



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

Lista de Figuras	v
Lista de Códigos	vii
1 Introducción	1
2 Datos	7
2.1 CHCantábrico	8
2.2 Agencia estatal de Meteorología (Aemet)	9
2.3 Meteorología y climatología de Navarra (MeteoNavarra)	10
2.4 El Agua en Navarra	13
3 Tecnologías	19
3.1 Sistema Operativo Debian	19
3.2 Framework necesarios	21
3.2.1 Librería vs Framework	21
3.2.2 Django Framework	22
3.3 Herramienta para Web Scraping	23
3.3.1 Procedimiento básico en Web Scraping	23
3.3.2 Scrapy Framework	24
3.4 Base de Datos	25
3.4.1 PostGreSQL	25
3.4.2 Arquitectura Preliminar	26
4 Diseño de la Plataforma	27
4.1 Trabajar con los Datos	28
4.1.1 Datos obtenidos por página	28

4.1.2 Formato de Datos	29
4.1.3 Filtrado de Datos	29
4.2 Arquitectura	30
4.2.1 Entornos virtuales	30
4.2.2 Spiders	31
4.2.3 Runners	31
4.2.4 Executers	32
4.2.5 Directorios de los JSONs de datos	32
4.2.6 Posts	32
4.2.7 Cron	33
4.2.8 API Django	33
4.2.9 Base de datos PostGreSQL	34
4.3 Entorno de ejecución	34
4.3.1 Preparación de entorno virtual	34
4.3.2 Instalación y configuración de PostGreSQL	35
4.3.3 Instalación y configuración de Django	37
5 Implementación	43
5.1 Creación de Spiders	43
5.1.1 Proceso de obtención de datos	45
5.1.2 Guardado de datos	47
5.1.3 Spider básica	47
5.1.4 Método <code>start_requests()</code>	48
5.1.5 Eliminar Log	49
5.2 Spiders usadas	49
5.2.1 Aemet	49
5.2.2 CHCantábrico	51
5.2.3 MeteoNavarra	57
5.2.4 Agua en Navarra	61
5.3 Runners	69
5.4 Executers	70
5.5 Formateo de datos	70
5.5.1 Aemet	71
5.5.2 CHCantábrico	71
5.5.3 MateoNavarra	71
5.5.4 Agua en Navarra	71
5.6 Filtrado de datos	71
5.7 Posts	73
6 Conclusiones y Trabajo Futuro	75
Referencias	77

Contenidos

iii**Glosario** **81****Anexos** **81**

Lista de Figuras

1.1 Emisiones globales de CO ₂	2
1.2 Efecto invernadero	2
1.3 Media de aumento de temperatura global entre 1901-2000	3
1.4 Precipitaciones 2022	4
1.5 Estructura de datos planteada	6
2.1 Confederaciones hidrográficas en España	7
2.2 Página Nivel de los ríos CHCantábrico	8
2.3 Página Aemet de la estación en Aranguren (Navarra)	9
2.4 Página estaciones MeteoNavarra	11
2.5 Apartado selección de datos MeteoNavarra	12
2.6 Página de datos de MeteoNavarra	13
2.7 Página principal de aforos de El Agua en Navarra	14
2.8 Página estación de El Agua en Navarra	15
2.9 Gráfica de datos en estación de El Agua en Navarra	16
2.10 Error al cargar directamente la página de datos numéricos en El Agua en Navarra	16
2.11 Datos numéricos de estaciones en Agua en Navarra	16
3.1 Diagrama patrón MVT	22
3.2 Web Scraping (Krotov y Tennyson 2018)	24
3.3 Idea original de la estructura de datos planteada	26
4.1 Estructura de datos usada en el proyecto	27
4.2 Arquitectura de obtención y tratamiento de datos	30
4.3 Estructuración básica de una Spider	31
4.4 Tablas de la base de datos	34

4.5	Lista de tablas en PostGreSQL	39
5.1	Estructura del proyecto recién creado	43
5.2	Directorio de almacenamiento de las Spider	44
5.3	URL de inicio para obtener los códigos de las estaciones de Aemet	45
5.4	Ruta chromedriver.exe	65

Lista de Códigos

2.1	HTML tabla estaciones en CHCantábrico	8
2.2	HTML tabla datos en Aemet	9
2.3	HTML coordenadas en Aemet	10
2.4	HTML tabla estaciones en MeteoNavarra	11
2.5	HTML tabla datos en MeteoNavarra	13
2.6	HTML mapa estaciones en El Agua en Navarra	14
2.7	HTML estaciones en El Agua en Navarra	15
2.8	HTML datos en El Agua en Navarra	16
4.1	Flujo de datos	28
4.2	Modelos API Django	38
4.3	Definición URLs API	40
4.4	Configuración URLs API	40
4.5	Definición funciones de recepción y almacenamiento de datos API	41
5.1	Spider recién generada	44
5.2	Estructura HTML de los datos deseados Aemet	45
5.3	Guardar datos	47
5.4	Configurar guardado en JSON	47
5.5	Spider de ejemplo (Aemet Code Spider)	47
5.6	Sobre-escritura de <i>start_requests()</i>	48
5.7	Configurar LOG	49
5.8	Selector en <i>parse()</i> de Aemet Data Spider	50
5.9	Trabajar sobre los datos de Aemet Data Spider	50
5.10	Guardado de datos de Aemet Data Spider	50
5.11	Selector en <i>parse()</i> de CHCantábrico Code Spider	51
5.12	Trabajar sobre los datos de CHCantábrico Code Spider	51
5.13	Comprobar límites de CHCantábrico Code Spider	51

5.14 Guardado de datos de CHCantábrico Code Spider	51
5.15 Función <i>start_requests()</i> CHCantábrico Nivel Spider	52
5.16 Función <i>parse()</i> CHCantábrico Nivel Spider	53
5.17 Guardado de datos de CHCantábrico Nivel Spider	54
5.18 Selector en <i>parse()</i> de CHCantábrico Coordinates Spider	55
5.19 Guardado de datos de CHCantábrico Coordinates Spider	55
5.20 Script de obtención de datos pluviometricos descartado	55
5.21 Selector en <i>parse()</i> de MeteoNavarra Code Spider	57
5.22 Guardado de datos de MeteoNavarra Code Spider	57
5.23 Uso de fechas en función <i>start_requests()</i> MeteoNavarra Data Spider . .	58
5.24 Selector en <i>parse()</i> de MeteoNavarra Data Spider	58
5.25 Trabajar sobre los datos de MeteoNavarra Data Spider	59
5.26 Comprobacion exitencia de datos y guardado de MeteoNavarra Data Spider	59
5.27 Navegacion a segunda página de datos en MeteoNavarra Data Spider .	60
5.28 Selector en <i>parse()</i> de MeteoNavarra Coordenates Spider	60
5.29 Guardado de datos de MeteoNavarra Coordenates Spider	60
5.30 Función <i>parse()</i> Agua en Navarra Spiders	61
5.31 Función <i>parse_area()</i> Agua en Navarra Spiders	61
5.32 Función <i>parse_estacion()</i> Agua en Navarra Code Spider	62
5.33 Guardado de datos de Agua en Navarra Code Spider	62
5.34 Función <i>parse_estacion()</i> Agua en Navarra Data Spider	62
5.35 Selector en <i>parse_data()</i> de Agua en Navarra Data Spider	63
5.36 Selector en <i>parse_data()</i> de Agua en Navarra Data Spider	63
5.37 Agua en Navarra configuración Selenium	64
5.38 Configuración chromedriver Debian	66
5.39 Instalación Playwright	67
5.40 Configuración Playwright	67
5.41 Playwright basic Request	68
5.42 Agua en Navarra Playwright Request	68
5.43 CHCantábrico Data Runner	69
5.44 Aemet Data Runner	69
5.45 MeteoNavarra Data Runner	69
5.46 Ejecucion de entorno virtual y selección de proyecto Scrapy	70
5.47 Ejecución de <i>data_spider_runner.py</i>	70
5.48 Ejecución de <i>code_spider_runner.py</i>	70
5.49 Import necesarios filtrado de datos	71
5.50 Declaración rutas JSONs y nombre de fichero	71
5.51 Declaración función <i>openFile()</i>	71
5.52 Declaración función <i>saveFile()</i>	71
5.53 Declaración función <i>searchEstacionData()</i>	72
5.54 Declaración función <i>refinedData()</i>	72

5.55 Declaración rutas JSONs	73
5.56 Import necesarios post	73
5.57 Declaración variables code_post.py	73
5.58 Declaración variables data_post.py	74
5.59 Llamada POST	74

1

Introducción

Las Naciones Unidas definen el cambio climático como el conjunto de "*cambios a largo plazo de las temperaturas y los patrones climáticos*" [1]. Estos pueden ocurrir de forma natural cada cierto tiempo, ya sea debido a erupciones volcánicas, fluctuaciones de la radiación solar e, incluso, variaciones en la órbita terrestre, llegando a causar climatologías extremas como pueden ser las glaciaciones. Actualmente, a este proceso se le debe añadir como causa la actividad humana, alterando y agravando los efectos, siendo los principales causantes la contaminación, sobre-población y la deforestación.

Partiendo desde principios de la revolución industrial a mediados del siglo XVIII, el efecto de la actividad humana como causa de estos cambios ha ido en aumento, siendo desde el siglo XIX hasta llegar a pleno siglo XXI la principal causa del cambio climático. Debido al uso extendido de combustibles fósiles tales como el carbón, petróleo y el gas, formando las principales fuentes de energía durante muchos años. Figura 1.1.

La quema de estos produce los llamados gases de efecto invernadero, principalmente dióxido de carbono y metano. Estos gases impiden la correcta liberación de la radiación irradiada por el suelo al ser calentado por el sol, absorbiendo parte de esta y liberándola nuevamente hacia la tierra, aumentando la temperatura de la superficie terrestre. Como puede verse en la figura 1.2.

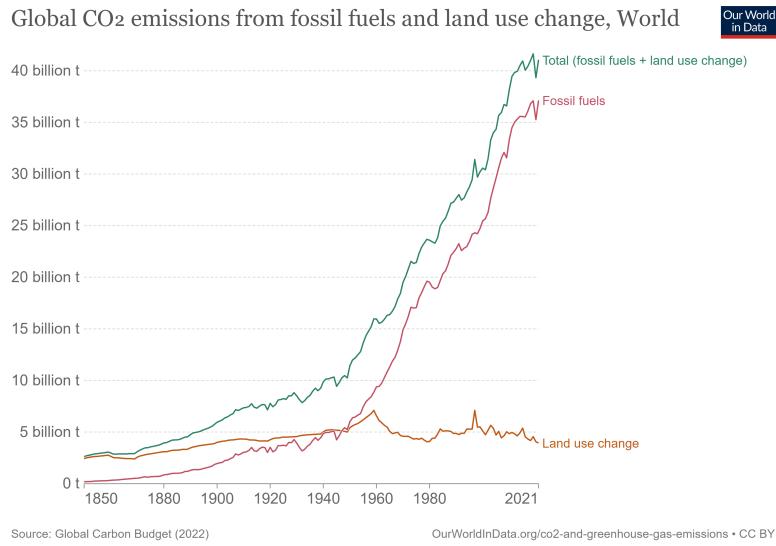


Figura 1.1: Emisiones globales de CO₂ ¹

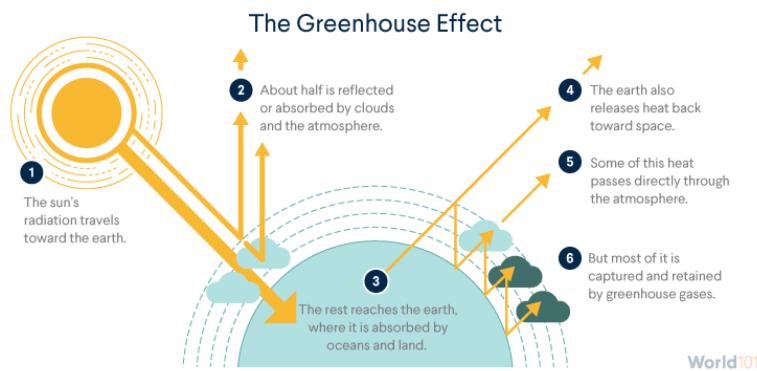


Figura 1.2: Efecto invernadero ²

El aumento de emisiones de estos gases, a fomentado una crecida sustancial en la velocidad de elevación de la temperatura terrestre. Llegando a medirse un aumento regional de, entre 0.8°C hasta 2°C en altas latitudes, estimando un aumento en la media global de 1.1°C en comparación con finales del siglo XIX [2].

Década tras década las temperaturas han ido aumentando, llegando a un punto en el que prácticamente anualmente se batén récords de temperatura máximas por todo el globo. Siendo globalmente el verano de 2023 el más caluroso registrado desde

¹ <https://ourworldindata.org/co2-emissions>

² <https://world101.cfr.org/global-era-issues/climate-change/greenhouse-effect>

1850. Figura 1.3 [3].

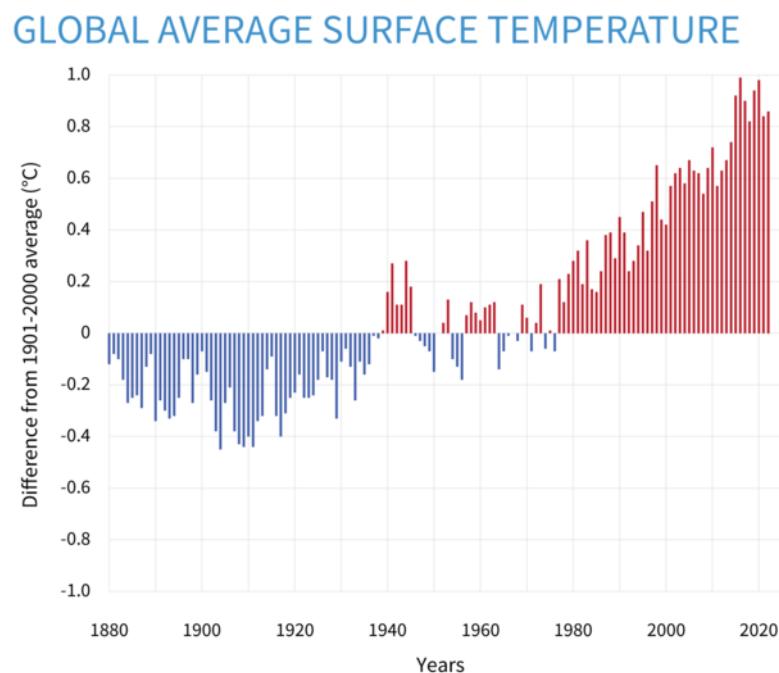


Figura 1.3: Media de aumento de temperatura global entre 1901-2000 ³

Son múltiples los efectos adversos causados por el aumento de la temperatura, entre ellos, la subida del nivel marino, reduciendo e inundando zonas costeras, la desertificación de zonas actualmente áridas y la alteración del comportamiento de especies tanto animales como vegetales, llegando a suponer una reducción sobre la población en multitud de especies [4] [5].

Es por eso que múltiples países son ya los que optan por comprometerse a alcanzar una cota de emisiones cero para 2050, tratando de reducir las emisiones globales a la mitad cara 2030, con el fin de mantener el aumento de la temperatura media por debajo de 1.5°C , estimando esta como un punto reflexivo a la hora de controlar el impacto climático, con el objetivo de mantener un clima habitable.

Pero el problema no reside únicamente en el aumento de la temperatura, como da a entender la definición proporcionada por la ONU, llegando a influenciar sobre los patrones climáticos. Empezando por las estaciones, se ha llegado a observar fluctuaciones en estas, anticipando de la llegada de la primavera, prolongando el verano, retrasando el otoño y reduciendo en la temporada invernal [6].

³<https://www.climate.gov/news-features/understanding-climate/climate-change-global-temperature>

Sumado a esto, son cada vez más los efectos visibles causados sobre los fenómenos naturales. Es tal el impacto que, ya son más de veinte las anomalías climáticas de una importancia significante remarcadas por el centro nacional de información ambiental (NCEI) en 2022 [3]. Entre ellas, sequías debido al aumento de las temperaturas, llegando a fomentar la aparición de incendios y, fuertes lluvias, marcadas por el aumento de tormentas, huracanes y tifones durante las estaciones lluviosas, causando múltiples inundaciones.

El cambio climático agrava los efectos de los fenómenos naturales, reduciendo su periodo de retorno (periodo estimado de años para la aparición de un fenómeno climático), causante de esta meteorología extrema. Zonas planetarias enteras se ven azotadas por graves sequías y tormentas, modificando la disponibilidad del agua sobre la superficie terrestre, cosa que, afecta severamente a la agricultura. La precipitación acumulada se mantiene debajo de la media globalmente, mientras que la frecuencia de aparición de precipitaciones anómalas va en aumento, figura 1.4, fomentadas por una mayor evaporación de agua a causa del aumento de la temperatura, agravando la frecuencia de tormentas de magnitudes extremas.

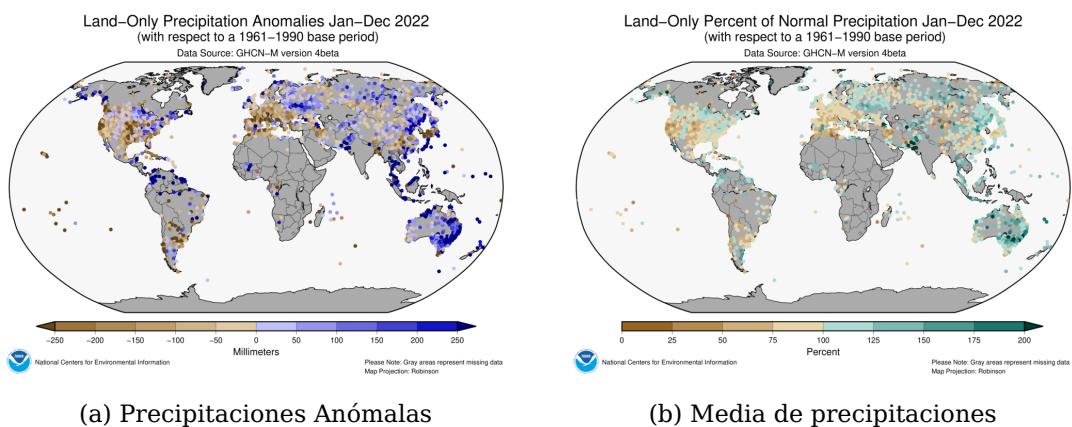


Figura 1.4: Precipitaciones 2022

Estos efectos no tienen por que darse independientemente, sin ir más lejos, en 2022, España batió el récord como tercer año más seco registrado, solo superado por los años 2005 y 2017, más tarde, ese mismo año en diciembre, fuertes lluvias causaron múltiples inundaciones provocando daños materiales sobre viviendas y carreteras [7].

Las inundaciones causadas por lluvias torrenciales son una de las principales causantes de daños materiales a nivel global [8], estimando una perdida anual de 800 millones de euros tan solo en España [9]. Es por eso que, la monitorización y predicción climática son primordiales a la hora de procurar reducir su efecto a nivel económico

como humano.

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/iden>) en Navarra o los datos o el sistema automático de información publicada por las diferentes confederaciones hidrográficas en España.

A fin de obtener datos tanto fluviales como pluviométricos, disponiendo de los presentes en las siguientes agencias y organismos.

La Confederación Hidrográfica del Cantábrico (<https://www.chcantabrico.es/>) es un organismo autónomo adscrito al Ministerio para la Transición Ecológica y el Reto Demográfico. Es responsable de la gestión de las cuencas hidrográficas de los ríos que vierten al mar Cantábrico, ejerciendo sobre las comunidades autónomas de: Principado de Asturias, Cantabria, Castilla y León, Galicia, País Vasco y Comunidad Foral de Navarra.

La Agencia Estatal de Meteorología (AEMET) (<https://www.aemet.es/es/portada>), también adscrita al Ministerio para la Transición Ecológica y el Reto Demográfico, tiene como objetivo el desarrollo, implantación, y prestación de los servicios meteorológicos del Estado, apoyando al ejercicio de otras políticas públicas y actividades privadas, contribuyendo a la seguridad de personas y bienes, y al bienestar y desarrollo sostenible de la sociedad.

Meteo Navarra (<http://meteo.navarra.es/>), es la agencia encargada de la monitorización de la meteorología y climatología de Navarra, trabaja junto al gobierno de Navarra, el Ministerio de Agricultura, Pesca y Alimentación (MAPA), el Instituto Navarro de Tecnologías e Infraestructuras Agroalimentarias (INTIA), la Agencia Estatal de Meteorología (AEMET) y la Universidad Pública de Navarra (UPNA).

El agua en Navarra (http://www.navarra.es/home_es/Temas/Medio+Ambiente/Agua/), otra plataforma del gobierno de Navarra, esta vez centrada en los recursos hídricos disponibles en Navarra.

Con el fin de posibilitar la creación un modelo de predicción de inundaciones sobre los ríos en Navarra, el objetivo de este trabajo es crear una plataforma habilitada en la obtención y centralizando de la información proveniente de estas diferentes fuentes,

figura 1.5. Mediante esta plataforma se pretende disponer de un conglomerado de datos sobre los que trabajar con el propósito de prevenir sobre inundaciones a la población navarra por medio de un sistema de notificaciones automáticas.

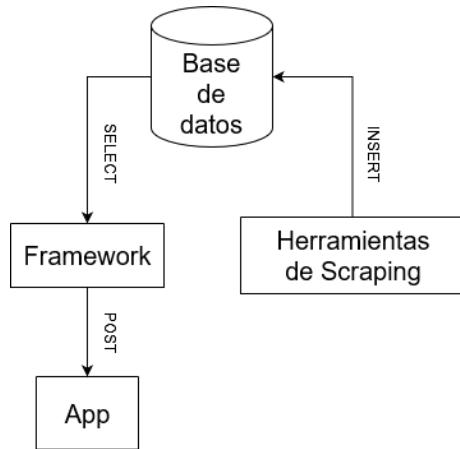


Figura 1.5: Estructura de datos planteada

Este trabajo pretende obtener la mayor cantidad de datos posibles con el fin de predecir dichas inundaciones causadas por las crecidas de los ríos, de modo que las administraciones y la población puedan actuar con tiempo y reducir el efecto devastador y económico producido por estos efectos. Como prueba de concepto trabajaremos a nivel regional en España. En concreto, la Comunidad Foral de Navarra.

El resto de las memoria se divide de la siguiente manera, la Sección 2 presenta las fuentes de obtención de datos y los datos necesarios para realizar el proceso de predicción; la Sección 3 presenta las tecnologías necesarias para realizar la arquitectura planteada; en la Sección 4 se ahonda en el diseño de la arquitectura, explicando los componentes de esta y como preparar el entorno para realizarla; la Sección 5 presenta y explica el código implementado con el fin de crear la plataforma. Finalmente, en la Sección 6 se presentan las conclusiones y el trabajo futuro.

2

Datos

Navarra dispone de dos confederaciones hidrográficas, la del cantábrico y la del ebro. Estas son organismos encargados de la gestión de cuencas hidrográficas que discurren por múltiples comunidades autónomas. Tomando como cuenca hidrográfica a la superficie por la cual fluye un conjunto de agua, como ríos, arroyos y lagos hasta su desembocadura en el mar.

La del cantábrico ejerce sobre aquellos ríos cuya desembocadura va a parar al cantábrico, mientras, la del ebro trabaja sobre las zonas limitadas al cauce del ebro.

No se hace uso de la página de la confederación del ebro al disponer de datos de otras páginas como la de la agencia estatal de meteorología y, la de agua en navarra.



Figura 2.1: Confederaciones hidrográficas en España

Creadas en 1926, las confederaciones son organismos autónomos, con plena autonomía funcional, adscritas al Ministerio para la Transición Ecológica y el Reto De-

mográfico. El papel desempeñado por estas recae entre otras cosas en, la planificación hidrológica, gestión de recursos y aprovechamientos, protección del dominio público hidráulico, control de calidad del agua y los bancos de datos.

Por todo esto, es de gran importancia para este proyecto la adquisición de datos relacionados a estas entidades que ejercen en Navarra.

2.1 CHCantábrico

En la sección de nivel del río de la pagina de CHCantábrico, se encuentra la tabla con las estaciones, figura 2.2. Una peculiaridad de esta, es que aun seguir una estructuración típica de tabla mediante el uso de *table*, dispone de una tabla adicional en cada una de las filas.



Figura 2.2: Página Nivel de los ríos CHCantábrico

```

1  <table class="tablefixedheader niveles">
2    <thead></thead>
3    <tbody>
4      <tr class="centrado">
5        <td class="codigo">1207</td>
6        <td class="verde">...</td>
7        <td class="verde">...</td>
8        <td class="valor">...</td>
9        <td class="tendencia">...</td>
10       <td class="actualizacion">...</td>
11       <td>
12         <table class="umbrales_gr">
13           <tbody></tbody>
14         </table>
15       </td>
16       <td>...</td>
17     </tr>
18   ...
19   </tbody>
20 </table>
```

Código 2.1: HTML tabla estaciones en CHCantábrico

A diferencia del resto de las páginas visitadas, CHCantábrico no muestra los datos sobre la web, por el contrario, implementa un botón con el que descargarlos directamente.

mente en formato CSV. Los datos que se pueden obtener, son aquellos presentes en la tabla secundaria de las fila, los datos de pre-alerta, alerta y seguimiento para cada una de las estaciones, siendo datos valiosos a la hora de intentar anticipar inundaciones. Las coordenadas a su vez, tampoco vienen dados sobre la web, por lo que hace falta visitar la web del Centro de Estudios Hidrológicos (<https://ceh.cedex.es/>) para obtenerlas.

2.2 Agencia estatal de Meteorología (Aemet)

La figura 2.3 muestra la página web de Aemet, dentro 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 no son tan relevantes para este proyecto. A su vez, no todas las estaciones muestran estos datos, ocurriendo lo mismo con los de presión atmosférica.



Figura 2.3: Página Datos Aemet

```

1   <table id="table">
2     <thead></thead>
3     <tbody>
4       <tr class="fila_par">
5         <td class="evenselected">27/08/2023 12:00</td>
6         <td class="">16.6</td>
7         <td class="">16</td>
8         <td class="">...</td>
9         <td class="">36</td>
10        <td class="">...</td>
11        <td class="">0.0</td>
12        <td class=""></td>
13        <td class=""></td>
14        <td class="">67.0</td>
15      </tr>
16      ...
17    </tbody>
18  </table>

```

Código 2.2: HTML tabla datos en Aemet

Los datos en HTML vienen dados dentro de un elemento *table*, lo que facilita su adquisición, pues permite tomar todas las filas (elementos *tr*) pertenecientes al elemento *tbody* de esta. Una vez obtenidas las filas, la forma de lograr los datos deseados sería elegir dentro de cada elemento *tr* los elementos *td* deseados, puesto que HTML empieza a contar elementos desde el uno (en vez de cero como suele ser común en lenguajes de programación), los *td* a obtener serían: uno, fecha y hora; dos, temperatura; siete, precipitación; diez, humedad.

```
1 <span class="geo">
2   <span class="font_bold">Latitud</span>
3   :
4   <abbr class="latitude" title="42.77611111111111">42° 46' 34'' N</abbr>
5   -
6   <span class="font_bold">Longitud</span>
7   :
8   <abbr class="longitude" title="-1.5325000000000001">1° 31' 57'' O</abbr>
9   -
10 </span>
```

Código 2.3: HTML coordenadas en Aemet

Las coordenadas, dentro de un *span*, están incluidas en elementos *abbr* con sus respectivas clases “*latitude*” y “*longitude*”, haciendo posible su obtención fácilmente mediante los elementos *abbr*.

2.3 Meteorología y climatología de Navarra (MeteoNavarra)

En la página de meteorología y climatología de Navarra encontramos una gran sección de estaciones de las cuales poder obtener datos. Figura 2.4.

Consultar datos

Haga clic sobre una estación del mapa o de la lista para consultar sus datos.

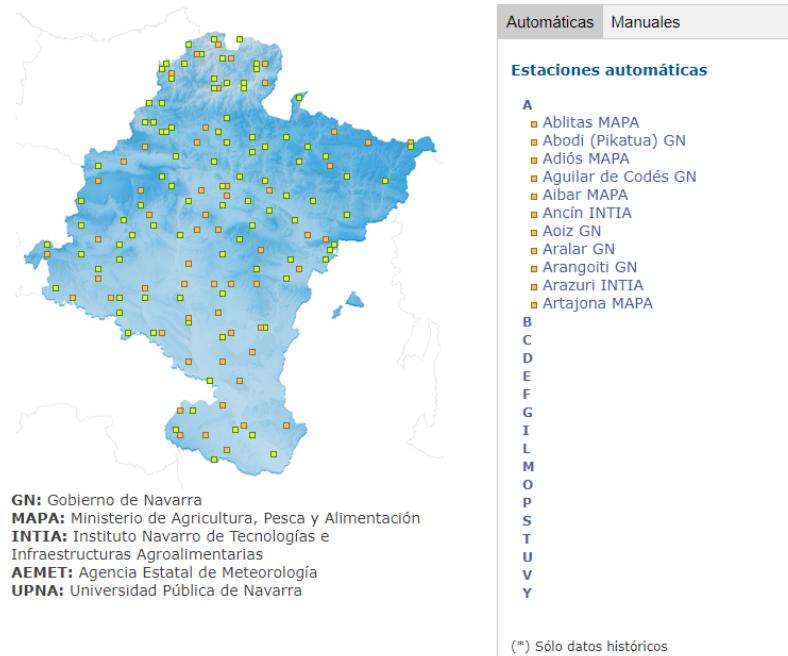


Figura 2.4: Página estaciones MeteoNavarra

De entre los dos tipos e estaciones disponibles, automática y manual, únicamente se va a trabajar con los datos de las estaciones automáticas. El motivo de esto es que las manuales solo proveen datos diarios de, la temperatura máxima, mínima y de la precipitación acumulada. Esto hace que no se disponga 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.

```

1 <table width="260" border="0" cellspacing="0">
2   <tbody>
3     <tr>
4       <td colspan="2" height="170" valign="top">
5         <div style="margin-left:5px; margin-right:5px;z-index:200">
6           <div onclick="javascript:menu_openclose('AUTO_A'); " style="width:100px; "></div>
7           <div id="AUTO_A" style="margin-left:5px;display:none;">
8             <script>
9               dolimage(274,6,'/img/2/estacionautomatica.gif','Abilitas MAPA','AUTO');
10              </script>
11              
12              
13              <a href="estacion.cfm?IDestacion=274" onmouseover="showEstacion('274_info','6','AUTO')"
14                onmouseout="hideEstacion('274_info','AUTO')> Abilitas MAPA</a>
15              ...
16            </div>
17            ...
18          </div>
19          <br><br>
20        </td>
21      </tr>
22    </tbody>
```

Código 2.4: HTML tabla estaciones en MeteoNavarra

Debido a la estructuración HTML usada para mostrar las estaciones, se puede ver que, aun usar el elemento `table`, este solo dispone de una única fila y columna, haciendo de la columna un elemento `div` sobre el que se insertan los datos.

Las estaciones automáticas proporcionan tanto datos en periodos de diez minutos como datos diarios. Tras usar el mismo razonamiento que con las estaciones manuales, no tomamos los datos diarios y, entre los actualizados cada diez minutos, se toman la temperatura, humedad relativa, radiación global y precipitación. El resto se omitirán. Figura 2.5.

1. Parámetros	
Parámetros 10 minutos	Parámetros Diarios
<input checked="" type="checkbox"/> Temperatura <input checked="" type="checkbox"/> Humedad relativa <input checked="" type="checkbox"/> Radiación global <input type="checkbox"/> Insolación <input checked="" type="checkbox"/> Precipitación <input type="checkbox"/> Velocidad viento 10 m <input type="checkbox"/> Dirección viento 10 m <input type="checkbox"/> Velocidad racha máxima 10 m <input type="checkbox"/> Dirección racha máxima 10 m <input type="checkbox"/> Velocidad viento 2 m <input type="checkbox"/> Dirección viento 2 m <input type="checkbox"/> Velocidad racha máxima 2 m <input type="checkbox"/> Dirección racha máxima 2 m <input type="checkbox"/> Desviación dirección viento 10 m <input type="checkbox"/> Desviación dirección viento 2 m <input type="checkbox"/> Desviación velocidad 10 m <input type="checkbox"/> Desviación velocidad 2 m <input type="checkbox"/> Humección hoja (resistencia) <input type="checkbox"/> Radiación Solar PAR acumulada 10min <input type="checkbox"/> Temperatura suelo	<input type="checkbox"/> Temperatura media <input type="checkbox"/> Temperatura máxima <input type="checkbox"/> Temperatura mínima <input type="checkbox"/> Humedad relativa med. <input type="checkbox"/> Humedad relativa máx. <input type="checkbox"/> Humedad relativa mín. <input type="checkbox"/> Precipitación acumulada <input type="checkbox"/> Radiación global <input type="checkbox"/> Insolación total <input type="checkbox"/> Velocidad media viento 10 m <input type="checkbox"/> Dirección viento 10 m (MODA) <input type="checkbox"/> Velocidad racha máx 10 m <input type="checkbox"/> Dirección racha máx 10 m <input type="checkbox"/> Velocidad media viento 2 m <input type="checkbox"/> Dirección viento 2 m (MODA) <input type="checkbox"/> Velocidad racha máx 2 m <input type="checkbox"/> Dirección racha máx 2 m <input type="checkbox"/> Radiación PAR diaria acumulada <input type="checkbox"/> Temperatura media suelo
Datos en horario solar .	
2. Fechas	
hoy	ayer
Desde	<input type="text"/>
Hasta (excluido)	<input type="text"/>

Figura 2.5: Datos MeteoNavarra

Los datos se muestran en formato tabla, tanto visualmente, en la figura 2.6 como a nivel de HTML, cosa que facilita su obtención.

Estación de Ablitas MAPA

Altitud 338 m.

Datos desde el 20/08/2023 hasta el 21/08/2023 en **horario solar**
 Los datos son PROVISIONALES

Fecha	Temperatura		Humedad relativa		Radiación global	Precipitación
	°C	%	W/m ²	l/m ²		
20/08/2023 0:00	23.6	55	0.0	0.0		
20/08/2023 0:30	22.9	52	0.0	0.0		
20/08/2023 1:00	22.9	56	0.0	0.0		
20/08/2023 1:30	22.9	62	0.0	0.0		

Figura 2.6: Página de datos de MeteoNavarra

```

1 <table class="border" border="0" cellpadding="3" cellspacing="1" bgcolor="#FFFFFF">
2   <tbody>
3     <tr bgcolor="#FFFFFF"></tr>
4     <tr bgcolor="#FFFFFF"></tr>
5     <tr bgcolor="#E82FFF">
6       <td>26/08/2023 0:00</td>
7       <td align="center">
8         <font color="0">17.6</font>
9       </td>
10      <td align="center">
11        <font color="0">82</font>
12      </td>
13      <td align="center">
14        <font color="0">0.0</font>
15      </td>
16      <td align="center">
17        <font color="0">0.0</font>
18      </td>
19    </tr>
20    ...
21  </tbody>
22 </table>
```

Código 2.5: HTML tabla datos en MeteoNavarra

2.4 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 períodos de 10 minutos, siendo una de las fuentes principales de datos.

La sección de aforos en la página, figura 2.7, muestra el mapa de Navarra junto a varias estaciones. Aunque no todas las disponibles en la web, conformando únicamente el grupo de estaciones principales.

Con el fin de acceder a todas las estaciones disponibles, debemos centrarnos en el mapa de la esquina inferior derecha. Estructurada en 6 regiones, Norte, Arga, Ega, Ebro alto, Ebro bajo y Aragón, por medio de este mapa accedemos a cada región, mostrando el mapa de la región, dando acceso a todas las estaciones en la zona.

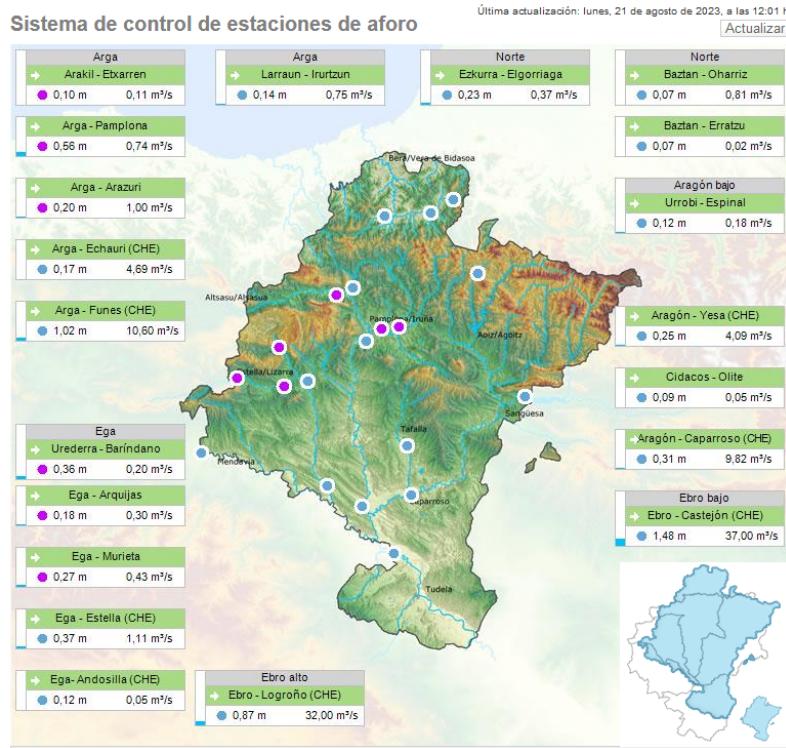


Figura 2.7: Página El Agua en Navarra

```

1 <map id="mapCajetines_mapa" name="mapCajetines">
2   <area shape="rect" coords="5,5,180,59" alt="Arakil — Etxarren"
3     href="https://administracionelectronica.navarra.es/aguaEnNavarra/ctaDatosEstacion.aspx?IdEstacion=50"
4     onmouseover="javascript:document.getElementById('imgMapa').src='mostrarImagen.aspx?idMapa=1&"
5     "amp;IdCajetin=62&Vista=internet&IdOrigenDatos=1'" target="_self"
6     onmouseout="javascript:document.getElementById('imgMapa').src='mostrarImagen.aspx?idMapa=1&"
7     "amp;Vista=internet&IdOrigenDatos=1'">
8   <area shape="circle" coords="317,243,11" alt="Arakil — Etxarren"
9     href="https://administracionelectronica.navarra.es/aguaEnNavarra/ctaDatosEstacion.aspx?IdEstacion=50"
  onmouseover="javascript:document.getElementById('imgMapa').src='mostrarImagen.aspx?idMapa=1&"
  "amp;IdCajetin=62&Vista=internet&IdOrigenDatos=1'" target="_blank"
  onmouseout="javascript:document.getElementById('imgMapa').src='mostrarImagen.aspx?idMapa=1&"
  "amp;Vista=internet&IdOrigenDatos=1'">
...
</map>
```

Código 2.6: HTML mapa estaciones en El Agua en Navarra

El HTML del mapa se estructura de la siguiente manera, mostrando un par de elementos `area` por estación, uno con `shape="rect"`, representando las tarjetas que rodean el mapa y otro con `shape="circle"`, siendo los puntos en el mapa. Pulsar sobre cualquiera de ellos es equivalente, aunque posteriormente hagamos uso de los elementos con `shape="rect"` dentro del código.

Las páginas de cada estación se muestran como en la figura 2.8.



Figura 2.8: Página estación El Agua en Navarra

```

1 <div id="bloq_iconos">
2   <div id="descripcion">
3     Descripción
4     <br>
5     <span class="tit_icon">
6       <span id="lblDescripcion">Urederra en Barndano</span>
7     </span>
8   </div>
9   <div id="municipio">
10    Municipio
11    <br>
12    <span class="tit_icon">
13      <span id="lblMunicipio">Amescoa Baja</span>
14    </span>
15  </div>
16  <div id="rio">
17    Río
18    <br>
19    <span class="tit_icon">
20      <span id="lblRio">Urederra</span>
21    </span>
22  </div>
23  <div id="coordenadas_UIM">
24    Coordenadas UIM (EPSG:25830)
25    <br>
26    <span class="tit_icon">
27      <span id="lblUTM">X. 571569,5 | Y. 4735145 | Z. 500</span>
28    </span>
29  </div>
30 </div>

```

Código 2.7: HTML estaciones en El Agua en Navarra

De ella se obtienen los próximos datos, descripción, municipio, río y coordenadas. Ademas de dar acceso a los datos de nivel y caudal del río. Mostrados dentro de el div con `id="blog_icons"`. Una vez dentro del div todos los datos siguen la misma estructuración, `//div/span/span`, permitiendo la adquisición de todos los elementos a la vez.

Una vez se accede a los datos de la estación, se mostrara una gráfica como la de la figura 2.9. A su vez, mediante el botón "Datos Numéricos", tendremos la posibilidad de observar los datos en forma numérica. Pero no sin antes haber visitado la versión gráfica. Pues da el error de la figura 2.10.



Figura 2.9: Gráfica datos estación El Agua en Navarra



Figura 2.10: Error datos numéricos en El Agua en Navarra

Los datos numéricos están presentados en formato tabla como se aprecia en la figura 2.11. Por el contrario, tras observar el HTML, realmente es un elemento *div*, que engloba los conjuntos de elementos *span* que representan las líneas, separados por elementos *br*. El formato en el que se presentan los datos, no hace más que representar una mayor complejidad para, posteriormente, trabajar y adquirir los datos.

Datos de Nivel Río (últimos 15 días)			Última actualización: lunes, 21 de agosto de 2023, a las 13:10 h.
			Exportar a Excel Volver
Fecha	Valor	Unidad	
08/08/2023 14:40	0,03	m	
08/08/2023 14:50	0,03	m	
08/08/2023 15:00	0,03	m	
08/08/2023 15:10	0,03	m	
08/08/2023 15:20	0,03	m	

Figura 2.11: Datos numéricos de estaciones en Agua en Navarra

```

1 <div id="lista">
2   <span class="cont_fecha_gra">
3     14/08/2023 16:10
4   </span>
5   <span class="cont_valor_gra">0,36</span>
6   <span class="cont_unidad_gra">m</span>
7   <span class="cont_vacio_gra"></span>
8   <br>
9   ...

```

10 | </div>

Código 2.8: HTML datos en El Agua en Navarra

De las webs mencionadas se dispone de los siguientes datos para este proyecto:

- Nivel (m)
- Caudal (m^3/s)
- Precipitación (mm)
- Temperatura (°C)
- Humedad (%)
- Radiación (W/m^2)

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

Para este proyecto, es necesario el uso de dos maquinas virtuales para crear los servidores necesarios, un servidor de base de datos y otro para el despliegue de una API y la obtencion de datos. 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 más vulnerable, ademas, a nivel de proyecto, proporciona un mayor grado de complejidad, necesitando configurar la comunicación por red de estas.

Primero que todo es necesario un sistema operativo capaz de servir este propósito.

3.1 Sistema Operativo Debian

En 1993, tras varios intento 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 [13] [15].

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 [16] [17].

A su vez, debido al apoyo del software libre, hace uso de múltiples licencias de soft-

ware 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. [18]

Definido por estas licencias, el software libre debe cumplir al menos las que están consideradas la cuatro libertades esenciales: [19]

- Ejecutar el programa como se deseé, con cualquier propósito.
- Estudiar cómo funciona el programa, y cambiarlo para que haga lo que se deseé.
- 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.

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. [23]

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.

Puesto que no afecta negativamente al proyecto no disponer de las versiones más recientes de software y debido a la necesidad de un sistema 24/7 gratuito, parece

lógica la elección de Debian como sistema operativo para usar en el servidor.

Una vez seleccionado sistema operativo, hace falta seleccionar un Framework para crear y trabajar con una API.

3.2 Framework necesarios

Un framework es software que provee una infraestructura básica sobre la que desarrollar tus proyectos. Aportan 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 [24].

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: Full Stack, Micro y Asíncrono [25].

Full-stack es un tipo de framework 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 es un tipo de framework ligero 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 relevantes, el 'micro' de microframework significa que el núcleo del framework es simple pero extensible.

Asíncrono es un tipo de framework dirigido por eventos. En vez de hacer un manejo operacional linea a linea 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.

3.2.1 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, 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, sin comprometer la flexibilidad de desarrollar las funcionalidades necesarias para su funcionamiento, aportando herramientas sobre las que trabajar. [26]

3.2.2 Django Framework

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. [27] [28]

Hace uso de su auto denominado patrón Modelo-Vista-Template (MVT) 3.1 una variante del conocido Modelo-Vista-Controlador (MVC). [29]

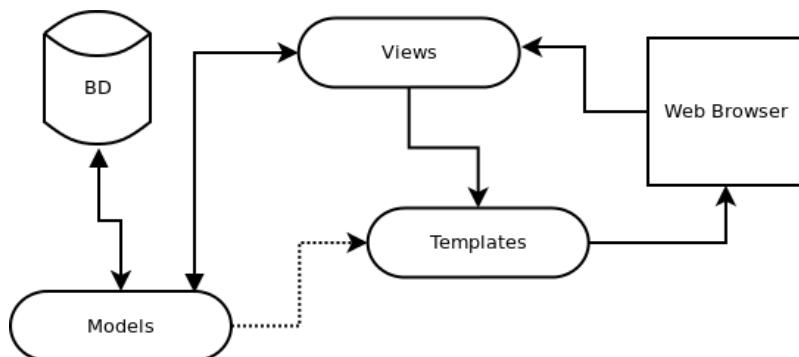


Figura 3.1: Diagrama patrón MVT¹

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

Django es uno de los frameworks más reputados en Python, 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.

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

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 decantarse por Django, aunque no sea única de él, es su compatibilidad con bases de datos relacionales de forma nativa, cosa que facilitará el uso de estas en el proyecto.

A su vez, con el fin de obtener los datos, se necesita de una herramienta centrada en el scraping de datos.

3.3 Herramienta para 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 [30] [31].

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.

3.3.1 Procedimiento básico en Web Scraping

El proceso de recolección de datos de Internet se puede dividir en tres procesos secuenciales, el estudio de la página sobre la que trabajar, adquirir los recursos web y, luego, organizar la información deseada. [3.2](#)

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.

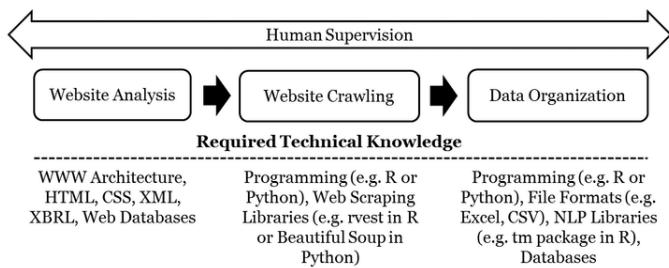


Figura 3.2: Web Scraping ²

3.3.2 Scrapy Framework

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, procesado y estructurado de los datos adquiridos, además de dar la opción de extender funcionalidades en caso necesario, haciéndolo extremadamente versátil. [32]

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

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

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

La segunda por el contrario, no es una herramienta de 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 si que ha

²https://www.researchgate.net/figure/Web-Scraping-Adapted-from-Krotov-and-Tennyson-2018_fig1_324907302

sido usada junto con Scrapy para navegar entre webs y sacar los datos de estas. Proporcionando la posibilidad de hacer uso de JavaScript sobre las páginas, pues Scrapy no dispone de renderizado JavaScript de forma nativa.

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 deseas. 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, la naturaleza de Scrapy le hace tener el mejor rendimiento de entre las tres. [26]

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

Para satisfacer el segundo servicio necesario se requerirá de una base de datos.

3.4 Base de Datos

Debido a la naturaleza de los JSON obtenidos, se pueden trasladar fácilmente a tablas relacionales, por lo que se hará uso de una base de datos relacional.

3.4.1 PostGreSQL

PostGreSQL, es un sistema de gestión de bases de datos relacional de código abierto. PostgreSQL destaca por su sistema de gestión de bases de datos, su soporte para consultas complejas, transacciones ACID, integridad referencial y escalabilidad.

Además, ofrece una amplia gama de tipos de datos, incluyendo geoespaciales y JSONB, almacenando JSONs de forma binaria (de ahí la B) para su fácil acceso.

La comunidad activa detrás de PostgreSQL contribuye continuamente con mejoras y extensiones, lo que lo convierte en una solución versátil y confiable. Lo que la hace popular tanto para aplicaciones empresariales como para proyectos web.

Entre las bases de datos con soporte oficial en Django se encuentran, PostgreSQL, MariaDB, MySQL, Oracle y SQLite. A excepción de Oracle, la única base de datos comercial, el resto son de código abierto, aunque puedan disponer de versiones de pago.

A nivel de funcionalidades, todas ofrecen implementadas de una forma u otra una misma gama de estas. Es por eso que, la elección se hizo en base a la flexibilidad y robustez de estas. Lo que hace a PostGreSQL una opción interesante en proyectos que lleguen a requerir de datos geoespaciales y JSON.

3.4.2 Arquitectura Preliminar

Tras la selección de las herramientas a usar, la arquitectura queda de las manera mostrada en la figura 3.3. Como herramienta de scraping, el framework Scrapy; como base de datos PostGreSQL y, como servicio API el framework Django.

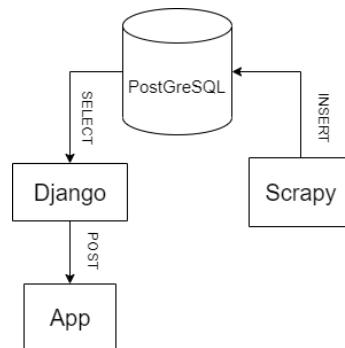


Figura 3.3: Estructura descartada

El hecho de seleccionar estos Framework, impone la necesidad de usar como lenguaje de programación 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" [10]. Python a su vez es un lenguaje multiplataforma, fuertemente tipado, dinámico y multiparadigma, pues soporta programación orientada a objetos, imperativa y funcional [11] [12].

4

Diseño de la Plataforma

La arquitectura planteada en la figura 3.3 es simple y fácil de implementar, pues scrapy mismo puede insertar los datos scrapeados de las web de forma directa en PostGreSQL, aunque presenta un problema. Djando 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 no serán 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 anteriormente pedidos son la versión mas reciente, luego no es necesario realizar llamada alguna a la BBDD.

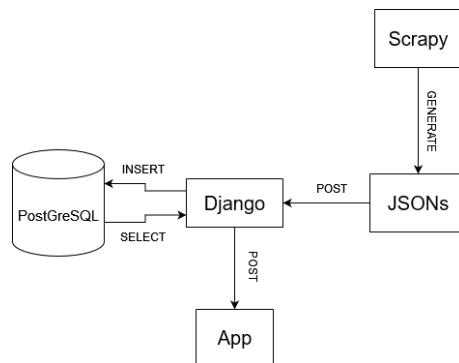


Figura 4.1: Estructura usada

Pasa solventar el problema, inicialmente se encontró el plugin scrapy_djangoitem, promete la comunicación entre scrapy y Django, pero se decidió no usarlo al llevar siete años sin revisiones. Como alternativa, se ha optado por la arquitectura de la figura 4.1, que incluye un paso intermedio al almacenar 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.

4.1 Trabajar con los Datos

```

1 Pedir datos
2 Formatear datos
3 Mandar datos a la BBDD
4 Marcar datos como old

5
6 While(true)
7   Pedir datos
8   Formatear datos
9   Filtrar datos
10  Mandar datos a la BBDD
11  Marcar datos como old

```

Código 4.1: Flujo de datos

El flujo que realizan los datos es el siguiente, inicialmente se pedirán los datos, como los datos obtenidos no son los mismos para cada web, se formatean para compartir una estructura heterogénea, en caso de ser la primera iteración, se mandan directamente los datos a la BBDD, de no ser la primera iteración, se filtran los nuevos datos de aquellos marcados como old (viejo), se envían los datos a la base de datos y, una vez enviados son marcados como old para ser el punto de comparación respecto a los datos que se obtendrán en la próxima iteración.

4.1.1 Datos obtenidos por página

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.

4.1.2 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.

Debido a ello, para cada web se ha creado una función para parsear (formatear) los datos obtenidos, de esta manera se dispone de una estructura única para los datos recibidos, haciendo su uso posterior más fácil, ya sea a la hora de tratarlos 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]
```

4.1.3 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 únicamente las instancias nuevas de los datos recogidos, pues no disponemos de ninguna manera de filtrar los datos a la hora de obtenerlos. Estos datos serán guardados en un tercer JSON.

4.2 Arquitectura

Para poder realizar el trabajo se ha diseñado la siguiente arquitectura.[4.2](#)

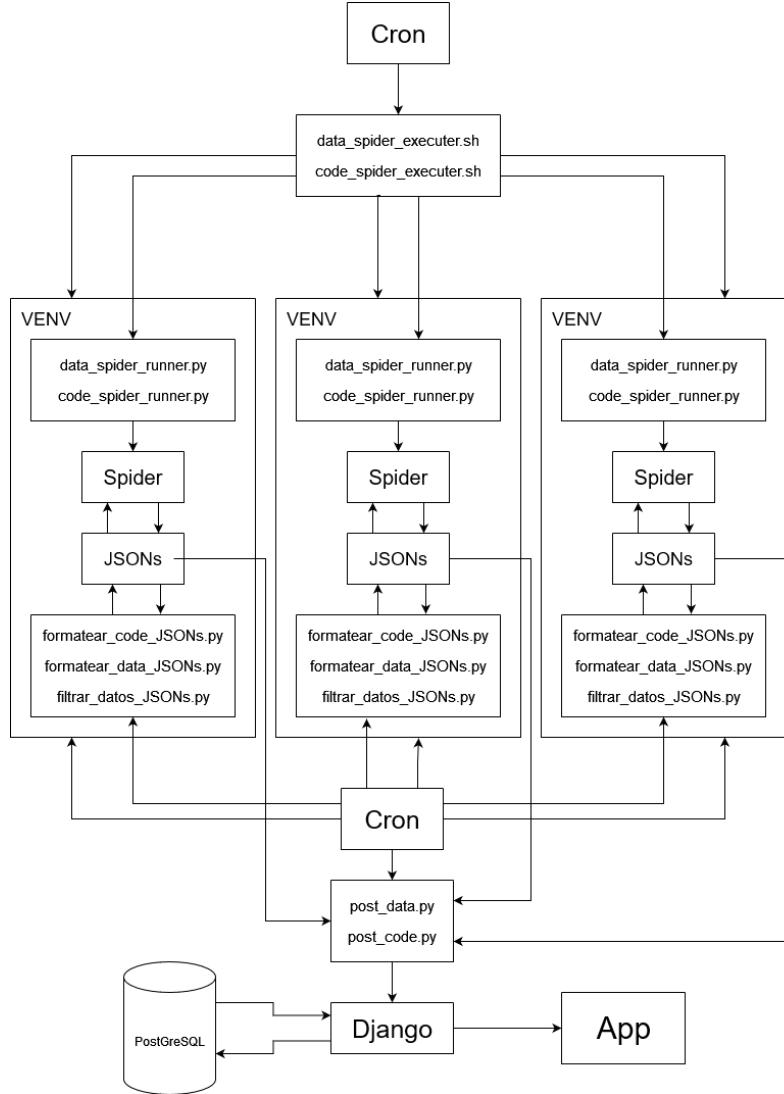


Figura 4.2: Arquitectura de obtención y tratamiento de datos

4.2.1 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.

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

4.2.2 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, se dividen las tareas con la posibilidad de ejecutar aquella que mejor venga en cada momento.

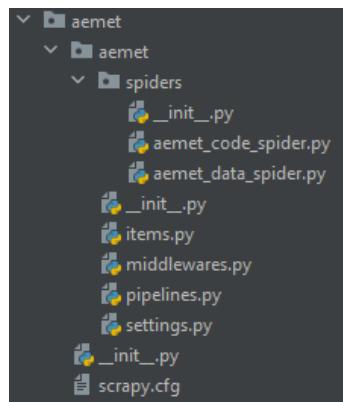


Figura 4.3: Estructuración básica de una Spider

4.2.3 Runners

Runner es la forma en la que han sido nombrados los Scripts cuya función es permitir 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.

4.2.4 Executers

Aumentando un poco mas 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 pasandoles un único argumento representando la web de la que quieras obtener los datos, haciendo que la agregación de nuevas Spiders junto con sus entornos sea tan sencillo como respetar las rutas y nombres predefinidos.

4.2.5 Directorios de los JSONs de datos

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:

Directorios de los datos de estación:

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

Directorios de los códigos de estación:

1. RawCode
2. ParsedCode
3. RefinedCode
4. OldCode

El primer nivel es aquel que almacena los JSON que genera la llamada con la Spider a la web. El segundo, los resultantes 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 almacena los ficheros generados 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 guarda el fichero original ya formateado tras la comparación.

4.2.6 Posts

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

Diseñados bajo el mismo principio de fomentar la ampliabilidad del proyecto que los Executer, 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.

4.2.7 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.

4.2.8 API 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 una aplicación de predicción, se encargaría de pedir los datos almacenados y enviarlos mediante POST a la aplicación.

4.2.9 Base de datos PostGreSQL

La base de datos, dispone de las siguientes dos tablas de la figura 4.4.

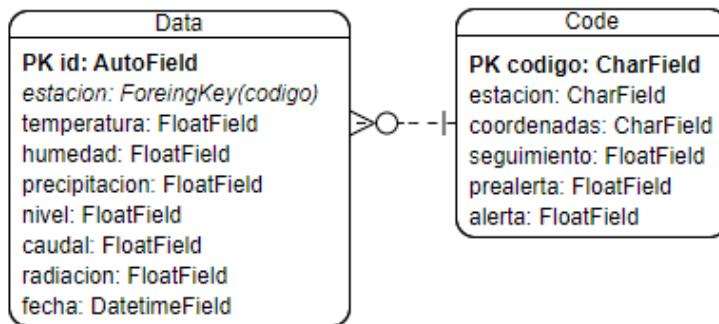


Figura 4.4: Tablas de la base de datos

Tienen una relación uno a muchos, siendo el campo estacion de la tabla Data la ForeignKey usada.

Para la tabla Data, al almacenar miles de datos, se ha optado por usar una PrimaryKey auto incremental. A su vez, a excepción del campo fecha, cualquiera de los campos pueden ser nulos, pues cada web proporciona únicamente un conjunto de los datos.

En la tabla Code, exceptuando la clave primaria y el nombre de la estación, el resto de campos tienen la posibilidad de ser nulos.

4.3 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.

4.3.1 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:

```
1 #Instalamos las dependencias
2 user@host:~$ sudo apt install python3-venv python3-dev
3
4 #Creamos un nuevo directorio sobre el que trabajar
5 user@host:~$ mkdir ~/dirdemiproyecto
6 user@host:~$ cd ~/dirdemiproyecto
7
8 #Creamos el entorno virtual
9 user@host:~/dirdemiproyecto$ python3 -m venv envdemiproyecto
10
11 #Activamos el entorno virtual
12 user@host:~/dirdemiproyecto$ source envdemiproyecto/bin/activate
13
14 #Una vez seguidos los pasos nuestra terminal debe mostrarse asi
15 (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 CHcantábrico se llamará chcantabrico. Otro entorno virtual es usado para el servidor Django y los scripts encargados del envío de datos.

4.3.2 Instalación y configuración de PostGreSQL

Los comandos necesarios para la instalación inicial de PostGreSQL, la creación de un usuario y una base de datos son.

```
1 #Instalamos PostGreSQL y las dependencias
2 user@host:~$ sudo apt install libpq-dev postgresql postgresql-contrib
3
4 #Iniciamos sesion usando el role postgres y accedemos a PostGreSQL
5 user@host:~$ sudo -u postgres psql
6
7 #Si todo esta bien la terminal deberia mostrarse as
8 postgres=# 
9
10 #Creamos un nuevo usuario
11 postgres=# CREATE USER miusuario WITH PASSWORD 'micontraseña';
12
13 #Configuramos varios parametros del usuario para una mejor integracion con
14 # Django
14 postgres=# ALTER ROLE miusuario SET client_encoding TO 'utf8';
15 postgres=# ALTER ROLE miusuario SET default_transaction_isolation TO 'read
16 committed';
16 postgres=# ALTER ROLE miusuario SET timezone TO 'Europe/Madrid';
```

```

17
18 #Creamos la Base de Datos
19 postgres=# CREATE DATABASE bbddmiproyecto;
20
21 #Otorgamos permisos de administrador a nuestro usuario en la base de datos
22 postgres=# GRANT ALL PRIVILEGES ON DATABASE bbddmiproyecto TO miusuario;
23
24 #Una vez finalizado
25 postgres=# \q

```

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

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

```

1 #Abrimos el archivo postgresql.conf con un editor
2 user@host:~$ sudo nano /etc/postgresql/11/main/postgresql.conf
3
4 #Buscamos la linea "#listen_addresses = 'localhost'", borramos la
5 #almohadilla y
6 sustituimos localhost por *, permitiendo la escucha de cualquier direccion IP
7 listen_addresses = '*'
8
9 #Guardamos y cerramos el fichero
10
11 #Abrimos el archivo pg_hba.conf con un editor
12 user@host:~$ sudo nano /etc/postgresql/11/main/pg_hba.conf
13
14 #Por defecto solo permite conexiones desde localhost
15 # IPv4 local connections:
16 host all all 127.0.0.1/32 md5
17
18 #Configurararemos el permiso de conexion remota desde cualquier IP a adiendo
19 la siguiente linea debajo de la anterior
20 host all all 0.0.0.0/0 md5
21
22 #O podemos configurar el permiso únicamente para la IP de la otra maquina
23 virtual
24 host all all 127.18.83.198/19 md5
25
26 #Guardamos y cerramos el fichero
27
28 #Finalmente permitimos el trafico mediante el puerto 5432 (puesto por defecto)
29 user@host:~$ sudo ufw allow 5432/tcp

```

Una vez realizados los siguientes pasos ya está la instancia de PostGreSQL configurada.

4.3.3 Instalación y configuración de Django

Antes de instalar Django, es necesario generar un entorno virtual sobre el que trabajar.

```
1 #Instalamos Django y las dependencias
2 (envdemiproyecto) user@host:~/dirdemiproyecto$ pip install django psycopg2
3
4 #Creamos un nuevo proyecto de Django
5 (envdemiproyecto) user@host:~/dirdemiproyecto$ django-admin startproject
6     djangoAPI
7 (envdemiproyecto) user@host:~/dirdemiproyecto$ cd djangoAPI
8
9 #Creamos una nueva app Django
10 (envdemiproyecto) user@host:~/dirdemiproyecto/djangoAPI$ python manage.py
11     startapp appAPI
```

Una vez realizado lo siguiente, quedaría configurar la API.

```
1 #Abrimos el archivo settings.py con un editor
2 (envdemiproyecto) user@host:~/dirdemiproyecto/djangoAPI$ nano
3             djangoAPI/settings.py
4
5 #A adimos nuestra app en la lista de INSTALLED_APPS
6 INSTALLED_APPS = [
7     'django.contrib.admin',
8     'django.contrib.auth',
9     'django.contrib.contenttypes',
10    'django.contrib.sessions',
11    'django.contrib.messages',
12    'django.contrib.staticfiles',
13    'appAPI.apps.AppapiConfig',
14 ]
15
16 #Configuramos la base de datos sustituyendo
17 DATABASES = {
18     'default': {
19         'ENGINE': 'django.db.backends.sqlite3',
20         'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
```

```

20     }
21 }
22
23 por
24
25 DATABASES = {
26     'default': {
27         'ENGINE': 'django.db.backends.postgresql_psycopg2',
28         'NAME': 'bbddmiproyecto',
29         'USER': 'miusuario',
30         'PASSWORD': 'micontraseña',
31         'HOST': '172.18.83.197',
32         'PORT': '5432',
33     }
34 }
35
36 #Configuramos las IP admitidas
37 ALLOWED_HOSTS = [ '172.18.83.197', 'localhost' ]
38
39 #Configuramos la zona horaria
40 TIME_ZONE = 'Europe/Madrid'
41
42 #Configuramos el tamaño maximo de datos a None, con el fin de poder
43 enviar cualquier cantidad de datos
44 DATA_UPLOAD_MAX_MEMORY_SIZE = None
45
46 #Guardamos y cerramos el fichero
47
48 #Finalmente permitimos el tráfico mediante el puerto 8000 (puesto por defecto)
49 user@host:~$ sudo ufw allow 8000

```

Tras configurar la API, toca crear los modelos de la base de datos.

```

1 #Abrimos el archivo models.py dentro de la carpeta appAPI con un editor
2 (envdemiproyecto) user@host:~/dirdemiproyecto/djangoAPI$ nano
3 appAPI/models.py

```

Dentro definimos los modelos Data y Code.

```

1 from django.db import models
2
3 class Data(models.Model):
4     temperatura = models.FloatField(null=True)
5     humedad = models.FloatField(null=True)

```

```

6   precipitacion = models.FloatField(null=True)
7   nivel = models.FloatField(null=True)
8   caudal = models.FloatField(null=True)
9   radiacion = models.FloatField(null=True)
10  fecha = models.DateTimeField()
11  estacion = models.ForeignKey("Code", on_delete=models.CASCADE)
12
13 class Code(models.Model):
14     estacion = models.CharField(max_length=50)
15     codigo = models.CharField(max_length=20, primary_key=True)
16     coodenadas = models.CharField(max_length=50)
17     seguimiento = models.FloatField(null=True)
18     prealerta = models.FloatField(null=True)
19     alerta = models.FloatField(null=True)

```

Código 4.2: Modelos API Django

Para importar los modelos a la base de datos, con el fin de crear las respectivas tablas.

```

1 #Actualizamos los datos realizados en models.py
2 (envdemiproyecto) user@host:~/dirdemiproyecto$ python3 manage.py
3           makemigrations
4
5 #Migramos los datos
6 (envdemiproyecto) user@host:~/dirdemiproyecto$ python3 manage.py migrate

```

Si se quiere confirmar la correcta creación de las tablas, desde la terminal de PostGre.

```

1 #Nos conectamos a la base de datos
2 postgres=# \c bbddmiproyecto
3
4 #Mostramos las tablas
5 postgres=# \d

```

Si todo a ido bien, se deberían mostrar como en la figura 4.5.

Listado de relaciones			
Esquema	Nombre	Tipo	Dueño
public	appAPI code	tabla	oderiz
public	appAPI data	tabla	oderiz
public	appAPI data id seq	secuencia	oderiz

Figura 4.5: Lista de tablas en PostGreSQL

Tras configurar la API y crear las tablas de la base de datos, se creará la API como tal.

```

1 #Creamos el archivo urls.py con un editor dentro de la carpeta appAPI
2 (envdemiproyecto) user@host:~/dirdemiproyecto/djangoAPI$ nano appAPI/urls.py

```

Dentro se incluye el siguiente código.

```

1 from django.urls import path
2 from django.views.decorators.csrf import csrf_exempt
3
4 from . import views
5
6 urlpatterns = [
7     path("storeData", csrf_exempt(views.storeData), name="storeData"),
8     path("storeCode", csrf_exempt(views.storeCode), name="storeCode"),
9 ]

```

Código 4.3: Definición URLs API

Aquí se definen las rutas para realizar una llamada a la API, en este caso storeData y storeCode.

Si se quiere que el proyecto tenga constancia de ellas.

```

1 #Abrimos el archivo urls.py dentro de la carpeta djangoAPI con un editor
2 (envdemiproyecto) user@host:~/dirdemiproyecto/djangoAPI$ nano
3 djangoAPI/urls.py

```

Dentro se incluye el siguiente código.

```

1 from django.contrib import admin
2 from django.urls import path, include
3
4 urlpatterns = [
5     path('admin/', admin.site.urls),
6     path('appapi/', include('appapi.urls'))
7 ]

```

Código 4.4: Configuración URLs API

Para finalizar, se crean las vistas a las que dirigen las rutas.

```

1 #Abrimos el archivo views.py dentro de la carpeta appAPI con un editor
2 (envdemiproyecto) user@host:~/dirdemiproyecto/djangoAPI$ nano appAPI/views.py

```

Dentro se incluye el siguiente código.

```
1 import json
2
3 from django.http import JsonResponse
4 from .models import Data, Code
5
6 # Create your views here.
7 def storeData(request):
8     data = json.loads(request.body.decode("utf-8"))
9     for estacion in data:
10         for datos in estacion['datos']:
11             dato = Data(
12                 temperatura=datos['temperatura ( C )'],
13                 humedad=datos['humedad (%)'],
14                 precipitacion=datos['precipitacion (mm)'],
15                 nivel=datos['nivel (m)'],
16                 caudal=datos['caudal (m^3/s)'],
17                 radiacion=datos['radiacion (W/m^2)'],
18                 fecha=datos['fecha y hora'],
19                 estacion=estacion['estacion']
20             )
21             dato.save()
22     return JsonResponse(data, safe=False)
23
24 def storeCode(request):
25     data = json.loads(request.body.decode("utf-8"))
26     for estacion in data:
27         dato = Code(
28             estacion=estacion['estacion'],
29             codigo=estacion['codigo'],
30             coodenadas=estacion['coodenadas'],
31             seguimiento=estacion['seguimiento'],
32             prealerta=estacion['prealerta'],
33             alerta=estacion['alerta'],
34         )
35         dato.save()
36     return JsonResponse(data, safe=False)
```

Código 4.5: Definición funciones de recepción y almacenamiento de datos API

Estas funciones se encargan de recibir los datos enviados, iterar por ellos y subirlos a la base de datos con el método `save()`.

Con esto concluiría la creación y configuración de la API.

Implementación

5.1 Creación de Spiders

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

```
1 scrapy startproject miproyecto
```

El cual crea un nuevo directorio con el siguiente contenido.

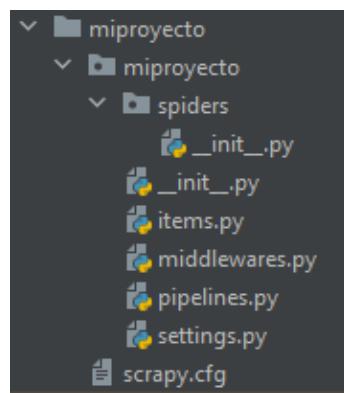


Figura 5.1: Estructura del proyecto recién creado

En el directorio recientemente creado, se ejecuta el comando encargado de crear la Spider.

```
1 cd miproyecto
2 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.

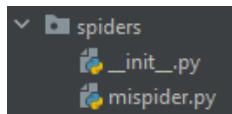


Figura 5.2: Directorio de almacenamiento de las Spider

Una vez abierto el archivo 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

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 referenciar la Spider a la hora de ejecutarla; *allowed_domains*, indica que dominios está permitido visitar, 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 deseada. Este método es invocado automáticamente por la Spider, sin necesidad real de hacerlo 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 se debe ir a la web deseada e inspeccionar su estructuración HTML. Para ello, como ejemplo, se va a usar la web de Aemet.

Hoy y Últimos días. Comunidad Foral de Navarra																					
Datos horarios		Resumen lunes 21		Resúmenes diarios anteriores																	
Mapa de España		Mapa		Tabla																	
<small>Actualizado: lunes, 21 agosto 2023 a las 10:22 hora oficial</small>																					
<small>Fecha y hora: lunes, 21 agosto 2023 a las 10:00 hora oficial</small>																					
Exportar a excel Exportar a csv																					
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	22.6	0	-	4	↑	0.0	986.9	0.5	79.0											
Areso	Navarra																				
Bardenas Reales, Base Aérea	Navarra	24.3	5	↗	11	→	0.0	986.9	0.5	67.0											
Baztan, Irurita	Navarra	21.9	0	-			0.0			89.0											

Figura 5.3: Web de Aemet para obtención de datos

Accediendo mediante el F12 a la herramienta de inspección, se busca el elemento representativo de los datos deseados. En este caso, el nombre y código de la estación. Ambos se encuentran en el mismo elemento que forma la primera columna de la tabla.

```

1 <table id="table">
2   <thead></thead>
3   <tbody>
4     <tr class="fila_par">
5       <td>
6         <a href="/es/eltiempo/observacion/ultimosdatos?k=nav&">
7           "amp; l=9263X&w=0& datos=det&">
8           "amp; f=precipitacion ">Aranguren, Ilundain</a>
9       </td>
10      ...
11    </tr>
12    ...
13  </tbody>
14</table>
```

Código 5.2: Estructura HTML de los datos deseados Aemet

En este caso es posible filtrar fácilmente los datos, obteniendo todos directamente, aunque lo más común sería obtener las filas primero para luego iterar por cada una de ellas. Para obtenerlos se puede hacer 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, click derecho sobre él y, en el apartado copiar se mostrará la posibilidad de copiar ambos selectores. Es probable que el selector proporcionado no sea del todo lo que se busque o se pueda simplificar, por lo que es recomendable comprobarlo manualmente.

```

1 rows = response.xpath('//div[@id="contenedor_tabla"]/table/tbody/tr/td/a')
2
3 rows = response.css("div#contenedor_tabla tbody tr a")

```

Esto devuelve una lista de objetos tipo Selector, cosa que permite conforme se itera por cada elemento, volver a usar un selector para filtrar únicamente los datos deseados. En este caso.

```

1 path = rows[i].xpath("@href").get()
2 name = rows[i].xpath("./text()").get()
3
4 path = rows[i].css("*::attr(href)").get()
5 name = rows[i].css("*::text").get()

```

De esta forma, mediante el uso de la función `get()`, se pasa 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 es de interés el código de la estación (parámetro l de la query empezando por 0), se filtra mediante `split()`.

```

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

```

5.1.2 Guardado de datos

Scrapy almacena los datos en forma de múltiples diccionarios, tantos como webs usadas. Para acceder a esta información Scrapy 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.

```

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

Código 5.3: Guardar datos

Actualmente, si se ejecuta la Spider imprimirá 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 este proyecto, al querer ejecutarlas de forma automática mediante el uso de Runners, se debe 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 5.4: Configurar guardado en JSON

Indicando que, en la ruta que se especifica, 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

Tras realizar los pasos, esta es la Spider básica terminada.

```

1  import scrapy
```

```

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 5.5: Spider de ejemplo (Aemet Code Spider)

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 se quiere obtener los datos de todas las estaciones dentro de un mismo dominio, se sobre-escribe la función para recorrer el JSON con los códigos de estas y, hacer una llamada por estación con Request.

El código quedaría de la siguiente manera.

```
1 import json
```

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

Código 5.6: Sobre-escritura de *start_requests()*

Al definir la función de esta manera, no es necesario declarar la variable *start_urls*, por lo que siempre que se necesite sobre-escribir la función, no se usará la variable.

5.1.5 Eliminar Log

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

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

Código 5.7: Configurar LOG

5.2 Spiders usadas

Para este proyecto se han creado cuatro proyectos de Scrapy, uno por cada web.

5.2.1 Aemet

Siendo la Spider de obtención de códigos la usada como ejemplo, no hay mucho más que comentar al respecto de ella, obtiene los nombres y códigos de cada estación.

Data Spider

La Spider de obtención de datos es un poco más compleja, necesitando sobre-escribir la función de *start_requests()* como se muestra en el ejemplo del apartado anterior. Primero lee el fichero JSON con los códigos de las estaciones, itera por ellos y, crea una llamada Request por cada estación.

Dentro de la función *parse()*:

```

1 latitud = response.css('abbr.latitude ::text').get()
2 longitud = response.css('abbr.longitude ::text').get()
3 estacion = response.css("a.separador_pestanas").get()
4 rows = response.css('tbody tr')

```

Código 5.8: Selector en `parse()` de Aemet Data Spider

Obtiene las coordenadas, latitud, longitud; una URL con el código de la estación, variable estacion y, todas las filas presentes en el cuerpo de la tabla de datos, rows.

```

1 datos = []
2 for row in rows:
3     dato = {
4         'fecha y hora': row.xpath('./td[1]/text()').get() + ':00',
5         'temperatura (C)': row.xpath('./td[2]/text()').get(),
6         'humedad (%)': row.xpath('./td[10]/text()').get(),
7         'precipitacion (mm)': row.xpath('./td[7]/text()').get(),
8     }
9
10    if dato['precipitacion (mm)'] != " ":
11        datos.append(dato)

```

Código 5.9: Trabajar sobre los datos de Aemet Data Spider

Se crea una lista datos vacía para almacenar los datos. Posteriormente, se recorren las filas obtenidas, creando un objeto JSON llamado dato, que almacena, obteniendo mediante selectores, la fecha y hora (añadiendo :00 al final para tener el mismo formato dd/mm/aa hh:mm:ss que en el resto de webs), la temperatura, la humedad y la precipitación.

En caso de que el apartado precipitación no esté vacío, pues puede darse el caso en el que la web aun no disponga de el a cierta hora, pero si muestre esta franja horaria pues dispone de otros datos como pueden ser aquellos relacionados con el viento, el objeto dato sera almacenado en la lista datos.

```

1 yield {
2     'coordenadas': latitud + ' | ' + longitud,
3     'estacion': estacion.split('=')[3].split('&')[0],
4     'datos': datos,
5 }

```

Código 5.10: Guardado de datos de Aemet Data Spider

Finalmente se almacenan las coordenadas con un formato global para todas las plataformas y, se filtra mediante splits el código de la estación.

5.2.2 CHCantábrico

Code Spider

La Spider para la adquisición de códigos de CHCantábrico no necesita de sobreescritura del método `start_requests()`, por lo que el código a analizar sera el presente en la función `parse()`.

```
1 rows = response.xpath('//table[@class="tablefixedheader niveles"]/tbody/tr')
```

Código 5.11: Selector en `parse()` de CHCantábrico Code Spider

Puesto que la estructuración HTML de la web no proporciona una manera clara de lograr los datos mediante un selector CSS, los datos se logran filtrar por su selector XPath. De este, se obtendrán las filas que representan cada estación.

```
1 for row in rows:
2     codigoBusqueda = row.css('td.codigo::text').get()
3     limites = row.css('table.umbrales_gr td.datos::text').getall()
4     path = row.xpath('./td/a/@href').getall()[-1]
5     estacion = row.xpath('./td/a/text()').getall()[-3]
```

Código 5.12: Trabajar sobre los datos de CHCantábrico Code Spider

Una vez obtenidas las filas, se itera sobre ellas para obtener, dos códigos, el primero, `codigoBusqueda` como código representativo de la estación cara a obtener las coordenadas y, el código necesario para acceder a los datos, inicialmente almacenado en una URL en la variable `path`; los límites marcados de pre-alerta, alerta y seguimiento en la variable `limites`, siendo estos una buena base para empezar con las predicciones de inundación y, el nombre de la estación.

Inicialmente, para las variables `path` y `estacion`, se obtienen todo los elementos que corresponden con el selector asignado, para posteriormente quedarse con el que realmente interesa. Se hace así al no poder filtrar mediante HTML los datos concretos.

```
1 for i in range(len(limites)):
2     if limites[i] == 'No definido':
3         limites[i] = None
```

Código 5.13: Comprobar límites de CHCantábrico Code Spider

Aun dentro del bucle, se comprueba si los límites están definidos, marcando como `None` (null) aquellos que no lo estén.

```
1 yield {
2     'estacion': estacion,
3     'codigo': path.split("=".split("=", path)[-1]),
4     'codigoSecundario': codigoBusqueda,
```

```

5     'seguimiento': limites[0],
6     'prealerta': limites[1],
7     'alerta': limites[2],
8 }
```

Código 5.14: Guardado de datos de CHCantábrico Code Spider

Por ultimo, se filtra el código de la estación (aquel usado para acceder los datos) de la URL y, se guardan los datos.

Data Spiders

CHCantábrico 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.

Puesto que ambas comparten la misma estructuración de código, solo se explicara una de ellas. La de nivel del río.

```

1 def start_requests(self):
2     with open("JSONs/RawCode/codigos_chcantabrico.json", encoding="utf-8") as
3         f:
4             data = json.load(f)
5             for estacion in data:
6                 params_nivel = {
7                     'p_p_id': 'GraficaEstacion_INSTANCE_wH0LL6jTUysu',
8                     'p_p_lifecycle': '2',
9                     'p_p_state': 'normal',
10                    'p_p_mode': 'view',
11                    'p_p_resource_id': 'downloadCsv',
12                    'p_p_cacheability': 'cacheLevelPage',
13                    '_GraficaEstacion_INSTANCE_wH0LL6jTUysu_cod_estacion':
14                        f'{estacion["codigo"]}',
15                    '_GraficaEstacion_INSTANCE_wH0LL6jTUysu_tipodato': 'nivel',
16                }
17
18                url = 'https://www.chcantabrico.es/evolucion-de-niveles'
19
20                yield scrapy.FormRequest(url=url,
21                    method='GET',
22                    formdata=params_nivel,
23                    callback=self.parse,
24                    cb_kwargs={'estacion': estacion['codigo']})
```

Código 5.15: Función `start_requests()` CHCantábrico Nivel Spider

Como se necesitan los datos de todas las estaciones, se reescribirá la función `start_requests()`.

A su vez, como CHCantábrico 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, se hace uso de FormRequest para hacer una llamada GET, que simular la llamada a un formulario y obtiene los datos que este devuelve.

De esta forma, pasandole los parámetros necesarios en el argumento `formdata` a la URL indicada, definidos como un diccionario en la variable `params_nivel`, se obtendrán 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 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 numero de argumentos a la función `parse()` de los que normalmente recibe.

```
1 def parse(self, response, estacion):
2     if not response.text.startswith('—'):
3         urlData = response.text
4         rawData = pd.read_csv(io.StringIO(urlData), delimiter=';',
5                               encoding='utf-8', header=1)
6         rawData.columns = ['fecha y hora', 'nivel (m)']
    parsedData = rawData.to_json(orient="records")
```

Código 5.16: Función `parse()` CHCantábrico Nivel Spider

Debido al argumento `cb_kwargs`, la función `parse()` recibe un tercer argumento, 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 la 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 seria esperado.

-
FECHA;VALOR(mm)

Una alternativa para deshacerse de esta comprobación sería eliminar aquellas estaciones que no proporcionen datos o filtrandolas para no hacer la llamada directamente, aunque esto no solo resultaría más complejo, si no que 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, el texto 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
...
...
```

Al tener formato CSV se lee mediante la función `read_csv()` incluida en la librería pandas, se indica el delimitador, la codificación y la linea 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, se cambian los nombres de las cabeceras a aquellos definidos de forma global para todas las webs y, se convierte el DataFrame en un JSON con la función `to_json()` indicando que el formato sea "records", esto implica que cada linea del DataFrame va a representar un objeto JSON.

```
1 yield {
2     'estacion': estacion,
3     'datos': json.loads(parsedData)
4 }
```

Código 5.17: Guardado de datos de CHCantábrico Nivel Spider

Con el uso de la función `loads()` de la librería json nos aseguramos el correcto formato del JSON.

Coordenates Spider

Aunque en la web misma de CHCantábrico 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 se pueden encontrar mediante el "codigoSecundario" anteriormente obtenido, desgraciadamente, no todas las estaciones incluidas en CHCantábrico están listadas en esta página, siendo el mayor inconveniente para el correcto funcionamiento del apartado de predicción.

```
1 longitud = response.css('p::text')[6].get().strip()
2 latitud = response.css('p::text')[7].get().strip()
3 estacion = response.css('font::text')[14].get().strip()
```

Código 5.18: Selector en *parse()* de CHCantábrico Coordinates Spider

```
1 yield {
2     'coordenadas': f'Lat: {latitud} | Lon: {longitud}',
3     'estacion': estacion,
4 }
```

Código 5.19: Guardado de datos de CHCantábrico Coordinates Spider

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.

Código descartado

Siendo CHCantábrico 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 están presentes en la web y, que Scrapy no dispone de interacción JavaScript por defecto, la única alternativa viable conocida 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', encoding='utf-8') as
7     f:
8     data = json.load(f)
```

```

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

```

Código 5.20: Script de obtención de datos pluviometricos descartado

Todo el apartado de tratamiento de los datos es prácticamente idéntico al realizado con la Spider, solo que en vez de usar FormRequest, se hace 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, guardando los datos en la lista `datos`. Una vez realizadas todas las llamadas, guardamos los datos obtenidos usando la función `json.dump()`.

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 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 quedó claro que, aun siendo posible automatizar el proceso con el script anterior, iba a suponer un problema para la modularidad del proyecto. Pues tener múltiples scripts de diferentes fuentes, solo aumentaba la complejidad a la hora de crear un script ya sea en Python o Bash encargado de ejecutar cada parte individual de cada web. Siendo la alternativa proporcionada por Scrapy de ejecutar múltiples Spider mediante un simple script Python la mejor opción.

Es por eso que se investigó 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

Code Spider

La estructuración HTML de la web y la falta de experiencia son el motivo principal asociado a la complejidad de esta Spider, la mayor parte del tiempo a la hora de crearla ha sido invertido explorando el código HTML buscando la forma más óptima de lograr los datos necesarios. Son múltiples las iteraciones sufridas hasta llegar a la versión actual.

```
1 rows = response.css('div#tabAUTO script::text').getall()
```

Código 5.21: Selector en `parse()` de MeteoNavarra Code Spider

Del selector se obtiene el código JavaScript perteneciente a cada fila de la tabla, siendo la forma más sencilla de lograr tanto el nombre como el código de la estación.

```
1 for row in rows:
2     yield {
3         'estacion': row.split(',')[3],
```

```

4     'codigo': row.split(',') [0].split('(') [1],
5 }
```

Código 5.22: Guardado de datos de MeteoNavarra Code Spider

A la hora de guardar los datos, se itera por filas y se filtra del código aquellos deseados.

Data Spider

La necesidad de recorrer múltiples estaciones, obliga como en el resto de Spiders de obtención de datos, a sobre escribir la función *start_requests()*.

```

1 import datetime
2
3 current_date = datetime.date.today()
4 delta = datetime.timedelta(days=1)
5 tomorrow_date = current_date + delta
6 yesterday_date = current_date - delta
7 tomorrow_date_format = tomorrow_date.strftime("%d/%2F %m/%2F%Y").replace(
8     '0', '')
9 yesterday_date_format = yesterday_date.strftime("%d/%2F
    %m/%2F%Y").replace('0', '')
```

Código 5.23: Uso de fechas en función *start_requests()* MeteoNavarra Data Spider

La curiosidad en esta radica en que, a diferencia del resto de páginas, que definen automáticamente una franja de fechas a mostrar, meteoNavarra fuerza la necesidad de que el usuario elija las fechas que deseé ver, es por eso que hace uso de la librería *datetime*, pues facilita el trabajo con fechas.

Con la intención de que funcione proporcionando fechas distintas cada día se realiza el siguiente proceso. Se obtiene la fecha actual con la función *today()*, con el método *timedelta(days=1)* se crea un objeto *timedelta* que representa una duración de un día, gracias a él, se calcula el día de mañana y el de ayer, sumando y restando esa duración al día de hoy. Para terminar, puesto que la fecha debe ir incluida en una URL, mediante el método *strftime()*, formateamos los *datetime* calculados para que correspondan con el codificado URL.

```

1 rows = response.css('table.border tr:not([bgcolor="#FFFFFF"])')
2 estacion = response.css('table a::attr(href)')[7].get()
```

Código 5.24: Selector en *parse()* de MeteoNavarra Data Spider

De los siguientes selectores cabe mencionar el primero. Aunque en la web se hace uso del elemento HTML table, este no indica que elementos tr pertenecen a la cabecera o al cuerpo de la tabla con el uso de thead y tbody, por el contrario, las filas correspondientes a la cabecera se muestran con el color de fondo representado hexadecimalmente #FFFFFF. Es por eso que se indica explícitamente no obtener esas filas con el uso de tr:not([bgcolor*="#FFFFFF"]).

```

1  datos = []
2  for row in rows:
3      dato = {
4          'fecha y hora': row.xpath('.//td[1]/text()').get().strip().replace(' ', ' ')
5              + ':00',
6          'temperatura (C)': row.xpath('.//td[2]/font/text()').get(),
7          'humedad relativa (%)': row.xpath('.//td[3]/font/text()').get(),
8          'radiacion global (W/m^2)': row.xpath('.//td[4]/font/text()').get(),
9          'precipitacion (1/mm^2)': row.xpath('.//td[5]/font/text()').get(),
10     }
11
12     if dato['radiacion global (W/m^2)'] == '--':
13         dato['radiacion global (W/m^2)'] = None
14
15     if dato['precipitacion (1/mm^2)'] != '--':
16         datos.append(dato)

```

Código 5.25: Trabajar sobre los datos de MeteoNavarra Data Spider

En la función `parse()`, como en otras Spider, se crea una lista `datos` vacía que es llenada con los objetos JSON obtenidos al recorrer cada fila de las anteriormente obtenidas y, filtrar mediante selectores aquellos datos de utilidad.

Luego, se comprueba la disponibilidad de un valor numérico de radiación global, pues en caso de no existir, la web proporciona '- -'. 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.

```

1  if datos:
2      yield {
3          'estacion': estacion.split('idestacion=')[1].split('&')[0],
4          'datos': datos,
5      }

```

Código 5.26: Comprobacion exitencia de datos y guardado de MeteoNavarra Data Spider

A continuación, en caso de que existan datos a guardar, pues algunas de las estaciones puedes llegar a no disponer de datos, se realiza el *yield*.

```

1  next_page = response.css("table a::attr(href)").getall()
2  page_number = response.xpath("//b/text()").getall()
3  if not page_number[0] == page_number[1]:
4      next_page = response.urljoin(next_page[1])
5      yield scrapy.Request(next_page, callback=self.parse)

```

Código 5.27: Navegacion a segunda página de datos en MeteoNavarra Data Spider

Finalmente, aun dentro de *parse()*, como los datos de ciertas estaciones están repartidos en dos páginas, adquiere la posible URL a la segunda página y, el numero de la página actual y siguiente, en caso de que los números sean distintos, navega 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 sera ejecutado, llegando a haber múltiples *yield* en una misma función.

Coordenates Spider

Por último, necesitamos de una Spider para la obtención de las coordenadas pues no están disponibles para obtener dentro de las páginas anteriores, aunque sí dentro del mismo dominio.

Mediante *start_requests()*, realiza tantas llamadas como estaciones.

```

1  coordenadas = response.css('td::text')[19].get()
2  estacion = response.css('input::attr(value)').get()

```

Código 5.28: Selector en *parse()* de MeteoNavarra Coordenates Spider

Una vez en la función *parse()*, toma las coordenadas mostradas en una tabla y, obtiene el código de la estación del atributo value de un input.

```

1  yield {
2      'coordenadas': coordenadas.strip().replace('\r\n\t\t', ' ').replace(
3          '*', ''),
4      'estacion': estacion,
}

```

Código 5.29: Guardado de datos de MeteoNavarra Coordenates Spider

Finalmente, las coordenadas, con el fin de verse bien en la web, en vez de estar estilizadas con CSS, vienen estilizadas con elementos textuales, los cuales son eliminados.

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 Seleimiun para poder interactuar con la web mediante JavaScript.

Ambas Spider, Code y Data, parten de las misma `start_urls` y van recorriendo las páginas presentes hasta llegar a aquella que muestre los datos deseados. 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 a visitar. La navegación por la web se realiza usando múltiples funciones de la misma naturaleza que `parse()`.

Para Data Spider esto implica no necesitar de un fichero JSON del cual leer los códigos de las estaciones.

```
1 def parse(self, response):
2     for link in response.css('dl#navarramap a::attr(href)'):
3         if link.get() != 'ctaMapa.aspx?IdMapa=1&IDOrigenDatos=1':
4             yield response.follow(link.get(), callback=self.parse_area)
```

Código 5.30: Función `parse()` Agua en Navarra Spiders

La función `parse()` es compartida por ambas Spider. Extrae los link representantes de cada área definida sobre el mapa y, navega por todos ellos a excepción del link de origen, aquel en el que se encuentra.

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 el uso de Request esta sería la manera.

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

De ambas formas, en el argumento `callback` se indica a qué función debe realizar la llamada, pasando de `parse()` a `parse_area()`.

```
1 def parse_area(self, response):
2     for link in response.css('area[shape="rect"]::attr(href)'):
3         yield response.follow(link.get(), callback=self.parse_estacion)
```

Código 5.31: Función `parse_area()` Agua en Navarra Spiders

Siendo la última función implementada de la misma manera en las Spider. Se realiza el mismo proceso, obtiene todos los link a las webs, ya sí, de la estación y, llama a `parse_estacion()`.

Code Spider

```

1 def parse_estacion(self, response):
2     urls = response.xpath('//span/a/@href').getall()
3     estacion = response.css('div#bloq_iconos span span::text').getall()
4     codigoEstacion = response.css("form#frmDatosEstacion ::attr(action)").get()
5
6     codigos = []
7     for url in urls:
8         codigos.append(url.split('=')[1])

```

Código 5.32: Función *parse_estacion()* Agua en Navarra Code Spider

Una vez en la función *parse_estacion()* de Code Spider, como ocurre con CHCantábrico, 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.

Tanto el código de nivel y caudal del río originalmente se almacenan en la variable *urls*, almacenando las direcciones en las cuales están presentes. Posteriormente, se toma de la URL y se guardan en la lista *codigos*.

La variable estación es una lista que almacena, la descripción de la estación, el municipio al que pertenece, el río y, las coordenadas.

```

1 yield {
2     'descripcion': estacion[0],
3     'municipio': estacion[1],
4     'rio': estacion[2],
5     'coordenadas': estacion[3],
6     'estacion': codigoEstacion.split('=')[-1],
7     'codigos': codigos,
8 }

```

Código 5.33: Guardado de datos de Agua en Navarra Code Spider

Al final estos datos son almacenados.

Data Spider

```

1 from scrapy_selenium import SeleniumRequest
2 from selenium.webdriver.common.by import By
3 from selenium.webdriver.support import expected_conditions as EC
4
5 def parse_estacion(self, response):
6     for link in response.css('div#divResultadosAforo a::attr(href)'):
7         yield SeleniumRequest(

```

```
8     url=response.urljoin(link.get()),
9     wait_time=2,
10    wait_until=EC.element_to_be_clickable((By.ID, 'btnDatosNumericos')),
11    callback=self.parse_data,
12    script='document.querySelector("#btnDatosNumericos").click()',
13 )
```

Código 5.34: Función `parse_estacion()` Agua en Navarra Data Spider

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, se lida 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 SeleniumRequest y llama a la función `parse_data()`.

Como el link proporcionado dirige a una página que muestra los datos en forma de gráfica y, para acceder a los datos numéricos es necesario pulsar un botón, SeleniumRequest se configura de tal forma que, espera 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` indicamos que se debe pulsar sobre este, llevándonos finalmente a los datos numéricos.

```
1 tipo = response.css('span#lblSenal::text').get()
2 estacion = response.css('li#cabecera_nombreEstacion a::attr(href)').get()
3 fechas = response.css('span.cont_fecha_gra::text').getall()
4 valores = response.css('span.cont_valor_gra::text').getall()
```

Código 5.35: Selector en `parse_data()` de Agua en Navarra Data Spider

En la función `parse_data()`, se obtienen 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 formateandolos posteriormente con CSS. Esta forma de mostrar los datos fuerza tener que obtener las fechas por un lado y los valores por otro, resultando en dos listas que debemos unir. La variable `tipo` (Nivel o caudal) es necesaria para posteriormente guardar los datos de forma correcta.

```
1     datos = []
```

```

2   if tipo == "Nivel Rio":
3       for i, fecha in enumerate(fechas):
4           dato = {
5               'fecha y hora': fecha.strip() + ':00',
6               'nivel (m)': valores[i].strip(),
7           }
8           datos.append(dato)
9   else:
10      for i, fecha in enumerate(fechas):
11          dato = {
12              'fecha y hora': fecha.strip() + ':00',
13              'caudal (m^3/s)': valores[i].strip(),
14          }
15          datos.append(dato)
16
17     yield {
18         'estacion': estacion.split('=')[-1],
19         'datos': datos,
20     }

```

Código 5.36: Selector en *parse_data()* de Agua en Navarra Data Spider

Al hacer uso de una Spider para obtener todos los datos, estando presentes en distintas páginas, hace necesaria la comprobación inicial con la variable tipo de que datos estamos recibiendo, para poder almacenarlos correctamente. Una vez realizada, se itera la lista de fechas y se unen con su respectivo valor (el orden de los valores corresponde con el de las fechas) y se insertan en la lista datos. Finalmente son guardados.

Configuración Selenium

Para que Selenium 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 DOWNLOADERS_MIDDLEWARES = {
13     'scrapy_selenium.SeleniumMiddleware': 800
14 }

```

Código 5.37: Agua en Navarra configuración Selenium

En este caso se indica que el navegador deseado para realizar el trabajo es Chrome y mediante el método `which` selecciona 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 abriendose 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 se desea 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 quedaría 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.

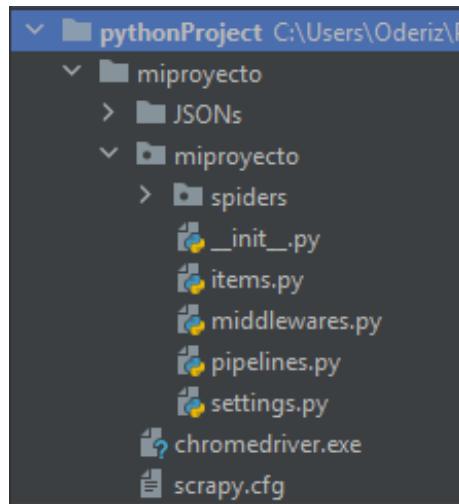


Figura 5.4: Ruta chromedriver.exe

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

```

1 #Otorgamos permiso de ejecucion a chromedriver
2 sudo chmod +x chromedriver
3
4 #Movemos el .exe a directorio /usr/local/share/
5 sudo mv chromedriver /usr/local/share/chromedriver
6
7 #Creamos la relacion entre el .exe y el directorio
8 sudo ln -s /usr/local/share/chromedriver /usr/bin/chromedriver
9
10 #Ejecutamos
11 chromedriver --version
12
13 #En caso de haber realizado correctamente el proceso deberia aparecer
14 una linea parecida a esta
15 ChromeDriver 115.0.5790.110
16 (5e87dfef0c85687ea835e444d33466745cc0725f-refs/branch-heads/5790_90@{#20})

```

Código 5.38: Configuración chromedriver Debian

Hecho esto ya estaría configurado chromedriver.

Alternativas probadas

Antes de saber que es necesario dirigirse a la pagina donde se muestra la gráfica para 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 del error mostrado en la figura 2.10.

Es por eso que, partiendo de un uso de Scrapy sin uso de extensiones de terceros, se probó una solución al problema que seguía la metodología de recorrer las webs, de esta manera, se visita inicialmente la web con la gráfica para posteriormente redirigirse 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. Tras volver a la comprobación manual, resultó 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 página, lo mismo se actualizaban los datos para mostrar los de la nueva estación como no lo hacían, mostrando los datos de la anterior.

Esto causó la incertidumbre de si era posible obtener datos la web de forma fiable. Finalmente, tras barajar 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, al ser usado mediante la extensión `scrapy_selenium`, resultó 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 ejecutarse 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, al prometer mejoras en los tiempos de ejecución y una mejor optimización.

Playwright

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.

```
1 #Instalamos scrapy-playwright
2 pip install scrapy-playwright
3
4 #Instalamos los navegadores compatibles
5 playwright install
```

Código 5.39: Instalación Playwright

Para hacer uso de Playwright con Scrapy, primero que todo es configurar las dependencias de Playwright, implementadas mediante `custom_settings` en una Spider o, si se desea, en `settings.py`.

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

Código 5.40: Configuración Playwright

Finalmente, para hacer uso de la herramienta, a diferencia de con Selenium, esta no dispone 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 5.41: Playwright basic Request

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     ),
16 )

```

Código 5.42: Agua en Navarra Playwright Request

Los PageMethods indican las acciones a realizar una vez se hace la request, la alternativa optada por Playwright al uso de JavaScript. Primero, espera a que el botón esté cargado dentro de la página, pulsa sobre él y, espera 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.

5.3 Runners

Todos los Runners comparten la misma estructura que se explica. Posteriormente, se indicará el código de los Runners usados.

```
1  from pagina.spiders.pagina_spider import PaginaSpider
2  from scrapy.crawler import CrawlerProcess
3  from scrapy.utils.project import get_project_settings
4
5
6  def main():
7      settings = get_project_settings()
8      process = CrawlerProcess(settings)
9      process.crawl(PaginaSpider)
10     process.start()
```

Primero se incluyen las dependencias, entre ellas la Spider a usar, en caso de querer ejecutar múltiples Spider se deberán importar todas ellas y se ejecutaran en paralelo. En la función *main()*, se toma la configuración del fichero *settings.py* y en caso de existir de la variable *custom_settings*, CrawlerProcess prepara el proceso requerido para ejecutar las Spider, se indica con el método *crawl()* la Spider a usar y, se da comienzo al proceso con *start()*.

```
1  from datosEstaciones.spiders.chcantabrico_pluvio_spider import
   ChcantabricoPluvioSpider
2  from datosEstaciones.spiders.chcantabrico_nivel_spider import
   ChcantabricoNivelSpider
3  from datosEstaciones.spiders.chcantabrico_coord_spider import
   ChcantabricoCoordSpider
4
5  process.crawl(ChcantabricoPluvioSpider)
6  process.crawl(ChcantabricoNivelSpider)
7  process.crawl(ChcantabricoCoordSpider)
```

Código 5.43: CHCantábrico Data Runner

```
1  from datosEstaciones.spiders.aemet_spider import AemetDataSpider
2
3  process.crawl(AemetDataSpider)
```

Código 5.44: Aemet Data Runner

```
1  from datosEstaciones.spiders.meteoNavarra_coord_spider import
   MeteonavarraCoordSpider
```

```

2   from datosEstaciones.spiders.meteoNavarra_spider import
3       MeteonavarraDataSpider
4
5   process.crawl(MeteonavarraDataSpider)
6   process.crawl(MeteonavarraCoordSpider)

```

Código 5.45: MeteoNavarra Data Runner

5.4 Executers

Ficheros data_spider_executer.sh y code_spider_executer.sh

```

1  #!/bin/bash
2
3  # Activate the virtual environment
4  source Scrapy/$1/bin/activate
5
6  # Change to the directory containing the Python script
7  cd Scrapy/$1Spider

```

Código 5.46: Ejecucion de entorno virtual y selección de proyecto Scrapy

Primero ejecutan el comando para activar el entorno virtual indicada por argumento y se desplazan al directorio del proyecto Scrapy requerido.

```

1  # Run the Python script
2  python3 data_spider_runner.py

```

Código 5.47: Ejecución de data_spider_runner.py

En caso de data_spider_executer.sh, ejecuta el Runner encargado de obtener los datos.

```

1  # Run the Python script
2  python3 code_spider_runner.py

```

Código 5.48: Ejecución de code_spider_runner.py

Y en caso de code_spider_executer.sh, ejecuta el Runner encargado de obtener los códigos.

5.5 Formateo de datos

Para formatear los datos de forma global, cada proyecto individual dispone de dos scripts encargados de hacerlo, uno para los códigos y otro para los datos per se.

5.5.1 Aemet

5.5.2 CHCantábrico

5.5.3 MateoNavarra

5.5.4 Agua en Navarra

5.6 Filtrado de datos

El siguiente paso a realizar, es el paso de los JSON por el script `filtras_datos_JSONs.py`. En él se realiza lo siguiente.

```
1 import json
2 from datetime import date, datetime
3 from pathlib import Path
```

Código 5.49: Import necesarios filtrado de datos

Se importan las dependencias.

```
1 OldDir = "JSONs/OldData/"
2 ParsedDir = "JSONs/ParsedData/"
3 RefinedDir = "JSONs/RefinedData/"
4 DataJSON = "datos_aemet.json"
```

Código 5.50: Declaración rutas JSONs y nombre de fichero

Se declaran las variables representativas de las rutas y el nombre del fichero JSON.

```
1 def openFile(fileDir):
2     try:
3         with open(fileDir + DataJSON, "r", encoding="utf-8") as f:
4             file = json.loads(f.read())
5     except FileNotFoundError:
6         file = None
7     return file
```

Código 5.51: Declaración función openFile()

La función prueba a abrir el fichero anteriormente declarado (`datos_aemet.json`) sobre la ruta indicada (`fileDir`), en caso de no existir, se marca el fichero como `None`, finalmente devuelve el fichero. El tratado de la excepción `FileNotFoundException` se realiza pensada en la primera vez que se haga uso de la plataforma, ya que esta no dispondría de datos posteriores con los que hacer una comparación.

```
1 def saveData(jsonDir, DataFile):
2     Path(jsonDir).mkdir(parents=True, exist_ok=True)
3     with open(jsonDir + DataJSON, 'w', encoding='utf-8') as outfile:
```

```
4     json.dump(DataFile, outfile)
```

Código 5.52: Declaración función saveFile()

Recibiendo como argumento el directorio sobre el que se desea guardar el fichero (jsonDir) y un diccionario JSON con los datos a guardar (DataFile), esta función, comprueba la existencia del directorio y en caso de no hacerlo lo crea. Tras ello, guarda los datos en el directorio con el uso de `json.dump()`.

```
1 def searchEstacionData(code, dataFile):
2     for data in dataFile:
3         if data[ 'estacion' ] == code:
4             return data[ "datos" ]
```

Código 5.53: Declaración función searchEstacionData()

Como no es seguro que los datos recibidos estén siempre en el mismo orden, se ha definido esta función para asegurarse de trabajar siempre con los datos de la estación correcta. Recorre el fichero viejo y cuando encuentra la estación con el mismo código devuelve los datos de esta.

```
1 def refineData(newFile, oldFile):
2     refinedFile = []
3
4     for i, item in enumerate(newFile):
5         newData = []
6         oldData = searchEstacionData(item[ 'estacion' ], oldFile)
7         for data in item[ "datos" ]:
8             if data[ "fecha y hora" ] not in [x[ "fecha y hora" ] for x in oldData]:
9                 newData.append(data)
10    if newData:
11        refinedFile.append(
12            {
13                "coordenadas": item[ "coordenadas" ],
14                "estacion": item[ "estacion" ],
15                "dato": newData
16            }
17        )
18
19    saveData(RefinedDir, refinedFile)
```

Código 5.54: Declaración función refinedData()

Esta se encarga de la comparación de los ficheros, recibidos en los argumentos newFile y oldFile.

Empezando con la explicación, se define la lista vacía `refinedFile`, encargada de almacenar los datos nuevos de todas las estaciones; se itera por cada estación en `newFile`;

se define una segunda lista newData para almacenar los nuevos datos por estación; se llama a *searchEstacionData()* para buscar los datos viejos de la estación; se itera por todos los datos de los presentes en la estación; se comprueba que no exista previamente en los datos de esa misma estación en oldData; en caso de que el dato no este presente, se almacena en newData; si existe algún dato nuevo (newData no esta vacía), se crea un objeto JSON que sigue la estructuración definida para los ficheros de datos y, se almacena en refinedData; finalmente los datos se guardan en un fichero en la ruta especificada RefinedDir.

```
1 def main():
2     newFile = openFile(ParsedDir)
3     oldFile = openFile(OldDir)
4
5     if oldFile is None:
6         saveData(RefinedDir, newFile)
7         return
8
9     refineData(newFile, oldFile)
```

Código 5.55: Declaración rutas JSONs

La función *main()*, abre los ficheros indicados en las rutas ParsedDir y OldDir, en caso de que oldFile sea None, guarda los nuevos datos directamente en el directorio refinedDir y el programa finaliza. Si se dispone de datos en oldFile, llama a *refineData()*.

5.7 Posts

Como Posts, se hace referencia a los Scripts code_post.py y data_post.py. Como ambos comparten la misma codificación se explicaran de forma general y, se mostraran las diferencias.

```
1 import json
2 import sys
3 import requests
```

Código 5.56: Import necesarios post

Se importan las dependencias.

```
1 url = "http://localhost:8000/appAPI/storeCode"
2
3 with open(f"JSONs/ParsedCode/codigos_{sys.argv[1]}.json", encoding="utf-8"):
4     as f:
5         data = json.load(f)
```

Código 5.57: Declaración variables code_post.py

```
1 url = "http://localhost:8000/appAPI/storeData"
2
3 with open(f"JSONNs/ParsedData/datos_{sys.argv[1]}.json", encoding="utf-8") as
4     f:
    data = json.load(f)
```

Código 5.58: Declaración variables data_post.py

Se declara la url a la que se desea hacer la llamada y, se lee el fichero que se ha indicado al pararlo por argumento a la hora de ejecutar el Script.

```
1 headers = {'content-type': 'application/json'}
2 r = requests.post(url, json=data, headers=headers)
```

Código 5.59: Llamada POST

Finalmente, se indica en headers el tipo de contenido que dispone la llamada, en este caso JSON y, se realiza la llamada con el método *post()* proporcionado por la librería requests.

6

Conclusiones y Trabajo Futuro

El proyecto a sido llevado a cabo finalmente con las siguientes herramientas, PostGreSQL el la base de datos, Django para la API, Scrapy y Selenium en la plataforma de adquisición de datos y Cron para la automatización de las tareas. Con esto, se ha podido realizar una plataforma, que ejecuta scripts de forma regular en intervalos de tiempo predefinidos para cada página usada, con el fin de no generar más trafico web del realmente necesario.

En este proyecto se ha pretendido crear una arquitectura lo más modular posible, ya sea por el uso de múltiples máquinas virtuales, como por el uso de un entorno virtual por cada uno de 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.

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. Aunque se perdería gran parte de la flexibilidad proporcionada.

Django no cabe duda de ser un Framework completo sobre el que trabajar, pero a resultado ser en cierta mediada más un problema que una solución, su sistema de gestión de versiones sobre los cambios realizados en la base de datos, supuso tener que rediseñar la arquitectura a una versión más compleja. A su vez, al crear las tablas de la base de datos a partir de los modelos de Django, hizo que no se pudiera hacer uso de una clave primaria compuesta, ya que Django no da soporte a tablas así, siendo una de las peticiones que más ha realizado la comunidad de Django.

A nivel de obtención de datos, se han presentado múltiples problemas sobre las dis-

tintas webs. Como se ha visto, algunas no hacen un uso correcto de HTML, creando tablas a partir de CSS sobre elementos *div* en vez de usar las tablas definidas en HTML; o no hacen un buen uso de los atributos *id* y *class*, haciendo la obtención de datos algo tedioso, llegando a tener que filtrar datos mediante los atributos CSS, siendo un claro ejemplo la plataforma de El Agua en Navarra, donde para obtener los códigos de la estación, son filtrados aquellos elementos con el atributo CSS *shape="rect"*.

Otros problemas encontrados están relacionados con las coordenadas, pues el que CHCántabrico no disponga de las coordenadas de las estaciones presentes, siendo un dato que debería ser de fácil acceso público, casi hace que no pueda ser usada y, no ha sido así gracias a que se lograron las coordenadas de un segunda página. Y eso que no se ha llegado a disponer de todas las coordenadas. El segundo problema radica en que todas las estaciones hacen uso de un estándar distinto a la hora de mostrar las coordenadas, por lo que habría que crear un Script capaz de convertir las coordenadas a un mismo estándar para poder ser usadas correctamente.

Centrado en técnicas de scraping web, este proyecto esta 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 esta 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.

Referencias

- [1] "United Nations what is climate change." <https://www.un.org/es/climatechange/what-is-climate-change>. Accessed: 2023-09-19.
- [2] W. F. Ruddiman, "The anthropogenic greenhouse era began thousands of years ago," *Climatic change*, vol. 61, no. 3, pp. 261–293, 2003.
- [3] "National Centers for Environmental Information annual 2022 global climate report." <https://www.ncei.noaa.gov/access/monitoring/monthly-report/global/202213>. Accessed: 2023-09-19.
- [4] N. W. Arnell, J. A. Lowe, A. J. Challinor, and T. J. Osborn, "Global and regional impacts of climate change at different levels of global temperature increase," *Climatic Change*, vol. 155, pp. 377–391, 2019.
- [5] M. New, D. Liverman, H. Schroder, and K. Anderson, "Four degrees and beyond: the potential for a global temperature increase of four degrees and its implications," 2011.
- [6] T. H. Sparks and A. Menzel, "Observed changes in seasons: an overview," *International Journal of Climatology: A Journal of the Royal Meteorological Society*, vol. 22, no. 14, pp. 1715–1725, 2002.
- [7] "National Centers for Environmental Information annual 2022 global climate report precipitation." <https://www.ncei.noaa.gov/access/monitoring/monthly-report/global/202213#precip>. Accessed: 2023-09-19.
- [8] C. Wasko, S. Westra, R. Nathan, H. G. Orr, G. Villarini, R. Villalobos Herrera, and H. J. Fowler, "Incorporating climate change in flood estimation guidance," *Philosophical Transactions of the Royal Society A*, vol. 379, no. 2195, p. 20190548, 2021.

- [9] "Ministerio para la transformación ecológica y el reto demográfico gestión de los riesgos de inundación." <https://www.miteco.gob.es/es/agua/temas/gestion-de-los-riesgos-de-inundacion.html>. Accessed: 2023-09-20.
- [10] C. Dierbach, "Python as a first programming language," *Journal of Computing Sciences in Colleges*, vol. 29, no. 3, pp. 73–73, 2014.
- [11] "Python 3.11.3 Documentation general python faq." <https://docs.python.org/3/faq/general.html>. Accessed: 2023-04-07.
- [12] L. E. Borges, *Python para desenvolvedores: aborda Python 3.3*. Novatec Editora, 2014.
- [13] M. F. Krafft, *The Debian system: concepts and techniques*. No Starch Press, 2005.
- [14] R. Hertzog and R. Mas, *The Debian Administrator's Handbook: Debian Jessie From Discovery To Mastery*. Freexian, 2015.
- [15] "Debian Project Documentation a brief history of debian." <https://www.debian.org/doc/manuals/project-history/index.en.html>. Accessed: 2023-04-12.
- [16] "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.
- [17] R. P. Pollei, *Debian 7: System Administration Best Practices*. Packt Publishing, 2013.
- [18] "Debian Project Documentation what does free mean?." <https://www.debian.org/intro/free.en.html>. Accessed: 2023-04-12.
- [19] "GNU About Free Software Documentation what is free software?." <https://www.gnu.org/philosophy/free-sw.en.html>. Accessed: 2023-04-12.
- [20] "Debian Project Documentation debian social contract version 1.2 ratified on october 1st, 2022.." https://www.debian.org/social_contract.en.html. Accessed: 2023-04-12.
- [21] "Debian Project Documentation what is the partners program?." <https://www.debian.org/partners/index.en.html>. Accessed: 2023-04-12.
- [22] "Debian Project Documentation how to donate to the debian project." <https://www.debian.org/donations.en.html>. Accessed: 2023-04-12.
- [23] "Debian Project Documentation reasons to use debian." https://www.debian.org/intro/why_debian.en.html. Accessed: 2023-04-12.

- [24] D. Ghimire, "Comparative study on python web frameworks: Flask and django," 2020.
- [25] "Python Documentation web frameworks for python." <https://wiki.python.org/moin/WebFrameworks>. Accessed: 2023-04-12.
- [26] 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.
- [27] "Django Documentation design philosophies." <https://docs.djangoproject.com/en/3.0/misc/design-philosophies/>. Accessed: 2023-04-12.
- [28] M. Alchin, *Pro Django*. Apress, 2013.
- [29] A. Ravindran, *Django Design Patterns and Best Practices*. Packt Publishing Ltd, 2015.
- [30] B. Zhao, "Web scraping," *Encyclopedia of big data*, pp. 1–3, 2017.
- [31] V. Krotov and L. Silva, "Legality and ethics of web scraping," 2018.
- [32] 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.

Anexos