Capítulo 4

INTEGRACIÓN DE MONGODB Y MATPLOTLIB EN KIBOTICS

En este capítulo se describe el estado inicial de Kibotics Webserver, tanto arquitectura de la aplicación web como tecnologías ya utilizadas. Además, se explica la implementación de MongoDB como base de datos, y de Matplotlib como generador de gráficas.

4.1. Estado inicial de Kibotics Webserver

En esta sección se describe la arquitectura que poseía Kibotics Webserver al comienzo del desarrollo de este proyecto.

4.1.1. Arquitectura

Kibotics Webserver es un servicio web desarrollado en Django. Compuesto por varias partes:

- Kibotics Webserver: Servicio web desarrollado en Django, es el centro de Kibotics y el generador de los logs sobre los que se trabajará en este proyecto fin de carrera.

- Kibotics websim: Simulador robótico que se ejecuta unicamente en el lado del cliente. Desarrollado en herramientas como A-Frame, HTML5, JavaScript, entre otras, permite a los usuarios aprender los fundamentos de la programacón robótica y visión artificial.

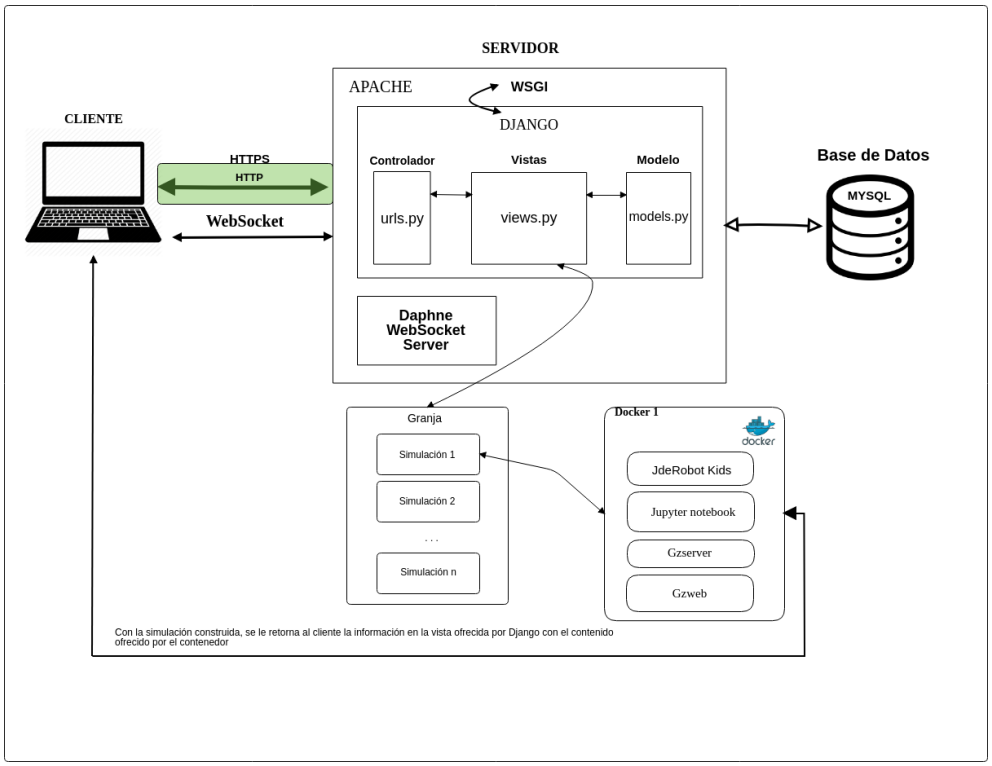

Figura 4.1: Arquitectura Kibotics.

Cuando un usuario accede a la plataforma lo hace a través del protocolo HTTPS hacia el servidor Django principal. Este servidor es el orquestador de eventos que tienen lugar en el resto de máquinas.

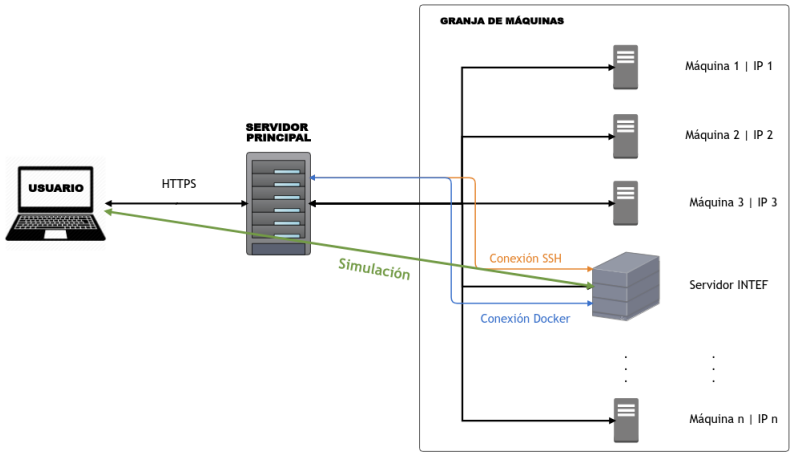

Figura 4.2: Infraestructura Kibotics.

El centro de Kibotics son las simulaciones. Una simulación está compuesta principalmente de dos partes: el código que programa el estudiante (izquierda) y la ventana de simulación (derecha), en la que el usuario verá su código ejecutado en tiempo real.

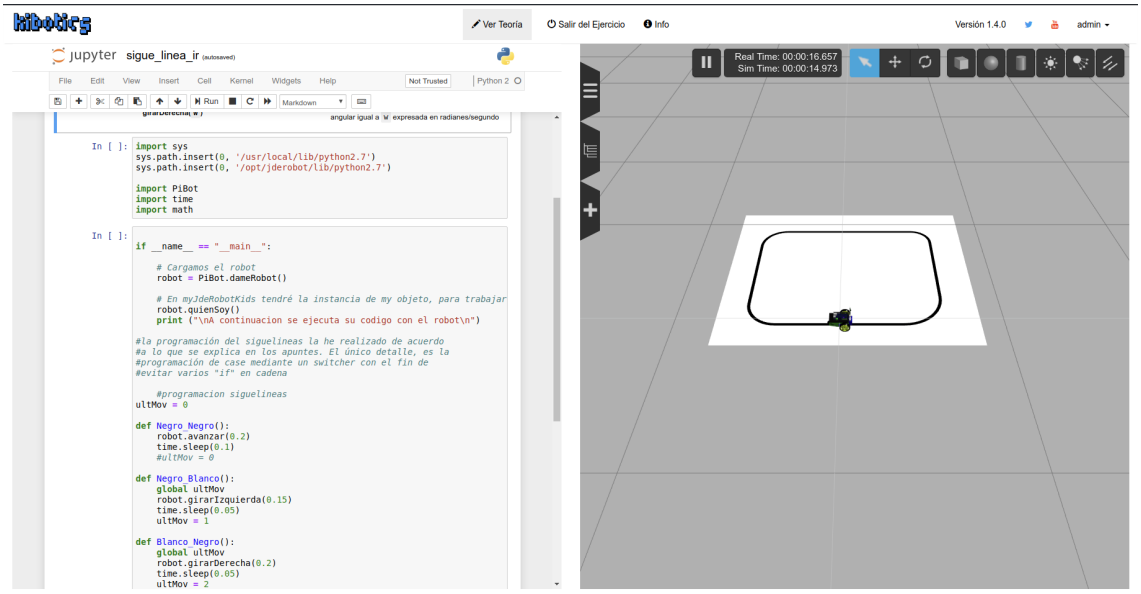

Figura 4.3: Simulador Kibotics.

4.1.2. Logs

Los logs generados por la aplicación Django se guardan en una serie de archivos indexados en el servidor con el formato YYYY-MM-DD-log.txt. Teniendo así, un fichero por día con todos los eventos registrados.

Esta metodología disponía de un sistema numeral de códigos para identificar el evento que había generado cada registro de log. Cada campo de un mismo registro estaba separado por la cadena de caracteres " | ".

Estos códigos y su estructura son los siguientes:

- Log in: "1 | date | user name | user IP | HTTP_USER_AGENT"
- Log out: "2 | date | user name | user IP | HTTP_USER_AGENT"
- Comienzo ejecicio: "3 | date | user name | user IP | simulation type | exercise ID | host IP | HTTP_USER_AGENT"
- Fin ejecicio: "4 | date | user name | user IP | simulation type | exercise ID | host IP | HTTP_USER_AGENT"
- Error 500: "5 | 500 Internal Server Error"

Estos log, se generaban en el servicio Django yse guardaban en los ficheros con sentencias Python similares a la siguiente:

```
log = open(DIRECTORY + "/logs/" + str(date.today()) + "-log.txt", "a")

traze = "1 | " + str(datetime.now().strftime("%d/%m/%Y %H:%M:%S"))
+ " | " + username + " | " + client\_ip  + " | " + user\_agent + "\\n"\\

log.write(traze)

log.close()
```

Además de estos logs, se disponía de los generados de forma automática por Apache. El cual, separa los eventos registrados en dos ficheros, uno con la salida general de la aplicación, asociada a los prints y excepciones producidas. Y el otro, es un archivo de acceso al servidor que muestra las peticiones HTTP que este ha recibido.

4.2. Desarrollo local

En está sección se describe la evolución que han sufrido los logs de la aplicación. Así como una primera prueba de concepto de la herramienta de analíticas.

4.2.1. MongoDB en Kibotics Webserver

Kibotics webserver disponía de una tecnología muy primitiva de trazabilidad, con limitados eventos y una distribución en ficheros txt. La cual, limitaba la explotación masiva de estos datos para la generación de estadísticas útiles.

En Kibotics se espera un crecimiento muy notable en los usuarios. Por lo tanto, la capacidad de procesamiento, almacenamiento y consulta de los logs debía aumentar, haciendo uso de ficheros de texto plano no se podrá conseguir la velocidad de procesamiento necesaria.
Se decide cambiar a un motor de bases de datos, MongoDB, una base de datos externa a Django, a la que se realizarán consultas de Python mediante la librería pymongo. La cual ofrece las herramientas de consulta y guardado necesarias para una interacción ágil con MongoDB.

Para realizar esta migración de tecnología, es necesario primero instalar MongoDB:

```
$ sudo apt-get install mongodb-org
```

Una vez instalado, es necesario iniciar el proceso para levantar el servicio MongoDB:

```
$ sudo systemctl start mongod
```

Adicionalmente a esto, podemos reiniciar o parar el servicio con los siguientes comandos respectivamente:

```
$ sudo systemctl restart mongod
$ sudo systemctl stop mongod
```

Con el servicio MongoDB ya instalado lo único necesario para completar la migración es modificar las sondas para evitar que escriban en ficheros. Haciendo uso de la librería pymongo la tarea se simplifica mucho. Importamos la librería y abrimos la conexión con la base de datos, para ello:

```
import pymongo
myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["kiboticsDDBB"]
```

Ya con las conexiones necesarias realizadas, las sondas se transforman a, por ejemplo, las de nueva sesión y simulación:

```
# Nueva sesión
mydict = {
    "date" : datetime_object_test,
    "username" : "USERNAME_TEST",
    "client_ip" : "CLIENT_IP_TEST",
    "user_agent" : "USER_AGENT_TEST"
}
mydb["newSession"].insert_one(mydict)


# Nueva simulación
mydict = {
    "date" : datetime_object,
    "username" : "USERNAME_TEST",
    "client_ip" : "CLIENT_IP_TEST",
    "simulation_type" : "SIMULATION_TYPE_TEST",
    "exercise_id" : "EXERCISE_ID_TEST",
    "host_ip" : "HOST_IP_TEST",
    "container_id" : "CONTAINER_ID_TEST",
    "user_agent" : "USER_AGENT_TEST"
}
mydb["newSimulation"].insert_one(mydict)
```

Con estos pasos, el logueo en la nueva base de datos MongoDB está completa, para recuperar la información y tratarla en python, una vez más, haciendo uso de pymongo las Queries de búsqueda serán:

```
# Query nueva sesión
dataNSES = mydb["newSession"].find({
    "username" : {'$regex' : "USERNAME_TEST"},
    "date" : {'$lte': first_day_test, '$gte': last_day_test}
});

# Query fin de sesión
dataESES = mydb["endSession"].find({
    "username" : {'$regex' : "USERNAME_TEST"},
    "date" : {'$lte': first_day_test, '$gte': last_day_test}
});

# Query nueva simulación
dataNSIM = mydb["newSimulation"].find({
    "username" : {'$regex' : "USERNAME_TEST"},
    "date" : {'$lte': first_day_test, '$gte': last_day_test}
});

# Query fin de simulación
dataESIM = mydb["endSimulation"].find({
    "username" : {'$regex' : "USERNAME_TEST"},
    "date" : {'$lte': first_day_test, '$gte': last_day_test}
});
```

Como se puede observar estas queries o sentencias de búsqueda filtran tanto por usuarios como por rangos de fechas. Esto aporta mucha flexibilidad para posteriormente simplemente los datos necesarios y evitar tener q recorrer ficheros extra descartando registros de log.

4.2.2. Matplotlib en Kibotics Webserver

Matplotlib es una librería Python de generación de gráficos. Haciendo uso de ella, se han generado todas las visualizaciones necesarias para el primer prototipo.

Estas visualizaciones inicialmente se han separado en dos secciones, analíticas de simulaciones y sesiones, ambas pueden ser filtradas tanto por usuarios como rangos de fechas.

Con el guardado de datos que se ha realizado surge un problema inicial, los datos de inicio y fin, tanto para las sesiones como para las simulaciones, están separados en distintas tablas de MongoDB. Por lo tanto, para hacer una relación entre ellos es necesario cruzarlos en Python para unificarlos en un único evento de sesión o simulación.
Para esto, se creó la funcionalidad para unificar estos registros para que proporcionasen información más útil:

```
def formatDatesUser(newData, endData):
    USERS = newData.distinct("username")
    newData.sort([('Username', -1), ('date', -1)])
    endData.sort([('Username', -1), ('date', -1)])
    Dict = {}

    for user in USERS:
        for d in newData:
            for dd in endData:
                if(dd['username'] == d['username'] == user):
                    if(d['date']<dd['date']):
                        if(user not in Dict):
                            Dict[user] = {d['date'] :
                                {
                                    "totalTime" : dd['date']-d['date'],
                                    "endTime" : dd['date']
                                }
                            }
                        else:
                            Dict[user].update({d['date'] :
                                {
                                    "totalTime" : dd['date']-d['date'],
                                    "endTime": dd['date']
                                }
                            })
                    break;
                endData.rewind()
        newData.rewind()

return Dict
```

Este código extrae todos los usuarios y posteriormente recorre los registros de principio a fin buscando por el campo hora hasta encontrar la inmediatamente siguiente de cierre. Nos devolverá un diccionario de diccionarios con cada uno de los eventos sesión/simulación para cada usuario del que haya ocurrencias.

La clave de este diccionario serán los usuarios. El valor, será otro diccionario con las fechas de comienzo y fin del evento así como su duración.

Un ejemplo de respuesta sera:

```
{
    "USERNAME_TEST_1" : {
        start_date_1 : {
            "endTime" : end_date_1,
            "totalTime" : end_date_1 - start_date_1
        },
        start_date_2 : {
            "endTime" : end_date_2,
            "totalTime" : end_date_2 - start_date_2
        },

        ...
        start_date_N : {
            "endTime" : end_date_N,
            "totalTime" : end_date_N - start_date_N
        },
    },

    ...
    "USERNAME_TEST_N" : {
        start_date_1 : {
            "endTime" : end_date_1,
            "totalTime" : end_date_1 - start_date_1
        },
        start_date_2 : {
            "endTime" : end_date_2,
            "totalTime" : end_date_2 - start_date_2
        },

        ...
        start_date_N : {
            "endTime" : end_date_N,
            "totalTime" : end_date_N - start_date_N
        },
    }
}
```

Una vez con estos datos más completos, ya se puede enviar a métodos de generación de gráficas, estos tienen una estructura muy similar entre ellos:

- Primero, recorrerán los datos de entrada formateandolos a la estructura de ejes necesaria para cada una de las gráficas. Generalmente se compondrá de dos listas o arrays, uno con los datos del eje-X y otro con los referentes al eje-Y. En ciertos casos como el mapa de calor, necesitaremos una matriz de datos para la correcta representación de la información.

- Segundo, se creará la gráfica y se le añadirán los datos que se formatearon en el punto primero. Este es el paso en el que se explicitará qué tipo de gráfica se insertará para cada caso. Junto a la primera parte, es lo que más cambiará entre métodos.

- Tercero, se ajustará el diseño de la gráfica para que encaje estéticamente tanto con las otras gráficas generadas para la funcionalidad de analíticas, como con el diseño ya existente en la aplicación.

- Finalmente, ya generada la gráfica, se guardará la figura en un objeto BytesIO. Este objeto de bytes, se codificará a formato png y se devolverá por la salida del método.

Con estos objetos de imagen ya guardados, lo único que queda es devolverlos por el contexto de la respuesta HTML de Django, que es simplemente un diccionario de variables. Es a este contexto al que la platilla HTML enriquecida de Django accederá para mostrar las imágenes.

Un ejemplo de una sección en estas plantillas será:

```
...
<div class="main">
    <br><br><h2>INICIOS POR DIA DE LA SEMANA</h2>
    <hr/>

    <div class='left' >
        <h4>Sesiones</h4><br>
        <img class='analyticsIMG'src=" {{ imageWeekSesion }}"
        alt="Data not loaded properly" width=500 height=auto />
    </div>

    <div class='right'>
        <h4>Simulaciones</h4><br>
        <img class='analyticsIMG'src=" {{ imageWeekSimulation }}"
        alt="Data not loaded properly" width=500 height=auto />
    </div>
</div>
...
```

A continuación se puede ver el resultado final, con unos datos de prueba, no productivos. El prototipo nos muestra mucha información acerca de la actividad en la aplicación web.

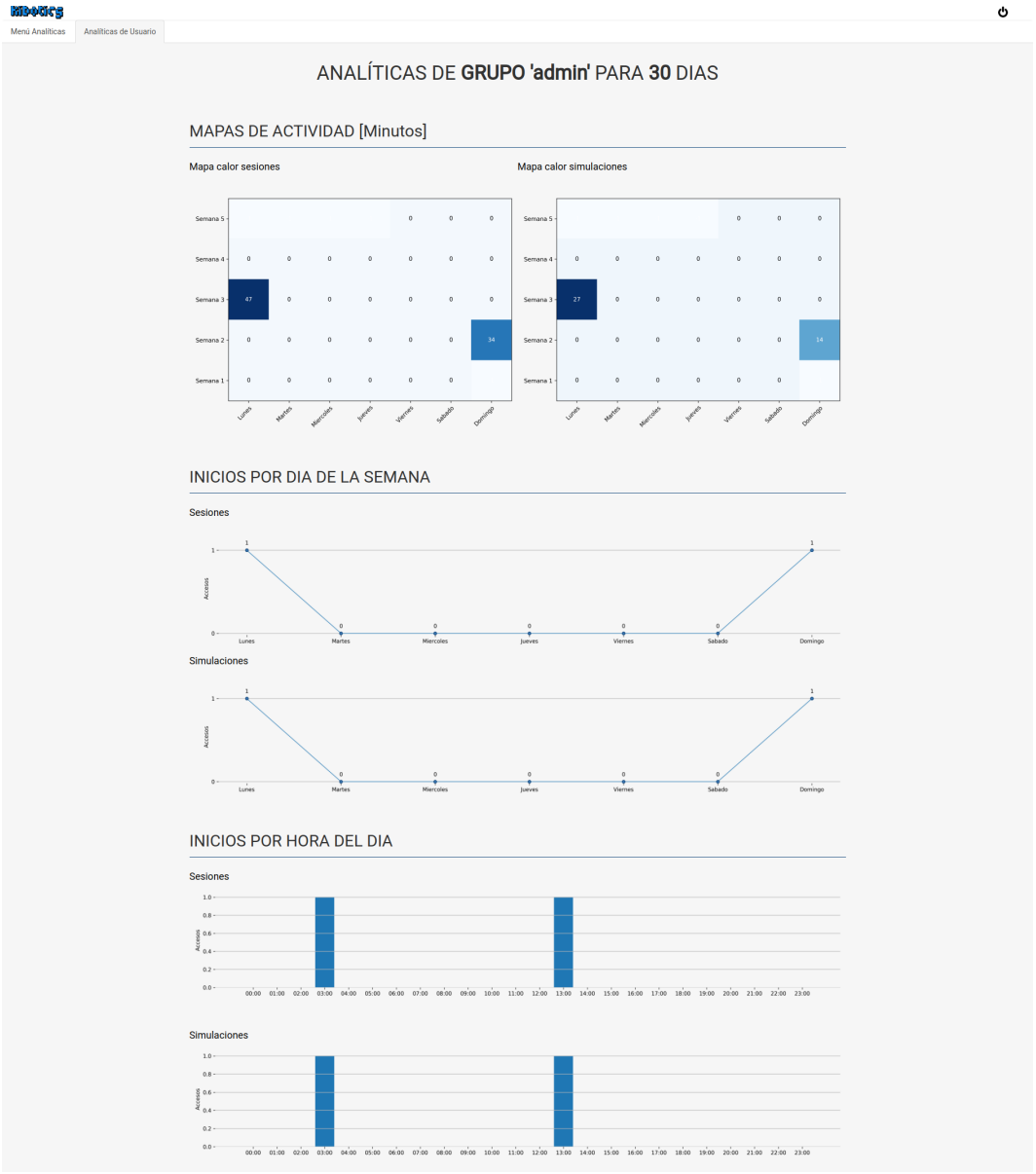

Figura 4.4: Primer prototipo parte 1.

En la primera figura, se puede observar una primera parte con un mapa de calor tanto para simulaciones como para sesiones, la cual representa la actividad en minutos para el grupo de usuarios analizado.

En una segunda parte, se representar 4 gráficas más con los accesos a sesiones y simulaciones separadas en dos grupos. Una primera agrupación con los accesos por día de la semana, a continuación, el segundo grupo con accesos divididos por la hora del día a la que fueron realizadas.

Por último en la siguiente figura, se representan las dos últimas gráficas. Una primera con los tiempos totales y medios que el grupo de usuarios o usuario ha pasado en cada uno de los ejercicios a los que ha accedido.

Finalmente, la última gráfica representa con un mapa geográfico la localización desde la que cada acción se ha realizado.

Este primer prototipo es bastante completo pero tiene ciertos inconvenientes. Primero, falta cierta información útil que podría ser representada, como desde que dispositivos acceden los usuarios.

Esta gráficas, al estar insertadas como imágenes, carecen de interactividad, la cual sería muy útil para tener información extra o poder realizar mas filtrado de los datos.

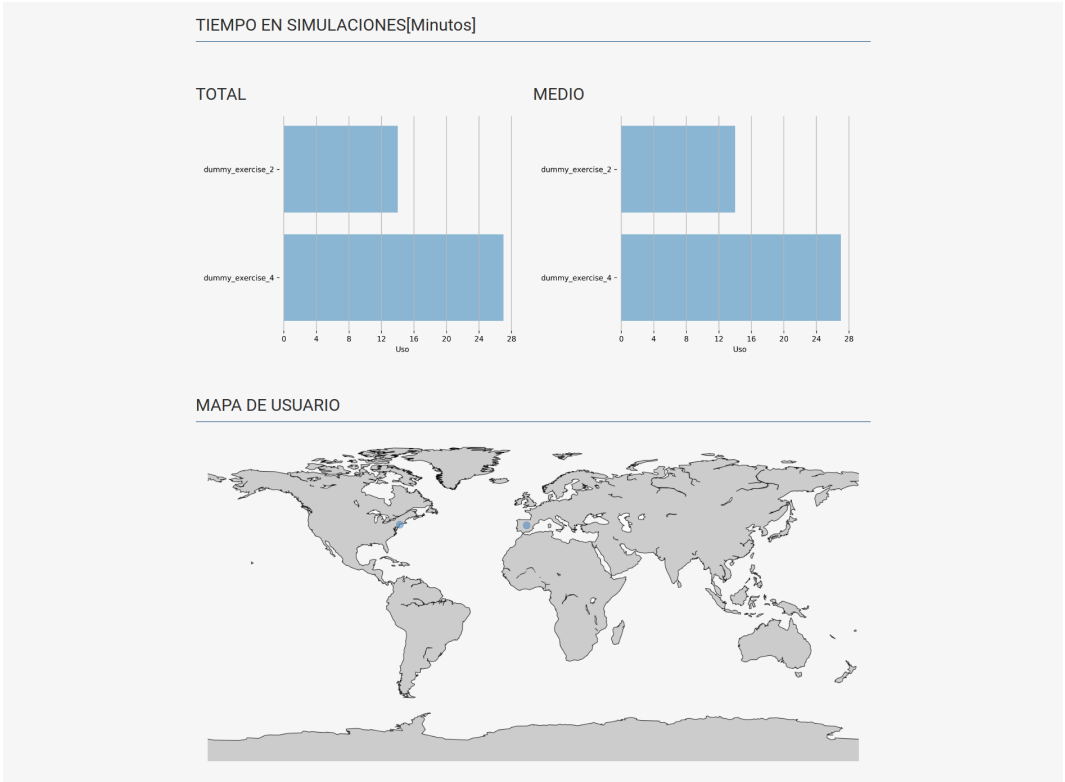

Figura 4.5: Primer prototipo parte 2.