Trabajo de Fin de Grado

Grado en Ingeniería de las Tecnologías de Telecomunicación

# Servicio de búsqueda automatizada de anuncios de viviendas de alquiler basado en Python, Telegram y SQL

Autor: Óscar León Silva

Tutor: María Teresa Ariza Gómez

**Dpto. de Ingeniería Telemática**
**Escuela Técnica Superior de Ingeniería**
**Universidad de Sevilla**

Sevilla, 2023

Trabajo de Fin de Grado

Grado en Ingeniería de las Tecnologías de Telecomunicación

# Servicio de búsqueda automatizada de anuncios de viviendas de alquiler basado en Python, Telegram y SQL


Autor:

Óscar León Silva

Tutor:

María Teresa Ariza Gómez

Profesor Titular

Dpto. Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla



Sevilla, 2023

Trabajo de Fin de Grado: Servicio de búsqueda automatizada de anuncios de viviendas de alquiler basado en Python, Telegram y SQL

Autor: Óscar León Silva
Tutor: María Teresa Ariza Gómez

El tribunal nombrado para juzgar el trabajo arriba indicado, compuesto por los siguientes profesores:

Presidente:

Vocal/es:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

# Agradecimientos

A mi padre, don *Óscar León del Río*, principal artífice de la persona que soy hoy día, tanto personal como profesionalmente. Si no fuese por él, no habría descubierto el maravilloso mundo de las telecomunicaciones, ni existiría por ende este documento. A mi pareja, doña *Macarena Sánchez Gutiérrez*, por creer siempre en mí, incluso cuando ni yo mismo lo hice. Como no, a mi hermana mayor, doña *Miriam León Silva*, por cuidarme en los buenos y sobre todo en los malos momentos. Ellos tres forman los pilares fundamentales de mi vida. También, a don *Pablo López Vílchez*, por ayudarme siempre en todo y hacer de esta etapa aún más bonita. Y por último, gracias a mí, por nunca rendirme, por pelear hasta el último minuto de cada examen, por cada hora y minuto de estudio que dediqué a aprender todo aquello que me ha hecho la persona que soy.

*Óscar León Silva*

*Sevilla, 2023*

# Resumen

La ***automatización*** de procesos es uno de los aspectos clave dentro del mundo de la ingeniería. La gran evolución sufrida durante los últimos 20 años en el mundo ingenieril, especialmente en el apartado de las telecomunicaciones y la informática, ha permitido al ser humano descargarse de tareas monótonas y tediosas a través de la aplicación de esta característica, haciendo incluso estas tareas de una forma más veloz y eficaz.

Nuestra sociedad, la española, sufre a día de hoy un grave problema en cuanto a la adquisición de la vivienda, especialmente la gente joven, a los cuales se nos resiste el hecho de poder independizarnos y comenzar nuestra vida adulta de forma completa. Actualmente vivimos una etapa en la que rebosa la especulación de la vivienda, y la idea de poder encontrar un alquiler en condiciones dignas puede ser un infierno. En la situación en la que nos encontramos, una vivienda que sale al mercado en forma de alquiler puede durar en el mismo a penas días, a veces horas incluso, debido a la enorme demanda que hay en comparación con la oferta existente. Es por ello que para lograr este objetivo, es clave ser especialmente rápido y constante, lo que conlleva pasar horas frente al ordenador cada día, consultando los anuncios de los principales portales de vivienda (*Idealista, Fotocasa, …*). Es ahí donde la característica de la automatización nombrada al comienzo, de forma que se libere al usuario del desempeño de esta tarea mediante la implementación de una herramienta la cual lleve a cabo el desarrollo de esta actividad . ¿Se imaginan escribir en un formulario tus preferencias para encontrar una vivienda y que un servidor escanease los distintos portales de forma periódica, compartiéndote en un chat o un correo anuncios acorde a tus parámetros? De eso trata esta memoria.

# Abstract

Process automation is one of the key aspects in the world of engineering. The great evolution suffered during the last 20 years in the engineering world, especially in the area of telecommunications and computer science, has allowed the human being to unload of monotonous and tedious tasks through the application of this feature, making even these tasks in a faster and more efficient way.

Our society, the Spanish, suffers today a serious problem regarding the acquisition of housing, especially young people, to whom the fact of being able to become independent and start our adult life in a complete way resists us. We are currently living in a period in which housing speculation is rampant, and the idea of being able to find a decent rental can be hell. In the situation in which we find ourselves, a house that comes on the market in the form of rent can last only days, sometimes even hours, due to the enormous demand that exists in comparison with the existing supply. That is why to achieve this goal, it is key to be especially fast and constant, which means spending hours in front of the computer every day, consulting the ads on the main housing portals (Idealista, Fotocasa, ...). It is there where the characteristic of automation named at the beginning, so that the user is freed from the performance of this task through the implementation of a tool which carries out the development of this activity. Can you imagine writing in a form your preferences to find a house and a server scanning the different portals periodically, sharing with you in a chat or email ads according to your parameters? That's what this memory is all about.

# Índice

## Contenido

# Índice de figuras

# Índice de tablas

# 1 Introducción

Este primer capítulo está orientado a presentar al lector de manera genérica y superficial el proyecto diseñado e implementado. Para ello, en cada uno de los siguientes subapartados, se irá desglosando un aspecto concreto del proyecto, de forma que el lector pueda ir comprendiendo de forma suave y ordenada la naturaleza de este.

## 1.1 Contexto

Partiendo de la situación comentada en la sección *Resumen* de esta memoria, este subapartado tiene como fin complementar al anterior, dando al lector la idea al completo del motivo del nacimiento de este proyecto. Bien, dicho esto, ¿de dónde nace la idea que se presenta en esta memoria? Pues la respuesta es más bien sencilla, y no es otra que de la más pura ingeniería. Surge una necesidad, la cuál es *encontrar una vivienda digna asequible sin tener que perder una cantidad de tiempo considerable*, y para ello se plantea como solución hacer uso de la automatización y la delegación de dicha tarea a través de un servidor. De esta forma, el usuario se descargaría de una gran cantidad de horas frente a la pantalla, teniendo que consultar solamente los anuncios que el servidor encontró por él según los parámetros que este mismo indicó previamente. Esto, a fin de cuentas, produciría una mayor eficacia, celeridad y aumento de calidad de vida del propio usuario.

## 1.2 Motivación

En cuanto a la motivación de la idea que estamos desarrollando, esta se ciñe única y exclusivamente a mejorar la vida de la población, minimizando los percances causados al tener que pasar por el trámite de tener que buscar un nuevo hogar. Al fin y al cabo, ese es el mayor fin de la propia ingeniería, solucionar problemas existentes dentro de nuestra sociedad, y mejorar nuestra calidad de vida.

## 1.3 Objetivos

En cuanto a objetivos del proyecto, cabe mencionar que lo que se quiere conseguir con este es obtener una herramienta que desempeñe la búsqueda de anuncios de vivienda de alquiler de forma automatizada, en base a unos parámetros dados. Para ello, se marcan los siguientes objetivos:

- **Interfaz de usuario**: como es lógico, el usuario necesitará alguna forma para interactuar con nuestro servicio. Para ello, se plantea que esta sea lo más amigable posible con este, por lo que se plantea la implementación de esta característica a través del desarrollo de un bot de Telegram. ¿Las razones de ello? Sencillo, el amplio uso de las aplicaciones de mensajería por parte de la población, la familiarización de los usuarios con estas aplicaciones y la no necesidad de instalación de aplicaciones externas (a excepción del propio Telegram) son algunas de ellas.
- **Parte servidor**: en cuanto a este aspecto, nuestra necesidad es extraer datos de portales web, ya que es la única forma de acceder a ellos. Bien, dicho esto, ¿qué herramientas existen actualmente que nos permitan extraer información de un portal web? Pues la más utilizada hoy en día es el *Web Scraping*. Esto realmente no es una herramienta informática en sí, si no un concepto que es llevado a cabo a través de lenguajes de programación tales como *Python* o *Javascript*. En el presente, *Python* es de los lenguajes de programación más avanzados en cuanto a lo que el *Web Scraping* se refiere, y dado la fuerte dependencia de nuestro proyecto en esta característica, es la herramienta que utilizaremos.
- **Persistencia de datos**: es un hecho que si tenemos que atender peticiones de diferentes clientes y además tener en cuenta las preferencias de cada uno de ellos, tendremos que tratar los datos de alguna forma. Necesitamos una tecnología que nos brinde una fuerte consistencia, que esté madura en el mercado y que se ajuste al tipo de tratamiento de datos que llevaremos a cabo, que será relacional, ya que para cada usuario tendremos unas preferencias y por tanto unos resultados distintos. Por este motivo, se ha decidido que la mejor opción es el uso de *PostgreSQL* para el tratamiento de datos.

Dicho todo lo anterior, y por tanto, ya marcados los objetivos con los que queremos cumplir y como se llevarán estos a cabo, quedarían por mencionar las características con las que nuestra herramienta debe de cumplir, las cuales son las siguientes:

1. **Eficiencia**: la actividad llevada a cabo, para ser útil, primeramente, debe de ser eficiente, si no fuese así, no tendría interés ninguno para el usuario, ya que lo que se busca es el propio desarrollo de la tarea delegada en nuestro servidor de forma hábil, consumiendo la menor cantidad de tiempo y recursos posibles. De esta manera, cumpliremos con nuestro cometido de cara al usuario.

2. **Efectividad**: aun cumpliendo con el punto anterior, esto no es suficiente. Durante el desarrollo de nuestra actividad, debemos tener en cuenta las preferencias del cliente, de manera que los datos que encontremos en la red para cada usuario tengan el mayor parecido posible con estas mismas.

3. **Disponibilidad**: aun cumpliendo con los dos puntos anteriores, esto sería insuficiente si no cumplimos este último. La disponibilidad es uno de los puntos más característicos de todo servicio, ya que si este no se cumple no podemos prestar nuestra actividad a los usuarios y por tanto nuestra utilidad pasaría a ser nula.

## 1.4 Antecedentes

Actualmente partimos de la situación en la cual existen diferentes portales de vivienda, tales como *Idealista*, *Fotocasa*, *Milanuncios.com*, y muchas más, en los cuales se publican una gran cantidad de anuncios. En ellos, existe la posibilidad de registrarse, guardar anuncios en tus favoritos, notificación de avisos de bajada de precio e incluso recibir notificaciones de nuevos anuncios publicados relacionados con tu búsqueda. En resumen, podríamos decir que a día de hoy existen funcionalidades suficientes las cuales permiten desarrollar de forma hábil la búsqueda de un hogar.

Aun así, estas herramientas existentes no son suficientes para abordar las necesidades actuales de los usuarios. No existe forma de realizar una búsqueda unificada en los principales portales de forma simultánea. Para llevar esto a cabo, sería necesario entrar en cada uno de los diferentes portales, realizar la búsqueda y pararse a analizar cada uno de los resultados para ver si son de nuestro agrado. Eso en cuanto a lo que sería realizar una simple búsqueda, si ya quisiéramos realizar algo más complejo como registrar un aviso de bajada de precio de un anuncio concreto o notificaciones de anuncios relacionados con nuestra búsqueda, nos veríamos obligados a registrarnos en el portal de turno, lo que conllevaría que un tercero tuviera acceso a datos personales nuestros para utilizarlos con fines publicitarios posiblemente (*spam*) e inundarnos la bandeja de entrada de nuestro correo de mensajes indeseados. Todo ello hace que este proceso sea aun más tedioso y monótono.

## 1.5 Descripción de la solución propuesta

Teniendo ya a estas alturas una visión general de la problemática comentada y de las actuales herramientas disponibles, ¿qué solución nos brindaría una mejora sobre lo que ya existe? Bien, pues como ya se ha mencionado anteriormente, la mejor manera de llevar a cabo esta tarea es automatizando y delegando el proceso en una máquina servidora, en base a unos parámetros los cuales sirven a la propia máquina para buscar de manera efectiva contenido de nuestro interés, permitiendo al usuario despreocuparse de manera casi total de la realización de esta tarea, teniendo únicamente que consultar los anuncios que el propio servidor encontró por él, para que este decida entre los posibles candidatos, el que sea de mayor agrado para el mismo.

Vista ya de forma genérica tanto el problema como la solución planteada, ¿qué es lo que verdaderamente se propone? Pues tras un profundo análisis de la problemática en cuestión, de las herramientas actuales y de las necesidades del usuario, se propone lo siguiente: *implementar un servicio de búsqueda automatizada mediante el uso del Web Scraping con una interfaz de usuario tan simple como un chat de Telegram a través de un bot el cual gestiona las peticiones del cliente*. El resultado de este sistema el cual llevamos desengranando de manera paulatina durante todo este capítulo se muestra a continuación en la *Ilustración 1*

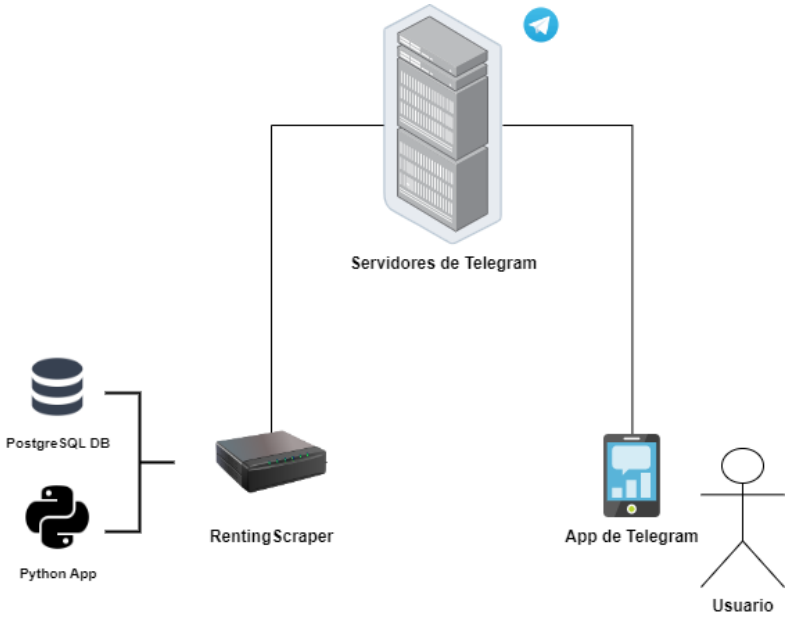



*Ilustración 1- Solución propuesta*

En cuanto al diagrama anterior, se debe comentar que el equipo situado a la izquierda llamado '*RentingScraper'* hace alusión al equipo servidor de la aplicación desarrollada. Como se puede apreciar, este aloja una base de datos *Postgresql* y una aplicación *Python* la cual contiene la lógica del servicio desarrollado.

## 1.6 Estructura del documento

Una vez expuesta de forma genérica y superficial la razón de ser de este proyecto, podemos pasar a una explicación detallada de este, para que así el lector pueda comprender de forma completa la solución propuesta. A continuación, se muestra la estructura al completo de este mismo documento:

- Capítulo 1 – Introducción: es el capítulo actual, en el cual hemos expuesto de manera breve y superficial la razón de ser de este proyecto. Este subapartado el cual se encuentra usted leyendo, es con el cual finaliza este capítulo.
- Capítulo 2 – Tecnologías empleadas: en él se detallan las herramientas de las cuales se ha hecho uso para la implementación de este proyecto.
- Capítulo 3 – Arquitectura del software: en este capítulo se comentará como está estructurado el software que ha sido desarrollado para implementar la solución propuesta.
- Capítulo 4 – Implementación: es el capítulo en el cual se desarrollará con amplio detalle cómo se ha llevado a cabo la implementación del proyecto.
- Capítulo 5 – Despliegue y puesta en marcha: en él se comentarán aspectos relacionados con el lanzamiento del servicio al usuario, una vez que el desarrollo software ha alcanzado un nivel de madurez considerable y por tanto se puede llegar a obtener una versión estable de este.
- Capítulo 6 – Conclusiones: una vez visto y comentado el contenido al completo el contenido del proyecto, es necesario tomar perspectiva para poder realizar un análisis global del mismo, analizando aspectos como sus puntos fuertes, las posibles mejoras que se pudieran implementar, el grado de utilidad de la herramienta, su posible impacto en el mercado, etc.

# 2 Tecnologías empleadas

Tras haber expuesto la solución planteada en el apartado anterior, el siguiente punto en nuestra hoja de ruta sería ir desengranando esta misma de una forma más pausada y detallada, explicando cada uno de los aspectos de interés. En este segundo capítulo, se desglosarán las distintas herramientas utilizadas en el proyecto.

## 2.1 Web Scraping

Anteriormente se ha mencionado que la principal actividad de nuestro proyecto se basa en la extracción de datos de los distintos portales de vivienda utilizados actualmente, y además comentamos que para ello, la mejor opción existente a día de hoy era a través del *Web Scraping*. Bien, una vez hemos mencionado estos detalles de nuevo, ¿qué es el *Web Scraping*? Este, se define como el hecho de extraer datos y/o elementos de una página web a través de palabras clave, como pueden ser las etiquetas *HTML* o selectores de *CSS*, pero ¿qué utilidades puede tener esto? Eso es lo que trataremos a continuación. En la *Ilustración 2*, se muestra una imagen animada la cual trata de plasmar el concepto del *Web Scraping* de una forma clara y simple. Como se puede observar, se muestra un robot el cual haciendo uso de sus brazos mecánicos extrae los elementos presentes en pantalla. En este caso, se podría decir que el robot ejerce el papel de lo que se conoce como *Bot* en la informática actual (*fuente: https://www.mozenda.com/web-scraping-workflow/*).

*Ilustración 2: Web Scraping*

### 2.1.1 Web Scraping: ventajas e inconvenientes

Es un hecho que en la actualidad las personas pasamos una gran parte de nuestro tiempo en Internet, ya sea en redes sociales, páginas Web, como en tantos otros servicios existentes, como también lo es la enorme cantidad de información existente en la red, a veces inmanejable incluso. Este detalle, es uno de los causantes de la aparición del *Web Scraping*, filtrar entre todo el contenido existente dentro de una página Web con el objetivo de obtener únicamente aquella información de la que estamos interesados. Vale, pero ¿qué ventajas [1] tiene este proceso? A continuación, se detallan algunas de ellas:

- **Mejora de la productividad**: el hecho de poder filtrar de una forma rápida entre todo el contenido existente dentro de una página web, seleccionando solamente aquello que nos interesa, nos permite conseguir una mayor productividad, ahorrando tiempo en el desarrollo de este tipo de tareas.
- **Automatización**: al ser un concepto el cual se aplica en el área de la programación, es un proceso que por tanto puede ser automatizado, permitiéndonos delegar la tarea de selección de la información en un programa informático.

- ***Testing***: este concepto, proveniente del inglés, hace referencia a la realización de pruebas sobre una aplicación web. ¿Y qué relación tiene esto con el *Web Scraping*? Pues que al poder obtener elementos *HTML* de las páginas web e incluso poder interactuar con los mismos (por ejemplo, botones, *inputs* de texto, etc), esto permite la realización de pruebas sobre estos entornos de manera programática, lo cual lo convierte en una importante herramienta para el desarrollo de este tipo de aplicaciones.

Aun así, como en cualquier aspecto de la vida, no todo son ventajas dentro de este proceso. Existen detalles los cuales hacen que la puesta en práctica de este proceso sea algo más indeseable de lo que comentábamos al principio. A continuación, se exponen algunas de estas:

- **Legalidad**: aunque la práctica en sí del *Web Scraping* no es ilegal [2], la extracción o el propio tratamiento de la información de los datos que manejan sí pueden llegar a ser un problema, por lo que para cierto tipo de prácticas puede que la práctica de este proceso no sea lo más adecuado, aunque como hemos dicho la práctica en sí del mismo no es ilegal, siempre que cumplamos con los derechos de autor, no realicemos competencia desleal, y cumplamos con la *Ley Orgánica de Protección de Datos* (LOPD).
- ***Bot detection***: como es comprensible, la inmensa mayoría de los administradores de portales web existentes no desean que su sitio web sea inundado por peticiones realizadas por ordenadores en lugar de personas, y teniendo en cuenta que el porcentaje de tráfico que circula por la red es mayoritariamente generado por *bots* [3], es un aspecto que suele estar bastante vigilado, por lo que el uso del *Web Scraping* a veces puede estar parcial o incluso totalmente limitado por los propios servidores web.

### 2.1.2 Web Scraping: tecnologías actuales

Debido a su amplia utilidad y sumado además a la tendencia de los últimos años de la automatización de tareas monótonas (como puede ser la búsqueda de información, que es el caso de este proyecto), el *Web Scraping* es un recurso que se ha ido mejorando e integrando con herramientas como pueden ser *Python* o *Javascript*, los cuales son ambos los lenguajes de programación más usados de la actualidad.

Antes de pasar a comentar las distintas tecnologías utilizadas a día de hoy para la práctica del *Web Scraping*, es necesario hacer un inciso en el cuál pongamos en situación al lector de los diferentes tipos de páginas web que existen a día de hoy, ya que en función de esta, la práctica a llevar a cabo es diferente. Bien, a continuación, se exponen los distintos tipos de páginas web existentes:

- **Páginas web estáticas**: en este tipo de páginas, el cliente recibe el contenido *HTML* de la página web al completo en la respuesta de la petición, por lo que no precisa de ninguna comunicación adicional con el servidor para obtener más información mientras permanezca en esa página.
- **Páginas web dinámicas**: a diferencia del anterior, en este tipo de páginas, el cliente recibe el contenido de la página web de forma parcial, obteniéndose el resto de forma dinámica a través de peticiones asíncronas con el servidor (estas peticiones son conocidas como peticiones *AJAX*).

Bien, dicho esto, pasamos a mostrar las diferentes herramientas existentes, centrándonos exclusivamente en los lenguajes de programación *Python* y *Javascript*, ya que ambos son los más empleados para esta tarea [4]. Tanto para *Python* como para *Javascript*, existen distintas librerías utilizadas para el *Web Scraping*. Todas ellas se pueden agrupar en 2 grupos distintos, según el tipo de página web de la cual se desean extraer los datos, los cuales son los siguientes:

- **Web Scraping para contenido estático**: en este caso, el procedimiento es similar para ambos lenguajes de programación, además de ser bastante simple. Primeramente, se procede a realizar una petición *HTML* a un servidor, este último nos devolverá el contenido en la respuesta. Para lograr esto, en *Python* existen librerías como *requests* o *requests-html*, ambas nos permiten realizar peticiones *HTML* añadiendo las cabeceras necesarias. En cuanto al caso de *Javascript*, tenemos disponibles herramientas como *fetch api* o *axios*. Una vez hemos obtenido la respuesta, tenemos que '*parsear'* (concepto utilizado en la programación, el cual se refiere al análisis de unos datos) los datos de la misma para poder proceder con el *Web Scraping*. Para ello, en *Python* disponemos de la librería *bs4*, o más conocida como *BeautifulSoup*, y para *Javascript* se dispone de *DOMParser*. Finalmente, ya podríamos proceder a la extracción de los datos del documento *HTML*, por ejemplo, en *Python*, si quisiéramos obtener todos los elementos etiquetados como párrafo en *HTML*, bastaría con hacer uso de la función *find_all*, perteneciente a la librería anteriormente mencionada, *bs4*. A continuación, en la *Ilustración 3* se expone un sencillo código de ejemplo utilizando *Python:*

```python
from bs4 import BeautifulSoup
import requests

url = 'https://stackoverflow.com/'
page = requests.get(url)
soup = BeautifulSoup(page.text, 'html.parser')
paragraphs = soup.find_all("p")

for p in paragraphs:
    print(p.text)
```

*Ilustración 3: Código de ejemplo para web estática*

- **Web Scraping para contenido dinámico**: este segundo caso es algo más complejo, debido a que precisamos del renderizado del código *Javascript* incluido en la página web. Para ello, necesitamos hacer uso de un navegador, que son los entornos donde se ejecuta este código de aplicación, pero claro, un navegador por sí solo implica muchos recursos y puede ser que esto nos suponga un problema para el desarrollo de nuestra actividad. En ese caso, para minimizar el uso de recursos, existe la posibilidad de ejecutar el navegador en modo *headless* [5], es decir, sin interfaz gráfica, aunque cabe decir también que la visita de algunos sitios web haciendo uso de este recurso puede dar problemas (carga de contenido *HTML* de forma incompleta, detección de este recurso provocando un bloqueo del mismo, etc). Bien, dicho esto, tanto en *Python* como en *Javascript* existen librerías para el desempeño del *Web Scraping* en este tipo de páginas. En el caso de *Python*, tenemos disponible la librería *Selenium*, la cual nos permite hacer uso de un navegador de forma automatizada. En cuanto a *Javascript*, se dispone de *Puppeteer*, que es una *API* de alto nivel la cual nos permite manejar *Google Chrome* de forma automatizada.

Finalizando con la sección referente al *Web Scraping*, cabe decir que varios de los conceptos mencionados en esta subsección serán explicados en mayor detalle, como pueden ser las librerías de *Python* referenciados, el uso de navegadores en modo *headless*, etc.

## 2.2 Python

El lenguaje de programación *Python* [6] es un lenguaje de alto nivel multiparadigma. Este lenguaje es interpretado por el intérprete, valga la redundancia, el cual va ejecutando cada instrucción del código de aplicación línea a línea, de manera que si existiera algún tipo de error por inconsistencias en el propio código, se produciría lo que se conoce como error en tiempo de ejecución. Esta herramienta de la que hablamos es uno de los lenguajes más utilizados actualmente por la comunidad de desarrolladores. Este último aspecto que comentamos se puede apreciar en la *Ilustración 4*, la cual muestra una gráfica (*fuente: https://spectrum.ieee.org/top-programming-languages-2022*) de los lenguajes de programación más utilizados en el año 2022:

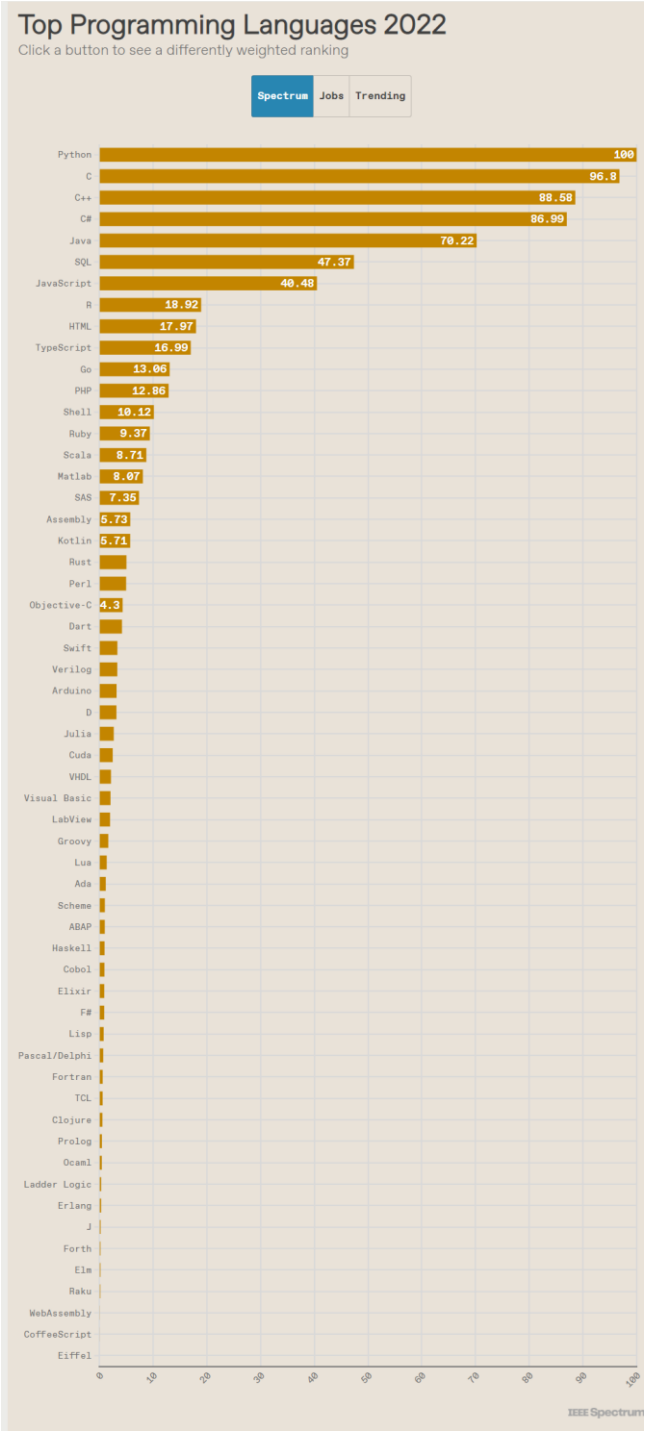



*Ilustración 4: Ranking lenguajes de programación*

Bien dicho esto, ¿qué nivel de aplicabilidad tiene *Python* para el *Web Scraping*? ¿por qué utilizar *Python* para este proyecto y no otro lenguaje? Como se ha mencionado al comienzo de este apartado, *Python* es un lenguaje multiparadigma, lo que hace que este sea una herramienta altamente versátil y pueda utilizarse para infinidad de tareas [7] (*data analysis*, *machine learning*, *programación web*, etc). Respondiendo a las preguntas anteriores, se debe decir que el motivo está ampliamente relacionado con la información mostrada en la *Ilustración 4*, y es que debido a su gran uso y a su naturaleza, existen una gran cantidad de librerías ya implementadas en *Python* para el *Web Scraping*, las cuales hacen que sea una de las opciones favoritas frente a otras alternativas como pueden ser *Java* o *Javascript*. Además, otra de las razones de su aplicación dentro de este proyecto es la compatibilidad con otras herramientas, como puede ser *Telegram* por ejemplo, otra de las herramientas que usaremos en este proyecto.

### 2.2.1 Python: Requests-html

*Requests-html* es una de las librerías de *Python* [8] la cual nos permite realizar peticiones a una URL dada, de forma que podamos obtener el contenido de una página web. Se debe decir que otra de las librerías igualmente usadas para este fin es la librería *Requests* [9], aunque ambas presentan una gran diferencia, y es que la librería *Requests* solo permite interactuar con páginas web estáticas (es decir, no es capaz de renderizar el código javascript incluido en la página), mientras que *Requests-html* puede trabajar tanto con páginas web estáticas como dinámicas, ya que esta última integra el uso del navegador *Chromium* en modo *headless* para ello, pudiendo hacer uso de él para renderizar el contenido dinámico de la página que queremos visualizar. En la *Ilustración 5* se puede observar un pequeño y sencillo fragmento de código el cual muestra el funcionamiento de esta librería:

```
1  from requests_html import HTMLSession
2
3  url = "https://stackoverflow.com/"
4  session = HTMLSession()
5  page = session.get(url)
6  page.html.render(scrolldown=256, sleep=20)
7  print (page.text)
```

*Ilustración 5: Código de ejemplo requests-html*

Como aclaración, debemos decir que en el código anterior realizaría las siguientes acciones:

1) Crear una instancia de la clase *HTMLSession*, la cual se encarga de abrir el navegador *Chromium* en modo *headless*.
2) Hacer una petición *HTTP GET* a la URL guardada en la variable *url*.
3) Ejecutar el código *Javascript* incluido en el propio contenido *HTML* de la página obtenida en la respuesta una vez hayan transcurrido 20 segundos desde la lectura de esta instrucción por parte del intérprete, además de realizar un desplazamiento hacia debajo de 256 píxeles (pudiendo cargar así contenido asociado al desplazamiento de la propia página).
4) Mostrar por pantalla el contenido *HTML* obtenido de la página en texto plano.

### 2.2.2 Python: BeautifulSoup

La librería *BeautifulSoup* es una de las herramientas más populares de *Python* [10] para el desarrollo del *Web Scraping*, ya que esta nos permite realizar el '*parseo*' (concepto anteriormente visto) del contenido de una página web obtenida en texto plano a formato *HTML*, permitiéndonos así poder proceder a la manipulación de los elementos contenidos dentro de la propia página. A continuación, en la *Ilustración 6* se muestra un breve código de ejemplo haciendo uso de esta librería:

```
1   from requests_html import HTMLSession
2   from bs4 import BeautifulSoup
3
4   url = "https://www.marca.com/"
5   session = HTMLSession()
6   page = session.get(url)
7   page.html.render()
8
9   soup = BeautifulSoup(page.text, 'html.parser')
10  links = soup.find_all('a')
11
12  for link in links:
13      print(link.get('href'))
```

*Ilustración 6: Código de ejemplo beautifulsoup*

Del código de ejemplo mostrado en la *Ilustración 6*, podemos decir lo siguiente:

1) Tras obtener la página *HTML* perteneciente a la URL mostrada en la imagen y renderizar el código de *Javascript* incluido en la propia página, se realiza el '*parseo*' del contenido en texto plano al formato *HTML* (línea 9).
2) Una vez hecho, se obtienen todos los elementos de la etiqueta *a* de *HTML* contenidos en la página.
3) Por último, se imprime por pantalla el valor del atributo *href* de cada una de las etiquetas *a* incluidas en la página.

Como se puede apreciar, la acción que se está llevando en el código anterior es un proceso sencillo de *Web Scraping*. Antes de finalizar con este apartado, es preciso mencionar que tanto la librería anteriormente vista, *Request-html*, como la librería *Selenium* (la cual veremos a continuación), ambas permiten la manipulación de elementos *HTML*, al igual que la librería *BeautifulSoup*, pero debido a que esta última es de uso específico para esta tarea, su amplia documentación y su extensión entre la comunidad de desarrolladores, se ha preferido su uso por delante de las dos anteriores.

### 2.2.3 Python: Selenium

*Selenium* es una librería de *Python* [11] la cual permite el uso automatizado del navegador a través de un *driver* (se verá más en detalle en un apartado posterior). Esta librería es una gran herramienta y se encuentra muy extendida debido a su versatilidad, además de tener una documentación bastante extensa debido a su uso. Los motivos de la popularidad de esta librería son las funcionalidades que ofrece, ya que esta nos permite llevar a cabo dos tareas muy destacadas como son el *Testing* y el *Web Scraping* (ambas comentadas en el capítulo de *Introducción* de esta memoria). Cabe mencionar que por motivos relacionados con el *Bot detecting*, han surgido variantes de esta librería como pueden ser *Selenium-stealth* [12] o *Undetected-chromedriver* [13], ambas tienen como objetivo eludir los mecanismos que los servidores web utilizan para bloquear el acceso a su sitio web a usuarios no deseados como son los propios *Bots*.

Como venimos haciendo en los apartados anteriores, para comprender de forma superficial el funcionamiento de la librería, hacemos uso de un ejemplo en sí. En la *Ilustración 7* de a continuación, se muestra un breve ejemplo del uso de esta librería:

```
1  from selenium import webdriver
2  from selenium.webdriver.chrome.service import Service
3  from fake_useragent import UserAgent
4
5  # #Browser setup
6  opt = webdriver.ChromeOptions()
7  ua = UserAgent()
8  userAgent = ua.random
9  opt.add_argument("--no-sandbox")
10 opt.add_argument("--headless")
11 opt.add_argument("--disable-gpu")
12 opt.add_argument("--start-maximized")
13 opt.add_argument(f'user-agent={userAgent}')
14 opt.add_experimental_option("excludeSwitches", ["enable-automation"])
15 opt.add_experimental_option('useAutomationExtension', False)
16 opt.add_argument("--disable-blink-features=AutomationControlled")
17 opt.add_argument("--disable-extensions")
18
19 #Launch browser app
20 driver = webdriver.Chrome(service=Service("/usr/bin/chromedriver"), options=opt)
21 url = 'https://www.fotocasa.es/es/'
22 driver.get(url)
23 scroll = range(1, 100)
24     for i in scroll:
25         n = f"{i/100}*document.body.scrollHeight"
26         driver.execute_script(f"window.scrollTo(0, {n})")
```

*Ilustración 7: Código de ejemplo selenium*

Se puede apreciar con facilidad que este ejemplo es algo más complejo que los anteriores. La razón de ello es que *Selenium* nos permite una amplia configuración del navegador (líneas 6 a la 17 en el ejemplo). En este ejemplo concretamente, podemos apreciar cómo se hace uso del navegador en modo *headless* (línea 10). El resto de las opciones de configuración serán comentadas más adelante, concretamente en el capítulo de *Implementación*.

### 2.2.4 Python: Telebot

La librería *Telebot* contiene las herramientas necesarias para poder gestionar un *Bot* de *Telegram* [14] [15] ya previamente creado, pudiendo definir las acciones que este mismo realice ante determinados *inputs* que han sido contemplados en el diseño del *Bot*. Para ello, se hace uso de los *decoradores* [16] de *Python*, los cuales nos permiten definir y/o ampliar el comportamiento de una función. En la *Ilustración 8* se muestra un pequeño fragmento de código, el cual asocia un comando del *Bot* que estamos gestionando a esa función a través del decorador *message_handler* definido en la propia librería de *Telebot*:

```
@bot.message_handler(commands=['hello'])
def say_hello(message):
    bot.reply_to(message, "¡Hola!")
```

*Ilustración 8: Código de ejemplo telebot*

Definido este comportamiento, cuando lancemos la ejecución del *Bot*, tendremos que este responderá al cliente con el mensaje "*¡Hola!*" ante el envío del comando *hello*. De esta manera, la librería principalmente nos permite asociar funciones a eventos, lo que se conoce como *Application Programming Interface* (API) en el campo de la programación.

Bien, hemos introducido de forma breve la librería de *Telebot*, pero no se ha comentado hasta ahora como se lleva a cabo la creación de un *Bot* en sí [17]. Para ello, se hace uso de otro *Bot* de *Telegram*, el cual se llama *BotFather*. Este nos permite gestionar nuestros *Bots*, de forma que podemos crear uno nuevo, eliminar uno anterior y modificar parámetros de uno ya existente (como la descripción del *Bot*, su nombre, su foto de perfil, etc). En la *Ilustración 9* se muestra un fragmento de la conversación con *BotFather* para la creación del *Bot* que ha sido utilizado en este proyecto:

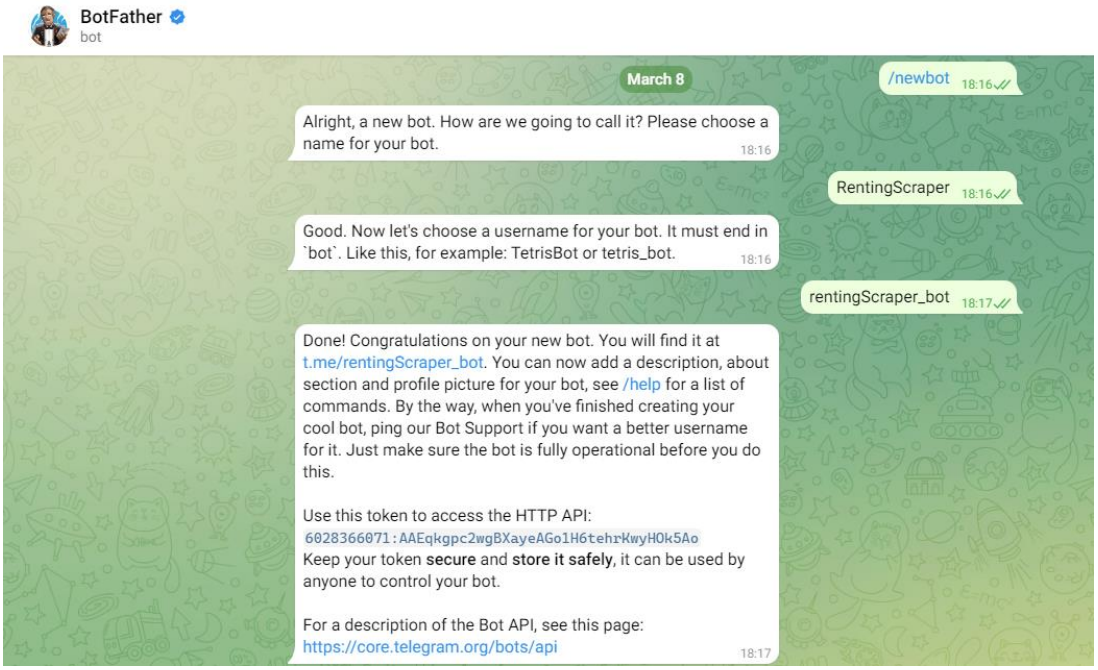



*Ilustración 9: Creación Bot de Telegram*

En la captura se puede apreciar cómo le indicamos al *Bot* a través del comando *newbot* (en *Telegram* los comandos se inician con el carácter '/') que queremos crear un nuevo *Bot*. Este nos responde pidiéndonos el nombre y el *username* que queremos que tenga nuestro *Bot*. Una vez estos datos han sido facilitados, este nos indica que la operación se ha llevado a cabo con éxito y nos devuelve un *token*, que no es más que un código identificativo de nuestro *Bot*, con el cuál podremos acceder a él programáticamente a través del uso de la librería *Telebot*.

## 2.3 Google Chrome y Chromedriver

Una vez vistos todos los principales recursos de *Python* que han sido utilizados en el proyecto (en especial la librería *Selenium*), podemos pasar a detallar otra de las herramientas utilizadas en el proyecto, como lo es *Google Chrome* y a su vez *Chromedriver*. *Google Chrome* es uno de los navegadores web más extendidos en la actualidad. Este cuenta con una gran cantidad de extensiones que permiten mejorar su funcionamiento, las cuales pueden ser encontradas en el propio *market* del navegador. Otra de las características de este navegador (como no puede ser de otra manera debido a su gran extensión) es su integración con otras herramientas, como por ejemplo *Python*, a través del uso de la librería *Selenium*.

Aunque existen otros navegadores webs ampliamente extendidos en el mercado que también son compatibles con *Selenium*, como es el caso de *Firefox*, este no permite el mismo nivel de configurabilidad que *Chrome*. Este es el principal motivo por el cual se ha decidido el uso de *Google Chrome* en lugar de *Firefox*.

Bien, se ha explicado qué es *Google Chrome* y el motivo de su uso frente a otras alternativas, pero ¿qué es *Chromedriver*? Esta no es más que una herramienta *open-source* [18] en la que se basa *Selenium* para la automatización del navegador *Google Chrome*. Sin ella no sería posible hacer uso del navegador de forma programática. En cuanto a Firefox, existe su análogo llamado *Geckodriver* [19].

## 2.4 Postgresql

La persistencia de datos es uno de los puntos más comunes en cualquier proyecto en el que se realice tratamiento de datos, y este proyecto no iba a ser diferente debido a su naturaleza, ya que tendremos que registrar las diferentes subscripciones de cada uno de los usuarios, y los resultados obtenidos de cada uno de ellos. Por tanto, se ha tomado al servidor *Postgresql* como el ideal para este proyecto, por su amplia documentación y su asentamiento en el mercado.

*Postresql* es un servidor de base de datos [20] *open-source* de tipo relacional el cual hace uso del lenguaje de consultas *SQL*. Dentro de sus características más destacadas [21] se encuentran la compatibilidad con el protocolo de seguridad *SSL*, el protocolo de red *IPv6* y el control de acceso (a través de contraseñas, certificados, etc). En cuanto a su integración con otras herramientas como *Python*, existe una librería específica para ello llamada *Psycopg2* [22], la cual ha sido utilizada también en este proyecto.

## 2.5 Github

Uno de los aspectos más importantes dentro del proceso *software*, es el control de versiones. La puesta en práctica de este tipo de herramientas es fundamental durante toda la etapa de desarrollo del sistema, ya que esta nos permite trabajar de forma más eficiente, disponer de trazabilidad del sistema, corregir *bugs* que puedan producirse, y un largo etc.

*Github* es una de las plataformas de control de versiones de software más extendidas en la actualidad. Esta nos permite disponer de un repositorio remoto (o los que se necesiten) en el cual podemos almacenar nuestra aplicación de forma segura, llevando un control de la misma a través de las distintas herramientas que *Git* nos ofrece (creación de ramas, *commits*, *logs*, etc). En la *Ilustración 10* se muestra una captura del repositorio remoto creado en *Github* para el seguimiento de este proyecto:

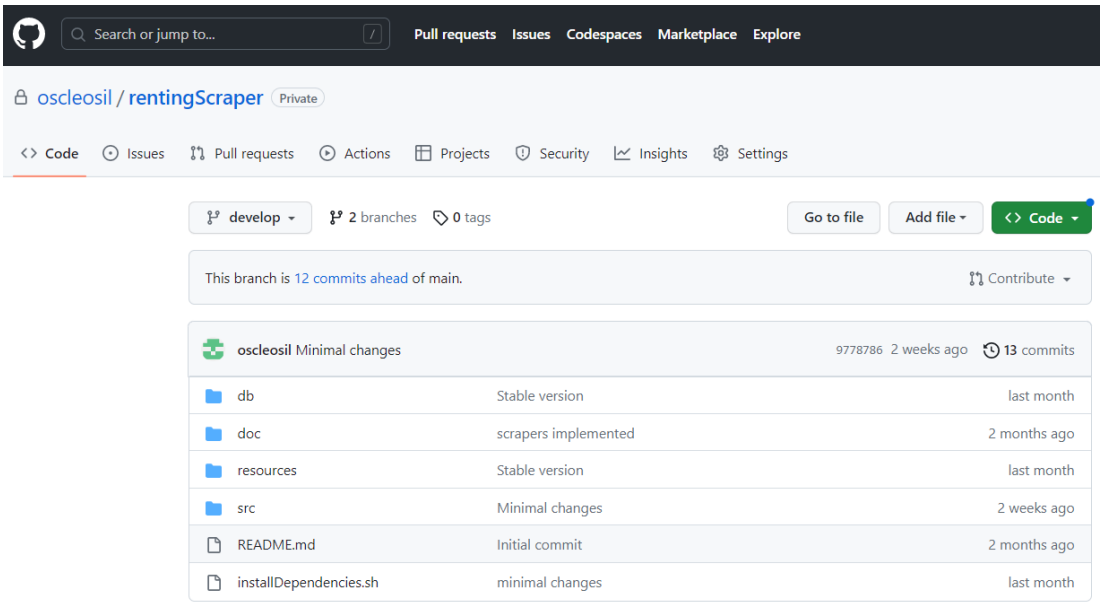



*Ilustración 10: Repositorio del proyecto*

En la captura anterior se puede apreciar ligeramente la estructura de directorios seguida para el *software*. Esta será comentada más adelante, concretamente en el capítulo *Arquitectura del software*.

# 3 Arquitectura del software

En este tercer capítulo se pretende plasmar al lector la estructura seguida para la implementación del servicio al que hemos denotado como '*RentingScraper*'. Para ello, nos hemos basado en una arquitectura clásica *Cliente-Servidor*, delegando la parte cliente en los servidores de *Telegram* mediante el uso de un *Bot* gestionado a través de *Python*. En la *Ilustración 11* se muestra un simple esquema de la arquitectura *Cliente-Servidor*, en la cual solo interactúan estos dos únicos equipos en cuanto a lo que a nivel de aplicación se refiere:


*Ilustración 11: Arquitectura clásica cliente-servidor*

Continuando con la explicación de nuestra arquitectura en concreto, tenemos un pequeño detalle diferenciador con respecto al esquema anterior, y es que al hacer uso de los servidores de *Telegram* como se ha comentado previamente, estos actúan como *proxy*, encaminando los mensajes del cliente hacia nuestro servidor, donde finalmente serán procesados de forma completa. Teniendo esto en cuenta, y continuando con el esquema de la *Ilustración 1*, dando un poco más de detalle a este último, nos quedaría una arquitectura como la que se muestra en la *Ilustración 12*:

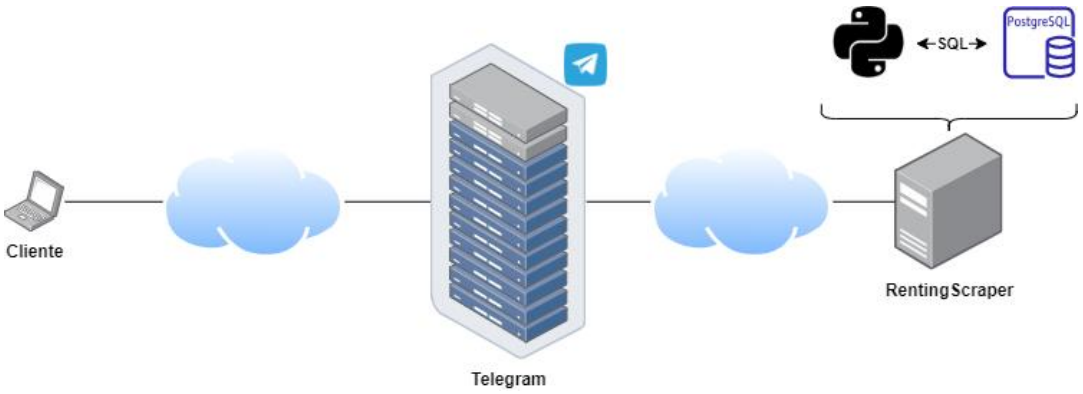



*Ilustración 12: Arquitectura implementada*

Como podemos apreciar en el diagrama anterior, siempre que el cliente quiera realizar algún tipo de transacción con nuestro servidor tendrá que acceder previamente por *Telegram*. La desventaja de ello es que si por casual los servidores de esta aplicación sufrieran algún tipo de incidencia la cual imposibilitara su uso o estos se encontrasen inaccesibles debido a un problema de encaminamiento en la red, nuestra aplicación dejaría de prestar servicio independientemente de su correcto funcionamiento. Este aspecto que comentamos evidentemente es muy negativo, ya que nosotros no tenemos control sobre servidores externos, pero es el coste que tenemos que asumir al querer delegar la parte del cliente en un servidor ajeno al nuestro, en lugar de implementarlo nosotros mismos. Esto nos permitiría conseguir un ahorro temporal considerable, el cual podríamos invertir en un mejor desarrollo de la parte servidor.

En cuanto a lo que al servidor se refiere, este contendría un sistema operativo *Ubuntu 22.04*, el cual contaría con los siguientes recursos:

- Python 3.8.10 o superior: para poder ejecutar la aplicación *Python* desarrollada, incluyendo además las librerías comentadas en la sección 2 del capítulo *Tecnologías empleadas* (además de otras librerías de menor relevancia, pero también necesarias, como por ejemplo la librería *regex*).
- PostgreSQL 12 o superior: para almacenar los datos relevantes para el correcto funcionamiento de nuestro servicio (como por ejemplo las suscripciones vigentes en el sistema).
- Google Chrome 112: para poder acceder a los distintos portales web para extraer la información requerida.
- Chromedriver 112: para poder controlar el navegador *Google Chrome* programáticamente.

Además, cabe añadir que el servicio desarrollado, será gestionado por la herramienta *Systemd* [22] (incluida en cualquier distribución de *Linux* actualmente). Esta no es más que una herramienta de gestión del sistema, la cual nos permite manejar recursos del mismo, en especial los servicios, de forma que podamos llevar un control de ellos a través del uso de ficheros de registro, también conocidos como *logs*.

# 4 Implementación del servicio

En este cuarto capítulo se describirá con un alto nivel de detalle la implementación del servicio *RentingScraper*, de forma que el lector pueda comprender de forma completa cómo está estructurada la aplicación, las distintas funcionalidades que esta realiza, los problemas encontrados durante este proceso y otros aspectos de interés.

## 4.1 Estructura de directorios

Antes de proceder con la explicación de la implementación del servicio, es conveniente que el lector conozca la estructura de directorios utilizada en el proyecto, de forma que este tenga una visión general de como se ha desarrollado el mismo antes de entrar en detalles. En la *Ilustración 13* se muestra un esquema de la estructura de directorios utilizada en el proyecto:

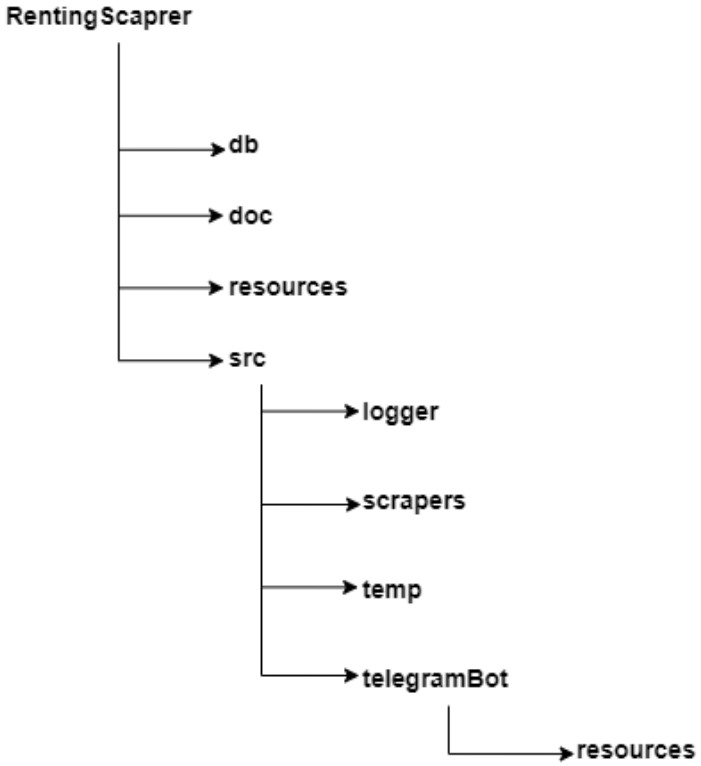



*Ilustración 13: Estructura de directorios*

Como se puede observar, dentro del directorio raíz tenemos 4 directorios distintos, los cuales cada uno de ellos albergan el siguiente contenido:

- **db**: este directorio contiene los ficheros *sql* y *scripts* de *bash* necesarios para la puesta en marcha y configuración del servidor de base de datos *PostgreSQL*.
- **doc**: en él se recoge la documentación del proyecto, es decir, este documento.
- **resources**: contiene recursos software necesarios para la implementación del proyecto, como pueden ser *Google Chrome* y su controlador *Chromedriver*.

- **src**: es el directorio más relevante de todos, ya que en él se recoge el código fuente de nuestra aplicación, el cual se encuentra organizado de la siguiente manera:

    - **logger**: recoge el código fuente relacionado con el *logger* del servicio, de forma que el propio servicio pueda interactuar con la herramienta del sistema *syslogd*.
    - **scrapers**: contiene el código fuente necesario para la extracción de datos de los distintos portales web utilizados por la aplicación (los cuales son *Fotocasa*, *Milanuncios*, *Pisos y Yaencontre*).
    - **temp**: esta carpeta es utilizada por la aplicación para almacenar ficheros temporales durante la ejecución del propio servicio.
    - **telegramBot**: contiene las herramientas necesarias para la gestión del *Bot* de *Telegram*. En él podemos encontrar la carpeta '*resources*', la cual contiene ficheros de extensión '*Markdown*', con distintos mensajes utilizados por el *Bot* a la hora de interactuar con el usuario.

Finalmente, una vez ha sido presentada la estructura del proyecto, podemos pasar a comentar a fondo el contenido del mismo, de modo que el lector pueda conocer en profundidad los detalles de este.

## 4.2 Base de datos

Como se ha mencionado en la introducción de este capítulo, los ficheros relacionados con la base de datos se encuentran dentro del directorio *db*. En él, podemos encontrar los siguientes:

- **subscriptions.sql**: en este fichero se encuentran las sentencias *SQL* necesarias para crear la tabla *subscriptions* dentro de la base de datos *rentingscraper* recogida en el servidor *PostgreSQL*. En la *Ilustración 14* se puede observar el contenido de dicho fichero:

```sql
CREATE TABLE IF NOT EXISTS subscriptions (
    subscriptionId VARCHAR(13),
    userId VARCHAR(30) NOT NULL,
    chatId VARCHAR(30) NOT NULL,
    city VARCHAR(20) NOT NULL,
    amplitude VARCHAR(15) NOT NULL,
    houseType VARCHAR(15) NOT NULL,
    furnished VARCHAR(3),
    maxPrice INT,
    minPrice INT,
    minBath INT,
    minRooms INT,
    maxSurface INT,
    minSurface INT,
    lastUpdate TIMESTAMP,
    PRIMARY KEY (subscriptionId, userId)
);
```

*Ilustración 14: Fichero subscriptions.sql*

Como se puede observar en la ilustración, para cada una de las subscripciones recogeremos datos identificativos, como son el *subscriptionId*, el *userId* y el *chatId*, siendo el primero un *hash* aleatorio de 32 *bits* generado durante el procesado de la subscripción, y los dos últimos, valores proporcionados por el propio *Telegram* para la identificación del usuario. En cuanto al resto de parámetros, todos hacen referencia a las preferencias de búsqueda del propio usuario, exceptuando al parámetro *lastUpdate*, el cual recoge la fecha y hora de la última actualización de la subscripción, de modo que el servidor pueda revisar las subscripciones que necesitan ser actualizadas en función del valor de este campo. Actualmente, el servicio se encuentra configurado para que actualice cada subscripción recogida en el sistema cada 4 horas. Este parámetro no es configurable por el usuario a día de hoy, aunque podría ser configurado en el futuro. En cuanto a las claves primarias de la tabla, tenemos que estas son el *subscriptionId* y el *userId*, permitiendo así que cada usuario pueda tener más de una subscripción registrada en el sistema.

- **searchResults.sql**: este otro fichero contiene las sentencias *SQL* para la creación de la tabla *searchresults*, la cual almacenará los resultados encontrados para cada uno de los usuarios. Esta tabla tiene dos funciones principales dentro del servicio, las cuales son las siguientes: 1) Recoger los resultados encontrados para cada usuario, para devolverle más tarde los que el necesite. 2) Verificar si un resultado obtenido ya había sido encontrado anteriormente para ese usuario, de manera que no se tenga en cuenta si esto ocurre (es decir, no se registra en la tabla, y por tanto no se tiene en cuenta como un nuevo resultado para ese usuario, de forma que este solo obtenga las novedades desde el momento de su subscripción y no resultados duplicados). En la *Ilustración 15* mostrada a continuación se muestra el contenido de dicho fichero:

```sql
CREATE TABLE IF NOT EXISTS searchResults (
    subscriptionId VARCHAR(13),
    userId VARCHAR(30),
    website VARCHAR(20) NOT NULL,
    link VARCHAR(200) NOT NULL,
    title VARCHAR(200) NOT NULL,
    price INT NOT NULL,
    descript VARCHAR(5000),
    telephone VARCHAR(16),
    surface INT,
    rooms INT,
    bathrooms INT,
    floor INT,
    furnished BOOLEAN,
    elevator BOOLEAN,
    swPool BOOLEAN,
    garage BOOLEAN,
    terrace BOOLEAN,
    airConditioning BOOLEAN,
    heating BOOLEAN,
    score FLOAT(5),
    searchDate TIMESTAMP,
    PRIMARY KEY (subscriptionId, link),
    FOREIGN KEY (subscriptionId, userId) REFERENCES subscriptions(subscriptionId, userId) ON DELETE CASCADE
);
```

*Ilustración 15: Fichero searchResults.sql*

Como podemos observar, las claves de dicha tabla son los campos *subscriptionId* y *link*, ya que para que exista un resultado es necesario que exista previamente una subscripción (de ahí la clave externa), y además se debe tener en cuenta que un anuncio (que viene dado por un *link* o *url*) puede obtenerse como resultado para más de un usuario. El resto de los campos, son datos extraídos del portal web al que pertenece el anuncio mediante la técnica del *Web Scraping*, exceptuando los dos últimos, siendo el atributo *score* un valor numérico calculado de forma interna (se explicará más adelante) en función de las características del anuncio de forma que se pueda catalogar cada uno de ellos, y el campo *searchDate*, el cual permite conocer el instante en el que fue obtenido dicho resultado (para que se pueda discriminar si es un resultado reciente o no).

- **setUpDataBase.sh**: este último fichero perteneciente al directorio *db*, contiene las sentencias *bash* necesarias para la puesta en marcha y configuración de la base de datos utilizada en el proyecto. En la *Ilustración 16* se muestra su contenido:

```bash
#!/bin/bash

echo "[ INFO ]: Setting up rentingScraper DB"

if echo ${PWD} | grep "rentingScraper/db" -q
then
    echo "[ INFO ]: Dependencies are going to be installed"
else
    echo "[ ERROR ]: This script must be called from the same directory where it is stored"
    exit 1
fi

echo "[ INFO ]: DB is going to be configured"
sudo systemctl restart postgresql
psql -U dit -d dit -c 'create database rentingscraper;'
psql -U dit -d rentingscraper -a -f subscriptions.sql
psql -U dit -d rentingscraper -a -f searchResults.sql

```

*Ilustración 16: Fichero setUpDataBase.sh*

Este *script* que se muestra en la *Ilustración* 16 verifica si el directorio de trabajo coincide con el directorio del ejecutable, en cuyo caso procede a la puesta en marcha y configuración necesaria de *PostgreSQL*, creando la base de datos para la aplicación y las tablas necesarias para ello.

## 4.3 Aplicación Python: *RentingScraper*

Centrándonos ya en el código de aplicación, este se encuentra distribuido en cuatro subdirectorios distintos dentro del directorio *src*, los cuales han sido comentados anteriormente. Estos serán explicados detalladamente a continuación en un subapartado diferente para cada uno de ellos, pudiéndonos centrar de forma exclusiva en cada uno de ellos.

### 4.3.1 Aplicación Python: *scrapers*

Este directorio contiene el corazón del código fuente de nuestro servicio, ya que en él se recogen las instrucciones necesarias para la extracción de cada uno de los portales de vivienda. En el podemos encontrar los siguientes ficheros:

- **common_resrc.py**: este fichero contiene recursos utilizados por cualquiera de los *scrapers* del sistema. A continuación, se muestra el código de aplicación de cada una de las funciones de este módulo de *Python*, acompañadas de sus respectivos comentarios:

```
15
16  def deleteAccentMarks(chain):
17      toChange = ["á", "é", "í", "ó", "ú"]
18      replaceWith = ["a", "e", "i", "o", "u"]
19      i=0
20
21      for letter in toChange:
22          if(letter in chain):
23              chain = chain.replace(letter, replaceWith[i])
24          i+=1
25
26      return chain
27
```

*Ilustración 17: Función deleteAccentMarks*

La función *deleteAccentMarks*, mostrada en la *Ilustración 17*, se encarga de eliminar caracteres acentuados dentro de una cadena, de modo que esta prevenga errores que pudieran ser provocados por estos mismos.

```
27
28  def calculateScore(values):
29      score = (1/values['price'])*1000
30      if(values['surface']!='NULL'):
31          score += (float(values['surface']))/100
32      if(values['rooms']!='NULL'):
33          score += float(values['rooms'])*0.5
34      if(values['baths']!='NULL'):
35          score += float(values['baths'])*0.5
36      if(values['furnished']!='NULL'):
37          score += 1.0
38      if(values['elevator']!='NULL'):
39          score += 1.0
40      if(values['swPool']!='NULL'):
41          score += 1.0
42      if(values['garage']!='NULL'):
43          score += 1.0
44      if(values['terrace']!='NULL'):
45          score += 1.0
46      if(values['airConditioning']!='NULL'):
47          score += 1.0
48      if(values['heating']!='NULL'):
49          score += 1.0
50
51      return score
52
```

*Ilustración 18: Función calculateScore*

La función mostrada en la *Ilustración 18* es la encargada de puntuar un anuncio en función de sus características. Este proceso se realiza tomando los datos extraídos de cada uno de los anuncios y ajustándolos mediante unos coeficientes. También se tiene en cuenta el tipo de proporcionalidad que se debe usar para cada parámetro de cara al cálculo de la puntuación. Por ejemplo, el precio de la vivienda puntúa a través de una proporcionalidad inversa, de manera que si el importe de la renta de la vivienda es mayor, la puntuación dada a ese anuncio será menor, a diferencia del dato de la superficie, por ejemplo, que puntuará de forma positiva en caso de que la vivienda cuente con un mayor número de metros cuadrados.

```
52
53  def alreadyInDB(subscriptionId, userId, link):
54      saved = False
55      conn = psycopg2.connect(host="localhost", database="rentingscraper", user="dit", password="dit")
56      cursor = conn.cursor()
57      sql = f"select 1 from searchresults where subscriptionId=\'{subscriptionId}\' and userId=\'{userId}\' and link=\'{link}\'"
58      cursor.execute(sql)
59      res = cursor.fetchall()
60      if(len(res)!=0):
61          saved = True
62      cursor.close()
63      conn.close()
64
65      return saved
66
```

*Ilustración 19: Función alreadyInDB*

La función *alreadyInDB*, mostrada en la *Ilustración 19* tiene como cometido verificar si un resultado obtenido ha sido encontrado previamente para un usuario y una subscripción concreta. De esta manera podremos conocer si ese resultado había sido encontrado anteriormente, evitando así generar duplicidades.

```
67  def saveSearchResult(values, searchDate):
68      conn = psycopg2.connect(host="localhost", database="rentingscraper", user="dit", password="dit")
69      cursor = conn.cursor()
70      score = calculateScore(values)
71      sql = 'insert into searchResults(subscriptionId, userId, website, link, title, price, descript, telephone, surface, rooms,'
72      sql += ' bathrooms, floor, furnished, elevator, swPool, garage, terrace, airConditioning, heating,score,searchDate) '
73      sql += f"values(\'{values['subscriptionId']}\',\'{values['userId']}\',\'{values['website']}\',\'{values['link']}\',\'{values['title']}\',"
74      sql += f"{values['price']},\'{values['descript']}\',\'{values['telephone']}\',{values['surface']},{values['rooms']},"
75      sql += f"{values['baths']},{values['floor']},{values['furnished']},{values['elevator']},{values['swPool']},{values['garage']},{values['terrace']},"
76      sql += f"{values['airConditioning']},{values['heating']}, {score}, \'{searchDate}\')"
77      cursor.execute(sql)
78      conn.commit()
79      cursor.close()
80      conn.close()
```

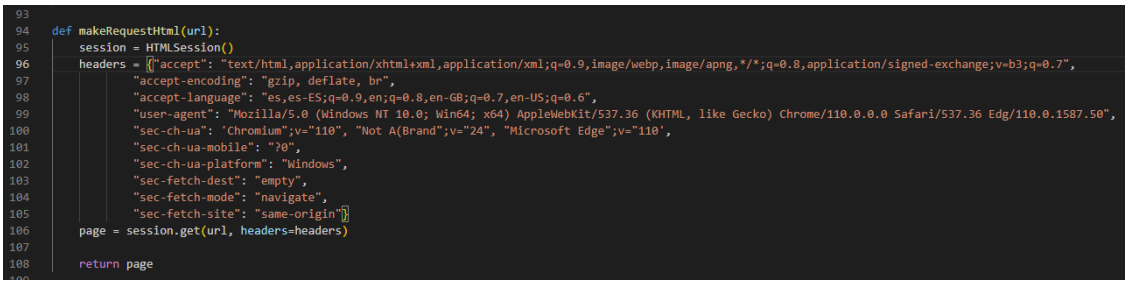
*Ilustración 20: Función saveSearchResult*

La función mostrada en la *Ilustración 20* se encarga de persistir los datos de un resultado previamente obtenido.

```
82  def filtersAdded(paramValues):
83      containFilters = False
84
85      if(paramValues['amueblado']=='si' or paramValues['precio minimo'] > 0
86          or paramValues['precio maximo'] > 0 or paramValues['superficie minima'] > 0
87              or paramValues['superficie maxima'] > 0 or paramValues['minimo de baños'] > 0
88                  or paramValues['minimo de habitaciones'] > 0):
89
90                      containFilters=True
91
92      return containFilters
```

*Ilustración 21: Función filtersAdded*

La función mostrada en la *Ilustración 21* comprueba si los parámetros dados por el usuario contienen algún tipo de filtro.



```
def makeRequestHtml(url):
    session = HTMLSession()
    headers = {"accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7",
            "accept-encoding": "gzip, deflate, br",
            "accept-language": "es,es-ES;q=0.9,en;q=0.8,en-GB;q=0.7,en-US;q=0.6",
            "user-agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/110.0.0.0 Safari/537.36 Edg/110.0.1587.50",
            "sec-ch-ua": 'Chromium";v="110", "Not A(Brand";v="24", "Microsoft Edge";v="110',
            "sec-ch-ua-mobile": "?0",
            "sec-ch-ua-platform": "Windows",
            "sec-fetch-dest": "empty",
            "sec-fetch-mode": "navigate",
            "sec-fetch-site": "same-origin"}
    page = session.get(url, headers=headers)

    return page
```

*Ilustración 22: Función makeRequestHtml*

La función *makeRequestHtml*, mostrada en la *Ilustración 22* realiza una petición *HTTP GET* a una página web estática, como pueden ser los portales web *yaencontre.com* o *pisos.com*. Esta devolverá la respuesta obtenida al completo una vez haya sido obtenida.

```
def setUpAndRunBrowser():
    # Browser setup
    opt = webdriver.ChromeOptions()
    ua = UserAgent()
    userAgent = ua.random
    opt.add_argument("--no-sandbox")
    opt.add_argument("--headless")
    opt.add_argument("--disable-gpu")
    opt.add_argument("--start-maximized")
    opt.add_argument(f'user-agent={userAgent}')
    opt.add_experimental_option("excludeSwitches", ["enable-automation"])
    opt.add_experimental_option('useAutomationExtension', False)
    opt.add_argument("--disable-blink-features=AutomationControlled")
    opt.add_argument("--disable-extensions")

    #Launch browser app
    driver = webdriver.Chrome(service=Service("/usr/bin/chromedriver"), options=opt)

    return driver
```

*Ilustración 23: Función setUpAndRunBrowser*

La función mostrada en la *Ilustración 23* tiene como objetivo configurar y lanzar el navegador *Google Chrome* de manera adecuada para posteriormente proceder con el *Web Scraping* de una página web dinámica. Dentro de las distintas opciones de configuración utilizadas, nos encontramos por ejemplo con *headless* y *disable-gpu*, las cuales nos permiten arrancar el navegador sin interfaz gráfica. En cuanto al resto de las opciones, estas tienen como objetivo configurar el navegador de manera que pase lo más desapercibido posible ante los mecanismos de *bot detecting* de los portales web visitados. La razón de esto es que la librería *Selenium* utiliza ciertas propiedades del navegador las cuales son fácilmente detectables. Además, cabe añadir en cuanto a este aspecto, que el ejecutable *chromedriver* es una versión manipulada manualmente, ya que esta contiene ciertas variables de *Javascript* predefinidas las cuales pueden provocar que el servidor al que accedamos detecte el uso de *Selenium* y termine bloqueándonos el acceso. En la sección de *Bibliografía* se adjuntará más información sobre este tema, para que el lector pueda conocer más sobre este aspecto si así lo desea.

- **fotocasa.py**: este fichero de *Python* contiene las funciones necesarias para la extracción de los datos del portal web *Fotocasa*. En él podemos encontrar dos funciones, las cuales serán comentadas a continuación, sin adjuntar capturas de ellas debido a su extensión:

  - *createFotocasaURL*: se encarga de generar la URL apropiada en función de las preferencias dadas por el usuario para posteriormente proceder a la búsqueda de los anuncios de interés.
  - *getDataFromFotocasa*: una vez ha obtenido la URL adecuada haciendo uso de la función anterior, llama a la función *setUpAndRunBrowser*, accede a la URL obtenida a través del navegador y obtiene el contenido HTML completo de la página deslizando de forma continuada hasta el final de la página. Una vez se ha cargado el contenido HTML de forma completa, este se guarda en memoria, se analiza haciendo uso de *BeautifulSoup* y posteriormente se filtra, quedándonos solamente con los anuncios mostrados en la propia página. Por último, se procede a la extracción de los datos de cada uno de los anuncios, y en caso de que este no exista en la base de datos, se almacena.

- **milanuncios.py**: este fichero es similar al anterior, contiene una función para la generación de la URL deseada, llamada *createMilanunciosURL*, y otra función para la extracción de los datos, llamada *getDataFromMilanuncios*. El procedimiento para ambas funciones es muy similar a los utilizados en el fichero *fotocasa.py*, pero con una pequeña peculiaridad en cuanto a la extracción de los datos. La cuestión es que el servidor web *Milanuncios*, tras varios accesos haciendo uso del navegador automatizado mediante *Selenium* terminaba bloqueándonos el acceso, y esta al ser una web dinámica no podía ser accedida a través de otros métodos, por lo que se optó por la siguiente solución:

  1. Realizar una petición *HTTP GET* al servidor con la petición obtenida de la función *createMilanunciosURL* haciendo uso de la función makeRequestHtml, obteniendo así el HTML inicial con el código *Javascript* incluido.
  2. Guardar el contenido HTML de la respuesta localmente en el directorio *temp*, el cual fue mencionado al comienzo de este capítulo.
  3. Abrir el archivo HTML almacenado en el servidor con el navegador en modo headless haciendo uso de *Selenium* (con la función *setUpAndRunBrowser*).
  4. Cargar el contenido dinámico de la página con ayuda del navegador, realizando un desplazamiento vertical hacia abajo.
  5. Proceder a la extracción de los datos y persistir los que sean necesarios.

- **pisos.py**: este fichero contiene las funciones necesarias para la extracción de los datos de los anuncios encontrados en el portal *pisos.com*. En él, podemos encontrar las funciones *createPisosURL*, la cual tiene exactamente el mismo objetivo que *createFotocasaURL*, pero teniendo en cuenta las peculiaridades de este servidor web, y por último, la función *getDataFromPisos*, la cual es muy similar a la función *getDataFromFotocasa* en lo que al proceso se refiere, con la peculiaridad de que el portal web *Pisos.com* es de contenido estático, por lo que para obtener el HTML se hace uso de la función *makeRequestHtml*.
- **yaencontre.py**: este último fichero del directorio *scrapers*, contiene los recursos necesarios para la extracción de los datos de los anuncios encontrados en el portal *yaencontre.com*. En él, podemos encontrar dos funciones, las cuales son *createYaencontreURL*, que se encarga de generar la URL apropiada en función de las preferencias del usuario, y *getDataFromYaencontre*, que se encarga de extraer los datos de los anuncios encontrados en dicho portal web siguiendo exactamente el mismo proceso que en la función *getDataFromPisos* (ya que ambos portales usan solamente contenido estático).

### 4.3.2 Aplicación Python: *telegramBot*

En este apartado se detallará el contenido del directorio *telegramBot*, el cual contiene el código fuente encargado de gestionar el *bot* de nuestro servicio *RentingScraper*. A continuación, se explica cada uno de los distintos ficheros contenidos en dicho directorio:

- **resources**: este subdirectorio contiene mensajes predefinidos por el desarrollador encapsulados en ficheros de extensión *markdown*, los cuales son utilizados por el *bot* para responder a diferentes peticiones del usuario. En la *Ilustración 24* se muestra como ejemplo el contenido del fichero *help.md*, utilizado por el *bot* cuando el usuario solicita ayuda:

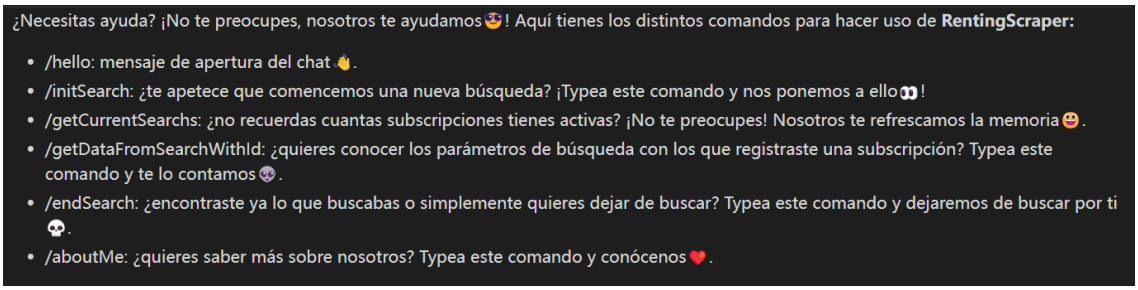



*Ilustración 24: Fichero help.md*

- **rentingScraperBot.py**: contiene el conjunto de funciones utilizadas para gestionar las peticiones del usuario, las cuales son las siguientes:

  - *send_welcome_message*: esta función es utilizada por el *bot* cuando este recibe los comandos *hello* o *start* por parte del cliente. En ella se hace uso del fichero *welcome.md* contenido en el directorio *resources* para responder al usuario. En la *Ilustración 25* se muestra el contenido de dicha función:

```python
@bot.message_handler(commands=['hello', 'start'])
def send_welcome_message(message):
    LOGGER.debug(f"Un nuevo cliente con userId '{message.from_user.id}' ha establecido conexión con el servidor")
    f = open("telegramBot/resources/welcome.md", "r")
    response = f.read()
    bot.reply_to(message, response, parse_mode="Markdown")
```

*Ilustración 25: Función send_welcome_message*

Como se puede observar, la función se asocia a los comandos indicados anteriormente mediante el uso del decorador *message_handler*.

  - *send_help_message*: esta función es muy similar a la anterior, difiriendo solamente en el comando al que se encuentra asociada y el contenido del mensaje de respuesta al usuario. En este caso esta función se encuentra asociada al comando *help*, y ante este responderá con el contenido del fichero *help.md*, recogido en la carpeta de *resources*.
  - *send_aboutMe_message*: también muy similar a las dos anteriores, esta se ejecuta ante la recepción del comando *aboutMe*, y responde al usuario con el contenido del fichero *aboutMe.md*, el cual podemos observar a continuación en la *Ilustración 26*:

**RentingScraper** es una aplicación diseñada para la *búsqueda automatizada de anuncios de viviendas de alquiler*, con la idea de prestar asistencia al usuario sin que este deba pasarse horas frente al ordenador consultando los distintos portales utilizados a dia de hoy. RentingScraper **realiza consultas en los principales portales Web de viviendas utilizados a día de hoy según tus preferencias**, y al obtener los datos comparte los anuncios obtenidos con el usuario a través del propio chat de Telegram. Tras realizar un primer sondeo y mandar los resultados al usuario, **RentingScraper realizará búsquedas de manera periódica y compartirá los resultados con el usuario**, para que así este pueda estar al día de las últimas publicaciones.

*Ilustración 26: Fichero aboutMe.md*

  - *initSearch*: esta función, la cual se encuentra asociada al comando *initSearch*, es la utilizada para registrar una nueva subscripción de búsqueda en el servicio. Para ello, primeramente, el *bot* responde ante este comando con tres mensajes distintos, los cuales son meramente informativos, con el fin de dar a conocer al usuario el método de subscripción del usuario. Más tarde, una vez el usuario ha comprendido el método de registro (el cual se basa en cumplimentar un formulario con unas determinadas características), y ha completado el registro de forma adecuada, la subscripción es gestionada por el propio servicio, registrándola en la base de datos, y realizando una primera búsqueda, y devolviendo posteriormente al usuario los distintos resultados encontrados.
  - *getCurrentSearchs*: esta función es ejecutada por el servicio ante la recepción del comando *getCurrentSearchs*. Su finalidad no es otra que devolver al usuario las diferentes subscripciones que tiene actualmente activas. Para ello, se realiza una consulta a la base de datos, obteniendo los distintos *subscriptionId* (campo mencionado anteriormente en el apartado *4.2*) que existen para ese usuario, y devolviéndoselos a este último.
  - *getDataFromSubWithId*: asociada al comando *getDataFromSearchWithId*, esta función devuelve los parámetros de búsqueda asociados a una subscripción del propio usuario. Para ello, el *bot* le solicita al usuario el *subscriptionId* (el cual puede ser obtenido a través del comando *getCurrentSearchs*), y una vez dado, el *bot* responde al usuario con los parámetros de dicha subscripción en caso de que esta última se encuentre recogida en el sistema.
  - *endSearch*: esta función se encuentra asociada al comando *endSearch*, y tiene como objetivo eliminar una subscripción de un usuario en caso de que este último lo desee.

- *send_default_message*: esta función es utilizada por el *bot*, en caso de que el mensaje recibido por parte del usuario no fuese uno de los contemplados por el sistema. A continuación, en la *Ilustración 27* se muestra el contenido de dicha función:

```
@bot.message_handler(func=lambda msg: True)
def send_default_message(message):
    response = "¿Necesitas ayuda para comenzar? Typea el comando /help para aprender a utilizar RentingScraper"
    bot.reply_to(message, response)
```

*Ilustración 27: Función send_default_message*

Como se puede observar, haciendo uso del decorador *message_handler* y con ayuda de una expresión lambda (no es más que una pequeña función en *Python* de pequeño contenido), se asocia la ejecución de esta función al envío de cualquier mensaje no contemplado por el sistema.

El resto de las funciones del fichero que se explican a continuación no están asociadas a ningún comando del *bot* como tal. Algunas de ellas son funciones que están asociadas a pasos intermedios de un comando, por ejemplo, en la función *initSearch*, se hace uso del método *register_next_step_handler* de la librería *telebot*, para indicar que el próximo mensaje recibido del cliente será manejado por la función *processSubscription*. El resto de las funciones son herramientas que el propio servicio utiliza para manejar el *bot* de acorde a las necesidades de la aplicación, como es el caso de la función *refreshSubscription* (se explicará a continuación).

- *searchAnnounces*: utilizada por el servicio para realizar la búsqueda de anuncios en los distintos portales web utilizados (*Fotocasa*, *Milanuncios*, *Yaencontre* y *Pisos*). Para ello se hace uso de las funciones explicadas en el apartado *4.3.1*.
- *refreshSubscription*: esta función es la que usa el servicio para realizar una nueva búsqueda de una subscripción previamente registrada en el sistema la cual precisa ser actualizada. Para ello se apoya en la función anterior, *searchAnnounces* y en otra función *showAnnounces* (se explicará a continuación).
- *processSubscription*: esta función es la que utiliza el *bot* una vez ha recibido el formulario de subscripción por parte del cliente. Su lógica es la siguiente:

  1. Verifica que los parámetros dados por el usuario son válidos mediante el uso de la función *checkParameters* (definida en el fichero *auxFunctions.py*).
  2. En caso de que los parámetros sean válidos, se genera un nuevo *subscriptionId* (un *hash* de 32 *bits*) y se registra la subscripción en el sistema.
  3. Si la subscripción ha sido almacenada correctamente en la base de datos, se procede a realizar la búsqueda de anuncios teniendo en cuenta los datos indicados por el usuario.
  4. Una vez terminado el proceso de búsqueda, se indica al usuario el número de anuncios que han sido encontrados, además de las distintas opciones de visualización de estos (se comentará más en detalle en el capítulo siguiente).

- *showAnnounces*: esta función tiene como finalidad mostrar los anuncios encontrados al usuario, teniendo en cuenta la opción de visualización que este haya indicado. Para ello, hace una consulta a la base de datos solicitando los anuncios que correspondan, filtrando por las preferencias de visualización y por la fecha en la que fueron encontrados, escogiendo los de fecha y hora más reciente.
- *getParametersFromSubscription*: esta función es la que se ejecuta una vez el usuario ha indicado el *subscriptionId* de la subscripción que quiere consultar, habiendo ejecutado previamente el comando *getDataFromSearchWithId*. Al ejecutar la función, se realiza una consulta a la base de datos, solicitando los parámetros de búsqueda registrados para la subscripción que el usuario indicó, y una vez obtenidos, los devuelve al usuario a través del chat.
- *ensureBeforeDeleteSubscription*: utilizada por el *bot* para preguntar al usuario si realmente desea eliminar la subscripción que indicó tras llamar al comando *endSearch*. Para ello, verifica si existe una subscripción registrada en el sistema con el *subscriptionId* indicado por el usuario, en caso afirmativo le pregunta al usuario si está seguro de eliminar la subscripción.

- *confirmUnsubscription*: utilizada por el *bot* para eliminar una subscripción del sistema una vez el usuario ha asegurado que desea finalizar la subscripción con el identificador que indicó previamente.

Cabe mencionar que además de dichas funciones, en este fichero *Python* podemos observar la creación de una instancia de la clase *Telebot* de la propia libería *telebot* de *Python*. Esta instancia se guarda en una variable de ámbito global, con el fin de que pueda ser manejado posteriormente por el propio ejecutable del servicio (el cual será explicado más adelante). A continuación, se expone en la *Ilustración 28* lo anteriormente mencionado:

```
BOT_TOKEN = "6028366071:AAEqkgpc2wgBXayeAGo1H6tehrKwyHOk5Ao"
bot = telebot.TeleBot(BOT_TOKEN)
```

*Ilustración 28: Creación de la instancia de Telebot*

Como podemos observar, para la creación de la instancia hacemos uso del *token* que obtuvimos durante la creación del *bot* mediante el chat con *BotFather*.

- **auxFunctions.py**: este fichero de *Python* contiene un conjunto de funciones auxiliares utilizadas por el módulo anteriormente explicado *rentingScraperBot.py*. Las funciones en cuestión son las siguientes:

  - *deleteAccentMarks*: esta función es exactamente la misma que la del módulo *common_resrc.py* del directorio *scrapers*. Su objetivo es eliminar caracteres acentuados del formulario dado por el usuario, con el fin de evitar errores futuros.
  - *createRegex*: esta función se encarga de generar una expresión regular para la extracción de cada uno de los campos del formulario de registro.
  - *checkParameters*: su objetivo es verificar si el formulario fue cumplimentado correctamente por el usuario. Para ello, se apoya en la función anterior, *createRegex*, para extraer los datos de cada uno de los campos del formulario. El siguiente diagrama de flujo de la *Ilustración 29* muestra la lógica utilizada en esta función para cada uno de los campos:

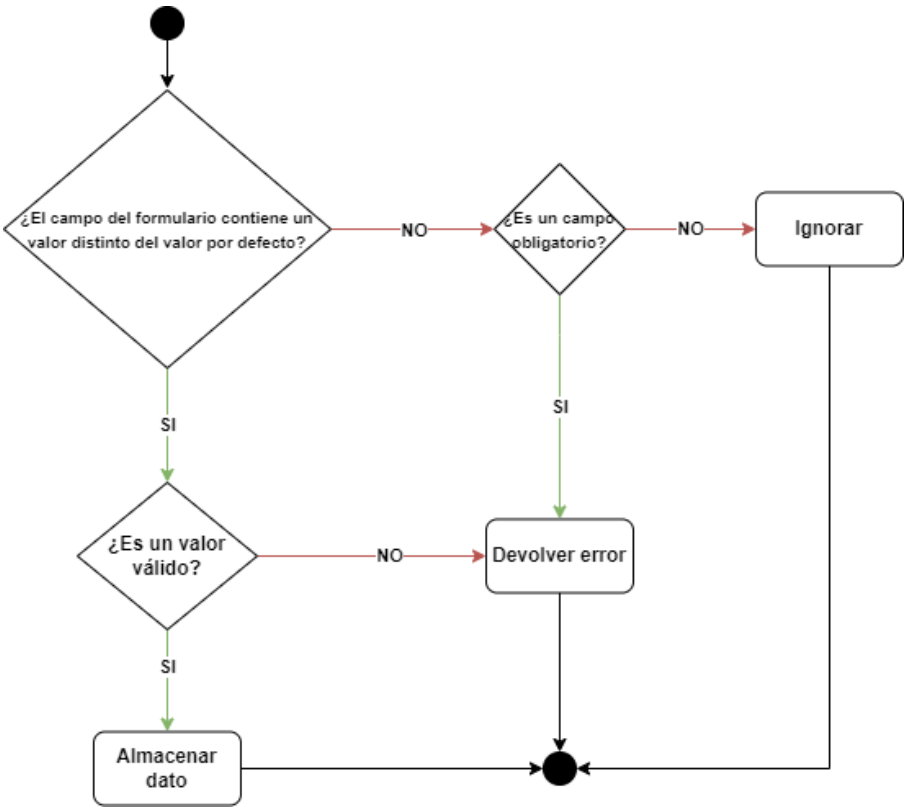



*Ilustración 29: Diagrama de flujo checkParameters*

En caso de que se detectase algún tipo de error en alguno de los campos, este se notificaría al usuario a través del chat.

- *checkIfSubscriptionExists*: comprueba si el usuario en cuestión tiene una subscripción vigente con los mismos parámetros que indicó en el formulario, de manera que no existan duplicidades.
- *saveSubscription*: se encarga de almacenar una subscripción en la base de datos del sistema.
- *deleteSubscription*: esta función es utilizada por el *bot* para eliminar una subscripción de la base de datos del sistema.

- **validData.py**: este fichero de *Python* contiene dos listas constantes las cuales son utilizadas por el *bot* para verificar si los parámetros 'Provincia y 'Tipo de vivienda' indicados por el usuario son válidos o no. A continuación, se exponen las dos constantes recogidas en este fichero:

  - *cities*: esta lista de *Python* almacena en cadenas de texto las distintas provincias de la nación española, de forma que si el usuario indica una provincia que no se encuentra en esta lista, se detenga el procesado de la subscripción, reportando al usuario el error producido.
  - *houseTypes*: esta otra lista contiene los distintos posibles valores para el parámetro 'tipo de vivienda', de manera que si el usuario indica un valor que no se encuentra recogido en esta lista, se detenga el procesado de la subscripción, reportando al usuario que el valor dado para ese parámetro no es válido.

### 4.3.3 Aplicación Python: *logger*

En este apartado se mostrará el contenido del directorio *logger* del proyecto. En él podemos encontrar el fichero *myLogger.py*, el cual contiene una instancia única de la clase *Logger*, de manera que todos los módulos que importen este fichero hagan uso del mismo *logger*, y no se creen diferentes instancias del mismo. A continuación, se muestra en la *Ilustración 30* el contenido de dicho fichero:

src > logger > myLogger.py

```
1  import logging
2  from systemd.journal import JournalHandler
3
4  LOGGER = logging.getLogger('rentingScraper')
5
```

*Ilustración 30: Fichero myLogger.py*

Como vemos, la única acción que se lleva a cabo es la creación de la instancia de la clase *Logger*. Esta se configura en el ejecutable del servicio, el cual será explicado en el apartado de a continuación.

### 4.3.4 Aplicación Python: *ejecutable del servicio*

En este apartado se detallará el contenido del ejecutable del servicio, el cual será llamado por la herramienta *systemd* en el arranque del servicio. El archivo ejecutable, el cual se encuentra en el propio directorio *src*, contiene una clase llamada *RentingScraper*, que a su vez contiene dos métodos, ambos se explican a continuación:

- *__init__*: es el método constructor de la clase. Este método es llamado al crear una nueva instancia de la clase *RentingScraper*. En la *Ilustración 31* se muestra el contenido de dicho método:

```
class RentingScraper():

    def __init__(self):

        while True:
            self.checkSubscriptionsStatus()
            LOGGER.info("Refrescando peticiones pendientes de atender en el bot")
            bot.process_new_updates(bot.get_updates(offset=(bot.last_update_id + 1), timeout=30))
            sleep(5)
```

*Ilustración 31: Clase RentingScraper*

Como podemos observar, en dicha función tenemos un bucle infinito, en el cual llamamos al método de la clase *checkSubscriptionsStatus*, reporta un mensaje en el *logger* y actualiza el estado del *bot*, llamando al método *process_new_updates*. Tras ello, se duerme la ejecución del proceso durante 5 segundos mediante la instrucción *sleep*. En cuanto al método *checkSubscriptionsStatus*, tenemos que este se encarga de verificar si existen subscripciones vigentes en el sistema que necesiten ser actualizadas o no, en cuyo caso, se realizará una nueva búsqueda para cada una de las subscripciones que lo precisen, compartiendo los resultados con el usuario una vez estos han sido correctamente obtenidos. Finalmente, tras compartir los resultados con el usuario, se actualiza el campo *lastUpdate* de dicha subscripción en la base de datos, asignando el instante en el que la subscripción fue actualizada por última vez. A continuación, se expone en la *Ilustración 32* el código de aplicación del método en cuestión:

```
def checkSubscriptionsStatus(self):
    sql = "select * from subscriptions where EXTRACT(EPOCH FROM (current_timestamp - lastupdate))>14400" # 4 hours

    conn = psycopg2.connect(host="localhost", database="rentingscraper", user="dit", password="dit")
    cursor = conn.cursor()
    LOGGER.info("Verificando si existen subscripciones pendientes de actualizar")
    cursor.execute(sql)
    subscriptions = cursor.fetchall()
    if(cursor.rowcount!=0):
        LOGGER.info(f"Hay {len(subscriptions)} subscripciones pendientes de actualizar")
        for sub in subscriptions:
            LOGGER.debug(f"La subscripción con ID {sub[0]} va a ser actualizada")
            paramValues = {'provincia': sub[3], 'amplitud': sub[4], 'tipo de vivienda': sub[5], 'amueblado': sub[6], 'precio maximo': sub[7],
                'precio minimo': sub[8], 'minimo de baños': sub[9], 'minimo de habitaciones': sub[10], 'superficie maxima': sub[11],
                    'superficie minima': sub[12]}
            try:
                success = refreshSubscription(subscriptionId=sub[0], userId=sub[1], chatId=sub[2], paramValues=paramValues)
                if(success==True):
                    sql = f"update subscriptions set lastupdate=\'{datetime.now().strftime('%Y-%m-%d %H:%M:%S')}\' where subscriptionId=\'{sub[0]}\'"
                    cursor.execute(sql)
                    conn.commit()
                    LOGGER.debug(f"La subscripción con ID {sub[0]} ha sido correctamente actualizada")
            except Exception as e:
                LOGGER.error(e)
    cursor.close()
    conn.close()
```

*Ilustración 32: Método checkSubscriptionsStatus*

Por último, dentro del fichero ejecutable podemos observar la llamada a la construcción la única instancia de la clase *RentingScraper*, la cual se encargará de gestionar el servicio de forma completa, además de la configuración del *logger* creado en el fichero *myLogger.py* comentado en el apartado anterior. A continuación, se puede observar en la *Ilustración 33* lo anteriormente mencionado:

```
if __name__ == '__main__':
    JOURNALD_HANDLER = JournalHandler()
    JOURNALD_HANDLER.setFormatter(logging.Formatter('[%(levelname)s] %(message)s'))

    LOGGER.addHandler(JOURNALD_HANDLER)
    LOGGER.setLevel(logging.INFO)
    LOGGER.info("El servicio rentingScraper va a ser inicializado")
    RentingScraper()
```

*Ilustración 33: Creación de la instancia de RentingScraper*

Además del ejecutable del servicio, dentro del directorio *src* también podemos encontrar el fichero de extensión *service*, el cual debe ser copiado al directorio */etc/systemd/systemd* del sistema para que la aplicación pueda ser gestionada por *systemd*. A continuación, en la *Ilustración 34*, podemos observar el contenido de dicho fichero:

src > rentingScraper.service

```
1  [Unit]
2  Description= RentingScraper
3  After=multi-user.target
4  Requires=postgresql.service
5
6  [Service]
7  Type=simple
8  Restart=always
9  ExecStart=/usr/bin/python3 rentingScraper.py
10 WorkingDirectory = /home/dit/rentingScraper/src
11
12 StandardOutput=syslog
13 StandardError=syslog
14 SyslogIdentifier=RentingScraper
15
16 [Install]
17 WantedBy=multi-user.target
```

*Ilustración 34: Fichero rentingScraper.service*

Del contenido de dicho fichero podemos destacar que el arranque de nuestro servicio es totalmente dependiente del servidor *PostgreSQL*. Esto se asigna a través de la directiva *Requires*, la cual implica las dependencias que el servicio necesita para su arranque.

Por último, se debe comentar que dentro de la carpeta raíz del proyecto podemos encontrar el *shell script* llamado *installDependencies.sh* el cual puede ser utilizado para instalar las librerías y los recursos necesarios para que el servicio pueda ser arrancado y operar de manera adecuada. A continuación, se expone en la *Ilustración 35* el contenido de dicho fichero:

$ installDependencies.sh

```
#!/bin/bash

if [ "$EUID" -ne 0 ]
  then echo "Please run as root"
  exit
fi

if echo ${PWD} | grep "rentingScraper" -q
then
    echo "[ INFO ]: Dependencies are going to be installed"
else
    echo "[ ERROR ]: This script must be called from the same directory where it is stored"
    exit 1
fi

apt-get update

if [ -f "/usr/bin/python3" ]
then
    echo "[ INFO ]: Python already installed"
else
    echo "[ INFO ]: Installing python3 package"
    apt update
    apt install python3
fi

if [ -f "/usr/bin/pip" ]
then
    echo "[ INFO ]: Pip already installed"
else
    echo "[ INFO ]: Installing python3-pip"
    apt install python3-pip
fi

if [ -f "/usr/bin/psql" ]
then
    echo "[ INFO ]: Postgresql already installed"
else
    echo "[ INFO ]: Installing PostgreSQL package"
    apt update
    apt install postgresql
fi

# Install necessary python modules
echo "[ INFO ]: Installing python modules"
pip3 install selenium
pip3 install telebot
pip3 install pyTelegramBotAPI
pip3 install bs4
pip3 install requests_html
pip3 install regex
pip3 install pathlib
pip3 install psycopg2-binary

# Install google-chrome package and chromedriver
echo "[ INFO ]: Installing required software for web scraping"
cd resources
apt install ./google-chrome-stable_current_amd64.deb
cp chromedriver /usr/bin/
cd ..

# Configure app as service
echo "[ INFO ]: Configuring application as service"
cp src/rentingScraper.service /etc/systemd/system/
chmod 644 /etc/systemd/system/rentingScraper.service
systemctl daemon-reload
```

*Ilustración 35: Fichero installDependencies.sh*

Como se puede apreciar, es necesario que dicho fichero sea ejecutado como usuario *root*. Una vez ejecutado, este verificará si el software necesario se encuentra instalado en el sistema, instalándolo en caso negativo. Además, se instalarán las librerías de *Python* necesarias a través del administrador de paquetes de *Python*, *PIP*, solo en caso de que estas no se encuentren previamente instaladas. También se instalará la la versión de *Google Chrome* utilizada y su *driver Chromedriver* (manualmente modificado por el desarrollador como se comentó en el apartado *2.3*). Por último, copiará el fichero *rentingScraper.service* en el directorio del sistema */etc/systemd/system/* para que este pueda ser gestionado por *systemd*.

# 5 Despliegue del proyecto

Antes de detallar como se llevaría a cabo el despliegue de la aplicación, debe tenerse en cuenta que este proyecto ha sido desarrollado para proveer un servicio a nivel local, con objetivos experimentales, de manera que lo explicado en el capítulo 4 de esta memoria debería ser adaptado para un despliegue a gran escala para dar un servicio a nivel nacional.

Teniendo en cuenta el comentario anterior, para que pudiéramos proveer un servicio de un nivel de calidad óptima al usuario deberíamos de tener un *Data Center* operativo con un significativo número de servidores, todos ellos replicas unos de otros, siendo estos conectados a un equipo *Proxy* el cual redirige la petición del usuario a uno de ellos, de forma que la carga del servicio se encuentre correctamente balanceada y no haya ningún equipo servidor saturado por peticiones de distintos usuarios. Además, la base de datos del sistema tendría que estar centralizada en un equipo externo a los que contienen el código de la aplicación, accesible a cada uno de los equipos servidores y por supuesto con sus respectivas replicas, de manera que si una falla, el servicio siga estando operativo. También deberíamos de contar con un conjunto importante de servidores *Proxy* con diferentes direcciones IP públicas utilizados para reencaminar las peticiones *HTTP* hacia los distintos portales web que utiliza el servicio para la extracción de los datos. La razón de esto es que si los servidores web que utilizamos para buscar la información detectan un acceso elevado [24] [25] a sus servidores [por parte de una misma dirección IP, estos podrían bloquearnos el servicio, impidiéndonos así el poder realizar nuestra actividad de manera normal. Este comportamiento que se menciona ha sido experimentado durante la etapa de desarrollo del proyecto, tras realizar un importante número de accesos a los distintos servidores web utilizados para el *Web Scraping* de los datos. En la *Ilustración 36*, puede apreciarse este aspecto de forma clara:

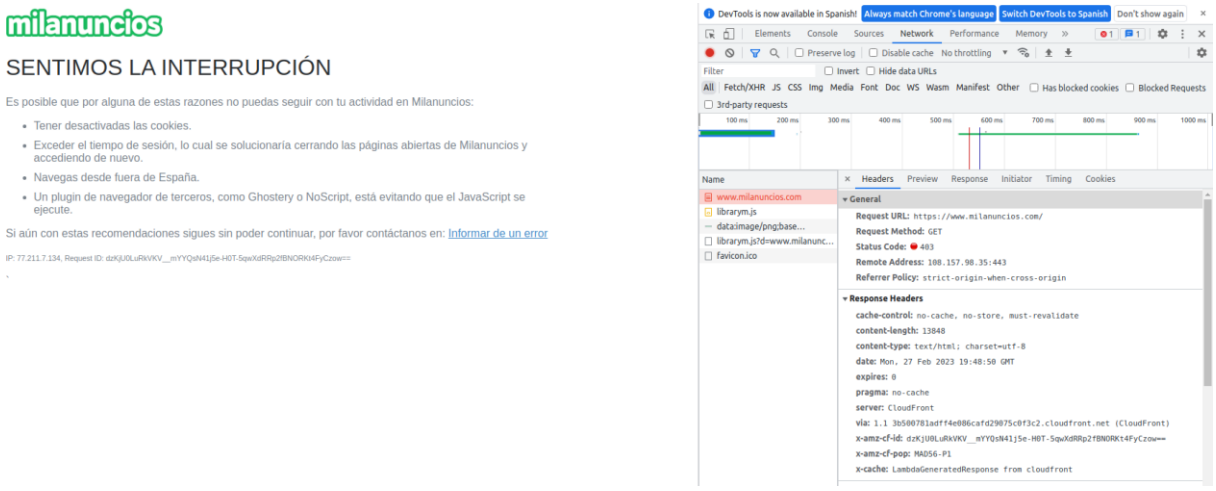



*Ilustración 36: Denegación de acceso a Milanuncios*

Teniendo en cuenta todo lo anterior explicado en este apartado, la arquitectura para el despliegue de nuestro servicio para una escala a nivel nacional quedaría aproximadamente tal y como se muestra en la *Ilustración 37*:

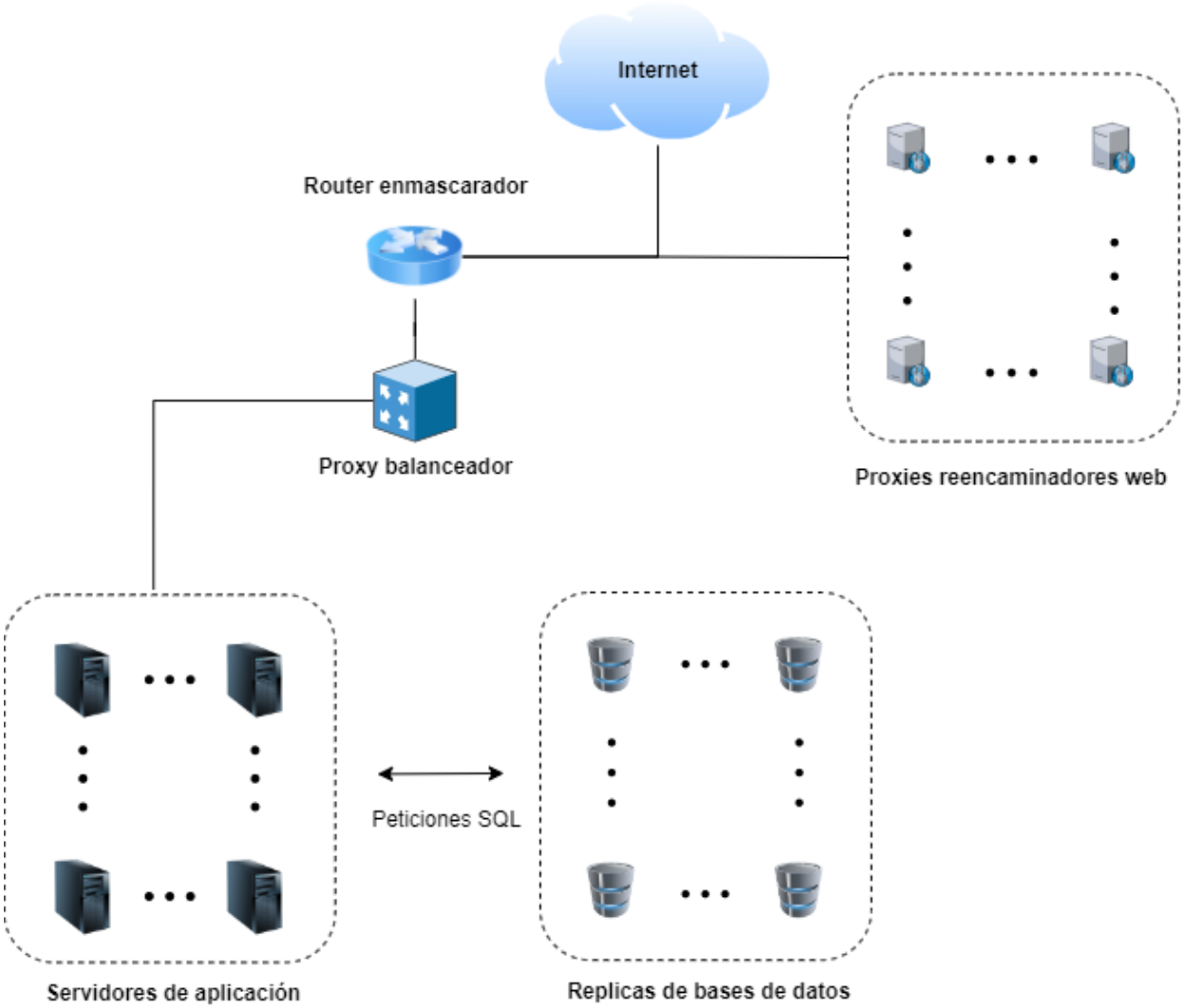



*Ilustración 37: Despliegue del servicio*

El diagrama anterior sería una aproximación del escenario de despliegue del servicio en la realidad. Como podemos ver tendríamos nuestros equipos servidores y bases de datos en una subred privada de acceso restringido enmascarada por el *router* del operador el cual nos diese servicio. Esta red privada sería de acceso restringido mediante un *firewall* (el cual podría estar implementado en el mismo equipo que el *proxy* balanceador de carga) el cual permitiría el acceso desde el exterior solamente a las peticiones del cliente del servicio. En cuanto a las consultas que se hicieran a los distintos portales web para conseguir los datos requeridos por el cliente, haríamos uso del conjunto de *proxies* los cuales deberían de ser públicos desde el punto de vista de la red (es decir, cada uno de ellos con una dirección IP pública), pero con un acceso restringido de forma que solo pudieran ser utilizados por nosotros, permitiéndonos así balancear el tráfico para que los distintos servidores web que consultamos no nos denieguen el acceso.

# 6 Demostración práctica

Una vez el servicio ha sido correctamente desplegado, y este se encuentra disponible de cara al usuario, podemos proceder a su puesta en marcha, de manera que los distintos usuarios que requieran el uso de la aplicación puedan hacerlo. A continuación, se irán mostrando distintas ilustraciones que enseñarán el funcionamiento de la aplicación una vez desplegada, siendo estas acompañadas de sus respectivos comentarios de forma que el lector pueda enlazar el contenido ilustrado con las explicaciones dadas a lo largo de todo este documento, especialmente el capítulo 4 *Implementación del servicio*.

Para comenzar a utilizar el servicio, primeramente, deberíamos crearnos una cuenta de *Telegram*, suponiendo que no dispongamos de ella. Para ello, se precisa tener de un *smartphone* junto con una línea móvil con acceso internet. Teniendo esto, debemos acceder al *market place* del sistema operativo de nuestro *smartphone*, *Play Store* o *App Store*, según tengamos *Android* o *iOS*. Una vez instalada la aplicación en el terminal, complementados los pasos de registro hasta finalizar y ya podríamos acceder al servicio *RentingScraper*. Para ello, solamente tendríamos que escribir en el buscador de *Telegram* el nombre del *bot*, el cual es *RentingScraper*. En la *Ilustración 38* puede observarse este último paso que se menciona:

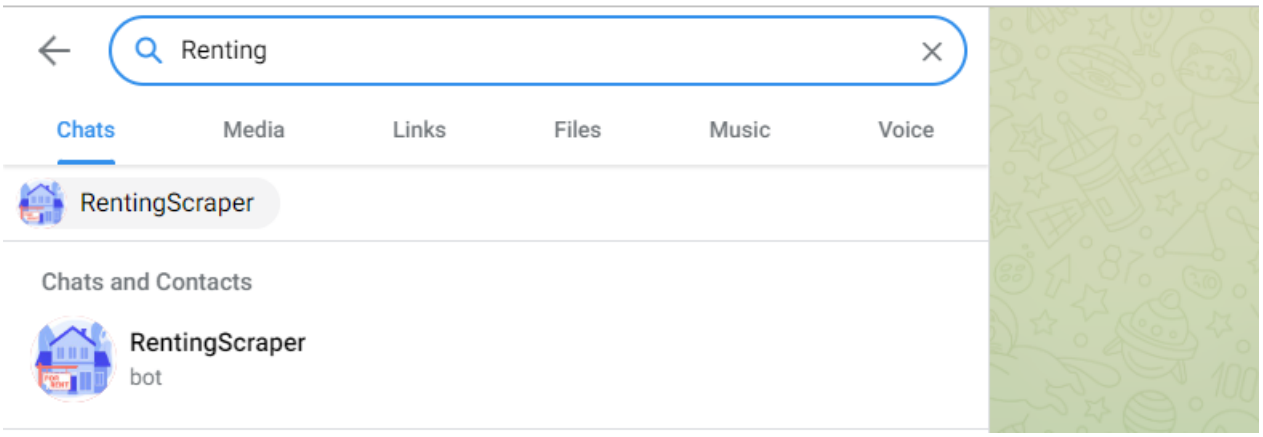



*Ilustración 38: Acceso al bot RentingScraper*

Como podemos observar, nos aparece el *bot* disponible en el buscador una vez comenzamos a escribir el nombre de este. Haciendo click o pulsando sobre el mismo (según estemos usando la aplicación de *Telegram* o su versión web), podremos acceder al chat del *bot*. A continuación, se muestra en la Ilustración 39 el aspecto inicial del chat una vez lo abrimos por primera vez:

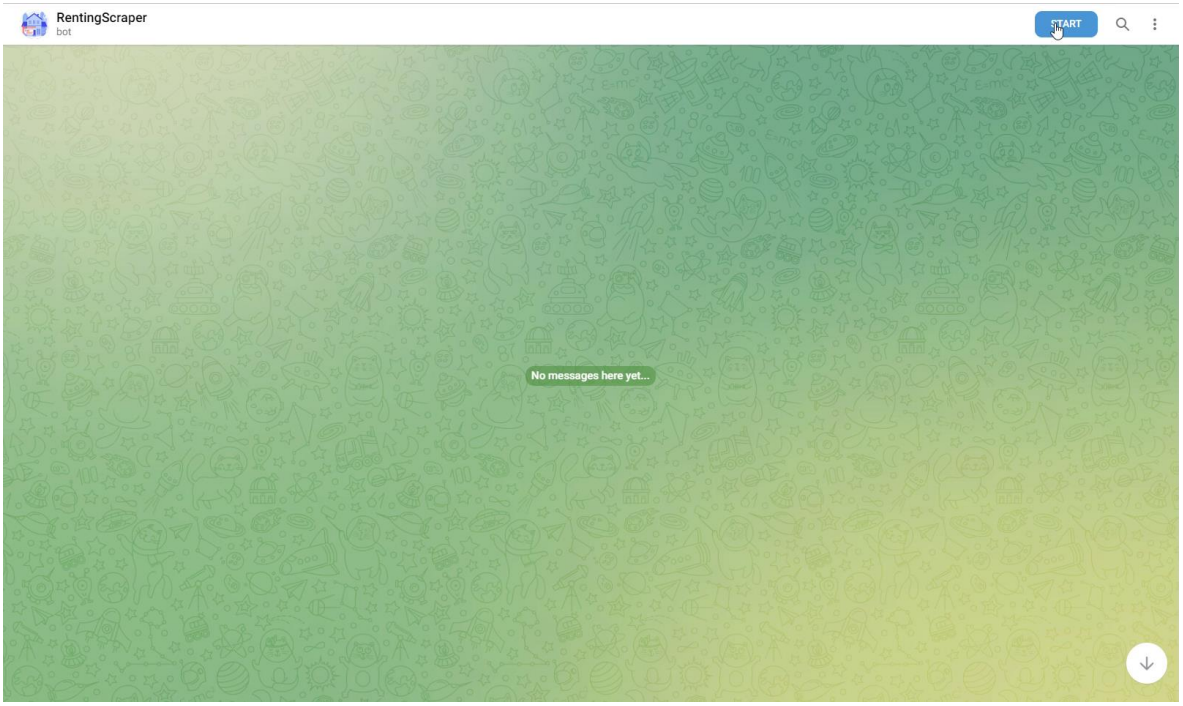



*Ilustración 39: Chat de RentingScraper*

Si presionamos sobre el botón *Start*, el cual se encuentra en la esquina superior derecha del chat, obtenemos como resultado lo mostrado en la *Ilustración 40*:

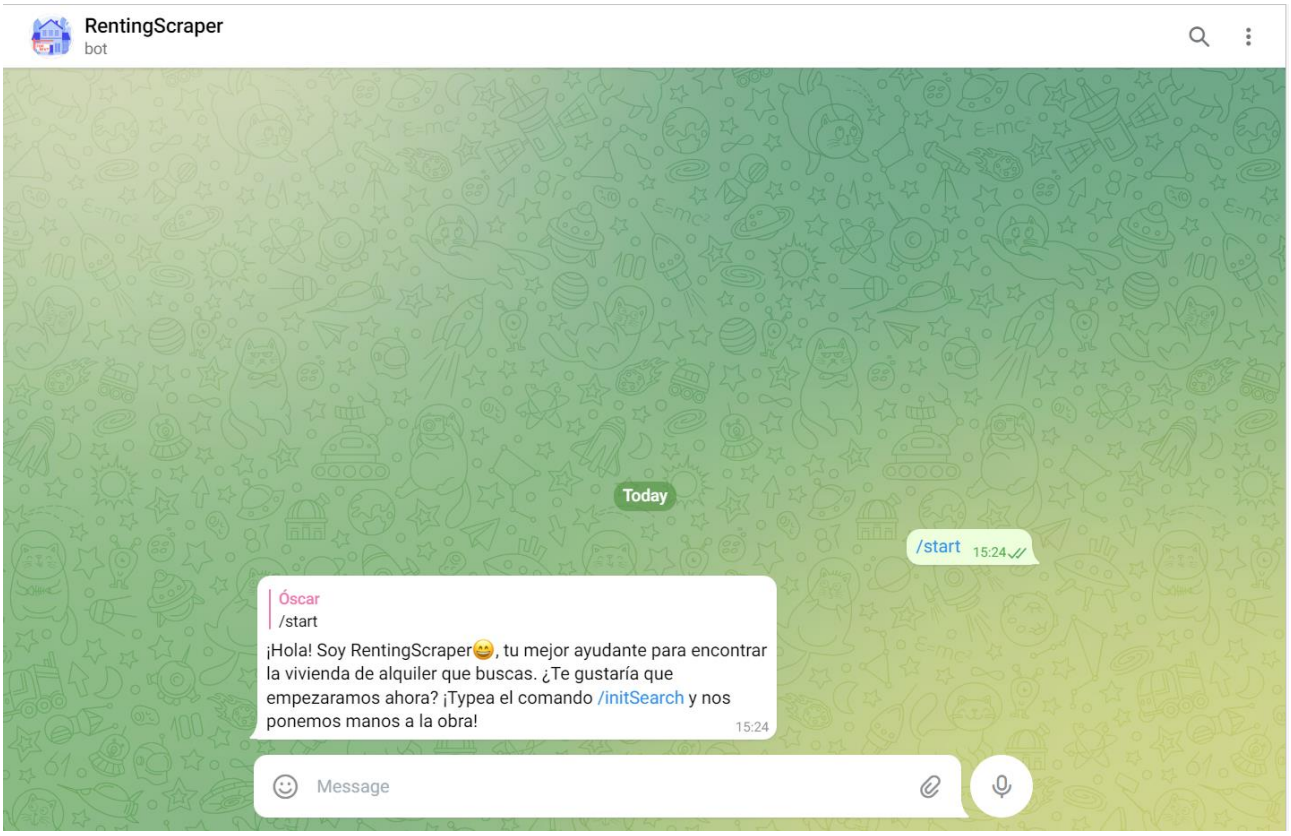



*Ilustración 40: Inicio de conversación con RentingScraper*

Como podemos observar, ante el comando *start*, el *bot* nos responde con el contenido del archivo *welcome.md*, el cual como se comentó anteriormente, se encuentra en el directorio *resources*. Al comienzo de una nueva conversación, como se aprecia en la imagen, el *bot* nos sugiere empezar una nueva búsqueda mediante el uso del comando *initSearch*. En cuanto al *logger* del servicio. Según el nivel de *log* que usemos obtendremos una mayor o menor información de salida. En este caso, el nivel de log por defecto es *INFO*, por lo que la salida que obtenemos inicialmente es la mostrada en la Ilustración 41:

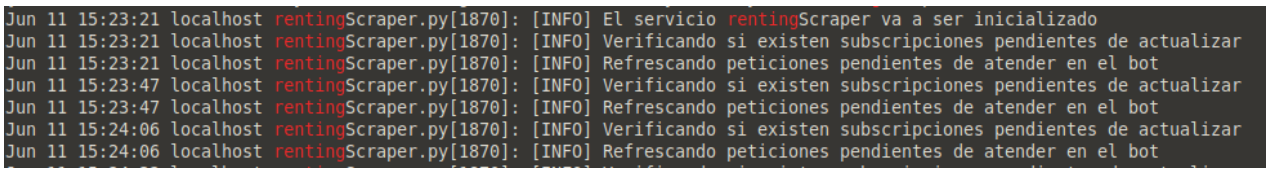



*Ilustración 41: Información inicial del logger*

Como vemos, el *logger* nos informa de que se va a proceder con la inicialización del servicio, y tras ello, comienza con la verificación periódica del estado de las diferentes suscripciones que se encuentren registradas en el sistema. Si continuamos con la conversación actuando como un usuario normal el cual desconoce totalmente el funcionamiento de la aplicación, tenemos el resultado que se muestra a continuación en la *Ilustración 42*:

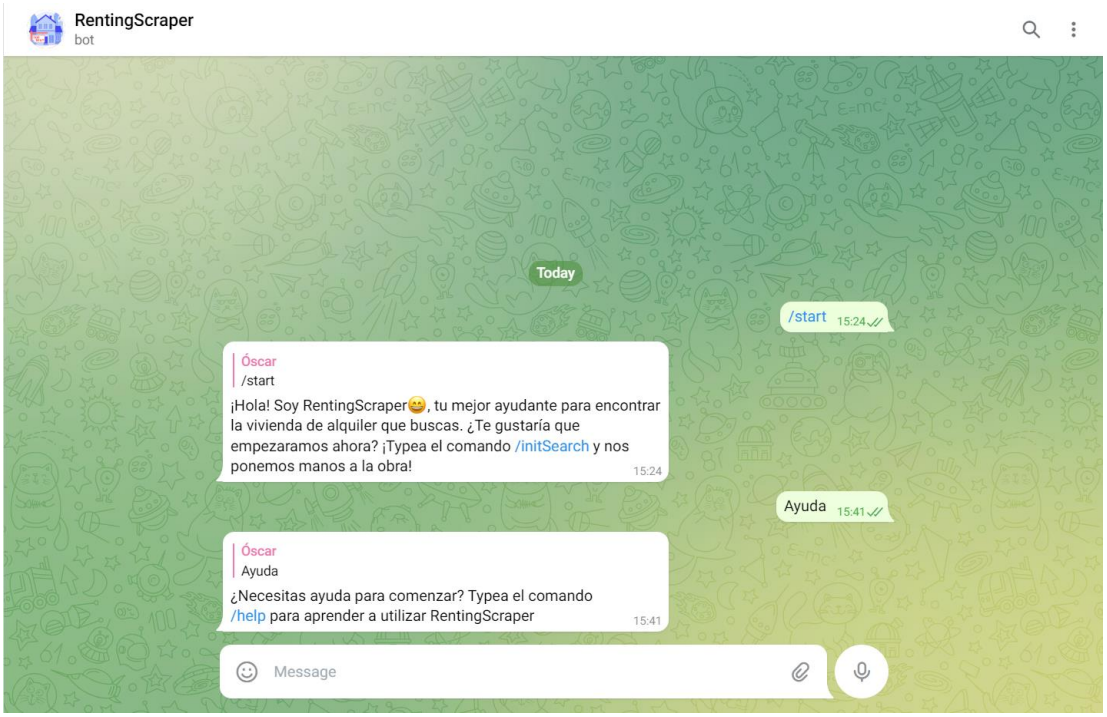



*Ilustración 42: Mensaje por defecto del bot*

Como usuario que desconoce el funcionamiento de *bot*, y por ende, los distintos comandos que este utiliza, el usuario mandaría un mensaje el cual el *bot* no contempla, y por tanto, este respondería haciendo uso del mensaje por defecto. Como vemos, el *bot* al no entender lo que queremos decirle, este nos sugiere que hagamos uso del comando *help* para empezar a conocer el funcionamiento de la herramienta. Si escribimos dicho comando en el chat, obtenemos lo que se muestra en la *Ilustración 43*:


*Ilustración 43: Mensaje de ayuda del bot*

Como vemos, ante el envío del comando *help*, el bot nos responde con el contenido del fichero *help.md* almacenado en el directorio *resources*, el cual se encuentra dentro de la carpeta *telegramBot*. En la respuesta recibida, tenemos que el *bot* nos indica los distintos comandos disponibles con los cuales podemos gestionar el propio *bot*. Dichos comandos son exactamente los mismos que los comentados en el capítulo 4 de la memoria. El funcionamiento de todos ellos se íra mostrando a lo largo de este capítulo.

Continuando con la conversación del chat, si hacemos uso del comando *aboutMe*, para poder conocer un poco más la herramienta y saber cuál es la motivación de ella, obtenemos como respuesta el mensaje mostrado en la *Ilustración 44*:

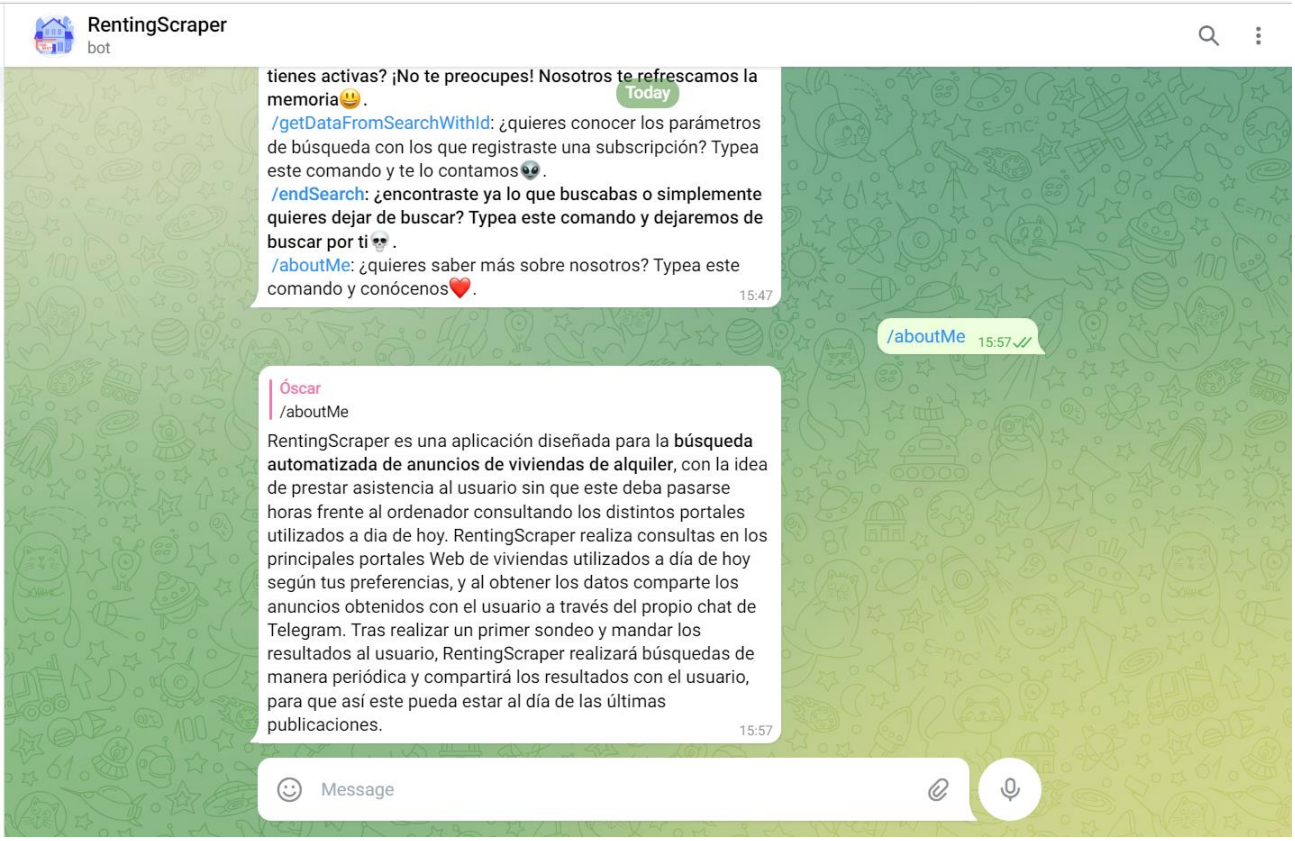



*Ilustración 44: Mensaje aboutMe*

Como se aprecia, el mensaje recibido nos da la información suficiente para conocer sobre qué trata la herramienta y qué funcionalidad nos ofrece esta misma. El mensaje mostrado pertenece al fichero *aboutMe.md*, el cuál podemos observar en la *Ilustración 26* de esta memoria.

Bien, una vez el usuario conoce suficientemente todos los detalles de la herramienta, este puede proceder a realizar su primera subscripción en el servicio. Pero antes, comprobamos que efectivamente, no existe ninguna subscripción nuestra. Para ello, usando el comando *getCurrentSearchs* podemos obtener dicha información. A continuación, se muestra en la *Ilustración 45* el resultado de escribir dicho comando:

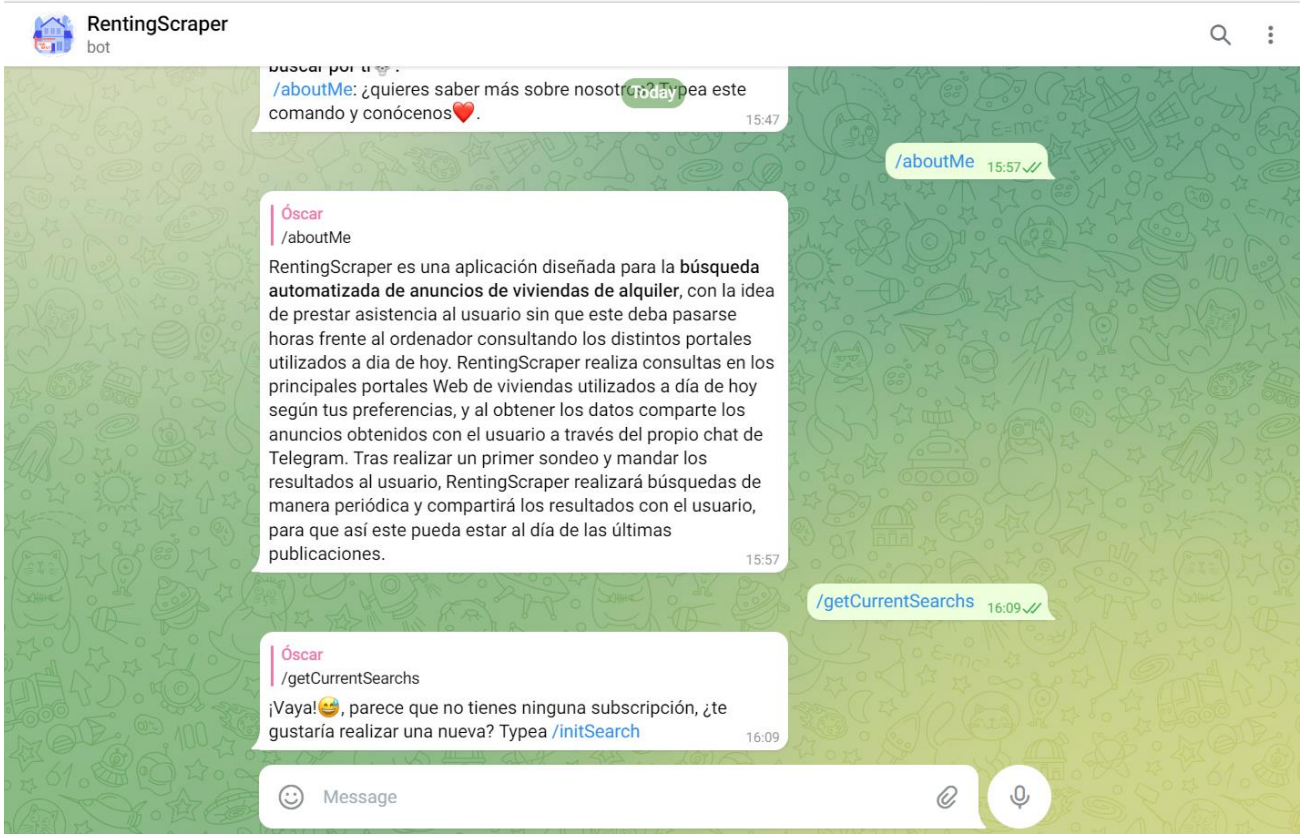



*Ilustración 45: Subscripciones iniciales en el sistema*

Como se puede apreciar en la captura anterior, efectivamente no consta ninguna subscripción nuestra en el servicio. Para comenzar con una nueva subscripción, debemos hacer uso del comando *initSearch*, el cual el *bot* nos sugiere su uso tal y como accedemos al chat de este. Antes de comenzar, cabe mencionar que en la implementación actual, el servicio permite tener tantas subscripciones diferentes como se quieran para un mismo usuario. Bien, dicho esto podemos proceder con la subscripción. Si escribimos el comando mencionado, obtenemos como resultado el mostrado en la *Ilustración 46*:

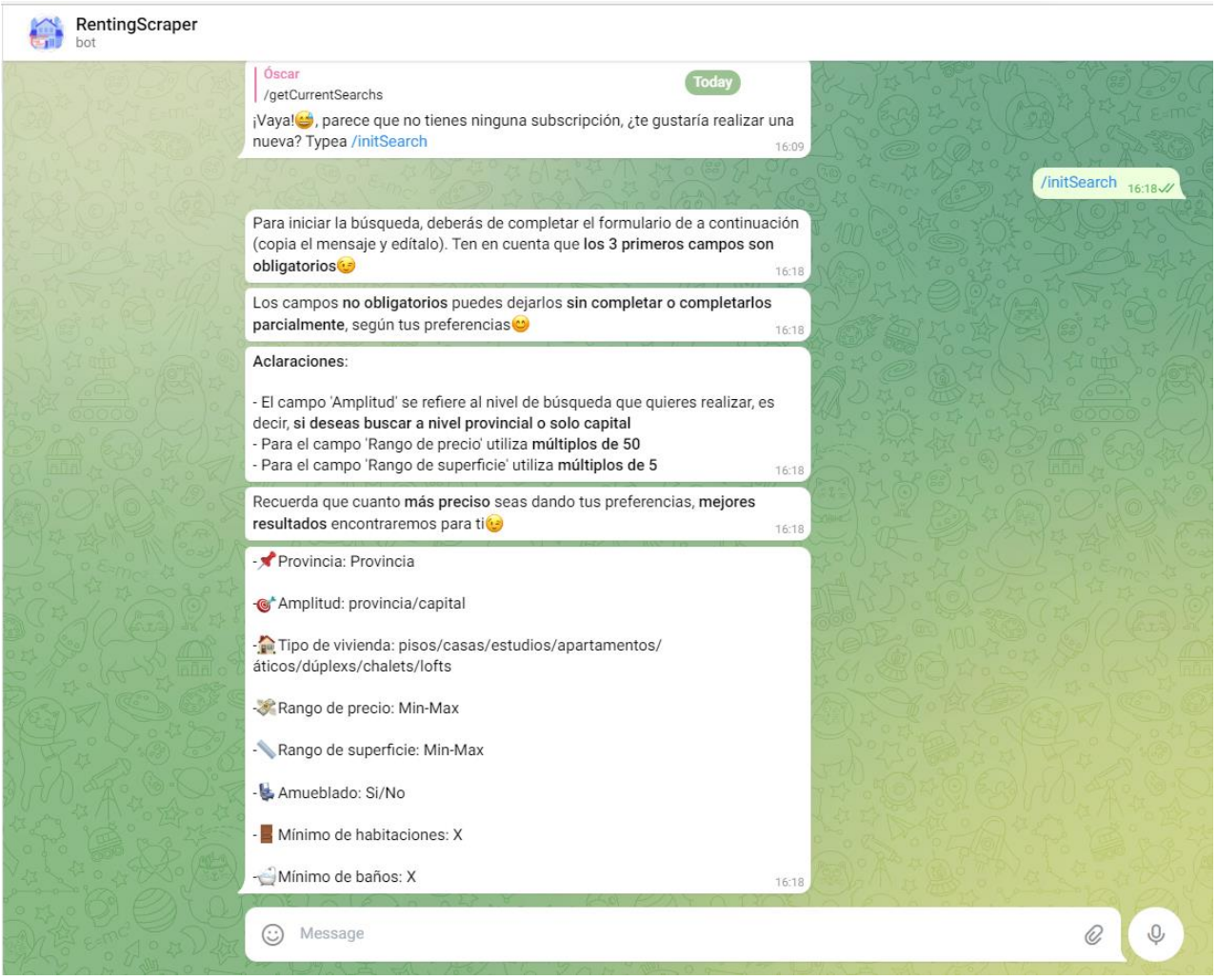



*Ilustración 46: Respuesta al comando initSearch*

Como respuesta por parte del *bot*, obtenemos una serie de mensajes descriptivos los cuales nos indican la forma adecuada de proceder con el registro de la subscripción. El primer mensaje en cuestión nos indica que copiemos el último mensaje y lo editemos para cumplimentar el registro, resaltando que los 3 primeros campos son obligatorios. El segundo mensaje nos comenta que los campos no obligatorios podemos dejarlos sin completar, es decir, por defecto, si no tenemos interés en esos campos. El tercer mensaje nos indica como debemos proceder cumplimentando ciertos campos del formulario. En cuanto al cuarto mensaje, este nos recalca que cuanto más preciso seamos al dar la información, mejores resultados encontrará, por lo que debemos tenerlo en cuenta. Por último, tenemos el quinto y último mensaje, el cual se corresponde con el formulario de subscripción del servicio. Si cumplimentamos este último de manera adecuada, obtendremos un resultado similar al mostrado en la *Ilustración 47*:


*Ilustración 47: Respuesta ante una subscripción exitosa*

Como vemos, tras cumplimentar el formulario correctamente, el *bot* nos indica que nuestra subscripción será procesada y en un plazo corto de tiempo obtendremos nuestros resultados. A continuación, en la *Ilustración 48* podemos observar la respuesta obtenida una vez la búsqueda ha finalizado:


*Ilustración 48: Respuesta del bot tras finalizar la búsqueda*

Como podemos observar, tras finalizar la búsqueda este nos indica el número de anuncios encontrados, y nos manda otro mensaje indicándonos las distintas opciones que tenemos para la visualización de los mismos. Como se observa, todas ellas son bastante explícitas y descriptivas. Profundizando un poco más en la opción 4, tenemos que esta opción nos despliega los 10 anuncios que el servicio ha considerado que son los de mejores prestaciones mediante el uso de la función *calculateScore*, detallada en el capítulo anterior. Si consultamos el lado del servidor, podemos observar que la información reportada en el *log* del servicio es la mostrada en la *Ilustración 49*:

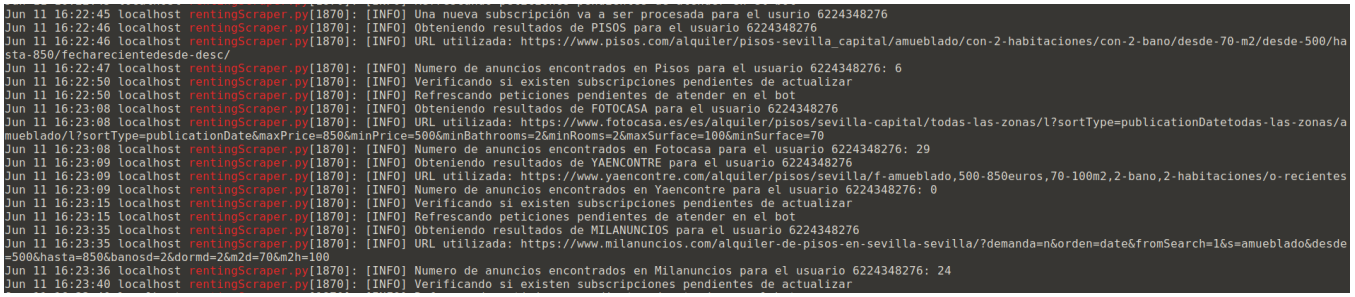



*Ilustración 49: Salida del logger al registrar una nueva subscripción*

Como se puede apreciar en la captura anterior, el *logger* nos informa que una nueva subscripción se va a procesar y nos despliega el número de resultados obtenidos para cada uno de los portales web consultados. Si observamos el estado de la base de datos, obtenemos que el resultado de la tabla *subscriptions* es el mostrado en la *Ilustración 50*:

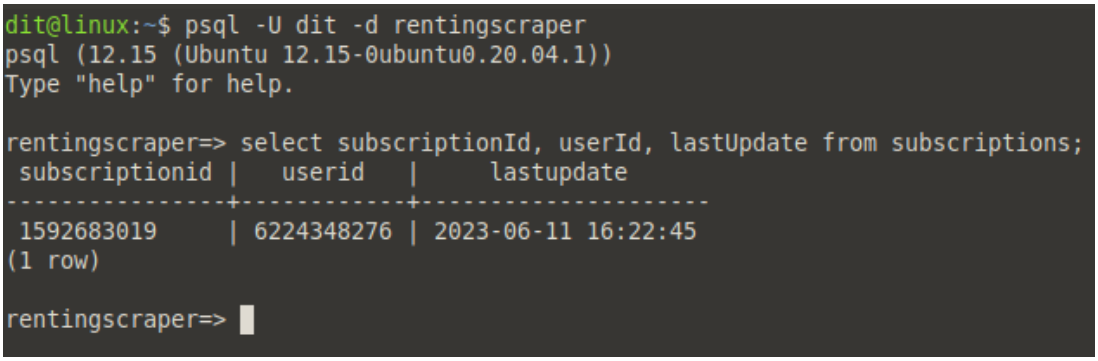



*Ilustración 50: Estado de la base de datos tras la subscripción*

Como vemos la subscripción ha quedado correctamente recogida en el sistema, y por tanto esta será actualizada periódicamente cada 4 horas mientras el usuario no decida eliminarla. Si continuamos con la parte cliente del servicio, y seleccionamos una de las distintas opciones que tenemos disponibles, tenemos como resultado lo que se muestra en la *Ilustración 51*:

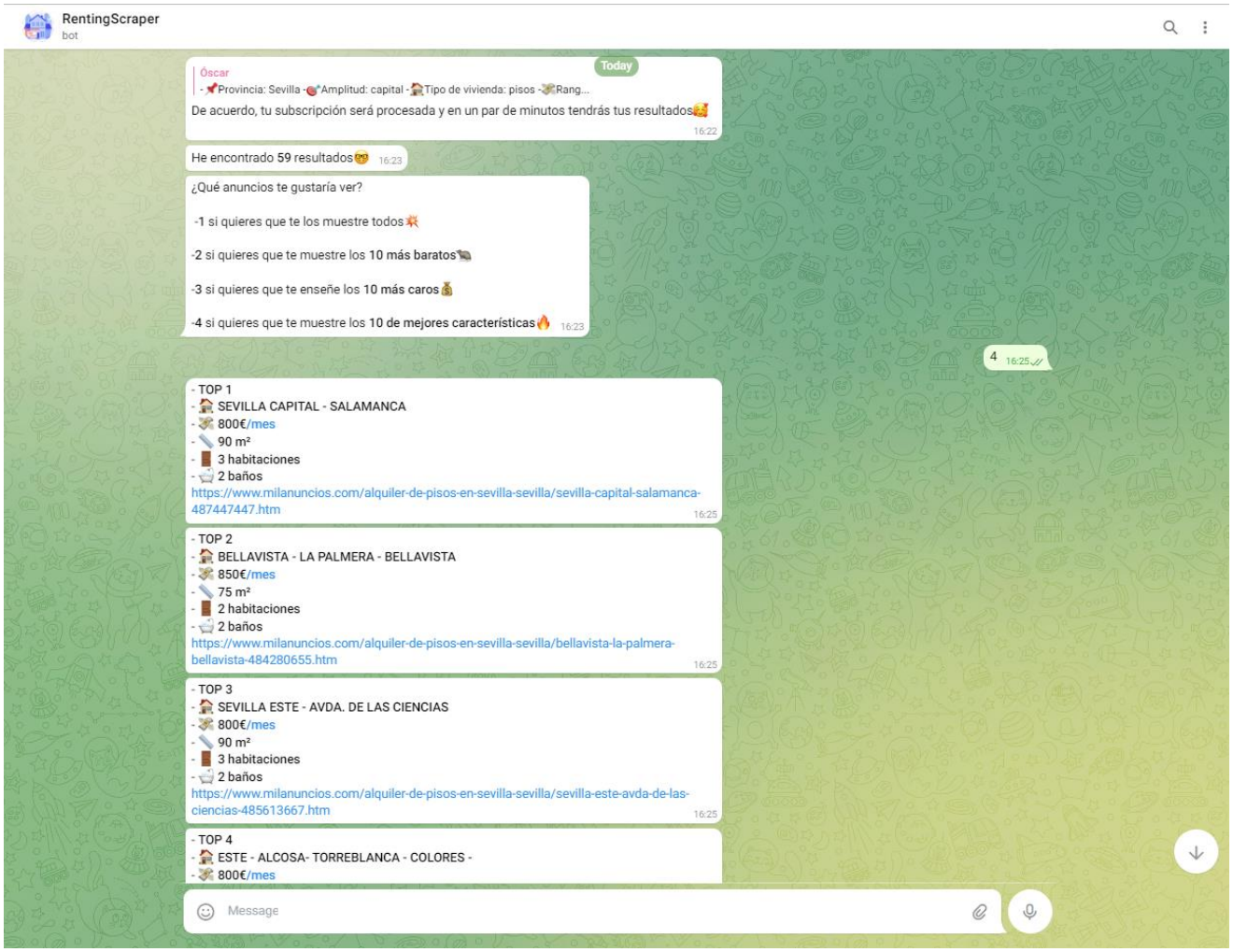



*Ilustración 51: Anuncios obtenidos tras la búsqueda*

Tras seleccionar la opción número 4, tenemos que el *bot* nos manda en una lista descendiente los anuncios de mejores características encontrados en esta primera búsqueda. Como vemos, en cada uno de los mensajes aparece el título del anuncio, el precio de la vivienda, sus principales características (superficie, habitaciones y baños) y por último, el enlace del mismo, por si queremos consultar más información. Si accedemos al enlace del anuncio de mejores características según el servicio, por ejemplo, obtenemos la página mostrada en la *Ilustración 52*:

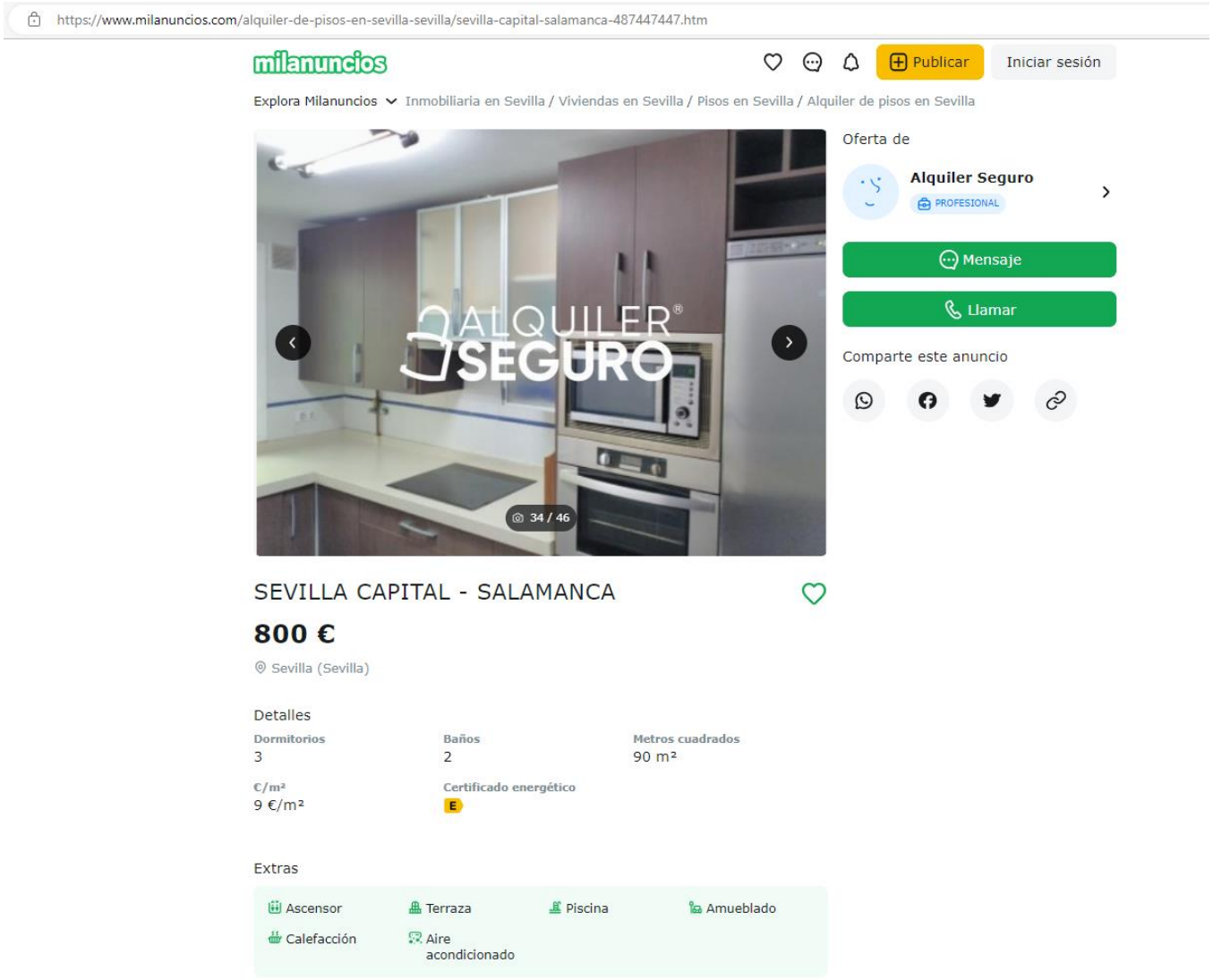



*Ilustración 52: Mejor resultado obtenido*

Como se puede apreciar en la parte inferior de la captura, se observa que la vivienda cuenta con una gran cantidad de extras, como piscina, terraza, ascensor, etc. Además, si nos fijamos en las características de este y comprobamos con los filtros de búsqueda que introdujimos (*Ilustración 47*), vemos que el anuncio cumple todas las especificaciones que indicamos, por lo que tanto el filtrado de anuncios como la puntuación de los mismos parece que funciona como se espera. Volviendo al chat, si observamos el anuncio número 10 mostrado en la *Ilustración 53*, el cuál es el de peores características de los 10 dados según el servicio:

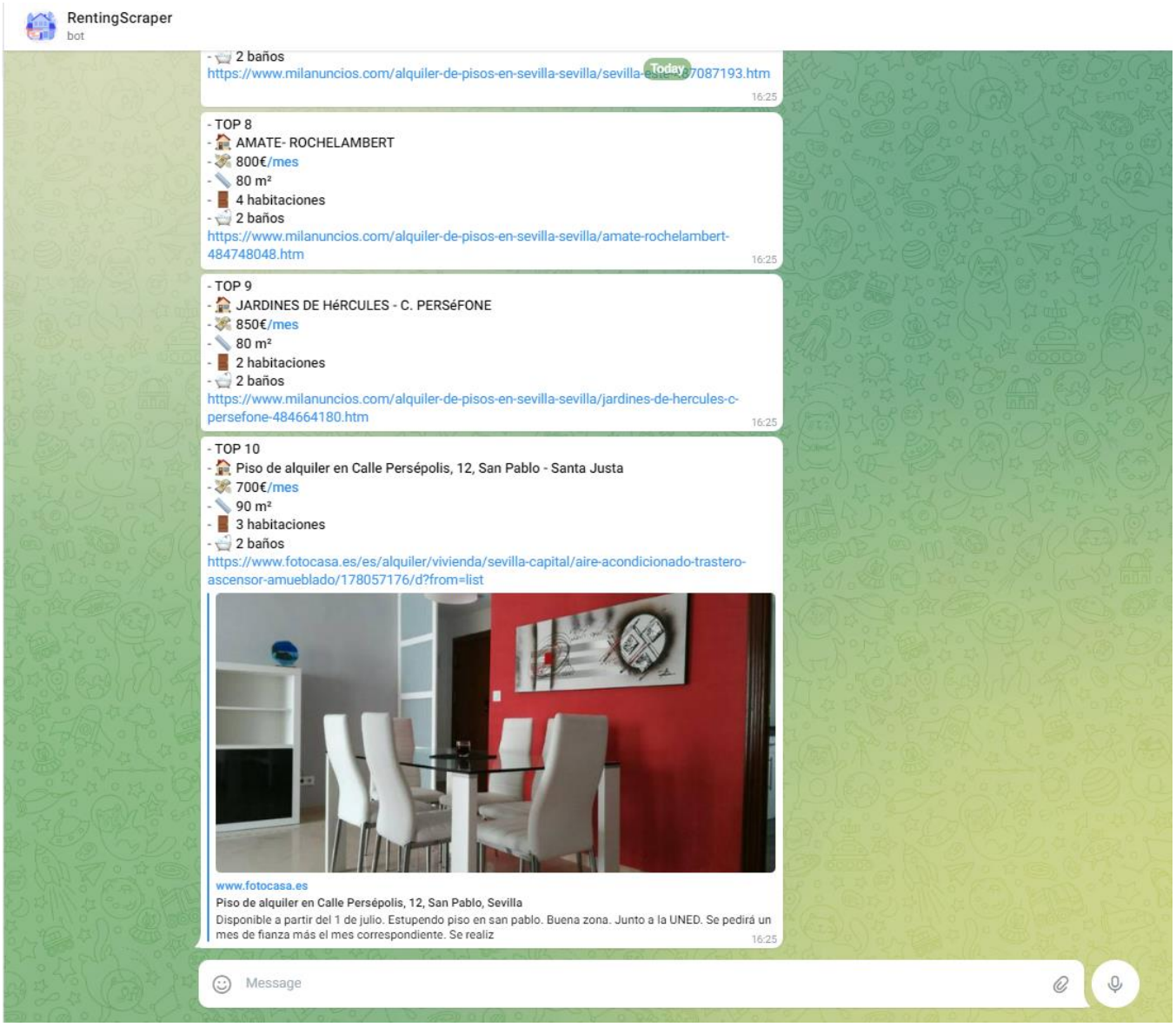



*Ilustración 53: Anuncios de peor puntuación según el servicio*

Como vemos, a diferencia de los otros anuncios, este último pertenece al portal de *Fotocasa*, además, el chat nos renderiza la previsualización del enlace, permitiéndonos obtener así mucha más información. Si nos dirigimos al enlace adjuntado en el mensaje, tenemos como resultado lo mostrado en la *Ilustración 54*:

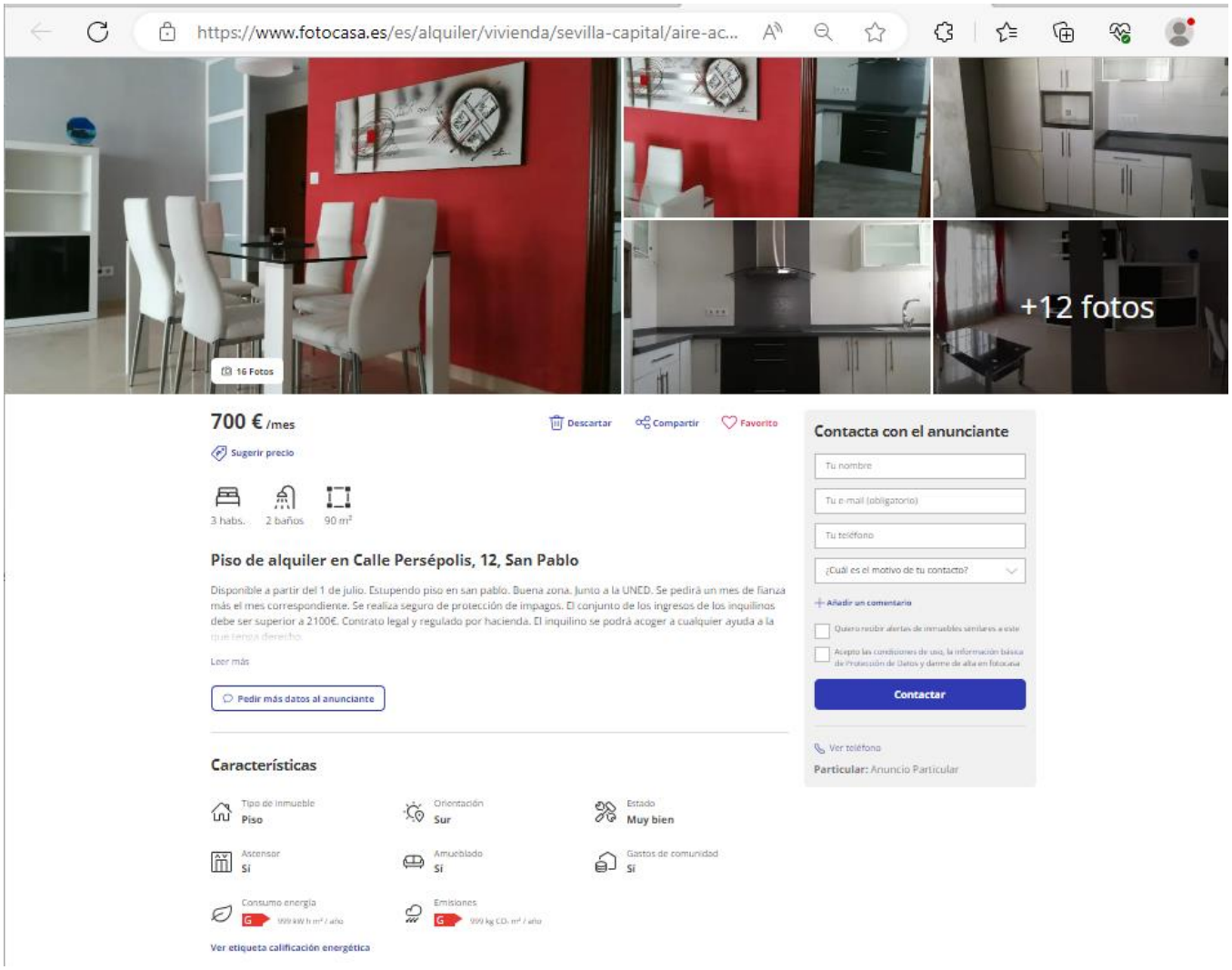



*Ilustración 54: Resultado de peor puntuación según el servicio*

Como podemos observar, aunque el precio de la vivienda es menor que el anterior que consultamos, esta brinda unas prestaciones menores que el anterior, puesto que no consta de terraza ni piscina, por lo que el servicio cumple con lo que se espera en cuanto a la puntuación de los anuncios encontrados. Volviendo al lado del servidor, si consultamos los resultados registrados en la base de datos para esta subscripción, enfocándonos en el apartado de la puntuación, se tiene como resultado lo mostrado en la *Ilustración 55*:

```
rentingscraper=> select link,score from searchresults where userId='6224348276' order by score desc limit 10;
                                                                                                                                                  link                                                                                                                                                  |   score
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------+-----------
 https://www.milanuncios.com/alquiler-de-pisos-en-sevilla-sevilla/sevilla-capital-salamanca-487447447.htm                                                                                                                                             |     10.65
 https://www.milanuncios.com/alquiler-de-pisos-en-sevilla-sevilla/bellavista-la-palmera-bellavista-484280655.htm                                                                                                                                      |  9.926471
 https://www.milanuncios.com/alquiler-de-pisos-en-sevilla-sevilla/sevilla-este-avda-de-las-ciencias-485613667.htm                                                                                                                                     |      8.65
 https://www.milanuncios.com/alquiler-de-pisos-en-sevilla-sevilla/este-alcosa-torreblanca-colores-484784339.htm                                                                                                                                       |       8.6
 https://www.milanuncios.com/alquiler-de-pisos-en-sevilla-sevilla/sevilla-capital-calle-castillo-de-con-483065074.htm                                                                                                                                 |   8.52647
 https://www.milanuncios.com/alquiler-de-pisos-en-sevilla-sevilla/sevilla-este-486624775.htm                                                                                                                                                          |      8.45
 https://www.milanuncios.com/alquiler-de-pisos-en-sevilla-sevilla/sevilla-este-487087193.htm                                                                                                                                                          |  8.228572
 https://www.milanuncios.com/alquiler-de-pisos-en-sevilla-sevilla/amate-rochelambert-484748048.htm                                                                                                                                                    |      8.05
 https://www.milanuncios.com/alquiler-de-pisos-en-sevilla-sevilla/jardines-de-hercules-c-persefone-484664180.htm                                                                                                                                      | 7.9764705
 https://www.fotocasa.es/es/alquiler/vivienda/sevilla-capital/aire-acondicionado-trastero-ascensor-amueblado/178057176/d?from=list                                                                                                                    | 7.8285713
(10 rows)
```

*Ilustración 55: Resultados obtenidos vistos desde la base de datos*

Como vemos el resultado de la consulta se corresponde con los obtenidos en el propio chat. En la imagen, se puede apreciar que el anuncio de mejores características obtuvo una puntuación de 10.65, mientras que el de peores características según el servicio, obtuvo una puntuación de 7.83 aproximadamente.

Retomando la parte cliente, si escribimos los comandos *getCurrentSearchs* y *getDataFromSearchWithId* respectivamente, obtenemos como resultado lo mostrado en la *Ilustración 56*:

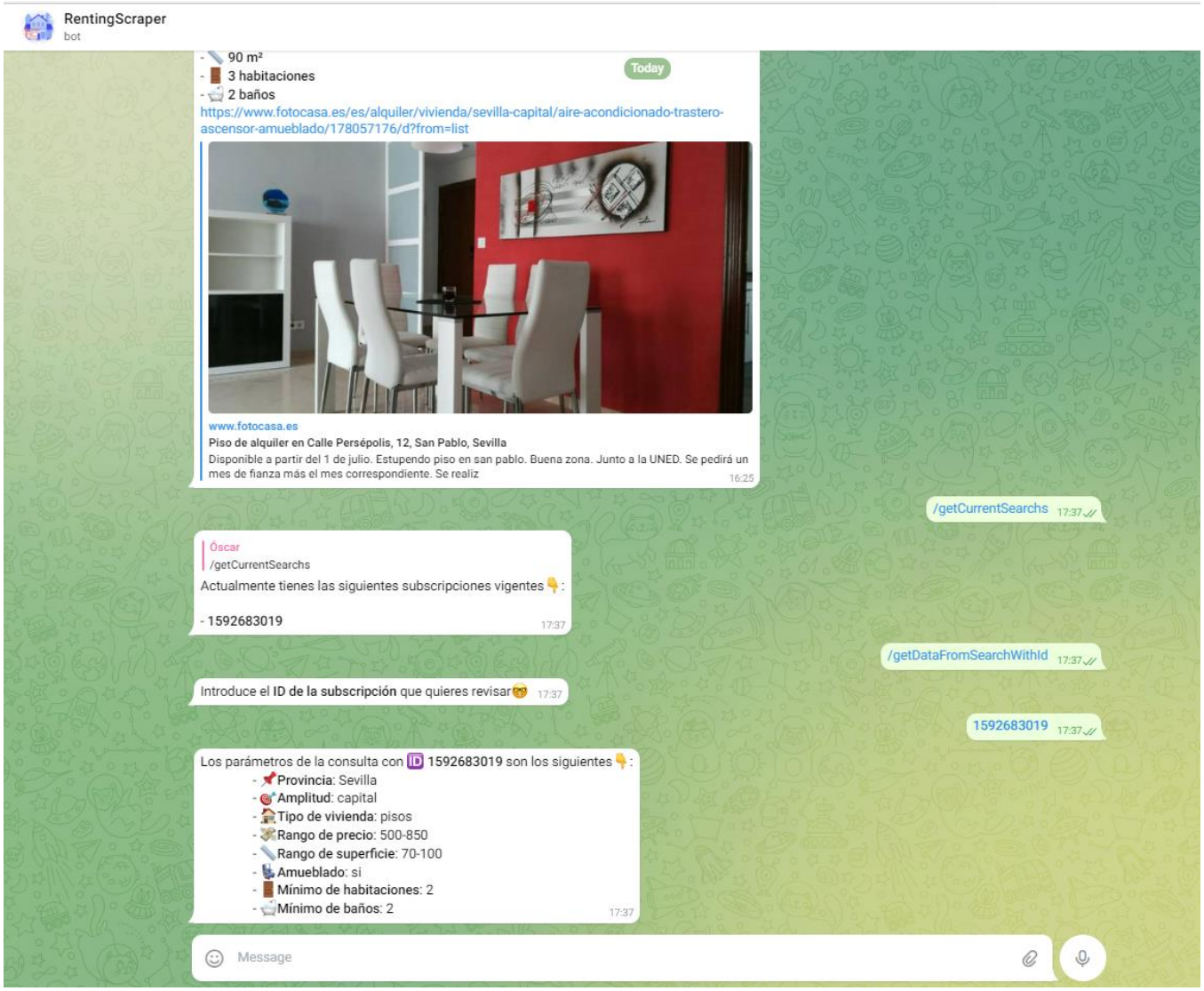



*Ilustración 56: Obtención de los datos de una subscripción registrada*

Como se puede apreciar en la imagen anterior, los datos obtenidos son idénticos a los que fueron introducidos en el momento de la subscripción (*Ilustración 47*).

En cuanto a la actualización de cada una de las subscripciones, esta se realiza periódicamente en el servidor, de forma que si se encontrasen nuevos resultados, estos se compartirían con el usuario. En la *Ilustración 57*, puede observarse la información del *logger* cuando una (o varias) subscripción precisa ser actualizada:

```
Jun 11 17:39:14 localhost rentingScraper.py[1870]: [INFO] Refrescando peticiones pendientes de atender en el bot
Jun 11 17:39:39 localhost rentingScraper.py[1870]: [INFO] Verificando si existen subscripciones pendientes de actualizar
Jun 11 17:39:39 localhost rentingScraper.py[1870]: [INFO] Hay 1 subscripciones pendientes de actualizar
Jun 11 17:39:39 localhost rentingScraper.py[1870]: [INFO] Obteniendo resultados de PISOS para el usuario 6224348276
Jun 11 17:39:39 localhost rentingScraper.py[1870]: [INFO] URL utilizada: https://www.pisos.com/alquiler/pisos-sevilla_capital/amueblado/con-2-habitaciones/con
-2-bano/desde-70-m2/desde-500/hasta-850/fecharecientedesde-desc/
Jun 11 17:39:39 localhost rentingScraper.py[1870]: [INFO] Numero de anuncios encontrados en Pisos para el usuario 6224348276: 0
Jun 11 17:39:41 localhost rentingScraper.py[1870]: [INFO] Obteniendo resultados de FOTOCASA para el usuario 6224348276
Jun 11 17:39:41 localhost rentingScraper.py[1870]: [INFO] URL utilizada: https://www.fotocasa.es/es/alquiler/pisos/sevilla-capital/todas-las-zonas/l?sortType=
publicationDatetodas-las-zonas/amueblado/l?sortType=publicationDate&maxPrice=850&minPrice=500&minBathrooms=2&minRooms=2&maxSurface=100&minSurface=70
Jun 11 17:39:41 localhost rentingScraper.py[1870]: [INFO] Numero de anuncios encontrados en Fotocasa para el usuario 6224348276: 0
Jun 11 17:39:42 localhost rentingScraper.py[1870]: [INFO] Obteniendo resultados de YAENCONTRE para el usuario 6224348276
Jun 11 17:39:42 localhost rentingScraper.py[1870]: [INFO] URL utilizada: https://www.yaencontre.com/alquiler/pisos/sevilla/f-amueblado,500-850euros,70-100m2,2
-bano,2-habitaciones/o-recientes
Jun 11 17:39:42 localhost rentingScraper.py[1870]: [INFO] Numero de anuncios encontrados en Yaencontre para el usuario 6224348276: 0
Jun 11 17:39:50 localhost rentingScraper.py[1870]: [INFO] Obteniendo resultados de MILANUNCIOS para el usuario 6224348276
Jun 11 17:39:50 localhost rentingScraper.py[1870]: [INFO] URL utilizada: https://www.milanuncios.com/alquiler-de-pisos-en-sevilla-sevilla/?demanda=n&orden=dat
e&fromSearch=1&s=amueblado&desde=500&hasta=850&banosd=2&dormd=2&m2d=70&m2h=100
Jun 11 17:39:51 localhost rentingScraper.py[1870]: [INFO] Numero de anuncios encontrados en Milanuncios para el usuario 6224348276: 0
Jun 11 17:39:51 localhost rentingScraper.py[1870]: [INFO] Refrescando peticiones pendientes de atender en el bot
```

*Ilustración 57: Información del logger al actualizar una subscripción*

Como vemos, el *logger* informa de que hay una subscripción pendiente de ser actualizada y por tanto, el servidor procede a la búsqueda de nuevos anuncios en los diferentes portales web usados para la consulta. Como se observa en las trazas de *log* impresas en la terminal, en ninguno de los portales web consultados existen nuevos anuncios, por tanto, no se comparte ningun tipo de información con el usuario a través del chat. En el caso de que se encontrasen nuevos anuncios, el usuario recibiría un mensaje similar al mostrado en la *Ilustración 58*:

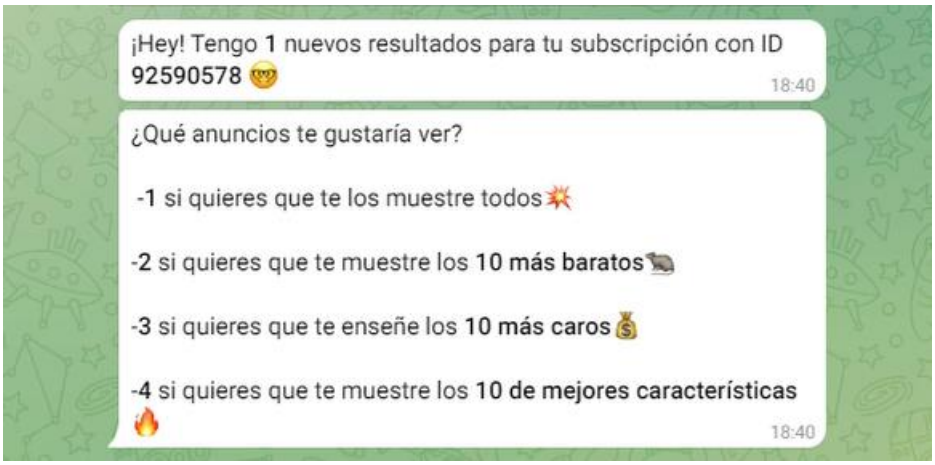



*Ilustración 58: Mensaje de actualización de una subscripción vigente*

Como se puede observar, el *bot* nos mandaría 2 mensajes nuevos en caso de encontrar nuevos resultados para una subscripción nuestra, indicándonos en el primero de ellos el número de nuevos resultados, y el identificador de la subscripción al que pertenecen. En cuanto al segundo mensaje, este es exactamente el mismo que el de la *Ilustración 48*, de modo que podamos seleccionar los anuncios que deseamos observar de esta última búsqueda.

En cuanto a la gestión de errores de la herramienta, esta implementa mensajes de error de carácter informativo, tanto en el *logger* de este como en el chat de *Telegram*, de manera que, si ocurriera algún tipo de error durante el servicio, tanto el administrador del mismo como el usuario, ambos serían notificados a través de estos. Centrándonos en el error más común y probable de suceder, siendo este la introducción de datos por parte del cliente, tenemos que si el servidor detecta que algún dato no fue correctamente dado por parte del usuario, este sería notificado a través del chat como se muestra en la *Ilustración 59*:

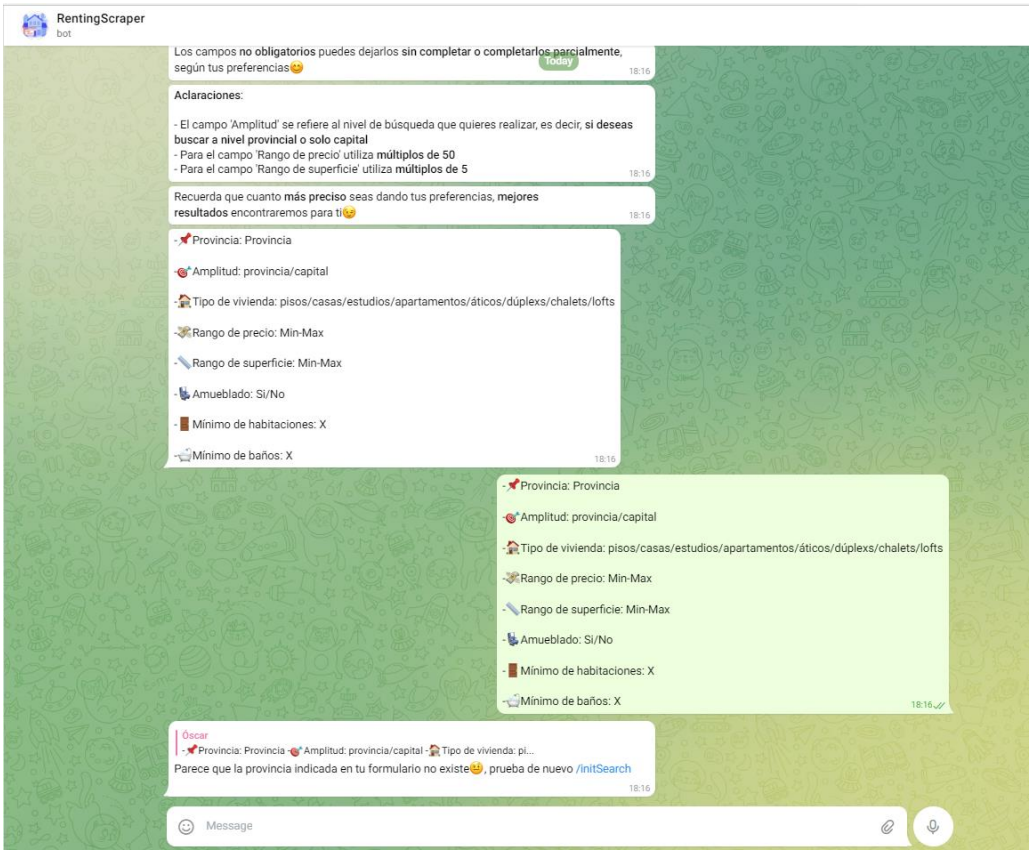



*Ilustración 59: Campo obligatorio del formulario no cumplimentado*

En el ejemplo de la imagen anterior, el usuario ha tratado de realizar una nueva subscripción, pero en este caso no ha modificado los campos del formulario, por lo que la subscripción no se procesa. Por poner otro ejemplo, en la *Ilustración 60* se muestra otro error al cumplimentar el formulario, dando un dato no válido para uno de los campos:

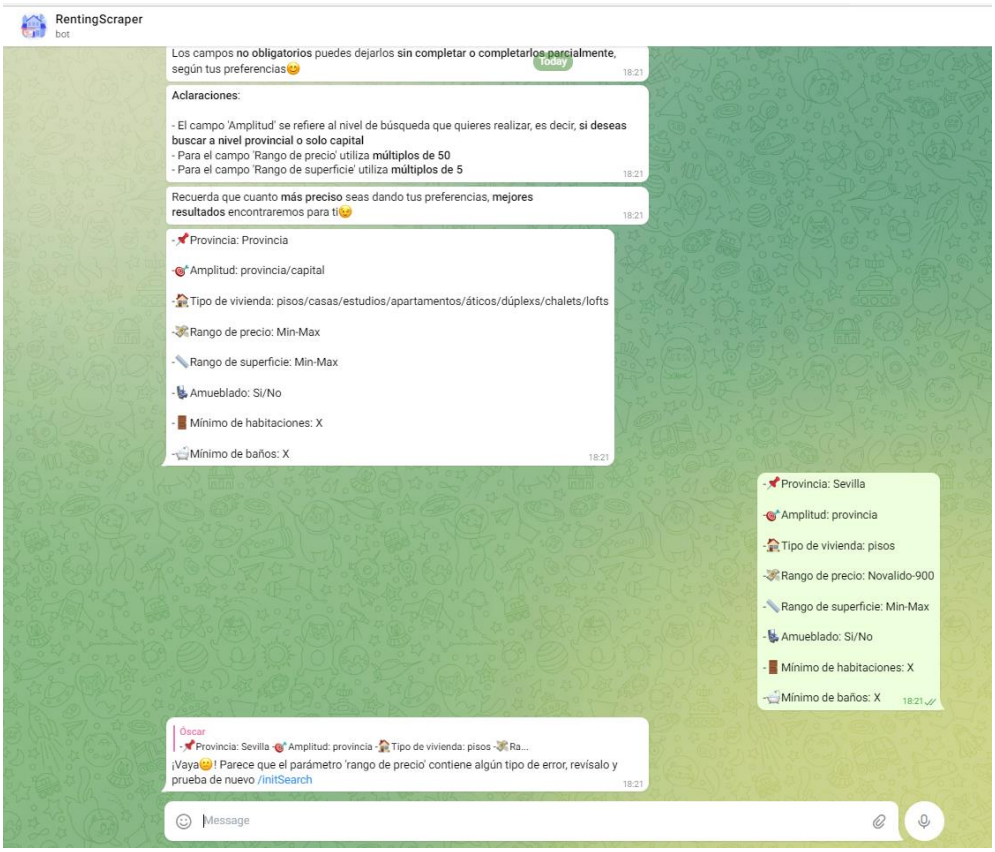



*Ilustración 60: Valor inválido para el campo 'rango de precio'*

Como se observa en la imagen, el propio *bot* nos avisa de que hay un error en el campo 'rango de precio', indicándonos que revisemos este último, ya que no fue correctamente cumplimentado. Por último, se debe añadir que debido al uso de *Telegram* para la parte cliente de la aplicación, esto provoca que toda la gestión de errores que realicemos recaiga sobre el servidor, produciéndose así una carga adicional en el equipo, la cual podría haberse evitado si se hubiera decidido por usar otra alternativa para la parte cliente, como pudiera ser una aplicación móvil o una aplicación web.

Por último, para finalizar con este apartado, si quisiéramos terminar con una subscripción vigente, tendríamos que hacer uso del comando *endSearch*, el cual nos permite eliminar una subscripción del sistema. A continuación, en la Ilustración 61, se muestra el resultado de aplicar dicho comando:


*Ilustración 61: Cancelación de una subscripción*

Como podemos apreciar en la imagen, tras introducir dicho comando, el *bot* nos solicita el identificador de la subscripción, (el cual puede ser consultado mediante el comando *getCurrentSearchs*), y tras indicárselo, este nos pregunta si estamos realmente seguros de eliminar la subscripción o no. Como en este caso estamos seguro de proceder con la eliminación de esta, le indicamos que sí y por tanto la subscripción queda eliminada de la base de datos del sistema.

# 7 Conclusiones

Finalmente, tras haber expuesto de forma ordenada y detallada cada una de las etapas del desarrollo del proyecto, solo queda tomar perspectiva y analizar las distintas características de este, de manera que verifiquemos si verdaderamente se ha logrado el objetivo del proyecto, proveyendo al usuario de una solución al problema planteado en el *capítulo 1* de este documento.

## 7.1 Revisión de cumplimiento de objetivos

Antes de determinar si se ha cumplido correctamente con el objetivo marcado, es preciso rememorar el problema planteado, para proceder a la comparación entre la situación inicial de la cual partíamos, y la actual, de forma que podamos observar las diferencias entre ambas, permitiéndonos así detectar fácilmente si hemos logrado solventar el problema existente.

A lo largo del *capítulo 1*, especialmente en el apartado *Antecedentes*, comentábamos que para la búsqueda de anuncios de viviendas de alquiler el usuario disponía de distintos portales web, tales como *Fotocasa* e Idealista, por ejemplo, los cuales satisfacían las necesidades de los usuarios proveyendo a estos de herramientas como avisos, filtros de búsqueda etc. Aun así, también se mencionaba que estas herramientas hacían de esta tarea un proceso tedioso y monótono, ya que requerían una cantidad de tiempo excesiva y una constancia diaria. Para solucionar este problema, se planteó implementar un servicio software en el cual se delegase el desarrollo de esta tarea, de manera que el usuario se viera con la mínima carga de trabajo posible en cuanto a lo que esta tarea se refiere. Este servicio, como dijimos, se apoyaría en conceptos tales como la automatización y el *Web Scraping*, los cuales nos permitirían llevar a cabo la ejecución de la solución propuesta, además de las distintas herramientas posteriormente mencionadas a lo largo de esta memoria.

Tras rememorar de manera abreviada la situación inicial y la solución que se planteó al inicio de este documento, podemos proceder con la comparación entre el antes y el después de la implementación del servicio. A continuación, se expone en dos párrafos diferentes, la forma de proceder con el desarrollo de la tarea en la situación inicial y la actual:

- **Situación inicial**: anteriormente, si el usuario deseaba buscar una nueva vivienda en concepto de alquiler, este tendría que visitar de forma diaria cada uno de los distintos portales web de vivienda, introduciendo los diferentes filtros de búsqueda en cada uno de ellos cada vez que accediera a la aplicación web de turno. Además, si este quisiera obtener notificaciones de nuevas publicaciones, tendría que registrarse en el portal web de turno (en caso de que este proveyese esta funcionalidad), llenando la bandeja de entrada de su correo personal de mensajes indeseados. Todo ello convierte esta tarea en un proceso tedioso y fatídico.
- **Situación actual**: tras el despliegue del servicio *RentingScraper*, si el usuario lo desease, este podría ahorrarse el suplicio de utilizar las anticuadas herramientas disponibles por parte de los diferentes portales web, haciendo uso del servicio implementado en este proyecto. Para ello, solo tendría que disponer de una cuenta de *Telegram*, pudiendo acceder a ella a través de la versión móvil o web, buscar el *bot RentingScraper*, realizar una nueva subscripción con los parámetros de búsqueda deseados y esperar a recibir los resultados encontrados por parte del *bot*, teniendo solamente que consultar los anuncios que le causen mayor interés. Todo ello sin tener que salir de la aplicación de *Telegram*, e introduciendo una única vez los filtros de búsqueda deseados.

Tras la comparativa anterior, se muestra a continuación una tabla la cual muestra resumidamente las diferencias existentes entre ambas situaciones:

| | **Situación inicial** | **Situación actual** |
|---|---|---|
| **Interacción del usuario** | Alta | Baja |
| **Coste temporal** | Alto | Bajo |
| **Recopilación de la información** | Gestionada por el usuario | Gestionada por el servicio |
| **Consumo de recursos** | Alto | Medio-bajo |

*Tabla 1: Comparativa de situaciones*

Como se puede observar tras toda la información expuesta en este apartado, se ha mejorado notablemente la situación del usuario, descargando su nivel de implicación en el desarrollo de la búsqueda de vivienda en concepto de alquiler, y además, brindando a este con una herramienta a la altura de los tiempos actuales, la cual cumple con las expectativas creadas al inicio de este documento, por lo que finalmente se puede afirmar rotundamente que se ha cumplido con el objetivo planteado en el *capítulo 1* de esta memoria.

## 7.2 Debilidades y fortalezas

Tras verificar en el apartado anterior que la solución implementada cumple debidamente con su cometido, podemos pasar a analizar los puntos fuertes y débiles del servicio, permitiéndonos así detectar los aspectos que se podrían mejorar en el futuro.

Como fortalezas, podemos destacar el ***nivel de automatización del servicio***, el cual podríamos señalar como un nivel alto, ya que como hemos podido comprobar, este elimina en un alto porcentaje el nivel de implicación del usuario en el desarrollo de la tarea. Por otro lado, podríamos añadir la ***sencillez***, ya que para interactuar con el servicio, el usuario únicamente tiene que hacer uso de un chat de una aplicación de mensajería como *Telegram*, la cual se encuentra ampliamente extendida en la sociedad, por lo que salvo pequeñas excepciones, el uso del servicio sería muy intuitivo para el usuario. Otro aspecto que podemos destacar es el ***rendimiento*** del servicio, ya que con tan solo enviar unos pocos mensajes de texto, podemos obtener una gran cantidad de anuncios relacionados con nuestras preferencias en apenas un par de minutos. También podemos destacar que debido al uso de Telegram, ***no precisamos mantener ni desarrollar código de aplicación en el lado del cliente***, ya que el peso de la ejecución recae de forma completa en el servidor. Por último, podemos añadir la ***recopilación de la información***, ya que el usuario tiene la opción de disponer de todos los resultados obtenidos en una simple conversación de *Telegram*.

En cuanto a las debilidades, podemos mencionar la ***fuerte dependencia del servicio en una aplicación externa*** como *Telegram*, ya que en caso de que esta sufra de algún tipo de incidencia que impida su funcionamiento de forma normal, nuestro servicio se encontraría indisponible independientemente del estado de nuestros servidores, viéndonos obligados a esperar que *Telegram* solucionase la incidencia. Por otro lado, podemos destacar la ***descompensación de la carga de trabajo entre el lado cliente y el servidor,*** ya que actualmente la carga de trabajo recae al completo en el lado servidor debido al uso de la aplicación *Telegram*. Con respecto a las funcionalidades del servicio, como hemos podido comprobar estas son ***excesivamente básicas***, ya que actualmente no existe la opción de hacer consultas a la base de datos por parte del cliente, por ejemplo, además de no existir una opción de recuperar los resultados encontrados si el contenido del chat se borra para el usuario, y algunas otras funcionalidades que podrían ser ampliadas o añadidas (*parámetros de búsqueda ofrecidos en el formulario de subscripción*, *mejora del algoritmo de puntuación de anuncios*, *etc*).

Finalmente, como hemos podido observar, a pesar de que el servicio de *RentingScraper* es una gran herramienta, este consta de numerosas debilidades, las cuales podrían ser minimizadas o incluso eliminadas en el futuro a través de distintas mejoras que se podrían llevar a cabo.

## 7.3 Continuación del proyecto

Antes de finalizar con esta memoria, en este apartado se tratará de mostrar la senda de continuación del proyecto que se podría seguir en el futuro, dando al lector una idea sobre el desarrollo de las posibles versiones futuras del servicio y sus funcionalidades.

Como hemos visto a lo largo de la memoria, especialmente en los últimos capítulos, el servicio *RentingScraper* es una gran herramienta la cual podría facilitar al usuario una ayuda enorme en el proceso de búsqueda de una nueva vivienda, pero aun así esta presenta numerosas debilidades las cuales deberían ser mitigadas en el futuro. La mayor de ellas, como se ha mencionado en el apartado anterior, es la fuerte dependencia del servicio en la aplicación *Telegram*, de manera que si surge algún tipo de problema nos dejaría con las manos atadas. Esto podría solventarse mediante la implementación de una aplicación móvil o aplicación web propia, lo que además produciría un alivio de la carga de trabajo del servidor, pudiendo delegar ciertas tareas en el lado del cliente. Aunque esta solución conllevaría un incremento de los costes de mantenimiento del servicio.

Aun con todo lo mencionado, el uso de *Telegram* tiene sus ventajas (mencionadas en la memoria), y podría mantenerse en el futuro como interfaz de usuario alternativa a una posible aplicación móvil/web desarrollada, brindando así al usuario con distintas alternativas de uso, de manera que este pudiera escoger la que mejor se adapte a sus preferencias.

Otro aspecto que se ha mencionado con anterioridad es la escasez de parámetros de búsqueda en el formulario de subscripción, pudiendo ser este más completo añadiendo campos como, por ejemplo, si deseamos que la vivienda conste de garaje o ascensor, o si bien preferimos buscar solamente en cierto barrio de una ciudad, y un largo etc. Otro detalle es el algoritmo utilizado por la aplicación para puntuar los anuncios encontrados. Como vimos en el *capítulo 4*, este es un algoritmo bastante sencillo, basado en simples constantes las cuales hacen que cada una de las características de la vivienda se ajusten a un orden de carácter unitario, sumando puntos en función de su valor, si hablamos del precio o la superficie, o de si están presentes o no determinadas características como que tenga piscina o ascensor. Como decimos, a pesar de que el algoritmo es funcional y cumple con su cometido, este podría ser dotado de una mayor complejidad, haciendo este más preciso. Otro aspecto que cabe destacar y se podría rediseñar en el futuro es el uso de servidores *proxies* para reencaminar las peticiones web a los distintos portales utilizados. Como dijimos anteriormente, esta es la alternativa de la cual hacemos uso para poder realizar las distintas consultas sin que los portales web utilizados para las consultas nos bloqueen el acceso. Si embargo, en el futuro podría negociarse algún tipo de API de consultas con los portales web o buscar alguna otra tecnología que nos permita extraer los datos disminuyendo los recursos utilizados.

## 7.4 Conclusión final

Tras todo el contenido visto a lo largo de la memoria, podemos llegar a la conclusión de que la automatización de tareas es una herramienta muy potente la cual nos permite la delegación de tareas y ser mucho más productivos en la realización de las mismas. Además, desde el punto de vista del autor, se cree que la puesta en práctica de la automatización es más que necesaria en la actualidad, especialmente en la búsqueda de información, ya que a día de hoy, en *Internet* reside una cantidad de datos inmanejable para un ser humano. Herramientas como esta que se ha desarrollado en esta memoria, inteligencias artificiales como *Perplexity*, *ChatGPT*, *CI/CD* en *Gitlab* y otras muchas han venido para facilitar la vida a las personas y hacer que estas seamos mucho más productivas y eficientes.

# Anexo: Instalación del servicio

En esta sección de la memoria del proyecto se detallará el procedimiento a seguir para la instalación del servicio, tanto la instalación manual como la instalación automatizada (a través del script llamado *installDependencies.sh*, mostrado anteriormente en la *Ilustración 35*).

## Instalación manual

Para realizar la instalación manualmente, debemos tener en cuenta las distintas herramientas que precisa el servicio para operar, las cuales son las siguientes:

- Equipo servidor con *Ubuntu 20.04* o superior como sistema operativo.
- Navegador Google Chrome de versión 112 o superior.
- Driver *Chromedriver* compatible con la versión de *Google Chrome instalada*.
- Intérprete de *Python* de versión 3.8.10 o superior.
- Librerías de *Python*: *Requestts-html*, *BeautifulSoup*, *Selenium*, *Psycopg2*, *regex*, *telebot*, *pathlib*.
- Servidor *Postgresql* versión 12 o superior.

Por simplicidad, omitiremos la instalación del sistema operativo Ubuntu en este manual, dando por hecho que ya disponemos del equipo con el software requerido. Bien, con respecto al resto de herramientas software, tendremos que realizar los siguientes pasos:

- *Google Chrome*: para su instalación en el equipo, podremos hacer uso de la página oficial de *Google*, la cual nos provee el software con la última versión disponible para nuestro sistema operativo (este es detectado a través de las cabeceras *HTTP* intercambiadas al acceder al sitio web indicado, por lo que no tenemos que indicarlo nosotros). En el siguiente enlace podemos descargar el navegador https://www.google.com/chrome/.
- *Chromedriver*: en este caso, accederemos a la página oficial del proveedor, https://chromedriver.chromium.org/downloads, y descargaremos la versión que se corresponda con la que hayamos instalado de *Google Chrome*. En la *Ilustración 62* se muestra cómo obtener la versión del navegador instalada en nuestro sistema

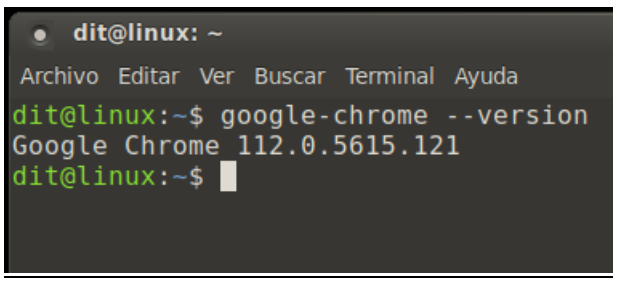



*Ilustración 62: Versión instalada de Google Chrome*

Una vez ha sido correctamente descargado el binario *Chromedriver*, deberemos situar este en la carpeta del sistema */usr/bin*, para que este se encuentre dentro del *PATH* del sistema. Para ello, podemos hacer uso del comando *mv*.

- *Python*: para instalar el intérprete de *Python*, podemos hacer uso de los siguientes comandos, siendo estos ejecutados en consola en el orden mostrado:

  - *sudo apt update*: para actualizar la lista de paquetes.
  - *sudo apt upgrade -y*: para actualizar los paquetes existentes en el sistema.
  - *sudo apt install python3*: para instalar *Python* en nuestro sistema.

  Como podemos observar, necesitaremos estar en la lista de *sudoers*, para poder ejecutar dichos comandos como administrador. En cuanto a las librerías requeridas, lo más apropiado es hacer uso del administrador de paquetes de *Python*, conocido como *PIP*. Este, puede ser instalado a través de los siguientes comandos:

  - *sudo apt update*: para actualizar la lista de paquetes.
  - *sudo apt install python3-pip*: para instalar el administrador de paquetes de *Python*.

  Una vez instalados ambos recursos, podemos verificar su instalación a través de los comandos mostrados en la *Ilustración 63*:

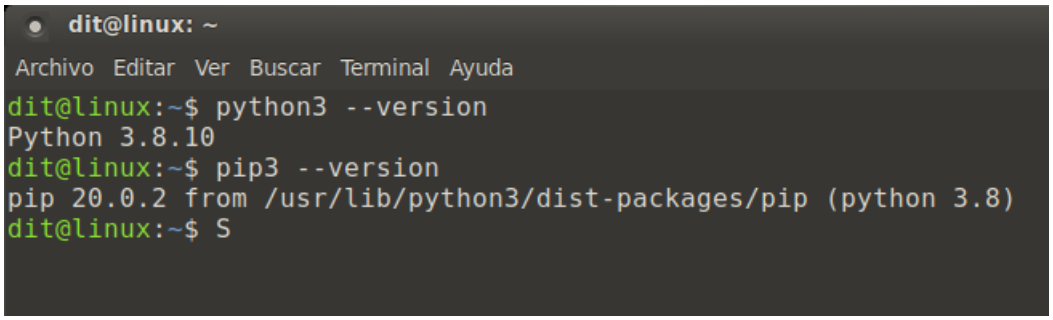



*Ilustración 63: Verificación de la correcta instalación de Python*

En cuanto a la instalación de las distintas librerías de *Python* utilizadas por el servicio. Podemos hacer uso de los comandos mostrados en la *Ilustración 64*:

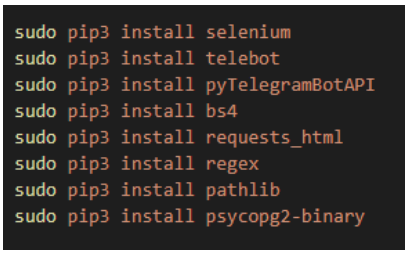



*Ilustración 64: Instalación de librerías de Python*

- Servidor *Postgresql*: al igual que en los casos anteriores, podemos hacer uso del gestor de paquetes del sistema *apt*. Para ello, ejecutando los comandos *sudo apt update* y *sudo apt install postgresql*, podemos instalar el software necesario para hacer uso del servidor de base de datos. Tras ello, podremos ejecutar el *script setUpDatabase.sh* , mencionado en el *capítulo 2*, el cual se encarga de configurar el servidor de forma apropiada para el correcto funcionamiento del servicio *RentingScraper*.

Finalmente, una vez hemos instalado todas las dependencias del servicio, solamente queda copiar el archivo *rentingScraper.service* (el cual podemos encontrar dentro del proyecto) en la ruta */etc/systemd/system*, para que este pueda ser gestionado a través de la herramienta del sistema *Systemd*. Una vez hecho, tendremos que ejecutar el comando *systemctl daemon-reload* para que *systemd* pueda recargar su configuración y tenga en cuenta el nuevo fichero añadido. Una vez hecho, si ejecutamos el comando *systemctl status rentingScraper*, deberíamos obtener el resultado mostrado en la *Ilustración 65*:

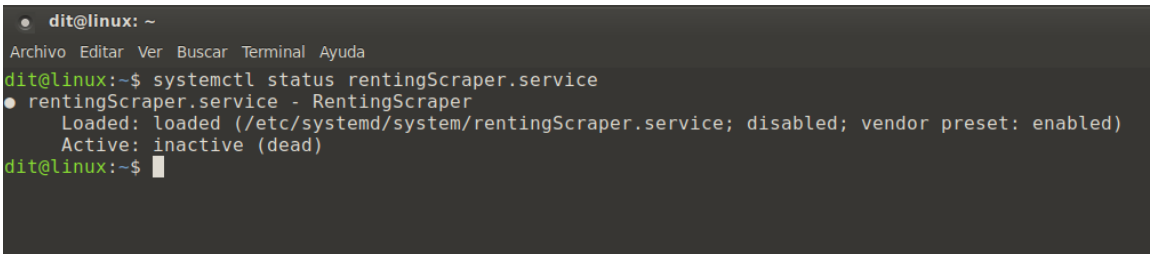



*Ilustración 65: Estado del servicio una vez cargado con Systemd*

Por último, si quisiéramos habilitar el arranque del servicio junto con el arranque del sistema, podríamos hacer uso de los comandos *systemctl enable postgresql* y *systemctl enable rentingScraper*.

## Instalación automatizada

Por razones obvias, el procedimiento seguido en el apartado *Instalación manual* ha sido automatizado en un script de *bash*, de forma que al ejecutarlo se instalen todas las dependencias requeridas del servicio, además de la copia de los archivos necesarios en las rutas indicadas (ficheros *Chromedriver* y *retingScrpaer.service*). Para ello, debemos ejecutar dicho script como usuario administrador del sistema. Dicha acción puede ser llevada a cabo a través del comando *sudo ./installDependencies.sh*, por ejemplo (otra opción sería ejecutar *su – root -c ./installDependencies.sh*). Por último, se expone a continuación en la *Ilustración 66* el contenido de dicho script (aunque ya fue expuesto en la *Ilustración 35*):

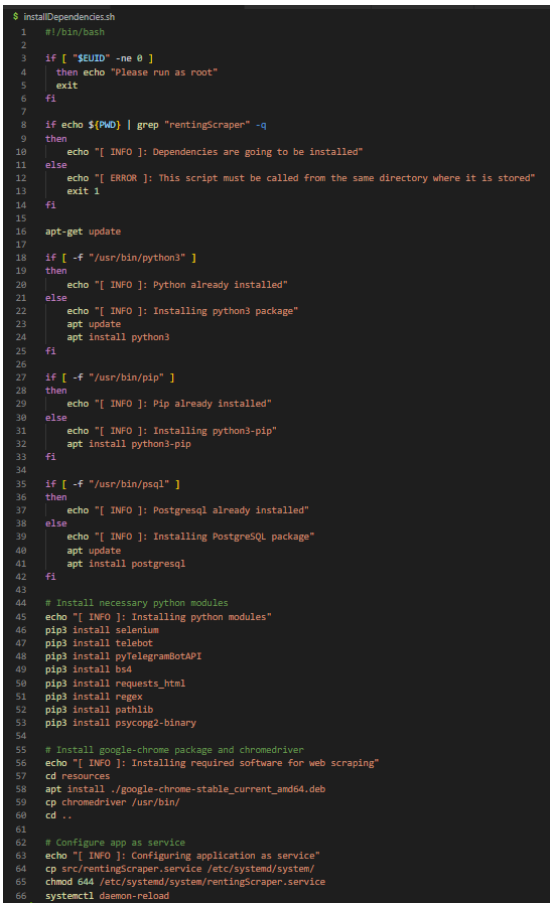



*Ilustración 66: Contenido fichero installDependencies.sh*

# Bibliografía

A continuación, se exponen los recursos utilizados a lo largo de todo el desarrollo de este proyecto, de manera que si el lector lo desea pueda profundizar en algún aspecto el cual haya llamado su atención:


[ 1 ] https://www.mindbowser.com/web-scraping-advantages/

[ 2 ] https://datstrats.com/blog/scraping-es-legal-espana/

[ 3 ] https://www.techspot.com/news/98721-nearly-half-all-internet-traffic-2022-came-bots.html

[ 4 ] https://blog.apify.com/web-scraping-javascript-vs-python-2022/

[ 5 ] https://es.wikipedia.org/wiki/Navegador_sin_interfaz_gr%C3%A1fica

[ 6 ] https://www.python.org/doc/essays/blurb/

[ 7 ] https://www.coursera.org/articles/what-is-python-used-for-a-beginners-guide-to-using-python

[ 8 ] https://pypi.org/project/requests-html/

[ 9 ] https://pypi.org/project/requests/

[ 10 ] https://pypi.org/project/beautifulsoup4/

[ 11 ] https://pypi.org/project/selenium/

[ 12 ] https://pypi.org/project/selenium-stealth/

[ 13 ] https://pypi.org/project/undetected-chromedriver/

[ 14 ] https://pypi.org/project/telebot/

[ 15 ] https://core.telegram.org/bots/api

[ 16 ] https://builtin.com/software-engineering-perspectives/python-symbol

[ 17 ] https://www.freecodecamp.org/news/how-to-create-a-telegram-bot-using-python/

[ 18 ] https://chromedriver.chromium.org/

[ 19 ] https://www.toolsqa.com/selenium-webdriver/selenium-geckodriver/

[ 20 ] https://www.postgresql.org/about/

[ 21 ] https://www.postgresql.org/about/featurematrix/

[ 22 ] https://pypi.org/project/psycopg2/

[ 23 ] https://www.digitalocean.com/community/tutorials/what-is-systemd

[ 24 ] https://www.zenrows.com/blog/selenium-avoid-bot-detection#rotating-http-header-information-and-user-agent

[ 25 ] https://stackoverflow.com/questions/33225947/can-a-website-detect-when-you-are-using-selenium-with-chromedriver