UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

Desarrollo de una Herramienta de Análisis y Procesado de imágenes Geoespaciales

Trabajo Fin de Grado

Grado en Ingeniería Informática

AUTOR/A: Ferriols Jiménez, Francisco Miguel

Tutor/a: Pelechano Ferragud, Vicente

Cotutor/a externo: MENGUAL BORRAS, ALEJANDRO

CURSO ACADÉMICO: 2021/2022

Resumen

Desarrollo de una aplicación para analistas de imágenes geo-espaciales, esta aplicación se construirá como un plugin para *QGIS* en el cual los analistas podrán utilizar distintas herramientas para generar Eventos.

Cada evento consta de diversas propiedades como un conjunto de polígonos que delimitan la mancha, el tipo de derrame que se ha producido, la cantidad, etc...

Las herramientas han de permitir a un analista, mediante una imagen captada por un satélite, delimitar este polígono, establecer los parámetros de cada evento, asociarlos a regiones(Regiones del mundo delimitadas por fronteras) y otras tantas funcionalidades futuras que dependen de las necesidades de la empresa.

Además esta incluirá unas funcionalidades extra para pre-procesar las imágenes provenientes de los distintos satélites asociados a la empresa de forma automática, para que la aplicación *QGIS* pueda leerlas correctamente.

Palabras clave: Python, Geo-spatial, GDAL, image processing, analysis, events, oil spills, interfaces.

Abstract

Development of an application for Geo-spatial images analysts. It will be built as a *QGIS plugin* in which analysts can make use of a wide variety of tools to generate custom Events related to oil spills and more.

Each event consists of various properties such as its polygonal structure, the type of spill that has been produced, its range, a prediction of its spread, etc...

These tools will allow the analyst to process an image by forming the spill polygon, then manually or automatically establish specific event parameter. After that they can assign the event to a Region (Real world regions) and to a Client, which will get notified of the event after being published. More tools could be added according to the Company necessities.

Furthermore, the application will include extra functionalities to pre-process images incoming from different Satellites so that the tools can read and process them properly.

Key words: Python, Geo-spatial, GDAL, image processing, analysis, events, oil spills, interfaces.

Índice general

Índice de figuras

Índice de tablas

CAPÍTULO 1
INTRODUCCIÓN

El análisis de imágenes geo-espaciales es necesario para nuestra vida cotidiana. Desde GPS, rutas de reparto y vuelos hasta control aplicaciones de defensa, monitorización de catástrofes naturales y análisis de impacto ambiental entre otros. Toda esta información viene dada por estas imágenes. Desde el punto de vista de un usuario, pondremos como ejemplo Google Maps, solo observas los datos procesados, como podría ser una imagen de un mapa con puntos de interés, nombres, calles, etc... Pero en la realidad estas imágenes son generadas mediante algoritmos usando la información recibida por distintos satélites.

1.1 Motivación

La contaminación marítima es un problema de gran calibre y a menudo olvidado por todos. A diario se encuentran manchas de diferentes contaminantes en la superficie marítima, y a la mayor parte de ellas se les hace oídos sordos. *OrbitalEOS* pretende, con sus servicios, facilitar una herramienta para detectar de forma clara estos contaminantes, mejorando así el estado de nuestros mares y océanos.

Por otro lado, se encuentra como una parte de este problema, el análisis de estas manchas, ahí es donde entra esta aplicación. El trabajo del analista es muy complejo, hay que tener en cuenta muchos datos y detalles que una simple imagen no puede mostrar. Por eso se pretende hacerle la vida más fácil y agilizar este arduo proceso.

1.2 Objetivos

Al final de esta memoria se busca proveer a los analistas de una herramienta completamente funcional, preparada para adiciones futuras, la cual les permita utilizar sus conocimientos para generar Eventos por cada mancha de aceite u otros elementos contaminantes encontrados en la superficie marítima.

Estos eventos serán sucesos en el tiempo que serán visibles por los clientes mediante la otra herramienta provista por la empresa, el visualizador. Los eventos representarán un análisis realizado sobre un espacio en el tiempo a ciertas imágenes provistas por satélites asociados a la empresa. Cada evento constará de una capa poligonal representando el tamaño real de la mancha a la que hace referencia para poder ser mostrada sobre un mapa

completo del mundo, así mismo los eventos también incluirán todo tipo de información relacionada a la procedencia, el estado y la composición de dicha mancha[1].

Dicha herramienta será capaz de permitir incluir todo tipo de detalle e información necesaria para informar a un Cliente externo. Además tendrá disponible un reporte automático capaz de generar archivos *PDF* en varios idiomas con los datos de dichos eventos y permitirá a los analistas trabajar con diferentes formatos de archivos y múltiples satélites.

1.3 Estructura de la memoria

La memoria consta de cinco capítulos. A continuación se presenta un breve de resumen del contenido de cada uno.

1. Introducción, este mismo capítulo donde se introduce el trabajo a realizar, se presentan las motivaciones y objetivos además de la organización de la memoria.
2. Tecnologías es el segundo capítulo, donde se presenta una comparación de las diferentes tecnologías barajadas y la elección final para este trabajo.
3. Preparación del entorno, es el tercer capítulo en el cual se explica paso a paso como preparar el entorno para el desarrollo de la herramienta tratada.
4. Desarrollo es el cuarto capítulo y el más extenso, en este se sigue al completo el proceso de desarrollo, pasando por la especificación de requisitos, el diseño de los diferentes elementos de la herramienta, la metodología a seguir en el proceso y finalmente la implementación de la herramienta y sus pruebas.
5. Conclusión es el quinto y último capítulo en el cual se ofrece una conclusión tratando el desarrollo de los objetivos y el cumplimiento de los mismos además de la satisfacción y las adversidades encontradas en el mismo.
6. Finalmente se encuentran una serie de anexos donde aparecen acrónimos y palabras técnicas utilizadas junto a sus definiciones, además de las referencias utilizadas.

[1] Cabe destacar que las manchas no han de ser específicamente de petróleo, pero será el compuesto más común.

CAPÍTULO 2
TECNOLOGÍAS

En este capitulo serán discutidas las tecnologías a emplear en el desarrollo de la aplicación y se barajarán las opciones en función de las necesidades de la empresa y las utilidades aportadas por dichas herramientas [14].

2.1 Base de datos

La base de datos es uno de los apartados más importantes de toda aplicación dado que todo el tratado de información gira al rededor de la misma. Por este motivo es realmente importante tomar una decisión adecuada y estar seguro de la misma [14]. Para esto basaremos nuestra elección en los apartados discutidos a continuación.

2.1.1. CAP

Los principios del *teorema de CAP* o *teorema de Brewer* dice que en un sistema distribuido, la consistencia la disponibilidad y la tolerancia de partición no pueden ser compatibles entre sí. Estos principios nos obligan a elegir ciertas características sobre otras, dado que no se pueden obtener todas las propiedades al mismo tiempo.

La **consistencia** asegura como resultado de cada operación en la base de datos, un estado válido y un error en caso de fallo de los mismos. Por otro lado la **disponibilidad** dice que cada petición recibe una respuesta sin garantizar una consistencia entre los datos, es decir, no asegura que los datos sean los más recientes y se encuentren actualizados y por último la **tolerancia de partición** garantiza un continuo funcionamiento del sistema a pesar de los posibles errores encontrados.

De este modo podemos encontrar las 3 combinaciones, alta disponibilidad y consistencia, alta disponibilidad y tolerancia a la partición y alta consistencia y tolerancia a la partición.

Consecuentemente, para nuestro proyecto quedará definida la necesidad de una base de datos que tenga una alta disponibilidad y consistencia, siendo estas las características más necesarias para un entorno que proporcionara un servicio constante de flujo de datos.

2.1.2. SQL vs NoSQL vs NewSQL

En el siguiente paso de nuestra elección nos topamos con la decisión de *NoSQL* vs textitSQL [18], por suerte en este caso la decisión es bastante sencilla.

Con textitNoSQL podemos encontrar una gran variedad de tipos de bases de datos como podrían serlo las bases de datos Gráficas, las de Documentos, las de clave-valor, etc...

NewSQL[19] por otro lado se concibe para usarse en operaciones de carga ligera y repetitivas, principalmente con el fin de ser utilizadas en el procesado en linea.

En este caso, buscamos una base de datos común y al uso y diseñada desde la raíz por nosotros mismos. Por esto mismo la mejor opción es asentarnos con una base de datos textitSQL ya que todo el equipo ha trabajado previamente con ella y nos proporciona una base sólida y bien documentada para nuestra aplicación.

2.1.3. Relacional vs No-relacional

Por último, es importante recalcar que diseñaremos una base de datos relacional. Esta decisión se debe a que el objetivo es tener relaciones entre Regiones, Eventos y Clientes a groso modo y utilizar una base de datos no-relacional complicaría más esta tarea aunque pudiera beneficiarnos en otros aspectos.

Cabe destacar que una base de datos relacional, como su nombre indica, relaciona tablas entré si utilizando claves primarias y claves ajenas, en cambio las no-relacionales no nos aportan esta funcionalidad.

2.1.4. Conclusión

En definitiva, buscamos utilizar una herramienta de bases de datos que sea Relacional, utilice *SQL* y cumpla las características de Alta disponibilidad y consistencia.

Encontramos diversas alternativas como podría ser **MicrosoftSQL** o **Azure SQL database**. En este caso, usaremos los servicios de **Amazon**: *AWS*, para almacenar una base de datos en la nube utilizando **PostgreSQL** ya que cumple con nuestras necesidades a la perfección y nos proporciona una serie de ventajas muy útiles para el proyecto.

PostgreSQL[15] nos proporciona una gran variedad de tipos nativos, de los cuales haremos uso, dado que trabajaremos con distintos tipos de polígonos e imágenes. Además presenta la opción de crear *triggers* los cuales activarán acciones pre-establecidas en la base de datos al realizar una modificación sobre una tabla especifica. Estos *triggers* podrán desde ejecutar transformaciones especiales hasta ayudarnos a solucionar errores con nuestros datos de manera personalizada.

2.2 Backend

En cuanto a la selección de *Backend* se refiere, hemos de tener en cuenta que esta decisión afectará por igual a nuestra aplicación y al visualizador web, considerando que por decisión de la empresa esos dos servicios han de estar centralizados ya que son, en cierto modo, dependientes entre sí. Tenemos a nuestra disposición una buena variedad de *frameworks* muy útiles y preparadas para cubrir nuestras necesidades, en este caso discutiremos la elección entre *.NET*, *Spring* y *Django*.

2.2.1. .NET

Este *framework* [2] es una extensión del lenguaje C desarrollado por Microsoft a finales de los 90 e introducido por primera vez en el año 2002. El objetivo de Microsoft era eliminar todo el manejo de memoria y seguridad necesario para lenguajes como C o C++.

.NET [3] nos provee de muchas librerías diferentes, en nuestro caso haríamos uso de ASP.NET, librería dedicada al desarrollo web. Además de ser un framework rápido, de bajo coste y con un gran soporte del lenguaje, ASP.NET es compilado, lo cual nos proporciona fácil resolución de errores vía los errores de compilación, además de proporcionarnos con las mismas funcionalidades que un lenguaje interpretado.

Por otro lado, ASP.NET nos lo pone complicado a la hora de utilizar framework en el apartado *frontend* de la aplicación dado que funciona mejor utilizando una combinación de frontend y backend desarrollada en el mismo entorno y utilizando su versión interna de *html* con la sintaxis proporcionada por ASP.NET. En el contexto externo, este framework funciona muy bien con ciertas bases de datos, pero no con la elegida en la sección previa 2.1.

2.2.2. Spring

Spring nos proporciona un modelo de programación y configuración de aplicaciones basadas en *Java*, en cualquier tipo de plataforma de implementación. La fortaleza más importante de *Spring* es el soporte de infraestructura que aporta a nivel de aplicación, dado que se elimina complicaciones a los desarrolladores permitiéndoles centrarse en el desarrollo de la lógica sin tener que preocuparse por vínculos innecesarios con entornos de implementación específicos.

Spring[4] nos provee de una gran cantidad de tecnologías y utilidades como inyección de dependencias, eventos, validación, etc... Al igual que en *.NET* podemos encontrar *Spring MVC* un web *framework* que nos permitiría desarrollar el *frontend* utilizándolo de forma interna en este caso siendo más permisivo a la hora de incluir *framework* de *frontend* externos. Por otro lado también nos aporta una gran cantidad de librerías de testeo y del mismo modo es un lenguaje compilado.

2.2.3. Django

Django[5] es un *framework* para el lenguaje de programación *Python* diseñado con la intención de agilizar y facilitar la vida a los desarrolladores de *Python* ya que no requiere de una gran configuración del sistema para poder tener una aplicación funcional. *Django* utiliza una versión modificada del *MVC* llamada *MVT* el cual nos proporciona una forma flexible de separar la capa de contexto y lógica.

Por otro lado cabe destacar su alta velocidad de procesado, su robusta seguridad sin necesidad de configuraciones complejas, su eficiente escalabilidad y un *framework REST* el cual facilita la creación de *APIs REST* enormemente. Por otro lado hay que tener en cuenta que no es muy eficiente en aplicaciones y proyectos pequeños puesto que de base necesita una alta capacidad de procesado y ancho de banda para funcionar correctamente y la falta de una convención, ya que *Python* no es un lenguaje compilado y bastante libre para los desarrolladores nos podemos encontrar con que cada proyecto esta programado de maneras diferentes lo cual puede complicar su lectura y entendimiento.

2.2.4. Conclusión

Una vez revisados los *frameworks* más populares podemos concluir que lo son con buenos motivos. En el desarrollo de nuestra aplicación tenemos que tener en cuenta las necesidades y requerimientos de la empresa y las decisiones tomadas en otros campos. En la sección anterior 2.1 decidimos el uso de *PostgreSQL* como base de datos y en la sección siguiente 2.3 elegiremos un *framework* de *frontend*. Esto nos limita un poco la elección si

queremos aprovecharnos al máximo de uno de ellos. Por otro lado *Python* forma gran parte de este proyecto ya que el *plugin* tendrá que ser desarrollado en este lenguaje. Esto nos facilita en gran medida la elección, pues podemos pues concluir en el uso de *Django* como *framework* para nuestro *backend*.

Como se menciona previamente en esta sección, *Django* es una muy buena elección para la creación de *APIs REST* y nos facilita la creación de proyectos eliminando gran parte de la complicación de las configuraciones iniciales, lo cual es una gran ventaja para la empresa.

2.3 Frontend

A la hora de seleccionar nuestro *framework* de *frontend* tenemos muchísimas alternativas hoy día, aquí discutiremos las más utilizadas a nivel profesional y aquellas adecuadas para el uso de la empresa.

El *frontend* se dividirá en dos partes, por un lado encontramos la interfaz de la herramienta propia, la cual será utilizada por los analistas y por otro lado, la otra parte del proyecto de la empresa, el visualizador de las imágenes analizadas en la herramienta. Hablaremos de ambos ya que nuestra herramienta interactuará de forma directa con el visualizador.

2.3.1. Herramienta de análisis

En este caso no se nos presentan muchas opciones válidas para nuestro proyecto, esto es debido a que *QGIS*, la herramienta sobre la que utilizaremos la nuestra, no admite código que no sea en *Python*, por eso tendremos que basarnos en las librerías del mismo para poder desarrollar nuestra aplicación.

Por otro lado, la propia herramienta de *QGIS* nos proporciona de forma directa una herramienta de desarrollo de interfaces para *Python* utilizando el módulo *PyQt5*, uno de los muchos módulos de generación de interfaces disponibles para este lenguaje.

Realmente *QGIS* nos permite utilizar cualquier otro módulo como podría ser *TKinter*, pero la elección es obvia siendo que *PyQt5* es una de los mejores módulos a nuestra disposición y además tiene una integración bastante directa con la herramienta sobre la cual vamos a desarrollar la nuestra.

2.3.2. Visualizador

Por otro lado, el visualizador será una herramienta web y no tiene ningún tipo de limitación. En este caso podríamos desarrollar en *Javascript* plano, lo cuál evaluando el tamaño del objetivo deseado sería una locura. También tenemos a nuestra disposición diferentes *frameworks* disponibles como son *Angular, React, NextJs o VueJs*.

De esta manera, la disputa real la encontramos entre *Angular y React*, dos *frameworks* muy utilizados en el sector, ambos permiten la utilización de estados (de esto hablaremos más tarde) y principalmente nos presentan la capacidad de crear componentes individuales y reutilizables, siendo esta su característica principal.

2.3.3. Angular

Angular[6] es una herramienta de desarrollo de software introducida por *Google* en el año 2009 y es uno de los *frameworks* más utilizados a día de hoy. Entre su apartado

tecnológico podemos encontrar una gran variedad de herramientas útiles para cualquier tipo de desarrollo. *Angular* destaca gracias a una enorme cantidad de librerías soportadas por la comunidad.

Entre estas librerías encontramos *RxJs*, una librería encargada de manejar peticiones asíncronas con eventos múltiples y *Angular CLI*, una interfaz de consola de comandos que facilita enormemente el manejo y creación de proyectos.

Angular posee una arquitectura orientada a componentes que permite encapsular las funcionalidades en diferentes jerarquías. Esto hace que tenga una buena reusabilidad, y que sea muy fácil de leer. Además permite una buena integración de pruebas automáticas, lo cual trae consigo una buena mantenibilidad.

Además *Angular* trae consigo *TypeScript*, mejorando de este modo todas las características mencionadas previamente, ya que este lenguaje añade funcionalidades de control en compilación, mejor manejo de errores, tipado e interfaces, convirtiendo así el lenguaje de *Javascript* en uno más sólido y coherente.

Otras funcionalidades que nos aporta Angular pueden ser la inyección de dependencias, el manejo de estados mediante *state stores* y un modelo de interfaces basado en *MVC*.

Por otro lado, Angular es un lenguaje relativamente más complejo de lo habitual, haciendo así que la curva de aprendizaje sea bastante pronunciada y no sea muy útil para proyectos pequeños donde no se quiera invertir tiempo en el aprendizaje. Además del mismo modo, ejerce en cierta manera la sobre-ingeniería en ciertos aspectos que a priori no es necesaria en proyectos del calibre que nosotros manejamos con nuestra aplicación.

2.3.4. React

React[7] es un *framework* creado por *Facebook* utilizado por primera vez en la misma plataforma en el año 2011 y posteriormente en *Instagram* el año 2012.

Del mismo modo que *Angular* 2.3.3, *React* basa su arquitectura en la creación de componentes reutilizables. Teniendo de este modo unas características muy similares a las mencionadas previamente con *Angular*.

Por otro lado, encontramos que *React* es un lenguaje bastante menos verboso y complejo. Esto trae consigo un gran potencial, dado que reduce enormemente la curva de aprendizaje, pero al mismo tiempo puede entorpecer el desarrollo si no se trata con cuidado.

React puede llegar a producir un código muy limpio y simplificado pero también puede crear amalgamas de componentes apilados unos encima de los otros sin ninguna facilidad para entender que esta sucediendo.

Funcionalmente *React* es bastante similar a *Angular*, solo que trae consigo su propio manejo de estados interno, cada componente puede manejar tantos estados como deseé y no necesita de una *store* para funcionar. Además estos estados pueden ser transmitidos entre componentes y ser actualizados tanto por los padres como los hijos, proporcionando de nuevo una mayor facilidad en el desarrollo.

Estos estados consisten de variables que permiten modificar los propios componentes que están siendo renderizados en la aplicación web, para poner un ejemplo, un estado podría ser un contador el cual tras pulsar un botón incremente su numero, el framework estaría renderizando la web utilizando el valor del estado del contador, y cada vez que este se actualice, automáticamente renderizaría la web para mostrar el nuevo estado actualizado.

A pesar de no necesitar una *store* inicialmente, *React* cuenta con una librería *REDUX*, utilizada principalmente para el control de estados globales.

Además, en lugar de generar directamente elementos *HTML* y modificar el *DOM*, los componentes generados por *React* crean un modelo del mismo en la memoria. Una vez generado este *DOM* virtual, *React* busca las diferencias entre el real y el virtual y realiza únicamente las modificaciones necesarias en el *DOM* real, mostrando así solo los apartados del documento que hayan sido modificados.

2.3.5. Conclusión

En este capítulo hemos discutido todas las tecnologías disponibles para nuestro proyecto teniendo en cuenta inclusive la parte del frontend del visualizador web ya que la empresa pretende que todos los servicios interactúen entre sí de forma adecuada y eficiente. Tras una ardua investigación sobre las necesidades de la empresa y las tecnologías presentes se ha llegado a la conclusión de que las tecnologías a utilizar en el proyecto serán las siguientes.

El servidor será desarrollado en *Python* utilizando el *framework* de *Django*, por otro lado nuestra base de datos utilizará *PostgreSQL*. El visualizador se construirá haciendo uso de *React* como *framework* principal y el *Plugin* de QGIS en *Python* haciendo uso del módulo **PyQt5** para desarrollar las interfaces gráficas con mayor facilidad y compatibilidad con *QGIS*.

CAPÍTULO 3
DESARROLLO

En este capítulo seguiremos paso a paso el proceso de desarrollo, explicando la metodología escogida, los requisitos y su especificación, el diseño de los diversos entornos de la aplicación y su implementación final además de las pruebas automáticas realizadas.

3.1 Metodología

La metodología es un factor crucial a tener en cuenta a la hora de plantear el desarrollo de una aplicación. Dicho desarrollo se ve impulsado gracias a los principios definidos en la metodología seleccionada.

Hay muchas metodologías diferentes, entre ellas aparecen algunas como la metodología ágil, muy trabajada a lo largo de la carrera y una de las más utilizadas hoy en día, la metodología incremental, en espiral, etc... [22].

Estas metodologías se separan en diferentes categorías, podemos encontrar las siguientes.

- Tradicional
- Evolutiva
- Orientado a Objetos
- Basadas en la reutilización
- Ágiles
- Para sistemas web

De entre todas estas, nosotros haremos uso de la categoría de metodologías tradicionales [1] [23]. Estas son consideradas realmente efectivas en proyectos de innovación como el que se presenta en este TFG. Además, son sencillas y muy intuitivas de implementar ya que llevan en funcionamiento mucho más tiempo que otras metodologías.

3.1.1. Prototipos

Dentro de las metodologías tradicionales encontramos la metodología de prototipos, esta será nuestro foco principal en el desarrollo de la aplicación. La metodología de prototipos consiste en generar implementaciones más sencillas a la final para ir recibiendo de forma continua *feedback* del cliente o usuario de dicha aplicación. De esta forma entendemos que la metodología de prototipos es una de las más flexibles dentro de la rigidez de las metodologías tradicionales, y en cierta medida nos será muy útil a lo largo del proyecto.

Esta implementación encaja a la perfección con las necesidades de la empresa, dado que los clientes forman parte del equipo. De este modo somos capaces desarrollar diversos prototipos a lo largo del desarrollo con el fin de obtener un *feedback* prácticamente instantáneo por parte de los analistas mientras se continua con la implementación.

Gracias a esta metodología, ciertas necesidades de la aplicación podrán variar según el *feedback* recibido a lo largo del desarrollo de la misma, así mismo esta podrá ajustarse mejor a las necesidades reales de los analistas, ya que la respuesta será inmediata y de primera mano.

Enfoque

En la metodología de prototipos encontramos 3 tipos diferentes de enfoque para el desarrollo de la aplicación. El enfoque **desechable** donde el prototipo se crea con la intención de ser eliminado posteriormente, el enfoque **evolutivo** en el que el prototipo deberá ser eventualmente el producto buscado y el enfoque **mixto** donde se combinan los dos enfoques previos de manera combinada.

Estos tres enfoques poseen cada uno sus ventajas y desventajas, pero dada la situación del proyecto el adecuado salta a la vista. A lo largo del desarrollo haremos uso del enfoque Evolutivo, generando una primera aplicación base mediante la cual obtener *feedback* y proveer de ciertas utilidades a los analistas mientras se continua con el resto del proyecto.

El enfoque **evolutivo** siguiendo el modelo **iterativo**[11] proporciona una buena adaptación al proyecto mientras se va evolucionando el prototipo inicial para obtener la aplicación final deseada. Este nos permite solucionar los problemas encontrados mediante el uso de la aplicación sin generar una gran curva de aprendizaje para los usuarios, además nos presenta una muy buena flexibilidad para obtener nuevos requisitos.

El modelo iterativo[24] es esencial en el desarrollo de este proyecto ya que nos permitirá entregar de forma consistente diferentes versiones del software aportándonos la flexibilidad necesaria que las metodologías tradicionales echan en falta. De este modo seremos capaces de cumplir los plazos de entrega mediante prototipos que nos servirán a modo de obtener nuevos requisitos y posibles fallos de los sistemas.

Entregas

A lo largo del desarrollo de la aplicación se realizaron cuatro entregas diferentes de prototipos, cada uno obteniendo mejoras del anterior. En la sección siguiente 3.7.2 se encuentran todas las herramientas de las que se hablará a lo largo de las entregas y su composición. Cabe destacar que este desarrollo sucedió tras la implementación del *backend* de la aplicación, dado que este fue desarrollado previamente al inicio de la aplicación *frontend*.

Primera Entrega

Para la primera entrega se propuso trabajar en todas las herramientas de generación de recursos. Estas herramientas son las que permitirán a los analistas generar nuevos eventos 3.7.2, regiones 3.7.2 y fuentes 3.7.2. Ya que se considera esta la parte esencial

además de ser necesario para el buen funcionamiento de la otra aplicación de la empresa, el visualizador. Además se implementará una primera versión del *dashboard* donde se encontrarán todas las herramientas.

La implementación de estas herramientas fue bastante sencilla y sin muchas complicaciones, lo único necesario era crear una conexión entre nuestra herramienta y la *API* creada previamente 3.7.1. Esta conexión se realiza mediante el uso de peticiones con el módulo *requests* de *Python*.

Una vez realizada esta conexión, todas las peticiones de creación, edición y obtención de eventos, regiones o fuentes funcionan a la perfección. Finalmente el último paso para un correcto intercambio de información será transformar los datos obtenidos por la herramienta a un objeto de tipo *JSON*. Haciendo uso del módulo *json* de *Python* podemos realizar una conversión sencilla de un grupo de valores a un *JSON*.

Para la implementación del *dashboard* primeramente se hará uso de un panel sencillo conteniendo botones para abrir las herramientas en una nueva ventana. Esto será posteriormente modificado en la tercera entrega, donde observaremos el diseño definitivo de la misma 3.7.2. Además se incluye una pequeña herramienta de inicio de sesión 3.10.

Tras esta entrega se obtiene como *feedback* ciertos errores y algunas mejoras que se aplicarán durante la segunda entrega. Primeramente un error que no permitia la creación de eventos iguales con nombres distintos, a primera vista puede parecer intencionado, pero dado que varios clientes pueden requerir un mismo análisis y no se pretende compartir información entre clientes, lo idoneo sería generar otro evento nuevo. Seguidamente un pequeño *bug* que forzaba a los usuarios a iniciar sesión de nuevo al cerrar *QGIS*.

Por último se obtienen ideas de los analistas para mejorar la estabilidad de las interfaces y una mejor separación de los elementos.

Segunda Entrega

A lo largo de la segunda entrega, se pretende incluir las herramientas utilizadas para preprocesar imágenes además de las utilizadas para transmitir las mismas a los proveedores o a nuestro propio servidor. Esta tarea será ciertamente más compleja que la realizada en la entrega anterior y por eso este prototipo será el más importante de todo el desarrollo.

Se comienza esta segunda fase arreglando los errores encontrados en la previa y aplicando las ideas obtenidas del *feedback* de los analistas. Los errores son relativamente sencillos de arreglar, modificando las propiedades de la base de datos podemos permitir la generación de eventos con datos iguales y al observar con detenimiento la herramienta de inicio de sesión se descubre que la *checkbox* que permitía mantener la sesión iniciada no estaba funcionando correctamente.

Seguidamente se inicia la implementación de las herramientas de subida de imágenes 3.7.2 y 3.7.2. Para esto utilizaremos el previamente mencionado módulo de *requests* con el fin de publicar recursos a las páginas de los proveedores o nuestro propio servidor utilizando los parámetros introducidos en las interfaces de las herramientas.

Ambos procesos tendrán que comunicarse con nuestra *API* 3.7.1 y gracias a la configuración de los *CORS* podremos reenviar estos recursos a los proveedores asociados.

Finalmente se introducen las herramientas de edición de imágenes 3.7.2, 3.7.2 y 3.7.2. Para poder utilizar las herramientas de edición primeramente se habrán de tratar las

imágenes obtenidas de los proveedores dado que la aplicación *QGIS* solo admite ciertos tipos de capas con las que trabajar.

Mediante la herramienta de preprocesado de *Sentinnel* 3.7.2 se obtendrán las imágenes de los proveedores vía una *url* introducida por el usuario y se le aplicarán los parámetros establecidos por el mismo, haciendo uso de un algoritmo especial de la empresa estas imágenes obtendrán una mejor calidad para el análisis y serán usables dentro de la aplicación de *QGIS*.

Por otro lado las herramientas de edición permitirán a los analistas modificar estas capas previamente generadas haciendo uso de diferentes parámetros. Para poder conseguir este funcionamiento, es necesario trabajar con las funcionalidades provistas por *QGIS* dentro de nuestro entorno, automatizando procesos triviales para que los usuarios únicamente tengan que introducir los parámetros de edición deseados y sean aplicados automáticamente.

Tras esta entrega, los usuarios manifiestan la necesidad de un mejor manejo de ventanas para las herramientas dado que estas solo les permitían utilizar una herramienta al mismo tiempo y no eran capaces de analizar y procesar diversas imágenes simultáneamente ralentizando así su trabajo. Además se encuentran diversos *bugs* relacionados con la introducción de parámetros en las distintas herramientas de edición.

Tercera Entrega

En esta tercera entrega se se solucionan rápidamente los *bugs* anteriormente mencionados, observando con cautela las diferentes interacciones entre el usuario y la herramienta se obtiene que se produjeron errores con los valores introducidos siendo cambiados de orden a la hora de ser enviados.

Por otro lado a lo largo de esta entrega se desarrolla la herramienta de generación de reportes. Principalmente esta no debía requerir demasiado trabajo pero como se mencionará en el diseño de la misma 3.7.2 fue necesaria la creación de una librería de *Python* para poder interactuar de la forma deseada con los archivos *PDF*.

Para este generador de reportes se requiere poder introducir varios eventos diferentes para un mismo cliente siendo este reporte un informe final de todo un periodo de análisis.

Al final de esta tercera entrega no se encuentra ningún *bug* relevante y se sugieren mejoras estéticas en los reportes que son comunicados al diseñador gráfico de la empresa para posteriormente ser modificados en el producto final.

Cuarta Entrega

En la entrega definitiva se prepara la aplicación para el uso de la inteligencia artificial que esta siendo desarrollada en paralelo a esta herramienta. De esta forma se introducen las dos herramientas que permitirán a los analistas subir y consultar las imágenes disponibles para el entrenamiento de dicha inteligencia artificial.

Con la herramienta de Upload Patches 3.7.2 los analistas podrán enviar ciertas imágenes preprocesadas al servidor de la inteligencia artificial del mismo modo que se enviaban archivos ya analizados a nuestro propio servidor o a los proveedores en las herramientas desarrolladas en la primera entrega.

Por otro lado mediante la herramienta Patch Table los usuarios pueden consultar las imágenes presentes en el servidor de dicha inteligencia artificial.

Finalmente tras esta entrega se obtiene el producto final de la herramienta de análisis, no sin antes solventar dos errores encontrados en el uso de la aplicación de generación de reportes, estos *bugs* aparecen tras un uso continuado de la misma donde se observa que tras generar varios reportes seguidos ciertos elementos gráficos aparecen en ubicaciones establecidas en los reportes creados anteriormente. Esto se soluciona tras una ardua investigación la librería creada para la generación de dichos archivos *PDF*, donde se observa que las plantillas utilizadas para crear nuevas páginas son capaces de mantener los datos si estos no son modificados. Sencillamente se añade un reinicio de los datos de las páginas tras la generación de cada documento y su funcionamiento vuelve a ser el esperado.

3.2 Organización

Para un buen desarrollo de la aplicación será también necesario hacer uso de metodologías organizativas, estas facilitan enormemente la evolución del proyecto, ya que un proyecto limpio y ordenado es mucho más fácil de navegar y hacer *debug*.

3.2.1. Git flow

Para este proyecto se ha decidido del mismo modo, organizar de forma correcta y adecuada el proyecto al completo, para esto ha sido empleado el *Git Flow*.

Esta es una metodología de organización creada por Vincent Driessen y basada en *Git*. Esto quiere decir que define un modelo de ramificaciones estricto al rededor de todo el proyecto, proporcionando así un marco robusto donde sostener el proyecto en su totalidad.

Esta es una metodología ideal para proyectos que cuentan con fechas de lanzamiento programadas y muy fácil de comprender si se tienen conocimientos previos de *Git*, puesto que hace uso de comandos básicos presentes en el uso habitual de dicha herramienta. Además, se complementa de manera adecuada con la metodología de prototipos 3.1.1 utilizada en el desarrollo de la aplicación.

Master y Dev

Para hacer uso de esta metodología se definen dos ramas principales: **Master** y **Dev**. En la rama de master encontramos registrado el historial de lanzamientos de la aplicación, es decir, cada salida de una nueva versión completa se integra en master y se pone a disponibilidad de los usuarios. Mientras tanto en la rama dev los desarrolladores van integrando las nuevas funcionalidades provenientes de las distintas ramas de características para después, agrupar ciertas de estas funcionalidades en una nueva versión integrada y probada primeramente en dicha rama y ser subida como una nueva versión de la rama master.

Ramas de características

Las ramas de características, mejor conocidas como *feature branches* son aquellas en las que los desarrolladores integran nuevas características del proyecto de forma individual.

Cada rama dependerá de dev y se utilizará para implementar una característica descrita en una tarea. Esto se consigue generando una nueva rama en el proyecto utilizando una nomenclatura especial. En nuestro caso, dado que utilizaremos Trello 3.2.2 como manejador de tareas y este nos permite integrar la metodología *Git flow*, podremos generar numeración automáticamente para cada tarea y utilizar esta para diferenciar las diferentes ramas de características. Un ejemplo de rama de características sería el siguiente:

f147_assign_client_events

Ramas de solución de errores

Estas ramas, también conocidas como *hotfix branches*, se utilizarán, como su nombre indica, para solucionar errores presentes en el proyecto. Estas ramas se crean con el fin de arreglar ciertos errores urgentes que no puedan permitirse esperar a la publicación de la próxima versión de la aplicación.

Del mismo modo que las ramas de características 3.2.1, estas poseerán una nomenclatura identificativa, esta será la siguiente:

h185_fix_user_label

Ramas de versión

Este tipo de ramas, también conocidas como *release branch* se utilizan para agrupar todos los contenidos de la siguiente versión a publicar. En estas ramas se realiza la agrupación de todas las ramas de características previstas para el lanzamiento próximo. En ellas también se realiza el testeo completo de la versión. Su nomenclatura especial será la siguiente:

r2_1

3.2.2. Trello

Como se menciona previamente, haremos uso de Trello como herramienta de generación de tareas y organización.

¿Que es Trello?

Trello 3.1 [8] es una herramienta de gestión de trabajo para diseñar planes, proyectos y organizar flujos de trabajo mientras se hace seguimiento de los mismos de forma visual. Esta herramienta esta basada en el *método Kanban*, organizando las actividades por medio de tableros visuales compuestos por tarjetas y tareas asignadas por columnas. Todo esto siendo personalizable, además de contar con soporte para *plugins*.

¿Como usar Trello?

Usar Trello es muy sencillo, crearemos una cuenta en su página oficial [9] y crearemos un nuevo tablero. En este, podremos invitar a todos los colaboradores, añadir o eliminar columnas y renombrarlas, establecer códigos de colores y crear nuevas tarjetas.

Para crear una nueva tarjeta bastará con hacer click en el símbolo +, ponerle un nombre, una descripción y asignarle un color. Una vez avancemos con la tarea la podremos ir moviendo por las diferentes columnas.

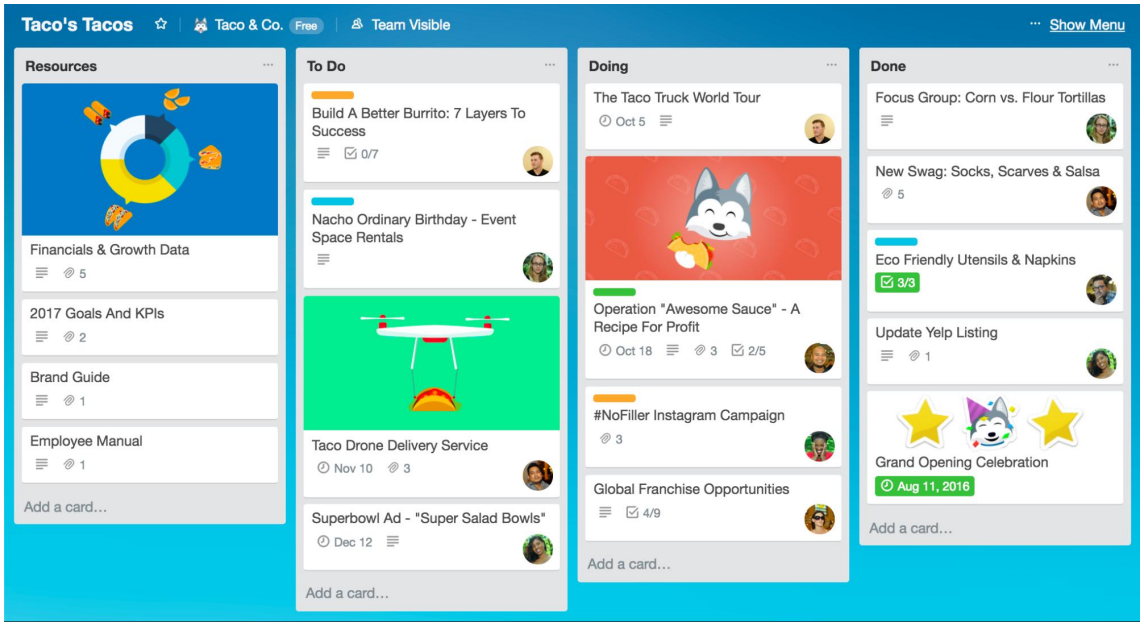

Figura 3.1: Ejemplo de tablero de Trello

En nuestro caso el uso de Trello irá basado al rededor del *Git flow* 3.2.1 implementado en el proyecto así que cada tarea tendrá un número asignado a su nombre que utilizaremos más tarde en la creación de ramas. Por último cabe destacar que las columnas utilizadas en el tablero del proyecto serán las siguientes:

TODO	Para las tareas pendientes de realizacion
WIP	Para las tareas que esten en desarrollo actualmente
READY FOR REVIEW	Para las tareas listas para ser revisadas
REVIEW	Para las tareas siendo realizadas
READY FOR TEST	Para las tareas listas para ser testeadas
TESTING	Para las tareas siendo testeadas
READY FOR MERGE	Para las tareas listas para ser mergeadas a la rama de dev

3.3 Especificación de requisitos

En está sección se tratará la especificación de requisitos con el objetivo de definir las necesidades de la aplicación de forma correcta, tal que permita obtener un desarrollo eficiente y efectivo. Respondiendo de forma clara y concisa las propuestas del estándar *IEE830*[17] para una buena especificación de requisitos.

3.3.1. Casos de uso

A continuación se mostrará el diagrama de casos de uso 3.2 generado para nuestra aplicación. En este se observan las necesidades de los Analistas y las diferentes acciones que deberán ser capaces de realizar haciendo uso de nuestra herramienta.

Un diagrama de casos de uso es un elemento gráfico empleado en el análisis y obtención de requisitos de un sistema con el fin de definir los requerimientos funcionales del mismo. Estos diagramas pueden estar compuestos por varios elementos **actor** que representarán una vista genérica de los posibles usuarios de la aplicación, estos podrían ser empleados, compradores, analistas, etc... Por otro lado aparecen en el diagrama los propios casos

de uso, estos son acciones que podrá llevar a cabo el actor al que estén relacionados y podrán relacionarse con varios actores e incluso otros casos de uso mediante tres tipos de relaciones diferentes.

Estas relaciones pueden ser **inclusión o *include***, que representan una unión entre dos funciones enriqueciendo una de ellas haciendo así el caso de subfunción de la misma. El siguiente tipo de relación será **extensión o *extends***, esta incorpora implícitamente el comportamiento de la función relacionada, siendo esta subfunción aquella que solo sucederá condicionalmente tras la primera haya sido ejecutada. Por último encontramos las relaciones de tipo **herencia o *inheritance***, mediante el uso de dicha relación obtendremos un caso de uso genérico completado por los casos de uso asociados al mismo, teniendo estos últimos las propiedades presentadas por el primero y completando su funcionalidad.

De este modo en el diagrama de casos de uso presente en esta especificación, encontraremos dos actores (analista y visualizador) seguidos de diversos casos de uso, siendo algunos relacionados con otros haciendo uso de la inclusión. Seguidamente encontramos las descripciones tabladas de cada caso de uso individual.

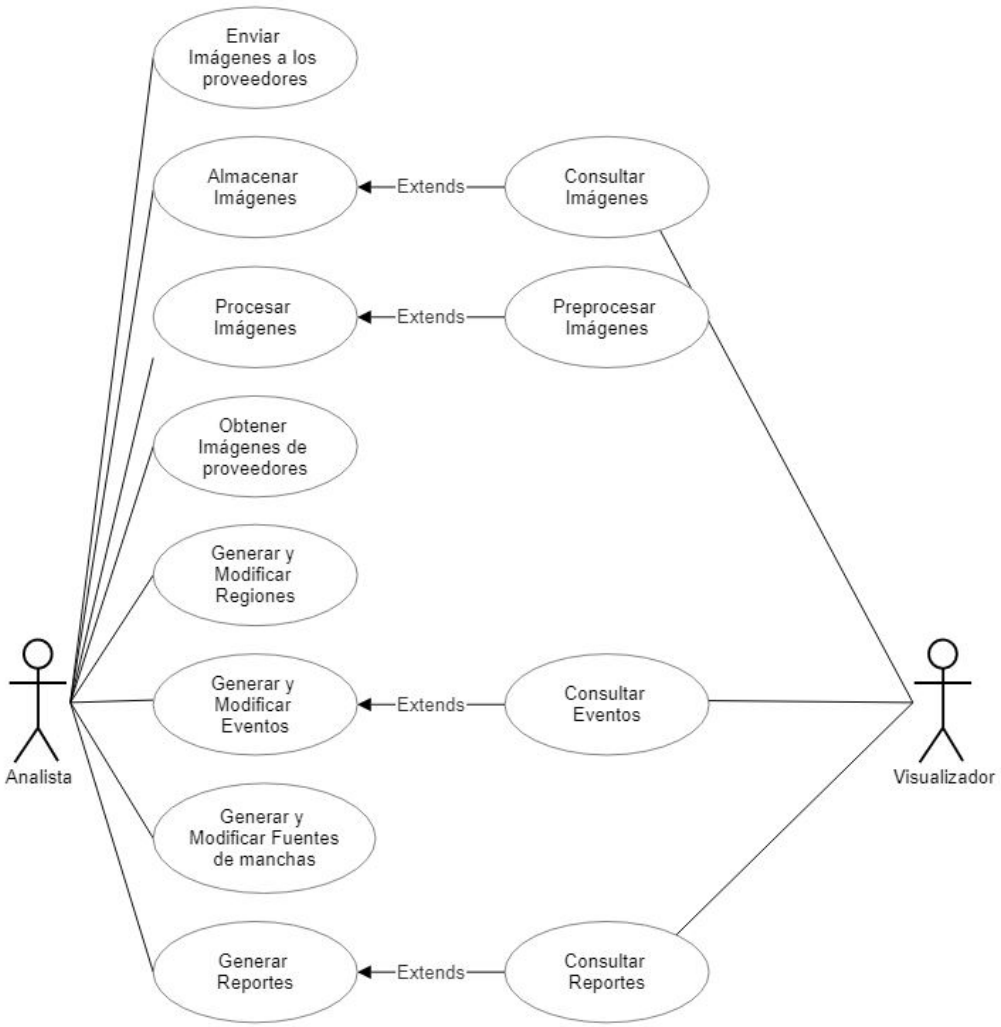

Figura 3.2: Caso de Uso de la aplicación

Identificador	CU01
Nombre	Generar y modificar eventos
Actor	Analista
Descripción	El analista ha de poder generar y modificar eventos a su antojo siendo estos almacenados en la aplicación.

Identificador	CU02
Nombre	Generar y modificar regiones
Actor	Analista
Descripción	El analista ha de poder generar y modificar regiones a las cuales asignar clientes.

Identificador	CU03
Nombre	Generar y modificar fuentes
Actor	Analista
Descripción	El analista ha de poder generar y modificar fuentes originarias de manchas, indicar su posición y características.

Identificador	CU04
Nombre	Generar reportes
Actor	Analista
Descripción	El analista ha de poder generar reportes en formato pdf, con traducción a varios lenguajes y añadiendo los eventos deseados para cierto cliente.

Identificador	CU05
Nombre	Obtener imágenes de proveedores
Actor	Analista
Descripción	El analista ha de poder obtener imágenes almacenadas en los servidores de los distintos satélites proveedores.

Identificador	CU06
Nombre	Almacenar imágenes
Actor	Analista
Descripción	El analista ha de ser capaz de almacenar en la aplicación imágenes ya procesadas o pre-procesadas para el entrenamiento de una inteligencia artificial.

Identificador	CU07
Nombre	Enviar imágenes a proveedores
Actor	Analista
Descripción	El analista ha de ser capaz de enviar imágenes de cualquier ámbito a los proveedores para usar como almacenaje de las mismas.

Identificador	CU08
Nombre	Procesar imágenes
Actor	Analista
Descripción	El analista ha de ser capaz de procesar imágenes haciendo uso de la herramienta, generando de este modo capas poligonales que muestren manchas encontradas en dicho análisis.

Identificador	CU09
Nombre	Preprocesar imágenes
Actor	Sistema
Descripción	El sistema ha de ser capaz de preprocesar imágenes para ser compatibles con el uso de la herramienta QGIS haciendo así que los analistas puedan realizar el procesado de las mismas.

Identificador	CU10
Nombre	Consultar Eventos
Actor	Visualizador
Descripción	El visualizador ha de ser capaz de obtener eventos generados mediante la aplicación para cada cliente y mostrar todas sus características.

Identificador	CU11
Nombre	Consultar imágenes
Actor	Visualizador
Descripción	El visualizador ha de ser capaz de obtener imágenes almacenadas en la aplicación y mostrarlas a cada cliente individual.

Identificador	CU12
Nombre	Consultar reportes
Actor	Visualizador
Descripción	El visualizador ha de ser capaz de obtener los reportes generados para cada cliente individual y mostrarlos a los mismos además de permitir la descarga.

Una vez obtenidos estos casos de uso, cabe destacar que los casos relacionados con el visualizador no aparecen en mucho detalle ni al completo debido a que se escapa del trabajo realizado para esta herramienta. Debido a que la herramienta engloba también la parte del servidor hemos decidido incluir las funcionalidades que permiten obtener los datos generados por nuestra aplicación.

3.3.2. Obtención de requisitos

Para comenzar una especificación de requisitos hay que definir los métodos a utilizar a la hora de obtener los requisitos. Hay muchos de ellos, como pueden ser encuestas, entrevistas, cuestionarios, lectura de documentos, observación...

En este caso procederemos a la realización de **entrevistas** a los diferentes analistas que forman parte de la empresa y al conjunto de trabajadores habituados al uso de la herramienta *QGIS*, además dedicaremos una parte del tiempo destinado a la especificación a la **observación** detallada de dichos analistas y de la herramienta en cuestión.

Gracias a estas dos herramientas de obtención de requisitos somos capaces de obtener y definir los requerimientos del proyecto, esto se dividen en requisitos funcionales y no funcionales, presentes en las siguientes secciones.

3.3.3. Requisitos funcionales

Los requisitos funcionales forman parte de aquellos requisitos asociados a la funcionalidad de la aplicación, estos en concreto son aquellos más específicos que no afectan de manera directa a un actor de los casos de uso. Los requisitos no funcionales de nuestra aplicación son los siguientes.

Identificador	RF01
Nombre	Obtención de archivos en multiple formato
Caso de uso asociado	CU05 y CU08
Descripción	La herramienta ha de permitir al analista obtener los archivos resultantes de diversas operaciones en formato de archivo *TIFF* o *SHAPEFILE*.

Identificador	RF02
Nombre	Preprocesado adecuado de imágenes
Caso de uso asociado	CU08
Descripción	La herramienta ha de procesar las imágenes obtenidas como entrada de tal forma que el analista pueda utilizar las mismas dentro de la propia aplicación y la herramienta *QGIS* para ser analizadas.

Identificador	RF03
Nombre	Uso de capas de *QGIS*
Caso de uso asociado	TODOS
Descripción	La herramienta ha de permitir al analista utilizar capas generadas dentro de *QGIS* como entrada para las diversas utilidades de la misma.

3.3.4. Requisitos no funcionales

Los requisitos no funcionales determinan el nivel de calidad, propiedades y características que ha de tener un sistema. Estos requisitos no son funcionalidades exactas que hayan de ser desarrolladas, si no requisitos específicos, cuantificables y verificables determinantes de ciertas propiedades necesarias para el funcionamiento adecuado de dicho sistema. Estos requisitos suelen ser analizados y estandarizados a través del modelo ISO/IEC 25010 [10].

Los requisitos obtenidos previamente se muestran en las siguientes tablas y se dividen en función de los diferentes grupos especificados en la ISO/IEC 25010 [10] que son: Adecuación funcional, eficiencia de desempeño, compatibilidad, usabilidad, fiabilidad, seguridad, mantenibilidad y portabilidad.

Identificador	RNF01
Nombre	Acceso controlado
Grupo	Seguridad
Descripción	El acceso a la aplicación de análisis será controlado en todo momento por el administrador de la aplicación y podrá ser modificado en todo momento.

Identificador	RNF02
Nombre	Patrones de Seguridad
Grupo	Seguridad
Descripción	El sistema ha de ser implementado siguiendo una variedad de patrones que favorezcan la seguridad y protección de los datos que forman parte del mismo.

Identificador	RNF03
Nombre	Velocidad de actualización
Grupo	Eficiencia
Descripción	Los datos actualizados por los analistas han de ser visibles por los usuarios del visualizador en menos de un minuto tras la publicación de los mismos.

Identificador	RNF04
Nombre	Trabajo en paralelo
Grupo	Eficiencia
Descripción	La herramienta de análisis tendrá que permitir a los analistas trabajar con diferentes herramientas simultáneamente con el fin de evitar bloqueos.

Identificador	RNF05
Nombre	Muestra de errores
Grupo	usabilidad
Descripción	El sistema ha de contar con una muestra de errores entendible para el lenguaje casual con el fin de permitir a los analistas diferenciar fallos de uso de la herramienta con fallos del programa e informar adecuadamente de los mismos.

Identificador	RNF06
Nombre	Disponibilidad total
Grupo	Fiabilidad
Descripción	El sistema ha de proporcionar una disponibilidad del 99,99 % de las veces que un analista intente acceder a la misma, esto es debido a que la aplicación se utilizará en entornos de presión y bajo cortos periodos de tiempo.

Identificador	RNF07
Nombre	Usable en todas las plataformas
Grupo	Portabilidad
Descripción	El sistema ha de estar disponible en todo tipo de plataforma y sistema operativo en el que la herramienta *QGIS* esté.

3.4 Diseño de base de datos

Para analizar el diseño de la base de datos, utilizaremos un modelado UML sencillo 3.3. En este aparecen los distintos campos de nuestra base de datos con sus tipos y propiedades necesarias.

Cabe destacar que podremos encontrar dos tipos de regiones en la aplicación, una de ellas delimitada por los analistas y otra siendo las regiones reales del mundo.

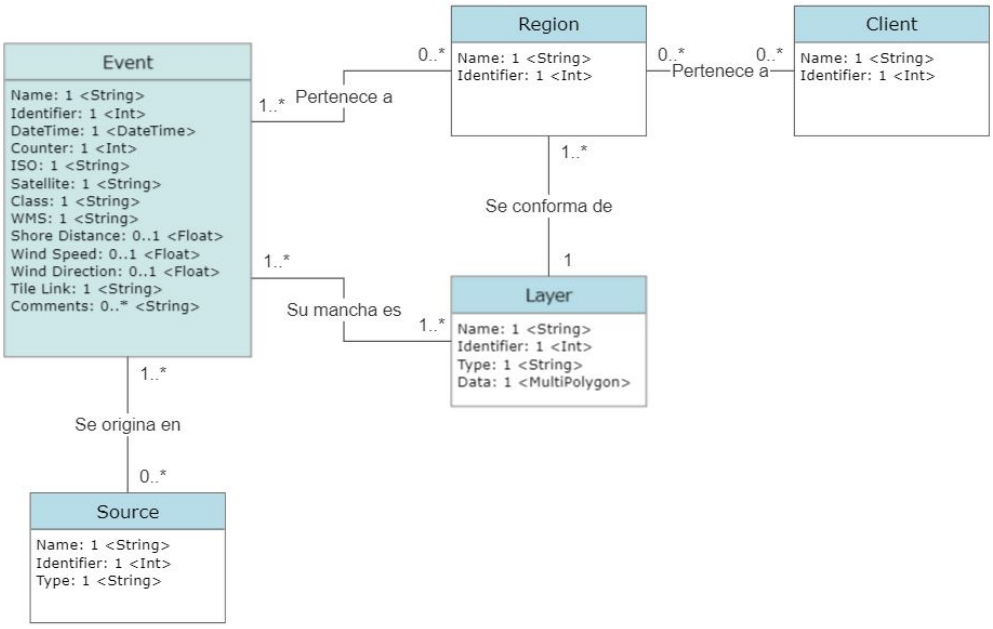

Figura 3.3: Modelado UML de la base de datos

3.5 Diseño de interfaces

Para el diseño de las interfaces se hará uso de una herramienta web llamada **Figma**[20]. La empresa busca centrarse completamente en la utilidad y olvidar en cierto modo un estilo concreto que requiera una gran inversión de tiempo. Por ello las interfaces a diseñar serán sencillas y efectivas en su función. Para consultar las imágenes diseñadas referirse al Anexo A.

3.5.1. Figma

En esta subsección se explicará a groso modo el funcionamiento de la aplicación web Figma 3.4 y como la utilizaremos en el diseño de las interfaces gráficas.

Esta herramienta se divide en varias secciones utilizables muy intuitivas. En el centro de la aplicación 3.4 se puede encontrar un lienzo donde se colocarán todas las figuras, textos y distintos diseños realizados. Estos elementos podrán ser arrastrados y modificados en diversos aspectos a través de los paneles laterales.

En la parte superior de la herramienta se encuentra una barra la cual presenta diferentes apartados, a la izquierda de este panel, aparecen los diferentes elementos que se pueden añadir al lienzo central. En el centro aparece el nombre del tablero sobre el que estamos trabajando, este se puede modificar. Por último en el lado derecho se encuentra el usuario utilizando la herramienta en este momento, el nivel de zoom y un botón para compartir el tablero.

En el panel lateral izquierdo, como ha sido mencionado previamente, se podrán modificar ciertos aspectos de las figuras introducidas en el lienzo. En este panel se podrá renom-

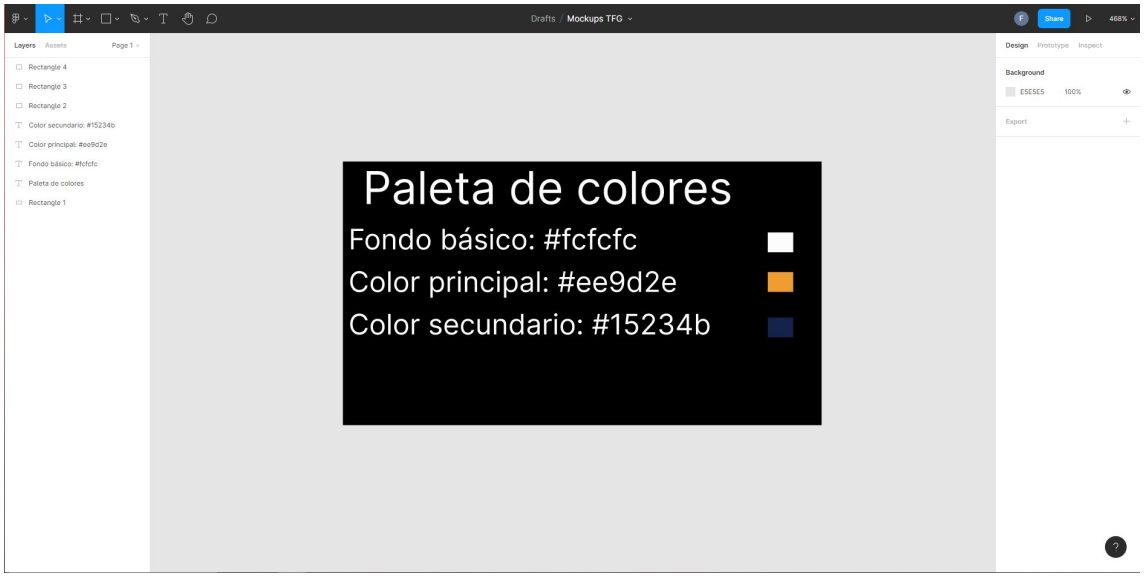

Figura 3.4: Ejemplo de la herramienta web de diseño Figma

brar las dichos elementos, agruparlos para generar un elemento compuesto y modificar el nivel de prioridad de cada capa o grupo.

En el panel lateral derecho, del mismo modo, encontramos modificadores de los elementos. En este caso, al seleccionar un elemento en el lienzo o en el panel lateral izquierdo, podremos modificar las propiedades gráficas del mismo.

3.6 Preparación del entorno

Para comenzar a desarrollar nuestra aplicación podremos afrontarlo de varias maneras, en este caso veremos como generar una plantilla básica de *Django* y ajustarlo a las necesidades de nuestro proyecto y como utilizar el Plugin-tool de *QGIS* para generar la base de nuestro *frontend* realizando las conexiones adecuadas con la base de datos. En este capítulo veremos paso por paso el software necesario y como instalarlo.

3.6.1. Instalación del software

En cuanto a software se refiere, nos bastará con tener la versión deseada de Python en nuestro ordenador de trabajo y un *IDE* a nuestro gusto, este caso nos valdremos de *Visual Studio Code*, el cual nos permite descargar herramientas que mejorarán nuestra experiencia de desarrollo. Además utilizaremos Git para compartir el código en la empresa mediante *BitBucket*. Para instalar este software procederemos con el siguiente comando:

```
$ sudo apt install git-all python3.7 code
```

Por otro lado, dado que usaremos *PostgreSQL* necesitamos instalarlo en local para no modificar los datos reales y alguna forma de visualizar los datos y realizar consultas sobre ellos. Primeramente necesitaremos descargar *PostgreSQL* para nuestro dispositivo con la siguiente linea:

```
$ sudo apt install postgresql postgresql-contrib
```

Este comando también instalará automáticamente el software en nuestra máquina. Por otro lado podemos utilizar tanto la versión Web como la de escritorio de *PgAdmin* para interactuar con la base de datos mediante una interfaz gráfica, en mi caso haré uso de la aplicación web. Por último procedemos a instalar *QGIS*.

Para ello bastará con acceder a la página web oficial de la aplicación [12] y clicar en descargar, esto nos llevará a una página de descargas, allí seleccionaremos la versión **OSGeo4W Network Installer**. Una vez hecho esto procedemos con la instalación, bastará con darle a *express install* y cuando nos pida seleccionar paquetes, dejaremos seleccionados *QGIS y GDAL*[1]. Con esto ya podemos lanzar QGIS para comprobar que todo funciona correctamente y estaremos listos para trabajar.

3.6.2. Configuraciones iniciales

Comenzaremos creando un nuevo repositorio para el servidor y clonándolo en nuestra máquina local. Acto seguido creamos un entorno virtual de *Python* fuera del repositorio, el cual nos permitirá descargar e instalar módulos de forma localizada sin tener que instalarlos de forma global en nuestra máquina. En este caso comenzaremos con los módulos de *Django* y los controladores de base de datos que usaremos más adelante.

Clonamos los repositorio en nuestra maquina, inicialmente estarán vacíos:

```
$ git clone ${url del repositorio del backend}
$ git clone ${url del repositorio del frontend}
```

Creamos el entorno virtual del *backend*[2]:

```
$ python -m venv /ubicacion/del/entorno/back
```

Activamos el entorno virtual del *backend*:

```
$ source /ubicacion/del/entorno/back
```

Instalamos los módulos necesarios:

```
$ pip install django=={3.1.3} sqlparse==0.3.0
```

Requerimientos

Una vez hecho esto, lanzaremos el siguiente comando para generar un archivo de dependencias en el repositorio el cual se podrá utilizar para instalar los módulos necesarios desde un nuevo ordenador. Este proceso se realizará cada vez que instalemos un nuevo módulo tanto en el *backend* como en el *frontend*.

```
$ pip freeze > requirements.txt
```

[1] GDAL nos proporciona una variedad de herramientas para el tratado de imágenes [13].
[2] La creación y activación de un entorno virtual será la misma para el *frontend*

Este comando lista todos los módulos instalados en el entorno virtual de manera que luego se puede almacenar en un archivo llamado requirements.txt. De esta forma, se puede utilizar el siguiente comando para instalar todos los módulos con su versión correspondiente desde cualquier máquina con la que se pretenda trabajar.

```
$ pip install -r requirements.txt
```

Configuración inicial de django

Seguidamente ejecutaremos los siguientes comandos de *Django* para generar nuestra aplicación base:

```
$ django-admin startproject pluginServer
```

Con esto listo podremos observar la siguiente disposición de carpetas en nuestro repositorio del *backend*:

```
backend/
├── manage.py
├── requirements.txt
└── pluginServer/
    ├── __init__.py
    ├── settings.py
    ├── urls.py
    ├── asgi.py
    └── wsgi.py
```

QGIS y PB_TOOL

Para generar nuestra aplicación *frontend* necesitaremos primero tener funcional el entorno de *QGIS* e instalar el *pb_tool*, necesario para generar nuestra plantilla.

Comenzaremos descargando la última versión de *QGIS* de la página oficial https://www.qgis.org/es/site/forusers/download.html y lo instalaremos añadiendo todas las librerías adicionales disponibles una vez el instalador nos pida seleccionarlas. Esto nos dará acceso a todas las funcionalidades extra de *QGIS* que utilizaremos en el desarrollo del *plugin*. Seguidamente lanzaremos la aplicación y comprobaremos que todo funciona correctamente.

Procedemos a descargar el *pb_tool* de la siguiente forma[3]:

```
$ pip install pb_tool
```

con esto listo lanzaremos el *pb_tool* no sin antes crear un archivo de configuración en nuestra carpeta del repositorio. Este archivo se llamará pb_tool.cfg y lo rellenaremos con los siguientes datos:

[3] para esto, será necesario activar el entorno virtual del *frontend* como se explica previamente en la sección 3.6.2.

```
[plugin]
# Nombre del plugin
name: analyst_panel
# Dejar vacio para utilizar la ubicacion por defecto
plugin_path:

[files]
# Ubicacion de los archivos de python necesarios para el plugin.
python_files: __init.py__ analyst_panel.py analyst_panel_mainView.py
# La ventana principal de la aplicacion.
main_dialog: analyst_panel_mainview.py
# Otros archivos necesarios para las interfaces.
compiled_ui_files:
# Archivo con los requerimientos de las interfaces.
resource_files: resources.qrc
# Otros archivos extra a incluir en el plugin.
extras: metadata.txt icon.png requirements.txt
# Directorios extras a desplegar con el plugin.
extra_dirs: App

[help]
# Archivos de ayuda, autogenerados por el pb_tool
dir: help/build/html
target: help
```

Ahora lanzaremos el *pb_tool* con el siguiente comando:

```
$ pbt pb_tool.cfg
```

Una vez completada la generación del *plugin* obtendremos la siguiente disposición de carpetas en nuestro repositorio *frontend*:

```
Analyst Panel/
├── App/
├── i18n/
├── scripts/
├── test/
├── __init__.py
├── analyst_panel.py
├── icon.png
├── metadata.txt
├── pb_tool.cfg
├── resources.py
├── resources.qrc
└── requirements.txt
```

Con esto presente, podemos proceder a trabajar en la generación de las interfaces mediante PyQt5 y a programar las funcionalidades necesarias tanto para nuestro *backend* como para nuestro *frontend*.

3.7 Implementación

En esta sección seguiremos paso a paso el proceso de desarrollo de nuestra aplicación, mostrando los procesos y algoritmos principales y la forma en la que los sistemas interactúan entre sí.

3.7.1. Backend

Comenzamos el desarrollo donde lo dejamos en la sección de configuración de *Django* 3.6.2. El primer paso será añadir las configuraciones básicas de *Django* como la configuración de *CORS* y *IPs* permitidas.

Configuraciones

Los *CORS* o *Cross-Origin Resource Sharing* es un sistema de regulación de conexiones entre servidores. Cuando una página web proveniente de un servidor realiza una petición a otro servidor ajeno se está realizando una *cross-origin request*. Estas peticiones podrían ser maliciosas o perjudiciales si no son intención de los propietarios de dichos servidores.

De este modo para permitir a dos o más servidores interactuar entre sí será necesario establecer un método de comunicación entre ambos, haciendo uso del **intercambio de recursos de origen cruzado**. Este permiso se obtiene cuando uno de los servidores permite al otro acceder a sus recursos cuando en la cabecera de las peticiones introduce una lista de los servidores adecuados para obtener estos recursos. Cuando estas comunicaciones se realizan correctamente, el usuario queda ajeno a ellas y los servidores son capaces de funcionar a la perfección en comunión.

Por este motivo, dado que la empresa es poseedora de diversos servidores proveedores, será necesario permitir en las peticiones el acceso a todos los servicios de OrbitalEOS para un buen funcionamiento en común.

Gracias a los *CORS* podemos evitar problemas futuros de ciberseguridad permitiendo únicamente a nuestro propio dominio realizar peticiones a nuestro servidor, después reforzamos los *CORS* restringiendo el acceso únicamente a nuestro *plugin* y página web.

Modelos

El siguiente paso consiste en utilizar los modelos especificados previamente en el diseño de la base de datos 3.4 para generar los modelos de *Django*. Para esto, nos dirigiremos al archivo "models.pyz programaremos los modelos y transformadores que nos permitirán convertir la información de la base de datos directamente a nuestro programa.

Un modelo de *Django* permite al *framework* auto-generar las tablas de la base de datos y asignarle los tipos de datos necesarios utilizando un sistema de clases sencillo de *Python* siguiendo el siguiente formato.

models.py

```python
from django.db import models
from django.contrib.auth.models import AbstractBaseUser


class CustomUser(AbstractBaseUser):
    email = models.EmailField(unique=True)
    zip_code = models.CharField(max_length=6)
    name = models.CharField(max_length=30)
    is_staff = models.BooleanField(default=False)
    REQUIRED_FIELDS = ['zip_code']
    USERNAME_FIELD = 'email'

    def get_short_name(self):
        return self.email

    def __str__(self):
        return self.email
```

Figura 3.5: ejemplo de un modelo de *Django*

Como podemos observar en este ejemplo, cada modelo o entidad será una clase de *Python* con variables internas compuestas por diferentes objetos importadas del módulo **models** de **django.db**. Este módulo nos proporciona los distintos tipos de datos disponibles para establecer en nuestra base de datos. Por otro lado también podemos añadir funciones al modelo e incluso generar serializaciones para automatizar la conversión de datos a un objeto *JSON* o similares. Gracias a los modelos podemos generar o obtener una entidad de la base de datos fácilmente.

API

¿Que es una API REST?

Una *API* o Interfaz de Programación de Aplicaciones es uno de los servicios más importantes de transmisión de información a día de hoy. Estas están diseñadas con el objetivo de facilitar el intercambio de información entre sistemas de forma correcta, sencilla y [25].

En nuestro caso, una *API REST* es una interfaz de conexión entre servidores y clientes la cual permite intercambiar datos haciendo uso de representaciones de los mismos, los más comunes suelen ser *JSON* o *XML*.

Diseño de la API

Para un correcto desarrollo de una *API REST* es necesario tener en cuenta diversos factores. Primeramente es esencial un correcto uso de los diferentes métodos *HTTP* y las

peticiones. Además será necesario mantener una buena organización de los contenidos, principalmente separando cada objeto en su propia dirección única, mediante la cual podremos acceder a sus diferentes propiedades haciendo uso de las peticiones *HTTP*.

Las diferentes peticiones *HTTP* conforman el acrónimo *CRUD*, este hace referencia a *Create, Read, Update, Delete*. Esto nos presenta cuatro tipos diferentes de petición que nuestros objetos tendrán que soportar.

POST: para crear un objeto.
GET: para obtener un objeto.
PUT: para actualizar los objetos con nueva información.
DEL: para eliminar un objeto.

Nuestro objetivo será poder intercambiar información con nuestro servidor de forma adecuada. Para esto generaremos las siguientes peticiones para cada uno de los recursos que queramos intercambiar con la aplicación *frontend* y esta información será transmitida a través de archivos *JSON*.

Implementación de la API

Comenzamos la implementación de nuestra *API* configurando el sistema de inicio de sesión de los usuarios, esto es gracias al sistema que *Django* trae consigo en el módulo **django.contrib.auth** para manejar peticiones con autenticación previa. Una vez establecido el uso de este módulo, cabe destacar que será necesario realizar un par de modificaciones en la configuración base del mismo. Nuestro objetivo es obtener una base de datos individualizada, es decir, que un usuario únicamente tenga acceso a los objetos que le pertenecen. Esta idea se ve bastante clara cuando tenemos en cuenta el diseño de la base de datos 3.4, ya que cada usuario tiene asociados una serie de eventos y regiones.

Como excepción tendremos a los analistas, usuarios de nuestra aplicación de *frontend* los cuales tendrán acceso a la base de datos en su totalidad, siendo estos los que tendrán que hacer las modificaciones pertinentes para asignar eventos y regiones a clientes.

Ahora necesitamos proteger mediante usuario y contraseña todas las funcionalidades de la *API*, esto es tan sencillo como añadir el marcador "@Login"justo en la linea previa a las funciones que vamos a crear para manejar las peticiones a nuestra *API*. Internamente, *Django* comprobará con nuestra base de datos si los credenciales se corresponden con los provistos por la aplicación *frontend* que está realizando la petición.

Para generar estos credenciales, tendremos que ingresar en nuestra aplicación y acceder a `www.nuestrapp.com/api/admin` y autenticarnos con unos credenciales que tengan permisos de administrador. Si es la primera vez accediendo a este apartado de la aplicación el usuario por defecto lo tendremos que crear vía consola de comandos utilizando la siguiente instrucción:

```
$ python manage.py createsuperuser
```

Seguidamente habrá que introducir un nombre para el administrador además de una contraseña y obtendremos nuestra cuenta de administrador para la aplicación.

Una vez terminados los preparativos, podemos comenzar a programar nuestra *API*. Primeramente necesitaremos generar **CRUD** para las tablas de nuestra base de datos. Un

CRUD nos permite crear, leer, actualizar y borrar objetos de nuestra base de datos. Esta es la interacción esencial de una *API* y con la cual podremos proveer a nuestra aplicación con los datos necesarios para funcionar adecuadamente.

En este caso seguiremos la siguiente convención para generar las *url* con las que nuestro *frontend* podrá interactuar. Todas las funcionalidades de la *API* se situarán dentro de `www.nuestrapp.com/api/` y separado por cada objeto o recurso. Esto nos deja con las siguientes url para funcionalidades *CRUD*:

```
/api/event
/api/client
/api/region
/api/layer
/api/source
```

Gracias a *Django*, podemos utilizar la herencia de clases para auto generar una vista web que nos permite interactuar con nuestra *API* de forma gráfica y sencilla, para esto únicamente tenemos que heredar de **APIView** en cada clase de la siguiente forma:

```
class eventApi(APIView):
  def get(request):
      # GET request
  def post(request):
      # POST request
  def put(request):
      # PUT request
  def del(request):
      # DEL request
```

Al añadir esta herencia a la clase y programando las diferentes peticiones, django nos proporcionará una vista web si accedemos a la *url* del recurso en cuestión desde un navegador. La vista provista nos permitirá realizar todas las peticiones incluidas en la clase que hemos creado previamente como se muestra en la siguiente imagen 3.6.

De esta forma podremos obtener una clase con todas las funcionalidades *CRUD* para cada uno de nuestros modelos de objetos. Esto junto con el sistema de autenticación implementado previamente, nos deja casi listos para proceder con el desarrollo de la aplicación *frontend*, donde este será capaz de realizar llamadas a nuestra recién creada *API* para intercambiar información.

Finalmente tendremos que desarrollar una serie de funcionalidades extra que nos harán la vida un poco más fácil a la hora de programar el resto de la aplicación. Primeramente añadiremos un filtrado a aplicar en las peticiones a nuestros objetos. Este filtro será utilizable del siguiente modo:

`https://nuestrapp.com/api/<objeto>?<campo,operador,valor>`

Y estos serán todos los operadores disponibles para utilizar en el filtrado:

```
  "eq": es igual a,
  "gt": mayor que,
```

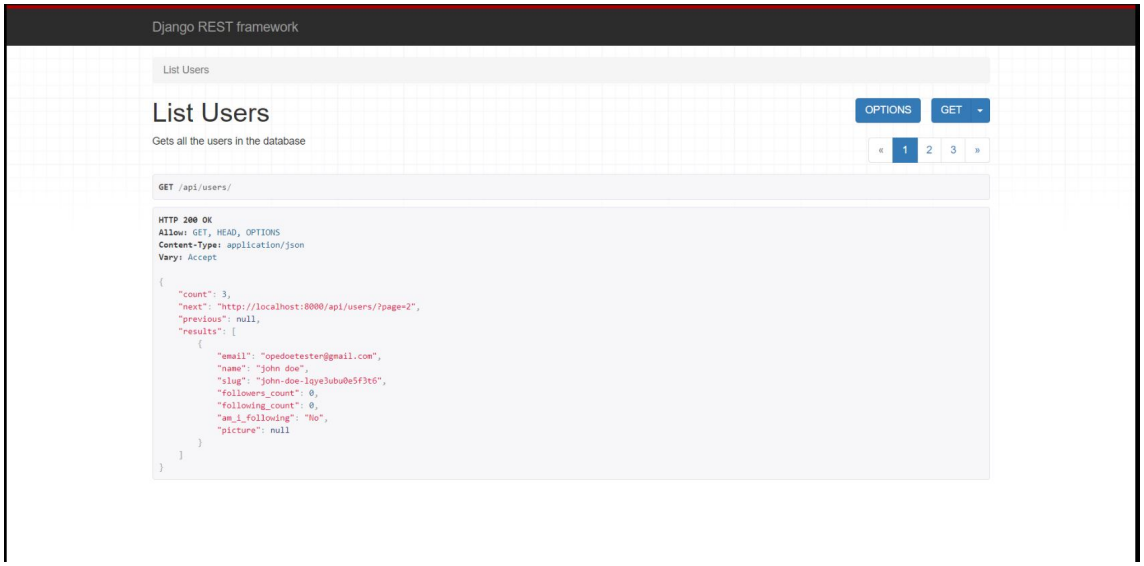

Figura 3.6: Ejemplo de una APIview de Django

```
"gte": mayor o igual a,
"lt": menor que,
"lte": menor o igual a,
"sw": empieza con (para cadenas de texto),
"isw": empieza con y ignora mayusculas/minusculas,
"ew": acaba con (para cadenas de texto),
"iew": acaba con y ignora mayusculas/minusculas,
```

3.7.2. Frontend

Continuamos el desarrollo del *frontend* a partir de la plantilla auto-generada en el apartado 3.6.2 donde se describe *QGIS y pb_tool*. En este apartado encontraremos los diferentes prototipos entregados a lo largo del desarrollo y una imagen del diseño final de todas las herramientas generadas.

Herramientas

Comenzaremos replicando los diseños de las diferentes herramientas utilizando la aplicación de *PyQt5* para modelar nuestras interfaces. A continuación desarrollaremos las funcionalidades de cada herramienta individualmente.

Esta sección comienza dando una breve explicación de la herramienta utilizada para modelar las interfaces gráficas, para seguidamente explicar cada una de ellas en detalle.

QtDesigner

Este diseñador de interfaces gráficas, aportado por el grupo creador del módulo *PyQt5*, nos permite generar de forma automática el código necesario para enlazar nuestras funcionalidades a una interfaz.

Podemos utilizar dos métodos de generación de código, el primero nos permite generar un único archivo conteniendo la información de la interfaz en forma de código de *Python* y permitiéndonos así añadirle funcionalidad interactuando de forma directa con los objetos creados mientras que el segundo método consigue extraer la funcionalidad de una interfaz de sus componentes gráficos, generando así un archivo *UI* que tomará una estructura similar a un archivo *XML* ejemplificado en la figura 3.7.

```
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
 <class>dialogTesterDialogBase</class>
 <widget class="QDialog" name="dialogTesterDialogBase">
  <property name="geometry">···
  </property>
  <property name="windowTitle">···
  </property>
  <property name="styleSheet">
   <string notr="true">
  </string>
  </property>
  <widget class="QWidget" name="gridLayoutWidget">
   <property name="geometry">
    <rect>
     <x>40</x>
     <y>50</y>
     <width>551</width>
     <height>171</height>
    </rect>
   </property>
   <layout class="QGridLayout" name="gridLayout">
    <item row="0" column="1">
     <widget class="QLineEdit" name="username_text">
      <property name="sizePolicy">
       <sizepolicy hsizetype="Expanding" vsizetype="Expanding">
        <horstretch>0</horstretch>
        <verstretch>0</verstretch>
       </sizepolicy>
      </property>
      <property name="maximumSize">
       <size>
        <width>400</width>
        <height>100</height>
       </size>
      </property>
```

Figura 3.7: Ejemplo de un archivo .ui generado por QtDesigner

Y un archivo de *Python* en el cual obtendremos la interfaz de la forma observada en la figura 3.8 pudiendo así acceder a las propiedades de la interfaz como elementos propios de la clase de *Python*. En nuestro caso y por una mejor organización del código y mayor limpieza visual, utilizaremos el segundo método.

```
# This loads your .ui file so that PyQt can
# populate your plugin with the elements from Qt Designer
FORM_CLASS, _ = uic.loadUiType(os.path.join(
    os.path.dirname(__file__), 'login_dialog.ui'))

class loginDialog(QtWidgets.QDialog, FORM_CLASS):
    def __init__(self, parent=None):
        """Constructor."""
        super(loginDialog, self).__init__(parent)
        # Setup Ui
        self.setupUi(self)
```

Figura 3.8: Ejemplo de código utilizado para importar archivos .ui

Para comenzar a crear una interfaz en QtDesigner clickaremos en *File->New* y obtendremos un menú el cual nos ofrecerá una serie de plantillas iniciales, dado que nuestra aplicación se compone de una ventana principal poblada por diversas herramientas en forma de cuadros de dialogo, empezaremos con la plantilla de Dialogo sin botones: *Dialog without Buttons* y con un tamaño de ventana por defecto para facilitar la explicación del *QtDesigner*.

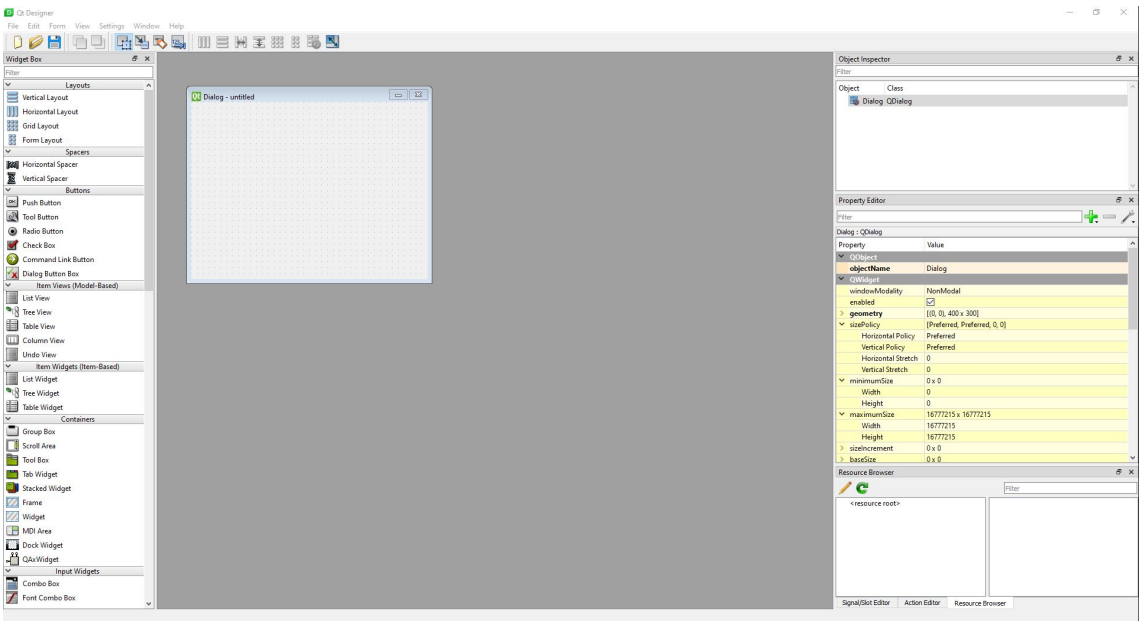

Figura 3.9: Ejemplo de plantilla inicial de QtDesigner

Una vez inicializada la plantilla 3.9 podremos comenzar a utilizar todo lo que el diseñador nos ofrece. En el lado izquierdo encontramos la caja de *widgets* a nuestra disposición, estos se dividen en diversas categorías según su utilidad. Los *widgets* utilizados a lo largo del proyecto se encuentran explicados en el Apéndice C.2.

Por otro lado en el centro aparece nuestra recién generada plantilla, aquí podremos interactuar con los diferentes *widgets* para modificar su tamaño y posición, además de alterar algunas características de los mismos.

Por ultimo en el lado derecho encontramos tres subapartados. El inspector de objetos, el cual nos presenta una jerarquía de los elementos insertados en nuestra interfaz, el editor de propiedades, utilizado para editar al completo cualquier *widget* y un buscador de re-

cursos al cual no le daremos uso durante el desarrollo ya que la empresa proporcionará todo lo necesario.

Antes de comenzar con el desarrollo de las interfaces, cabe destacar que utilizaremos un posicionamiento en cuadrícula utilizando los *widgets* situados bajo la sección ***Layouts***. No se insistirá en la explicación del posicionamiento a lo largo de la memoria puesto que consiste en modificar valores relativos de forma muy superficial pero repetitiva.

Login

Como pantalla inicial al abrir por primera vez la herramienta, los analistas serán presentados con esta sencilla pantalla de inicio de sesión 3.10.

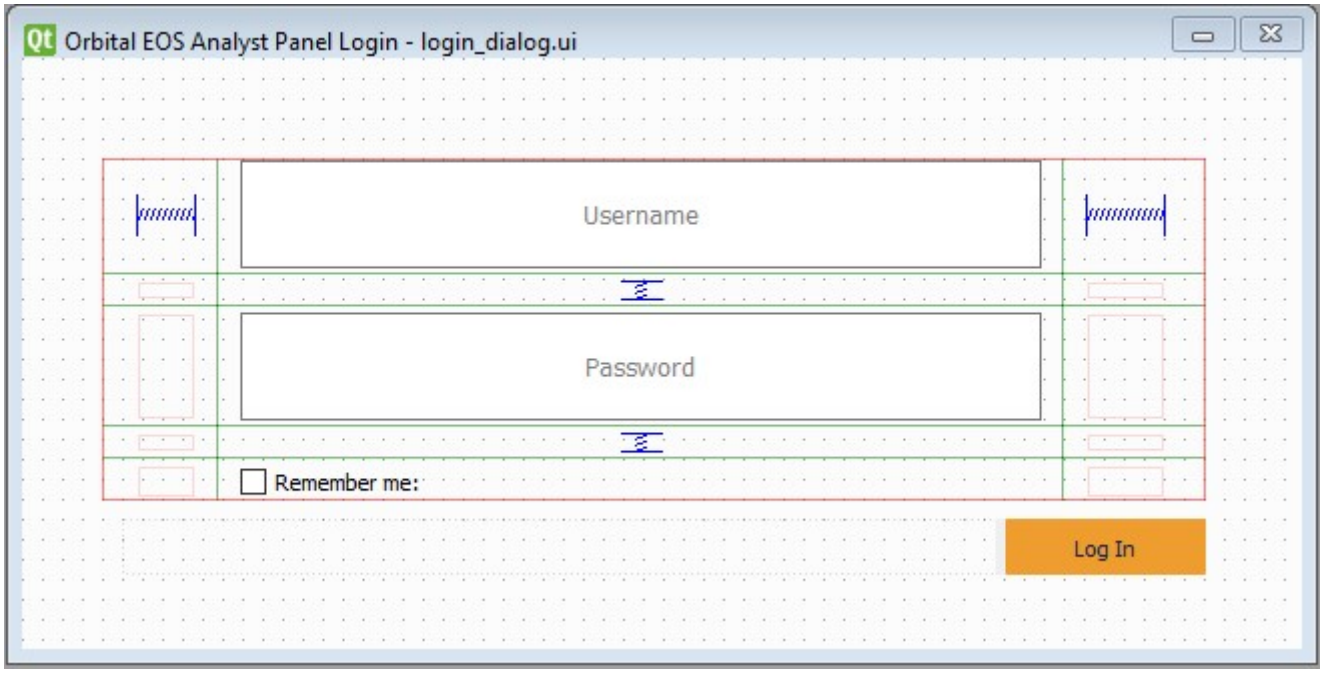

Figura 3.10: Diseño final del Login en PyQt5

Esta interfaz está diseñada haciendo uso de cuatro *widgets*. Dos elementos **QLineEdit** en los cuales se introducirán el usuario y la contraseña del analista, una **QCheckBox** que establece si mantener la sesión iniciada al cerrar la aplicación, y un **QPushButton** para iniciar la sesión una vez rellenados los credenciales.

Dashboard - Panel de control

Esta interfaz 3.11 3.12 será nuestro panel principal, en él, los analistas podrán seleccionar cualquiera de los **QPushButtons** situados en el panel lateral izquierdo para abrir una nueva pestaña en el panel central. En dicho panel podremos trabajar con diversas pestañas abiertas al mismo tiempo ya que ciertas herramientas se bloquearán mientras estén en funcionamiento. Esta es una de los requerimientos más importantes pues agiliza el trabajo de los analistas enormemente.

El panel izquierdo consiste de una **QScrollArea** compuesta por una serie de **QPushButtons** añadidos via codigo. Está es la unica interfaz que presenta *widgets* añadidos via codigo, esto es así ya que facilita la generación de nuevas herramientas sin tener que editar la interfaz cada vez que se genere una herramienta nueva, facilitando así el desarrollo y el crecimiento de la aplicación. Lo unico necesario será importar la herramienta en la ventana principal y enlazar la herramienta a un boton iterando por todas las herramientas actuales.

Por otro lado, el panel central es una **QMidiArea** con las siguientes propiedades modificadas:

1. **ViewMode**: TabbedView
2. **tabsClosable**: True
3. **tabsMovable**: True

Por último, en la parte superior encontraremos un **QMenuBar** con 3 submenús. *File* contiene la opción *Open TMP* la cual permite al analista acceder a la carpeta de archivos temporales en la que se situan los archivos generados durante los diversos procesos de la aplicación, esta tambien es la ubicación por defecto de guardado de los archivos generados. Seguidamente tenemos *Tools* donde se repiten las herramientas situadas en el panel lateral izquierdo, esta redundancia es un requerimiento de la empresa para facilitar el uso de la aplicación en pantallas grandes y usuarios de teclado. Por último *Options* contiene *AWS Credentials*, donde el analísta podrá editar los credenciales para los servicios de *AWS*, *Preferences* lanzará una nueva ventana donde se pueden editar los valores por defecto de las diferentes herramientas 3.7.2 y una opción de *Logout* para cerrar la sesión del usuario.

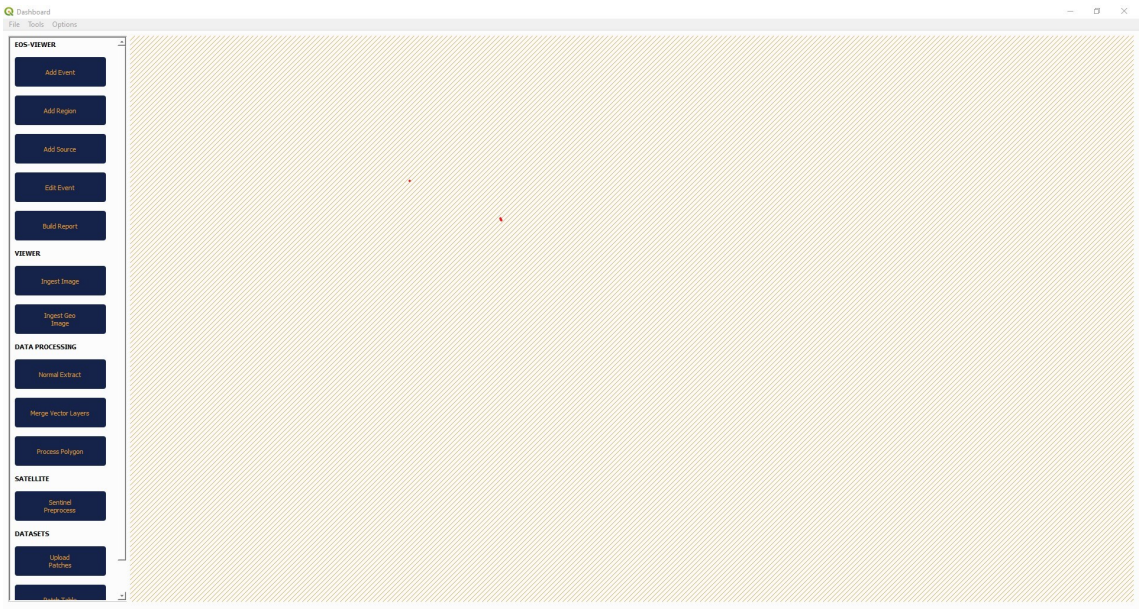

Figura 3.11: Diseño final del Dashboard sin pestañas

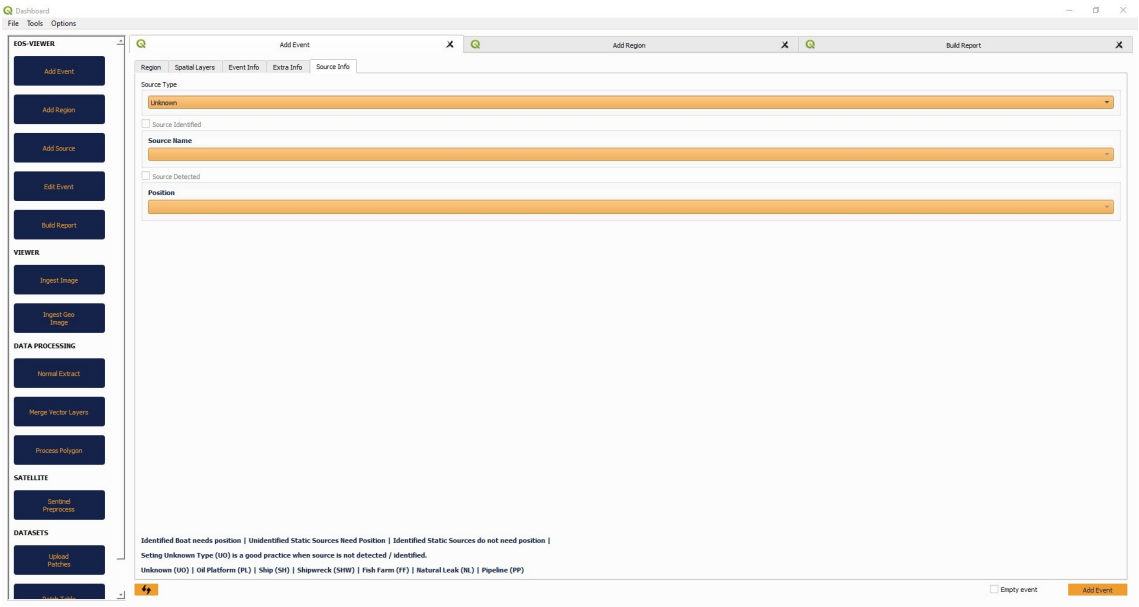

Figura 3.12: Diseño final del Dashboard con pestañas

Preferences

La siguiente interfaz 3.13, mencionada previamente en el panel de control 3.7.2, está compuesta por cinco **QGroupBoxes** utilizadas para dividir el contenido para las herramientas correspondientes. Por otro lado se utilizan **QLabels** para nombrar los *widgets* internos compuestos por **QSpinBoxes**, **QDoubleSpinBoxes** y **QCheckBoxes**.

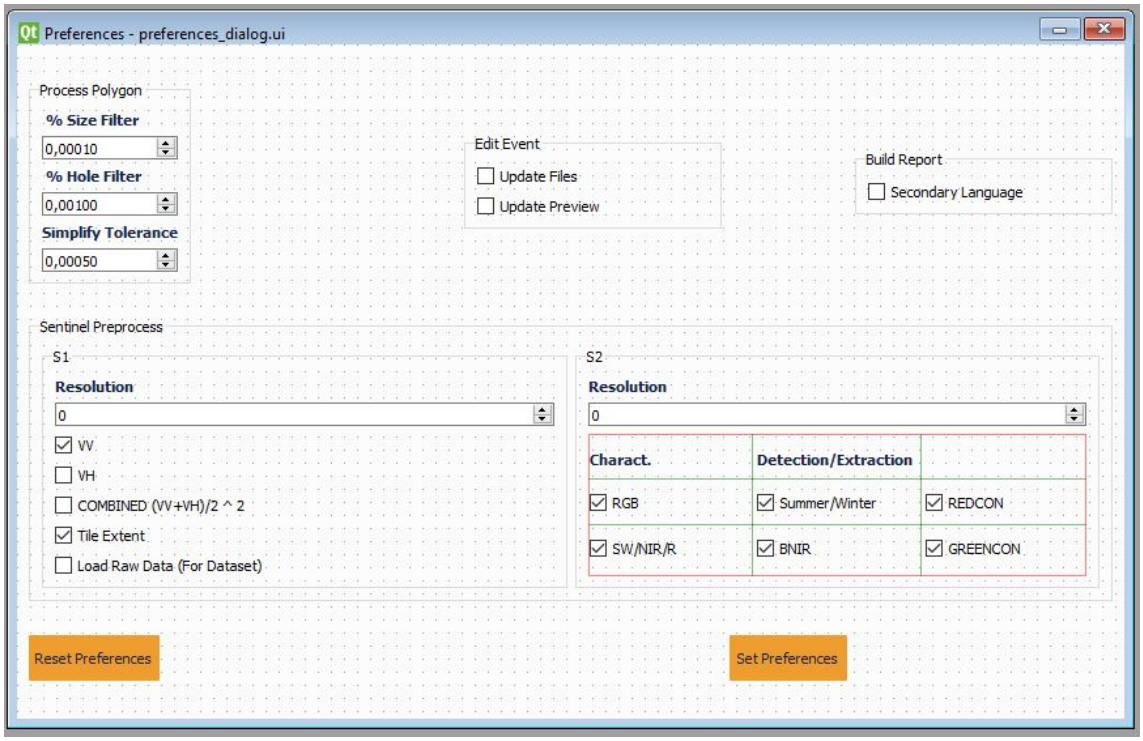

Figura 3.13: Diseño final de la ventana de preferencias

Las opciones representadas en esta ventana serán descritas en sus herramientas correspondientes.

Add Event

Comenzamos con la primera herramienta de adición de datos. Este conjunto de pestañas serán utilizadas para generar datos sobre eventos de forma manual. Encontramos diversos grupos de datos repartidos a lo largo de las pestañas de esta ventana.

La primera ventana general 3.14 está compuesta por un **QTabWidget**, que como su nombre indica nos permite dividir la ventana en diversas pestañas, además encontramos en la barra inferior dos **QPushButtons**, el izquierdo es un pequeño parche añadido para facilitar la recarga de la herramienta (esto se debe a problemas con el proveedor del servidor) y el derecho subirá el evento generado una vez terminado el rellenado de datos al servidor. Este segundo botón solo será clickable cuando todos los elementos y campos obligatorios se encuentren rellenados al completo.

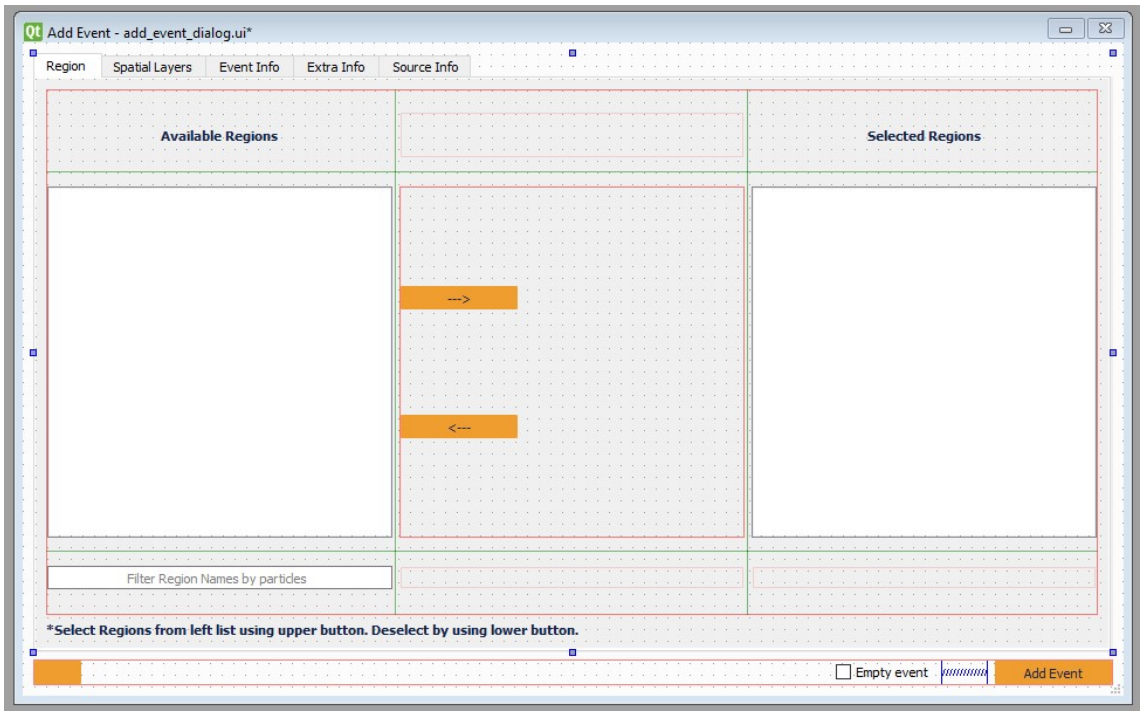

Figura 3.14: Primera pestaña del diseño final de la herramienta Add Event

En la primera pestaña 3.14 se pueden observar dos elementos **QListWidget**, el izquierdo se rellenará automáticamente una vez lanzada la aplicación con las regiones disponibles en el servidor. En el derecho aparecerán las regiones seleccionadas haciendo uso de los dos **QPushButtons** situados en el centro de la pestaña [4]. Por último en la parte inferior izquierda los analistas podrán filtrar las regiones por nombre utilizando un **QLineEdit**.

En la segunda pestaña 3.15 encontraremos una serie de campos a rellenar compuestos por tres **QgsMapLayerCombobox** donde el analista seleccionará las diferentes capas que componen el evento. Solo la capa de *Sheen* es obligatoria, las capas de *Thick* y *True Color* son opcionales y dependen de la imagen obtenida y el tipo de evento que se desee

[4] En el diseñador de *PyQt* estos botones aparecen descentrados pero a la hora de ejecutar la aplicación se sitúan correctamente, aparentemente es un error del editor

construir. Estos dos últimos componentes habrán de activarse mediante una **QCheckBox** antes de poder ser utilizadas. Además encontramos una capa más, la capa de *Preview* seleccionada mediante una **QgisCheckableComboBox** y que nos proporciona una imagen de previsualización del evento, utilizada principalmente en los reportes auto-generados 3.7.2 y el visualizador web. Por último tenemos la primera **QCheckBox**, que afecta al tipo de datos que son enviados al servidor definiendo si se ha incluido o no un análisis del grosor de la mancha, además de una pequeña descripción de uso mediante **QLabels**.

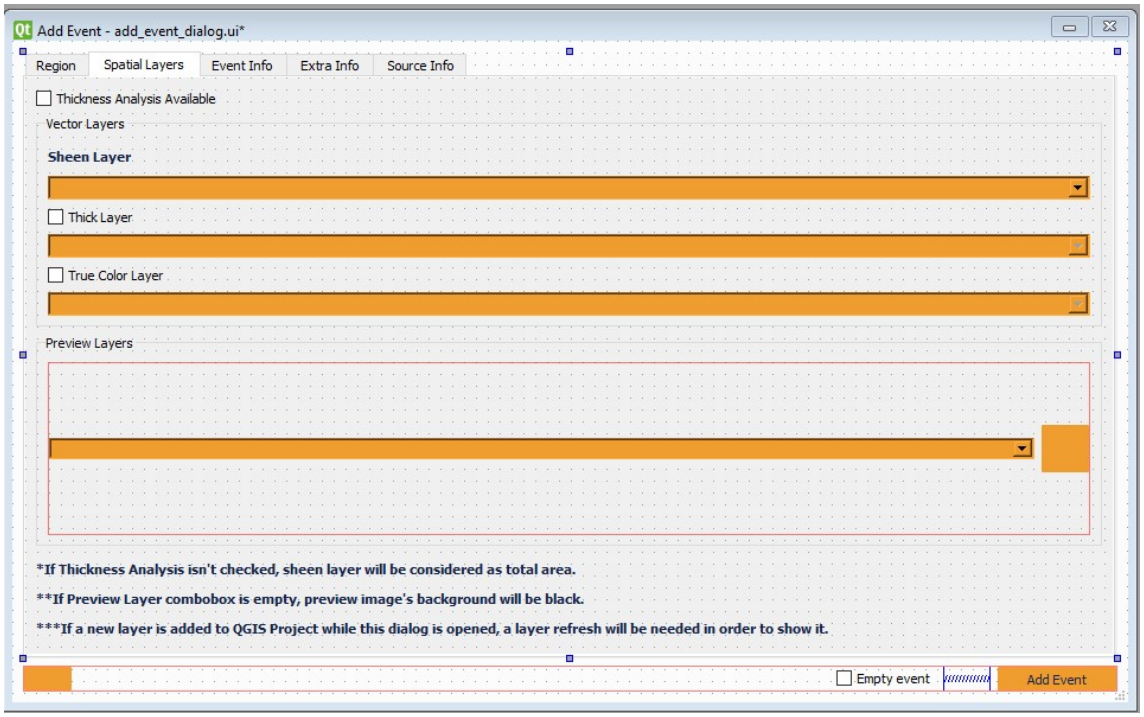

Figura 3.15: Segunda pestaña del diseño final de la herramienta Add Event

La tercera pestaña3.16 contiene un selector de fecha **QCalendarWidget**, para indicar la fecha en la que sucedió el evento, tres **QSpinBoxes** para establecer el contador[5] y la hora de detección. Seguidamente encontramos un **QLineEdit** para definir el código de la región(global) y por último tres **QComboBoxes** para establecer los valores de la clase del evento, el satélite proveedor y el nombre del proveedor de la capa (WMS).

En la cuarta pestaña 3.17 encontramos cinco **QLineEdits** donde el usuario podrá introducir datos obtenidos al realizar el análisis, además del enlace a la descarga de la imagen original y algunos comentarios extra que el mismo quiera introducir. Toda esta información será necesaria para generar el Reporte.

Por último en la quinta pestaña 3.18 el analista tendrá que añadir la información referente al origen de la mancha. Primeramente eligiendo el tipo de origen entre los diferentes provistos por el servidor en una **QComboBox**, una vez seleccionado el tipo se habilitarán dos **QCheckBox** que nos permitirán seleccionar un nombre para el origen mediante una **QComboBox** y la posición con un **QgsMapLayerComboBox**.

Cabe destacar la **QCheckBox** situada en la esquina inferior derecha, ésta servirá para generar eventos vacios. Estos son los eventos que indican que no se ha encontrado ningún tipo de mancha en el análisis.

[5]Numero asignado al evento según la cantidad de eventos detectados en un mismo día

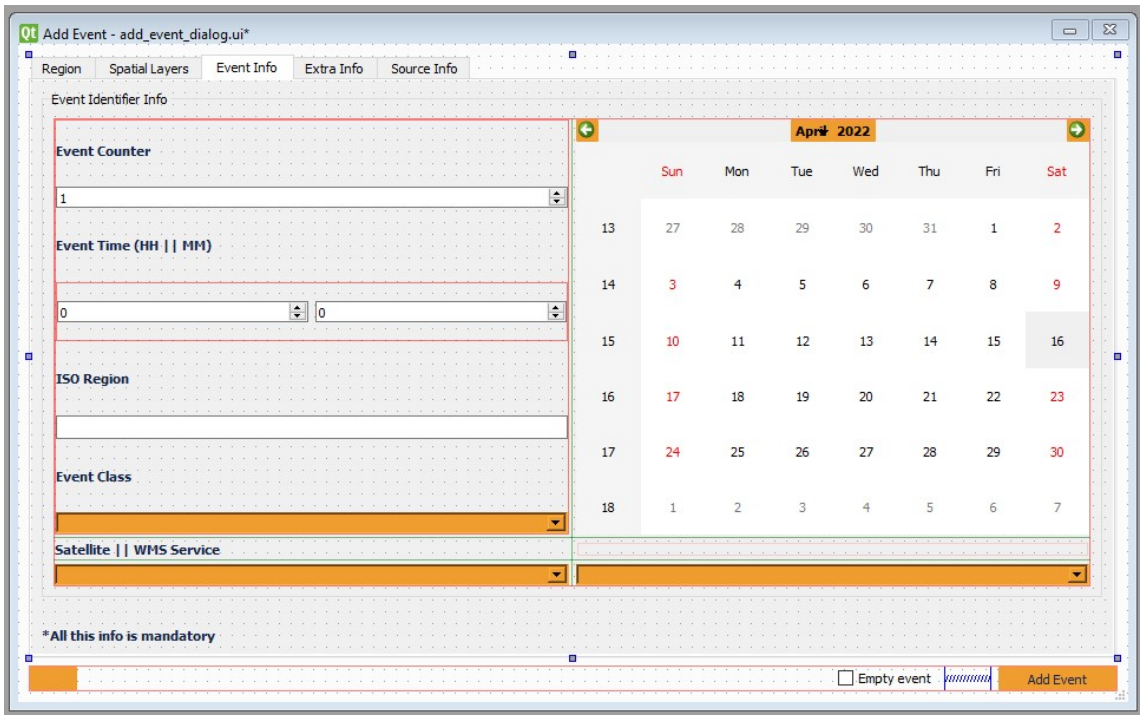

Figura 3.16: Tercera pestaña del diseño final de la herramienta Add Event

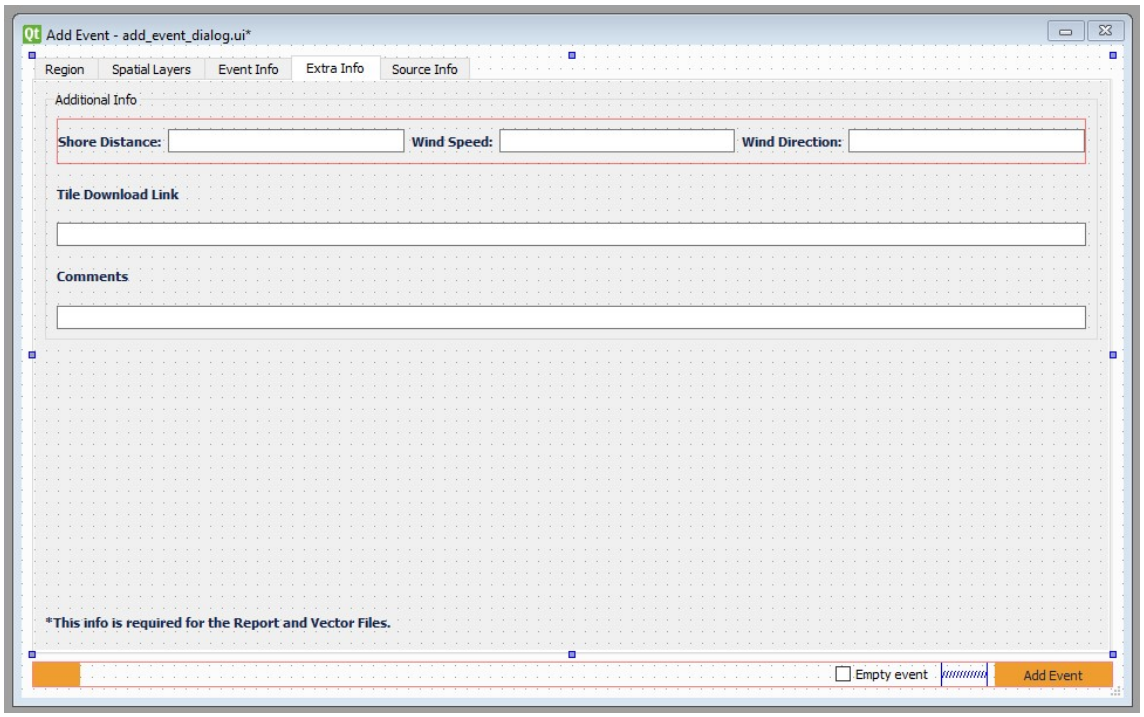

Figura 3.17: Cuarta pestaña del diseño final de la herramienta Add Event

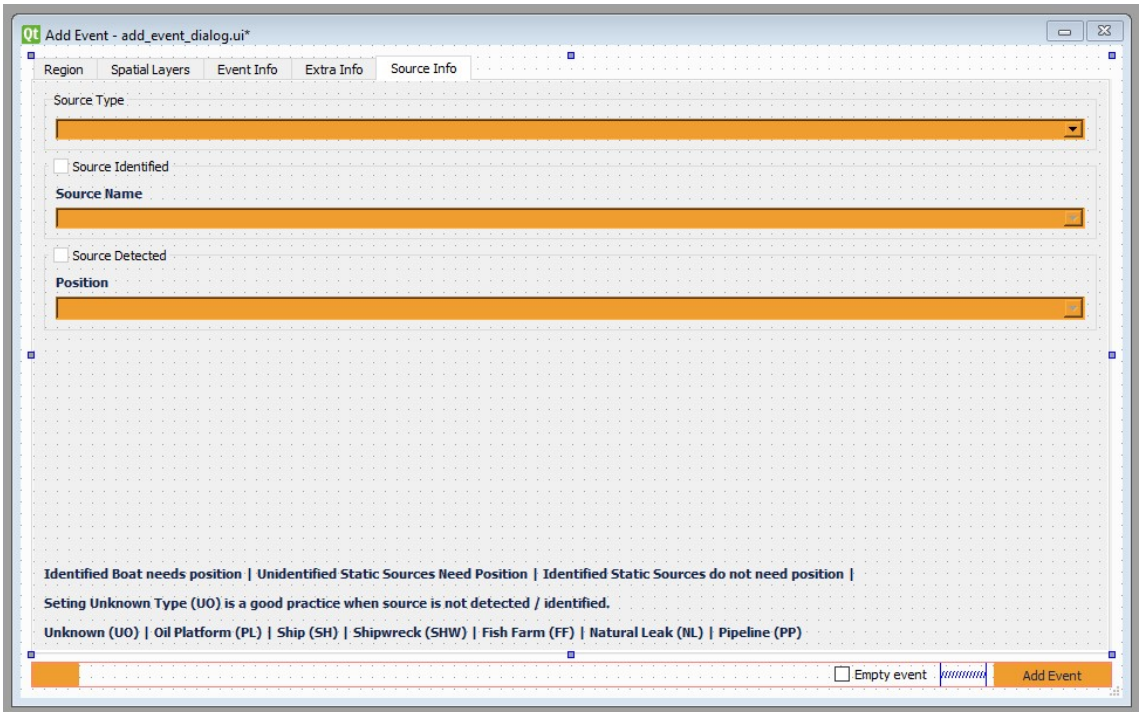

Figura 3.18: Quinta pestaña del diseño final de la herramienta Add Event

Add Region

La siguiente herramienta 3.19 permitirá añadir una nueva región al grupo de las mismas disponibles para la creación de eventos. En la misma encontramos una dos**QComboBox**, una para seleccionar una región y la otra para seleccionar el identificador de un cliente. Si seleccionamos una región, esta será modificada con los nuevos datos introducidos, en cambio si no seleccionamos ninguna región, una nueva será generada desde cero.

Por otro lado tenemos un **QLineEdit** para introducir el nombre de la nueva región y una **QgsMapLayerComboBox** para seleccionar la capa de *QGIS* que representará a la misma. Por último un **QPushButton para enviar nuestra recién generada región.**

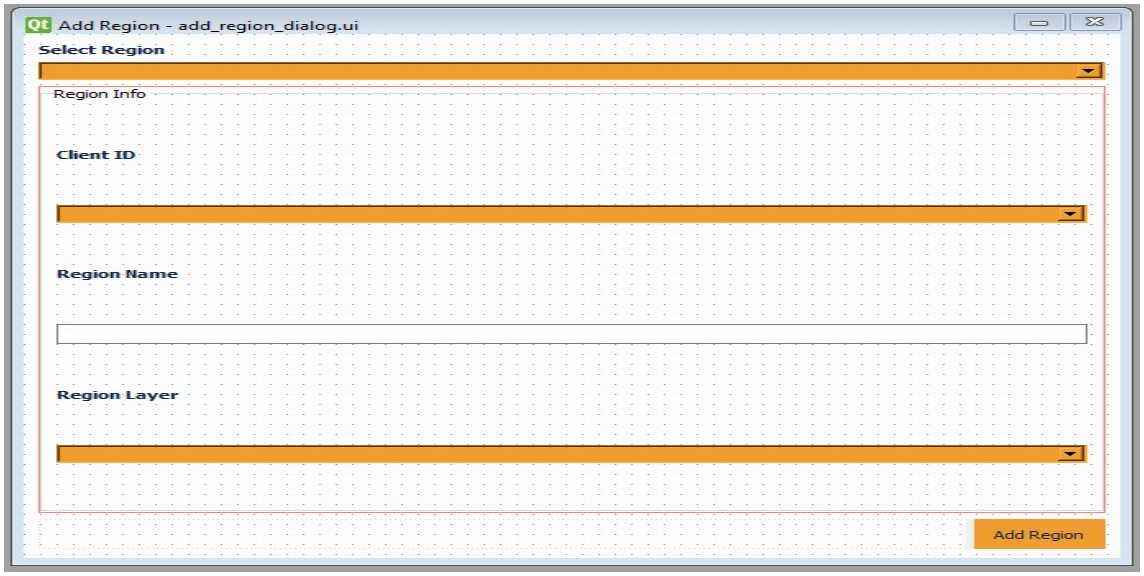

Figura 3.19: Diseño final de la herramienta Add Region

Add Source

La siguiente ventana 3.20 está diseñada con el fin de permitir a los analistas añadir un nuevo origen de contaminación marítima. Utilizarán esta herramienta compuesta por dos **QComboBox** para establecer el tipo[6] y seleccionar el origen.

Además tendrán que rellenarse una serie de datos que conforman la información relevante de la fuente encontrada. Entre ellos encontramos el nombre del origen, el propietario de la misma y un enlace *url* a una imagen de previsualización, todo esto como entradas del usuario mediante **QLineEdit**. A su lado será necesario introducir la posición de la fuente de origen mediante una **QgisMapLayerComboBox** en la cual se podrá seleccionar una capa generada que delimite las coordenadas en las que se encuentra encapsulada la fuente.

Por último será necesario especificar una serie de valores más detallados que dependerán del tipo de origen seleccionado. Si se trata de un barco o similar(SH/SHW) habrá que introducir datos como el tipo de barco, el estado de la bandera y demás, y si se trata de una planta o similar(PL) habrá que introducir el operador, la identificación del bloque y del cliente, todo esto haciendo uso de varios elementos **QLineEdit**.

[6]Tipos preestablecidos por la empresa.

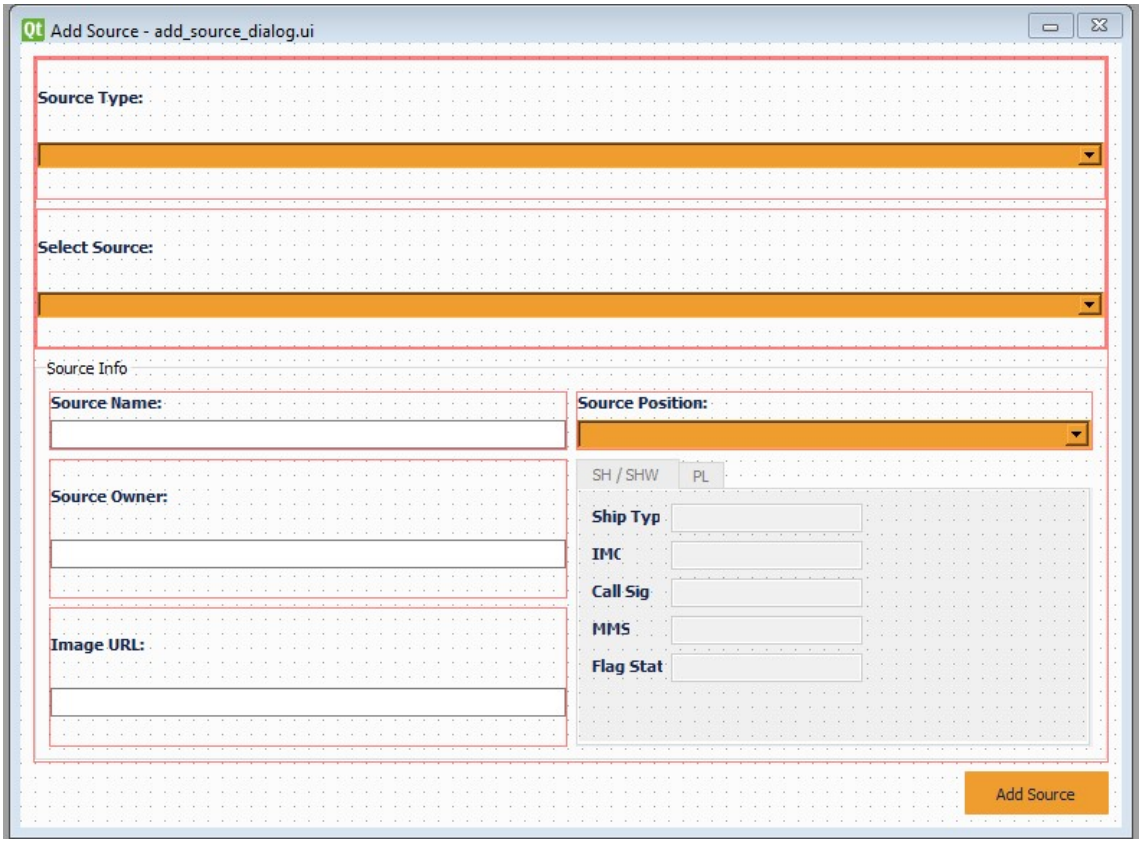

Figura 3.20: Diseño final de la herramienta Add Source

Edit Event

Encontramos que la próxima herramienta 3.21 es muy similar a Add Event, explicada previamente en la subsección 3.7.2. En concreto, únicamente difiere en la primera pestaña, esta herramienta posee una pestaña extra al comienzo que permite seleccionar un evento ya existente en la aplicación para ser editado.

Esta primera pestaña está compuesta de dos **QLineEdit** y dos **QComboBox** utilizados para filtrar y seleccionar una región existente y un evento en este orden. Seguidamente se encuentra disponible una opción **QGroupBox** la cual una vez habilitada nos permite utilizar información extra para seleccionar el evento mediante dos **QComboBox**.

Finalmente encontramos dos **QCheckBox** utilizadas para indicar si se quiere actualizar los archivos[7] y la imagen de previsualización del evento, además del **QPushButton** utilizado para finalizar la edición del proceso.

[7] Estos archivos se encuentran en un *.rar* y conforman los elementos obtenidos de la descarga del proveedor

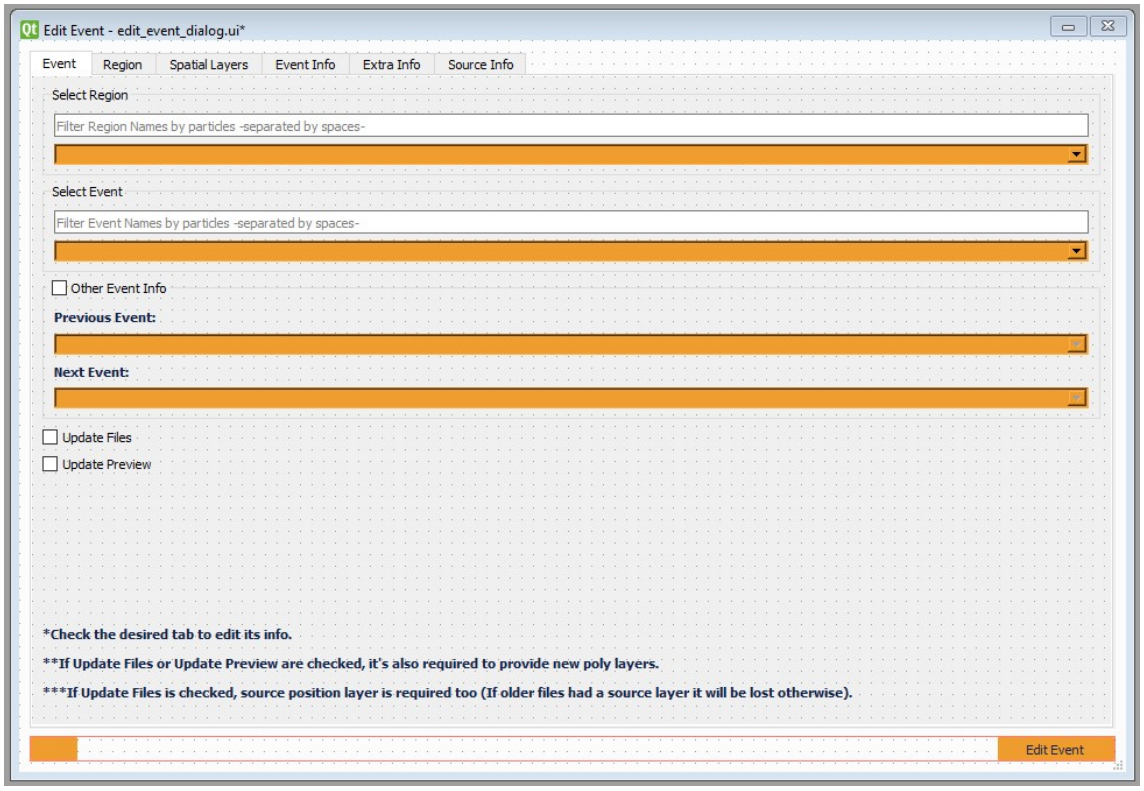

Figura 3.21: Primera pestaña del diseño final de la herramienta Edit Event

Build Report

En la siguiente herramienta 3.22 encontraremos una gran parte del esfuerzo de desarrollo junto a Sentinnel Preprocess 3.7.2, esto es debido a que para poder generar documentos *PDF* se ha creado una herramienta personalizada desde cero utilizando un par de módulos que permiten, de la manera más básica, interactuar con un *PDF*. Estos módulos únicamente permiten dibujar figuras geométricas básicas y caracteres utilizando posicionamiento en el eje x e y, esto quiere decir que editar un reporte detallado requeriría una cantidad insana de trabajo, debido a que cada carácter e imagen tendría que tener su posición especificada individualmente calculando su posición en relación al resto una vez tras otra.

La solución a este problema se obtiene al generar una clase que nos permite tratar el documento como si de una cuadrícula se tratase. Este editor de *PDF* se alimenta de diferentes objetos individuales que se agrupan para generar el documento completo. Tenemos como componentes:

```
Cell.py
Image.py
Table.py
Page.py
```

Siendo **Page** un componente genérico de Página[8] del cual heredarán todas las demás paginas. De este modo encontramos una serie de páginas diseñadas como componentes utilizables dentro del documento que son las siguientes:

```
AnalysisPage.py
BigOverviewPage.py
ContentPage.py
CoverPage.py
EmptyOverviewPage.py
EventPage.py
MediumOverviewPage.py
SingleOverviewPage.py
SourcePage.py
```

Una vez conocidas estas páginas podemos reducir la creación del reporte a una sencilla llamada con ciertos parámetros los cuales obtendremos de la herramienta en cuestión 3.22.

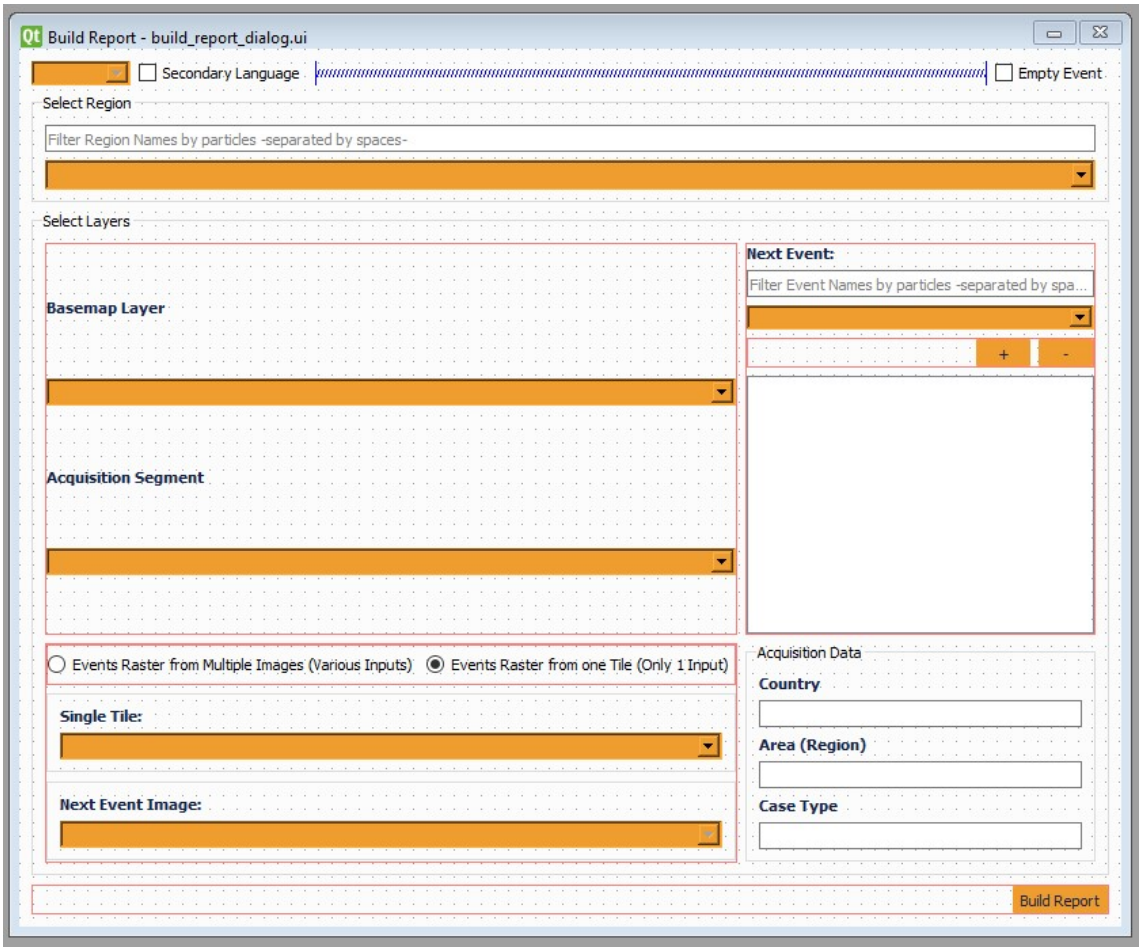

Figura 3.22: Diseño final de la herramienta Build Report

[8]Este componente genérico contiene datos que aparecerán en todas las páginas, como pueden ser el logo de la empresa y el título del reporte

Comenzaremos a utilizar la herramienta seleccionando el idioma[9], mediante una **QComboBox**, con el que será generado el reporte. Seguidamente indicaremos si se trata de un evento vacío con una **QCheckBox** y seleccionaremos la región a la que pertenece el cliente en una **QComboBox**. Después seleccionaremos la capa con la que se formará el mapa base[10] y el segmento de *Adquisición* mediante **QComboBox**.

Finalmente utilizamos los **QRadioButton** inferiores para seleccionar el tipo de entrada, una única imagen representando todos los eventos o múltiples imágenes pertenecientes a cada uno de estos, buscamos en el **QComboBox** superior derecho filtrando con un **QLineEdit** los eventos correspondientes y con los **QPushButton** de simbolo -ż .ªñadimos dichos eventos a la **QListView** inferior. Una vez hecho esto, rellenamos los datos de *Adquisición* en la esquina inferior derecha mediante tres **QLineEdit**, uno para el país, otro para el área de la región y el ultimo para el tipo[11].

Ingest Image

La siguiente herramienta 3.23 está diseñada con el fin de permitir a la empresa, especialmente a los analistas, subir imágenes de Satélite al proveedor de la empresa si este no dispone de las mismas. Se comienza seleccionando la imagen a subir, esta puede provenir de un archivo *tiff* o de una capa generada en *QGIS* mediante un **QgsExternalResourceWidget** o una **QgsMapLayerComboBox** respectivamente, habiendo sido seleccionado su **QRadioButton** correspondiente.

Seguidamente se selecciona la colección[12] a la que añadir la selección previa mediante otro **QComboBox**.

Por último hay que rellenar la información referente al *bucket* de *AWS* de la empresa, donde se almacenan todos los archivos relacionados con los diferentes satélites asociados. Para esto, comenzamos eligiendo el satélite proveedor de la imagen mediante una **QComboBox** y la fecha y hora de dicha imagen usando un **QCalendarWidget** y un **QTimeEdit**. Tras esto elegimos se crear o no una nueva carpeta en el *bucket* usando su **QRadioButton** asociado, y finalmente añadimos la información mediante una **QComboBox** para una carpeta existente o un **QLineEdit** si queremos crear una carpeta nueva.

Cabe destacar un **QLineEdit** bloqueado al uso en la esquina inferior derecha, esto se debe a que por el momento la empresa solo dispone de un *bucket* pero plantea tener más en el futuro.

[9] Por el momento solo se encuentran disponibles las traducciones al Español, Ingles y Árabe.
[10] Mapa del mundo sobre el cual dibujar las capas de los eventos
[11] Valor establecido internamente por la empresa
[12] El proveedor agrupa la información por colecciones, es trabajo del analista elegir a cual pertenece cada imagen añadida.

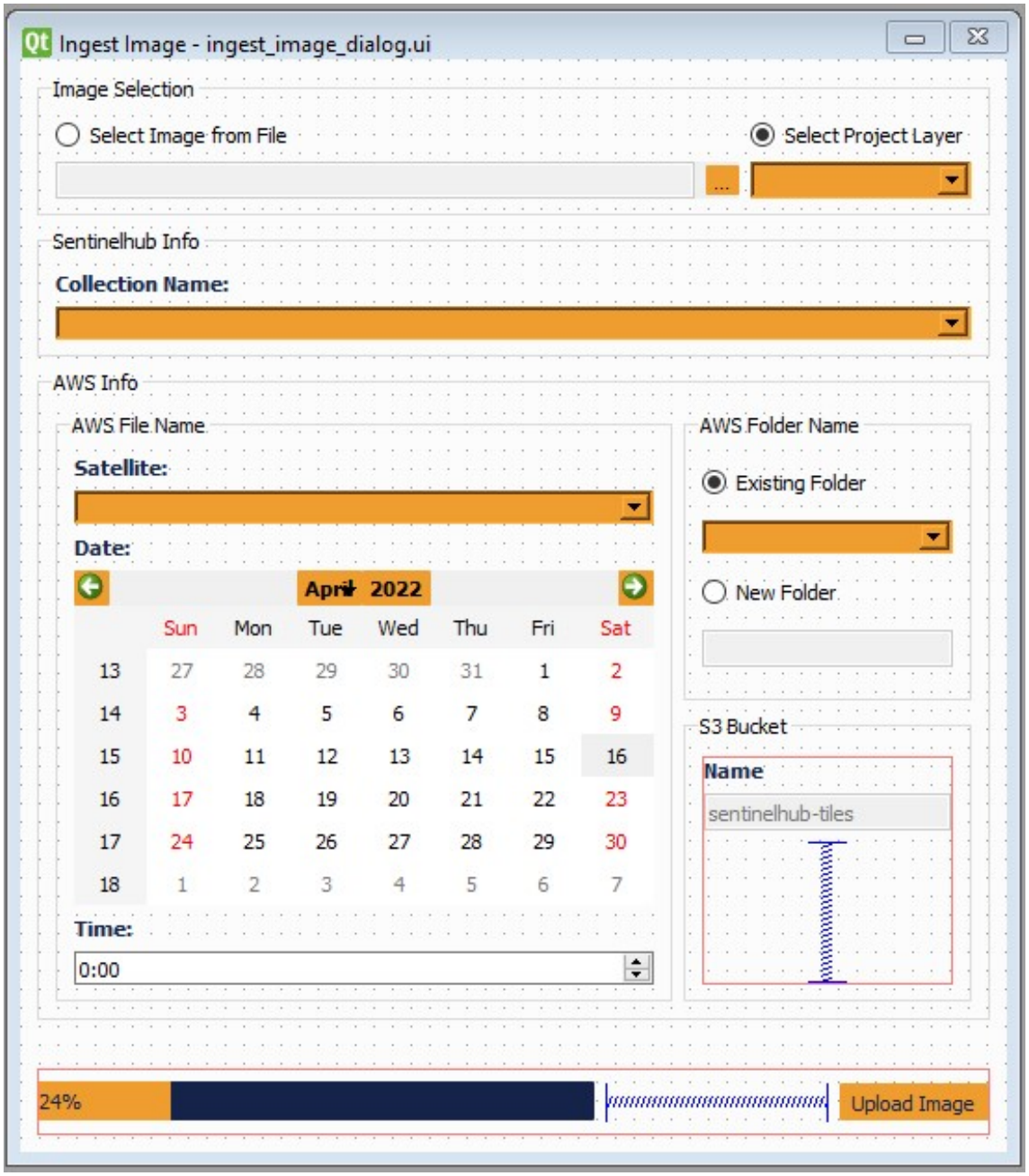

Figura 3.23: Diseño final de la herramienta Ingest Image

Ingest Geo Image

Al igual que en la herramienta anterior 3.7.2 encontramos un funcionamiento y apariencia similar, en este caso para satélites de *GeoServer* ya que necesitan de un procesado concreto y provienen de fuentes diferentes.

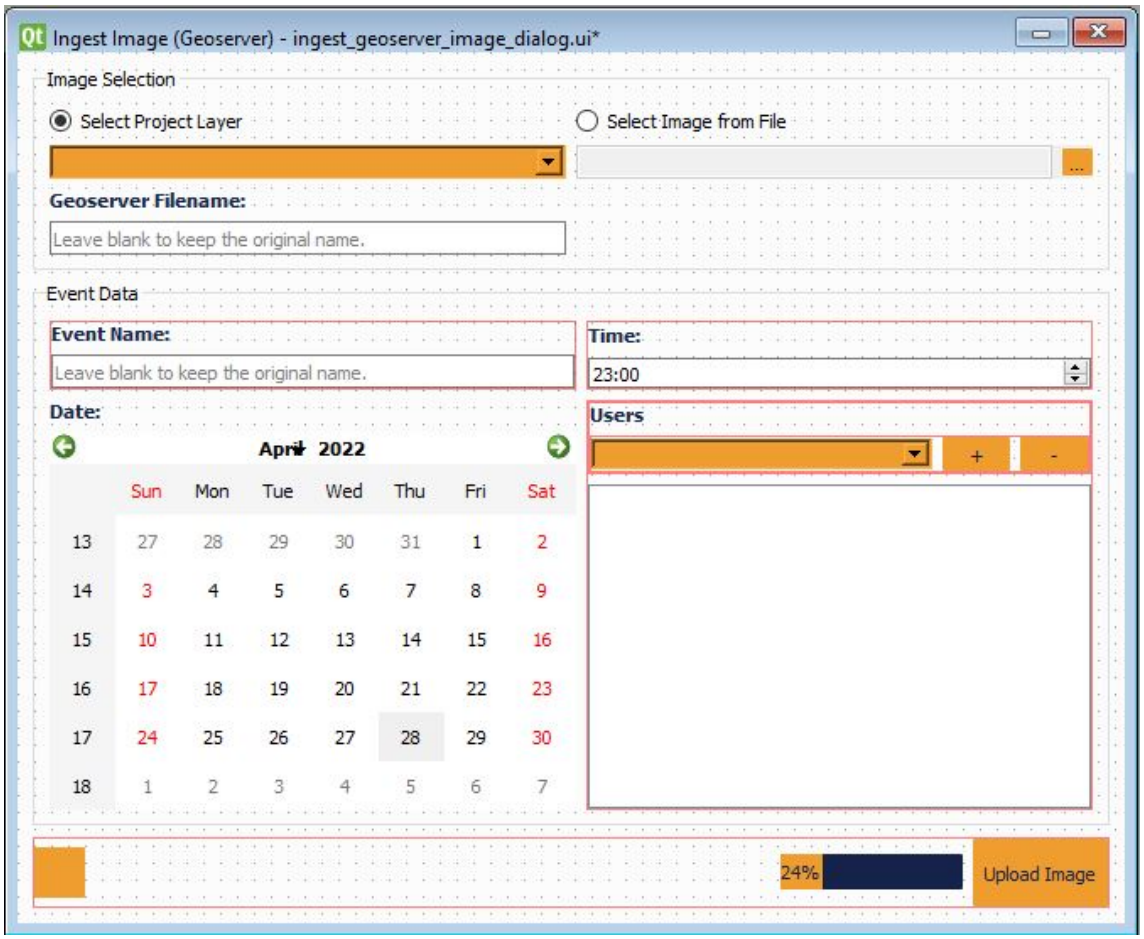

Figura 3.24: Diseño final de la herramienta Ingest Geo Image

Siendo que esta herramienta esta compuesta muy similarmente a la anterior, describiremos las diferencias. En este caso habrá se podrá añadir un nombre para el evento si así se desea mediante un **QLineEdit** y habrá que añadir mediante una **QComboBox** y los **QPushButton** adyacentes, los usuarios(Clientes) a la **QListView** inferior. Estos usuarios son aquellos a los que se le permitirá el acceso a dichos eventos con el fin de realizar la demostración adecuada.

Normal Extract

La siguiente herramienta 3.25, se compone de tres ventanas individuales que trabajan con un objetivo en común.

En la primera ventana 3.26 encontramos la toda la utilidad principal. Esta herramienta se divide en 5 pestañas ordenadas para hacer el uso de la misma lineal. Primeramente

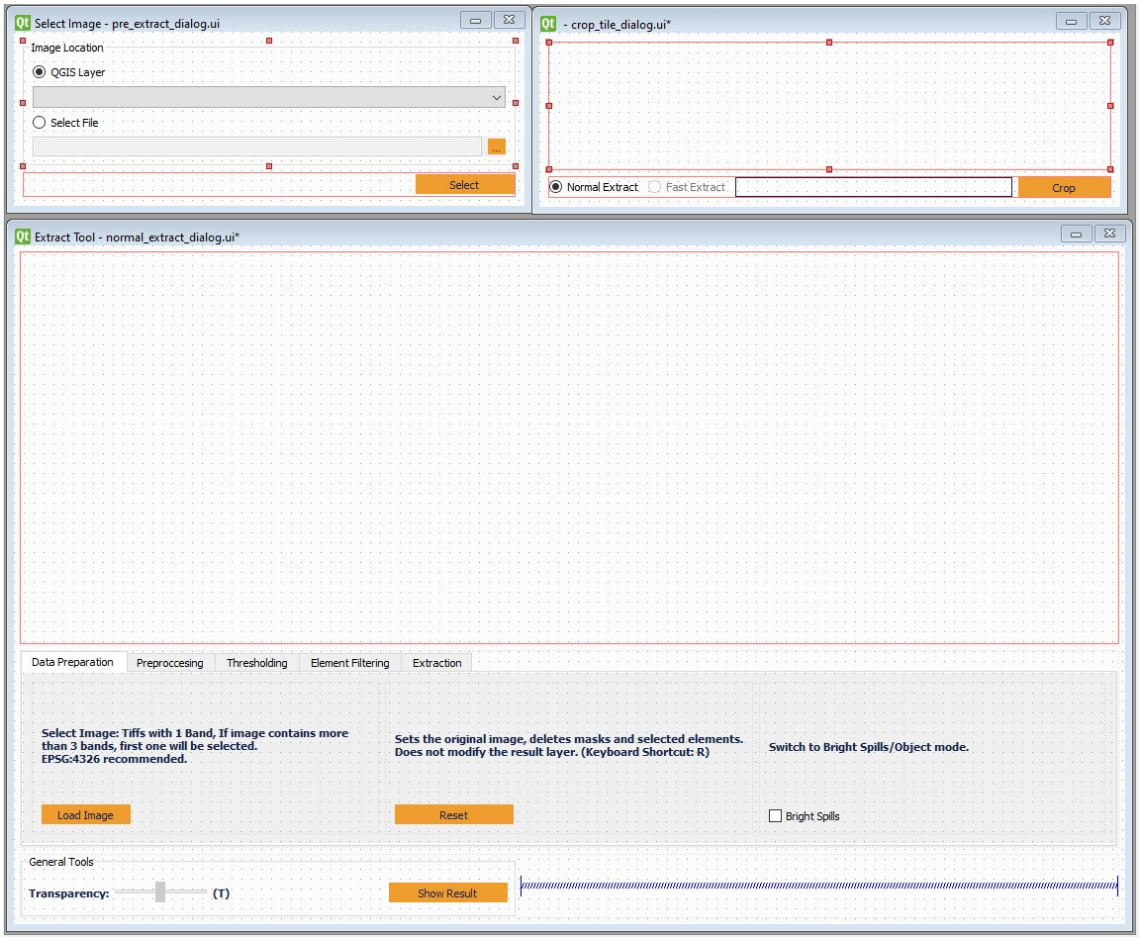

Figura 3.25: Diseño final de la herramienta Extract Spill

observamos un **QWidget** que utilizaremos de lienzo para mostrar las imágenes cargadas mediante la ventana 3.32, un **QTabWidget** donde aparecen las pestañas mencionadas previamente y un **QGroupBox** donde aparece un **QSlider** para controlar la transparencia de la imagen y un **QPushButton** para ver el resultado actual del proceso en cualquier momento.

En la primera pestaña *Data Preparation* 3.27 aparecen todo lo necesario para comenzar el proceso de extracción. Clickando el **QPushButton** *Load Image* se abrirá la ventana 3.32 donde podremos cargar la imagen a la herramienta. Seguidamente encontramos otro **QPushButton** con el cual podremos reiniciar el proceso de extracción y por último una **QCheckBox** para indicar si la imagen requiere un mayor brillo[13] cambiando ciertos valores en el procesado de umbrales.

En la segunda pestaña *Preprocessing* 3.28 encontramos las herramientas de preprocesado. Estas se dividen en filtros, situados a la izquierda y transformaciones a la derecha. Los filtros se componen de cinco **QPushButton** los cuales aplican los siguientes filtros:

```
Median Filter - Filtro de Mediana
Erode - Erosion
Dilate - Dilatacion
Opening - Aberturas
Closing - Cerrado
```

[13]Una imagen puede requerir un brillo más alto si las manchas son demasiado oscuras, ya que pueden confundirse con marea o ser inapreciables.

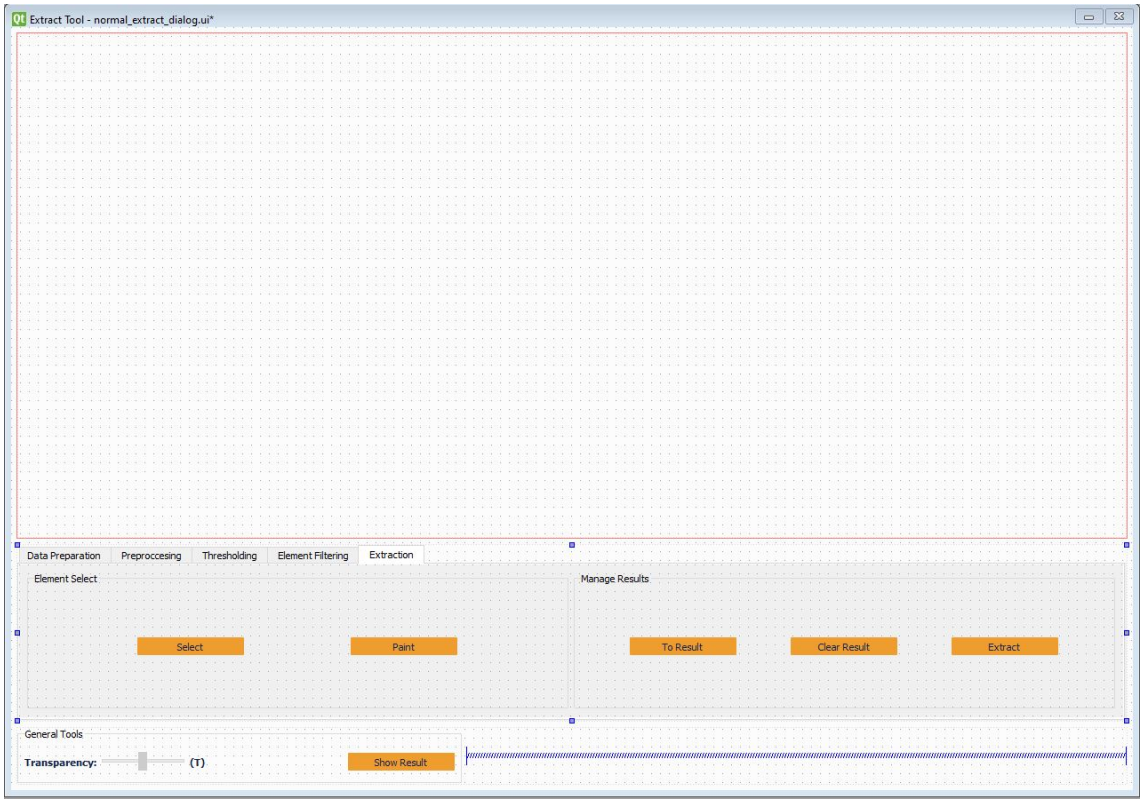

Figura 3.26: Diseño final de la primera venta de la herramienta Normal Extract

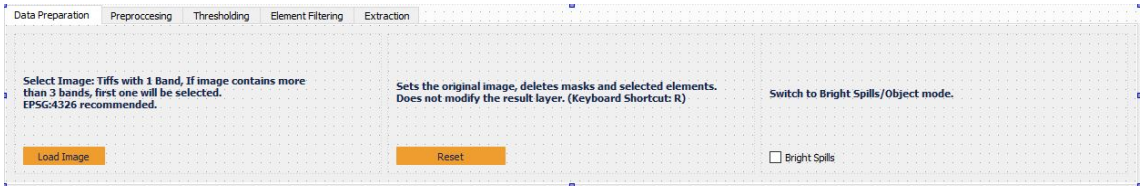

Figura 3.27: Diseño final de la primera pestaña de la herramienta Normal Extract

Además también encontramos un **QSpinBox** para delimitar el tamaño del kernel. En el lado derecho encontramos otros cuatro **QPushButton** para aplicar las siguientes transformaciones:

```
SQUARE UP - Imagen al cuadrado
CLAHE - Optimizacion del histograma por limite de contraste
Hist Equal - Optimizacion de histograma
Stretching - Estirar imagen
```

Este ultimo **QPushButton** va predecido por dos **QSpinBox** utilizados para delimitar el valor máximo y mínimo con el cual aplicar dicha transformación.

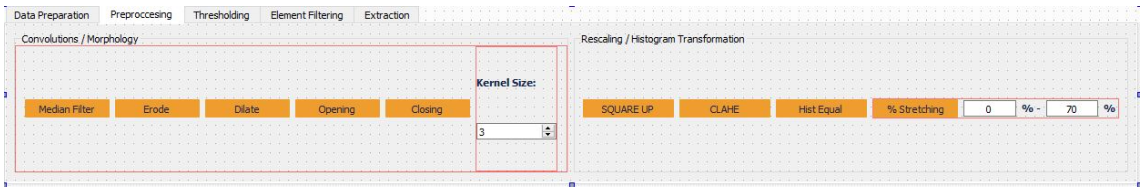

Figura 3.28: Diseño final de la segunda pestaña de la herramienta Normal Extract

En la tercera pestaña *Thresholding* 3.29 encontramos primeramente cuatro **QRadioButton** que habilitarán los elementos situados bajo ellos. Eligiendo *Normal* se habilitara el **QSlider** de *Normal Tresholding* y su **QSpinBox** adyacente. Seleccionando *Adaptative* activaremos los **QSlider** y **QSpinBox** situados bajo *Adaptative thresholding*. Con *Otsu* activado se inhabilitan todos los **QSlider** inferiores y activando *K+NN* se habilitará el **QPushButton** y **QSpinBox** situado bajo *K-Nearest Neighbour*.

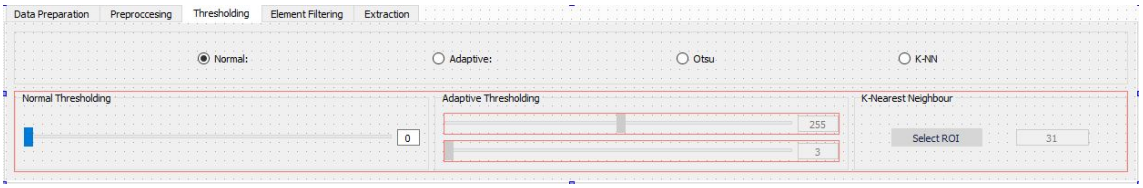

Figura 3.29: Diseño final de la tercera pestaña de la herramienta Normal Extract

En la cuarta pestaña *Element Filtering* 3.30 encontramos tres **QPushButton**, *Erase* y *Clip* eliminan dos tipos de objetos no deseados en el *threshold*. *Start* sirve para eliminar todos los objetos cargados, de tamaño inferior al especificado por el **QSlider** inferior. Este último filtrado es muy útil para eliminar puntos generados sobre elementos pequeños irrelevantes a la hora de realizar el análisis.

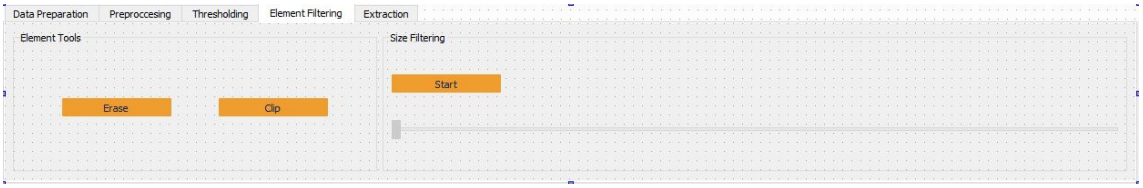

Figura 3.30: Diseño final de la cuarta pestaña de la herramienta Normal Extract

Por último en la quinta pestaña *Extraction* 3.31 podemos hacer uso de cinco **QPushButton**. *Select* permite seleccionar uno o varios elementos mostrados en el lienzo, *Paint* pinta el elemento seleccionado, utilizado principalmente para resaltar ciertos elementos a la hora del análisis. *To Result* permite al analista volver a iterar sobre la extraccion filtrando de nuevo sobre los elementos ya filtrados, *Clear Result* limpia la lista de elementos extraídos y por ultimo *Extract* finaliza el proceso generando una *capa vectorial* de *QGIS* y extrae los elementos como un documento *shapefile*.

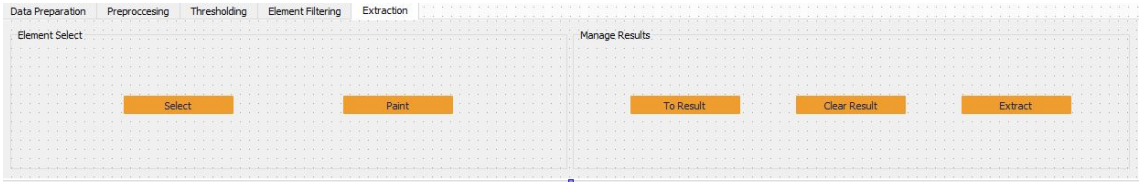

Figura 3.31: Diseño final de la quinta pestaña de la herramienta Normal Extract

En la segunda ventana 3.32 podemos seleccionar si vamos a utilizar una capa de *QGIS* o un archivo local en formato *tiff* mediante dos **QRadioButton** para luego utilizar un **QgisMapLayerComboBox** o un **QgsExternalResourceWidget** para seleccionar dichos archivos respectivamente.

Por último, en esta tercera ventana 3.33 podremos recortar la imagen seleccionada mediante la ventana previa 3.32 utilizando un **QWidget** como lienzo donde realizar el recorte. Seguidamente con dos **QRadioButton** seleccionaremos el tipo de extracción que

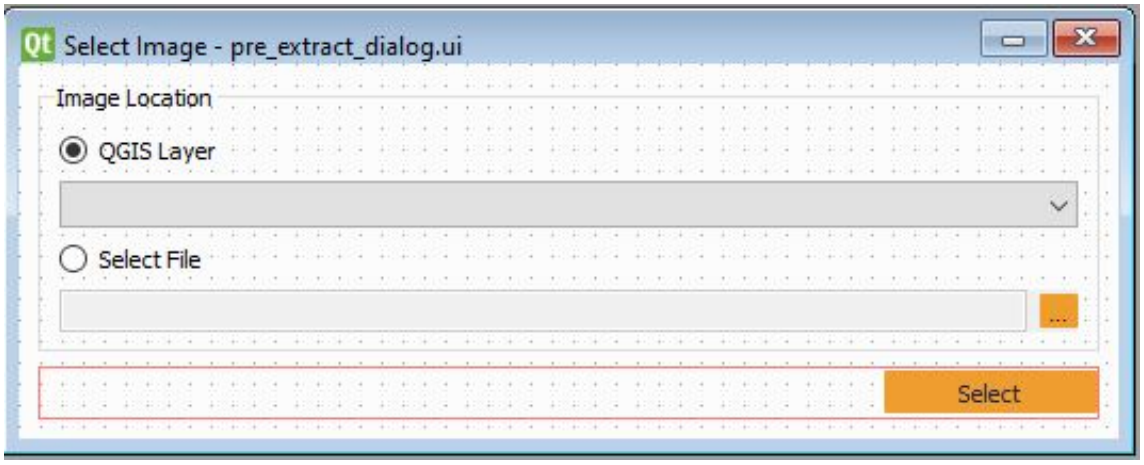

Figura 3.32: Diseño final de la segunda ventana de la herramienta Normal Extract

se quiere realizar, normal o rápida[14] y el **QPushButton** *Crop* para enviar la imagen a la primera ventana 3.26 y comenzar el proceso de filtrado y extracción.

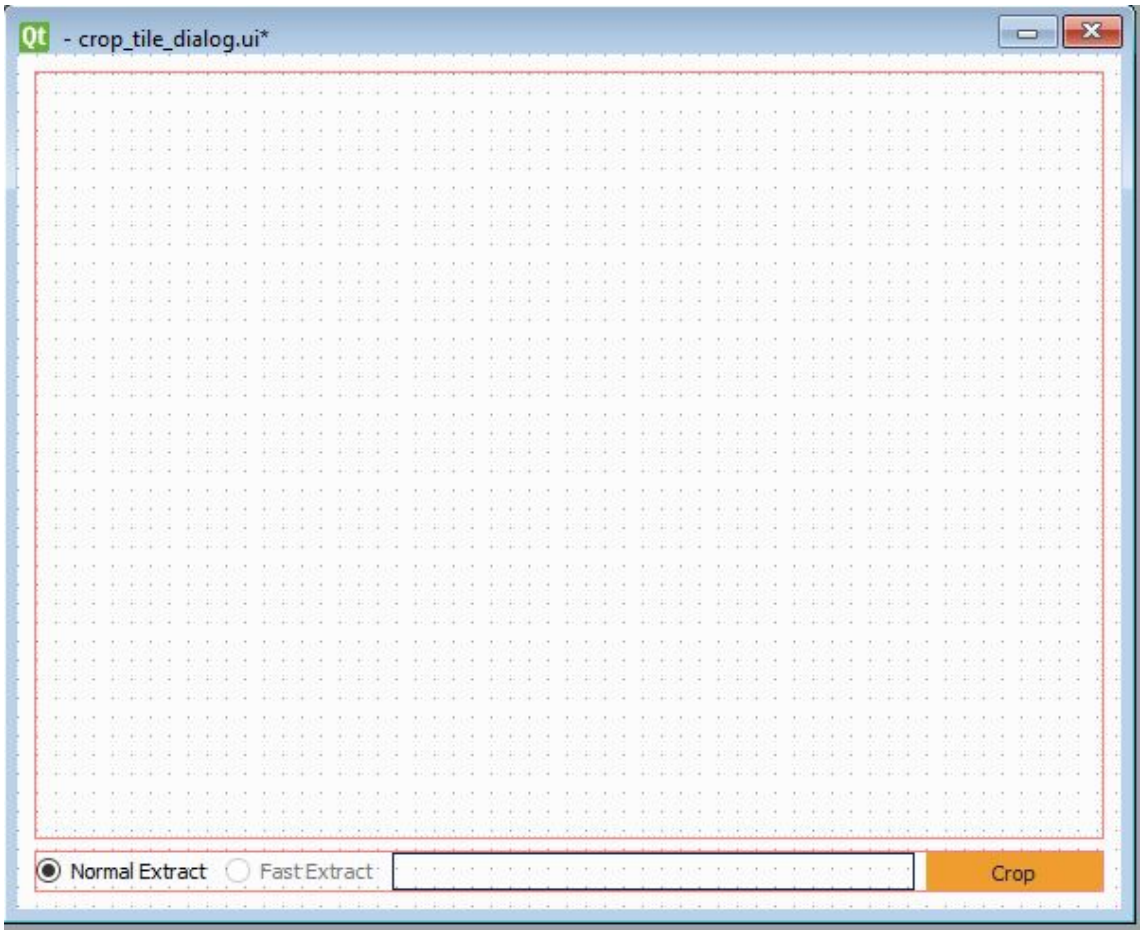

Figura 3.33: Diseño final de la tercera ventana de la herramienta Normal Extract

Merge Vector Layer

[14]Por el momento la funcionalidad de extracción rápida no se encuentra disponible.

La siguiente 3.34, es una herramienta muy sencilla pero muy útil al mismo tiempo. Con ella, se puede realizar la unión de dos capas vectoriales en una sola *capa vectorial* de *QGIS*.

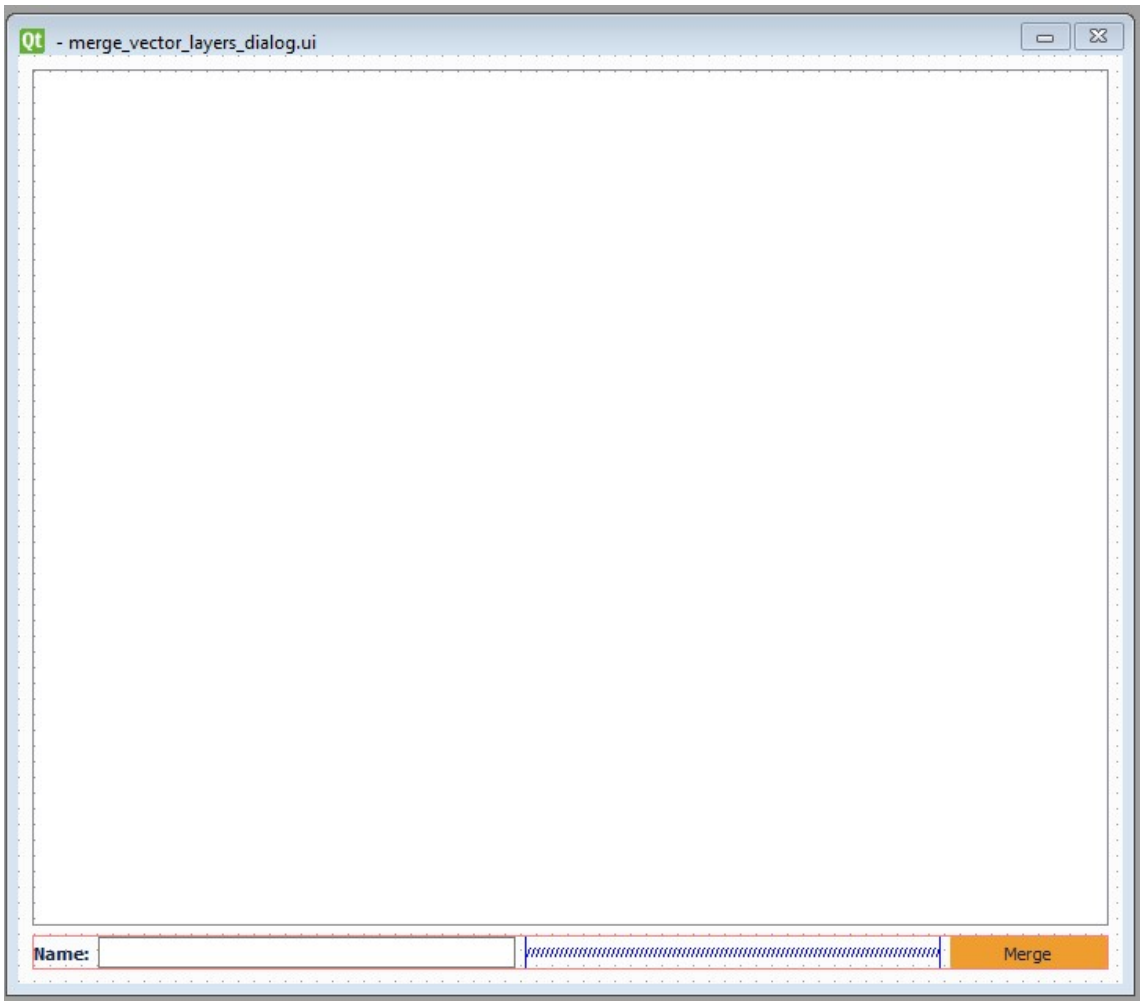

Figura 3.34: Diseño final de la herramienta Merge Vector Layer

Para esto hacemos uso de un **QListWidget** donde podremos ver las capas vectoriales cargadas en *QGIS*, seleccionaremos dos de estas y le daremos un nombre mediante un **QLineEdit**. Seguidamente accionando el **QPushButton** de Merge comenzará a realizar el algoritmo de procesado.

Dicho algoritmo no presenta mayor complicación que aquella de encontrar la solución correcta. En unas pocas lineas de código conseguimos unificar ambas capas en una, manteniendo la relación entre ellas. Esto es gracias a la funcionalidad aportada de *QGIS* y una conversión de datos entre las capas provistas, dado que previamente a la unión es necesario obtener el mismo tipo de capa en ambos archivos[15].

Process Polygon

[15]Hay varios tipos de capas geométricas dentro de *QGIS*, como pueden ser: Capas de puntos, Capas de polígonos, Capas de lineas, etc...

La siguiente 3.35, es una herramienta de simplificado de *capas vectoriales* mediante tres procesos, eliminación de objetos de tamaño menor al escogido, eliminación de objetos de tamaño mayor al escogido y eliminación de vértices en función a la distancia que los separa intentando conservar la geometría original.

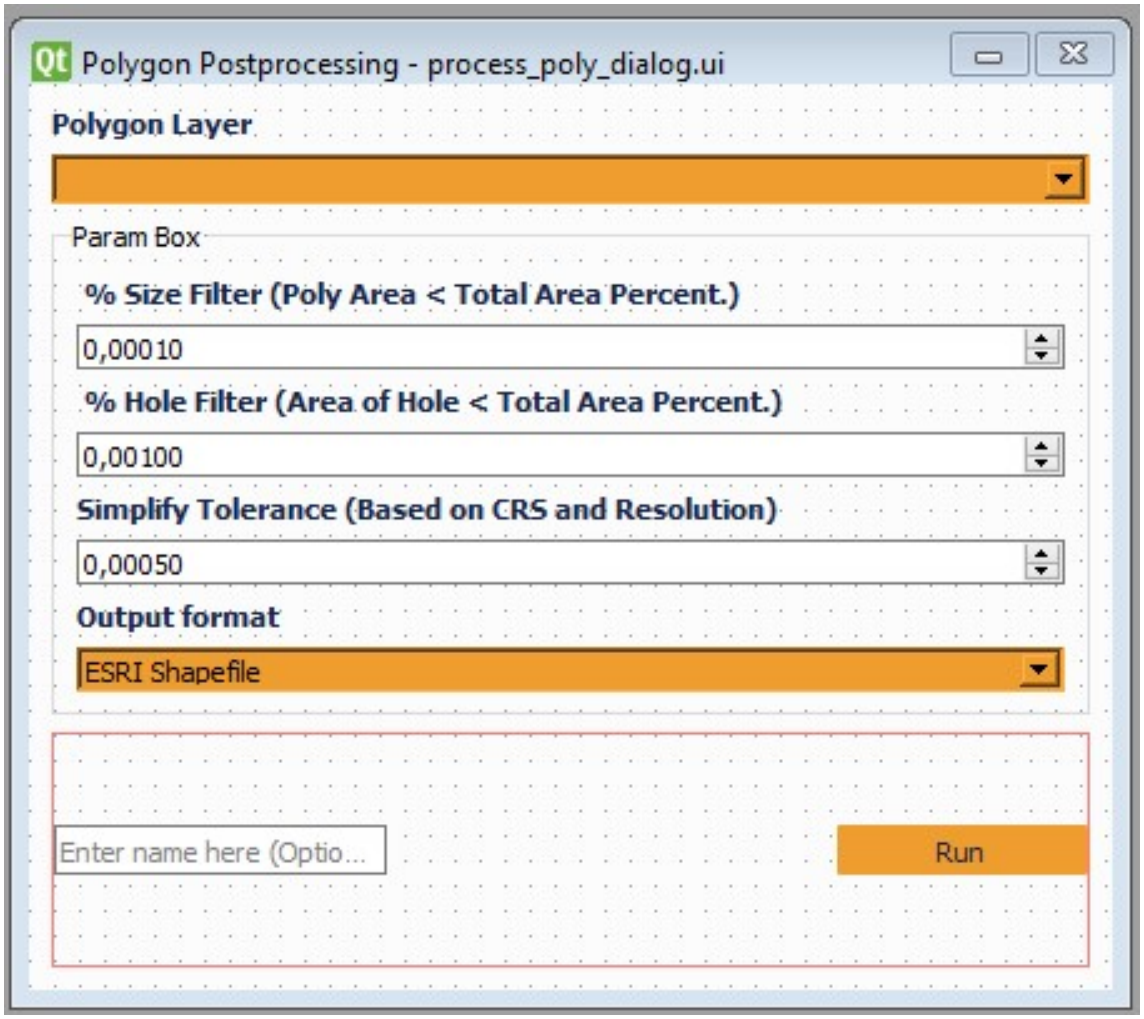

Figura 3.35: Diseño final de la herramienta Process Polygon

Todo esto se consigue eligiendo una capa mediante un **QgsMapLayerComboBox** y estableciendo los parámetros para el filtrado mencionado previamente con tres **QDoubleSpinBox**, finalmente elegimos el formato de salida que puede ser *ESRI Shapefile*, *GEOJSON* y GPKG. Opcionalmente se puede elegir un nombre para este archivo utilizando el **QLineEdit** situado en el lado inferior izquierdo.

Sentinel Preprocess

En la siguiente herramienta 3.36 encontramos uno de los procesos principales y más importantes del procesado de imágenes, el preprocesado de las mismas. Con esta herramienta los analistas pueden preparar las imágenes para ser tratadas y analizadas correctamente.

La empresa hace uso de dos proveedores, Sentinel 1 y Sentinel 2, estos son satélites públicos proporcionados por la Agencia Espacial Europea. Dado que recibimos dos tipos

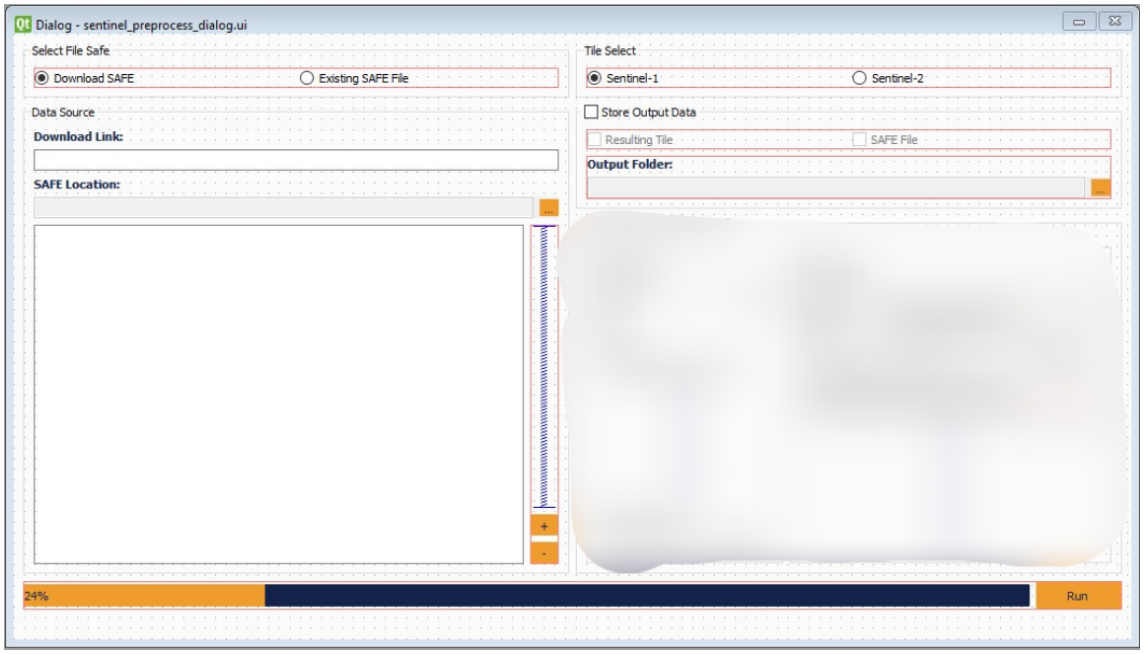

Figura 3.36: Diseño final de la herramienta Sentinel Preprocess

de imágenes diferentes de Sentinel, Radar y Óptico en este orden, y puede que esto se expanda en un futuro, utilizamos esta herramienta para preprocesar todas las imágenes a un formato con el que QGIS pueda trabajar.

Se comienza seleccionando una imagen clickando primero en un **QRadioButton** que determina si tenemos ya descargado el archivo o vamos a utilizar directamente el enlace proporcionado por el proveedor[16]. Seguidamente pegamos el enlace de descarga en el **QLineEdit** o seleccionamos la ubicación del archivo si ya lo tenemos descargado con un **QgsExternalResourceWidget**. Este proceso se puede repetir tantas veces quiera el analista, añadiendo estos archivos a la **QListView** inferior haciendo uso de los **QPushButton** al lado de la misma para añadir o eliminar dichos archivos.

Después haciendo uso de más **QRadioButton** elegimos el tipo de proveedor, con una **QCheckBox** se determina si se quiere almacenar la salida[17], si se ha elegido esta opción, habrá que seleccionar cómo queremos guardarlo mediante dos **QCheckBox** para el *Tile* y el archivo *SAFE*. Seguidamente mediante un **QgsExternalResourceWidget** se selecciona donde almacenar la salida.

Finalmente llegamos a la parte de la herramienta que aparece difuminada, considerando que forma parte de los parámetros necesarios para realizar dicha transformación y esto es información clasificada. A pesar de esto podemos explicar la funcionalidad de dicho algoritmo y esta es la siguiente. Mediante la generación de distintos índices se consigue generar un resalto de las verdaderas manchas situadas en el mar y hace que el contraste sea más efectivo, esto se consigue utilizando bandas de imágenes de distintos satélites.

Upload Patches

[16] Descargar los archivos facilita enormemente el trabajo a los analistas ya que pueden trabajar con el resto de herramientas mientras se descarga y procesa la imagen final.

[17] Por defecto las imagenes procesadas se cargan como capas de *QGIS*.

La siguiente interfaz 3.37 aparece para cubrir una necesidad a futuro de la empresa, se pretende integrar una inteligencia artificial para facilitar el trabajo de los analistas. Para lo siguiente, será necesario la subida de imágenes preparadas para que la *IA* utilice en su proceso de aprendizaje.

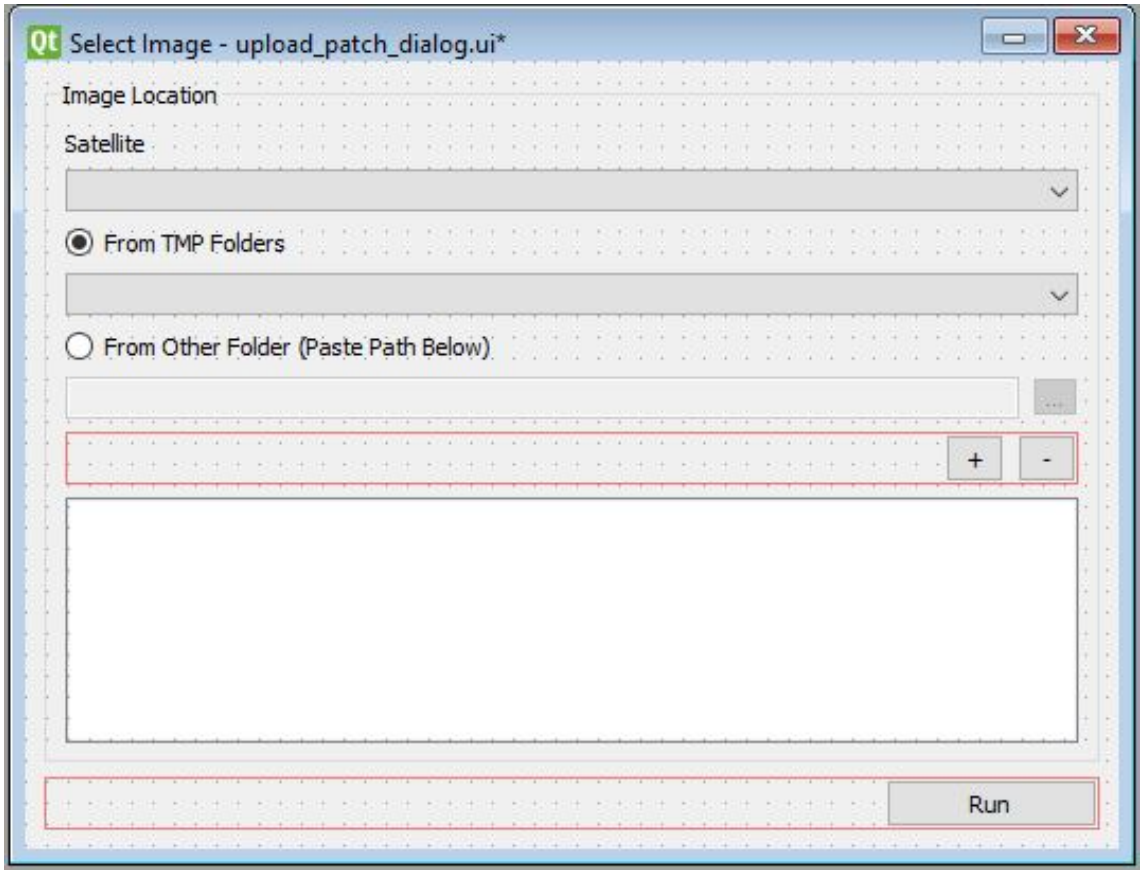

Figura 3.37: Diseño final de la herramienta Patch Table

Primeramente encontramos un selector de satélite mediante una **QComboBox** seguido de dos **QRadioButton** para seleccionar la ubicación de los archivos a subir, siendo posible utilizar una carpeta situada en la ubicación por defecto o una seleccionada manualmente haciendo uso de un **QComboBox** y un **QgsFileWidget** respectivamente. Después utilizaremos los dos **QPushButton** inferiores para añadir o eliminar archivos de la **QListView** bajo los mismos. Una vez hecho esto, los archivos se suben al servidor de la empresa al pulsar el **QPushButton** inferior derecho.

Patch Table

Esta herramienta 3.38 funciona como complemento de la mencionada previamente 3.7.2 sirviendo como visualizador de todos los elementos previamente subidos al servidor de la empresa.

Esto se consigue haciendo uso de un**QTableWidget** para mostrar todos los eventos y sus datos, mostrando principalmente el identificador del evento además del numero de *patches* que este tiene. Para poder mostrar eventos en esta tabla será posible buscarlos mediante un QLineEdit y un pequeño filtro para el tipo de satélite situado en la esquina superior derecha en forma de QComboBox seguido de un QPushButton para realizar la búsqueda.

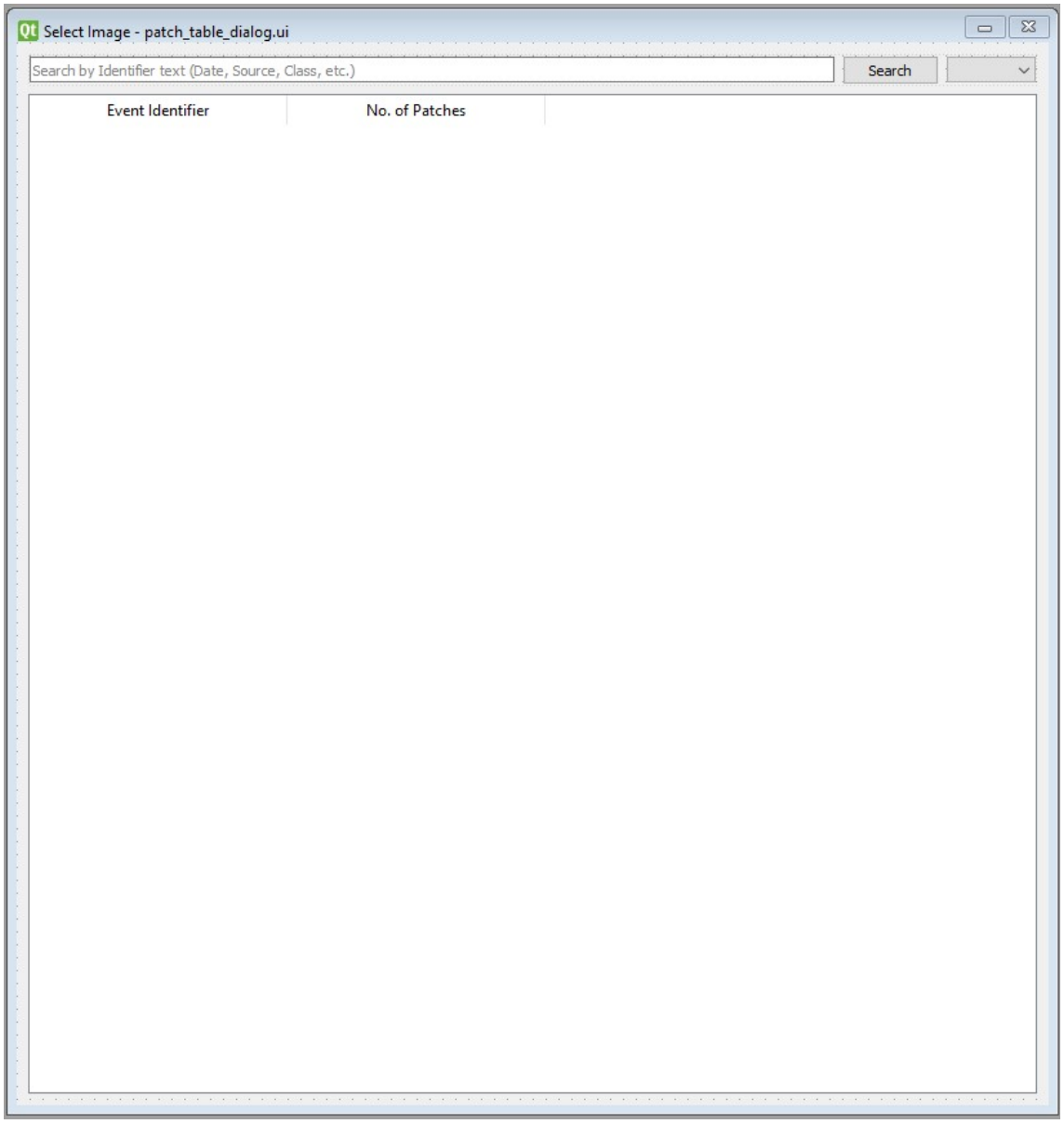

Figura 3.38: Diseño final de la herramienta Patch Table

Flujo de la aplicación

A continuación se muestra el flujo de uso de la aplicación estándar. Este proceso comienza con la carga de una imágen haciendo uso de la herramienta *Sentinel Preprocess* 3.7.2, en esta utilizaremos un enlace para obtener el *tile* usado en el análisis.

Tras haber cargado la imagen correctamente, esta se mostrará en *QGIS* como se observa en la imagen 3.39.

Seguidamente haremos uso de la herramienta *Normal Extract* 3.7.2 para recortar, filtrar y finalmente extraer el polígono que marcará la mancha encontrada. Este proceso se realiza intermanete en esta herramienta mediante sus diferentes pantallas como se puede observar en las siguientes imágenes 3.40.

Finalmente se utilizará esta imagen en la herramienta *Add Event* 3.7.2 junto con la información provista por el analista para generar un nuevo evento en la base de datos de la aplicación. Las capas obtenidas y generadas mediante la extracción previa se pueden

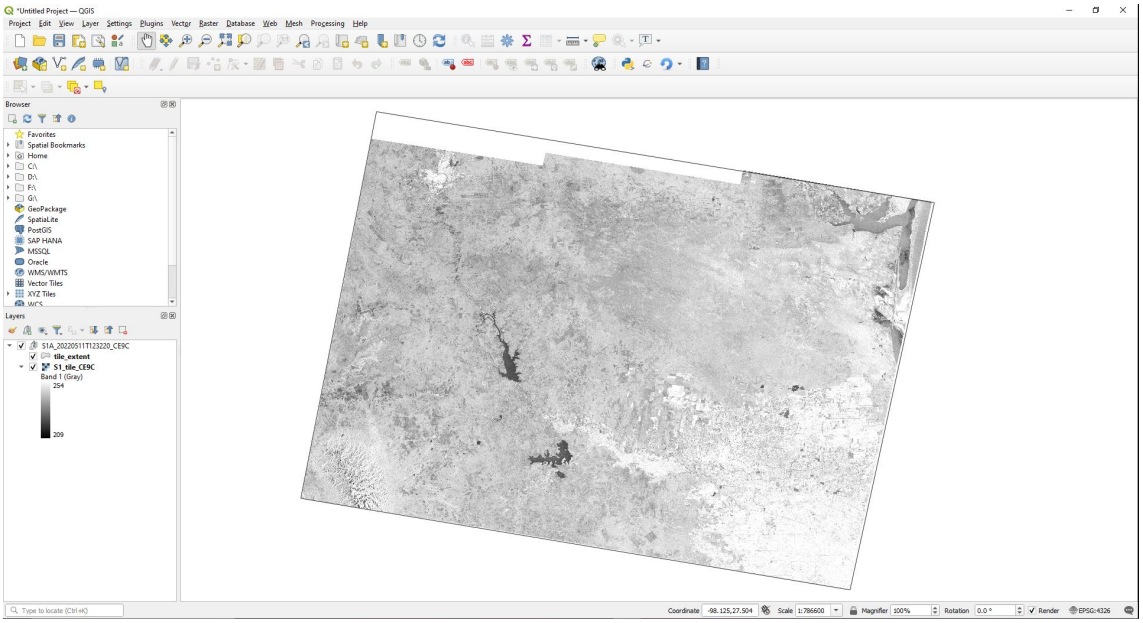

Figura 3.39: Imagen cargada en QGIS

añadir al evento en los selectores provistos en la segunda pestaña como se puede observar en la siguiente imagen 3.41.

Cabe destacar que parte de la información utilizada para la generación del evento, en concreto la región y la fuente, habrán sido creadas previamente mediante sus herramientas correspondientes *Add Source* 3.7.2 y *Add Region* 3.7.2.

Adicionalmente, si el cliente lo requiere, el analista podrá hacer uso de la herramienta *Build Report* 3.7.2 y los eventos generados previamente para crear un documento *PDF* que será entregado al cliente en su idioma de preferencia 3.42.

De esta forma obtendríamos un flujo completo de análisis para cierto cliente por parte de los usuarios de la aplicación. Adicionalmente el resto de herramientas podrían ser útiles dentro del flujo estándar bajo ciertas circunstancias que requieran el uso de las mismas para alterar las imágenes obtenidas.

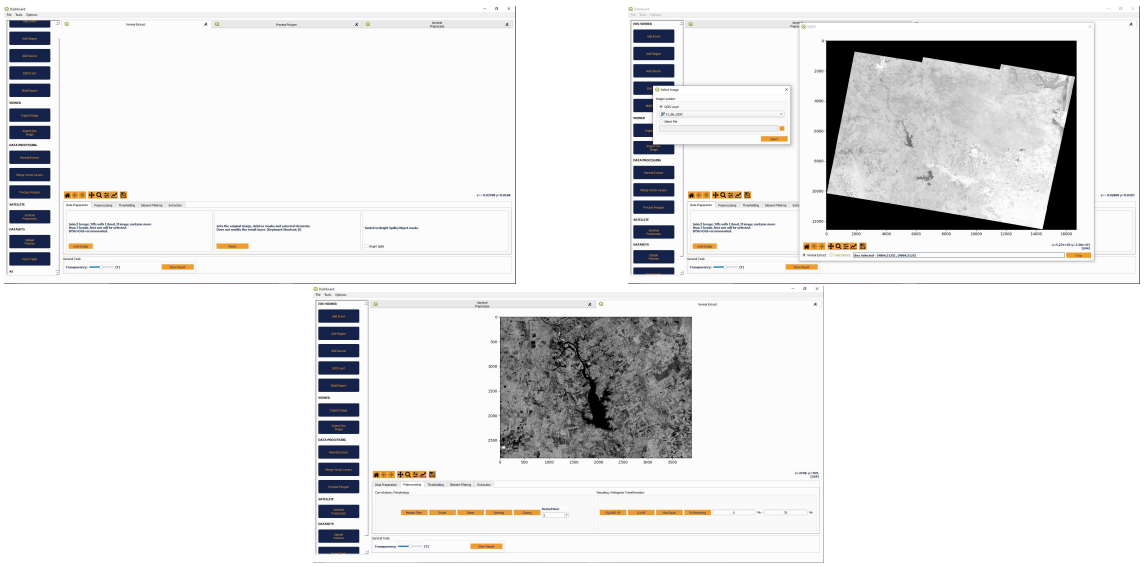

Figura 3.40: Uso de la herramienta Normal Extract

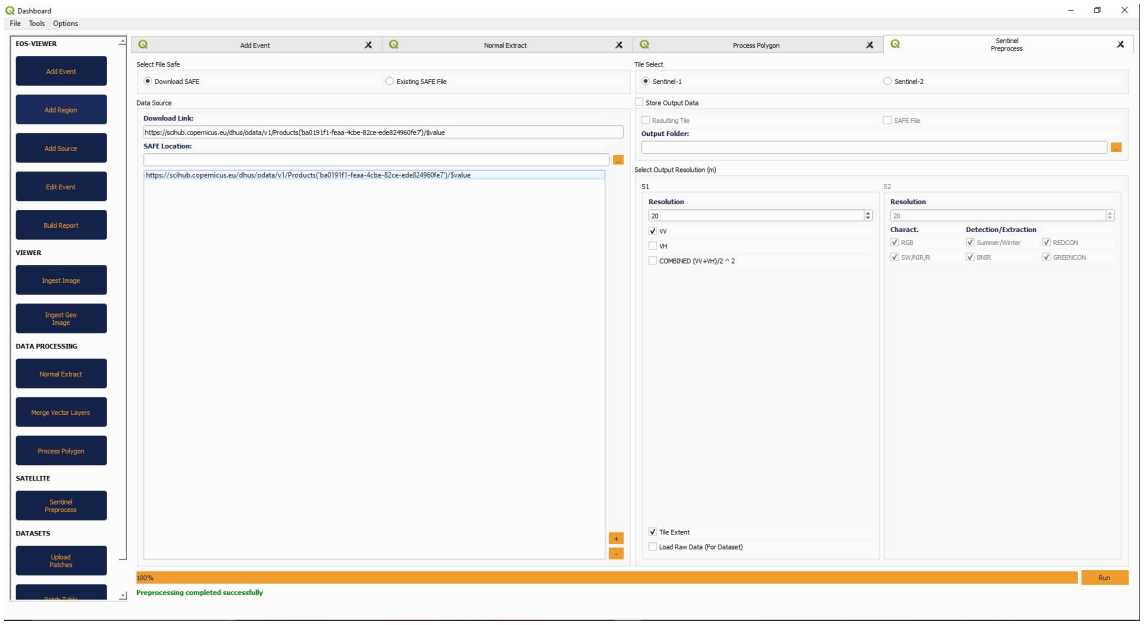

Figura 3.41: Uso de la herramienta Add Event

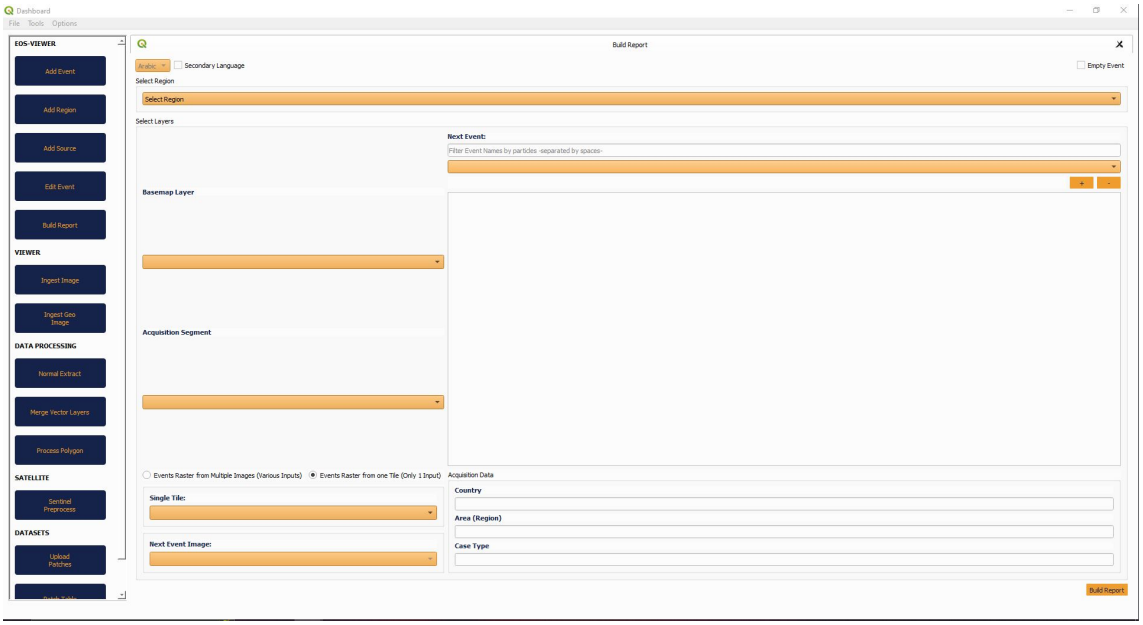

Figura 3.42: Uso de la herramienta Build Report

3.8 Pruebas - Testing

A la hora de realizar las pruebas de la aplicación se nos plantea el problema de la escasez de tiempo y recursos para la realización de las mismas. Dado que la metodología 3.1 a seguir consiste en generar prototipos para los clientes(analistas) y obtener su *feedback*, aprovecharemos sus pruebas sobre los protitipos para obtener cualquier tipo de errores sobre la aplicación *frontend*, aparte de tener como parte del equipo de desarrollo a un analista, que realiza las pruebas que podrían resultar más rompedoras para la aplicación permitiéndonos así solucionar dichos errores previo al lanzamiento de la siguiente versión de la aplicación.

Por otro lado, la parte del *backend* de la aplicación utiliza el *framework* de Django que facilita enormemente el testeo automático de la aplicación. De este modo podemos encargarnos de automatizar un frente de la aplicación sin mucho coste.

Pruebas automáticas, ¿Que son?

En el mundo de las pruebas automatizadas encontramos tres grandes tipos de pruebas, las unitarias, que consisten en probar individualmente los diferentes componentes de la aplicación, las de Regresión, que intentan replicar errores típicos para prevenir los mismos y las de integración que prueban la aplicación al completo en todo su conjunto, comprobando el buen funcionamiento del mecanismo al completo.

3.8.1. Pruebas en django

En este caso, la aplicación *backend* es únicamente una *API REST*, por lo tanto, solo existe la necesidad de realizar pruebas unitarias a cada funcionalidad de dicho servicio. Para ello se hará uso de un módulo provisto por **rest_framework.test**[21] llamado **APIRequestFactory**, con este módulo se podrán realizar peticiones de todo tipo al servidor y simular el proceso de autenticación.

Este módulo nos permite simular la solicitud de recursos que realizaría una aplicación *frontend* desde el propio servicio *backend*. Así mismo, podremos utilizar estas llamadas para comprobar con muestras de datos pre establecidas si el funcionamiento de los servicios es el adecuado y esperado.

```
# Django
from django.test import TestCase

# Python
import json

# Django Rest Framework
from rest_framework.test import APIClient
from rest_framework import status

# Models
from ejemplo.models import EjemploModel
```

```
from users.models import User

Class TestDeEjemplo(TestCase):
    def setUp(self):
        # Se inicializa el usuario para tener acceso a la base de
            ↪ datos
        user = User(email='...', password='...')

        #Creamos el cliente que se encargado de las peticiones
        client = APIClient()
        # Obtenemos la respuesta del inicio de sesion y guardamos en
            ↪  la clase el token de accesso
        response = client.post('/users/login', user)
        self.access_token = json.loads(response.content)['
            ↪ access_token']
        self.user = user

    def test_get_ejemplo(self):
        # creamos un cliente para realizar peticiones
        client = APIClient()
        # Pasamos las credenciales obtenidas en el setup
        client.credentials(HTTP_AUTHORIZATION='Toke' + self.
            ↪ access_token)
        # Creamos el resultado que supuestamente a obtener
        mockResult = '.....'
        # Realizamos la request a nuestro servicio
        response = client.get('/api/ejemplo')
        # Leemos la respuesta en json y la guardamos
        result = json.loads(response.content)
        # Comprobamos que el resultado obtenido y el esperado son el
            ↪  mismo
        self.assertEqual(result, mockResult)
```

De este modo comprobaríamos cada tipo distinto de petición (post, put, get, del) de cada uno de nuestros servicios, utilizando datos de los cuales conocemos su resultado, para poder preparar las comprobaciones a realizar tras obtener el resultado de nuestro servidor.

Como vimos previamente en la generación de la plantilla de *django* 3.6.2, tras la creación de cada diferente aplicación se generará un archivo llamado test.py donde situaremos todos los test unitarios para ese servicio concreto. Una vez hecho esto será tan sencillo como llamar al siguiente comando para lanzar todos los test de nuestro proyecto *django* al completo.

```
$ python manage.py test
```

Seguidamente en la consola donde se haya lanzado dicho comando, obtendremos un resultado en forma de tabla mostrándonos todas las pruebas realizadas, sus resultados y en caso de fallo, su ubicación y dicho error.

```
    FAIL: test_was_published_recently_with_future_poll (polls.tests.
        ↪ PollMethodTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/path/to/test/tests.py", line X, in some_function
    line content goes here
ErrorType: error motive


----------------------------------------------------------------------
Ran 1 test in 0.003s

FAILED (failures=1)
```

De esta forma obtendremos para cada uno de los elementos presentes en nuestra *API* una forma rápida y sencilla de comprobar automáticamente un funcionamiento correcto de los mismos.

CAPÍTULO 4
CONCLUSIONES

A lo largo de este trabajo hemos logrado implementar las diferentes características necesarias para agilizar y facilitar el trabajo de los analistas marítimos de la empresa. Aún queda mucho trabajo por realizar en esta herramienta y esta diseñada con esa idea, para permitir una buena expansión conforme los analistas vean necesarias más utilidades.

Durante el desarrollo de la herramienta nos hemos encontrado con ciertas dificultades técnicas dado que *QGIS* es una herramienta muy delicada y el tratado de imágenes nunca es una tarea fácil. Pero gracias a una buena planificación y mucho tiempo de investigación se ha conseguido obtener buenos resultados y a gusto de la empresa.

Bibliografía

[1] *Metodologías Tradicionales* https://www.ecured.cu/Metodolog%C3%ADas_Tradicionales

[2] *The good and the bad of NET framework progamming* https://www.altexsoft.com/blog/engineering/the-good-and-the-bad-of-net-framework-programming/.

[3] *Why use should I use NET* https://www.wakefly.com/blog/what-is-asp-net-and-why-should-i-use-it/.

[4] *Documentación de Spring* https://spring.io/projects/spring-framework

[5] *Documentación de Django* https://docs.djangoproject.com/en/4.0/

[6] *The good and the bad of Angular* https://www.altexsoft.com/blog/engineering/the-good-and-the-bad-of-angular-development/

[7] *Documentación de React* https://es.reactjs.org

[8] *Acerca de Trello* https://trello.com/es/about

[9] *Página oficial de Trello* https://trello.com

[10] *System and Software Quality Requirements and Evaluation* https://iso25000.com/index.php/en/iso-25000-standards/iso-25010.

[11] *Modelo de desarrollo iterativo* https://estudyando.com/modelo-de-desarrollo-iterativo-para-software/

[12] *QGIS* https://www.qgis.org/en/site/

[13] *Gdal Documentation* https://gdal.org https://pypi.org/project/GDAL/

[14] *Conceptos de bases de datos* https://support.microsoft.com/es-es/office/conceptos-básicos-sobre-bases-de-datos-a849ac16-07c7-4a31-9948-3c8c94a7c204

[15] *PostgreSQL* https://www.postgresql.org

[16] *Amazon AWS* https://aws.amazon.com/es/

[17] *Estándar IEEE830* https://www.ctr.unican.es/asignaturas/is1/ieee830_esp.pdf

[18] *SQL vs NoSQL* https://circleci.com/blog/SQL-vs-NoSQL-databases/?utm_source=google&utm_medium=sem&utm_campaign=sem-google-dg--emea-en-dsa-maxConv-auth-brand&utm_term=g_-_c__dsa_&utm_content=&gclid=CjwKCAjwlcaRBhBYEiwAK341jeBKGJuWYSWfRc7RilSq-ymjSHaM6Iq0FzhOLKXjcpRHWMOp-NqA9xoCHl4QAvD_BwE

[19] *NewSQL* https://www.techopedia.com/definition/29093/newsql

[20] *Figma* https://www.figma.com

[21] *Django Rest Framework Documentation* https://www.django-rest-framework.org/api-guide/testing/

[22] *Metodologías de desarrollo* https://www.universitatcarlemany.com/actualidad/metodologias-de-desarrollo-de-software

[23] *Metodologías Tradicionales Vs Ágiles* https://tecnitium.com/metodologias-de-desarrollo-de-software#:~:text=Dentro%20de%20las%20metodolog%C3%ADas%20tradicionales%20encontramos%20las%20siguientes%3A,mezcla%20entre%20cascada%20y%20prototipado%29%20Mas%20cosas...%20

[24] *Prototipado Iterativo* http://www.iterando.com.ar/2010/02/prototipado-iterativo.html

[25] *¿Cómo documentar correctamente mi API?* https://www.itdo.com/blog/como-documentar-correctamente-mi-api/

APÉNDICE A
Anexo A: Mockups

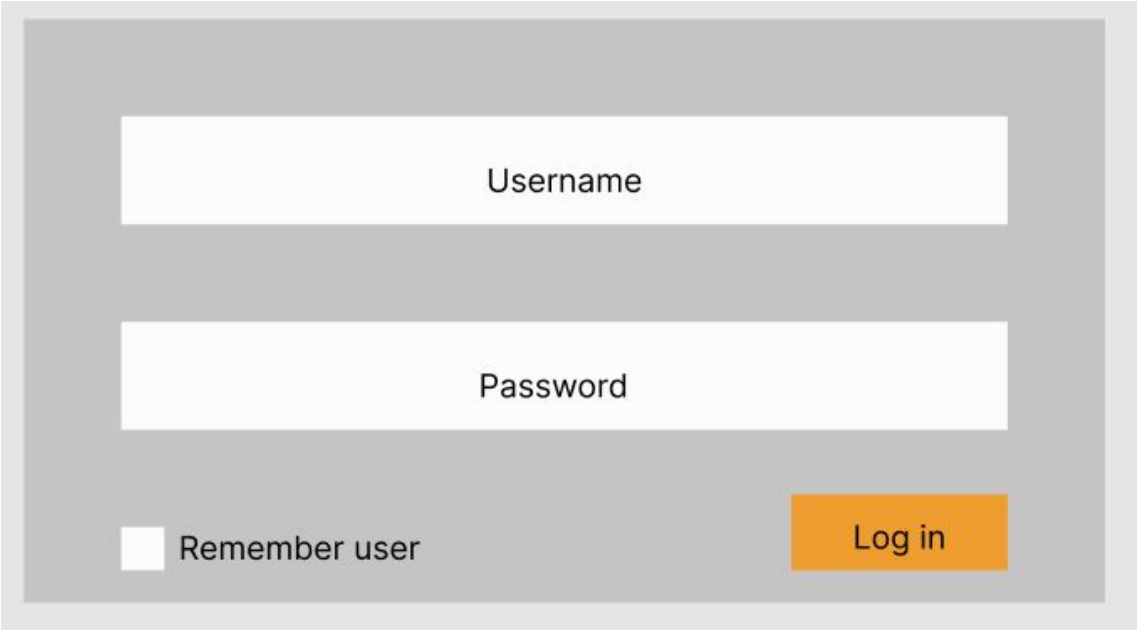

Figura A.1: Diseño del inicio de sesión de la herramienta

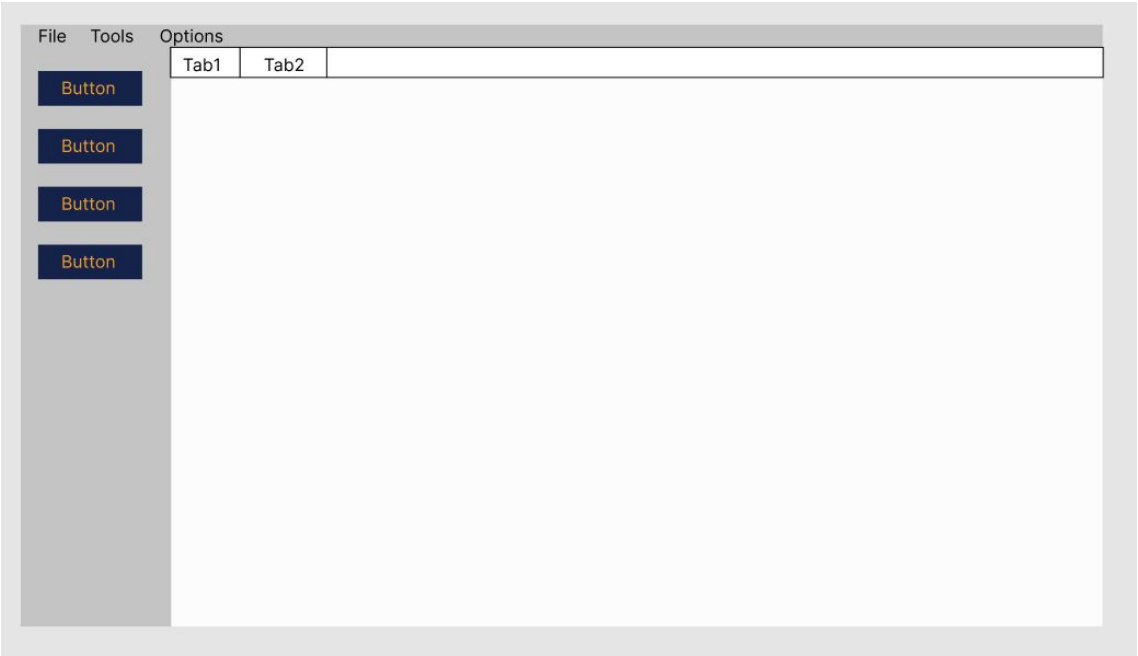

Figura A.2: Diseño del Dashboard de la aplicación

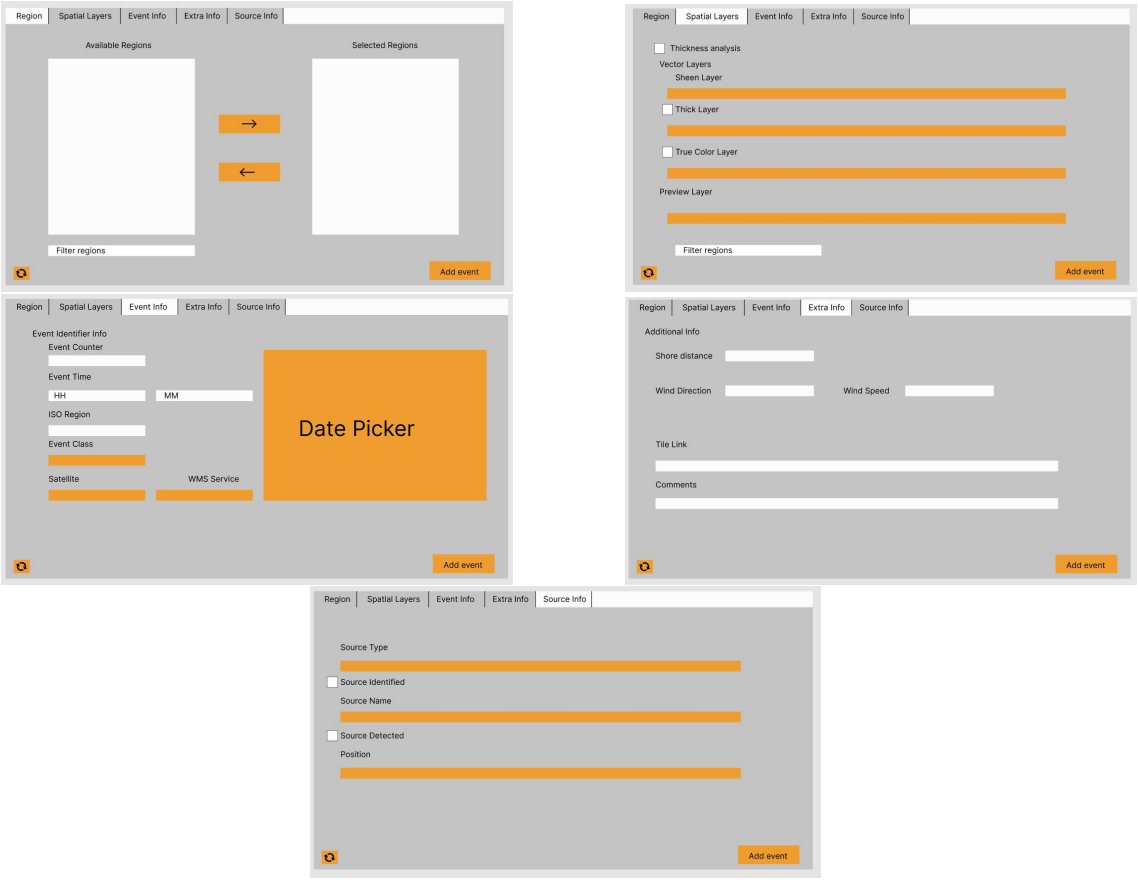

Figura A.3: Mockup en Figma de la herramienta Add Event

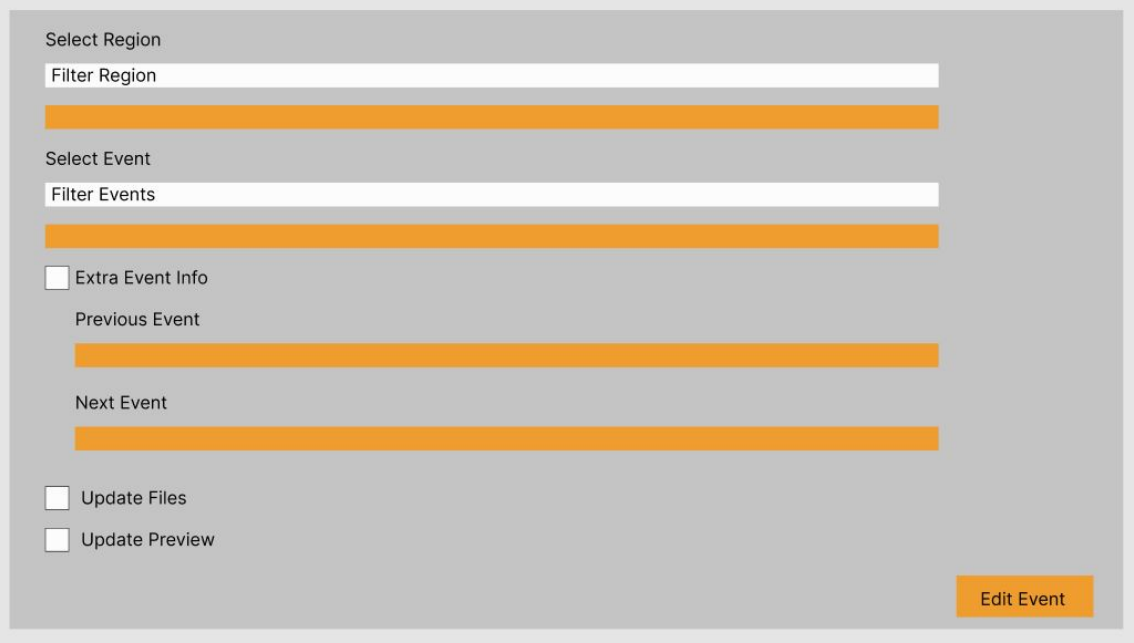

Figura A.4: Diseño en Figma de la herramienta Edit Event

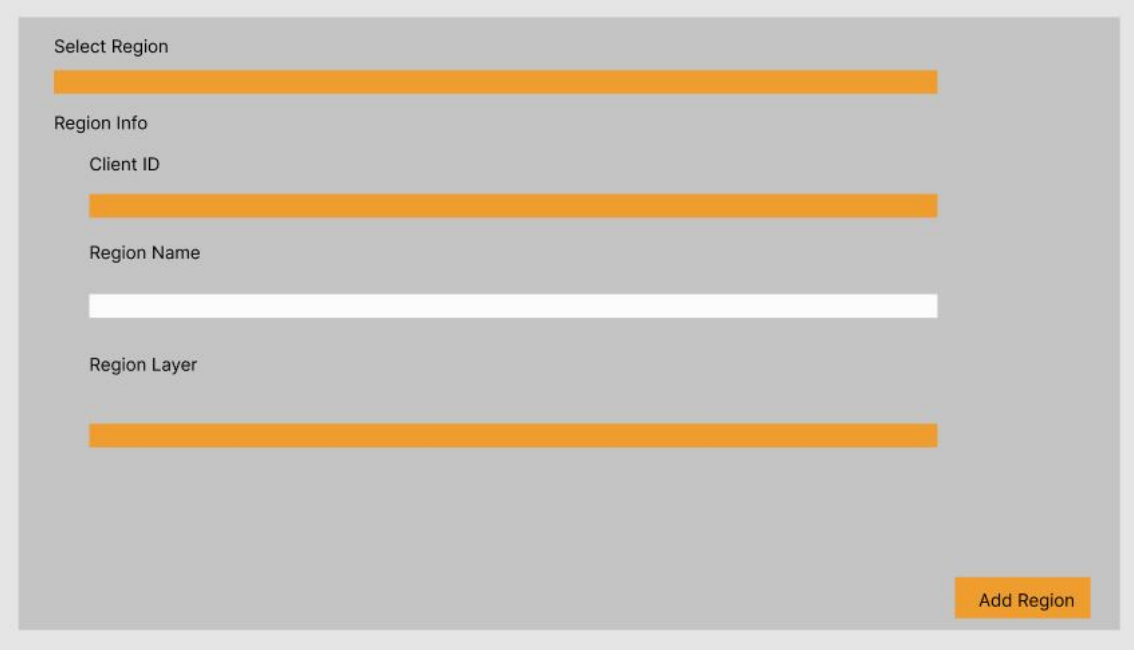

Figura A.5: Diseño de la herramienta Add Region

Figura A.6: Diseño de la herramienta Add Source

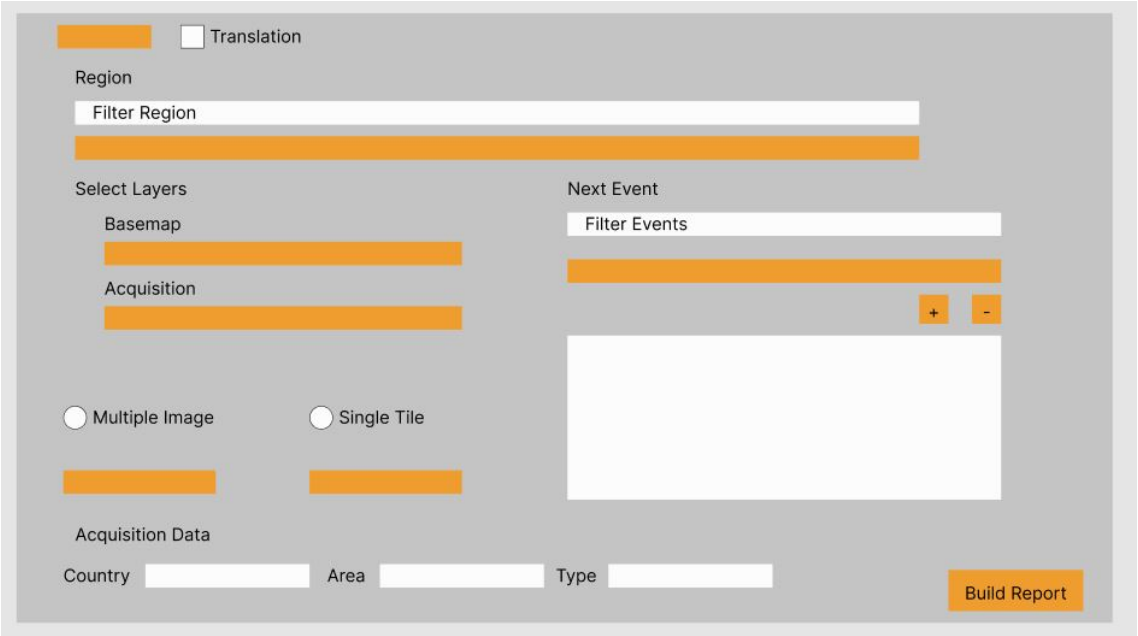

Figura A.7: Diseño de la herramienta Build Report

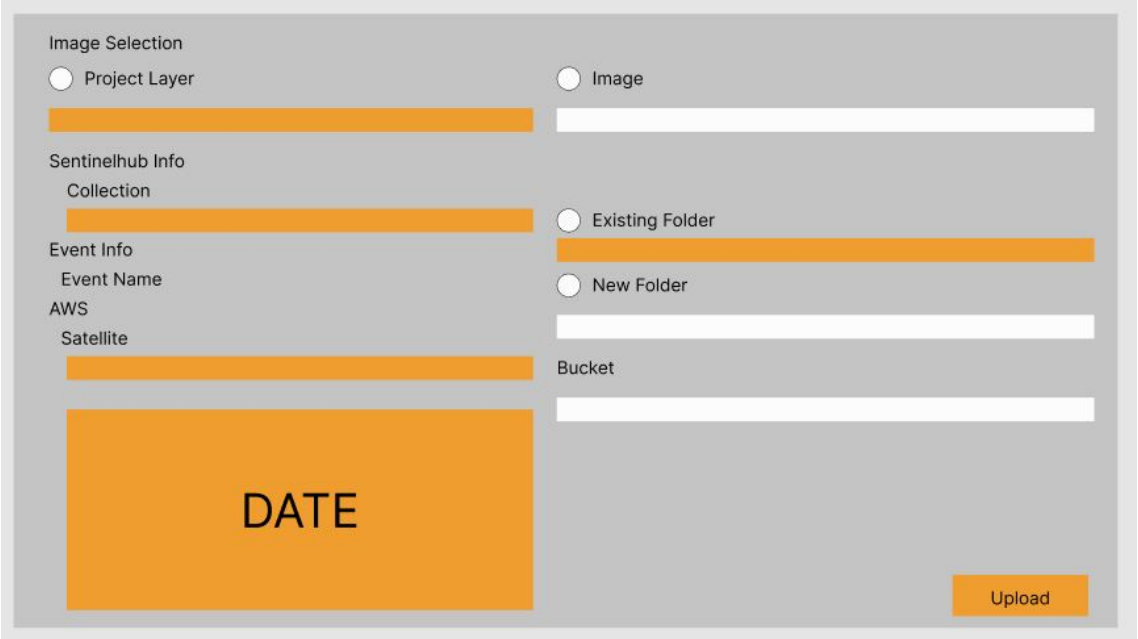

Figura A.8: Diseño de la herramienta Ingest Image

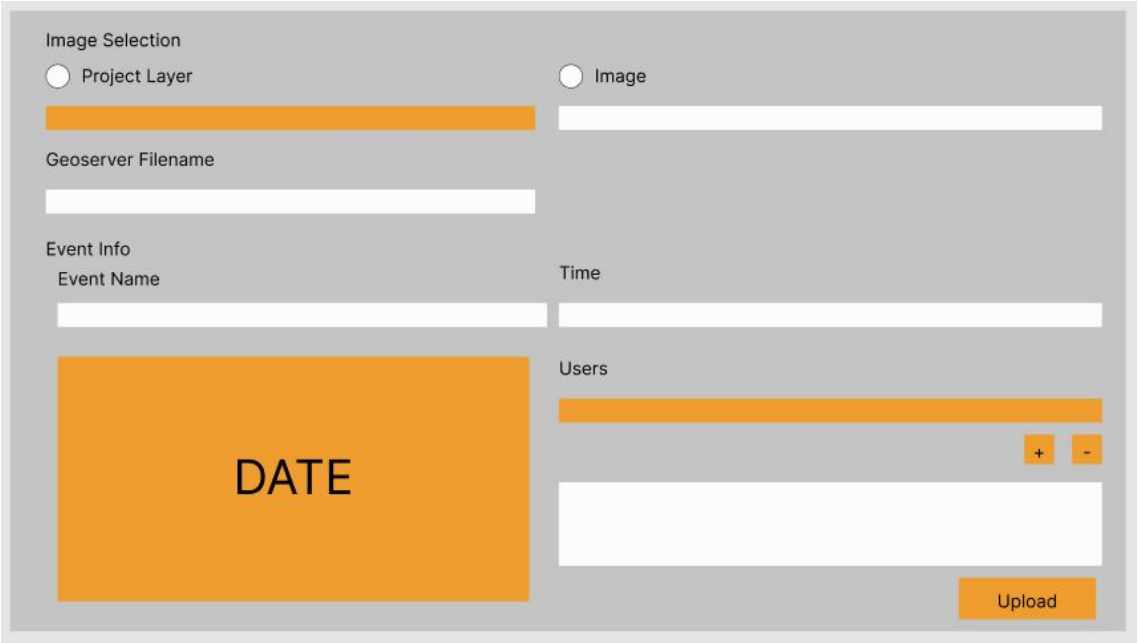

Figura A.9: Diseño de la herramienta Ingest Geo Image

Figura A.10: Diseño de la herramienta Merge Vector

Figura A.11: Diseño de la herramienta Patch Table

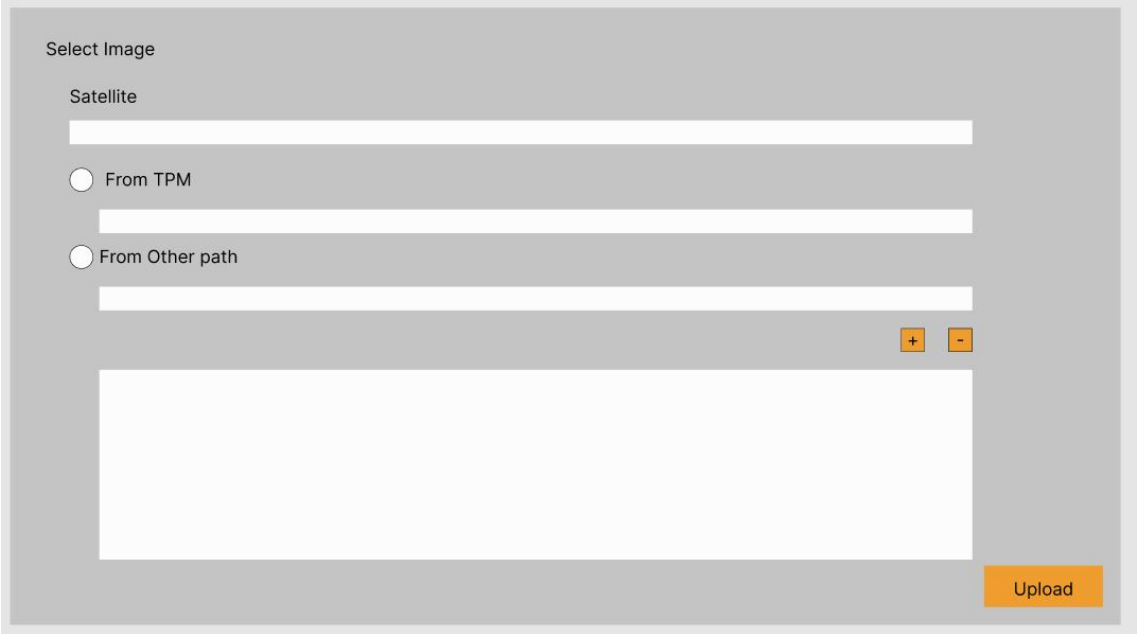

Figura A.12: Diseño de la herramienta Upload Patches

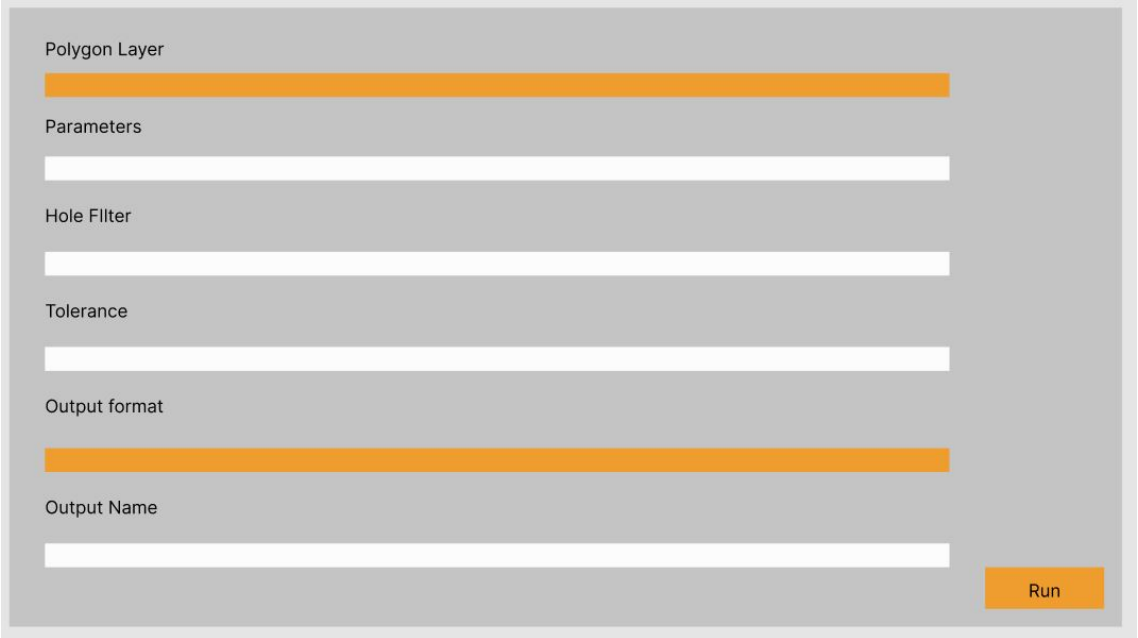

Figura A.13: Diseño de la herramienta Process Polygon

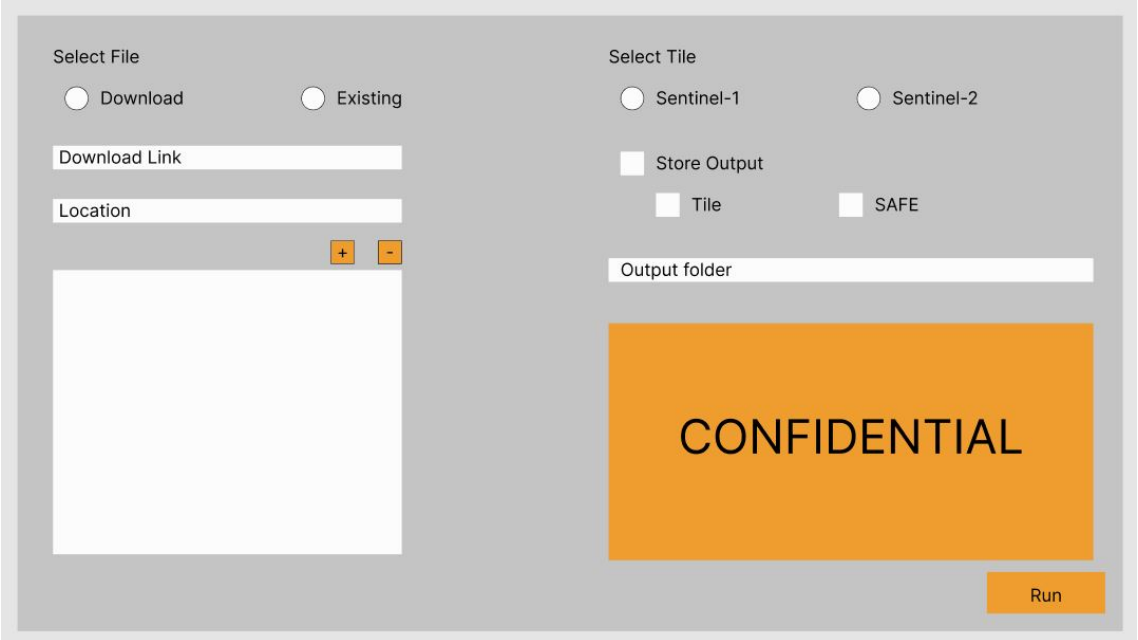

Figura A.14: Diseño de la herramienta Sentinnel Preprocess

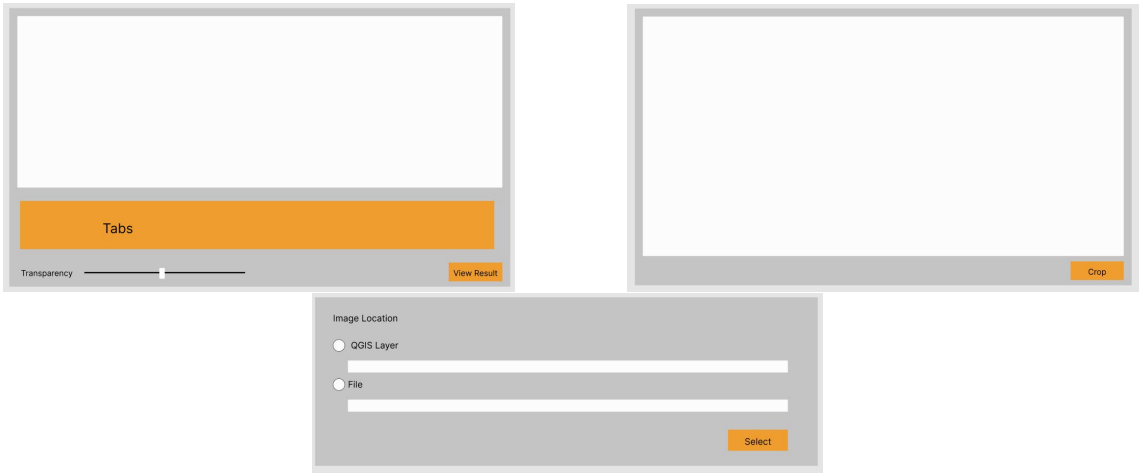

Figura A.15: Diseño de las diferentes ventanas de la herramienta Extract Spill

Figura A.16: Diferentes pestñas de la pantalla principal de la herramienta Extract Spill A.15

APÉNDICE B
Anexo B: Objetivos de Desarrollo Sostenible

Grado de relación del trabajo con los Objetivos de Desarrollo Sostenible (ODS).

Objetivos de Desarrollo Sostenibles	**Alto**	**Medio**	**Bajo**	**No procede**
ODS 1. Fin de la pobreza				X
ODS 2. Hambre cero				X
ODS 3. Salud y bienestar		X		
ODS 4. Educación de calidad				X
ODS 5. Igualdad de género				X
ODS 6. Agua limpia y saneamiento	X			
ODS 7. Energía asequible y no contaminante		X		
ODS 8. Trabajo decente y crecimiento económico				X
ODS 9. Industria, innovación e infraestructuras		X		
ODS 10. Reducción de las desigualdades				X
ODS 11. Ciudades y comunidades sostenibles				X
ODS 12. Producción y consumo responsables	X			
ODS 13. Acción por el clima	X			
ODS 14. Vida submarina	X			
ODS 15. Vida de ecosistemas terrestres			X	
ODS 16. Paz, justicia e instituciones sólidas				X
ODS 17. Alianzas para lograr objetivos				X

Reflexión sobre la relación del TFG/TFM con los ODS y con el/los ODS más relacionados.

El trabajo de final de grado aquí tratado trata de forma directa e indirecta una gran variedad de ODS presentes en la lista superior, dado que el objetivo de la empresa consiste en analizar y localizar posibles contaminantes situados en la superficie marítima. Gracias a esto, podremos responder a grandes preguntas relacionadas con la contaminación marítima como pueden ser: ¿Dónde hay un mayor volumen de contaminación marítima?, ¿de qué zonas proviene la mayoría de la contaminación marítima?, ... Además, esta herramienta puede resultar realmente útil para atajar dicho problema.

Una vez conocido el objetivo principal de la empresa, se puede ver bastante claro a que *ODS* afecta la aplicación tratada en este TFG, ya que esta herramienta será utilizada por diversos analistas con dichos fines. Es claro que la función principal de este proyecto es facilitar los procesos requeridos para realizar un mejor análisis de la contaminación en

el mar y tratar de servir como herramienta de apoyo a la hora de intentar erradicar este problema.

Claramente se observa la relación con los *ODS* de Agua limpia y saneamiento, Acción por el clima y Vida submarina, debido a la temática del proyecto, pero de forma secundaría, se obtiene también una Producción y consumo responsables al ser capaces de identificar las fuentes de contaminación y poder facilitar el control de dichas acciones contaminantes frente a las autoridades. Por otro lado la mejora de la situación sanitaria marítima afecta en cierta medida a la situación de la Vida de ecosistemas terrestres ya que tarde o temprano estos contaminantes podrían acabar afectando a la misma. Si un ecosistema marino se ve perjudicado por la contaminación, desencadenaría un deterioro de los ecosistemas terrestres de su alrededor, provocando cambios notables en términos de los ecosistemas como: cambios en las cadenas alimenticias, extinción de especies tanto animales como vegetales, migración de fauna o deterioro de la vegetación, entre otras muchas cosas.

Finalmente encontramos la relación con Energía asequible y no contaminante tomando medidas en el aspecto contaminante de la producción energética, atajando este problema de raíz y, además, Industria e innovación de infraestructuras se presenta al hacer uso de la tecnología satelital y el análisis geo-espacial con los objetivos tratados en este TFG, los cuales son indispensables en el proyecto realizado y permiten una solución simple y elegante del problema que se desea erradicar.

APÉNDICE C

Anexo C: Información extra y definiciones

C.1 Palabras Clave

En esta sección del apéndice se dan las definiciones de los conceptos utilizados a lo largo del documento.

Python: Lenguaje de programación multiparadigma, de alto nivel y interpretado.
C#: Lenguaje de programación multiparadigma y orientado a objetos, es descendente de C
Java: Lenguaje de programación orientado a objetos
JavaScript: Lenguaje de programación interpretado, orientado a objetos, basado en prototipos, imperativo, débilmente tipado y dinámico.
TypeScript: Superconjunto de JavaScript, que añade tipos estáticos y objetos basados en clases.
Frontend: Entorno encapsulado de una aplicación que se refiere a la parte de la misma vista por el usuario.
Backend: Entorno encapsulado de una aplicación que se refiere a la parte que trabaja tras los ojos del usuario.
Qgis: QGIS es un Sistema de Información Geográfica (SIG) de Código Abierto licenciado bajo GNU - General Public License . QGIS es un proyecto oficial de Open Source Geospatial Foundation (OSGeo). Corre sobre Linux, Unix, Mac OSX, Windows y Android y soporta numerosos formatos y funcionalidades de datos vector, datos ráster y bases de datos.
AWS: Amazon Web Services (AWS abreviado) es una colección de servicios de computación en la nube pública (también llamados servicios web) que en conjunto forman una plataforma de computación en la nube, ofrecidas a través de Internet por Amazon.com.
XML: eXtensible Markup Language, o 'Lenguaje de Marcado Extensible', es un metalenguaje que permite definir lenguajes de marcas utilizado para almacenar datos de forma legible.
Layout: Disposición de elementos
Widget: Un widget es una pequeña aplicación o programa diseñada para facilitar el acceso a las funciones más usadas de un dispositivo.
tiff: los archivos TIFF o Tag Image File Format son utilizados para almacenar información de imágenes en alta calidad para evitar perder dicha información.
Adquisición: Capa vectorial tipo polígono que representa la cobertura del satélite.
Capa vectorial: Capa generada en QGIS que conforma la información de un vector

Framework: Entorno o marco de trabajo que representa a un conjunto estandarizado de conceptos, prácticas y criterios para enfocar un tipo de problemática particular.
API: Interfaz de programación de aplicaciones(API) es un conjunto de definiciones y protocolos usados para diseñar e integrar el software de las aplicaciones.
REST: Representational State Transfer(REST), o transferencia de representación de estado es un estilo de arquitectura de software.
Plugin: Programa complementario a una aplicación que pretende ampliar las funcionalidades de la misma.
MVC: Modelo vista controlador es un patrón de arquitectura de software que separa los datos y la lógica de negocio de una aplicación.
MVT: Modelo vista plantilla es un patrón de arquitectura de software similar al MVC donde este interactua con modelos y plantillas al mismo tiempo con un objetivo similar al MVC.
Store: En el contexto de Angular, una store hace referencia a un almacén de estados en tiempo de ejecución.
IDE: Entorno de desarrollo.
Bitbucket: Servicio de alojamiento basado en web para proyectos que utilizan sistemas de control de versiones.
PgAdmin: Herramienta de desarrollo utilizada para interactuar de forma directa mediante el uso de una interfaz gráfica con las bases de datos de PostgreSQL.
UML: Lenguaje unificado de marcado, es un lenguaje de modelado visual común. Semántica y sintácticamente rico para la arquitectura y el diseño.
UI: UI hace referencia a interfaz de usuario, cuando un archivo tiene esa extensión implica que conforma una interfaz gráfica.
Debug: Proceso de encontrar y eliminar los errores que pueden cometer softwares y hardwares.

C.2 Widgets

En esta sección del apéndice se encuentran explicados los *widgets* utilizados en el desarrollo del frontend de la aplicación.

QLineEdit: Entrada de texto configurable.
QCheckbox: Elemento clicable de valor booleano al cual se le puede añadir texto.
QPushButton: Botón común.
QScrollArea: Contenedor de widgets el cual permite añadir propiedades de Scroll vertical y horizontal.
QMidiArea: Zona contenedora de widgets, esta permite el desplazamiento de dichos widgets como ventanas o como pestañas.
QMenuBar: Menú de herramientas clásico de herramientas de escritorio, permite añadir submenús personalizados.
QLabel: Widget para mostrar texto.
QSpinBox: Widget que permite la entrada de valores numéricos enteros.
QDoubleSpinBox: Widget que permite la entrada de valores numéricos enteros y decimales.
QRadioButton: Botón selector exclusivo, solo permite tener uno activo al mismo tiempo.
QgsMapLayerComboBox: Widget específico de la herramienta QGIS utilizado para introducir capas de la misma como entrada.
QgsExternalResourceWidget: Widget específico de QGIS que permite la entrada de elementos y archivos locales.
QCalendarWidget: Widget selector de fecha.

QTimeEdit: Widget selector de tiempo.