratorio. Esto se ve mermado cuando no se pueden completar las prácticas en horario de clase y hay que acudir por la tarde. En horario de clase, los alumnos van divididos de manera ordenada por grupos, pero, por la tarde, es muy común que hayan más alumnos que materiales disponibles.

Otro problema de estas asignaturas es el gran coste de los materiales necesarios. Estos precios varían desde un simple osciloscopio[1], que puede costar unos pocos cientos de euros, hasta una máquina CNC[2], con un coste de decenas de miles de euros.

1.2. Objetivos

Este proyecto pretende buscar una solución a estos problemas, proponiendo una alternativa a los alumnos para realizar una serie de prácticas a través de una aplicación web, sin tener que acudir a los laboratorios. Presenta un modelo que elimina los aparatos de medida y generación de señales mediante el uso de dispositivos hardware de bajo coste y *software open-source*. Esto último se refiere al modelo de desarrollo de *software* basado en la colaboración abierta, en el cual se ofrece acceso al código fuente y gratuidad, entre otras características.

Una primera versión de la aplicación web fue desarrollada hace un par de años, utilizando otras tecnologías, y llegó a utilizarse en la asignatura Automática, perteneciente al Grado de Ingeniería en Tecnologías Industriales [34]. En esta asignatura se introducen los fundamentos del análisis y diseño de sistemas de control para sistemas lineales continuos en el tiempo y se aborda el diseño e implantación de sistemas de automatización. De esta prueba se sacaron conclusiones y críticas para mejorar la experiencia del usuario.

Ahora, el desarrollo de la aplicación web es desde cero basándose experiencia previa, con tecnologías más modernas y teniendo en cuenta los resultados de los alumnos. Los usuarios sugirieron una mejora en el aspecto estético de la aplicación web, así que como que se pueda utilizar en dispositivos móviles. Los administradores, por otro lado, aconsejaron que debe haber una manera adecuada para gestionar los usuarios. Partimos entonces con una lista de requisitos funcionales y no funcionales.

1.2.1. Requisitos funcionales

La siguiente lista de requisitos funcionales fue establecida previo al comienzo del desarrollo de la aplicación.

- La aplicación web debe permitir a los usuarios solicitar la simulación de prácticas predefinidas, pero con valores parametrizables de tiempo y amplitud.

[1] Instrumento de visualización electrónico para la representación gráfica de señales eléctricas que pueden variar en el tiempo.

[2] Una máquina CNC (control numérico computarizado) es una máquina capaz de mover una herramienta al mismo tiempo en los tres ejes para ejecutar trayectorias tridimensionales.

- La aplicación web debe mostrar en una gráfica los resultados de la práctica que haya solicitado el usuario.
- La aplicación web debe permitir que los usuarios puedan visualizar de nuevo resultados de prácticas que hayan solicitado en el pasado.
- La aplicación web podría ser capaz de avisar a los usuarios, via email, cuando los resultados de la simulación solicitada estén disponibles, siempre que el usuario así lo desee. Esto sería el caso cuando un usuario no se queda esperando a que lleguen los resultados (e.g. cierra el navegador, navega a otra página, etc.).
- La aplicación web podría ser capaz de reaccionar a una demanda superior a los recursos disponibles. Esto es, registrar las prácticas pendientes de simular y solicitarlas progresivamente o en lotes cada cierto tiempo.
- La aplicación web debe tener una herramienta para administradores, donde sean capaces de gestionar de manera sencilla los usuarios y los permisos para poder realizar ciertas prácticas.
- La aplicación web debe permitir a los administradores activar o desactivar usuarios. Esto es, permitir que puedan iniciar sesión o no.

1.2.2. Requisitos no funcionales

La siguiente lista son los requisitos no funcionales establecidos para la aplicación.

- La aplicación web debe poseer interfaces gráficas e intuitivas.
- La aplicación web debe tener un comportamiento *responsive*[3], capaz de adaptarse a smartphones o tablets.
- La aplicación web debe proporcionar mensajes de error que sean informativos y orientados al usuario final.

1.3. Herramientas

Ahora, vamos a estudiar brevemente las distintas herramientas que se utilizan durante el desarrollo de este proyecto.

[3]Diseño adaptivo o *responsive*. Corresponde a un comportamiento de la aplicación web en el que, dependiendo del tamaño de la pantalla o del dispositivo en uso, se cambia el tamaño o la colocación del contenido o directamente se muestra un contenido distinto.

Python Lenguaje de programación de moda [6]. Destaca por ser multiparadigma, permitiendo a los desarrolladores optar por un estilo de programación orientado a objetos, programación imperativa, programación funcional o más. Otra característica que presenta es que usa tipado dinámico, o dicho de otra manera, permite que una misma variable puede tomar valores de distinto tipo en distintos momentos.

Flask Es un *framework*[4] minimalista escrito en Python que permite crear aplicaciones web rápidamente y con un mínimo número de líneas de código, sin depender de ninguna otra librería o herramienta. Soporta extensiones que añaden características a nivel de aplicación como si viniesen implementadas en Flask [8].

Jinja Es un motor de plantillas y el utilizado por defecto en Flask [9]. Un motor de plantillas web permite a diseñadores y desarrolladores web trabajar con *plantillas web* para generar automáticamente páginas web personalizadas. Se reutilizan elementos estáticos de la página web mientras se definen elementos dinámicos basados en parámetros de entrada.

SQLAlchemy Se trata de un kit de herramientas *open-source* de SQL y ORM para Python [10]. Un ORM es un mapeo objeto-relacional, proveniente del inglés *object-relational mapping*. Es una técnica de programación para convertir datos entre el sistema de tipos utilizado en un lenguaje de programación orientado a objetos y la utilización de una base de datos relacional como motor de persistencia. En la práctica esto crea una base de datos orientada a objetos virtual, sobre la base de datos relacional.

HTML Del inglés *HyperText Markup Language* (lenguaje de marcas de hipertexto), hace referencia al lenguaje de marcado para la elaboración de páginas web.

JavaScript Es un lenguaje de programación interpretado, es decir, no se compilan previamente a su ejecución, sino que se traducen las instrucciones a medida que van siendo necesarias. Se utiliza principalmente en su forma del lado del cliente (*client-side*), implementado como parte de un navegador web permitiendo mejoras en la interfaz de usuario y páginas web dinámicas.

Chart.js Una librería *open-source* para JavaScript que permite mostrar diversos tipos de gráficas al usuario con un comportamiento *responsive* [16].

Spacebrew Es un kit de herramientas *software* dinámico y abierto para coreografiar espacios interactivos. En otras palabras, una manera simple de conectar cosas interactivas entre ellas. Cada elemento se identifica como un suscriptor (lee datos de entrada) o un publicador (envía datos de salida). Se puede usar

[4]Un *framework* se define como un conjunto de herramientas y código reutilizable cuya funcionalidad puede verse modificada según la especificación del usuario.

una pizarra virtual para conectar o desconectar suscriptores y publicadores entre ellos [1]. Más adelante hablaré más detenidamente de esta herramienta, utilizada como puente de comunicación entre los usuarios de la aplicación web y las placas Arduino, responsables de llevar a cabo las simulaciones de los circuitos y devolver los resultados.

Bootstrap *Framework* de código abierto para diseño de sitios y aplicaciones web [14]. Contiene plantillas de diseño con tipografía, formularios, botones y más basado en HTML y CSS, así como extensiones de JavaScript adicionales.

WWW SQL Designer Herramienta para dibujar y crear esquemas de bases de datos en el navegador [27]. Permite guardar el progreso en un archivo XML y exportar la generación de tablas en un script SQL, entre otras cosas.

MySQL Sistema de gestión de bases de datos relacional [17]. Está considerada como la base datos de código abierto más popular del mundo, y una de las más populares en general junto a Oracle y Microsoft SQL Server, sobre todo para entornos de desarrollo web.

Oracle SQL Developer Herramienta gráfica gratuita para desarrollar sobre bases de datos Oracle [18]. En las últimas versiones ha incorporado mejoras como permitir conectar con bases de datos no Oracle, como MySQL.

Docker Es un proyecto de código abierto capaz de automatizar el despliegue de aplicaciones dentro de contenedores de software [21]. Los contenedores son paquetes de elementos que permiten ejecutar una aplicación determinada en cualquier sistema operativo. Esto nos permite flexibilidad y portabilidad en donde la aplicación se puede ejecutar, ya sea en nuestros servidores, la nube pública, nube privada, etc.

Nginx Pronunciado en inglés "engine X" es un servidor web ligero de alto rendimiento que puede ser utilizado como un proxy inverso[5], balanceador de carga[6], proxy para protocolos de correo electrónico y caché web[7]. Se trata de un *software* gratuito y *open-source*.

[5]Tipo de servidor proxy que recupera recursos en nombre de un cliente desde uno o más servidores. Estos recursos son entonces regresados al cliente como si se originaran en el propio servidor web.

[6]El balanceo de carga es la técnica usada para compartir el trabajo a realizar entre varios procesos, ordenadores, discos u otros recursos.

[7]Caché que almacena documentos web para reducir el ancho de banda consumido, la carga de los servidores y el retardo en la descarga.

PyCharm Quizás el mejor IDE[8] para desarrollar en Python. Ha facilitado enormemente la elaboración de este proyecto, puesto que desde una única aplicación tienes acceso directo a desarrollar código, lanzar una versión de prueba de la aplicación web, ver cambios en el código (via integración con Git), ejecutar comandos en una REPL[9] de Python, ejecutar comandos en una terminal y mucho más.

Git Software de control de versiones gratuito y *open-source*.

Bitbucket Servicio de alojamiento basado en web para los proyectos que utilizan el sistema de control de versiones Mercurial o Git.

1.4. Metodología

La aplicación ha sido desarrollada bajo la metodología SCRUM. En esta, se aplican de manera regular un conjunto de buenas prácticas para trabajar colaborativamente, en equipo, y obtener el mejor resultado posible de un proyecto. Se adopta una estrategia de desarrollo incremental, en lugar de la planificación y ejecución completa del producto. Podemos dividir el desarrollo en cinco etapas:

1. Formación. La primera fase consistirá en un gran esfuerzo por mi parte para aprender a utilizar el software requerido.

2. Investigación. Fase destinada a buscar la mejor manera de desarrollar el proyecto. Principalmente, investigar qué tecnologías resultan más asequibles y qué estructuración del proyecto sería la más profesional.

3. Desarrollo. Desarrollo de la Web App y de las funcionalidades descritas. Irá de la mano con el aprendizaje requerido de las tecnologías a emplear, así como de una continua fase de prueba y validación.

4. Pruebas. Validar que la aplicación se ejecuta bien tanto en ordenadores como en dispositivos táctiles. Comprobar requisitos funcionales y no funcionales.

[8]Un entorno de desarrollo integrado, en inglés *Integrated Development Environment* (IDE), es una aplicación informática que proporciona servicios integrales para facilitarle al desarrollador o programador el desarrollo de software. Consiste de un editor de código fuente, herramientas de construcción automáticas, un depurador y muchas más herramientas que varían entre un IDE u otro.

[9]Un bucle Lectura-Evaluación-Impresión, en inglés *Read-Eval-Print-Loop* (REPL), es un entorno de programación computacional simple e interactivo que toma las entradas individuales del usuario, las evalúa y devuelve el resultado al usuario.

5. Documentación. Redactar la memoria, que servirá también como una guía de uso para los profesores y alumnos que vayan a administrar y manejar, respectivamente, la aplicación.

1.5. Estructura de la memoria

El presente documento se estructura en los siguientes capítulos:

- **Infraestructura y comunicación**. Veremos cuál era el planteamiento original de la infraestructura del proyecto y cómo el uso final de Spacebrew modifica esto.
- **Esquema de la base de datos**. Se explicarán las tablas que conforman la base de datos de la aplicación web.
- **Análisis de la aplicación web**. En este capítulo se describirá la experiencia del usuario a través de las diversas pantallas. Además, se estudiará toda la comunicación que ocurre por detrás entre las diversas partes de la aplicación web.
- **Desplegando la aplicación web**. Aprenderemos a utilizar los scripts creados para preparar y ejecutar el despliegue de la aplicación web, así como todo el desarrollo detrás de ello.
- **Limitaciones de Spacebrew**. Se detallan una serie de problemas que supone utilizar la versión de hoy en día de Spacebrew y cómo se ha conseguido solucionarlos.
- **Conclusiones y futuras mejoras**. Para acabar, veremos las conclusiones y resultados obtenidos tras el desarrollo de la aplicación web. Además, se describen unas mejoras que favorecerían tanto el código de la aplicación web como la experiencia del usuario.

Capítulo 2

Infraestructura y comunicación

Contenido

2.1. Idea inicial

Al adentrarnos en este proyecto, teníamos claro el objetivo de que los alumnos puedan sean capaces de simular prácticas de control automático a través de una aplicación web. Para poder llevar esto a cabo, una investigación previa de tecnologías, llevada a cabo por mi tutor, concluyó en que usaríamos dos tecnologías clave: Spacebrew [1] y Processing [3].

Figura 2.1: Logos de Spacebrew y Processing.

La primera ya ha sido introducida previamente. La segunda, no, porque no entra en el alcance de este proyecto, sino que fue desarrollada por otro alumno. Sin embargo, es necesario conocer y comprender esta extensión del proyecto [35].

2.1.1. Processing

Se trata de un lenguaje de programación basado en Java. En este proyecto, se utiliza como intermediario entre Spacebrew y las placas Arduino encargadas de llevar a cabo las simulaciones.

En la propia página web de Spacebrew existen tutoriales de cómo comunicarse con Spacebrew desde una aplicación desarrollada con Processing [4]. Con esto, era fácil desde este lado del proyecto integrar a Spacebrew la simulación de circuitos en Arduino.

Por lo tanto, lo único que nos hace falta saber es que Spacebrew va a estar escuchando comunicaciones provenientes tanto del lado de la aplicación web como del lado de la aplicación de Processing. Es un puente de comunicación. Los mensajes que se envían desde un lado u otro los veremos más adelante.

2.1.2. Infraestructura inicial

Observemos ahora la figura 2.2 y pensemos en los requisitos funcionales listados previamente.

Con este esquema, cumplimos con que los usuarios puedan solicitar las simulaciones que deseen, siempre que tengan acceso a ello. Podemos ver que Spacebrew va a recibir comunicaciones tanto desde los usuarios, el *front-end*, como desde la aplicación web, el *back-end*. Esto es porque decidimos utilizar la versión en JavaScript de Spacebrew [2], pero, los resultados de las simulaciones tienen que llegar a la aplicación web, y estos vienen desde Spacebrew. Por ello, se pensó en utilizar

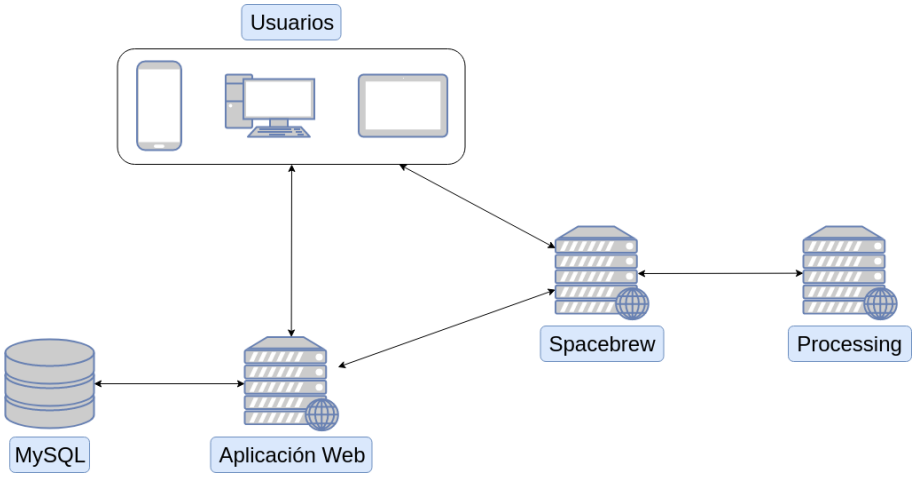

Figura 2.2: Diagrama de la infraestructura inicial.

la versión de Python de Spacebrew para poder comunicarse directamente. Veremos más adelante que esto cambió en la versión final.

La aplicación web tiene comunicación directa con la base de datos MySQL. Esto es para poder persistir los datos de usuarios, permisos y simulaciones. Damos la posibilidad a los alumnos de poder visualizar de nuevo resultados de simulaciones antiguas. También se abre la posibilidad de una gestión de usuarios. Hablaremos después del esquema de la base de datos.

En cuanto a los requisitos no funcionales, vemos que en el cuadro de usuarios hay dispositivos smartphone y tablet. Esto es por el esperado comportamiento *responsive*.

2.1.3. Flujo inicial de datos

El flujo de datos desde que un usuario pide la simulación de la práctica hasta que la recibe estaba pensado tal como se ve en la figura 2.3.

Los pasos enumerados cumplen las siguientes funciones:

1. El usuario, utilizando la versión JavaScript de Spacebrew, envía la petición de simulación de la práctica que haya seleccionado y personalizado con algunos datos de entrada.

2. Spacebrew, haciendo como puente entre el usuario y Processing, hace seguir el mensaje a la aplicación Processing.

3. Processing se comunica con las placas Arduino disponibles para llevar a cabo la simulación y envía a Spacebrew los resultados.

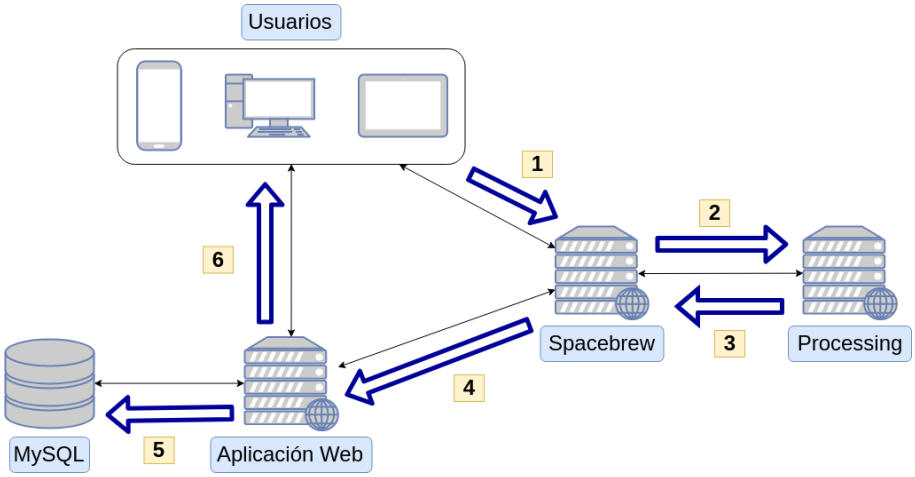

Figura 2.3: Flujo inicial de datos.

4. Spacebrew redirije los resultados a la aplicación web.
5. La aplicación web guarda los resultados en la base de datos.
6. Los resultados son enviados al usuario, para que el *front-end* pueda ocuparse de visualizarlos.

2.2. Cómo funciona Spacebrew

Seguimos avanzando en el planteamiento de la solución a este proyecto. Ahora, tenemos que especificar cómo va a ser la comunicación entre la aplicación web, los usuarios y la aplicación de Processing a través de Spacebrew. Pero, primero tenemos que estudiar el funcionamiento de Spacebrew y las distintas posibilidades que nos brinda.

2.2.1. Spacebrew

Lo primero que tenemos que saber de Spacebrew es que se trata de una aplicación, corriendo bajo un servidor, a la cual se le conectan *clientes*. Estos clientes llevan consigo una serie de *publicadores* y *suscriptores*. Los publicadores son los encargados de enviar datos, mientras que los suscriptores se encargan de recibir datos.

Sin embargo, los suscriptores no envían los datos simplemente al servidor de Spacebrew, sino que se los envían a uno o varios publicadores asignados, es decir, realiza un *broadcast*. De igual manera, los publicadores leen datos de uno o varios suscriptores asignados. Esto es muy relevante, veremos más adelante por qué.

Para poder asignar los enlaces entre suscriptores y publicadores, creamos *rutas* a través de la *pizarra virutal* de Spacebrew. La pizarra virtual, que podemos observar en la figura 2.4, nos sirve como una herramienta gráfica para enlazar o desenlazar clientes que se hayan conectado a nuestro Spacebrew, a través de los suscriptores y publicadores que lleven configurados.

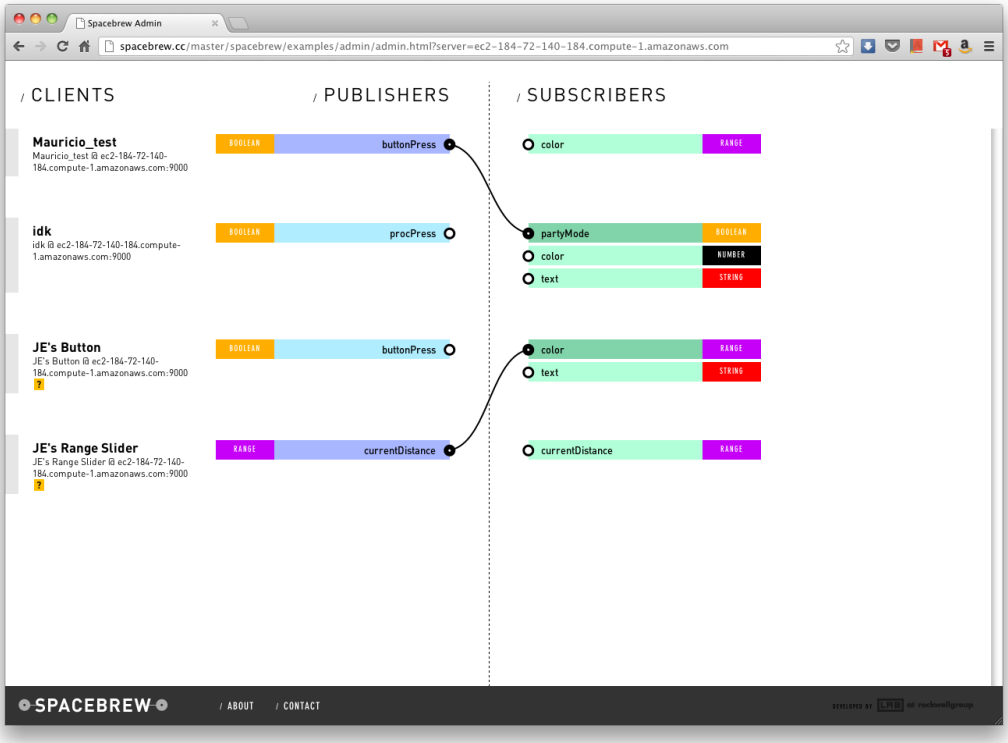

Figura 2.4: Pizarra virtual de Spacebrew. Imagen sacada de la página web oficial.

Lo siguiente a conocer son los tipos de datos que podemos transimitr. En la web de Spacebrew [1], en la sección *about*, se remarca claramente cuáles son las opciones que tenemos disponibles si queremos enviar datos hacia y desde Spacebrew. Si observamos la figura 2.4, podemos ver que los publicadores y los suscriptores tienen a sus laterales una etiqueta que muestra qué tipo de dato envía o recibe. Los datos van en uno de estos cuatros formatos:

boolean Un valor booleano, aceptando valores *true* o *false*.

range Un rango de números enteros, desde 0 hasta 1023, ambos inclusive.

string Una cadena de caracteres.

custom Un formato personalizado a especificar por el desarrollador.

No nos adentramos en la última opción, puesto que decidimos que la comunicación entre aplicaciones se realizaría a través de mensajes JSON [5], los cuales tendríamos que codificar en formato *string* para que puedan ser manipulados por Spacebrew.

2.2.2. Definiendo al cliente

Vamos ahora a ver un código ejemplo de cómo configurar un cliente para Spacebrew, utilizando la librería para JavaScript, en el ejemplo 2.1.

```
/*
Creamos una instancia de la clase Spacebrew. Le pasamos unos
valores configurables.
*/
var sb = new Spacebrew.Client({
        reconnect: true, // Marcamos esto para que nuestra
                         // instancia intente conectarse de nuevo
                         // al servidor Spacebrew en caso de una
                         // desconexión inesperada.
        name: "Nombre del cliente",
        server: "localhost", // El nombre o la IP del servidor
                             // donde se encuentra corriendo el
                             // Spacebrew al que queremos
                             // conectarnos.
        description: "Breve descripción del uso de este cliente.",
        port: 9000 // El puerto a través del cual escucha el
                   // Spacebrew al que queremos conectarnos.
        });

// Ahora añadimos los publicadores y suscriptores de nuestro
// cliente.

// Añadimos un publicador.
sb.addPublish("nombre_del_publicador",
              "string", // Especificamos el tipo de dato que
                        // enviará este publicador.
              "" // Opcionalmente, podemos especificar qué se va
                 // a enviar por defecto.
              );

// Añadimos un suscriptor.
sb.addSubscribe("nombre_del_suscriptor", "string");

// Nos conectamos a Spacebrew.
sb.connect();
```

Ejemplo 2.1: Creación de cliente para Spacebrew.

El resultado de este código lo podemos ver en la pizarra virtual, en la figura 2.5.

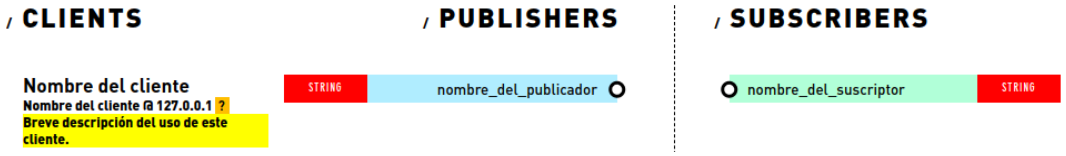

Figura 2.5: Resultado en la pizarra virutal del ejemplo 2.1.

La aplicación de Processing va a crear un cliente y va a ofrecer un publicador y un suscriptor. Del lado de la aplicación web se van a crear diversos clientes y cada uno de ellos va a tener su propia pareja de publicador y suscriptor. Cada cliente que se cree del lado de la aplicación web debe conectarse con el cliente de la aplicación Processing, a través de las rutas de Spacebrew.

Sabemos que tenemos la pizarra virtual para crear rutas entre suscriptores y publicadores. Pero, cuando nuestra aplicación esté corriendo y hayan múltiples usuarios conectándose, no podemos estar continuamente creando las rutas manualmente. Necesitamos una manera de automatizar este proceso.

2.2.3. Spacebrew Admin

La librería para JavaScript lleva consigo un *mix-in*[1] llamado Spacebrew Admin. Este mix-in nos da la posibilidad, entre otras, de crear y eliminar rutas de manera automática. Podemos ver en el ejemplo 2.2 un ejemplo de cómo se crean.

```
1  /*
2  Primero tenemos que añadirle el mix-in a nuestra instancia
3  Spacebrew creada previamente.
4  */
5  sb.extend(Spacebrew.Admin);
6
7  /*
8  Creación de una ruta de un publicador de mi cliente a un suscriptor
9  de otro cliente.
10 */
11 sb.addRoute(
12     "nombre_de_mi_cliente",
13     "localhost", // Nombre del servidor o IP desde el cual se ha
14                  // conectado mi cliente a Spacebrew.
15     "nombre_de_mi_publicador",
16     "nombre_del_otro_cliente",
17     "12.3.4.56", // Nombre del servidor o IP desde el cual se
18                  // conectó el otro cliente a Spacebrew.
```

[1]En los lenguajes de programación orientada a objetos, un mix-in es una clase que ofrece cierta funcionalidad para ser heredada por una subclase, pero no está ideada para ser autónoma. Heredar de un mix-in no es una forma de especialización sino más bien un medio de obtener funcionalidad.

```
    "nombre_del_suscriptor_del_otro_cliente"
    );

/*
Creación de una ruta de un suscriptor de mi cliente a un publicador
de otro cliente.
*/
sb.addRoute(
    "nombre_del_otro_cliente",
    "12.3.4.56",
    "nombre_del_publicador_del_otro_cliente"
    "nombre_de_mi_cliente",
    "localhost",
    "nombre_de_mi_suscriptor"
    );
```

Ejemplo 2.2: Creaciones de rutas.

Vemos como la función que crea estas rutas necesita que los primeros argumentos sean acorde al publicador y los siguientes acorde al suscriptor.

El único requerimiento que necesita esta función es que existan previamente todos los clientes, publicadores y suscriptores involucrados. Necesitamos una manera de saber esto previo a llamar la función. Para esto, hacemos uso de los *eventos* de Spacebrew.

2.2.4. Reaccionando a eventos en Spacebrew

Spacebrew nos da la posibilidad de configurar cómo reaccionamos a una serie de eventos. El mix-in nos ofrece aún más eventos. La manera de configurar estas reacciones es pasando el nombre de una función que queremos que se ejecute cuando se produzca ese evento. Aquí voy a a hablar los que nos interesan, la lista completa se puede encontrar en la página de github [2].

Uno de los eventos disponibles es la llegada de un mensaje a uno de nuestros suscriptores. Al igual que los suscriptores se dividen por el tipo de dato que reciben, este tipo de evento también tiene uno especializado para cada tipo de dato. Para nuestro caso, nos interesa la llegada de un dato de tipo *string*, que traerá consigo un JSON codificado. El evento se llama *onStringMessage*. Veamos un ejemplo sobre cómo configurar la reacción a este evento, en el ejemplo 2.3.

```
/*
Especificamos a nuestra instancia Spacebrew creada previamente el
nombre de la función a ejecutar cuando ocurra este evento.
*/
sb.onStringMessage = hacerAlgoConString;

/*
```

```
/*
Nuestra función para reaccionar a este evento siempre va a recibir
dos parámetros de entrada:
- El nombre del suscriptor que está reaccionando al evento.
- Los datos, en este caso un string, que le llegan.
*/
function hacerAlgoConString(nombre_suscriptor, datos) {
  // Si queremos manipular un JSON, tenemos que decodificarlo.
  objetoJson = JSON.parse(datos);
}
```

Ejemplo 2.3: Reaccionando a la llegada de un mensaje de tipo *string*.

El siguiente que nos interesa es uno de los que nos brinda el mix-in Admin: *onNewClient*. Traducido literalmente "en cliente nuevo", es un evento que ocurre cuando nuestra instancia Admin se le notifica que se ha conectado un nuevo cliente a Spacebrew. Y es que, al añadirle el mix-in Admin a un cliente, este cliente se entera de conexiones, desconexiones y modificaciones de cualquier otro cliente en Spacebrew, así como creaciones, eliminaciones o modificaciones de rutas. El uso que le sacamos a este evento lo podemos ver en el ejemplo 2.4.

```
/*
Especificamos a nuestra instancia Spacebrew creada previamente, y a
la cual le hemos añadido el mix-in Admin, el nombre de la función a
ejecutar cuando ocurra este evento.
*/
sb.onNewClient = intentarCrearRuta;

/*
Nuestra función para reaccionar a este evento siempre va a recibir
un único parámetro de entrada:
- Un objeto con información sobre el cliente que se ha conectado.
  Esta descripción está sacada del código fuente del mix-in:
  - {String} name         Nombre del cliente
  - {String} address      Dirección IP del cliente
  - {String} description  Descripción del cliente
  - {Array} publish       Un array con los publicadores del cliente
  - {Array} subscribe     Un array con los suscriptores del cliente
*/
function intentarCrearRuta(cliente) {
  /*
  Buscamos en Spacebrew los clientes que nos interesan. Utilizamos
  la función getClient, proveniente del mix-in. Esta función recibe
  como parámetros el nombre y la dirección del IP del cliente que
  buscamos.
  */
  var miCliente = sb.getClient("nombre_mi_cliente",
                               "12.3.4.56");
  var otroCliente = sb.getClient("nombre_otro_cliente",
                                 "78.9.10.11");
```

```
30
31    /*
32    Cuando no existen los clientes, la función nos devuelve como
33    resultado undefined. Tenemos que asegurarnos de que existen ambos
34    clientes.
35    */
36    if (miCliente != undefined && otroCliente != undefined) {
37      /*
38      Y ahora somos libres de crear la ruta como se ha visto en
39      un anterior ejemplo.
40      */
41    }
42  }
```

Ejemplo 2.4: Reaccionando a un cliente nuevo e intentando crear nuestras rutas deseadas.

Observamos que en el ejemplo 2.4 no utilizamos el parámetro *cliente* en la función *intentarCrearRuta*. Esto es porque puede que el cliente que se conecte a nosotros no siempre sea el que queremos usar para crear nuestra ruta, y la función *getClient* nos facilita esta comprobación.

2.3. Definiendo la comunicación por Spacebrew

Una vez estudiadas y analizadas las capacidades de Spacebrew, podemos proceder a especificar los mensajes que va a enviar la aplicación web y los mensajes que va a recibir de la aplicación de Processing. Esta especificación fue establecida en una reunión con el desarrollador de la aplicación de Processing [35].

Vamos a dividir esta sección primero en los prerrequisitos necesarios y luego en los mensajes a enviar para los distintos casos de uso de la aplicación web.

2.3.1. Nombres de clientes, publicadores y suscriptores

Como hemos visto en algunos ejemplos, tenemos que especificar estos nombres cuando queremos crear rutas. Por lo tanto, es importante que se mantengan fijos. Podemos pensar en la combinación cliente-publicador y cliente-suscriptor como un *endpoint*[2].

Los clientes con el mix-in de Spacebrew Admin van a estar únicamente del lado de la aplicación web, y son estos quienes van a crear las rutas para la comunicación la aplicación de Processing. Por lo tanto, esta última sólo se tiene que preocupar

[2]Los *endpoints* son las URLs de un API o un backend que responden a una petición. No están pensados para interactuar con el usuario final, usualmente sólo devolverán JSON o nada.

de conectarse como un cliente normal, pero, con unos nombres fijos. Los nombres elegidos fueron:

- *Arduino Simulator* como nombre del cliente.
- *arduino_result* como nombre del publicador, encargado de devolver los resultados de las simulaciones.
- *arduino_request* como nombre del suscriptor, encargado de procesar las solicitudes de simulación.

Los nombres de los clientes resultantes de la aplicación web varían según el usuario, pero los suscriptores y publicadores no:

- *<Username>* como nombre del cliente. Cada usuario crea su cliente y, como cada usuario tendrá un nombre de usuario, o *username*, único, nos aseguramos que este cliente tendrá un cliente único.
- *arduino_command* como nombre del publicador, encargado de crear y enviar la solicitud de una simulación.
- *arduino_reception* como nombre del suscriptor, encargado de procesar los resultados de la simulación.

Podemos observar un ejemplo del esquema resultante de la comunicación entre aplicaciones en la figura 2.6.

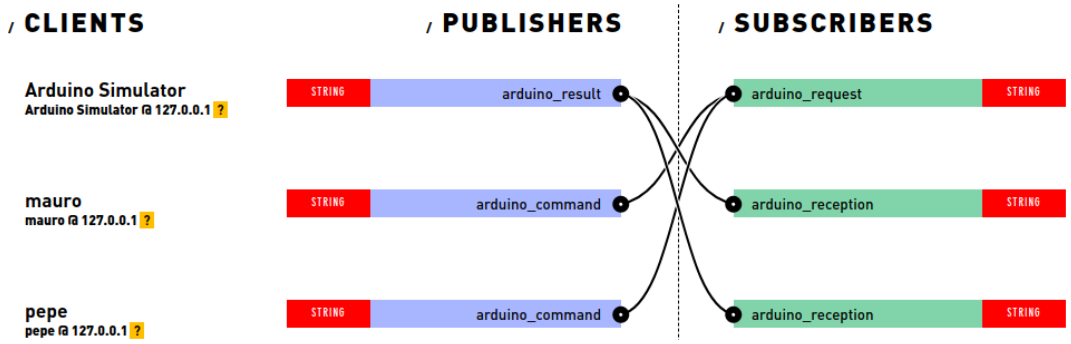

Figura 2.6: Comunicación entre clientes de las dos aplicaciones.

2.3.2. Pidiendo una simulación personalizada

Esto se entiende como el caso de uso mostrado en la figura 2.3 de la página 18. Pongamos atención ahora al ejemplo 2.5.

```
{
  "id": "0123456789ABCDEFGHIJKLMNOPQRSTUV",
  "subtipo": "a",
  "tiempo": 15,
  "amplitud": 1
}
```

Ejemplo 2.5: Petición de simulación.

Esta es la plantilla del mensaje JSON que enviamos desde la aplicación web hacia la aplicación de Processing. Los nombres de las claves y los tipos de datos de cada una tienen que estar en sincronización entre ambas aplicaciones, para que se pueda procesar el mensaje correctamente. Vamos a ver qué significa el valor de cada clave:

id Se trata de una cadena de 32 caracteres hexadecimales única y estrictamente en mayúscula. Es generado por la aplicación web singularmente para cada simulación que solicitan los usuarios.

subtipo Por el lado de Processing se definen dos *tipos* de circuitos: RC y RLC. Cada uno tiene distintos *subtipos*, en los que varían la fuerza de sus componentes. El nombre del subtipo es un string y nos lo dice la aplicación de Processing.

Un circuito RC es un circuito eléctrico compuesto de resistencias y condensadores, su forma más simple es el circuito RC de primer orden, compuesto por una resistencia y un condensador. Por otro lado, un circuito RLC es un circuito eléctrico que contiene una resistencia, una bobina y un condensador. Estas son las dos únicas prácticas que tenemos implementadas ahora mismo.

tiempo El primero de los valores configurables por el alumno solicitando la simulación; corresponde a los milisegundos de la simulación. Se trata de un número entero, que tiene que estar entre 2 y 500, ambos incluidos.

amplitud El segundo valor configurable y corresponde a la amplitud usada en la simulación. Es otro número entero en un rango predefinido: entre 0 y 500, ambos incluidos. Este rango y el anterior fueron establecidos antes del comienzo del proyecto, pero, pueden variar en el futuro dependiendo del tipo de práctica que se pretenda simular.

Más adelante veremos cómo se genera el campo "id" y cómo conseguimos la información de los "subtipos" y los rangos para los campos "tiempo" y "amplitud".

El campo "id" tiene un uso primordial en la comunicación entre las aplicaciones. Fijémonos de nuevo en la figura 2.6 de la página 25. Vemos que todos los suscriptores de los clientes de los usuarios están conectados al publicador único de la aplicación de

Processing. Como se ha mencionado antes, esto implica que todos estos suscriptores van a recibir cualquier mensaje que envíe este publicador, puesto que los publicadores hacen un *broadcast*. Por lo tanto, todos los clientes de los usuarios van a recibir los resultados de todas las simulaciones que realice Processing, sean o no los que desee cada usuario. El "id" sirve para diferenciar los mensajes entrantes; es importante que sea único para todas las simulaciones solicitadas.

Los resultados de las simulaciones vienen en un formato como el que aparece en el ejemplo 2.6.

```
{
  "id": "0123456789ABCDEFGHIJKLMNOPQRSTUV",
  "resultados" : [
    {
        "tiempo": 0,
        "valor": 0.02929732204165713
    },
    {
      "tiempo": 3,
      "valor": 0.03418020904859999
    },
    {
      "tiempo": 6,
      "valor": 0.05859464408331426
    }
  ]
}
```

Ejemplo 2.6: Resultados de una simulación.

Vemos presente, de nuevo, el importante campo "id" y los resultados, que vienen en el campo "resultados" y se componen como una lista de objetos con los campos "tiempo" y "valor", ambos con valores numéricos. Estos resultados son mostrados al usuario en una gráfica para que pueda proceder a estudiarlos. También se persisten en la base de datos, para que los alumnos puedan visualizarlos de nuevo.

Sin embargo, que un usuario solicite la práctica y simplemente espere a los resultados puede dar para una experiencia pobre cuando hay algún problema desde el lado de la aplicación de Processing. Por eso, existen dos respuestas más a las solicitudes.

La primera, de la cual podemos ver un ejemplo en el ejemplo 2.7, es una respuesta para demostrar que la solicitud de una simulación ha sido aceptada. Esto nos permite, por ejemplo, mostrarle un mensaje al usuario de que la simulación está siendo llevada a cabo y recibirá resultados. Esta respuesta debería ser inmediata por parte de la aplicación Processing y el cliente que haya solicitado una simulación esperaría después la respuesta con los resultados.

```
{
```

```
  "id": "0123456789ABCDEFGHIJKLMNOPQRSTUV"
}
```

Ejemplo 2.7: Respuesta de solicitud aceptada.

El ejemplo 2.8 corresponde a un ejemplo de mensaje que nos enviaría la aplicación de Processing en caso de que hubiese algún error en la solicitud o a la hora de llevar a cabo la simulación, e.g. sobrecarga de peticiones.

```
{
  "id": "0123456789ABCDEFGHIJKLMNOPQRSTUV",
  "error": "mensaje"
}
```

Ejemplo 2.8: Respuesta de error en la solicitud.

El valor del campo "error" describe por qué ha ocurrido el error. Esta respuesta nos permite mostrarle al usuario que su solicitud ha fallado y darle los motivos detrás de ello.

2.3.3. Pidiendo simulaciones disponibles

Existe un conjunto de información que sólo está disponible en la aplicación de Processing. Necesitamos obtener esta información para poder ofrecerle al usuario una experiencia buena e intuitiva.

La información que necesitamos es: los tipos de simulaciones, los subtipos de cada tipo y los valores usados para las componentes de cada subtipo, i.e. bobina, resistencia y condensador. Vemos en el ejemplo 2.9 cómo la pedimos.

```
{
  "id": "123"
}
```

Ejemplo 2.9: Petición de información de Processing.

El campo "id" aquí tiene la misma finalidad que en el caso anterior, sólo que no corresponde al identificador de una simulación. En este caso va a corresponder al identificador único del usuario en la base de datos. Este identificador es realmente un número, pero por conveniencia se convierte a un string para que el manejo del mensaje sea parecido al anterior caso.

Veamos ahora cómo es la respuesta de la aplicación de Processing: ejemplo 2.10.

```
{
  "id": "123",
  "config": [
    {
      "tipo": "RLC",
      "variantes": [
        {
```

```
8             "subtipo": "a",
9             "tiempoRecomendado": "0",
10            "R": "0",
11            "L": "0.0",
12            "C": "0.0"
13          },
14          {
15            "subtipo": "b",
16            "tiempoRecomendado": "0",
17            "R": "0",
18            "L": "0.0",
19            "C": "0.0"
20          }
21        ]
22      },
23      {
24        "tipo": "RC",
25        "variantes": [
26          {
27            "subtipo": "c",
28            "tiempoRecomendado": "0",
29            "R": "0",
30            "C": "0.0"
31          },
32          {
33            "subtipo": "d",
34            "tiempoRecomendado": "0",
35            "R": "0",
36            "C": "0.0"
37          }
38        ]
39      }
40    ]
41  }
```

Ejemplo 2.10: Respuesta de información de Processing.

Nos centramos en el campo "config". Este campo contiene una lista de objetos, y cada objeto corresponde a un tipo de práctica, dándonos información del nombre del tipo, a través del campo "tipo", y de los subtipos disponibles, a través del campo "variantes".

El campo "variantes" incluye una lista de objetos, cada uno perteneciente a cada subtipo disponible. Cada uno nos dice el nombre del subtipo, a través del campo "subtipo", y los valores de sus componentes, a través de los campos "R", "C" y, si corresponde, "L". Además, el campo "tiempoRecomendado" se refiere al tiempo recomendado para llevar a cabo la simulación y obtener mejores resultados.

Los valores de los campos "tiempoRecomendado", "R", "L" y "C" vienen en milisegundos (ms), ohmios (Ω), milihenrios (mH) y microfaradios (µF), respectivamente.

2.4. Infraestructura final

Después de haber estudiado a fondo las posibilidades de Spacebrew y haber establecido los mensajes que fluirán desde la aplicación web y la aplicación de Processing y viceversa, tenemos que revisar si va acorde a la infraestructura inicial.

Primero, vamos a mirar de nuevo la figura 2.2 en la página 17. La comunicación de la aplicación web con la base de datos y los usuarios, o el *front-end*, es completamente imprescindible. Sin embargo, si nos ponemos a mirar en la definición de los mensajes para las distintas peticiones a la aplicación de Processing, vemos que realmente no interviene la aplicación web en ninguna.

Puesto que necesitamos unos clientes únicos para cada usuario utilizando la aplicación web, van a ser ellos mismos, a través de la librería de Spacebrew para JavaScript (la cual se ha utilizado en todos los ejemplos), quienes se encarguen de crear los clientes, suscriptores, publicadores y rutas. Por lo tanto es necesario que cada cliente lleve consigo el mix-in Admin. También van a ser ellos mismos quienes manejen las peticiones a la aplicación de Processing.

Podemos ver una revisión de la figura 2.2 de la página 17 y de la figura 2.3 de la página 18 en las figuras 2.7 y 2.8, respectivamente.

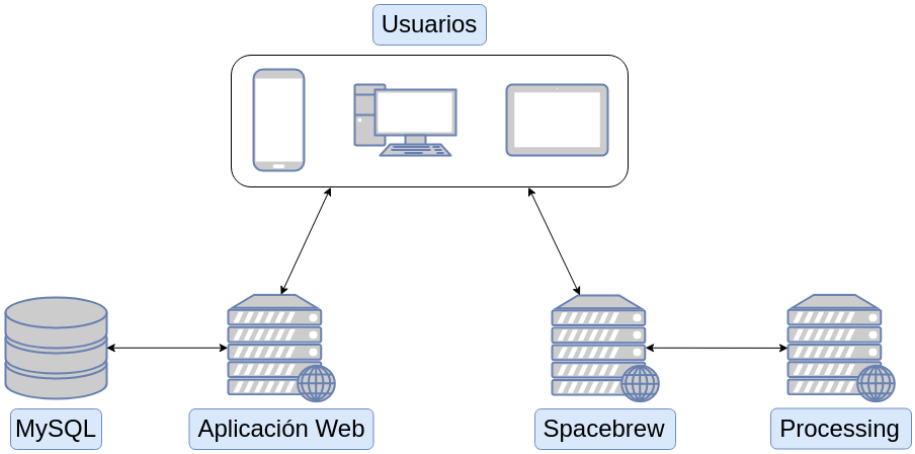

Figura 2.7: Diagrama de la infraestructura final.

Después del paso 4, el *front-end* ya tiene disponible los resultados para poder visualizarlos. Los pasos 5 y 6 son para persistir los resultados para futuras revisualizaciones, pero estos pasos son asíncronos e invisibles para el usuario.

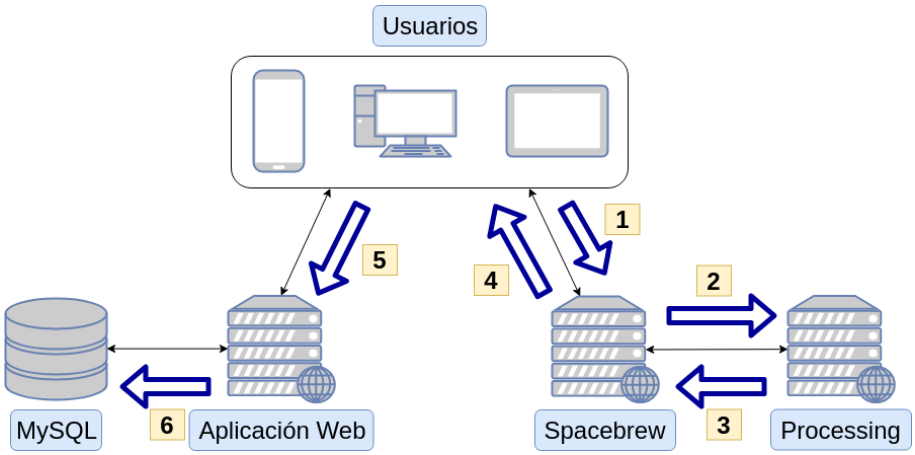

Figura 2.8: Flujo final de datos.

Capítulo 3

Esquema de la base de datos

Contenido

3.1. Revisión de requisitos

Antes de nada, tenemos que recordar el uso que va a tener la aplicación web y los requisitos que lleva consigo.

Para empezar, existen usuarios, lo cual implica que guardemos datos de ellos para que puedan volver a acceder a la aplicación web. Los usuarios se registran a través de un nombre de usuario, un correo electrónico y una contraseña.

Lo siguiente es darle a los usuarios la capacidad de volver a visualizar resultados de simulaciones solicitadas en el pasado. Para esto, tenemos que guardar los resultados generados, junto con el identificador único, el usuario que lo solicitó y otras propiedades que definen a una única simulación.

Por último, queremos que exista un división entre usuario normal y administrador. Para esto, necesitamos tener la definición de *rol*, que llevan consigo *permisos*. Además, los administradores pueden desactivar o activar a usuarios, por lo tanto los usuarios necesitan llevar consigo esta propiedad.

3.2. Esquema resultante

Queda evidente que necesitamos tres tablas: una para la información de los usuarios, otra para información de los permisos y otra para la información de las simulaciones. Sin embargo, puesto que un usuario puede tener varios permisos y un permiso puede pertenecer a varios usuarios, se crea una relación N a N, y, para ello, necesitamos una tabla extra para guardar estas relaciones.

Estas conclusiones nos llevan a crear el esquema tal como se observa en la figura 3.1. El diagrama ha sido creado con la herramienta WWW SQL Designer [27].

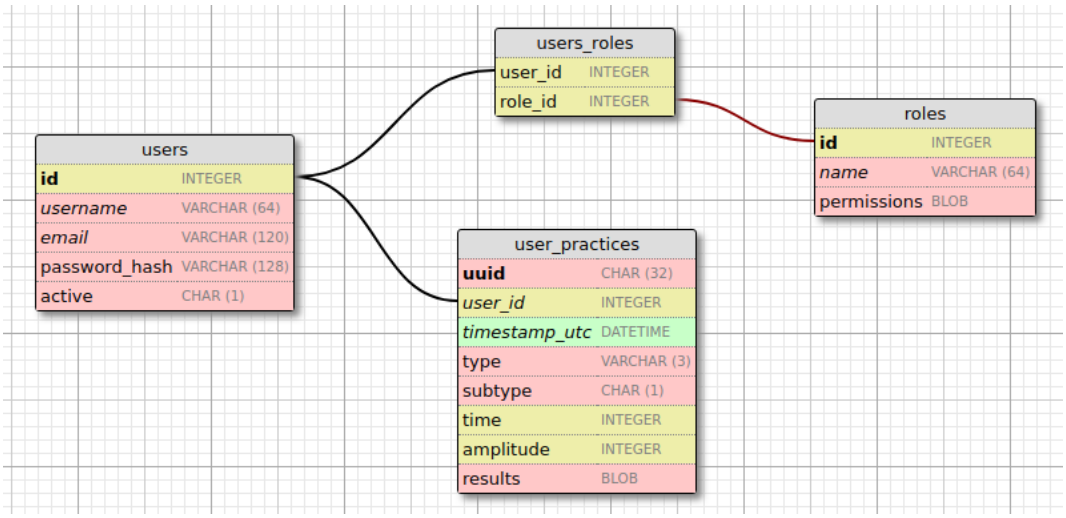

Figura 3.1: Esquema de la base de datos.

Puesto que este diagrama no es visualmente tan explicativo como podrían ser otros, voy a explicar más detenidamente cada tabla.

3.2.1. Usuarios

La tabla *users* guarda información sobre cada usuario que se haya registrado en la aplicación. El campo "id" es el identificador único para cada usuario y lo mantiene automáticamente el motor de la base de datos, MySQL. Cada vez que se añade una fila para un usuario nuevo en esta tabla, se autoincrementa este valor. Es, por tanto, la clave primaria de la tabla.

Los campos "username" e "email" guardan el nombre de usuario y el correo electrónico con el que se registra cada usuario. Están limitados a 64 y 120 caracteres, respectivamente. Además, no pueden haber dos usuarios con el mismo nombre de usuario o con el mismo correo electrónico.

El campo "password_hash" es una versión encriptada de la contraseña del usuario, con un límite de 128 caracteres, y, por último, el campo "active" un campo que acepta únicamente los caracteres *N* o *Y*. Estos caracteres significan que el usuario está desactivado o activado, respectivamente.

Veremos más adelante cómo la aplicación web maneja estas restricciones y comprobaciones.

3.2.2. Roles

La tabla *roles* mantiene los distintos roles que se han creado en la aplicación. Cada rol tiene asignado un "id", que funciona igual que en la tabla anterior, y unos permisos, bajo el campo "permissions", guardados como un *BLOB*[1]. Esto último es porque no se pretende que el sistema haga búsquedas por valores en este campo, así que sacamos ventaja de esto y persistimos en BLOB, que ocupa menos que una cadena de caracteres. Más adelante veremos cómo la aplicación web estructura este campo, una vez reconvertidos del binario.

El campo "name" se refiere al nombre del rol y ha de ser único en el sistema. Nos aseguramos de esto guardando el nombre siempre en minúscula.

Ahora podemos hablar de la tabla *users_roles*, que no es más que una tabla auxiliar para relacionar usuarios con sus permisos. Se compone de dos campos, "user_id" y "role_id", que son claves foráneas apuntando a los campos "id" de la tabla *users* e "id" de la tabla *roles*, respectivamente.

3.2.3. Simulaciones

Por último, vamos a ver la tabla *user_practices*. Conviene explicar los campos más detenidamente:

"uuid" Es la clave primaria de esta tabla. Se trata de un identificador único universal[2] que genera la aplicación web a través de distintos datos que hacen que una simulación sea única respecto al resto. Este es el "id" que luego viaja por los mensajes a través de Spacebrew. Para optimizar el almacenamiento de este campo, lo hemos especificado como CHAR de 32 caracteres, puesto que sabemos que su longitud nunca va a variar.

"user_id" Es una clave foránea apuntando al "id" de la tabla *users*. Cada simulación va asociada a un usuario y cada usuario puede tener múltiples prácticas

[1] Los BLOB, del inglés *Binary Large Objects*, en español objetos binarios grandes, son elementos utilizados en las bases de datos para almacenar datos de gran tamaño que cambian de forma dinámica.

[2] Un UUID, del inglés *universally unique identifier*, es un número de 16 bytes. Se puede expresar como una cadena de 32 dígitos hexadecimales.

(relación 1 a N).

"timestamp_utc" El *timestamp*[3] correspondiente a cuando la simulación fue solicitada por el usuario. Por consistencia, la convertimos a UTC. La apliacación web se encargará de reconvertirla a la zona horaria del usuario. Este campo se ha añadido por el hecho de que un usuario puede solicitar la misma simulación más de una vez, pero no garantizan los mismos resultados. Con este campo, tenemos un registro por cada simulación, y el usuario puede visualizar todos los resultados.

"type" y "subtype" Estos campos se refieren al tipo y subtipo de la simulación. Estos datos no existen en la base de datos, tenemos que solicitárselos a la aplicación de Processing, pero tenemos que guardarlos de alguna manera, ya que hacen única a cada simulación. Se guardan como cadenas de caracteres y tomamos ventaja de que el subtipo, de momento, va a ser siempre una cadena de un caracter, así que lo establecemos como un CHAR de longitud 1 para optimizar el campo.

En un futuro, sería conveniente que existiese esta información en la base de datos, e.g. bajo unas nuevas tablas *types* y *subtypes*. Sin embargo, esto implicaría posiblemente un esfuerzo mayor por el lado de la aplicación de Processing que por el lado de la aplicación web.

"time" y "amplitude" Se refieren a los valores parametrizables que tiene a disposición el usuario: el tiempo y la amplitud.

"results" Los resultados de la simulación. En el ejemplo 2.6 de la página 27 hemos visto cómo vienen estructurados los resultados. En el uso real de la aplicación, el JSON es mucho más largo, pudiendo ocupar más de 3000 líneas de código. Por tanto, puesto que el sistema no tiene pensado hacer búsquedas sobre este campo, los resultados se serializan a binario y se almacenan como BLOB.

Para optimizar las búsquedas por esta tabla filtradas por usuario y por fecha, se han creado índices sobre los campos "user_id" "timestamp_utc". Se espera que esta búsqueda se haga a menudo, puesto que realiza para la página donde se muestran todas las simulaciones que haya solicitado el usuario.

3.3. Generando el esquema

El código fuente del proyecto viene con un *script* (un archivo de órdenes) MySQL encargado de crear tablas, restricciones, claves foráneas, etc. Después de crear todo

[3]Un timestamp, o sello de tiempo, es una secuencia de caracteres que denotan la hora y fecha en la que ocurrió un determinado evento.

el esquema, introduce en él un primer usuario, llamado “admin”, con una contraseña preestablecida (la cual le será facilitada a mi tutor), un primer rol, llamado “admin” también, y un enlace entre este usuario y el rol. La aplicación web espera que este rol esté presente en la base de datos y entiende que cualquier usuario que tenga este rol va a tener acceso a todo.

El archivo en cuestión se encuentra en *webserver/datamodels/tables.sql*.

Capítulo 4

Diseño e implementación de la aplicación web

Contenido

4.1. Primeras pantallas

Vamos a empezar a ver las distintas pantallas que se va a encontrar el usuario, por orden de aparición.

4.1.1. Iniciar sesión

Cuando un usuario entra a la página web por primera vez, lo primero que se encuentra es la página para iniciar sesión, que se puede apreciar en la figura 4.1.

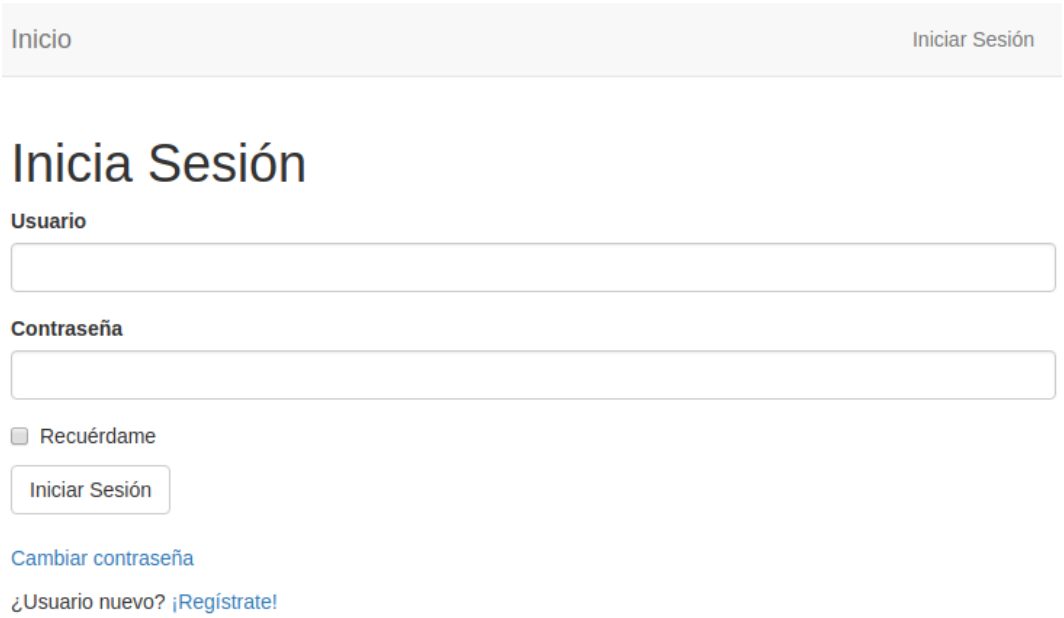

Figura 4.1: Página para iniciar sesión.

En lo alto de la página vemos la barra de navegación, que incluye botones para ir a la página "Inicio", a la cual no se puede acceder sin haber iniciado sesión, y a la página para iniciar sesión (esta misma). Debajo vemos campos de texto para introducir el nombre de usuario y la contraseña. Para introducir los datos, podemos pulsar el botón "Iniciar Sesión" o la tecla *Enter* en nuestro teclado. Existe la posibilidad de seleccionar la casilla de verificación "Recuérdame", para que la próxima vez que el usuario acceda a la página web sea redirigido automáticamente a la página "Inicio".

En la parte baja de la página de la página se encuentran dos enlaces para dirigirnos a páginas para cambiar nuestra contraseña o registrarnos, respectivamente.

La página web muestra una serie de mensajes informativos para varios casos:

- Intento de acceder a la página "Inicio" sin haber iniciado sesión.
- Intento de iniciar sesión con un usuario que no ha sido activado por los administradores.
- Intento de iniciar sesión con un usuario que no ha sido registrado.
- Intento de iniciar sesión con una contraseña equivocada.

- Intento de iniciar sesión sin haber rellenado alguno de los campos necesarios.

4.1.2. Registrarse

La siguiente página que visitaría un alumno sería la página para registrarse, en caso de que no lo haya hecho previamente. Podemos verla en la figura 4.2.

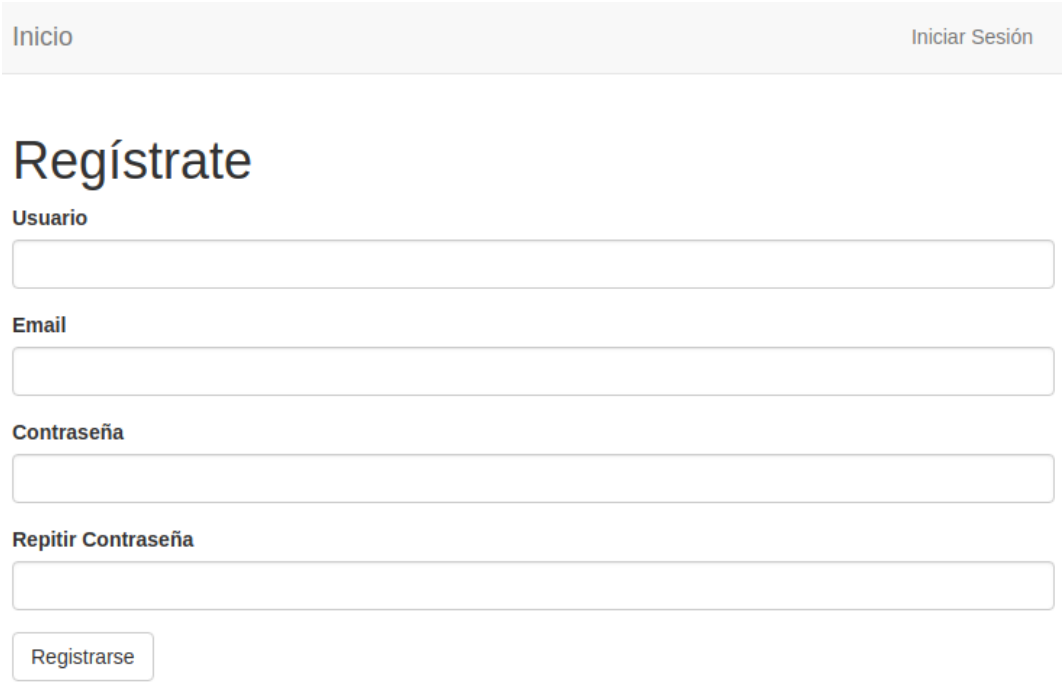

Figura 4.2: Página para registrarse.

Se presenta como un formulario bastante simple. El usuario tiene que especificar un nombre de usuario, un email y una contraseña, asegurándose de que la haya escrito bien. Al igual que la vista anterior, presenta mensajes informativos en caso de:

- Intentar registrar de nuevo un nombre de usuario o correo electrónico.
- No se han rellenado todos los campos.
- Las contraseñas no coinciden.

Hay que mencionar que cuando un usuario se registra, no puede acceder instantáneamente a la página web. El usuario se encuentra desactivado hasta que un administrador lo active a través de la página para gestionar usuarios.

Vamos a ver parte del código del *backend* encargado de hacer las comprobaciones del formulario.

Comprobando nombre de usuario y correo electrónico

El sistema no permite que existan dos usuarios o bien con el mismo nombre de usuario o bien con el mismo correo electrónico. Para lograr esto, el *backend* siempre realiza las comparaciones en minúscula para ambos campos. Sin embargo, a la hora de insertar los datos a la base de datos, el nombre de usuario se añade tal cual haya sido escrito por el usuario, pero, el correo electrónico se añade siempre en minúscula. Esto es así porque si alguien desea registrarse como *Pepe*, no queremos forzarle a que luego se tenga que conectar como *pepe*.

En el ejemplo 4.1 podemos ver código real de la aplicación web, aunque ha sido adaptado un poco para entrar en las dimensiones del folio. Forma parte del fichero *webserver/app/auth/forms.py*.

```
class RegistrationForm(FlaskForm):
    username = StringField('Usuario', validators=[DataRequired()])
    email = StringField('Email',
                        validators=[DataRequired(), Email()])

    # ...

    def validate_username(self, username):
        # Usernames are compared in lowercase.
        user = User.query.filter(
            func.lower(User.username) == func.lower(username.data)
        ).first()
        if user is not None:
            raise ValidationError(
                'El nombre de usuario ya existe.')

    def validate_email(self, email):
        # Emails are only in lowercase.
        user = User.query.filter_by(
            email=email.data.lower()).first()
        if user is not None:
            raise ValidationError('El email ya está siendo usado.')
```

Ejemplo 4.1: Definición de formulario de registro.

De este código lo importante es que se está creando una clase heredada de *FlaskForm*, que luego se puede renderizar fácilmente como un formulario web, con una serie de campos a introducir y validaciones a ejecutar para cada uno de ellos. En este caso, tenemos los campos *username* e *email* para los cuales creamos nuestras propias validaciones llamadas *validate_ username* y *validate_ email*.

Para la primera, puesto que el nombre de usuario se introduce tal cual lo escriba el usuario, tenemos que hacer la búsqueda en la base de datos asegurándonos de que convertimos a minúscula tanto los nombres de usuario en la base de datos como el nombre de usuario introducido en el formulario antes de hacer el filtrado por

comprobación de strings.

Para el segundo caso, basta que convirtamos el email introducido en el formulario a minúscula antes de hacer la búsqueda filtrada, ya que todos los emails en la base de datos van a estar en minúscula.

Guardando contraseñas de manera segura

En la figura 3.1 de la página 34 hemos visto que la tabla *users* contenía el campo "password_hash", la cual era una versión encriptada de la contraseña del usuario. Podemos ver en el ejemplo 4.2 parte del código del fichero *webserver/app/models.py*, en especial: la parte encargada de encriptar las contraseñas.

```
1  from werkzeug.security import generate_password_hash,\
2      check_password_hash
3
4  # ...
5
6  class User(UserMixin, db.Model):
7
8      # ...
9
10     password_hash = db.Column(db.String(128), nullable=False)
11
12     # ...
13
14     def set_password(self, password):
15         self.password_hash = generate_password_hash(password)
16
17     def check_password(self, password):
18         return check_password_hash(self.password_hash, password)
```

Ejemplo 4.2: Definición de formulario de registro.

La clase *User* es una representación de la tabla *users* de la base de datos y contiene una variable "password_hash". Esta variable se inicia a través de la función *set_password*, que a su vez llama a *generate_password_hash* con la contraseña, en texto plano, introducida por el usuario. Es esta última función la encargada de encriptar de manera segura la contraseña. Por defecto, utiliza SHA-256.

Cuando un usuario intenta iniciar sesión, se comprueba su contraseña llamando a la función *check_password*, que a su vez llama a *check_password_hash*, encargada de llevar a cabo la comprobación. Esta función recibe como parámetros la encriptación de la contraseña que existe en la base de datos y la contraseña introducida por el usuario en texto plano. Encripta la contraseña introducida y realiza la comprobación sobre las dos encriptas, lo cual se convierte en una comprobación de valores hexadecimales.

4.1.3. Cambiar constraseña

Acabamos este capítulo estudiando las páginas para cambiar la contraseña. Veamos primero en la figura 4.3 la página donde el usuario solicita cambiar de contraseña.

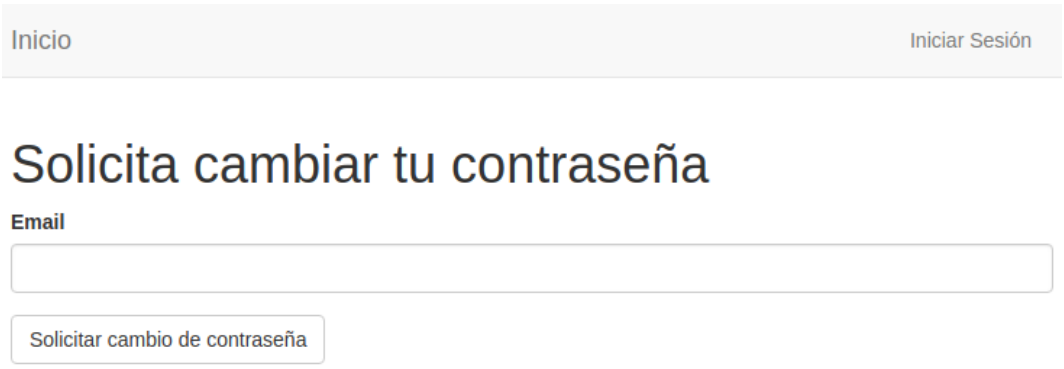

Figura 4.3: Página para solicitar cambiar la contraseña.

Se trata de una vista muy sencilla, donde simplemente hay que introducir una dirección de correo electrónico. Al introducir el formulario, se valida que el campo no esté vacío y que los datos efectivamente se traten de una dirección de correo electrónico. Si la dirección de correo electrónico pertenece a algún usuario en la base de datos, se envía un correo electrónico como el que podemos ver en la figura 4.4. Si no, la aplicación no avisa sobre ello, para mantener esta información confidencial.

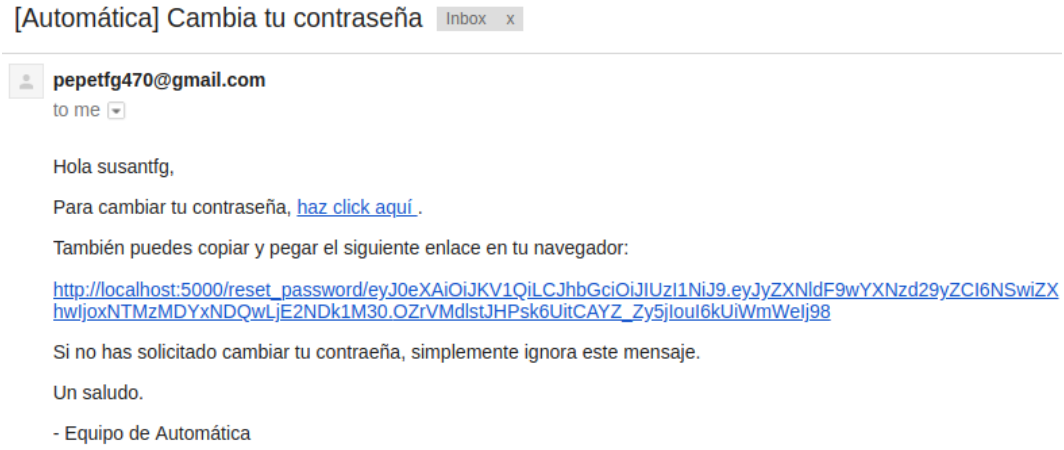

Figura 4.4: Email enviado para cambiar la contraseña.

Al principio del asunto del correo electrónico podemos leer “[Automática]”, esto no es más que una referencia a la asignatura para la cual, en principio, está pensada

esta aplicación web. El remitente es un correo electrónico que creé yo para probar esta funcionalidad y el receptor se trata de un usuario ficticio que también creé y tiene como nombre de usuario "susantfg". Podemos ver que su nombre aparece al principio del cuerpo del correo electrónico. Por último, vemos que el enlace nos dirige a *localhost*. Esto es porque estaba corriendo la aplicación web en mi máquina. Cuando la aplicación esté desplegada en un entorno real de producción, la plantilla del email se adaptará automáticamente.

La página a la que nos dirige el enlace la podemos ver en la figura 4.5. La cadena de caracteres de este enlace es una codificación del "id" del usuario al cual le pertenece la dirección de correo electrónico. Lo llamamos un *token*. Cuando se accede al enlace, se decodifica el *token* para extraer el "id". Sin embargo, este *token* sólo es válido durante 10 minutos. Si expira, el enlace ya no es válido y el usuario es redirigido silenciosamente.

Figura 4.5: Página para cambiar la contraseña.

Una vez introducida la nueva contraseña, el usuario es redirigido a la pantalla para iniciar sesión, junto con un mensaje informativo de que se ha cambiado con éxito su contraseña.

4.2. Pantalla "Inicio"

Cuando un usuario consigue iniciar sesión con éxito, es llevado a la pantalla "Inicio". En esta pantalla, que podemos ver en la figura 4.6, el usuario tiene a disposición la capacidad de enviar solicitudes de simulaciones de prácticas y elegir entre resultados de solicitudes anteriores para poder visualizarlos de nuevo. Vamos a estudiarla paso por paso.

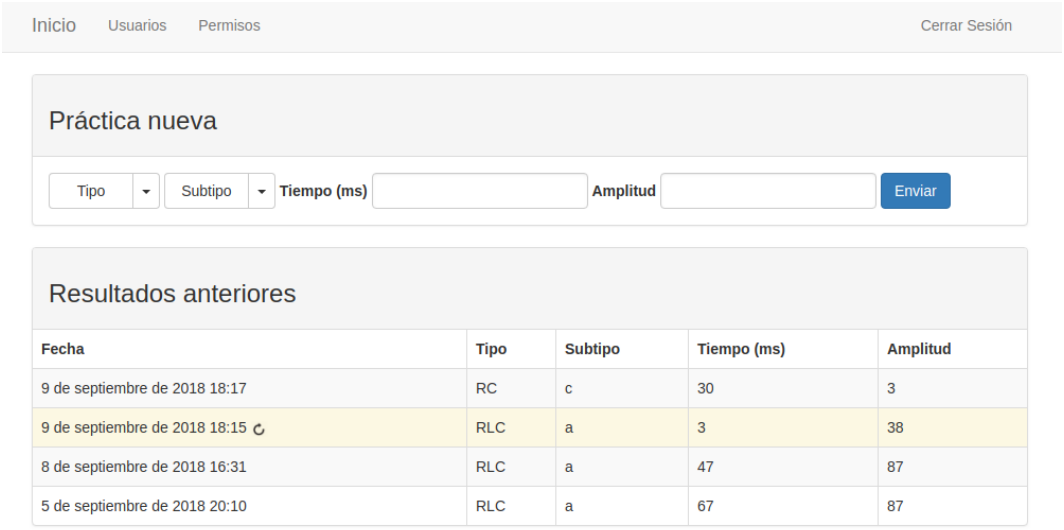

Figura 4.6: Página “Inicio”.

4.2.1. Lo que ve el usuario

Para empezar, vemos que la barra de navegación ha cambiado con respecto a las pantallas anteriores. Ahora, en el extremo superior derecho, vemos que existe un botón “Cerrar Sesión”. Al elegir esta opción, se cerrará la sesión del usuario y será dirigido a la página para iniciar sesión. Por otro lado, vemos dos botones nuevos: “Usuarios” y “Permisos”. Estos botones dirigirán al usuario a las páginas para gestionar los usuarios y los permisos en el sistema, respectivamente. Sin embargo, estos botones sólo aparecen para usuarios administradores, es decir, que tengan asignado el rol “admin”. Además, si un usuario normal intenta acceder a esas páginas a través de la URL, le aparecerá una pantalla como la que aparece en la figura 4.7.

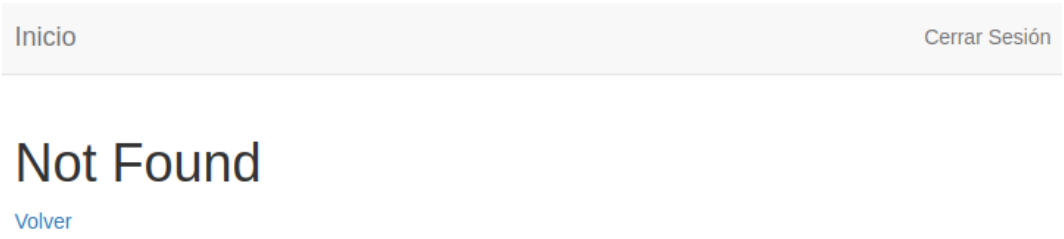

Figura 4.7: Página de error 404.

Puesto que la vista está dividida en dos secciones, vamos a estudiarlas en secciones distintas también.

Sección "Práctica nueva"

Esta sección es un formulario embebido en la pantalla que sirve para solicitar simulaciones nuevas. Comienza con dos listas desplegables, una para seleccionar el "Tipo" y otra para seleccionar el "Subtipo". La lista de subtipos aparece vacía si previamente no se ha seleccionado un tipo. Cuando se selecciona cualquiera de ellos, el texto de la caja cambia por lo que se haya seleccionado.

Una vez seleccionado el subtipo, aparece información adicional sobre los componentes que forman parte de él y, además, se rellena automáticamente el campo "Tiempo (ms)" con un tiempo recomendado. Si nos remontamos al ejemplo 2.10 de la página 28, recordamos que cada subtipo viene asociado con una serie de campos y valores; son éstos los utilizados. Podemos ver un ejemplo en la figura 4.8, donde todos los valores aparecen con 0.

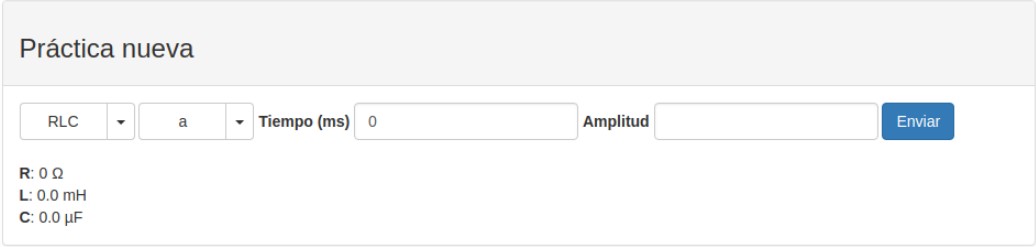

Figura 4.8: Vista tras seleccionar un subtipo.

Ahora, el usuario puede introducir los dos valores parametrizados y enviar la solicitud. Sin embargo, previo al envío del mensaje por Spacebrew, se realiza una validación de los campos. Se comprueba que se haya seleccionado un tipo y un subtipo y que los valores para el tiempo y la amplitud sean números y estén entre los rangos predefinidos. Podemos ver un ejemplo en la figura 4.9.

Sección "Resultados anteriores"

Se presenta como una tabla donde cada fila corresponde a una simulación previamente solicitada por el usuario. Cada fila contiene información sobre la fecha en la que fue llevada a cabo, el tipo y el subtipo de la simulación y los valores de tiempo y amplitud que introdujo el usuario para ello. Además, cada fila es seleccionable y redirige al usuario a la visualización de los resultados.

Los resultados están ordenados de más recientes a más antiguos. Las fechas corresponden al campo "timestamp_utc" de la tabla *user_practices*, sin embargo son reconvertidas a la zona horaria del usuario y son mostradas en un formato fácilmente legible.

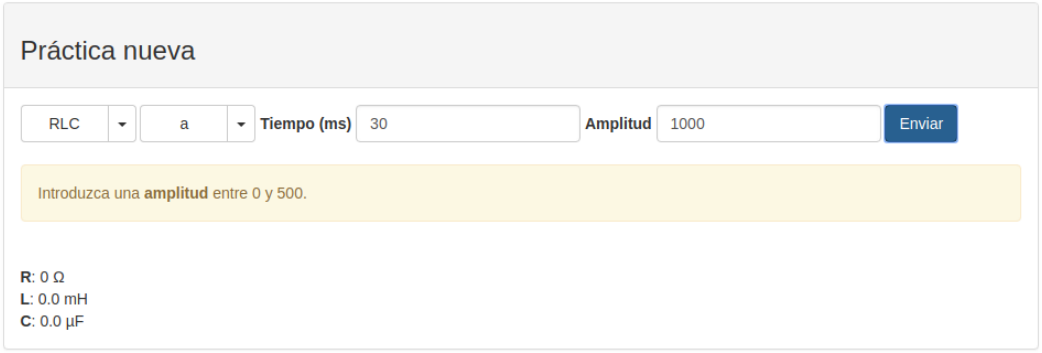

Figura 4.9: Vista tras fallar la validación de la solicitud.

Sin embargo, existe un límite de resultados para la tabla. Por defecto es 20, pero, es un parámetro fácilmente configurable antes de iniciar la aplicación web. Cuando el número de simulaciones que ha hecho un usuario supera este límite, aparecen debajo de la tablas enlaces para ir a la siguiente o a la anterior página de resultados. Existe una paginación, pero, no solo a nivel de *front-end*, sino también a nivel de las búsquedas que realiza el *back-end* a la base de datos. Esto hace que un usuario siempre tenga una experiencia fluida, aunque haya hecho cientos de simulaciones.

Por último, si nos fijamos en la tabla, vemos que hay una fila anaranjada y con un símbolo al lado de la fecha: ↻. Esto significa dos cosas. Por un lado, la fila anaranjada significa que esa simulación, por algún motivo, no tiene resultados en la base de datos. El símbolo, por otro lado, le muestra al usuario que es posible solicitar de nuevo la misma simulación. Por lo tanto, al seleccionar esta fila, se genera de nuevo todo el flujo de envíos de mensajes y se intenta de nuevo conseguir resultados.

4.2.2. Lo que no ve el usuario

Vamos a ver ahora lo que ocurre detrás de las escenas para que el usuario tenga todo tan accesible. Dividimos de nuevo en dos secciones para las distintas partes de la pantalla.

Sección “Práctica nueva”

Como se ha mencionado antes, la información mostrada aquí proviene de la aplicación de Processing. Por lo tanto, para poder adquirir esta información, es necesario contactar con esa aplicación via Spacebrew. Esto implica la creación de cliente, suscriptor y publicador para el usuario, así como las rutas con el suscriptor

y el publicador del cliente de Processing. En vez de copiar aquí todo el código que forma parte de este proceso, voy a explicarlo por pasos.

Lo primero que ocurre cuando un usuario accede a esta página es que el *back-end* envía la página HTML, generada con Jinja, y los ficheros estáticos, i.e. CSS, JavaScript e imágenes, para que el *front-end* pueda renderizar la página. En el cuerpo del HTML, se especifica una función a ejecutar cuando se termine de cargar.

Esta función realiza una operación GET asíncrona contra el *back-end* para obtener la información necesaria para crear el cliente de Spacebrew y conectarse, i.e. el nombre o la dirección IP donde está Spacebrew y el puerto por el que está escuchando y el nombre de usuario y el "id" del usuario accediendo a la página. Si esta operación tiene éxito, se procede a preparar el cliente para Spacebrew.

El cliente es preparado con el nombre de usuario como nombre. Se le añade el mix-in de Spacebrew Admin, el publicador y el suscriptor. Se definen también los nombres de las funciones a ejecutarse cuando ocurran los eventos de llegada de mensaje tipo string y de conexión de nuevo cliente. Por último, se conecta con Spacebrew.

La función que se ejecuta con la conexión de un nuevo cliente comprueba que existan en Spacebrew como clientes tanto el cliente del usuario como el cliente de Processing. Si existe, intenta crear las rutas entre los suscriptores y publicadores y, si tiene éxito, envía el mensaje para recibir la información de la aplicación de Processing.

La función que se ejecuta con la llegada de un mensaje de tipo string va a comprobar primero que el JSON contiene el campo "config", para asegurarnos de que es un mensaje de información de Processing, y luego va a comprobar que el campo "id" es igual al "id" del usuario que envió el mensaje. Si se cumplen ambas condiciones, se guarda la información en una *cookie*[1]. Cada vez que el usuario elija un tipo o un subtipo, se va a consultar esta *cookie* para generar la información.

Sin embargo, los usuarios no siempre van a tener permisos para realizar todas las simulaciones que nos ofrece la aplicación de Processing. Por lo tanto, la lista de tipos y subtipos disponibles para el usuario tiene que estar acorde a los permisos que tiene el usuario. Así que, después de conseguir la información de Processing, se hacer otra llamada GET asíncrona al *back-end* para obtener los permisos que tiene el usuario. Esta información también se guarda como una *cookie*.

Una vez esto último ha tenido éxito, se procede a poblar la lista de tipos para el usuario, fijándose si tiene permisos o no. Cuando seleccione un tipo, se hará lo mismo para la lista de subtipos, y, como los permisos son acumulativos, si un usuario tiene un rol que le da permisos al subtipo a del tipo RLC y también otro rol que

[1]Una *cookie* (galleta o galleta informática) es una pequeña información enviada por un sitio web y almacenada en el navegador del usuario, de manera que el sitio web puede consultar la actividad previa del navegador.

le da permisos al subtipo *b* del mismo tipo, aparecerán ambos subtipos en la lista desplegable. Si un usuario tiene el rol “admin”, tendrá permiso a todos los tipos y todos los subtipos.

Este proceso se lleva a cabo por completo cada vez que el usuario accede a la página y visitas a posteriores páginas asumen que el usuario haya pasado por esta y hayan guardado las *cookies*. En este sentido, se podría hacer un uso más eficiente de las *cookies* y, en posteriores páginas, se podría proteger sobre la falta de las *cookies* realizando los pasos necesarios para conseguirlas.

El otro proceso importante que ocurre en esta sección es el que se lleva a cabo cuando el usuario pulsa el botón “Enviar”. Como se ha visto antes, existe una validación en este paso, con mensajes indicando qué está mal en el formulario. Esta validación es llevada por el *front-end*. Cuando tiene éxito, el *back-end* recibe la información del formulario y procede a añadir un registro nuevo en la tabla *user_practices*. En este punto, no hay resultados de la simulación, por lo tanto este campo se va a quedar vacío en la base de datos hasta que recibamos los resultados desde la aplicación de Processing.

El *back-end* también se ocupa de generar el valor para el campo “uuid”. Para esto, primero convierte a string y concatena los siguientes campos, en orden: “user_id”, “timestamp_utc”, “type”, “subtype”, “time” y “amplitude”. Ahora, el string generado lo pasa sobre el algoritmo de encriptación MD5, el cual devuelve 16 bytes. Este resultado es convertido a 32 caracteres hexadecimales y, por último, se convierten a mayúscula.

Después de esto, el usuario será redirigido a la pantalla para esperar los resultados de Processing. Es allí donde se guardarán los resultados de la simulación. Lo veremos más adelante.

Sección “Resultados anteriores”

La generación de la tabla se realiza por completo en la generación del HTML por Jinja. Se le pasa a la plantilla HTML de la página la lista de resultados a añadir, ya ordenados por fecha. En la plantilla nos ocupamos de rellenar las columnas y de mostrar de color anaranjado las filas que corresponden a simulaciones sin resultados. Para formatear la fecha utilizamos *Flask-Moment*[2] [12]. También hacemos que cada fila sea seleccionable y que redirijan a la pantalla de visualización de resultados.

Para que una práctica se quede sin resultados, pueden ocurrir una variedad de cosas. Puede ser que la aplicación de Processing haya tenido algún error al realizar la simulación. También puede ocurrir que el usuario no se quede esperando a que lleguen los resultados, lo que haría que nunca se guardaran en la base de datos. Otra opción es que el mensaje del usuario del cliente no se envíe a Spacebrew, por algún

[2]Una extensión de Flask que permite formatear fechas utilzando moment.js.

problema de la conexión.

Sin embargo, puesto que la práctica se guarda con toda la información que la hace única, esto nos permite utilizarla de nuevo para solicitar de nuevo la simulación. La página de resultados tiene dos puntos de entrada: uno para simulaciones con resultados y otro para simulaciones sin resultados. Veremos más adelante cómo funciona la página de resultados y cómo se guardan los resultados.

Vamos a recordar dos requisitos funcionales: el relacionado con avisar al usuario via email cuando sus resultados están disponibles y el relacionado con relacionado con reaccionar a una demanda superior a los recursos disponibles. Estos requisitos funcionales eran opcionales y no se han llegado a cumplir por falta de tiempo. Realmente, sería cuestión de reutilizar parte del código para llevarlos a cabo, pero no he podido dedicarle tiempo a ello. Darle la posibilidad al usuario de solicitar manualmente los resultados es una alternativa a estos requisitos y nació de manera trivial debido a la infraestructura de la aplicación.

4.3. Pantalla "Resultados"

Veamos ahora la pantalla que visita el usuario para visualizar los resultados de sus simulaciones. Centrémonos primero en la figura 4.10.

4.3.1. Lo que ve el usuario

La página se presenta de una manera muy simple para el usuario. En la parte de arriba dispone de información sobre la práctica, que equivale a lo que podía ver en la tabla de "Resultados anteriores", pero sumándole los valores de los componentes "R", "C" y, si corresponde, "L". En la imagen tienen valor 0 porque se trata de un ejemplo.

En la parte inferior se presenta una gráfica, la cual es la visualización de los resultados de la simulación. Los ejes están nombrados para facilitar su compresión: el eje X es el "Tiempo (ms)" y el eje Y la "Tensión (V)".

Además, el usuario puede pasar su ratón sobre la línea de la gráfica para obtener un valor. Podemos ver un ejemplo en la figura 4.11. Al pasar el ratón por encima, aparece una caja con dos números. El número superior, en negrita, corresponde al valor respecto al eje X, mientras que el inferior corresponde al valor respecto al eje Y. Para que el usuario pueda entender esto más fácilmente se ha añadido el mensaje "Salida: " por delante del número inferior.

Esta página también muestra mensajes informativos al usuario cuando:

- Se está esperando a que se envíe la solicitud a través de Spacebrew.

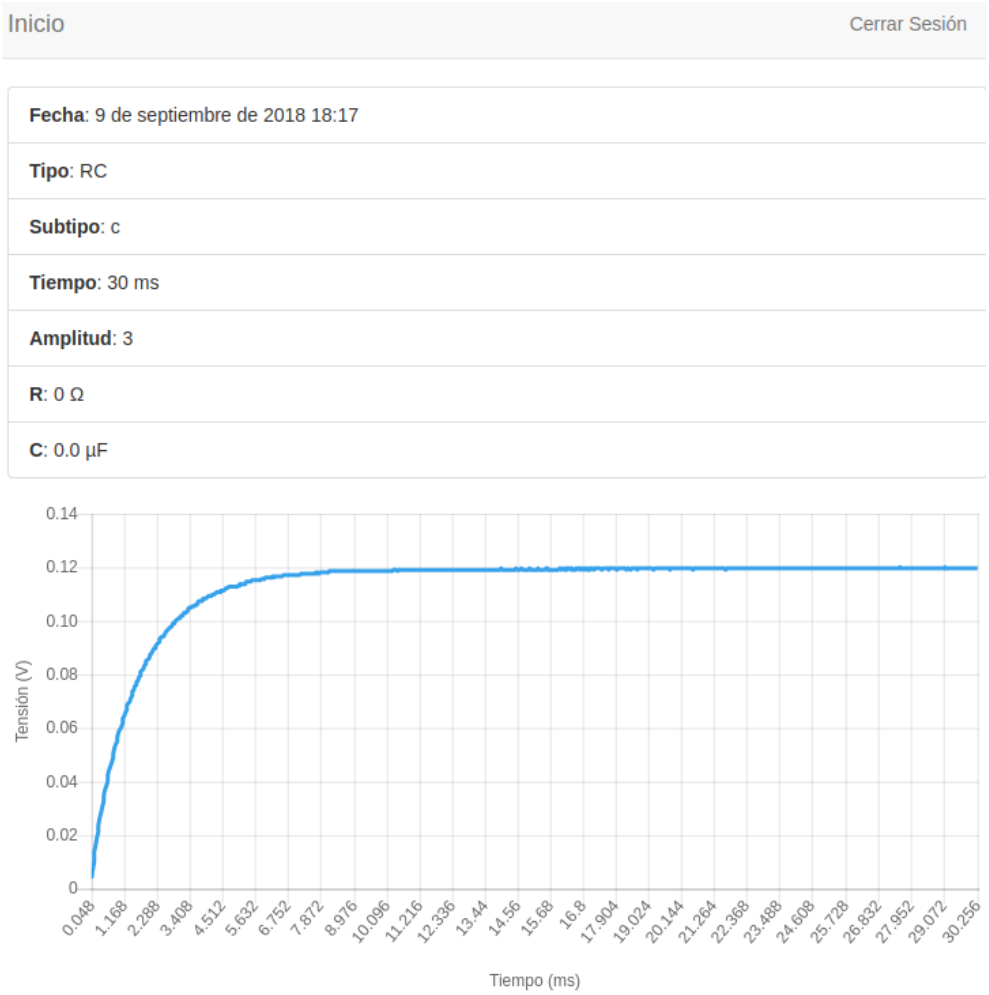

Figura 4.10: Página “Resultados”.

- El cliente Spacebrew del usuario ha recibido una respuesta como la del ejemplo 2.7 de la página 27 y se están esperando los resultados.
- Ha habido algún problema en la realización de la simulación y se ha recibido un mensaje como el del ejemplo 2.8 de la página 28.
- La simulación ya tiene resultados en la base de datos y se está esperando a recuperarlos y visualizarlos.

4.3.2. Lo que no ve el usuario

Mientras que lo que ve el usuario se presenta como algo bastante simple, lo que ocurre por detrás es más bien complejo. Se ha mencionado antes que esta página

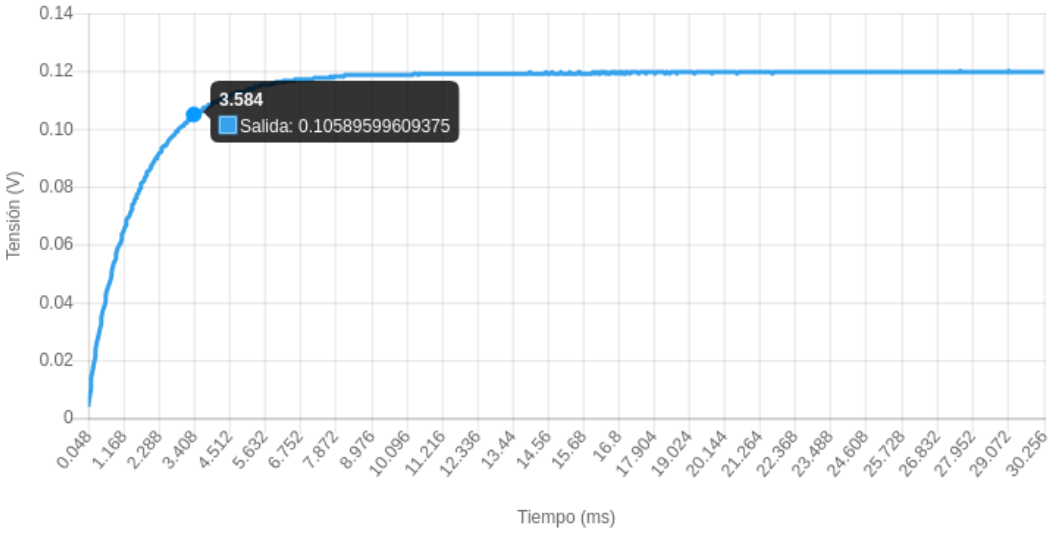

Figura 4.11: Pasando el ratón por encima de la gráfica.

tiene dos puntos entrada según la simulación a visualizar tenga resultados o no. Vamos a dividir esto en dos secciones.

Cuando hay resultados

El *front-end* realizar una llamada POST contra el *back-end* solicitando la información y resultados de la simulación intentando visualizarse. La URL de la página resultados lleva el "uuid" de la simulación, se utiliza éste para solicitar sus datos en concreto. Si esta operación tiene éxito, primero se muestra la información de la simulación, en la parte superior de la práctica, y posteriormente se crea la gráfica con los resultados, borrando consigo cualquier mensaje informativo presente en la página.

Para mostrar la información de la primera mitad de la página, hace falta que tengamos a disposición la información de las simulaciones que nos llega desde Processing, la cual podemos recordar en el ejemplo 2.10 de la página 28. Esta información se coge de las *cookies*, suponiendo que el usuario ha pasado por la pantalla "Inicio" y ha conseguido estos datos a través de Spacebrew. Si esto no se cumple, la página no carga por completo.

La librería utilizada para la gráfica es *Chart.js* [16]. Al iniciar la gráfica se aceptan una serie de parámetros configurables para cambiar los colores, los nombres de los ejes, el tipo de gráfica y mucho más. También hay que especificarle los datos en el momento de la creación. Estos datos se dividen en dos listas, uno para los valores del eje X y otro las del eje Y. Como restricciones, ambas listas tienen que tener la misma cantidad de objetos, cada elemento del eje X tiene que corresponder

al elemento en el eje Y con el mismo índice y los valores del eje X tienen que estar ordenados de menor a mayor. Esto último es una restricción porque si no los resultados se ven desordenados en la gráfica, sin embargo, para este caso, no tenemos que preocuparnos, ya que se ordenan antes de insertarse a la base de datos.

Cuando no hay resultados

Cuando no existen resultados en la base de datos, hay que pasar por Spacebrew para conseguirlos. Lo primero que ocurre es que se definen el mensaje a enviar, con el esqueleto del ejemplo 2.5 de la página 26, se muestra la información en la primera mitad de la página y se muestra el mensaje informativo de que se está esperando a que se envíe la solicitud. Seguidamente, se procede a crear el cliente de Spacebrew y conectarlo con el cliente de la aplicación de Processing, tal como se ha descrito para la pantalla “Inicio”.

La información para definir el mensaje de solicitud se adquiere siempre de la base de datos. Esto permite solicitar de manera trivial resultados para simulaciones que no las hubieran conseguido en su primer intento.

Las reacciones a los eventos de llegada de mensaje tipo string y de conexión de nuevo cliente varían respecto al caso anterior. En este caso, cuando se hayan creado con éxito las rutas, se envía el mensaje de solicitud de simulación y, cuando nos llega un mensaje de tipo string, realizamos otra serie de comprobaciones.

Cuando nos llega un mensaje de tipo string, comprobamos primero que el valor del campo “id” es igual al “uuid” de la simulación. Si tiene éxito, realizamos las siguientes tres comprobaciones, en orden:

1. Miramos si existe el campo “resultados”, indicándonos que se trata de una respuesta con resultados desde Processing. Si existen, cogemos la lista de resultados y las ordenamos por tiempo. Esto lo hacemos porque se conoce que la aplicación de Processing tuvo un problema y generaba resultados desordenados, lo cual causaba que la gráfica de resultados no tuviera sentido. Aunque esto ya fue arreglado, yo he decidido mantener este paso como una medida extra.

   Ahora, los resultados se los enviamos asíncronamente al *back-end*, a través de una llamada POST, para que los pueda persistir en la base datos. Podemos ver un pequeño extracto del código de la aplicación web en el ejemplo 4.3, se trata del código encargado de serializar los resultados, para guardarlos en la base de datos de manera eficiente, y de deserializarlos, para que los usuarios puedan revisualizarlos. Utilizamos la librería *pickle*[3] [13].

[3]Pickle es un módulo de Python que permite serializar casi cualquier objeto (objetos de tipos definidos por el usuario, colecciones que contienen colecciones, etc) y cuenta con algunos mecanismos de seguridad básicos.

```python
import pickle

# ...

class UserPractice(db.Model):

    # ...

    results = db.Column(db.Binary)

    # ...

    def set_results(self, results):
        self.results = pickle.dumps(results)

    def get_results(self):
        return pickle.loads(self.results)
```

Ejemplo 4.3: Serializando y deserializando resultados.

Por último, se muestran los resultados con *Chart.js* igual que cuando no hay resultados. Puesto que enviar los resultados al *back-end* se lleva a cabo a través de una llamada asíncrona, los usuarios no tienen que esperar a ello para poder visualizarlos.

2. Miramos si existe el campo "error", lo cual nos indica que ha habido algún error en la solicitud o si la aplicación de Processing no ha podido llevarla a cabo por algún problema interno. Cuando esto ocurre, mostramos un mensaje para informar al usuario.

3. Si no existen ninguno de los campos anteriores, entendemos que se trata de una respuesta como la del ejemplo 2.7 de la página 27 y mostramos un mensaje informando que estamos a la espera de resultados.

4.4. Herramientas administrativas

Las últimas pantallas por ver corresponden a las páginas que pueden visitar sólo los usuarios administradores, i.e. tienen el rol "admin". Recordemos que estas páginas forman parte de los requisitos funcionales de la aplicación.

4.4.1. Gestión de usuarios

Primero, vamos a ver cómo los administradores pueden gestionar usuarios. Centrémonos en la figura 4.12.

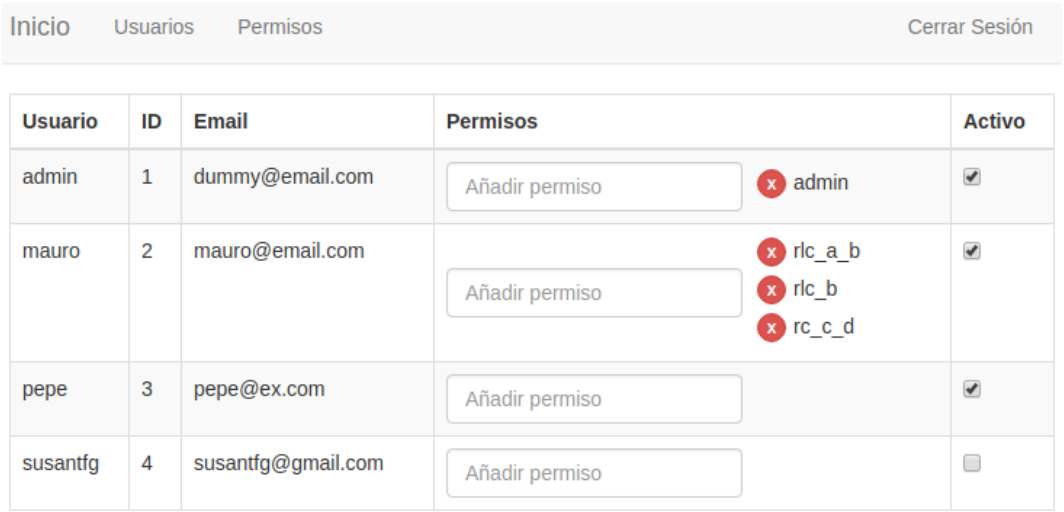

Figura 4.12: Página para gestionar usuarios.

Para llegar aquí, debemos pulsar en "Usuarios" en la barra de navegación. La página ha sido pensada que tener un uso muy simple pero potente. Se presenta una tabla donde se listan los usuarios con sus nombre de usuario, ID (en a la base de datos), correo electrónico, permisos e información de si está activado o no. Los usuarios están ordenados por el ID y la tabla está paginada, a nivel de base de datos para mayor velocidad de renderización, con un límite de 10 usuarios por página, por defecto. Al igual que en la sección de "Resultados anteriores" de la página "Inicio", este valor se puede configurar antes del inicio de la aplicación web. Cuando se supera el límite, aparecen enlaces debajo de la tabla para pasar a la siguiente o a la anterior página.

Para activar o desactivar un usuario, basta con marcar o desmarcar la casilla de verificación de cada usuario. Si está marcada, significa que el usuario está activo y puede utilizar la página web. Los usuarios por defecto están inactivos, los administradores tienen que activarlos uno por uno.

Para añadirle permisos a un usuario, basta con escribir el nombre del rol que deseamos añadirle y pulsar "Enter" en el teclado. Esto implica que tengamos que saber y escribir el nombre exacto del rol que queremos, aunque se admite escribirlo en mayúsculas. Si el rol no existe, nos saldrá un mensaje informativo. Si el usuario ya contiene el rol, no ocurre nada, pero nos saldrá un mensaje informándonos de que el rol se ha añadido, tal como ocurriría si el usuario no hubiese contenido el rol.

Por último, para quitarle permisos a un usuario basta con darle al icono rojo con una cruz. Nos saldrá un mensaje informándonos de que la operación ha tenido éxito.

La comunicación entre el *front-end* y el *back-end* se lleva a través de una serie de

llamadas POST asíncronas. Para cada una, el *back-end* modifica permanentemente la base de datos.

4.4.2. Gestión de permisos

Veamos ahora la página donde se listan los permisos disponibles en el sistema, a la cual se llega pulsando en "Permisos" en la barra de navegación. Estudiemos la figura 4.13.

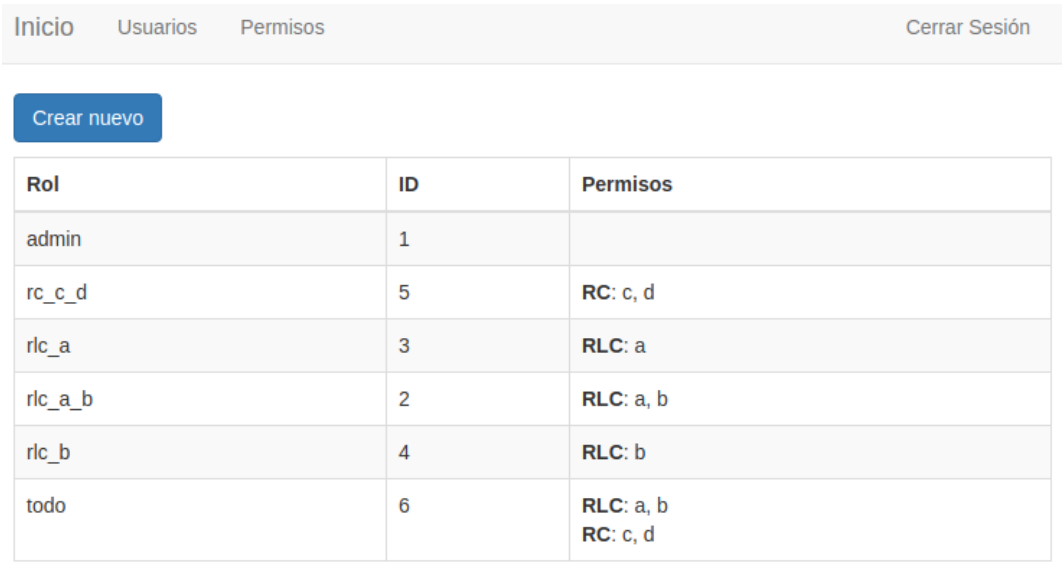

Figura 4.13: Página para gestionar permisos.

Se trata de una página muy simple también, donde simplemente podemos observar todos los permisos que existen en la base de datos. Esta tabla no está paginada, puesto que no se espera que tenga mucho contenido, sin embargo, se podría hacer fácilmente fijándose en las otras. La tabla está ordenada por el nombre del rol e incluye el "id" de la base de datos y a qué simulaciones da permisos cada una.

La tabla se pobla en el momento de la creación de la plantilla HTML con Jinja, a la cual se le pasan los roles ordenados alfabéticamente obtenidos tras una búsqueda en la base de datos.

Creando permisos

En la pantalla anterior hay un botón "Crear nuevo" que nos dirige a la pantalla para crear nuevos permisos. Podemos verla en la figura 4.14.

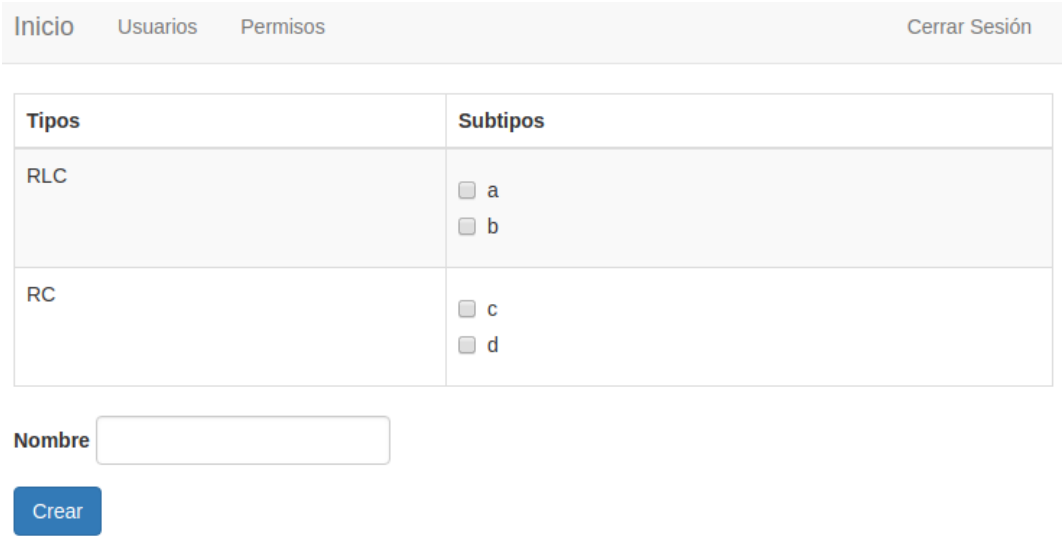

Figura 4.14: Página para crear permisos.

La tabla se crea a través de la información recibida de la aplicación de Processing. Una vez más, buscamos esta información en las *cookies*, asumiendo que el usuario ha pasado por la pantalla "Inicio" y ha conseguido estos datos.

Se crea una fila en la tabla para cada tipo disponible. La columna "subtipos" se pobla con todos los subtipos disponibles para cada tipo. Cada subtipo lleva consigo una casilla de verificación para que el administrador pueda seleccionar a qué subtipos va a dar acceso este permiso nuevo. Por lo tanto, entendemos que los permisos funcionan a nivel de subtipo.

Una vez seleccionado todo, el usuario puede especificar el nombre del nuevo rol y pulsar en el botón "Crear" o presionar la tecla "Enter" en el teclado. Al hacer esto, se reúne toda la información del formulario y es enviada al *back-end* de manera asíncrona a través de una llamada POST.

El *back-end* lleva a cabo una validación antes de aceptar y añadir a la base de datos el nuevo rol. Primero comprueba que el nombre del rol haya sido introducido y que no exista un rol en la base de datos con el mismo nombre (recordemos que los nombres de roles se convierten siempre a minúscula). Luego, comprueba que se haya seleccionado al menos un subtipo.

Se muestran mensajes informativos tanto cuando hay alguna restricción que no se cumple en la creación del rol como cuando se ha creado y añadido a la base de datos correctamente.

Para acabar, veamos el formato que tienen los permisos. Se trata de un JSON y es una lista de objetos en el que cada objeto contiene el nombre del tipo y una lista de subtipos correspondiente a las que se hayan seleccionado en la creación del rol.

podemos observarlo en el ejemplo 4.4.

```
[
    {
        "type":"RLC",
        "subtypes":["a","b"]
    },
    {
        "type":"RC",
        "subtypes":["c","d"]
    }
]
```

Ejemplo 4.4: Estructura de los permisos.

Este JSON es serializado para guardarlo en la base de datos y se deserializa cuando la aplicación lo necesita. Se realiza este proceso de igual manera que lo visto en el ejemplo 4.3 de la página 55.

Cuando el usuario tiene el rol "admin", el cual tiene permiso a todo, este JSON está simplificado. Lo podemos ver en el ejemplo 4.5.

```
[
    "admin"
]
```

Ejemplo 4.5: JSON de los permisos para administradores.

4.5. Comportamiento *responsive*

Uno de los requisitos no funcionales listados en la introducción de este proyecto, y quizás el más importante, es que la aplicación web tenía que tener un comportamiento *responsive* para que se pueda utilizar correctamente en dispositivos móviles. Este objetivo ha sido cumplido.

En su mayoría, es gracias a la utilización de Bootstrap, el cual ha sido muy fácil de integrar en el proyecto. Lo único difícil ha sido conseguir que la tabla de resultados con Chart.js se viese bien. Aunque esta última librería viene capacitada con un comportamiento *responsive*, no fue fácil configurarlo bien, sobretodo para pantallas pequeñas.

Podemos observar cómo se ve la aplicación web en dispositivos móviles en uno de los anexos.

Capítulo 5

Desplegando la aplicación web

Contenido

5.1. Instalar Git y Docker

Veamos ahora cómo se despliega la aplicación web. Es realmente sencillo, porque todo ha sido resumido a unos cuantos scripts. Lo único que necesitamos es instalar Git y Docker. Estos pasos han sido adaptados para que la aplicación pueda ejecutarse en una Raspberry Pi 3, que es donde está destinada a ser ejecutada. Este ordenador tiene un procesador ARM, por ello tenemos que instalarle un *software* más específico y compatible.

Git, en realidad, es opcional, ya que sólo se utiliza para obtener el código fuente de la aplicación. En caso no de tenerlo a disposición, podemos instalar Git siguiendo las instrucciones oficiales [28]. Para el caso de Ubuntu, basta con ejecutar el siguiente comando:

```
1 $ sudo apt install git-all
```

Ahora, tenemos que clonar el repositorio alojado en Bitbucket. Se trata de un repositorio privado, por lo cual hace falta tener acceso previo para poder realizar este paso.

```
$ git clone https://maudagos@bitbucket.org/maudagos/webserver.git
```

Una vez hemos conseguido el código fuente, lo siguiente es instalar Docker. Existen dos versiones: *Community Edition* y *Enterprise Edition*. A nosotros nos basta con la primera versión si no vamos a correr la aplicación en la Raspberry Pi y en la página oficial de Docker nos describen cómo instalarla según el sistema operativo [21]. Para el caso de la Raspberry Pi, necesitamos otra versión de Docker capaz de ser ejecutada en un procesador ARM. Esta versión se llama *Hypriot* y existe un tutorial muy sencillo para instalarla [22]. Tenemos que seguir los pasos al pie de la letra, asegurándonos que tenemos todos los requisitos, pero, es todo muy sencillo.

Una vez instalado Docker CE tenemos también una serie de pasos opcionales. Nos interesan los pasos para poder ejecutar Docker sin ser usuario *root*, es decir, sin tener que utilizar "sudo", y para que Docker se ejecute automáticamente cada vez que se arranque la máquina.

Ahora ya tenemos todo preparado para empezar a ejecutar los scripts que van a desplegar cada componente de la aplicación web. Están pensados para ser ejecutados en máquinas con sistemas operativos basados en *Linux*, soportando también aquellas con procesador ARM, y tienen que ejecutarse en un orden determinado, ya que un paso depende del anterior.

El gran beneficio que obtenemos de utilizar Docker es que nos da la posibilidad de empaquetar todas las dependencias que necesita la aplicación web en una simple imagen Docker preparada para ser ejecutada como un contenedor en cualquier máquina corriendo un sistema operativo Linux. Por lo tanto, a la hora de querer correr la aplicación web en una máquina nueva, nos basta con tener la imagen Docker y no hace falta instalar Python o cualquier tipo de librería. En nuestro caso también hace falta el código fuente, pero se podría haber incluido el código fuente en la imagen Docker. Se optó por no hacer esto, veremos enseguida la ventaja que tiene.

Vamos a estudiar el despliegue de cada componente en orden.

5.2. MySQL

Para empezar, se ha escrito un script que lanza un contenedor MySQL con la base de datos que hemos estudiado antes. Esto no entra en los objetivos de este proyecto de fin de grado, pero resulta útil, ya que no nos tenemos que preocupar de instalar manualmente MySQL y de realizar la creación de la base de datos. Se queda todo resumido a unos pocos pasos.

Antes de nada, tenemos que realizar una pequeña preparación previa. Primero, tenemos que preparar un nuevo directorio en *$HOME*, así:

```
$ mkdir $HOME/docker-mysql
$ mkdir $HOME/docker-mysql/startup
$ mkdir $HOME/docker-mysql/data
```

El script espera encontrar una serie de archivos en estos directorios, así que tenemos que copiarlos:

```
$ cp webserver/datamodels/my.cnf $HOME/docker-mysql
$ cp webserver/datamodels/tables.sql $HOME/docker-mysql/startup
```

El directorio *$HOME/docker-mysql/data* se queda vacío. Ahora, podemos ejecutar un script, que se encuentra en el directorio *webserver/docker/mysql*. Este directorio sólo contiene el script en cuestión, así que no tiene pérdida. Se ejecuta así:

```
$ ./mysql.sh
```

El script va a lanzar un contenedor de una imagen Docker *hypriot/rpi-mysql* [23]. Se trata de una imagen Docker adaptada para ejecutar en máquinas con un procesador ARM. La imagen original es altamente configurable, así que tomamos ventaja de algunas opciones que nos brinda [19]. Estas opciones se encuentran en el manual oficial [20].

Para empezar, vamos a montar en el contenedor los ficheros y directorios que hemos preparado previamente. El archivo *my.cnf* sirve para configurar el servidor MySQL para que escuche a la dirección IP 0.0.0.0, es decir, a cualquier petición que llegue desde la red interna.

Vamos a montar también el directorio *$HOME/docker-mysql/startup* en el directorio */docker-entrypoint-initdb.d* del contenedor. Cuando el contenedor arranque por primera vez, va a ejecutar todos los scripts SQL que hayan en este último directorio, en orden alfabético.

Por último, montamos el directorio vacío *$HOME/docker-mysql/data* en el directorio */var/lib/mysql*. Esto lo hacemos para que todos los datos de la base de datos MySQL del contenedor se escriban en ese directorio de nuestra máquina. De no hacer esto, se escribirían en un volumen autogenerado por Docker, el cual luego sería algo complicado de encontrar.

En el script también establecemos una serie de variables de entorno que utiliza el contenedor. Los que estamos utilizando sirven para configurar el nombre de la base datos, el usuario y las contraseñas tanto de este usuario como del usuario *root*. En el manual oficial se recomienda no hacer esto último, pero para este proyecto está bien.

También vemos que enlazamos el puerto 3306 del contenedor con el puerto 3306 de la máquina. Este es el puerto por defecto que utiliza MySQL. Cuando corremos el contenedor MySQL en el mismo servidor que la aplicación web, esto no es necesario, por cómo se ejecuta la aplicación web, que veremos enseguida. Sin embargo, aparte de darnos la posibilidad de ejecutar MySQL y la aplicación web en distintos servidores, también nos da la posibilidad de conectarnos al contenedor MySQL a través de Oracle SQL Developer. Así, tenemos una manera gráfica de interactuar y manipular la base de datos.

La imagen Docker que utilizamos no es capaz de ejecutar el script de creación de tablas y datos, por alguna razón desconocida. Por falta de tiempo, no ha habido tiempo para solucionarlo, pero a continuación se explican los pasos a seguir para terminar de montar la base de datos. Primero, vamos a ejecutar un *bash* en el contenedor de MySQL para poder manipularlo.

```
$ docker exec -it mysql bash
```

Una vez dentro, nos posicionamos en el directorio donde existe el script que deseamos ejecutar y nos conectamos a la base de datos con el usuario y la base de datos que hemos creado con el script anterior, *mysql.sh*. Nos pedirá la contraseña del usuario, podemos encontrarla en el mismo fichero.

```
$ cd docker-entrypoint-initdb.d/
$ mysql -u app_user -p --database=app
```

Ahora, ya estamos conectados a MySQL. Sólo nos queda ejecutar el script, pero, asegurándonos de que estamos en la base de datos que hemos creado anteriormente.

```
mysql> use app;
mysql> source tables.sql
```

Con esto ya tenemos la base de datos lista para que la aplicación web pueda utilizarla. Nos desconectamos de MySQL con CTRL+D y salimos del contenedor pulsando CTRL+P y CTRL+Q.

5.3. Aplicación web

Con la base de datos MySQL corriendo, ya podemos desplegar la aplicación web. La aplicación web se puede dividir en dos partes, una encargada de las peticiones dinámicas y otra de las estáticas. Las peticiones dinámicas son aquellas que tienen que pasar por la aplicación Flask, ya que requieren generar un HTML personalizado para cada usuario. Por otra parte, las estáticas no tienen que pasar por la aplicación Flask y se refieren, en general, a ficheros CSS, JavaScript, imágenes, etc.

Con esta separación podemos servir a los usuarios los ficheros estáticos directamente, sin tener que pasar por la aplicación. Esto mejora los tiempos de petición a la aplicación web. Veamos ahora cómo se lleva a cabo el despliegue de cada parte.

5.3.1. Parte dinámica

Nos dirigimos al directorio *webserver/docker/webserver/dynamic*. Lo primero que tenemos que hacer es crear una imagen Docker preparada para ejecutar la aplicación web. Para esto, tenemos que ejecutar el script encargado de ello:

```
$ ./build.sh
```

Este script va a realizar la construcción de la imagen Docker de acuerdo a la especificación en el fichero *Dockerfile* del mismo directorio. Nuestra imagen utiliza como imagen base otra que trae instalado Python 3.5.6 sobre Alpine Linux[1]. Tenemos que instalar en nuestra imagen las librerías necesarias para que la aplicación pueda comunicarse con la base de datos MySQL, lo hacemos basándonos en cómo lo hace otra imagen Docker que las trae de serie [31]. También preparamos un entorno virtual [30] e instalamos en él todas las librerías que utiliza la aplicación, que se encuentran en el fichero *webserver/requirements.txt*.

Ahora, instalamos *Gunicorn* [24], que utilizaremos para ejecutar la aplicación web en el contenedor, y establecemos el fichero que va a ejecutar a través de la variable de entorno "FLASK_APP". Le copiamos al contenedor el archivo *boot.sh* que se encuentra en nuestro directorio. Este script se va a ejecutar cuando se arranque el contenedor y se encarga de lanzar la aplicación con Gunicorn. Por último, exponemos el puerto 5000 del contenedor, que es el puerto que utilizan por defecto las aplicaciones Flask.

Ya tenemos la imagen Docker preparada para ser ejecutada en un contenedor. Para poder realizar esto, tenemos el script *run.sh*. Este fichero acepta como parámetro opcional el camino absoluto a un fichero *.env*. Cuando la aplicación arranca, intenta buscar este fichero en su carpeta raíz, i.e. *webserver*. Este fichero no es más que una lista de variables de entornos que queremos fijar para que luego la aplicación utilice. Veamos un ejemplo:

```
SECRET_KEY=<clave-secreta>
SPACEBREW_SERVER=192.168.1.50
SPACEBREW_PORT=9000
DATABASE_URL=mysql://app_user:01AppUser@dbserver/app
MAIL_SERVER=smtp.googlemail.com
MAIL_PORT=587
MAIL_USE_TLS=1
MAIL_USERNAME=pepetfg470
MAIL_PASSWORD=<contraseña-de-MAIL_USERNAME>
ADMINS=pepetfg470@gmail.com, susantfg@gmail.com
USERS_PER_PAGE=15
RESULTS_PER_PAGE=30
```

Ejemplo 5.1: Ejemplo de fichero *.env*.

Estos son todos los valores configurables posibles en la aplicación web. Voy a intentar explicar rápidamente cada uno.

- El valor "SECRET_KEY" lo utiliza la aplicación Flask como una clave criptográfica. Por ejemplo, se utiliza para proteger los formularios de ataques CSRF[2].

[1] Alpine Linux es una distribución Linux que tiene como objetivo ser ligera y segura por defecto sin dejar de ser útil para tareas de propósito general.

[2] El CSRF (del inglés *cross-site request forgery* o falsificación de petición en sitios cruzados) es

El administrador se tiene que asegurar que esta clave es lo suficientemente única.

- "SPACEBREW_SERVER" y "SPACEBREW_PORT" corresponden al nombre del servidor o la dirección IP donde está corriendo Spacebrew y al puerto por el que está escuchando, respectivamente. Todos los clientes de los usuarios se van a conectar a esta dirección, por lo tanto en un uso real de la aplicación habría que poner la dirección IP externa y realizar una redirección de puertos[3].
- El campo "DATABASE_URL" sirve para especificar cómo conectarse a la base de datos. No voy a entrar en detalle, pero si leemos el ejemplo de izquierda a derecha, es especifican: el motor de la base de datos, el usuario, la contraseña de este usuario, la dirección a la base de datos (veremos enseguida de dónde sale el nombre *dbserver*) y el nombre de la base de datos. Si en el uso final de la aplicación se ejecutan en el mismo servidor los contenedores de MySQL y de la aplicación web, este ejemplo sería el caso real.
- Los campos "MAIL_*" sirven para configurar el envío de emails a través de la aplicación. El ejemplo está configurado para utilizar una cuenta *gmail*. Si se quiere hacer esto, hay que asegurarse que la cuenta "MAIL_USERNAME" permita que aplicaciones menos seguras accedan a la cuenta. Esto se hace desde de la propia cuenta.
- "ADMINS" es una lista de direcciones de correo electrónico separada por comas. A todos ellos les van a llegar emails de fallos inesperados en la aplicación web, en caso de producirse. Además, la primera dirección es la utilizada para enviar los correos electrónicos a los usuarios que quieran cambiar su contraseña.
- Los valores "USER_PER_PAGE" y "RESULTS_PER_PAGE" corresponden a cuántos usuarios van a aparecer por página en la pantalla "Usuarios" y a cuántos resultados van a aparecer por página en la tabla "Resultados anteriores".

La definición de estos parámetros y sus valores por defecto se pueden consultar en el fichero *webserver/config.py*.

Supongamos que tenemos un fichero *.env* y queremos utilizarlo para arrancar un nuevo contenedor de la parte dinámica de la aplicación web, lo hacemos con el siguiente comando:

un tipo de *exploit* malicioso de un sitio web en el que comandos no autorizados son transmitidos por un usuario en el cual el sitio web confía.

[3]La redirección de puertos es la acción de redirigir un puerto de red de un nodo de red a otro. Esta técnica puede permitir que un usuario externo tenga acceso a un puerto en una dirección IP privada (dentro de una LAN) desde el exterior vía un router con NAT activado.

```
$ ./run.sh ~/.env
```

Recordemos que el fichero *.env* es opcional, pero, en caso de especificarlo, hay que especificar el camino absoluto a él.

El script se va a encargar de ejecutar un contenedor con la imagen Docker que hemos creado. A este contenedor, le montamos todo el código fuente de la aplicación en */var/www/webserver* y el fichero *.env* en */var/www/webserver/.env* (utilizar subdirectorios de */var/www* es un estándar). El script *boot.sh* que se ha copiado a la imagen Docker en el momento de su creación sabe que tiene que buscar el código en este subdirectorio.

Por último, utilizamos el parámetro "–link" que nos proporciona el comando de Docker. Con este parámetro podemos especificar conexiones de nuestro contenedor a otro. Para este caso, hemos especificado "–link mysql:dbserver". El primer valor corresponde al nombre del contenedor MySQL y el segundo será el nombre que utilizará nuestro contenedor para referirse a él. Ahora, si nos remontamos al ejemplo 5.1 en la página 65, podemos entender de dónde salía el nombre "dbserver". El uso de "–link" devolverá un error si el contenedor no está corriendo.

Para acabar, hay que asegurar que entendamos que la manera de la que está hecho este despliegue significa que la imagen Docker que creamos es totalmente independiente del código fuente. Podemos realizar cuantas modificaciones queramos en el código, no necesitaremos crear una imagen nueva, sólo habrá que detener el contenedor y volver a ejecutar el script *run.sh*. El único momento en el que necesitaremos crear una imagen nueva será cuando la aplicación web necesite utilizar alguna librería nueva y se vea reflejada en el fichero *webserver/requirements.txt*.

5.3.2. Parte estática

Vamos a localizarnos ahora en el directorio *webserver/docker/webserver/static*. Los pasos a realizar son idénticos a la parte dinámica, i.e. primero creamos una imagen Docker y después la desplegamos en un contenedor. Para el primer paso, basta con ejecutar el siguiente script:

```
$ ./build.sh
```

Si nos fijamos en el *Dockerfile*, veremos que la imagen no es más que una copia de una imagen Docker que trae Nginx instalado sobre Alpine Linux. Lo único especial del *Dockerfile* es la realización de un par de comandos, que sirven para asegurarnos que podamos visualizar los *logs*[4] de acceso. La parte estática va a ser simplemente un servidor proxy que sirva directamente los ficheros estáticos y redirija a la parte

[4]En informática, se usa el término *log*, historial de *log* o registro a la grabación secuencial en un archivo o en una base de datos de todos los acontecimientos (eventos o acciones) que afectan a un proceso particular (aplicación, actividad de una red informática, etc.). De esta forma constituye una evidencia del comportamiento del sistema.

dinámica las demás peticiones. Para entender la configuración, veamos el script encargado de lanzar esta parte. Se ejecuta así:

```
$ ./run.sh
```

Este script va a montar en el contenedor los ficheros estáticos, que se encuentran en *webserver/app/static*, en el directorio */var/www/webserver/app/static*, siguiendo el patrón establecido por la parte dinámica. Esta parte no necesita del resto del código fuente, puesto que sólo se encarga de servir estos ficheros. También va a montar el fichero *webserver.conf* en el contenedor como */etc/nginx/conf.d/default.conf*. La imagen Nginx va a cargar cualquier archivo *.conf*, que son archivos de configuración para Nginx, que se encuentren en el directorio */etc/nginx/conf.d*. Por defecto, existe el fichero *default.conf*, pero lo sobrescribimos con el contenido del nuestro: *webserver.conf*. Veamos el contenido de éste.

```
server {
  # listen on port 80 (http)
  listen 80;
  server_name _;

  # write access and error logs to /var/log
  access_log /var/log/webserver_access.log;
  error_log /var/log/webserver_error.log;

  location / {
    # forward application requests to the gunicorn server
    proxy_pass http://webserver-dynamic:5000;
    proxy_redirect off;
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
  }

  location /static {
    # handle static files directly, without forwarding to the
    # application
    alias /var/www/webserver/app/static;
    expires 30d;
  }
}
```

Ejemplo 5.2: Contenido del fichero *webserver.conf*.

Este archivo define la configuración del servidor proxy. Configuramos que escuche por el puerto 80 (puerto por defecto para peticiones HTTP) y dejamos vacío el nombre del servidor, "server_name", puesto que no existe dominio aún para nuestra aplicación. Establecemos los ficheros donde escribir nuestros *logs*, los cuales han sido expuestos en el *Dockerfile*, y configuramos el comportamiento del servidor proxy según las rutas a las que accedan las peticiones HTTP entrantes. Entendemos las rutas

como todo lo que siga al dominio, e.g. en una petición a "http://www.google.com/", la ruta sería "/".

Primero, configuramos el comportamiento para la ruta raíz, "/", el cual será sobrescrito si llegan peticiones a la ruta para ficheros estáticos, "/static". Una petición no estática será redirigida al contenedor corriendo la parte dinámica de la aplicación web. Si vemos el valor asignado a "proxy_pass", vemos que está definido con el nombre "webserver-dynamic". Este nombre está definido en la ejecución de la imagen como contenedor, a través del parámetro "–link", de igual manera que se especificó el servidor MySQL para la parte dinámica. El puerto utilizado es el 5000, que es el puerto que expusimos en el contenedor de la parte dinámica.

Las peticiones estáticas son resueltas de manera instantánea, sin ningún tipo de redirección, a través del comando "alias". Con el comando "expires" estamos diciendo que los ficheros pueden quedarse en la *caché*[5] del navegador hasta 30 días.

El script *run.sh* también enlaza el puerto 80 de la máquina con el puerto 80 del contenedor, para permitir accesos al contenedor de manera transparente a través de peticiones HTTP, e.g. "http://www.dominio.com/". Cuando queramos hacer la aplicación web accesible desde afuera de la red local, habrá que configurar en el router una redirección de puertos.

5.4. Infraestructura de la aplicación web

Veamos ahora un diagrama para entender mejor cómo es la comunicación entre los distintos componentes que acabamos de desplegar.

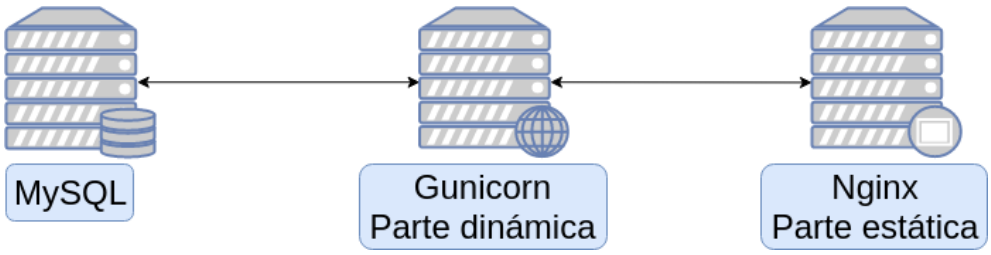

Figura 5.1: Diagrama de comunicación entre los contenedores.

Todas las peticiones que nos lleguen a la aplicación web van a entrar por Nginx, que se encargará de redirigirlas a Gunicorn en caso de no ser peticiones para ficheros

[5]En informática, una caché es un componente de hardware o software que almacena datos para que las solicitudes futuras de esos datos se puedan atender con mayor rapidez; los datos almacenados en un caché pueden ser el resultado de un cálculo anterior o el duplicado de datos almacenados en otro lugar, generalmente, de velocidad de acceso más rápido.

estáticos. Gunicorn, que corre la aplicación Flask, se comunicará con la base de datos MySQL cuando haga falta.

En los anexos podemos encontrar cómo es la estructura del código de la aplicación web y cómo ejecutar Spacebrew.

Capítulo 6

Limitaciones de Spacebrew

Contenido

6.1. Pensado para red local

Durante la realización del proyecto se han ido encontrado una serie de barreras impuestas por limitaciones de la versión de Spacebrew para NodeJS [2]. Para empezar, sólo permite crear un servidor Spacebrew que escuche a *localhost*. Necesitamos que los clientes que crea el *front-end* para cada usuario sean capaz de comunicarse con Spacebrew, estén donde estén. Para esto, es necesario que el servidor Spacebrew esté escuchando a todas las peticiones que le lleguen a la red, es decir, a la dirección 0.0.0.0.

Para conseguir esto, no ha quedado otra alternativa que modificar el código fuente. El fichero en cuestión es *spacebrew/node_ server.js* y podemos ver a continuación la modificación necesaria, que se encuentra casi al final del fichero.

```
1         persist_configs = {
2             // "host": "localhost",
3             "host": "0.0.0.0", //// Modificación Mauro
4             "port": defaultPort,
5             "logLevel": logger.debugLevel
6         }
```

No se ha incluido Spacebrew para NodeJS con esta modificación en el código fuente, puesto que se trata de un cambio muy sencillo. Sólo con esto, cualquiera en

la red local puede acceder a nuestro servidor Spacebrew. Si queremos que cualquiera pueda acceder desde fuera de la red local, hará falta hacer una redirección de puertos para el puerto 9000 o cualquiera que elijamos para ejecutar el servidor Spacebrew.

6.2. Limitación de *getClient*

La función *getClient* que trae el mix-in de Spacebrew Admin, la cual vimos en el ejemplo 2.4 de la página 23, requiere dos parámetros de entrada: el nombre del cliente y la dirección con la que se registró en el servidor Spacebrew. El primer parámetro es trivial, pero el segundo resultó no serlo tanto.

Cuando se realizaban las pruebas de la aplicación web en la máquina utilizada para desarrollar la aplicación web, había un cliente que actuaba como la aplicación de Processing, imitando su comportamiento. Puesto que el cliente existía en la misma máquina que el servidor Spacebrew, se registraba con la dirección 127.0.0.1 (*localhost*). Esta manera que tiene Spacebrew de registrar a los clientes resultaba confusa para encontrar a los clientes de los usuarios a través de la función *getClient*, no quedaba claro con qué dirección iban a estar registrados.

Sin embargo, se llegó a la conclusión de que la función está hecha así porque pueden existir varios clientes con el mismo nombre conectados al mismo servidor Spacebrew y la manera que tiene de diferenciarlos es a través de la dirección con la que se conectan. Según la infraestructura de nuestro proyecto, cada cliente va a tener un nombre único. Por lo tanto, se ha añadido una nueva función al mix-in de Spacebrew Admin: *getClientByName*.

```
1  /**
2   * Returns the client that matches the name and remoteAddress
      parameters queried.
3   *
4   * @param  {String} name            Name of the client application
5   * @param  {String} remoteAddress   IP address of the client apps
6   * @return {Object}                 Object featuring all client
      config information
7   *
8   * @memberOf Spacebrew.Admin
9   * @public
10  */
11 Spacebrew.Admin.getClient = function (name, remoteAddress){
12   var client;
13
14   for( var j = 0; j < this.admin.clients.length; j++ ){
15     client = this.admin.clients[j];
16     if ( client.name === name && client.remoteAddress ===
       remoteAddress ) {
17       return client;
```

```
18      }
19    }
20  }
```

Esta función está incluida en el código fuente de la aplicación, en el fichero *webserver/app/static/js/sb-admin-0.1.5.js*, y está escrita con el mismo formato que todas las demás de la librería. En ese directorio se encuentran todos los ficheros de la versión de Spacebrew para JavaScript.

Por lo tanto, nuestra aplicación utilizará esta nueva función, *getClientByName*, en vez de la que trae el mix-in, *getClient*.

6.3. Falta de soporte SSL

Spacebrew no tiene soporte para peticiones HTTP cifradas con SSL/TLS[1] (HTTPS). Lamentablemente, esto se descubrió después de haber preparado la aplicación web para correr con peticiones HTTPS. Por tanto, se decidió invertir tiempo e investigar el código para ver si tenía fácil arreglo, pero no hubo suerte.

El siguiente intento fue encapsular el servidor Spacebrew que lanzamos con NodeJS en un contenedor provisionado con Nginx, de tal manera que Nginx pusiera la capa SSL/TLS en el servidor Spacebrew. Esto se consiguió y el código para crear la imagen Docker y ejecutar el contenedor se encuentran en *webserver/docker/spacebrew*. Sin embargo, el código de los clientes, Spacebrew para JavaScript, no está preparado para conectarse al servidor con esta capa de cifrado.

Por lo tanto, por falta de tiempo, se decidió desistir. Además, no tenemos asegurados que la aplicación de Processing pueda comunicarse con Spacebrew a través de HTTPS, por lo que, aunque se consiguiera, podría ser en vano. No obstante, todo el código está ahí para futuro uso, incluida la configuración de la aplicación web para que pueda aceptar peticiones HTTPS.

Si deseamos correr la aplicación web con SSL/TSL, basta con ejecutar el script de ejecución de la parte estática de la aplicación web con un par de argumentos, por ejemplo:

```
1 $ ./webserver/docker/webserver/static/run.sh --ssl $HOME/certs/
```

Pasando el argumento "–ssl" estamos pidiendo que el contenedor sea preparado con la configuración necesaria para que la aplicación sea accedida a través de HTTPS. También hace falta especificar el camino absoluto al directorio en el que se encuentran los certificados SSL del servidor para poder realizar las criptografías. La configuración de Nginx espera que estos archivos tengan los nombres "key.pem"

[1]TLS (del inglés *Transport Layer Security*, en español seguridad de la capa de transporte) y su antecesor SSL (en inglés *Secure Sockets Layer*, en español capa de puertos seguros) son protocolos criptográficos que proporcionan comunicaciones seguras por una red, comúnmente Internet.

y “cert.pem”. A modo de prueba, estos certificados se pueden crear con el programa *openssl* [26].

Conclusiones y futuras mejoras

Este proyecto ha servido como una buena experiencia para el desarrollo de aplicaciones web. Empezando desde una simple idea para mejorar la experiencia de los alumnos en una asignatura hasta un proyecto bien consolidado en tecnologías modernas y con una comunidad enorme detrás para mantenerlas.

El desarrollo de Spacebrew, sin embargo, no tiene mucho desarrollo actualmente, pero tiene una pequeña comunidad que, de vez en cuando, va aportando mejoras. Como hemos visto, esto ha supuesto una serie de inconveniencias. La aplicación no ha sido probada para un entorno como este proyecto, le faltan cosas por refinar, pero se ha conseguido sobrepasarlas con una buena labor de ingeniería informática.

La aplicación web se considera que está lista para ser usada, a falta de especificar un dominio para ella y realizar la redirección de puertos en el router de la red donde vaya a ejecutarse. Aún así, hay cosas que podrían mejorarse.

A nivel visual de la página web, hace falta algo para remarcar que pertenece a la Universidad de Málaga. Por ejemplo, se intentó añadir el logo de la UMA en la parte inferior de la página, de manera que se quedase fijado, pero no hubo éxito. Otra idea para ello es la utilización de colores más acordes a la Escuela de Ingeniería Industriales, que es a quien va destinado este proyecto.

Se podría mejorar también la visualización del formulario de "Práctica nueva" en la pantalla "Inicio", de tal manera que ocupase todo el ancho posible del panel donde está contenido, y de la gráfica de resultados, para que los valores del eje X (el tiempo) tuviesen una distancia fija, e.g. de 10 en 10 milisegundos.

Siguiendo con el *front-end*, el código favorecería mucho que se usara de manera estándar jQuery y se empezara a usar React. En el código de JavaScript se utilizan funciones más antiguas que ocupan más espacio en el código, estas tecnologías mejoran esto y, además, introducen varias mejoras para la generación dinámica de las páginas. También sería interesante aliviar el trabajo de Jinja de la tablas, e.g. "Resultados anteriores", haciendo más uso AJAX. Es decir, en vez de generar y poblar la tabla con Jinja, Jinja simplemente crearía la tabla vacía y luego el *front-end*, con AJAX, solicitaría los datos para rellenar la tabla.

En cuanto a mejoras para el *back-end*, habría que invertir tiempo en utilizar el mix-in *RoleMixin* de la extensión Flask-Security. Esto limpiaría parte del código

que se encarga de comprobar si un usuario es administrador. Se intentó incluir este mix-in, pero no hubo éxito.

También hubiese sido interesante conseguir una serie de ideas que tenía mi tutor que no eran obligatorias conseguir. La primera es que los administradores reciban un email de que un usuario se ha registrado y, en el cuerpo del email, aparezca un enlace que, con simplemente abrir, se active el usuario en cuestión. Realmente, el código para conseguir esto está disponible, sólo es cuestión de ponerlo todo en pie. Por falta de tiempo no se ha intentado.

La otra idea era que se consiguiesen automáticamente los resultados de las simulaciones que no tenían y avisar posteriormente via email a los usuarios que las hubiesen solicitado. Esto es más complicado de llevar a cabo, pero como idea se podría poner una tarea en la aplicación que se encargue de buscar cada X tiempo en la base de datos simulaciones que no tienen resultados y enviar las solicitudes a través de un cliente Spacebrew que crearía la aplicación web. Para esto, haría falta utilizar la versión de Spacebrew para Python.

Un caso claro de mejora el uso de las *cookies* que se ha descrito mientras se explicaban las pantallas "Inicio", "Resultados" y "Crear permiso". Aquí habría que hacer una mejor labor para protegerse cuando no se hayan conseguido las *cookies* y habría que evitar todo el proceso de conseguirlas cuando ya están disponibles.

Por último, para mejorar los tiempos de respuesta de los ficheros estáticos y los tiempos para cargar el *front-end*, sería interesante minificar los ficheros **.js* y **.css*. Minificar es un proceso mediante el cual eliminamos todos los espacios en blanco y acortamos los nombres de variables para conseguir ficheros más pequeños. Existen varias herramientas preparadas para realizar esto.

Bibliografía

[1] Página oficial de Spacebrew.
http://docs.spacebrew.cc/
Última consulta en: junio 2018

[2] GitHub oficial de las versiones de Spacebrew para diferentes lenguajes.
https://github.com/Spacebrew
Última consulta en: septiembre 2018

[3] Página oficial de Processing.
https://processing.org
Última consulta en: mayo 2018

[4] Tutorial para comunicarse con Spacebrew a través de Processing.
http://docs.spacebrew.cc/tutorials/2015/10/12/basics-spacebrew-processing
Última consulta en: mayo 2018

[5] Página oficial de JSON.
http://www.json.org
Última consulta en: mayo 2018

[6] Página oficial de Python.
https://www.python.org
Última consulta en: agosto 2018

[7] Tutorial para crear la aplicación web con Flask. *The Flask Mega-Tutorial.*
https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-i-hello-world
Última consulta en: septiembre 2018

[8] Página oficial de Flask.
http://flask.pocoo.org
Última consulta en: agosto 2018

[9] Página oficial de Jinja.
http://jinja.pocoo.org
Última consulta en: julio 2018

[10] Página oficial de Flask SQLAlchemy.
http://flask-sqlalchemy.pocoo.org/2.3/
Última consulta en: agosto 2018

[11] Página oficial de Flask Login.
https://flask-login.readthedocs.io/en/latest/
Última consulta en: julio 2018

[12] Página oficial de Flask Moment.
https://github.com/miguelgrinberg/Flask-Moment
Última consulta en: julio 2018

[13] Manual oficial de pickle para serializar objetos Python.
https://docs.python.org/3/library/pickle.html
Última consulta en: agosto 2018

[14] Documentación oficial de Bootstrap v.3.3.6.
https://bootstrapdocs.com/v3.3.6/docs/getting-started/
Última consulta en: septiembre 2018

[15] Documentación oficial de la API de jQuery.
https://api.jquery.com
Última consulta en: agosto 2018

[16] Página oficial de Chart.js.
https://www.chartjs.org
Última consulta en: agosto 2018

[17] Página oficial de MySQL.
https://dev.mysql.com/doc/
Última consulta en: julio 2018

[18] Página oficial de Oracle SQL Developer.
https://www.oracle.com/database/technologies/appdev/sql-developer.html
Última consulta en: julio 2018

[19] Docker Hub de mysql/mysql-server.
https://hub.docker.com/r/mysql/mysql-server/
Última consulta en: agosto 2018

[20] Manual oficial de la imagen Docker de MySQL Server.
https://dev.mysql.com/doc/refman/5.7/en/linux-installation-docker.html
Última consulta en: septiembre 2018

[21] Manual oficial de Docker.
https://docs.docker.com
Última consulta en: julio 2018

[22] Tutorial para instalar Docker Hypriot en una Raspberry Pi.
https://gist.github.com/tyrell/2963c6b121f79096ee0008f5a47cf347
Última consulta en: septiembre 2018

[23] Docker Hub de hypriot/rpi-mysql.
https://hub.docker.com/r/hypriot/rpi-mysql/
Última consulta en: septiembre 2018

[24] Página oficial de Gunicorn.
https://gunicorn.org
Última consulta en: agosto 2018

[25] Manual oficial de Nginx.
https://nginx.org/en/docs/
Última consulta en: septiembre 2018

[26] Tutorial para ejecutar aplicaciones Flask con SSL/TLS.

https://blog.miguelgrinberg.com/post/running-your-flask-application-over-https

Última consulta en: agosto 2018

[27] GitHub oficial de WWW SQL Designer.

https://github.com/ondras/wwwsqldesigner

Última consulta en: agosto 2018

[28] Página oficial de Git.

https://git-scm.com

Última consulta en: mayo 2018

[29] Página oficial de Bitbucket.

https://bitbucket.org/product

Última consulta en: junio 2018

[30] Una pequeña introducción a los entornos virtuales de Python. *ENTORNO VIRTUAL EN PYTHON. CÓMO Y PARA QUÉ. UBUNTU, LINUX MINT, ETC*

https://www.atareao.es/como/entorno-virtual-en-python-como-y-para-que/

Última consulta en: junio 2018

[31] Dockerfile de una imagen Docker que trae las librerías para conectarse con MySQL.

https://github.com/tnir/mysqlclient/blob/master/Dockerfile

Última consulta en: agosto 2018

[32] Página oficial de PyCharm.

https://www.jetbrains.com/help/pycharm/install-and-set-up-pycharm.html

Última consulta en: julio 2018

[33] Cómo instalar el driver mysql-connector-java para Oracle SQL Developer.

https://stackoverflow.com/a/22169819

Última consulta en: agosto 2018

[34] Juan Cano López, Ingeniero en Automática y Electrónica Industrial (2017) *Laboratorio Remoto de Automática. Una Solución de Bajo Coste basada en Raspberry Pi y Arduino.* E.T.S.I Industrial, Universidad de Málaga.

[35] Louis Manzano Harmer, Grado en Ingeniería en Tecnologías Industriales (2018) *Desarrollo de una Solución HW-SW de Bajo Coste para la Realización Remota de Prácticas de Control Automático.* Escuela de Ingenierías Industriales, Universidad de Málaga.

Anexos

Anexo A

Imágenes de la aplicación en dispositivos móviles

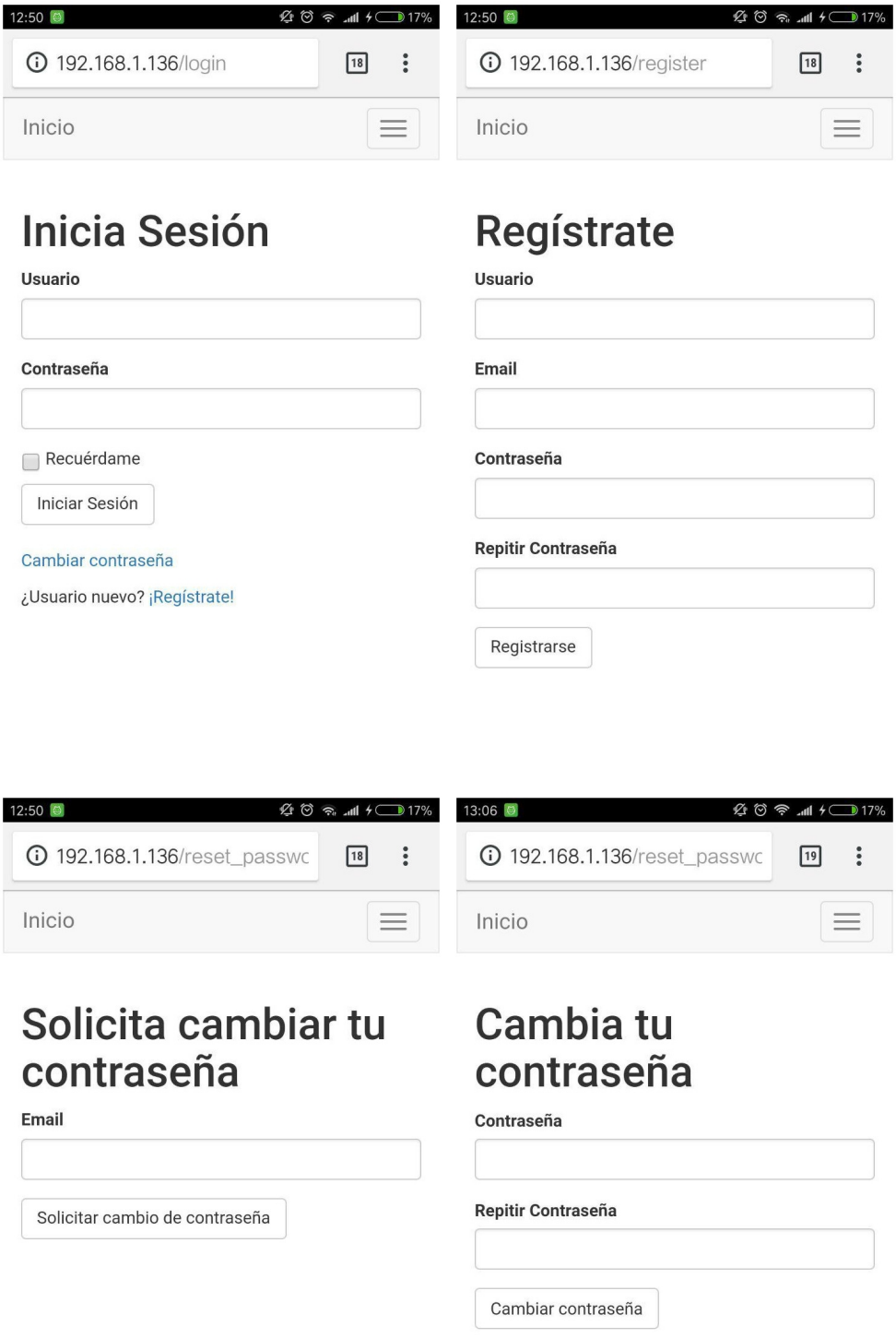

Figura A.1: Las primeras páginas que ve un usuario cuando accede desde un smartphone.

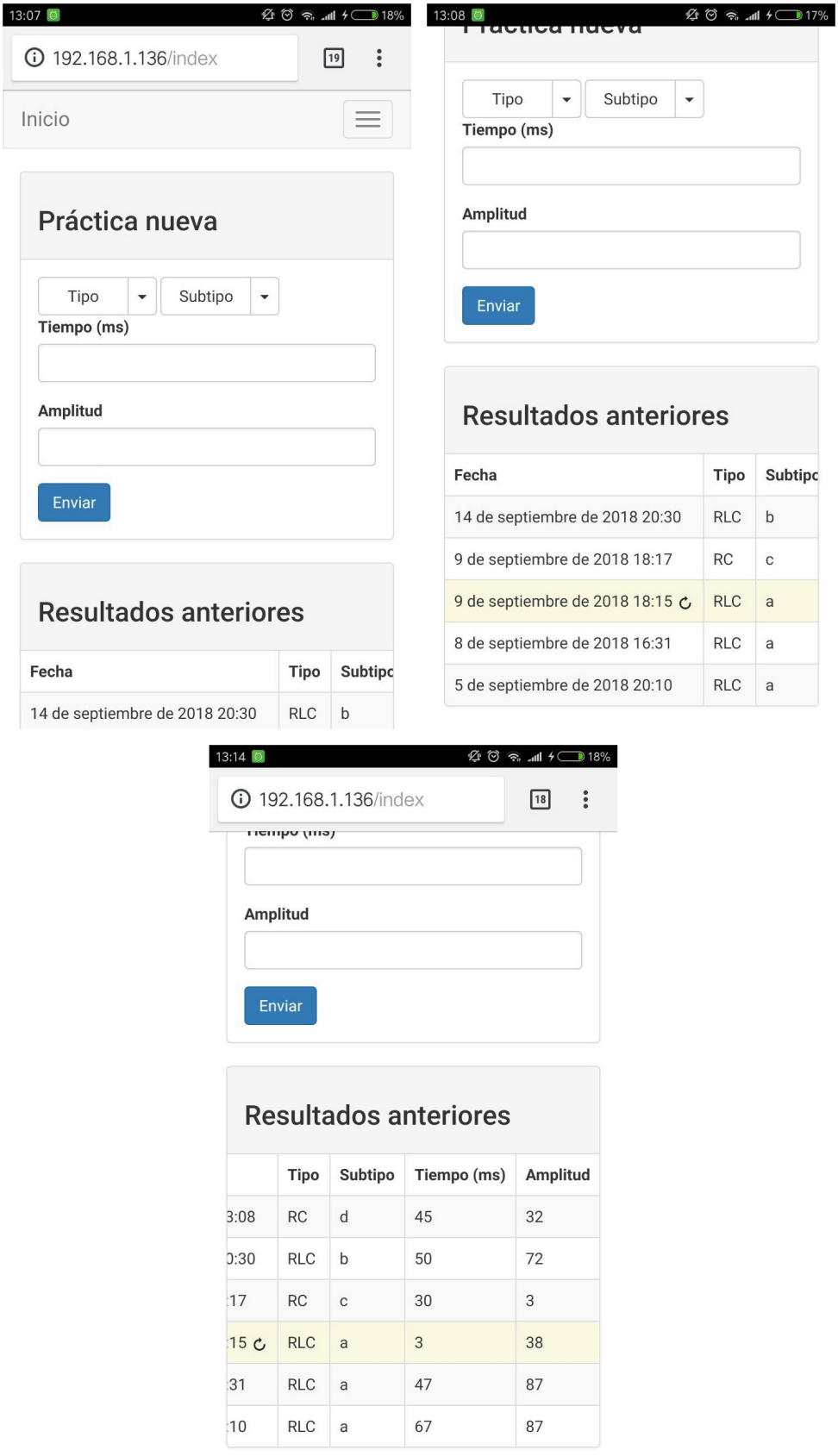

Figura A.2: Las página “Inicio” desde un smartphone. La tabla de “Resultados anteriores” es desplazable.

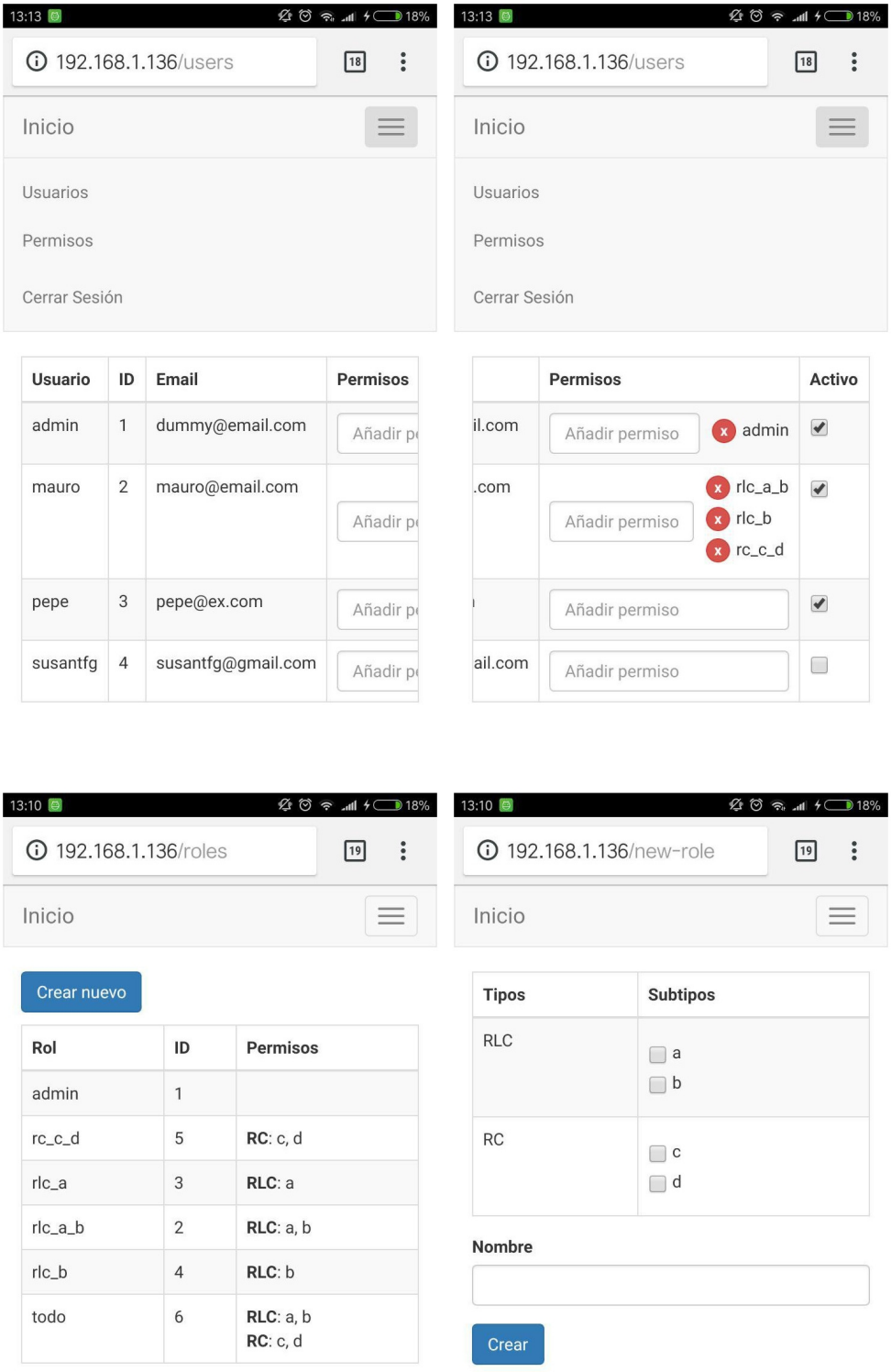

Figura A.3: Las páginas administrativas desde un smartphone. La tabla de usuarios es desplazable. Vemos también como la barra de navegación está contenida en el botón superior derecho.

Anexo B

Estructura de la aplicación Flask

En toda la memoria no se ha hablado de cómo está estructura la aplicación Flask. Por una parte por falta de tiempo y por otra por la limitación de páginas del bloque de contenido de la memoria.

Sin embargo, realmente no tiene nada del otro mundo. La realización del *back-end* está fuertemente inspirado en un gran tutorial disponible en internet: *The Flask Mega-Tutorial* [7]. Se tarda alrededor de una semana en pasarse el tutorial y con esto se entiende perfectamente la estructura de la aplicación. También, en el capítulo 15 del tutorial, se enseña cómo mantener el fichero *webserver/requirements.txt*, utilizado para la preparar la imagen Docker de la parte dinámica de la aplicación web.

Lo único que no enseña el tutorial es a desarrollar la aplicación con PyCharm, pero es muy fácil es configurar. En la propia página web [32] hay instrucciones sobre cómo instalarlo. Cuando arranquemos el IDE por primera vez, habrá que especificar la carpeta *webserver* donde está todo el código y automáticamente nos dejará todo listo. Con este IDE podemos desplegar la aplicación en nuestra propia máquina para poder desarrollar nuevas ideas y depurar fallos.

En el fichero *webserver/webserver.py*, encontramos este fragmento de código:

```
1 if __name__ == '__main__':
2     app.run(debug=True
3             # , host='0.0.0.0'
4             )
```

Cuando lanzamos la aplicación con PyCharm, esta condición se cumple y se lanza la aplicación de modo depurable. Esto significa que podemos hacer modificaciones en los ficheros Python y se compilarán de nuevo para que podamos ver inmediatamente los cambios. Además, si quitamos la almohadilla, podemos hacer que la aplicación corra con la dirección 0.0.0.0. Esto es útil para probar el comportamiento *responsive* desde dispositivos móviles.

Por último, hemos visto en la configuración del fichero *.env* para la parte dinámica que especificamos un valor "DATABASE_URL". Si nos fijamos en el archivo

webserver/config.py, vemos que hay un valor por defecto en caso de que no exista. El valor por defecto hace que la aplicación web utilice una base de datos SQLite. Esta base de datos se guarda en un fichero *webserver/app.db* y, en caso de que no exista, se prepara con todas las tablas, restricciones, índices, etc que tendría la base de datos MySQL. Esto ocurre gracias al archivo de migración dentro de *webserver/migratios*, en el tutorial también explican esto.

Lo único que no se crea en la base de datos SQLite son el usuario y el rol admin. Para añadir estos datos, basta con acceder a la base de datos con *sqlite3* a través de la terminal y ejecutar los últimos comandos del script *webserver/datamodels/tables.sql*.

Anexo C

Ejecutar Spacebrew con NodeJS

En el GitHub de Spacebrew [2] vienen instrucciones detalladas para preparar la ejecución de Spacebrew con NodeJS. Una vez se tenga todo listo, hay dos opciones lanzar Spacebrew. A nosotros nos resulta más conveniente *node_ server_ forever.js*, ya que hará que el servidor Spacebrew se reinicie automáticamente en caso de fallo y los logs se guardarán en un fichero.

```
1 $ nodejs node_server_forever.js
```

Hay que recordar que, si queremos que nuestro servidor Spacebrew reciba peticiones desde fuera de la máquina donde se esté ejecutando, tenemos que realizar la modificación descrita en la página 71.

Anexo D

Cómo utilizar Oracle SQL Developer

El uso de SQL Developer para manejar la base de datos MySQL puede llegar a ser muy útil para visualizar qué datos existen e incluso poder manipularlos. Para esto, primero tenemos que instalar el programa. En internet hay variedad de tutoriales, por si acaso no nos sirvieran las instrucciones de la página web oficial [18].

Una vez instalado, tenemos que instalar y configurar el driver *mysql-connector-java* [33]. Ahora, podemos añadir la conexión a la base de datos MySQL corriendo en un contenedor en nuestra máquina tal como se ve en la imagen debajo. La contraseña viene especificada en el fichero *webserver/docker/mysql/mysql.sh*.

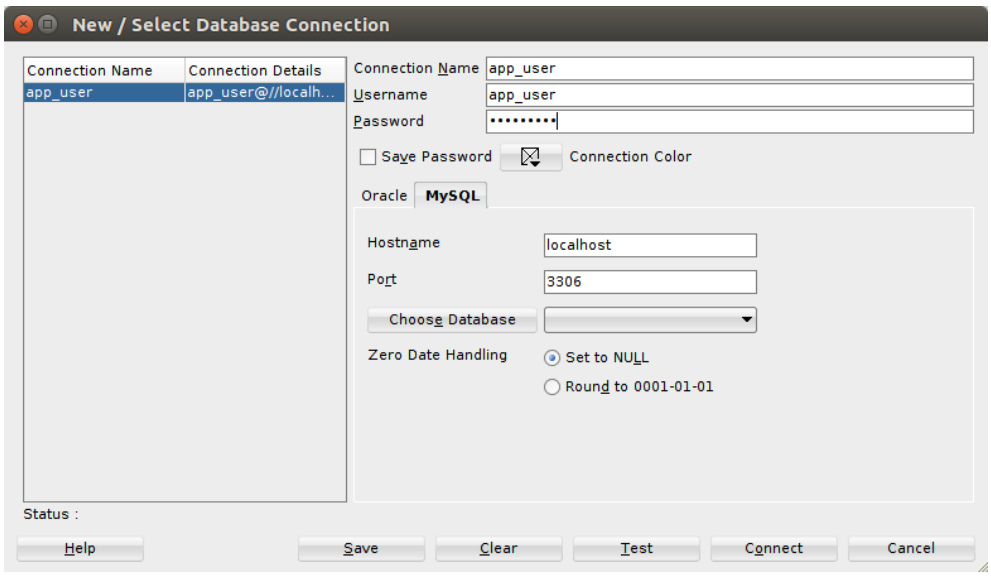

Figura D.1: Configuración para conectarse a MySQL desde SQL Developer.