



Universidad Pública de Navarra  
Nafarroako Unibertsitate Publikoa

Departamento de Estadística, Informática y Matemáticas

Trabajo de fin de grado

# **Plataforma de adquisición de datos pluviométricos para la predicción y aviso de inundaciones**

**Autor:**

**Arkaitz Oderiz Garin**

**Supervisor:**

**Unai Pérez-Goya**

Pamplona, 2023



## Contenidos

<b>Lista de Figuras</b>	<b>iii</b>
<b>Lista de Códigos</b>	<b>iv</b>
<b>1 Introducción</b>	<b>1</b>
<b>2 Tecnologías</b>	<b>3</b>
2.1 Lenguaje de Programación Python . . . . .	3
2.1.1 Qué es Python? . . . . .	3
2.1.2 Por qué Python? . . . . .	3
2.2 Sistema Operativo Debian . . . . .	4
2.2.1 Historia . . . . .	4
2.2.2 Filosofía . . . . .	4
2.2.3 Modelo de negocio . . . . .	5
2.2.4 Por qué Debian? . . . . .	5
2.3 Framework . . . . .	7
2.3.1 Qué es un framework? . . . . .	7
2.3.2 Librería vs Framework . . . . .	8
2.3.3 Django . . . . .	8
2.3.4 Scrappy . . . . .	10
2.4 Base de Datos . . . . .	13
2.4.1 Relacional vs No Relacional . . . . .	13
2.4.2 PostGreSQL . . . . .	13
2.4.3 Por qué PostGreSQL? . . . . .	13
<b>3 Datos</b>	<b>15</b>
3.1 Aemet . . . . .	15

3.2	El Agua en Navarra . . . . .	16
3.3	MeteoNavarra . . . . .	18
3.4	Entorno de ejecución . . . . .	19
3.4.1	Preparación de entorno virtual . . . . .	19
3.4.2	Instalacion y configuracion de PostGreSQL . . . . .	19
<b>4</b>	<b>Plataforma</b>	<b>23</b>
4.1	Estructuras Planteadas . . . . .	23
4.2	Flujo de Datos . . . . .	24
4.2.1	Datos obtenidos . . . . .	25
4.2.2	Formato de Datos . . . . .	25
4.2.3	Filtrado de Datos . . . . .	26
4.3	Arquitectura . . . . .	26
4.3.1	Elementos presentes . . . . .	27
<b>5</b>	<b>Código</b>	<b>33</b>
5.1	Creación de Spiders . . . . .	33
5.1.1	Proceso de obtención de datos . . . . .	35
5.1.2	Guardado de datos . . . . .	37
5.1.3	Spider básica . . . . .	38
5.1.4	Método start_requests() . . . . .	38
5.1.5	Eliminar Log . . . . .	39
5.2	Spiders usadas . . . . .	39
5.2.1	Aemet . . . . .	39
5.2.2	Chcantabrico . . . . .	42
5.2.3	MeteoNavarra . . . . .	51
5.2.4	Agua en Navarra . . . . .	56
<b>6</b>	<b>Conclusiones y Trabajo Futuro</b>	<b>65</b>
	<b>Referencias</b>	<b>67</b>
	<b>Glosario</b>	<b>69</b>
	<b>Anexos</b>	<b>69</b>

## Lista de Figuras

2.1 Diagrama patrón MVT . . . . .	8
2.2 Web Scraping (Krotov y Tennyson 2018) . . . . .	11
3.1 Página Aemet de la estación en Aranguren (Navarra) . . . . .	16
3.2 Página principal de aforos de El Agua en Navarra . . . . .	17
3.3 Error al cargar directamente la página de datos numéricos en El Agua en Navarra . . . . .	17
3.4 Apartado selección de datos MeteoNavarra . . . . .	18
4.1 Idea original de la estructura de datos planteada . . . . .	23
4.2 Estructura de datos usada en el proyecto . . . . .	24
4.4 Estructuración básica de una Spider . . . . .	28
4.3 Arquitectura de obtención y tratamiento de datos . . . . .	32
5.1 Estructura del proyecto recién creado . . . . .	33
5.2 Directorio de almacenamiento de las Spider . . . . .	34
5.3 URL de inicio para obtener los códigos de las estaciones de Aemet . . . . .	35
5.4 Inspector de webs de Chrome . . . . .	36
5.5 Ruta chromedriver.exe . . . . .	61



## Lista de Códigos

5.1 Spider recién generada . . . . .	34
5.2 Guardar datos . . . . .	37
5.3 Configurar guardado en JSON . . . . .	37
5.4 Spider de ejemplo . . . . .	38
5.5 Sobre-escritura de start_request() . . . . .	39
5.6 Configurar LOG . . . . .	39
5.7 Aemet Code Spider . . . . .	40
5.8 Aemet Data Spider . . . . .	40
5.9 Chcantabrico Code Spider . . . . .	42
5.10 Chcantabrico Nivel Spider . . . . .	44
5.11 Chcantabrico Pluviometric Spider . . . . .	45
5.12 Chcantabrico Coordinates Spider . . . . .	48
5.13 Script de obtención de datos pluviometricos descartado . . . . .	49
5.14 MeteoNavarra Code Spider . . . . .	51
5.15 MeteoNavarra Data Spider . . . . .	52
5.16 MeteoNavarra Coordinates Spider . . . . .	54
5.17 Agua en Navarra Code Spider . . . . .	56
5.18 Agua en Navarra Data Spider . . . . .	57
5.19 Agua en Navarra configuración Selenium . . . . .	60
5.20 Configuración Playwright . . . . .	62
5.21 Playwright basic Request . . . . .	63
5.22 Agua en Navarra Playwright Request . . . . .	63
A1 Mostrar notas. . . . .	69





## Introducción

Los últimos 15-20 años la Unión Europea ha realizado un gran esfuerzo en promover políticas de datos abiertos en realización a la información generada y monitorizada por los estados miembros. El objetivo de esta política es acercar estos datos a la población para tener un mayor control territorial y medio ambiental. El gobierno de Navarra y de España contribuyen a la oferta de datos abiertos publicando mediante diferentes organismos como el geoportal (<https://geoportal.navarra.es/es/idena>) en Navarra o los datos o el sistema automático de información publicada por las diferentes confederaciones hidrográficas en España.

El objetivo de este trabajo es crear una plataforma habilitada en técnicas de scraping, centralizando la información proveniente de diferentes fuentes para una posible futura predicción y aviso de inundaciones mediante los datos pluviométricos obtenidos de los ríos de Navarra.

La plataforma se dividirá en dos apartados, la obtención de los datos y el almacenamiento de estos. Para ello se harán uso de dos máquinas virtuales comunicándose entre si. Aunque hacer uso de una única máquina no solo es viable si no más sencillo, disponer de ellas, no solo aparta la plataforma de un diseño centralizado mas vulnerable, ademas, a nivel de proyecto, proporciona un mayor grado de complejidad, necesitando configurar la comunicación por red de estas.

La primera máquina sera la encargada del almacenamiento de los datos, proporcionando una base de datos en PostGreSQL. Esta recibirá los datos de la segunda máquina, encargada de la obtención de estos.

Como se ha mencionado, la plataforma busca centralizar los datos proporcionados en las distintas webs, tanto fluviales como pluviométricos, creando un punto de acceso global para su posterior uso. Es por eso que la segunda máquina dispone de una

plataforma automática de obtención de datos mediante scraping web que, junto a una API en Django posibilita el envío de los datos a la base de datos.

Esta plataforma, realizada usando el Framework de Python Scrapy para el apartado de obtención de datos y, Cron para la automatización, ejecuta los scripts de forma regular en intervalos de tiempo predefinidos para cada web, con el fin de no generar más tráfico web del realmente necesario.

En este proyecto se ha pretendido crear una plataforma lo más modular posible, ya sea por el uso de múltiples máquinas virtuales, como por los distintos apartados que la forman e, incluso en la forma de codificarla. Intenta ser lo más intuitiva posible a la hora de poder realizar cambios sobre esta, diferenciando claramente cada elemento.

Centrado en técnicas de scraping web, este proyecto está limitado por los datos ofrecidos de forma abierta en la web, siendo su punto débil, aquel del cual no disponemos control alguno. A día de hoy, la plataforma dispone de cuatro fuentes distintas de las cuales obtener datos, pero está comprometida a que se mantengan invariables en el tiempo. Es por esto, que en un proyecto así necesariamente se debe valorar la búsqueda de datos de forma activa. No solo para su mantenimiento a largo plazo sino como forma de extender su alcance.

## 2.1 Lenguaje de Programación Python

### 2.1.1 Qué es Python?

Siendo su primera aparición en el año 1991 por manos de Guido van Rossum, Python es un lenguaje de programación de alto nivel interpretado que prima la legibilidad del código, siendo este a veces nominado como "seudocódigo ejecutable". [\[1\]](#)

Python a su vez es un lenguaje multiplataforma, fuertemente tipado, dinámico y multiparadigma, pues soporta programación orientada a objetos, imperativa y funcional. [\[2\]](#) [\[3\]](#)

### 2.1.2 Por qué Python?

La elección de Python sobre otros lenguajes se basa en su sintaxis sencilla y clara que facilita la programación, junto a la gran cantidad de librerías y frameworks potentes de los que dispone para satisfacer las necesidades que presenta este proyecto, como pueden ser la extracción de datos meteorológicos mediante web scraping y la creación de una API.

## 2.2 Sistema Operativo Debian

### 2.2.1 Historia

Conforme la computación fue tomando terreno tanto en el ámbito comercial como en el escolar, no tener una forma de instalar y configurar los sistemas de una forma rápida sin tener que partir de cero y sin tener que compilar el software necesario manualmente se convirtió en un problema patente entre los usuarios. [4] [5]

En 1993, tras varios intentos fallidos por distintas empresas de solucionar el problema, Ian Murdock, por aquel entonces estudiante de la Universidad Purdue, encontró la solución al problema basándose en el reciente proyecto de Linus Torvalds, el kernel Linux. tras el anuncio de Ian para crear un sistema operativo de forma descentralizada en paralelo como es el caso del kernel Linux, docenas de usuarios se unieron para formar el proyecto Debian Linux con la intención de crear un sistema operativo de gran calidad y mantenimiento, publicando en enero de 1994 la primera versión de Debian 0.91. [6]

### 2.2.2 Filosofía

Debian no intenta seguir ni competir con los líderes del sector, por el contrario, desde sus inicios el proyecto se ha basado en una filosofía centrada en la robustez y estabilidad del sistema guiada por unos estrictos estándares de calidad, actualizándose conforme las necesidades de sus usuarios a la vez que promueve el software gratuito, lo que le ha ayudado a obtener fama entre los usuarios. [7] [8]

A su vez, debido al apoyo del software libre, hace uso de múltiples licencias de software como la Licencia Pública General GNU (GPL), licencias artísticas o del tipo BSD, lo cual ha llevado al desarrollo de las Directrices de Software Libre de Debian (DFSG) con el fin de definir la construcción del software libre. [9]

Definido por estas licencias, el software libre debe cumplir al menos las que están consideradas la cuatro libertades esenciales: [10]

- Ejecutar el programa como se desee, con cualquier propósito.
- Estudiar cómo funciona el programa, y cambiarlo para que haga lo que se desee.
- Redistribuir copias para ayudar a otros.
- Distribuir copias de sus versiones modificadas a terceros permitiendo ofrecer a toda la comunidad la oportunidad de beneficiarse de las modificaciones.

### **2.2.3 Modelo de negocio**

Todo el modelo del proyecto Debian se basa en su Contrato Social respecto a la comunidad de software libre creado en 1997, sobre el cual se estipulan las directrices a seguir para crear y distribuir software libre. [11]

No hay que confundir el termino libre, definido dentro de DFSG, con gratuito, pues un programa de software puede ser libre aunque sea de pago. Por el contrario, Debian es un proyecto libre y gratuito.

Debian se mantiene gracias a su comunidad, disponiendo más de mil desarrolladores y contribuidores por todo el mundo aportando al proyecto con su tiempo y conocimiento de forma gratuita. Esto, junto a la asistencia de múltiples empresas que dan soporte a Debian dentro del proyecto de socios [12] y el sistema de donaciones [13] usado para hardware, dominios, certificados criptográficos, conferencias, etc han ayudado en el éxito y crecimiento del proyecto dentro de su filosofía.

### **2.2.4 Por qué Debian?**

El mercado está repleto de distintas posibles alternativas a Debian, ya sean de pago, Windows Server OS o Red Hat Enterprise Linux (RHEL), gratuitos, Ubuntu Server y Fedora Server, o incluso en la nube, Amazon Web Services (AWS), Google's Cloud Platform.

Descartando todo sistema de pago, aunque las posibilidades se reducen aun hay múltiples opciones sobre las que elegir, pero son pocas las que ofrecen la misma usabilidad que Debian con sus más de 59000 paquetes en su versión estable o incluso puedes usar las versiones «en pruebas» o «inestable» en caso de querer probar las nuevas funciones antes de su lanzamiento oficial. [14]

Aunque todos los sistemas basados en Linux disponen de las mismas características,

siendo software libre y gratuito con soporte multi-usuario, multi-proceso y uso en tiempo real, Debian siendo uno de los sistemas más longevos del mercado a tomado fama entre la competencia por su seguridad y estabilidad. Siendo la base para muchas de las distribuciones más populares contra las que compite, como Ubuntu, Knoppix, PureOS o Tails.

Cabe mencionar, que parte de esta seguridad y estabilidad puede llegar a ser un limitante a la hora de elegir Debian como sistema operativo dependiendo de tus necesidades a nivel de uso pues el software compatible igual no es la versión más reciente.

Finalmente, una característica distintiva de Debian frente a la competencia gratuita es su compatibilidad con un uso 24/7, pues otras alternativas gratuitas o no dan soporte a esta característica, Fedora Server o, necesitas disponer de una licencia de pago como es el caso de Ubuntu Server mediante Ubuntu Pro.

Una vez contado todo esto, puesto que no me afecta negativamente el no disponer de las versiones más recientes de software y necesito de un sistema 24/7 gratuito, parece lógica la elección de Debian como sistema operativo para usar en el servidor.

## 2.3 Framework

### 2.3.1 Qué es un framework?

Un framework es software que provee una infraestructura básica sobre la que desarrollar tus proyectos, aportando las funcionalidades y estructuras básicas necesarias para este sin la necesidad de programar todo desde cero, ahorrando tiempo de desarrollo y aportando robustez al proyecto. [15]

Cada framework aportará su propia colección de módulos y paquetes específicos para ayudar en el desarrollo, es por esto que generalmente se clasifican en tres clases distintas según funcionalidades. [16]

#### Tipos de frameworks

##### Full Stack

Un framework full-stack es apto tanto para desarrollo back-end como front-end, aportando todas las herramientas posibles que ayuden con el desarrollo gráfico de la interfaz de usuario (UI), gestión de bases de datos, protocolos de seguridad y lógica de negocio entre tantos. Siendo Django un ejemplo de framework full-stack.

##### Micro

Los framework micro son ligeros por definición, siendo en cierta medida lo contrario de un framework full-stack, pues aunque los componentes que aportan como puede ser la gestión de bases de datos son los mismos, estos no vienen incluidos de forma nativa. Esto se debe a que buscan aportar flexibilidad y libertad a los desarrolladores para que incluyan únicamente aquellas herramientas que necesiten.

Como se explica en la documentación de Flask, uno de los framework tipo micro más relevante, el 'micro' de microframework significa que el núcleo del framework es simple pero extensible.

##### Asíncrono

Estos framework están dirigidos por eventos. en vez de hacer un manejo operacional línea a línea de las funciones en la que se van ejecutando una detrás de otra, el código asíncrono no es bloqueante por lo que no se espera que un evento termine para ejecutar el siguiente, ejecutándolo de forma simultánea. Debido a esto un framework asíncrono puede llegar a conseguir un gran rendimiento si se usa en un servidor que lo permita.

### 2.3.2 Librería vs Framework

Aunque ambas ofrecen funcionalidades operacionales, su mayor diferencia radica en la especificidad y complejidad de estas.

Las librerías están compuestas por múltiples métodos para un uso específico sin aportar mucha complejidad, realizando una tarea por función.

Por el contrario, como los framework tienen en cuenta las posibles necesidades de tu proyecto, pudiéndose permitir ser aún más específicos, ofreciendo la arquitectura y comportamiento básico de la aplicación, dejando la flexibilidad de desarrollar las funcionalidades necesarias para su funcionamiento, aportando herramientas sobre las que trabajar. [17]

### 2.3.3 Django

#### Qué es Django?

Django es un framework web de tipo full-stack gratuito y de código abierto para Python. Sigue el principio DRY "Don't Repeat Yourself" por lo que se enfoca en el menor uso de código, el desarrollo rápido y la reutilización de componentes. [18] [19]

Hace uso de su auto denominado patrón Modelo-Vista-Template (MVT) 2.1 una variante del conocido Modelo-Vista-Controlador (MVC). [20]

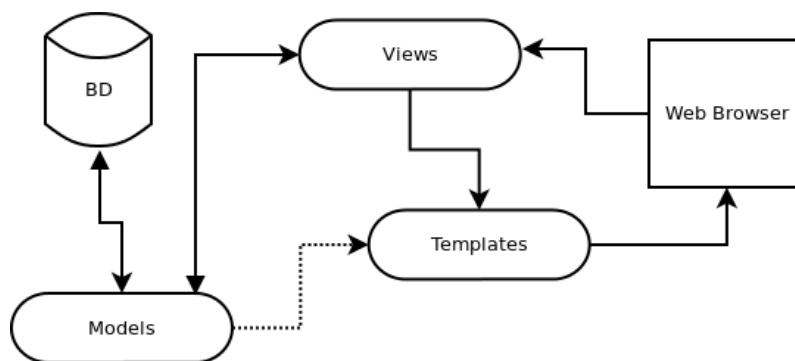


Figura 2.1: Diagrama patrón MVT <sup>1</sup>

Para poder trabajar con bases de datos relacionales tales como, Oracle, MySQL y PostgreSQL, Django usa un Mapeador Relacional de Objetos (ORM) permitiendo interactuar con ellas mediante SQL.

<sup>1</sup><https://docs.hektorprofe.net/django/web-personal/patron-mvt-modelo-vista-template/>



### Por qué Django?

Partiendo de que el lenguaje de programación a usar es Python, Django es uno de los frameworks más reputados dirigidos a este lenguaje, teniendo en cuenta su naturaleza gratuita y de código abierto. Siendo usado tanto por la comunidad como por empresas tales como Instagram, Mozilla y Facebook. Aunque no por ello significa que no disponga de poca competencia, siendo TurboGears, web2py, Bottle y Flask de las más notables.

El que sea un framework full-stack no va a aportar la flexibilidad, libertad y ligereza que da el uso de un microframework, sobre todo a la hora de agregar únicamente aquellas herramientas que considere necesarias, pero si que atenuará la carga de trabajo que puede suponer un microframework si no se dispone de la experiencia necesaria para usarlo.

Otra característica por la que me he decantado por Django, aunque no sea única de él, es su compatibilidad con bases de datos relacionales de forma nativa, cosa que facilitara el uso de estas en el proyecto.

### 2.3.4 Scrapy

#### Qué es el Web Scraping?

Web scraping, también conocido como web extraction o web harvesting, es una técnica de extracción de datos desestructurados de la World Wide web (WWW) y guardarlos de forma estructurada en una base de datos o en un sistema de ficheros en formato XML, JSON o CSV para su posterior recuperación o análisis. Generalmente, los datos web son adquiridos mediante el uso de Hyper-text Transfer Protocol (HTTP) o a través de un navegador web, ya sea de forma manual o automática mediante web crawlers, herramientas diseñadas con este propósito, siendo capaces de convertir páginas web enteras en información bien estructurada. [21] [22]

Debido a la gran cantidad de datos que son generados constantemente en la WWW, web scraping es considerado una forma eficiente y poderosa de amasar big data.

Web scraping puede ser usado en una gran variedad de entornos, como la recolección de comentarios en redes sociales, listado de la propiedad inmobiliaria o en este caso el monitoreo y comparación de los niveles de los ríos y datos pluviométricos.

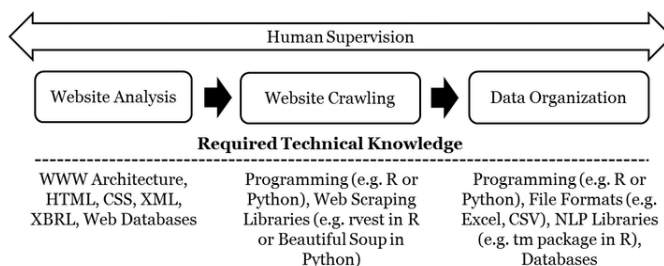
#### Procedimiento básico en Web Scraping

El proceso de recolección de datos de Internet se puede dividir en tres procesos secuenciales, el estudio de la pagina sobre la que trabajar, adquirir los recursos web y, luego, organizar la información deseada. 2.2

El primer paso consiste en mirar la estructuración de la web para seleccionar aquellos datos que queramos extraer, comprobando los recursos HTML, CSS y viendo si hace uso de JavaScript. Para realizar el segundo paso de adquisición de los recursos, el proceso empieza con los programas de web scraping enviando un request, ya sea mediante GET o POST, a la página web deseada. Una vez el request sea recibido y procesado por la web, esta enviará los recursos solicitados al programa. Después, pasaríamos al tercer paso, en el que analizaríamos los recursos obtenidos y los filtraríamos de tal forma que nos quedásemos únicamente con la información que nos sea necesaria. Finalmente almacenaríamos la información obtenida para su posterior análisis.

---

<sup>2</sup>[https://www.researchgate.net/figure/Web-Scraping-Adapted-from-Krotov-and-Tennyson-2018\\_fig1\\_324907302](https://www.researchgate.net/figure/Web-Scraping-Adapted-from-Krotov-and-Tennyson-2018_fig1_324907302)

Figura 2.2: Web Scraping <sup>2</sup>

### Qué es Scrapy?

Scrapy es un framework asíncrono para web crawling y web scraping gratuito y de código abierto. Por defecto proporciona todas las herramientas necesarias para realizar la tarea de extracción, procesamiento y estructuración de los datos adquiridos, además de dar la opción de extender funcionalidades en caso necesario, haciéndolo extremadamente versátil. [23]

Pensado para navegar entre webs y extraer información de forma estructurada de ellas, Scrapy es usado para múltiples propósitos, por ejemplo, el minado de datos o la monitorización y análisis de datos.

### Por qué Scrapy?

A la hora de buscar herramientas para web scraping en Python obtuve tres alternativas principales, BeautifulSoup, Selenium y Scrapy.

La primera es una librería de parseo de HTML y XML, que, aunque podría haberme servido para mi propósito inicialmente, a la larga su sencillez hubiera sido más un problema que una ayuda.

La segunda por el contrario, no es una herramienta para hacer web scraping como tal y se centra en la automatización de la navegación web como entorno de pruebas. Debido a esto, no es una herramienta que haya usado para el web scraping, pero sí que ha sido usada junto con Scrapy para navegar entre webs y sacar los datos de estas.

Scrapy fue elegida gracias a la flexibilidad y versatilidad que proporciona a la hora de trabajar, pudiendo crear proyectos sencillos en cuestión de minutos con las herramientas base proporcionadas o investigar como trabajar con estas herramientas y crear proyectos complejos que satisfagan tus necesidades de la manera que desees.

Esto implica que la curva de aprendizaje de Scrapy sea mayor que la que puede tener BeautifulSoup sobre todo al inicio, llegando a parecer abrumador. A su vez, Scrapy tiene el mejor rendimiento entre las tres. [17]

Finalmente, cabe mencionar que Scrapy no dispone de rotación de IP, renderizado JavaScript, ni geolocalización como pueden tener las alternativas de pago, cosa que no es un impedimento para llevar a cabo el proyecto.

## **2.4 Base de Datos**

### **2.4.1 Relacional vs No Relacional**

### **2.4.2 PostGreSQL**

### **2.4.3 Por qué PostGreSQL?**



Para este proyecto disponemos de los siguientes datos obtenidos de distintas páginas web:

- Nivel (m)
- Caudal ( $m^3/s$ )
- Precipitación (mm)
- Temperatura ( $^{\circ}C$ )
- Humedad (%)
- Radiación ( $W/m^2$ )

A su vez, dispondremos de la fecha y hora en la que se hicieron la lectura de los datos, junto con los códigos y coordenadas de las estaciones sobre las cuales obtenemos los datos.

### 3.1 Aemet

La Agencia Estatal de Meteorología (AEMET) es el Servicio Meteorológico Nacional y Autoridad Meteorológica del Estado, siendo su objetivo es la monitorización y predicción de fenómenos meteorológicos.

AEMET dispone de una API de obtención de datos llamada AEMET OpenData la cual no usaremos pues muchos de los datos que se obtienen de esta manera no nos son útiles.

Como se ve en la imagen [3.1](#), dentro de la web podemos encontrar de forma accesible

múltiples datos relacionados con la meteorología tomados cada hora, de los cuales seleccionaremos únicamente temperatura, precipitación y humedad, puesto que los datos relacionados con el viento o la presión atmosférica no son tan relevantes para este proyecto.

Actualizado: martes, 09 mayo 2023 a las 16:22 hora oficial  
 Ind. climatológico: 9263X - Altitud (m): 572  
 Latitud: 42° 46' 34" N - Longitud: 1° 31' 57" O - Posición: Ver localización  
 Municipio: Aranguren (Navarra) - Ver predicción

Exportar a excel Exportar a csv

Fecha y hora oficial	Temp. (°C)	V. vien. (km/h)	Dir. viento	Racha (km/h)	Dir. racha	Prec. (mm)	Presión (hPa)	Tend. (hPa)	Humedad (%)
09/05/2023 16:00	14.6	5	→	18	↘	0.0			88.0
09/05/2023 15:00	14.6	8	→	15	↘	0.4			89.0
09/05/2023 14:00	14.0	4	→	18	→	0.2			88.0
09/05/2023 13:00	13.7	9	→	26	↘	0.0			85.0
09/05/2023 12:00	13.7	6	↗	11	↑	0.0			73.0
09/05/2023 11:00	13.9	3	↑	6	↗	0.0			69.0
09/05/2023 10:00	14.0	0	-	4	→	0.0			66.0
09/05/2023 09:00	13.3	0	-	3	↙	0.0			69.0
09/05/2023 08:00	10.5	0	-	10	↙	0.0			80.0
09/05/2023 07:00	8.8	5	↙	9	↙	0.0			76.0
09/05/2023 06:00	8.7	7	↙	9	↙	0.0			77.0
09/05/2023 05:00	9.0	3	←	8	↙	0.0			85.0
09/05/2023 04:00	9.1	4	↙	8	↙	0.0			84.0
09/05/2023 03:00	9.3	5	↙	9	↙	0.0			83.0
09/05/2023 02:00	10.2	2	↓	9	↙	0.0			78.0
09/05/2023 01:00	10.5	3	↙	10	↙	0.0			81.0
09/05/2023 00:00	11.2	5	↙	8	↙	0.0			84.0
08/05/2023 23:00	12.2	4	↙	10	↙	0.0			81.0
08/05/2023 22:00	14.0	0	-	9	↖	0.0			73.0
08/05/2023 21:00	16.8	4	↓	10	↘	0.0			60.0
08/05/2023 20:00	19.2	6	↘	23	↘	0.0			48.0
08/05/2023 19:00	20.4	12	↘	27	→	0.0			40.0
08/05/2023 18:00	22.1	16	→	31	→	0.0			36.0
08/05/2023 17:00	22.1	16	→	34	→	0.0			37.0

Líneas por página 50 Página 1 de 1

Figura 3.1: Página Aemet

## 3.2 El Agua en Navarra

En esta página encontraremos los datos tanto del nivel de los ríos como de su caudal en los últimos 15 días en periodos de 10 minutos, siendo una de las fuentes principales de datos.

La sección de aforos en la página 3.2 muestra un mapa de Navarra y esta estructurada en 6 regiones, Norte, Arga, Ega, Ebro alto, Ebro bajo y Aragón, accediendo a cada región nos aparece el mapa de la región dándonos acceso a todas las estaciones en la zona.

Una vez dentro de la estación tendremos la posibilidad de observar los datos, primeramente en forma de gráfica y, en caso de quererlo, en forma numérica, pero no sin antes haber visitado la versión gráfica 3.3.



Debido a este problema, para la obtención de los datos primero cargamos la web gráfica y le pasamos un script JavaScript para pulsar el botón que finalmente nos da acceso a los datos deseados.

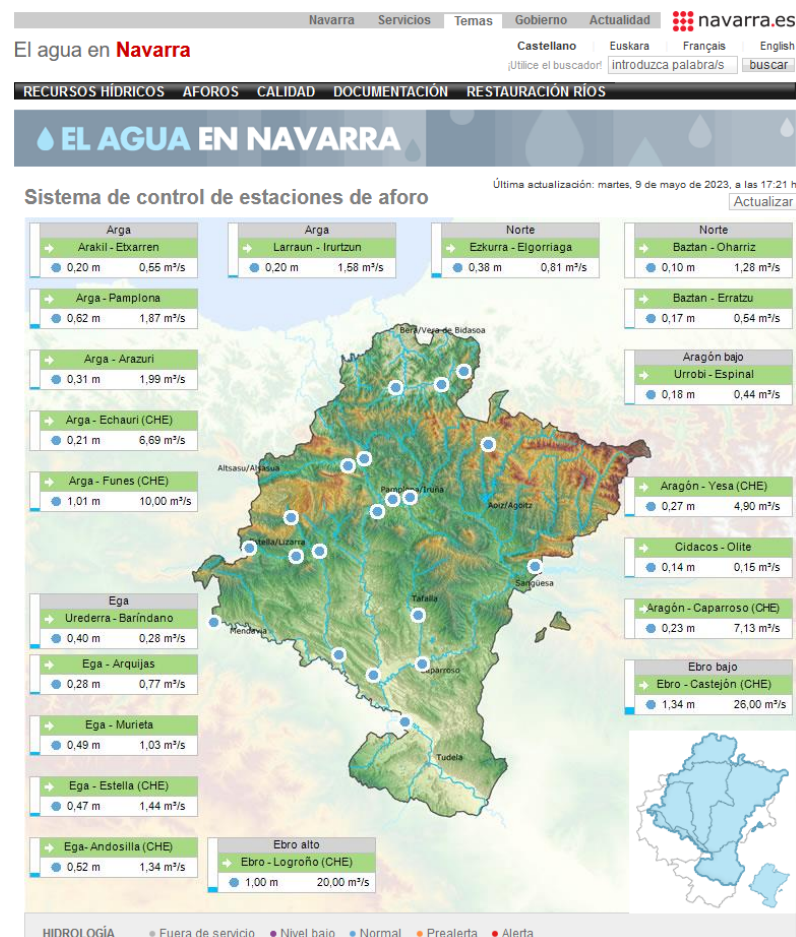


Figura 3.2: Página El Agua en Navarra



Figura 3.3: Error datos numéricos en El Agua en Navarra

### 3.3 MeteoNavarra

En la página de meteorología y climatología de Navarra encontramos una gran sección de estaciones de las cuales poder obtener datos.

De entre los dos tipos de estaciones disponibles, automática y manual, únicamente tomamos los datos de las estaciones automáticas, esto se debe a que las manuales solo proveen datos diarios de la temperatura máxima, mínima y de la precipitación acumulada. Esto hace que no dispongamos de ningún dato hasta que finalice el día, cosa que no es viable si lo que se propone es predecir cambios radicales en un periodo de tiempo reducido.

Las estaciones automáticas proporcionan tanto datos en periodos de diez minutos como datos diarios, siguiendo el mismo razonamiento que con las estaciones manuales no tomamos los datos diarios y, entre los actualizados cada diez minutos tomamos la temperatura, humedad relativa, radiación global y precipitación en un periodo de dos días.

#### 1. Parámetros

##### Parámetros 10 minutos

- ☒ Temperatura
- ☒ Humedad relativa
- ☒ Radiación global
- ☐ Insolación
- ☒ Precipitación
- ☐ Velocidad viento 10 m
- ☐ Dirección viento 10 m
- ☐ Velocidad racha máxima 10 m
- ☐ Dirección racha máxima 10 m
- ☐ Velocidad viento 2 m
- ☐ Dirección viento 2 m
- ☐ Velocidad racha máxima 2 m
- ☐ Dirección racha máxima 2 m
- ☐ Desviación dirección viento 10 m
- ☐ Desviación dirección viento 2 m
- ☐ Desviación velocidad 10 m
- ☐ Desviación velocidad 2 m
- ☐ Humectación hoja (resistencia)
- ☐ Radiación Solar PAR acumulada 10min
- ☐ Temperatura suelo

Datos en **horario solar**.

#### 2. Fechas

hoy      ayer

Desde

Hasta (excluido)

##### Parámetros Diarios

- ☐ Temperatura media
- ☐ Temperatura máxima
- ☐ Temperatura mínima
- ☐ Humedad relativa med.
- ☐ Humedad relativa máx.
- ☐ Humedad relativa mín.
- ☐ Precipitación acumulada
- ☐ Radiación global
- ☐ Insolación total
- ☐ Velocidad media viento 10 m
- ☐ Dirección viento 10 m (MODA)
- ☐ Velocidad racha máx 10 m
- ☐ Dirección racha máx 10 m
- ☐ Velocidad media viento 2 m
- ☐ Dirección viento 2 m (MODA)
- ☐ Velocidad racha máx 2 m
- ☐ Dirección racha máx 2 m
- ☐ Radiación PAR diaria acumulada
- ☐ Temperatura media suelo

Figura 3.4: Datos MeteoNavarra

## 3.4 Entorno de ejecución

El proyecto se dividirá en dos ordenadores (maquinas virtuales). Uno de ellos será el responsable de la base de datos en PostgreSQL, mientras que, el otro, se encargara de la ejecución del código de obtención de datos, tratarlos y enviarlos a la base de datos.

### 3.4.1 Preparación de entorno virtual

Primero instalaremos las dependencias para crear y activar un entorno virtual sobre el que trabajar, para ello ejecutaremos los siguientes comandos:

```
#Instalamos las dependencias
user@host:~$ sudo apt install python3-venv python3-dev

#Creamos un nuevo directorio sobre el que trabajar
user@host:~$ mkdir ~/dirdemiproyecto
user@host:~$ cd ~/dirdemiproyecto

#Creamos el entorno virtual
user@host:~/dirdemiproyecto$ python3 -m venv envdemiproyecto

#Activamos el entorno virtual
user@host:~/dirdemiproyecto$ source envdemiproyecto/bin/activate

#Una vez seguidos los pasos nuestra terminal debe mostrarse así
(envdemiproyecto) user@host:~/dirdemiproyecto$
```

Cada Spider dispondrá de su propio entorno virtual, nombrado tras la pagina web a la que representa, de esta forma, por ejemplo, el entorno virtual para la web de chcantabrico también se llamará chcantabrico. Otro entorno virtual sera usado para el servidor Django y los Scripts encargados del envío de datos.

### 3.4.2 Instalacion y configuracion de PostgreSQL

Comenzamos con los comandos necesarios para la instalación inicial de PostgreSQL y con la creación de un usuario y una base de datos.

```
#Instalamos PostgreSQL y las dependencias
user@host:~$ sudo apt install libpq-dev postgresql postgresql-contrib

#Iniciamos sesion usando el role postgres y accedemos a PostgreSQL
user@host:~$ sudo -u postgres psql
```

```
#Si todo esta bien la terminal deberia mostrarse así
postgres=#

#Creamos un nuevo usuario
postgres=# CREATE USER miusuario WITH PASSWORD 'micontraseña';

#Configuramos varios parametros del usuario para una mejor integracion con Django
postgres=# ALTER ROLE miusuario SET client_encoding TO 'utf8';
postgres=# ALTER ROLE miusuario SET default_transaction_isolation TO 'read committed';
postgres=# ALTER ROLE miusuario SET timezone TO 'Europe/Madrid';

#Creamos la Base de Datos
postgres=# CREATE DATABASE bbddmiproyecto;

#Otorgamos permisos de administrador a nuestro usuario en la base de datos
postgres=# GRANT ALL PRIVILEGES ON DATABASE bbddmiproyecto TO miusuario;

#Una vez finalizado
postgres=# \q
```

En este punto ya dispondríamos de una base de datos (aun sin tablas en ella) y de un usuario con el que conectarnos a esta desde Django.

Ahora quedaría configurar el permiso de comunicación entre dispositivos, más concretamente la conexión remota de Django con PostGreSQL.

```
#Abrimos el archivo postgresql.conf con un editor
sudo nano /etc/postgresql/11/main/postgresql.conf

#Buscamos la linea "#listen_addresses = 'localhost'", borramos la almohadilla y
sustituimos localhost por *, permitiendo la escucha de cualquier direccion IP
listen_addresses = '*'

#Guardamos y cerramos el fichero

#Abrimos el archivo pg_hba.conf con un editor
sudo nano /etc/postgresql/11/main/pg_hba.conf

#Por defecto solo permite conexiones desde localhost
# IPv4 local connections:
```

```
host    all                all                127.0.0.1/32      md5
```

#Configuraremos el permiso de conexión remota desde cualquier IP añadiendo la siguiente línea debajo de la anterior

```
host    all                all                0.0.0.0/0         md5
```

#0 podemos configurar el permiso únicamente para la IP de la otra máquina virtual

```
host    all                all                127.18.83.198/19  md5
```

#Guardamos y cerramos el fichero

#Finalmente permitimos el tráfico mediante el puerto 5432 (puerto por defecto)

```
sudo ufw allow 5432/tcp
```

Una vez realizados los siguientes pasos ya tendríamos nuestra instancia de PostgreSQL configurada.



## 4.1 Estructuras Planteadas

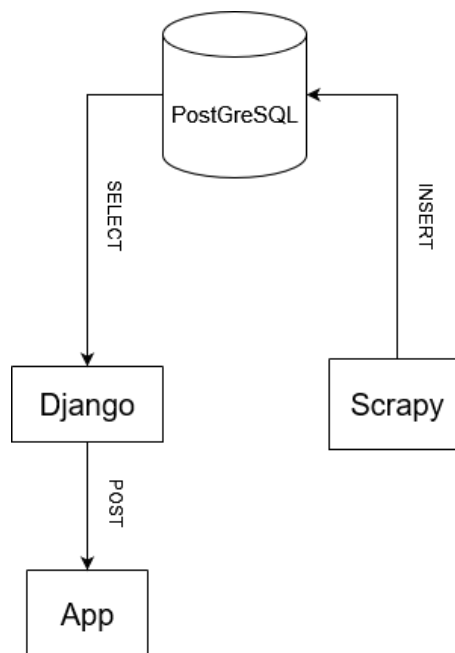


Figura 4.1: Estructura descartada

Problema: aunque sea la estructura más simple y fácil de implementar, pues scrapy mismo puede insertar los datos scrapeados de las web de forma directa en PostGreSQL, Django hace uso de un sistema de gestión de versiones de los cambios realizados en la BBDD con el fin de reducir la carga de peticiones a la BBDD, esto se aplica desde el lado de Django, lo que supondría un problema a la hora de insertar datos directamente de Scrapy a PostGreSQL, pues los nuevos datos podrían no llegar

a ser detectados por Django, generando la situación de que una vez se vayan a pedir los nuevos datos Django no devuelva nada pues desde su punto de vista los datos que ya dispongo son la versión mas reciente, luego no es necesario realizar llamada alguna a la BBDD.

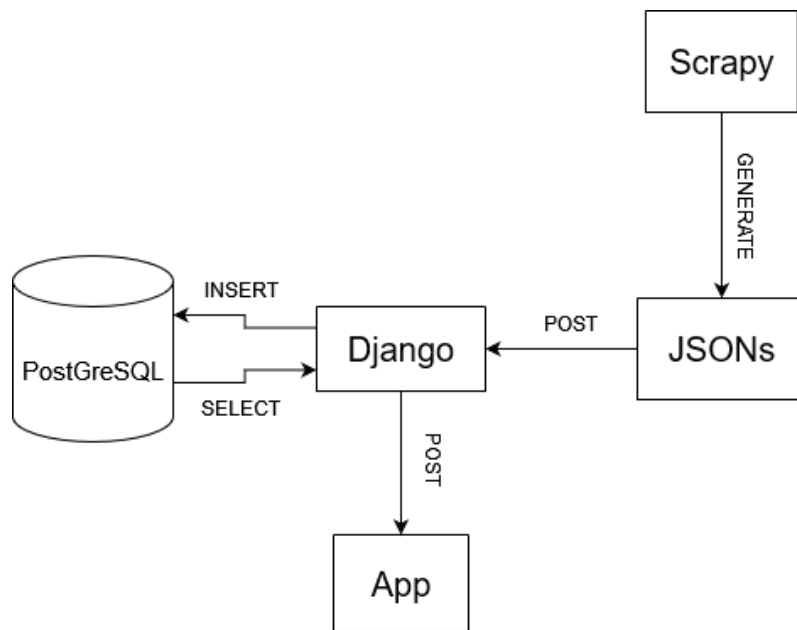


Figura 4.2: Estructura usada

Solución: Pasa solventar el problema hemos incluido un paso intermedio en el que inicialmente almacenamos los datos obtenidos de las distintas webs en formato JSON, para posteriormente enviárselos a Django, el cual se encargara de subir los datos a la BBDD, mientras, el resto de la arquitectura seria idéntico a la versión original.

## 4.2 Flujo de Datos

- 1.- Pedir datos (JSON1)
- 2.- Formatear datos en un nuevo JSON (JSON1 -> JSON2)
- 3.- Mandar datos a la BBDD (JSON2)
- 4.- Marcar JSON como old (JSON2 -> JSON3)
- 5.- Borrar JSON original (JSON1)

While(true)

- 1.- Pedir datos (JSON1)



- 2.- Formatear datos en un nuevo JSON (JSON1 -> JSON2)
- 6.- Crear JSON con las nuevas fechas (JSON2 != JSON3 y fecha de hoy -> JSON4)
- 7.- Mandar datos a la BBDD (JSON4)
- 4.- Marcar JSON como old (JSON2 -> JSON3)
- 5.- Borrar JSON original (JSON1)
- 8.- Borrar JSON viejo (JSON3)
- 9.- Borrar JSON diferencias (JSON4)

Puesto que los datos obtenidos no son los mismos para cada web, antes de ser enviados a Django deben ser parseados para compartir una estructura heterogénea, una vez obtenido el JSON parseado, en caso de ser la primera iteración se mandaran directamente los datos a la BBDD, de no ser la primera iteración, se tomaran unicamente los nuevos datos obtenidos como datos a enviar, una vez enviados los datos el JSON parseado sera marcado como old (viejo) para ser el punto de comparación respecto a los datos mas recientes que obtendremos de las webs.

#### 4.2.1 Datos obtenidos

Aemet:

temperatura, humedad, precipitación

meteoNavarra:

temperatura, humedad, precipitación, radiación

aguaEnNavarra:

nivel, caudal

chcantabrico:

nivel, precipitación, seguimiento, alerta, pre-alerta

En todas las webs se proporciona las coordenadas junto con la fecha y hora en la que se ha hecho la medida.

#### 4.2.2 Formato de Datos

No todas las webs presentan sus datos de la misma manera, es por eso que nos encontramos con que los datos que hemos obtenido pueden llegar a estar repartidos en distintas páginas, haciendo necesario el uso de múltiples spiders, resultando en multiples JSON.

Es por eso que para cada web se ha creado una función para parsear (formatear)

los datos obtenidos, de esta manera disponemos de una estructura única para los datos recibidos, haciendo su uso posterior más fácil, ya sea a la hora de tratarlos posteriormente como para almacenarlos en la base de datos.

Esquema obtenido:

```
1 [
2 {
3   "coordenadas": "X. 598270,3 | Y. 4659333 | Z. 37928",
4   "estacion": "64",
5   "datos": [
6     {
7       "fecha y hora": "01/06/2023 11:20:00",
8       "temperatura (C)": null,
9       "humedad (%)": null,
10      "precipitacion (mm)": null,
11      "nivel (m)": "0,05",
12      "caudal (m^3/s)": null,
13      "radiacion (W/m^2)": null
14    }
15  ]
16 }
17 ]
```

### 4.2.3 Filtrado de Datos

Una vez formateados los datos con el fin de reducir la carga a la base de datos, estos son filtrados mediante la comparación con los ficheros anteriormente marcados como old, de esta forma nos aseguramos de mandar a la base de datos unicamente las instancias nuevas de los datos recogidos, pues no disponemos de ninguna forma para filtrar los datos a la hora de obtenerlos. Estos datos serán guardados en un tercer JSON.

## 4.3 Arquitectura

Para poder realizar el trabajo se ha diseñado la siguiente arquitectura.[4.3](#)

### 4.3.1 Elementos presentes

#### Entornos virtuales

Con el fin de que la plataforma sea lo mas fácilmente ampliable se ha decidido que cada Spider disponga de su propio entorno virtual, esto permite añadir dependencias de tal forma que no afecten a el resto de los Scripts presentes, ayudando en la encapsulación de dependencias.

El mayor inconveniente de una plataforma así es la redundancia de código, al ser múltiples entornos independientes es necesario que mucho código sea repetido en cada uno de ellos, cosa que si fuera un único entorno en el que se ejecutara todo, con una clase sobre la que heredar y o una interfaz, seria mucho el código que nos ahorraríamos.

Actualmente la plataforma dispone de cuatro entornos virtuales para cada Spider y un entorno virtual sobre el que ejecutar el servidor de Django.

#### Spiders

Cada Spider representa una web, de tal forma que cada una de ellas obtiene los datos de la web sobre la que se ha diseñado exclusivamente, para poder realizar esta tarea por cada web han sido necesarias varias Spider, aunque de forma simplificada se pueden agrupar por, aquellas que obtienen las estaciones junto con sus códigos y, las de obtención de datos.

De esta forma dividimos las tareas dándonos la posibilidad de ejecutar aquella que mejor nos venga en cada momento.

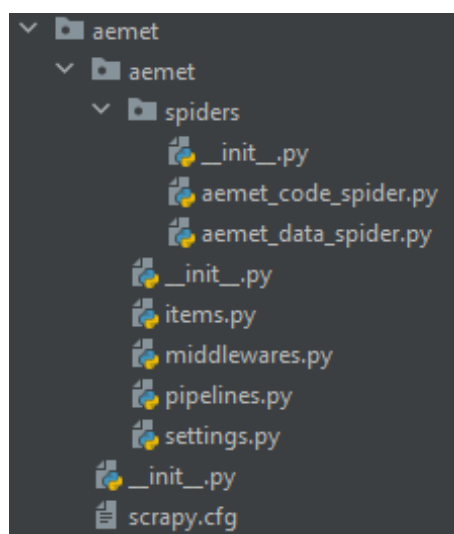


Figura 4.4: Estructuración básica de una Spider

## Runners

Runner es la forma en la que han sido nombrados los Scripts cuya función es posibilitar la ejecución en nuestro caso asíncrona de una o múltiples Spiders mediante un único comando.

Es un Script simple en el que una vez dispones de la estructura básica en caso de necesitar añadir o eliminar una Spider solo tienes que agregar o eliminar la Spider en cuestión y ya estaría listo.

A su vez, al añadir un intermediario el comando de ejecución pasa de ser, scrapy crawl nombreSpider por cada Spider que se desea ejecutar, a, python nombreRunner.py facilitando la automatización de ejecución de las Spider.

## Executers

Aumentado un poco más la abstracción nos encontramos con los Executers, Scripts en Bash encargados de activar el entorno virtual de la Spider deseada y acto seguido ejecutar su respectivo Runner.

Estos existen (en mayor medida que los Runners) con el fin de ayudar con el mantenimiento de la arquitectura, creando un nuevo intermediario en la cadena de ejecución.

Están diseñados de tal forma que funcionen pasándoles un único argumento representando la web de la que quieres obtener los datos, haciendo que la agregación

de nuevas Spiders junto con sus entornos sea tan sencillo como respetar las rutas y nombres predefinidos.

### **JSONs**

Este apartado es un conjunto de directorios en los que se van almacenando los JSON obtenidos tras los distintos procesos a los que son sometidos.

Los directorios en cuestión, siguiendo orden de creación para los JSON de datos son los siguientes:

1. RawData
2. ParsedData
3. RefinedData
4. OldData

Y los de códigos (estación):

1. RawCode
2. ParsedCode
3. RefinedCode (TODO)
4. OldCode (TODO)

El primer nivel es aquel que se obtiene de la llamada con la Spider a la web, devolviendo todos los datos de esta.

El segundo, es obtenido tras ejecutar ya sea `formatear_data_JSONs.py` en caso de querer parsear los datos o `formatear_code_JSONs.py` para los códigos.

El tercero se obtiene tras eliminar la duplicidad de datos en comparación con los ya almacenados en la base de datos mediante la ejecución de `filtrar_data_JSONs.py`.

Finalmente el cuarto es el subproducto obtenido tras la comparación, tomando el fichero original ya formateado y cambiándole de directorio y el nombre de tal forma que se distinga del resto de ficheros.

## Posts

Estos Scripts son los encargados de enviar los datos mediante un Post Request al servidor Django.

Diseñados bajo el mismo principio de fomentar la ampliabilidad del proyecto que los Executors, reciben el nombre de la web de la cual deseas enviar los datos a la hora de ejecutarlo junto al comando en forma de argumento.

## Cron

Cron es un administrador regular de procesos en segundo plano presente en los sistemas basados en Unix. Con él es posible programar la automatización de ejecución de procesos en intervalos de tiempo, pudiendo indicar el minuto, hora, día, mes e incluso día de la semana.

La especificación de los procesos se realiza en el archivo crontab y, su estructuración es la siguiente:

```
.----- minuto (0-59)
| .----- hora (0-23)
| | .----- día del mes (1-31)
| | | .----- mes (1-12) o meses en inglés
| | | | .--- día de la semana (0-6) (domingo=0 o 7) o días en inglés
| | | | |
* * * * * comando a ejecutar
```

En nuestro caso queremos el equivalente a dos instancias de Cron, una que se encargue de obtener los datos ya sea cada quince minutos, media hora y una hora manteniendo la base de datos actualizada para poder realizar las posteriores predicciones y otra que mensualmente compruebe la existencia de nuevas estaciones o el cese del uso de alguna de las ya disponibles.

Así pues, Cron es el encargado de ejecutar cada proceso necesario dentro de la plataforma, ya sea, ejecutar las Spiders, los Scripts de formateo como de filtrado y, el envío de los datos a Django para su inserción en PostgreSQL.

De esta forma dispondríamos de un sistema cerrado automático, el cual nos permitiría trabajar en otros apartados como puede ser la integración de mas webs en la plataforma o la mejora del sistema de predicción.

### **Django**

La instancia del servidor de Django es la encargada de tanto recibir los datos obtenidos como de enviarlos a la base de datos. Finalmente, una vez se dispusiera de la aplicación de predicción se encargaría de pedir los datos almacenados y de enviarlos mediante POST a la aplicación.

### **PostgreSQL**

La base de datos, ejecutada en una maquina virtual independiente, dispone de las siguientes dos tablas:

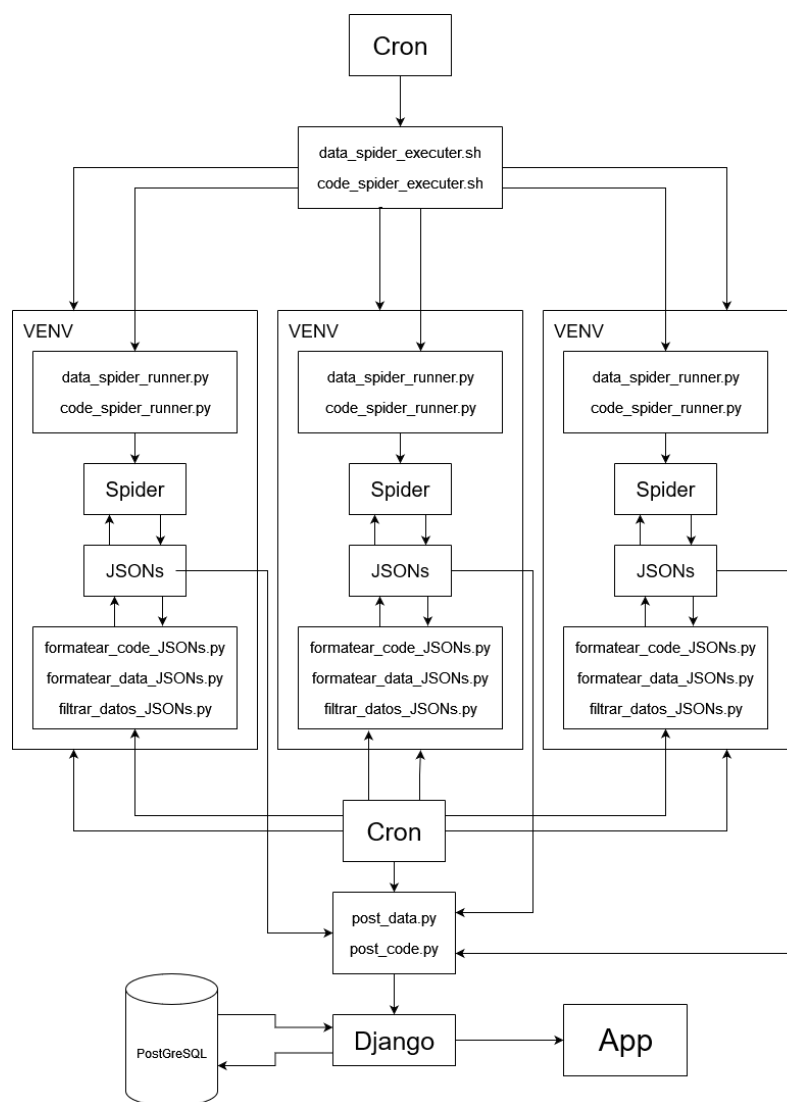


Figura 4.3: Arquitectura de obtención y tratamiento de datos



## 5.1 Creación de Spiders

Una vez instalado Scrapy, en nuestro directorio escogido escribimos el siguiente comando para generar un nuevo proyecto de Scrapy.

```
scrapy startproject miproyecto
```

Nos creara un nuevo directorio con el siguiente contenido.

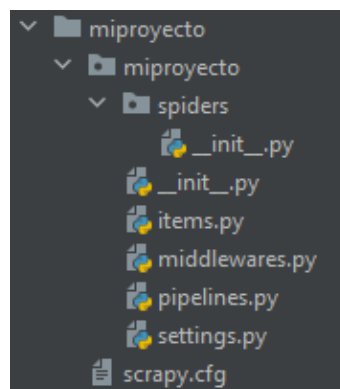


Figura 5.1: Estructura del proyecto recién creado

Primero entraremos en el directorio recientemente creado y luego ejecutaremos el comando encargado de crear la Spider.

```
cd miproyecto
scrapy genspider mispider webausar.com
```

En caso de no especificar el protocolo usado por la web Scrapy asumirá que usa HTTPS.

Tras ejecutar el comando la Spider habrá sido generada dentro de la carpeta spiders.

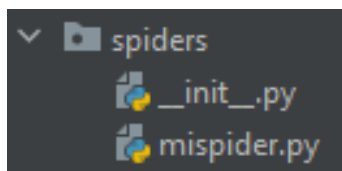


Figura 5.2: Directorio de almacenamiento de las Spider

Una vez abierto el archivo vemos que dispone del siguiente código.

```
1  import scrapy
2
3
4  class MispiderSpider(scrapy.Spider):
5      name = "mispider"
6      allowed_domains = ["webausar.com"]
7      start_urls = ["https://webausar.com"]
8
9      def parse(self, response):
10         pass
```

Código 5.1: Spider recién generada

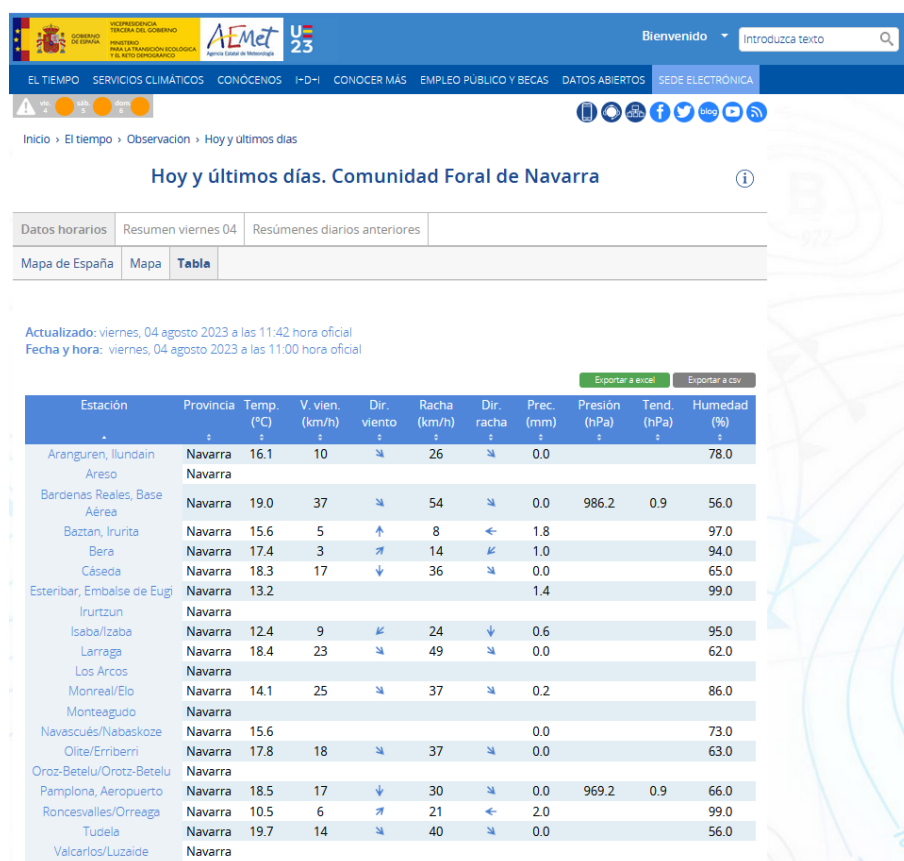
Como podemos ver Scrapy usa una programación orientada a objetos, siendo cada Spider una clase representada dentro del proyecto.

Analizando las variables definidas vemos las siguientes, name, nombre por el que debemos referenciar la Spider a la hora de ejecutarla; allowed\_domains, indica que dominios podemos visitar, negando la entrada a cualquier dominio que no este definido en ella, es importante no especificar protocolo, de esta manera funcionara para cualquier web ya sea HTTP como HTTPS que pertenezca a ese dominio, de lo contrario se limitara al protocolo indicado; start\_urls, URL inicial sobre la que se hará la request de petición de datos.

El método parse es aquel al que se envía la respuesta obtenida de la web, para realizar el filtrado de la información, quedándote unicamente con la deseada. Este método es invocado automáticamente por la Spider, sin necesidad real de hacerlo tu manualmente. Puede ser un método recursivo en caso de así quererlo o incluso se pueden definir nuevas funciones parse (usando un nombre distinto) en caso de necesitarlas.

### 5.1.1 Proceso de obtención de datos

Para poder realizar es la extracción de los datos, primero debemos ir a la web deseada e inspeccionar su estructuración. Para ello como ejemplo vamos a usar la web de aemet.



Actualizado: viernes, 04 agosto 2023 a las 11:42 hora oficial  
Fecha y hora: viernes, 04 agosto 2023 a las 11:00 hora oficial

Estación	Provincia	Temp. (°C)	V. vien. (km/h)	Dir. viento	Racha (km/h)	Dir. racha	Prec. (mm)	Presión (hPa)	Tend. (hPa)	Humedad (%)
Aranguren, Ilundain	Navarra	16.1	10	↘	26	↘	0.0			78.0
Areso	Navarra									
Bardenas Reales, Base Aérea	Navarra	19.0	37	↘	54	↘	0.0	986.2	0.9	56.0
Baztan, Irurita	Navarra	15.6	5	↗	8	↗	1.8			97.0
Bera	Navarra	17.4	3	↗	14	↗	1.0			94.0
Cáceda	Navarra	18.3	17	↘	36	↘	0.0			65.0
Esteribar, Embalse de Eugi	Navarra	13.2					1.4			99.0
Irurtzun	Navarra									
Isaba/Izaba	Navarra	12.4	9	↘	24	↘	0.6			95.0
Larraga	Navarra	18.4	23	↘	49	↘	0.0			62.0
Los Arcos	Navarra									
Monreal/Elo	Navarra	14.1	25	↘	37	↘	0.2			86.0
Monteagudo	Navarra									
Navascués/Nabaskoze	Navarra	15.6					0.0			73.0
Olite/Erriberri	Navarra	17.8	18	↘	37	↘	0.0			63.0
Oroz-Betelu/Orotz-Betelu	Navarra									
Pamplona, Aeropuerto	Navarra	18.5	17	↘	30	↘	0.0	969.2	0.9	66.0
Roncesvalles/Orreaga	Navarra	10.5	6	↗	21	↗	2.0			99.0
Tudela	Navarra	19.7	14	↘	40	↘	0.0			56.0
Valcarlos/Luzaide	Navarra									

Figura 5.3: Web de Aemet para obtención de datos

Una vez encontrada la web deseada, accediendo mediante el F12 a la herramienta de inspección, buscamos el elemento representativo del dato deseado. En nuestro caso queremos obtener tanto el nombre como el código de la estación. Ambos se encuentran en el mismo elemento que forma la primera columna de la tabla.



Figura 5.4: Inspector de webs

Como en este caso es posible filtrar fácilmente los datos, los obtendremos todos directamente, aunque lo más común sería obtener las filas primero para luego iterar por cada una de ellas. Para obtenerlos podemos hacerlo mediante el selector de XPath como con el de CSS.

Ambos se pueden obtener fácilmente en la herramienta de inspección, una vez seleccionado el elemento deseado, hacemos click derecho sobre el, vamos al apartado copiar y en el nos mostrara la posibilidad de copiar ambos selectores. Es probable que el selector proporcionado no sea del todo lo que busquemos o se pueda simplificar, por lo que es recomendable comprobarlo manualmente.

```
rows = response.xpath('//div[@id="contenedor_tabla"]/table/tbody/tr/td/a')
ó
rows = response.css("div#contenedor_tabla tbody tr a")
```

Esto nos devuelve una lista de objetos tipo Selector, cosa que nos permite conforme vamos iterando por cada elemento volver a usar un selector para filtrar unicamente los datos deseados. En nuestro caso.

```
path = rows[i].xpath("@href").get()
name = rows[i].xpath("./text()").get()
ó
path = rows[i].css("*:attr(href)").get()
name = rows[i].css("*:text").get()
```

De esta forma, mediante el uso de la función `get()`, pasamos de tener un objeto Selector a un String. El uso de `get()` sobre una lista devuelve el primer elemento, en caso de querer transformar toda la lista el método a usar es `getall()`.

Finalmente, como de la URL obtenida,

```
'/es/eltiempo/observacion/ultimosdatos?k=nav&l=9263X&w=0&datos=det&f=precipitacion'
```

solo nos interesa el código de la estación (parámetro l de la query), lo filtraremos.

```
code = path.split('&')[1].split('=')[1]
```

### 5.1.2 Guardado de datos

Scrapy almacena todos los datos en forma de múltiples diccionarios, tantos como webs usadas. Para acceder a esta información Scrapy nos proporciona dos alternativas, el uso de Items junto a ItemLoaders, siendo clases específicas de Scrapy o, mediante la palabra reservada yield de Python, que tiene una funcionalidad parecida a return, siendo esta la opción elegida debido a su fácil implementación.

De esta forma escribiremos.

```
1 yield {
2     'estacion': name,
3     'codigo': path.split('&')[1].split('=')[1],
4 }
```

Código 5.2: Guardar datos

Actualmente si se ejecuta la Spider nos imprimiría los datos obtenidos por pantalla, aunque pueden ser almacenados en un fichero tanto CSV como JSON, a la hora de ejecutar la Spider añadiendo en el comando "-o nombre.csv ó -o nombre.json".

Para un uso ligero de forma manual esa alternativa es más que suficiente, pero en nuestro caso, al querer ejecutarlas de forma automática mediante el uso de Runners, debemos implementar una variable llamada custom\_settings para cada una de las Spider. Esta permite, sin la necesidad de modificar el archivo settings.py, añadir configuraciones o dependencias independientes en las Spiders.

```
1 custom_settings = {
2     'FEEDS': {
3         'JSONs/RawCode/codigos_aemet.json': {
4             'format': 'json',
5             'encoding': 'utf-8',
6             'overwrite': True,
7         }
8     }
9 }
```

Código 5.3: Configurar guardado en JSON

Con esto indicamos que, en la ruta especificada, nos almacene un fichero JSON utf-8 y, que cada vez que se llame a esta Spider sobre-escriba el fichero anterior.

### 5.1.3 Spider básica

Una vez obtenemos los datos y los podemos almacenar, ya estaría nuestra Spider básica terminada.

```
1  import scrapy
2
3
4  class AemetCodeSpider(scrapy.Spider):
5      name = "aemet_code"
6      allowed_domains = ["www.aemet.es"]
7      start_urls = ["https://www.aemet.es/es/eltiempo/observacion/"
8                  "ultimosdatos?k=nav&w=0&datos=det&x=h24&f=precipitacion"]
9      custom_settings = {
10         'FEEDS': {
11             'JSONs/RawCode/codigos_aemet.json': {
12                 'format': 'json',
13                 'encoding': 'utf-8',
14                 'overwrite': True,
15             }
16         }
17     }
18
19     def parse(self, response):
20         rows = response.css("div#contenedor_tabla tbody tr a")
21
22         for row in rows:
23             path = row.xpath("@href").get()
24             name = row.xpath("./text()").get()
25             code = path.split('&')[1].split('=')[1]
26
27             yield {
28                 'estacion': name,
29                 'codigo': code,
30             }
```

Código 5.4: Spider de ejemplo

### 5.1.4 Método start\_requests()

El método `start_requests()` es llamado de forma automática al iniciar la Spider, siendo el encargado de hacer la llamada a la web indicada en `start_urls` y, una vez obtenidos

los datos llamar a la función parse, todo mediante un objeto Request de Scrapy, el cual devolverá un objeto tipo HTMLResponse. En caso de querer alterar el funcionamiento de la Spider este es el método a sobre-escribir.

Como en nuestro caso queremos obtener los datos de todas las estaciones dentro de un mismo dominio, reescribiremos la función para que recorra el JSON con los códigos de estas y, hacer una llamada por estación con Request.

El código quedaría de la siguiente manera.

```
1 def start_requests(self):
2     with open("JSONs/RawCode/codigos_aemet.json", encoding="utf-8") as
3         f:
4         data = json.load(f)
5     for estacion in data:
6         url = f'https://www.aemet.es/es/eltiempo/observacion'
7             '/ultimosdatos?k=nav&l={estacion["codigo"]}&w=0&'
8             'datos=det&x=&f=temperatura'
9         yield scrapy.Request(url, self.parse)
```

Código 5.5: Sobre-escritura de start\_request()

Al definir la función de esta manera no es necesario declarar la variable start\_urls, por lo que siempre que necesitemos sobre-escribir la función, no usaremos la variable.

### 5.1.5 Eliminar Log

Cuando se verifique el correcto funcionamiento de la Spider es recomendable quitar el máximo número de Log por pantalla posible, es por eso que, en el fichero settings.py escribiremos las siguientes líneas.

```
1 LOG_LEVEL = 'WARNING'
2 LOG_ENABLED = False
```

Código 5.6: Configurar LOG

## 5.2 Spiders usadas

Como se ha dicho anteriormente, para este proyecto se han creado cuatro proyectos de Scrapy, uno por cada web.

### 5.2.1 Aemet

```
1 import scrapy
2
3
4 class AemetCodeSpider(scrapy.Spider):
5     name = "aemet_code_spider"
6     allowed_domains = ["www.aemet.es"]
7     start_urls = ["https://www.aemet.es/es/eltiempo/observacion/"
8                  "ultimosdatos?k=nav&w=0&datos=det&x=h24&f=temperatura"]
9     custom_settings = {
10         'FEEDS': {
11             'JSONs/RawCode/codigos_aemet.json': {
12                 'format': 'json',
13                 'encoding': 'utf-8',
14                 'overwrite': True,
15             }
16         }
17     }
18
19     def parse(self, response):
20         rows = response.css("div#contenedor_tabla tbody tr a")
21
22         for row in rows:
23             path = row.xpath("@href").get()
24             name = row.xpath("./text()").get()
25             yield {
26                 'estacion': name,
27                 'codigo': path.split('&')[1].split('=')[1],
28             }
```

Código 5.7: Aemet Code Spider

Siendo esta la Spider usada como ejemplo no hay mucho más que comentar al respecto, se dedica a obtener los nombres y códigos de cada estación.

```
1 import json
2
3 import scrapy
4
5
6 class AemetDataSpider(scrapy.Spider):
7     name = "aemet_data_spider"
8     allowed_domains = ["www.aemet.es"]
```



```
9     custom_settings = {
10         'FEEDS': {
11             'JSONs/RawData/datos_aemet.json': {
12                 'format': 'json',
13                 'encoding': 'utf-8',
14                 'overwrite': True,
15             }
16         }
17     }
18
19     def start_requests(self):
20         with open("JSONs/RawCode/codigos_aemet.json", encoding="utf-8")
21             as f:
22             data = json.load(f)
23             for estacion in data:
24                 url = f'https://www.aemet.es/es/eltiempo/observacion/'
25                     'ultimosdatos?k=nav&l={estacion["codigo"]}&w=0&'
26                     'datos=det&x=&f=temperatura'
27                 yield scrapy.Request(url, self.parse)
28
29     def parse(self, response):
30         latitud = response.css('abbr.latitude::text').get()
31         longitud = response.css('abbr.longitude::text').get()
32         estacion = response.css("a.separador_pestanhas").get()
33         rows = response.css('tbody tr')
34
35         datos = []
36         for row in rows:
37             dato = {
38                 'fecha y hora': row.xpath('./td[1]/text()').get() + ':00',
39                 'temperatura (C)': row.xpath('./td[2]/text()').get(),
40                 'humedad (%)': row.xpath('./td[10]/text()').get(),
41                 'precipitacion (mm)': row.xpath('./td[7]/text()').get(),
42             }
43
44             if dato['precipitacion (mm)'] != " ":
45                 datos.append(dato)
46
47         yield {
48             'coordenadas': latitud + ' | ' + longitud,
49             'estacion': estacion.split('=')[3].split('&')[0],
```

```
49         'datos': datos,  
50     }
```

Código 5.8: Aemet Data Spider

Como vemos, la Spider de obtención de datos ya es un poco más compleja, necesitando sobre-escribir la función de `start_request()`. También se puede observar como inicialmente todos los datos son almacenados en una lista antes de ser guardados, el motivo de esto es muy simple y, es que de no hacerlo solo recibiría el primer dato de entre todos los recogidos.

A su vez dentro del bucle `for` se ha añadido un `if` para asegurarse de no guardar datos vacíos, pues puede darse el caso en el que la web aun no tenga el dato de precipitación a cierta hora pero si muestre esta franja horaria pues dispone de otros datos como pueden ser aquellos relacionados con el viento.

Por otro lado, como las coordenadas son mostradas en la misma página que los datos, en vez de crear otra Spider específicamente para obtenerlos, se hace uso de esta con el fin de minimizar código y tiempo de ejecución.

### 5.2.2 Chcantabrico

```
1  import scrapy  
2  
3  
4  class ChcantabricoCodeSpider(scrapy.Spider):  
5      name = "chcantabrico_code_spider"  
6      allowed_domains = ["www.chcantabrico.es"]  
7      start_urls = ["https://www.chcantabrico.es/nivel-de-los-rios"]  
8      custom_settings = {  
9          'FEEDS': {  
10             'JSONs/RawCode/codigos_chcantabrico.json': {  
11                 'format': 'json',  
12                 'encoding': 'utf-8',  
13                 'overwrite': True,  
14             }  
15         }  
16     }  
17  
18     def parse(self, response):  
19         rows = response.xpath('//table[@class="tablefixedheader"  
                                "niveles"]/tbody/tr')
```

```
20
21     for row in rows:
22         codigoBusqueda = row.css('td.codigo::text').get()
23         limites = row.css('table.umbrales_gr td.datos::text').getall()
24         paths = row.xpath('./td/a/@href').getall()
25         estaciones = row.xpath('./td/a/text()').getall()
26
27         for i in range(len(limites)):
28             if limites[i] == 'No definido':
29                 limites[i] = None
30
31         yield {
32             'estacion': estaciones[-3],
33             'codigo': paths[-1].split("=")[-1],
34             'codigoSecundario': codigoBusqueda,
35             'seguimiento': limites[0],
36             'prealerta': limites[1],
37             'alerta': limites[2],
38         }
```

Código 5.9: Chcantabrico Code Spider

Esta puede ser la Spider de obtención de códigos más compleja. esto se debe en parte por la estructuración de la web, mostrando una tabla con otra tabla integrada por cada una de las filas, como por la cantidad de información que dispone.

El primer problema fue como obtener las filas de la tabla principal, pues en muchos de los intentos realizados no solo tomaba estas filas, si no que tomaba aquellas que formaban parte de las tablas incluidas es ellas también. Finalmente, aunque con el Selector CSS no lo logré, mediante Xpath fue posible filtrarlas.

Otra cosa a mencionar es que Chcantabrico proporciona dos códigos por estación, uno referenciando a la estación en si, siendo el usado para posteriormente obtener las coordenadas, aquel que llamo codigoSecundario o codigoBusqueda mientras que, el segundo hace referencia a los datos como tal. Puesto que no todas las filas disponen de la misma cantidad de enlaces, se obtienen todos sabiendo que siempre el ultimo de ellos dispone del código deseado.

Finalmente, dentro de la tabla secundaria, esta es la única web que llega a proporcionar los valores de seguimiento, pre-alerta y alerta del río, siendo estos una buena base para empezar con las predicciones de inundación. En caso de no estar definido el valor simplemente se indicará como None.

```
1 import io
2 import json
3
4 import pandas as pd
5 import scrapy
6
7
8 class ChcantabricoNivelSpider(scrapy.Spider):
9     name = "chcantabrico_nivel_spider"
10    allowed_domains = ["www.chcantabrico.es"]
11    custom_settings = {
12        'FEEDS': {
13            'JSONs/RawData/datos_nivel_chcantabrico.json': {
14                'format': 'json',
15                'encoding': 'utf-8',
16                'overwrite': True,
17            }
18        }
19    }
20
21    def start_requests(self):
22        with open("JSONs/RawCode/codigos_chcantabrico.json",
23                 encoding="utf-8") as f:
24            data = json.load(f)
25            for estacion in data:
26                params_nivel = {
27                    'p_p_id': 'GraficaEstacion_INSTANCE_wH0LL6jTUysu',
28                    'p_p_lifecycle': '2',
29                    'p_p_state': 'normal',
30                    'p_p_mode': 'view',
31                    'p_p_resource_id': 'downloadCsv',
32                    'p_p_cacheability': 'cacheLevelPage',
33                    '_GraficaEstacion_INSTANCE_wH0LL6jTUysu_cod_estacion':
34                        f'{estacion["codigo"]}',
35                    '_GraficaEstacion_INSTANCE_wH0LL6jTUysu_tipodata': 'nivel',
36                }
37                url = 'https://www.chcantabrico.es/evolucion-de-niveles'
38                yield scrapy.FormRequest(url=url,
```

```
39         callback=self.parse,
40         cb_kwargs={'estacion': estacion['codigo']}
41     )
42
43     def parse(self, response, estacion):
44         if not response.text.startswith('-'):
45             urlData = response.text
46             rawData = pd.read_csv(io.StringIO(urlData), delimiter=';',
47                                   encoding='utf-8', header=1)
48             rawData.columns = ['fecha y hora', 'nivel (m)']
49             parsedData = rawData.to_json(orient="records")
50
51             yield {
52                 'estacion': estacion,
53                 'datos': json.loads(parsedData)
54             }
```

Código 5.10: Chcantabrico Nivel Spider

```
1  import io
2  import json
3
4  import pandas as pd
5  import scrapy
6
7
8  class ChcantabricoPluvioSpider(scrapy.Spider):
9      name = "chcantabrico_pluvio_spider"
10     allowed_domains = ["www.chcantabrico.es"]
11     custom_settings = {
12         'FEEDS': {
13             'JSONs/RawData/datos_pluvio_chcantabrico.json': {
14                 'format': 'json',
15                 'encoding': 'utf-8',
16                 'overwrite': True,
17             }
18         }
19     }
20
21     def start_requests(self):
22         with open("JSONs/RawCode/codigos_chcantabrico.json",
23                 encoding="utf-8") as f:
```

```

23     data = json.load(f)
24     for estacion in data:
25         params_pluvio = {
26             'p_p_id': 'GraficaEstacion_INSTANCE_ND81Xo17PIZ7',
27             'p_p_lifecycle': '2',
28             'p_p_state': 'normal',
29             'p_p_mode': 'view',
30             'p_p_resource_id': 'downloadCsvPluvio',
31             'p_p_cacheability': 'cacheLevelPage',
32             '_GraficaEstacion_INSTANCE_ND81Xo17PIZ7_cod_estacion':
33                 f'{estacion["codigo"]}',
34             '_GraficaEstacion_INSTANCE_ND81Xo17PIZ7_tipodato': 'pluvio',
35         }
36         url = 'https://www.chcantabrico.es/precipitacion-acumulada'
37         yield scrapy.FormRequest(url=url,
38             method='GET',
39             formdata=params_pluvio,
40             callback=self.parse,
41             cb_kwargs={'estacion': estacion['codigo']})
42
43     def parse(self, response, estacion):
44         if not response.text.startswith('-'):
45             urlData = response.text
46             rawData = pd.read_csv(io.StringIO(urlData), delimiter=';',
47                 encoding='utf-8', header=1)
48             rawData.columns = ['fecha y hora', 'nivel (m)']
49             parsedData = rawData.to_json(orient="records")
50
51             yield {
52                 'estacion': estacion,
53                 'datos': json.loads(parsedData)
54             }

```

Código 5.11: Chcantabrico Pluviometric Spider

Chacantabrico muestra datos tanto del nivel del río como de la precipitación, aunque lo hace en dos direcciones distintas, haciendo necesario el uso de dos Spiders.

A su vez, como Chcantabrico no muestra los datos por pantalla, incluyendo un botón sobre el que pulsar para obtenerlos descargando un fichero CSV, en vez de hacer una request básica mediante la clase Request, vamos a hacer uso de FormRequest para

hacer una llamada GET, pudiendo simular la llamada a un formulario y obtener los datos que este devuelve. De esta forma, pasándole los parámetros necesarios en el argumento `formdata` a la URL indicada, podemos obtener los datos sin la necesidad de ningún CSV, simulando en cierto modo una llamada mediante `cURL`. Cabe mencionar que, `Request` devuelve un `HTMLResponse` y que, la respuesta que obtenemos de estas llamadas no es código HTML, por lo que, aun en caso de que llegue a ser posible usar `Request`, es más correcto el uso de `FormRequest` devolviendo un `FormResponse` para este tipo de casos.

El uso del argumento `cb_kwargs` sirve para enviar un mayor número de argumentos a la función `parse()` de los que normalmente recibe, es por ello que la función `parse()` recibe un tercer argumento que hemos decidido llamar estación. En este caso para poder enviar a cada conjunto de datos recibidos el código de la estación a la que pertenecen, pues dentro de la respuesta obtenida solo se proporciona el nombre de esta.

Dentro de la función `parse()` lo primero que se hace es comprobar que realmente se ha recibido una respuesta correcta, pues, aunque todas las estaciones disponen de datos del nivel del río, no todas disponen los de precipitación, el problema viene cuando a estas estaciones se les piden los datos, ya que en vez de enviar un error 404 como sería esperado.

-  
FECHA;VALOR(mm)

Una alternativa para deshacerse de esta comprobación sería eliminando aquellas estaciones que no proporcionen datos o filtrándolas para no hacer la llamada directamente, aunque esto no solo nos resultaría más complejo, si no que nos crearía el problema de que cada cierto tiempo habría que comprobar si alguna estación ha empezado a proporcionar datos para incluirla nuevamente en la lista de estaciones a las cuales hacer llamada.

Una vez hecha la comprobación, en caso de que no sea una respuesta vacía, obtenemos el texto que viene proporcionado con el siguiente formato.

Ribera de Piquín  
FECHA;VALOR(m)  
03/08/2023 11:30:00;0.153  
03/08/2023 11:45:00;0.153  
03/08/2023 12:00:00;0.153  
...

Ese texto, al tener formato CSV lo leemos mediante la función `read_csv()` incluida en la librería `pandas`, indicamos el delimitador, la codificación y la línea que representa la cabecera, empezando de la 0, en este caso la 1, pues no nos interesa el nombre de la estación. Finalmente, como la función espera que se le pase una ruta a un fichero, lo que hacemos mediante `io.StringIO()` es crear un objeto con el que simular un fichero en memoria, pasando de disponer texto plano a un `DataFrame` de `pandas`.

Como últimos pasos, cambiamos los nombres de las cabeceras a aquellos definidos de forma global para todas las webs y, convertimos el `DataFrame` en un fichero JSON con la función `to_json()` indicando que el formato sea "records", esto implica que cada línea del `DataFrame` va a representar un objeto JSON.

```
1  import json
2
3  import scrapy
4
5
6  class ChcantabricoCoordSpider(scrapy.Spider):
7      name = "chcantabrico_coord_spider"
8      allowed_domains = ["ceh.cedex.es"]
9      custom_settings = {
10         'FEEDS': {
11             'JSONs/RawData/coordenadas_chcantabrico.json': {
12                 'format': 'json',
13                 'encoding': 'utf-8',
14                 'overwrite': True,
15             }
16         }
17     }
18
19     def start_requests(self):
20         with open("JSONs/RawCode/codigos_chcantabrico.json",
21                 encoding="utf-8") as f:
22             data = json.load(f)
23             for estacion in data:
24                 url = f'https://ceh.cedex.es/anuarioaforos/afo/estaf-datos.asp?'
25                 'indroea={estacion["codigoSecundario"]}'
26                 yield scrapy.Request(url, self.parse)
27
28     def parse(self, response):
29         longitud = response.css('p::text')[6].get().strip()
```



```
29     latitud = response.css('p::text')[7].get().strip()
30     estacion = response.css('font::text')[14].get().strip()
31
32     yield {
33         'coordenadas': f'Lat: {latitud} | Lon: {longitud}',
34         'estacion': estacion,
35     }
```

Código 5.12: Chcantabrico Coordinates Spider

Aunque en la web misma se proporciona un mapa indicando la localización de cada estación, las coordenadas de esta no están disponibles para adquirir dentro de la web, es por eso que es necesario el uso de la web del centro de estudios hidrológicos para poder obtener las coordenadas. En esta página las podemos encontrar mediante el "codigoSecundario" anteriormente obtenido, desgraciadamente, no todas las estaciones incluidas en Chacantabrico están listadas en esta página, siendo el mayor inconveniente para el correcto funcionamiento del apartado de predicción.

Exceptuando el uso del "codigoSecundario" para referenciar estaciones, esta es una Spider muy simple la cual no dispone de nada que no haya sido anteriormente explicado en el apartado tres.

### Código descartado

Siendo Chcantabrico la primera web de la que se obtuvo los datos, sin gran conocimiento de Scrapy y sobre todo, sin saber realmente como sería la plataforma, el planteamiento de la obtención de datos se realizó de forma ajena a las Spider de Scrapy. Viendo que los datos no estaban presentes en la web y que Scrapy no dispone de interacción JavaScript por defecto, la única alternativa viable en esos momentos fue probar a realizar una llamada cURL por terminal, al ver que efectivamente mediante cURL era posible obtener los datos deseados, se escribió un script para los datos de nivel y otro para los de precipitación.

```
1     import pandas as pd
2     import io
3     import requests
4     import json
5
6     with open('codigos_estaciones_chcantabrico.json', 'r',
7             encoding='utf-8') as f:
8         data = json.load(f)
9         datos = []
```

```
10 for item in data:
11
12     params_pluvio = {
13         'p_p_id': 'GraficaEstacion_INSTANCE_ND81Xo17PIZ7',
14         'p_p_lifecycle': '2',
15         'p_p_state': 'normal',
16         'p_p_mode': 'view',
17         'p_p_resource_id': 'downloadCsvPluvio',
18         'p_p_cacheability': 'cacheLevelPage',
19         '_GraficaEstacion_INSTANCE_ND81Xo17PIZ7_cod_estacion':
20             f'{item["codigo"]}',
21         '_GraficaEstacion_INSTANCE_ND81Xo17PIZ7_tipodato': 'pluvio',
22     }
23
24     response_pluvio =
25         requests.get('https://www.chcantabrico.es/precipitacion-acumulada',
26                     params=params_pluvio)
27     if response_pluvio.status_code == 200:
28         if not response_pluvio.text.startswith('-'):
29             urlData = response_pluvio.text
30             rawData = pd.read_csv(io.StringIO(urlData), delimiter=';',
31                                     encoding='utf-8', header=1)
32             rawData.columns = ['fecha y hora', 'precipitacion (mm)']
33             parsedData = rawData.to_json(orient="records")
34
35             estacion = {
36                 'estacion': item["codigo"],
37                 'datos': json.loads(parsedData)
38             }
39             datos.append(estacion)
40         else:
41             print(f'{item["estacion"]} Error retrieving data: 404')
42             print("-----")
43     else:
44         print(f'{item["estacion"]} Error retrieving data:
45             {response_pluvio.status_code}')
46         print("-----")
47
48 with open('../JSONs/RawData/datos_pluvio_chcantabrico.json',
49         'w', encoding='utf-8') as outfile:
50     json.dump(datos, outfile)
```

Código 5.13: Script de obtención de datos pluviométricos descartado

Todo el apartado de tratamiento de los datos es idéntico al realizado con la Spider, solo que en vez de usar FormRequest, hacemos uso de la librería requests para realizar la llamada get(), tras realizarla, se comprueba que haya sido exitosa (esta comprobación la realiza Scrapy automáticamente) y, en caso de serlo se realiza todo el tratamiento.

Aunque en esta versión se almacenan todos los datos en un mismo JSON, originalmente los datos eran guardados en un CSV por cada estación, de tal manera que el nombre del CSV era el mismo que el de la estación perteneciente. Más adelante al consolidar más la plataforma, sobre todo el uso de Django para la creación de una API, se vio que era más útil guardar los datos no solo en formato JSON si no que disponer de un único fichero por estación, de esta forma solo sería necesario realizar una única llamada por estación a la API para cargar los datos. Llegando a esta versión del script.

Posteriormente, llegado el momento de la automatización quedo claro que, aun siendo posible automatizar el proceso con el script anterior, iba a suponer un problema para la modularidad del proyecto. El tener múltiples scripts de diferentes fuentes solo aumentaba la complejidad, necesitando de un script de ejecución por cada web la cual se deseaba trabajar con. Implicando la necesidad de no solo crear el código de extracción y tratamiento de datos, sino el de ejecución también. Esto rompía hasta cierto punto la experiencia deseada, el usuario solo debería preocuparse de los datos y la única interacción que debería realizar con el proceso de ejecución sería el añadir una única línea de comando referenciando el nombre usado para el nuevo modulo.

Es por eso que se investigo la posibilidad de obtener los datos mediante Scrapy, estudiando los diferentes objetos Request proporcionados, hasta llegar a la versión actual con el uso de FormRequest.

### 5.2.3 MeteoNavarra

```
1  import scrapy
2
3
4  class MeteonavarraCodeSpider(scrapy.Spider):
5      name = "meteoNavarra_code_spider"
6      allowed_domains = ["meteo.navarra.es"]
7      start_urls =
8          ["http://meteo.navarra.es/estaciones/mapadeestaciones.cfm#"]
9      custom_settings = {
```

```

9      'FEEDS': {
10          'JSONs/RawCode/codigos_meteoNavarra.json': {
11              'format': 'json',
12              'encoding': 'utf-8',
13              'overwrite': True,
14          }
15      }
16  }
17
18  def parse(self, response):
19      rows = response.css('div#tabAUTO script::text').getall()
20
21      for row in rows:
22          yield {
23              'estacion': row.split(',')[3],
24              'codigo': row.split(',')[0].split('(')[1],
25          }

```

Código 5.14: MeteoNavarra Code Spider

Probablemente una de las Spider más simples entre todas las usadas, aunque como se comenta anteriormente, el mayor desafío fue la estructuración de la web y como obtener los datos de esta.

```

1  import datetime
2  import json
3
4  import scrapy
5
6
7  class MeteonavarraDataSpider(scrapy.Spider):
8      name = "meteoNavarra_data_spider"
9      allowed_domains = ["meteo.navarra.es"]
10     custom_settings = {
11         'FEEDS': {
12             'JSONs/RawData/datos_meteoNavarra.json': {
13                 'format': 'json',
14                 'encoding': 'utf-8',
15                 'overwrite': True,
16             }
17         }
18     }
19

```

```

20 def start_requests(self):
21     current_date = datetime.date.today()
22     delta = datetime.timedelta(days=1)
23     tomorrow_date = current_date + delta
24     yesterday_date = current_date - delta
25     tomorrow_date_format = tomorrow_date.strftime(" %d%%2F
        %m%%2F%Y").replace(' 0', '')
26     yesterday_date_format = yesterday_date.strftime(" %d%%2F
        %m%%2F%Y").replace(' 0', '')
27
28     with open("JSONs/RawCode/codigos_meteoNavarra.json",
        encoding="utf-8") as f:
29         data = json.load(f)
30         for estacion in data:
31             url =
32                 f'http://meteo.navarra.es/estaciones/estacion_datos_m.cfm?'
33                 f'IDEstacion={estacion["codigo"]}&p_10' \
34                 f'=1&p_10=2&p_10=3&p_10=11&fecha_desde={yesterday_date_format}'
35                 f'&fecha_hasta={tomorrow_date_format}'
36             yield scrapy.Request(url, self.parse)
37
38 def parse(self, response):
39     rows = response.css('table.border tr:not([bgcolor*="#FFFFFF"])')
40     estacion = response.css('table a::attr(href)')[7].get()
41
42     datos = []
43     for row in rows:
44         dato = {
45             'fecha y hora':
46                 row.xpath('./td[1]/text()').get().strip().replace(' ', '
47                 ') + ':00',
48             'temperatura (C)': row.xpath('./td[2]/font/text()').get(),
49             'humedad relativa (%)':
50                 row.xpath('./td[3]/font/text()').get(),
51             'radiacion global (W/m^2)':
52                 row.xpath('./td[4]/font/text()').get(),
53             'precipitacion (l/mm^2)':
54                 row.xpath('./td[5]/font/text()').get(),
55         }
56
57     if dato['radiacion global (W/m^2)'] == '- -':

```

```
52     dato['radiacion global (W/m^2)'] = None
53
54     if dato['precipitacion (l/mm^2)'] != '- -':
55         datos.append(dato)
56
57     if datos:
58         yield {
59             'estacion': estacion.split('idestacion=')[1].split('&')[0],
60             'datos': datos,
61         }
62
63     next_page = response.css("table a::attr(href)").getall()
64     page_number = response.xpath("//b/text()").getall()
65     if not page_number[0] == page_number[1]:
66         next_page = response.urljoin(next_page[7])
67     yield scrapy.Request(next_page, callback=self.parse)
```

Código 5.15: MeteoNavarra Data Spider

A diferencia del resto de páginas, que te definen automáticamente una franja de fechas a mostrar, meteoNavarra da la posibilidad de que el usuario elija las fechas que desee ver, es por eso que hace uso de `datetime`, obteniendo el día de ayer y el de mañana (meteoNavarra excluye los datos del día indicado), para su correcto funcionamiento, mediante el método `strftime`, formateamos el `datetime` para que corresponda con el codificado URL.

Tras obtener los datos lo primero que se realiza es si existe un valor de radiación global, esto se hace debido a que no todas las estaciones proporcionan este dato, posteriormente, puesto que la web muestra todas las franjas horarias dentro del que se corresponde con el día actual, se eliminan todas esas horas sin datos con la comprobación respecto a la precipitación. A continuación, en caso de que existan datos a guardar se realiza el `yield`.

Finalmente, como los datos de ciertas estaciones están repartidos en dos páginas, navegamos a esa segunda página para realizar el mismo proceso descrito anteriormente.

Esto es posible gracias a que `yield`, aunque a groso modo funcione como un `return`, no fuerza el final de la ejecución, por lo que todo código por debajo de este será ejecutado, llegando a haber múltiples `yields` en una misma función.

```
1     import json
2
```

```
3 import scrapy
4
5
6 class MeteonavarraCoordSpider(scrapy.Spider):
7     name = "meteoNavarra_coord_spider"
8     allowed_domains = ["meteo.navarra.es"]
9     custom_settings = {
10         'FEEDS': {
11             'JSONs/RawData/coordenadas_meteoNavarra.json': {
12                 'format': 'json',
13                 'encoding': 'utf-8',
14                 'overwrite': True,
15             }
16         }
17     }
18
19     def start_requests(self):
20         with open("JSONs/RawCode/codigos_meteoNavarra.json",
21                 encoding="utf-8") as f:
22             data = json.load(f)
23             for estacion in data:
24                 url = f'http://meteo.navarra.es/estaciones/estacion.cfm?'
25                 'IDestacion={estacion["codigo"]}'
26                 yield scrapy.Request(url, self.parse)
27
28     def parse(self, response):
29         coordenadas = response.css('td::text')[19].get()
30         estacion = response.css('input::attr(value)').get()
31
32         yield {
33             'coordenadas': coordenadas.strip().replace('\r\n\t', ' '),
34             'estacion': estacion,
```

Código 5.16: MeteoNavarra Coordinates Spider

Por último, necesitamos de una Spider para la obtención de las coordenadas pues no están disponibles para obtener dentro de las páginas anteriores, pero al igual que la de código no dispone realmente de nada relevante que mencionar.

### 5.2.4 Agua en Navarra

Las Spiders de Agua en Navarra son las más complejas entre todas, usando una filosofía ligeramente distinta al resto de Spiders y por el uso de Selenium para poder interactuar con la web mediante JavaScript.

```
1 import scrapy
2
3
4 class AguaEnNavarraCodeSpider(scrapy.Spider):
5     name = "aguaEnNavarra_code_spider"
6     allowed_domains = ["administracionelectronica.navarra.es"]
7     start_urls =
8         ["https://administracionelectronica.navarra.es/aguaEnNavarra/"
9          "ctaMapa.aspx?IDOrigenDatos=1&IDMapa=1"]
10    custom_settings = {
11        'FEEDS': {
12            'JSONs/RawCode/codigos_aguaEnNavarra.json': {
13                'format': 'json',
14                'encoding': 'utf-8',
15                'overwrite': True,
16            }
17        }
18
19    def parse(self, response):
20        for link in response.css('dl#navarramap a::attr(href)'):
21            if link.get() != 'ctaMapa.aspx?IdMapa=1&IDOrigenDatos=1':
22                yield response.follow(link.get(), callback=self.parse_area)
23
24    def parse_area(self, response):
25        for link in response.css('area[shape="rect"]::attr(href)'):
26            yield response.follow(link.get(), callback=self.parse_data)
27
28    def parse_estacion(self, response):
29        urls = response.xpath('//span/a/@href').getall()
30        estacion = response.css('div#bloq_iconos span
31                                span::text').getall()
32        codigoEstacion =
33            response.css("form#frmDatosEstacion::attr(action)").get()
34        codigos = []
35        for url in urls:
```



```

34         codigos.append(url.split('=')[1])
35
36     yield {
37         'descripcion': estacion[0],
38         'municipio': estacion[1],
39         'rio': estacion[2],
40         'coordenadas': estacion[3],
41         'estacion': codigoEstacion.split('=')[-1],
42         'codigos': codigos,
43     }

```

Código 5.17: Agua en Navarra Code Spider

Como se puede apreciar, esta Spider hace uso de tres funciones `parse()`, al contrario del resto que solo disponían de una. Esto se debe hasta cierto punto a la estructuración de la página, haciendo más sencillo navegar por ella que crear una Spider por cada página que tenemos que visitar.

Inspeccionando el código veremos que partimos de una `start_urls`, llamando con ella a `parse()`, de esa web nos interesan recorrer todos los link menos aquel en el que ya estamos (`'ctaMapa.aspx?IdMapa=1&IDOrigenDatos=1'`), llevándonos a la página de cada area de estaciones. El uso de `response.follow()` es equivalente a realizar un Request, pero no necesita pararle una URL completa, solo con la ruta es capaz de generar la llamada, en caso de necesitar explícitamente Request esta seria la manera.

```
yield Request(url=response.urljoin(link.get()), callback=self.parse_area)
```

Dentro de `callback` indicamos a que función debe realizar la llamada, pasando de `parse()` a `parse_area()`. En esta realizamos el mismo proceso, obtenemos todos los link a las webs, ya sí, de la estación y llamamos a `parse_estacion()`.

Una vez en `parse_estacion()` como ocurre con Chcantabrico, la estación dispone de múltiples códigos, aquel que referencia la estación, uno para el nivel y aunque no en todas, otro para el caudal del río.

```

1  import scrapy
2  from scrapy_selenium import SeleniumRequest
3  from selenium.webdriver.common.by import By
4  from selenium.webdriver.support import expected_conditions as EC
5
6
7  class AguaEnNavarraDataSpider(scrapy.Spider):
8      name = "aguaEnNavarra_data_spider"
9      allowed_domains = ["administracionelectronica.navarra.es"]

```

```
10 start_urls =
11     ["https://administracionelectronica.navarra.es/aguaEnNavarra/"
12     "ctaMapa.aspx?IDOrigenDatos=1&IDMapa=1"]
13 custom_settings = {
14     'FEEDS': {
15         'JSONs/RawData/datos_aguaEnNavarra.json': {
16             'format': 'json',
17             'encoding': 'utf-8',
18             'overwrite': True,
19         }
20     }
21
22 def parse(self, response):
23     for link in response.css('dl#navarramap a::attr(href)'):
24         if link.get() != 'ctaMapa.aspx?IdMapa=1&IDOrigenDatos=1':
25             yield response.follow(link.get(), callback=self.parse_area)
26
27 def parse_area(self, response):
28     for link in response.css('area[shape="rect"]::attr(href)'):
29         yield response.follow(link.get(), callback=self.parse_estacion)
30
31 def parse_estacion(self, response):
32     for link in response.css('div#divResultadosAforo a::attr(href)'):
33         yield SeleniumRequest(
34             url=response.urljoin(link.get()),
35             wait_time=2,
36             wait_until=EC.element_to_be_clickable((By.ID,
37                 'btnDatosNumericos')),
38             callback=self.parse_data,
39             script='document.querySelector("#btnDatosNumericos").click()',
40         )
41
42 def parse_data(self, response):
43     tipo = response.css('span#lblSenal::text').get()
44     estacion = response.css('li#cabecera_nombreEstacion
45         a::attr(href)').get()
46     fechas = response.css('span.cont_fecha_gra::text').getall()
47     valores = response.css('span.cont_valor_gra::text').getall()
48
49     datos = []
```

```
48     if tipo == "Nivel Rio":
49         for i, fecha in enumerate(fechas):
50             dato = {
51                 'fecha y hora': fecha.strip() + ':00',
52                 'nivel (m)': valores[i].strip(),
53             }
54             datos.append(dato)
55     else:
56         for i, fecha in enumerate(fechas):
57             dato = {
58                 'fecha y hora': fecha.strip() + ':00',
59                 'caudal (m^3/s)': valores[i].strip(),
60             }
61             datos.append(dato)
62
63     yield {
64         'estacion': estacion.split('=')[-1],
65         'datos': datos,
66     }
```

Código 5.18: Agua en Navarra Data Spider

Esta Spider se puede decir que es una amplificación de la anterior, pues parte de la misma URL de inicio y va recorriendo las mismas webs que la anterior, solo que esta recorre una página más. De esta forma a diferencia del resto de Spider destinadas a obtener datos, no necesita de un fichero con los códigos para funcionar.

Lo primero que llama la atención de esta Spider respecto al resto es el uso de `SeleniumRequest`, este tipo de objeto `Request` pertenece a la librería `scrapy_selenium`, siendo una alternativa sencilla de unificar las funcionalidades de Selenium en Scrapy, de esta forma, podemos lidiar con la falta de integración de interacción con JavaScript en Scrapy.

Tras realizar el mismo proceso que antes, la Spider llega a la función `parse_estacion()`, esta vez, en vez de obtener los datos de la estación, accedemos a los links que permiten mostrar los datos mediante el `SeleniumRequest`. Como el link al que somos dirigidos muestra los datos en forma de diagrama y, para acceder a los datos numéricos es necesario pulsar un botón, `SeleniumRequest` nos permite, esperar a que este esté correctamente cargado con el argumento `wait_until`, indicando que elemento deseas esperar a ser cargado; en caso de que no se cargue, es recomendable usar el argumento `wait_time`, que usado junto al anterior, establece el tiempo antes de dar la llamada por errónea; si el botón se carga correctamente, en el argumento `script_in-`

dicamos que se debe pulsar sobre este, llevándonos finalmente a los datos numéricos.

En la función `parse_data()`, obtenemos los datos por columna en vez de fila, pues la web no hace un correcto uso formateo por tabla, pues no existe esta tabla, insertando los datos sobre un elemento `div` y formateándolos posteriormente con CSS. Esta forma de obtener los datos fuerza la obtención de las fechas por un lado y los valores por otro, resultando en dos listas que debemos unir. A su vez, ya que hacemos uso de una Spider para obtener todos los datos aunque estén presentes en distintas páginas, hace necesaria la comprobación inicial de que datos estamos recibiendo para poder almacenarlos correctamente.

### Configuración Selenium

Para que `SeleniumRequest` funcione, es necesario incluir las siguientes líneas de código en el archivo `settings.py` generado por Scrapy.

```
1  from shutil import which
2
3  SELENIUM_DRIVER_NAME = 'chrome'
4  SELENIUM_DRIVER_EXECUTABLE_PATH = which('chromedriver')
5  SELENIUM_DRIVER_ARGUMENTS = ['--headless',
6  '--disable-logging',
7  '--disable-in-process-stack-traces',
8  '--log-level=1',
9  '--disable-extensions'
10 ]
11
12 DOWNLOADER_MIDDLEWARES = {
13     'scrapy_selenium.SeleniumMiddleware': 800
14 }
```

Código 5.19: Agua en Navarra configuración Selenium

En nuestro caso indicamos que el navegador deseado para realizar el trabajo es Chrome e indicamos mediante el método `which` la ruta del ejecutable del driver de chrome.

Los argumentos usados indican, `--headless` significa que las instancias de Chrome creadas sean sin interfaz, de esta forma no tendremos cientos de ventanas de Chrome abriéndose y cerrándose cada vez que se ejecute la Spider; `--disable-logging` como su nombre indica elimina el Log; `--disable-in-process-stack-traces` desactiva el stack-trace, esto se realiza únicamente con el fin de intentar mejorar el rendimiento; `--log-level` indica el tipo de log que desas mostrar, siendo `INFO = 0`, `ALERTA = 1`, `ERROR`

= 2 y FATAL = 3, aunque el Log este desactivado esta bien indicar que clase de Log deseas que aparezca en caso de que se vuelva a activar; `--disable-extensions` desactiva todas las extensiones que podamos tener instaladas, nuevamente con el fin de mejorar el rendimiento.

Una vez incluido el middleware, ya solo quedaria descargar el driver de chrome.

Chromedriver.exe puede ser obtenido en la web <https://chromedriver.chromium.org/downloads>, descargando aquel que corresponda a la versión de Chrome instalada. Una vez descomprimido debe ser incluido dentro de la carpeta de proyecto de Scrapy.

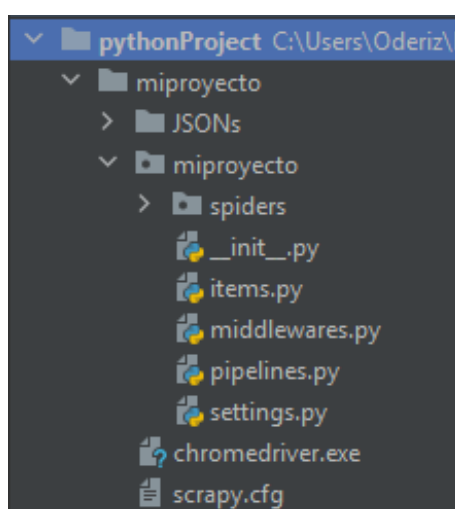


Figura 5.5: Ruta chromedriver.exe

En caso de estar en Windows con esto ya es suficiente, aunque al usar Debian son necesarios unos paso más para configurar chromedriver.

### Alternativas probadas

Antes de saber que es necesario dirigirse a la pagina donde se muestra la gráfica antes de poder acceder a la página con los datos numéricos, se probó con una Spider que accedía directamente a esta. Visto que el resultado obtenido estaba vacío y, que en apariencia la Spider era correcta, obteniendo teóricamente aquellos datos deseados, se comprobó la web manualmente, resultando en la obtención de un error 3.3.

Es por eso que, partiendo de un uso de Scrapy sin uso de extensiones de terceros, se probó una solución al problema siguiendo la metodología de recorrer las webs, de esta manera, visitaba inicialmente la web con la gráfica para posteriormente redi-

rigirse a los datos numéricos. Resultando en un nuevo fracaso. Esta vez si que se obtenían datos, aunque no de forma correcta, muchas estaciones aparecían repetidas múltiples veces, de tres a siete veces en el peor de los casos visto. Siendo el resultado esperado la aparición de una misma estación un total de dos veces, una con los datos del caudal y otra por los datos del nivel. Volviendo a la comprobación manual resulto en el mismo comportamiento, por alguna razón, una vez dentro de la web de los datos numéricos, al cambiar el código de la estación y recargar la pagina, lo mismo se actualizaban los datos para mostrar los de la nueva estación como no lo hacían, mostrando los datos de la anterior.

Esto causo la incertidumbre de si era posible obtener datos la web de forma fiable. Finalmente, barajando la posibilidad de pulsar el botón para acceder los datos, se dio con la herramienta usada en la versión final, Selenium y, tiempo después con una alternativa más moderna a esta, Playwright.

Por suerte, Selenium, siendo usado mediante la extensión scrapy\_selenium, resulto funcionar a la primera, aunque a un costo temporal relativamente alto, en comparación con las demás Spider, con una media de dos minutos y medio de ejecución.

Al ser código destinado a ser ejecutado en intervalos de quince minutos, un tiempo de ejecución así no debería acarrear ningún problema, pero aun y todo, se probó una cuarta alternativa mediante Playwright, prometiendo mejoras en los tiempos de ejecución y una mejor optimización.

```
#Instalamos scrapy-playwright
pip install scrapy-playwright
```

```
#Instalamos los navegadores compatibles
playwright install
```

Antes que nada, cabe mencionar que Playwright no es compatible con Windows y, tampoco lo es con Debian 11 en su versión actual, resultando ser Debian 11 la versión usada para este proyecto, por lo que no se ha podido llegar a comprobar el funcionamiento del siguiente código, aunque debería ser correcto.

Para hacer uso de Playwrith con Scrapy, primero que todo tenemos que configurar las dependencias de Playwright, que las implementaremos mediante custom\_settings dentro de una Spider o, si se desea, tambien pueden ser configuradas en settings.py.

```
1 custom_settings = {
```

```
2     "TWISTED_REACTOR":  
3         "twisted.internet.asyncioreactor.AsyncioSelectorReactor",  
4     "DOWNLOAD_HANDLERS": {  
5         "https":  
6             "scrapy_playwright.handler.ScrapyPlaywrightDownloadHandler",  
7         "http":  
8             "scrapy_playwright.handler.ScrapyPlaywrightDownloadHandler",  
9     }  
10 }
```

Código 5.20: Configuración Playwright

Finalmente, para hacer uso de la herramienta, a diferencia de con Selenium, no disponemos de un nuevo objeto Request, si no que hace uso del Request implementado en Scrapy indicándole por argumento que debe usar Playwright.

```
1     yield Request(  
2         url=url,  
3         callback=self.parse,  
4         meta=dict(  
5             playwright=True,  
6         ),  
7     )
```

Código 5.21: Playwright basic Request

En nuestro caso, para obtener el comportamiento previsto, la Request configurada es la siguiente.

```
1     from scrapy_playwright.page import PageMethod  
2  
3  
4     yield Request(  
5         url=url,  
6         callback=self.parse,  
7         meta=dict(  
8             playwright=True,  
9             playwright_page_methods=[  
10             PageMethod("wait_for_selector", selector="div.botoneraGrafico",  
11                 state="visible"),  
12             PageMethod("click", selector="input#btnDatosNumericos"),  
13             PageMethod("waitForEvent", event="click"),  
14         ],  
15     ),
```

15

)

## Código 5.22: Agua en Navarra Playwright Request

Los PageMethods indican las acciones a realizar una vez se hace la request, siendo la alternativa optada por Playwright al uso de JavaScript. Primero esperamos a que el botón esté cargado dentro de la página, pulsamos sobre el y, esperamos a que la acción de pulsar sea realizada.

Según la documentación y los ejemplos estudiados, esta alternativa debería funcionar, pero puesto que no se ha podido ejecutar no es seguro su funcionamiento.



## **Conclusiones y Trabajo Futuro**



## Referencias

- [1] C. Dierbach, "Python as a first programming language," *Journal of Computing Sciences in Colleges*, vol. 29, no. 3, pp. 73–73, 2014.
- [2] "Python 3.11.3 Documentation general python faq." <https://docs.python.org/3/faq/general.html>. Accessed: 2023-04-07.
- [3] L. E. Borges, *Python para desenvolvedores: aborda Python 3.3*. Novatec Editora, 2014.
- [4] M. F. Krafft, *The Debian system: concepts and techniques*. No Starch Press, 2005.
- [5] R. Hertzog and R. Mas, *The Debian Administrator's Handbook: Debian Jessie From Discovery To Mastery*. Freexian, 2015.
- [6] "Debian Project Documentation a brief history of debian." <https://www.debian.org/doc/manuals/project-history/index.en.html>. Accessed: 2023-04-12.
- [7] "Debian Project Documentation our philosophy: Why we do it and how we do it." <https://www.debian.org/intro/philosophy.en.html>. Accessed: 2023-04-12.
- [8] R. P. Pollei, *Debian 7: System Administration Best Practices*. Packt Publishing, 2013.
- [9] "Debian Project Documentation what does free mean?." <https://www.debian.org/intro/free.en.html>. Accessed: 2023-04-12.
- [10] "GNU About Free Software Documentation what is free software?." <https://www.gnu.org/philosophy/free-sw.en.html>. Accessed: 2023-04-12.

- [11] “Debian Project Documentation debian social contract version 1.2 ratified on october 1st, 2022..” [https://www.debian.org/social\\_contract.en.html](https://www.debian.org/social_contract.en.html). Accessed: 2023-04-12.
- [12] “Debian Project Documentation what is the partners program?.” <https://www.debian.org/partners/index.en.html>. Accessed: 2023-04-12.
- [13] “Debian Project Documentation how to donate to the debian project.” <https://www.debian.org/donations.en.html>. Accessed: 2023-04-12.
- [14] “Debian Project Documentation reasons to use debian.” [https://www.debian.org/intro/why\\_debian.en.html](https://www.debian.org/intro/why_debian.en.html). Accessed: 2023-04-12.
- [15] D. Ghimire, “Comparative study on python web frameworks: Flask and django,” 2020.
- [16] “Python Documentation web frameworks for python.” <https://wiki.python.org/moin/WebFrameworks>. Accessed: 2023-04-12.
- [17] D. Glez-Peña, A. Lourenço, H. López-Fernández, M. Reboiro-Jato, and F. Fdez-Riverola, “Web scraping technologies in an api world,” *Briefings in bioinformatics*, vol. 15, no. 5, pp. 788–797, 2014.
- [18] “Django Documentation design philosophies.” <https://docs.djangoproject.com/en/3.0/misc/design-philosophies/>. Accessed: 2023-04-12.
- [19] M. Alchin, *Pro Django*. Apress, 2013.
- [20] A. Ravindran, *Django Design Patterns and Best Practices*. Packt Publishing Ltd, 2015.
- [21] B. Zhao, “Web scraping,” *Encyclopedia of big data*, pp. 1–3, 2017.
- [22] V. Krotov and L. Silva, “Legality and ethics of web scraping,” 2018.
- [23] H. Yang, “Design and implementation of data acquisition system based on scrapy technology,” in *2019 2nd International Conference on Safety Produce Informatization (IICSPI)*, pp. 417–420, IEEE, 2019.

```
1 public void mostrarTodasNotas(){
2     int a = 1;
3     System.out.println("Estas son todas las notas que hay
4         guardadas.");
5     for(int j=0; j<lista_notas.size(); j++){
6         System.out.print(a+"-");
7         System.out.println(lista_notas.get(j).toString());
8         a++;
9     }
```

Código A1: Mostrar notas.