Universidad Zaragoza
1542

Trabajo Fin de Grado

Biblioteca y API web para la manipulación de datos geográficos sobre una malla global discreta

Autor/es

Javier Martínez Fernández

Director/es

Rubén Béjar Hernández

Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza

2020

Tabla de contenido

Resumen

El objetivo de este TFG ha sido crear una biblioteca para la importación, transformación, almacenamiento y recuperación de datos geográficos basados en un DGGS (Sistemas de Mallas Globales Discretas), una aproximación novedosa y versátil al modelizado e integración de datos geográficos, basada en jerarquías de celdas multi-resolución. Por su novedad, hay poco software disponible para crear y manipular datos con este tipo de modelos.

Se ha comenzado por estudiar alternativas de almacenamiento persistente (BD), seleccionando finalmente una base de datos NoSQL orientada a documentos, en concreto, MongoDB, tras comprobar las ventajas que ofrece frente a otras alternativas. Se ha continuado por diseñar e implementar una biblioteca Python que permite importar, manipular, recuperar y dar persistencia a datos sobre los modelos diseñados y en la BD elegida. Permite además la transformación de datos basados en un DGGS hacia formatos más comunes de información geográfica como el modelo vectorial o ráster, y viceversa.

Se ha diseñado e implementado una API web, desarrollando un servidor web con el framework Django REST, que permite la integración de la funcionalidad principal en aplicaciones web o móviles, de modo que un cliente pueda importar, recuperar o manipular datos basados en el modelo implementado.

Se ha colaborado con investigadores del grupo IAAA del I3A (Instituto Universitario de Investigación de Ingeniería de Aragón) y GEOT del IUCA (Instituto Universitario de Ciencias Ambientales) que están trabajando en un proyecto que persigue desarrollar aplicaciones para este tipo de sistemas. Esto ha permitido trabajar con necesidades, requisitos y datos reales. Además, estos investigadores han creado un cliente web para la captura de datos sobre DGGS, que utiliza la API web y el almacenamiento en BD diseñados e implementados en este TFG, lo que ha ayudado en su diseño y pruebas. Esta colaboración se ha traducido en un artículo conjunto enviado al congreso internacional GEOProcessing 2020 (pendiente de revisión).

Tras la realización del trabajo, se ha visto que una de las líneas de trabajo futuro podría ser realizar una implementación más eficiente en relación con el almacenamiento de Cell Datasets, “almacenamiento ráster" de los DGGS, así como la incorporación de más formatos hacia los que realizar transformaciones de datos basados en un DGGS. También, otra de las líneas de trabajo fututo claras, sería la integración de este sistema en workflows de procesamiento de datos geográficos.

1. Introducción

Los DGGS (Sistemas de Mallas Globales Discretas) son una aproximación novedosa y versátil al modelizado e integración de datos geográficos, basada en jerarquías de celdas multi-resolución. Por su novedad, hay poco software disponible para crear y manipular datos con este tipo de modelos. El objetivo de este TFG ha sido crear una biblioteca para la importación, transformación, almacenamiento y recuperación de datos geográficos basados en un DGGS. Se han estudiado alternativas de almacenamiento persistente (BD) y también se ha implementado una API web que permite la integración de la funcionalidad principal en aplicaciones web o móviles.

Se ha colaborado con investigadores (grupo IAAA del I3A y GEOT del IUCA) que están trabajando en un proyecto que persigue desarrollar aplicaciones para este tipo de sistemas. Esto ha permitido trabajar con necesidades, requisitos y datos reales. Además, estos investigadores han creado un cliente web para la captura de datos sobre DGGS, que utiliza la API web y el almacenamiento en BD diseñados e implementados en este TFG, lo que ha ayudado en su diseño y pruebas.

El resto del documento se estructura de la siguiente manera: primero se sitúa este trabajo en su contexto, se sigue con el análisis del problema, explicando los requisitos de este; se continúa con el diseño de la solución, detallando su arquitectura, la implementación y las pruebas realizadas; se explica la gestión del proyecto y las conclusiones y trabajo futuro en relación con el mismo.

1.1 Contexto

Un DGGS (Discrete Global Grid System) es una secuencia de cuadrículas globales discretas, generalmente de resolución cada vez más fina. Una cuadrícula global discreta es una partición finita de la superficie de un elipsoide, junto con un conjunto de puntos distinguidos, un punto en cada elemento de partición. Un elemento de partición se llama celda y su punto asociado único se llama núcleo [1].

Debido a que hay varios DGGS definidos y no hay uniformidad en los criterios sobre lo que es y no es un DGGS, en 2017, el consorcio de estandarización OGC (Open Geospatial Consortium) propuso una especificación abstracta para establecer los requisitos que deben cumplir [2].

El objetivo de un DGGS es mejorar la manera en la que se georreferencian datos geográficos sin tener que hacer uso de sistemas de coordenadas proyectadas. Son una aproximación novedosa y versátil basada en jerarquías de celdas multi-resolución. Debido a su novedad, hay poco software disponible para crear y manipular datos con este tipo de modelos.

Los DGGS facilitan la integración de datos geográficos creados en distintas condiciones, por ejemplo, distintas escalas, bajo distintos sistemas de referencia de coordenadas y proyecciones cartográficas, etc. Permiten, por ejemplo, la armonización de datos ráster, vectoriales y de nube de puntos en un marco común y coherente, lo que permite superar algunos desafíos clave

presentados por los enfoques SIG tradicionales. El estándar OGC DGGS Abstract Specification define el modelo conceptual y un conjunto de reglas para construir arquitecturas altamente eficientes para el almacenamiento, integración y análisis de datos espaciales.

Los DGGS representan la Tierra como secuencias jerárquicas de teselaciones de igual área en la superficie de la Tierra, cada una con cobertura global y con una resolución espacial progresivamente más fina. Las observaciones individuales pueden asignarse a una celda que corresponda tanto a la posición como al tamaño del fenómeno que se está observando. Los DGGS cuentan con un conjunto de algoritmos funcionales que permiten un rápido análisis de datos de un gran número de celdas y, por su propia naturaleza, son muy adecuados para aplicaciones de procesamiento en paralelo con múltiples resoluciones espaciales.

La ventaja de los Sistemas de Mallas Globales y Discretas es que permiten fácilmente definir un procedimiento de identificadores únicos para cada celda, y que se encuentra asociado a un conjunto de algoritmos que facilitan el análisis espacial eficiente de un enorme número de celdas, estando especialmente dispuestos para ser paralelizados. Por ello, se podía considerar a los DGGS como el sistema más adecuado si se habla, por ejemplo, de Big Data geográfico, ya que permite una exploración, extracción y visualización rápidas y precisas de los datos. DGGS podría representar el cambio de paradigma que permitiría superar algunas de las barreras críticas que impiden alcanzar el verdadero potencial que Big Data puede ofrecer en el ámbito geográfico.[0]

Un DGGS utiliza poliedros sólidos (Figura 1), por ejemplo, tetraedros, cubos, octaedros, para modelar la Tierra, y estas teselaciones se proyectan inversamente para crear el sistema de referencia.

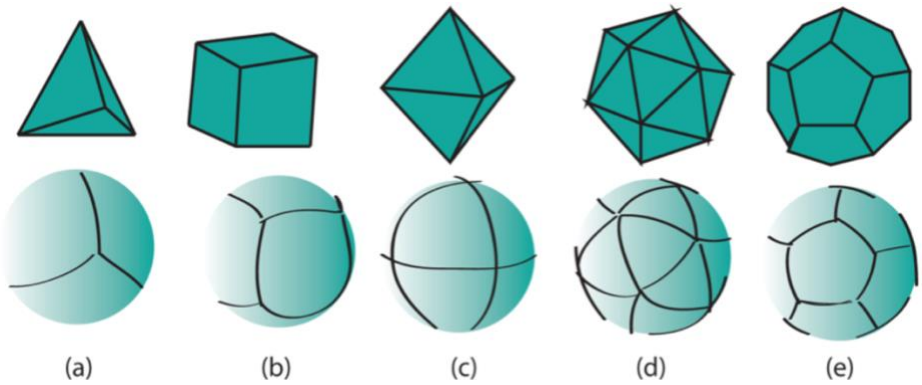

FIGURA 1 – POLIEDROS UTILIZADOS EN UN DGGS

Una teselación de celdas debe cumplir con un conjunto de criterios para ser considerado como DGGS según la especificación de OGC, algunos de ellos son:[1]

[0] https://www.ogc.org/pressroom/pressreleases/2656
[1] https://www.geoawesomeness.com/discrete-global-grid-system-dggs-new-reference-system/

- La teselación debe cubrir toda la Tierra, aunque los datos referidos a ella pueden cubrir solo una parte.
- Las celdas no deben superponerse.
- Se debe declarar el método de refinamiento de las celdas.
- En cualquier nivel de refinamiento, las celdas deben ser de igual área. Sin embargo, se pueden permitir pequeñas desviaciones de la igualdad exacta del área siempre que se declare la precisión.
- En cada nivel de refinamiento sucesivo, el área total de las celdas hijas debe ser igual al área total de las celdas madres.
- Las celdas deben tener un sistema de referencia sistemático.

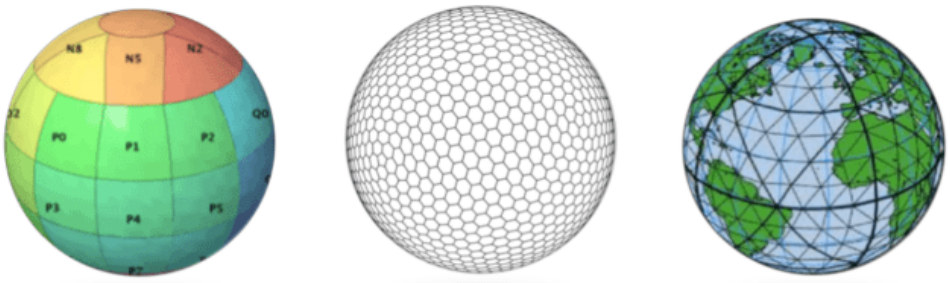

FIGURA 2 - EJEMPLOS DE DGGS BASADOS EN EL MAPEO DE LAS CARAS DE LOS SÓLIDOS PLATÓNICOS

En este TFG se ha implementado software para la manipulación de distintos DGGS que hayan sido definidos de acuerdo con la especificación de OGC. Este software permite importar, almacenar, recuperar y transformar datos geográficos basados en un DGGS, e integrar esta funcionalidad en aplicaciones web o móviles.

Debido a su novedad, es importante poder transformar datos basados en el modelo DGGS hacia formatos más comunes de información geográfica como el modelo vectorial o ráster, y viceversa.

Un modelo de datos geográfico es una forma de reducir las propiedades de la realidad geográfica a un conjunto finito de elementos que se puedan manipular. Se distinguen dos tipos principales de modelos de datos geográficos: ráster y vectorial. [3]

El modelo ráster (Figura 3) se basa en una división sistemática del espacio que lo cubre por completo (teselación) en unidades elementales (celdas) que tienen valores asociados. Es utilizado típicamente para variables continuas que toman valores en todo el espacio de trabajo (altura sobre el nivel del mar, temperatura, etc.)

En el modelo vectorial (Figura 3) no se cubre todo el espacio, solo unas partes delimitadas por elementos geométricos con valores asociados. La disposición de estos elementos geométricos no es sistemática, y depende de los objetos geográficos en la zona de estudio. Es utilizado típicamente para elementos discretos de la realidad (carreteras, ciudades, edificios, lagos etc.)

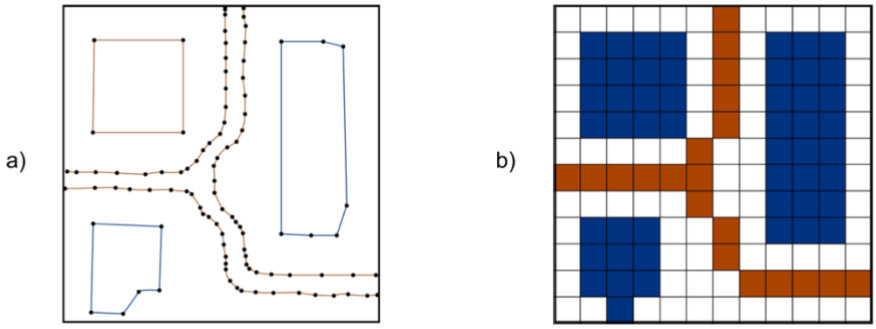

FIGURA 3 - COMPARACIÓN ENTRE LOS ESQUEMAS DEL MODELO DE REPRESENTACIÓN VECTORIAL (A) Y RÁSTER (B)

Para transformar datos basados en un DGGS hacia un modelo vectorial o ráster, y viceversa, es necesario realizar una reproyección, es decir, convertir las coordenadas sobre una proyección a coordenadas sobre otra. Una proyección cartográfica permite transformar las coordenadas sobre la superficie curva de la Tierra en coordenadas sobre una superficie plana.

En concreto, en este TFG, se realizan conversiones de coordenadas sobre la proyección rHEALPix, explicada en la sección 2, a coordenadas sobre una proyección geodésica, es decir, utilizando un sistema de coordenadas donde la posición de un punto se define usando los elementos latitud, longitud, en concreto, sobre el elipsoide WGS84.

Durante el desarrollo de este trabajo, además, se ha colaborado con investigadores (grupo IAAA del I3A y GEOT del Dpto. de Geografía y Ord. del Territorio) que están trabajando en un proyecto, COLABOTUR[2], que persigue desarrollar aplicaciones para este tipo de sistemas. Estos investigadores han creado un cliente web para la captura de datos sobre DGGS, que utiliza la API web y el almacenamiento en BD diseñados e implementados en este TFG, lo que ha ayudado en su diseño y pruebas.

[2] https://www.iaaa.es/colabotur

2. Análisis del problema

El objetivo de este TFG es diseñar e implementar una biblioteca que permita importar, manipular, recuperar y dar persistencia a datos geográficos basados en un DGGS sobre los modelos diseñados y en la BD elegida.

Se van a estudiar alternativas para la persistencia de los datos (BD) y seleccionar una adecuada. También se va a diseñar e implementar una API web que permita la integración de la funcionalidad principal en aplicaciones web o móviles, en concreto, que dé el soporte de back-end al cliente web que está siendo desarrollado por los otros investigadores.

El diseño del sistema se debe basar en un modelo conceptual existente. A continuación, se presenta dicho modelo (Figura 5) y se realiza un análisis de los conceptos más relevantes y a partir de los cuales se ha realizado este trabajo.

- **rHEALPix**: DGGS basado en el DGGS HEALPix, que inicialmente se definió solo para esferas, que se puede utilizar en elipsoides de revolución como el elipsoide WGS84. El DGGS rHEALPix puede considerarse como un mapeo de un elipsoide de revolución en un poliedro regular, es decir, un cubo (Figura 4), seguido de una división jerárquica simétrica de las caras poliédricas junto con una selección de núcleos, seguido del mapeo inverso del resultado en el elipsoide (Figura 6). Por lo tanto, es un ejemplo de un DGGS geodésico cúbico. El DGHS rHEALPix y sus matemáticas asociadas han sido completamente descritas por R G Gibb. [4]

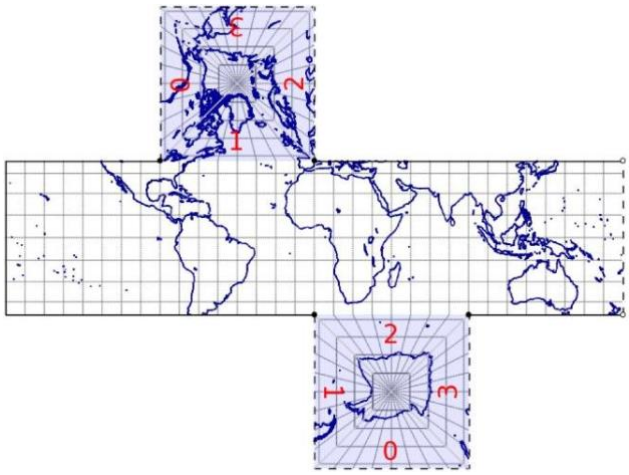

FIGURA 4 – LA PROYECCIÓN (1,3) -RHEALPIX DEL ELIPSOIDE WGS84

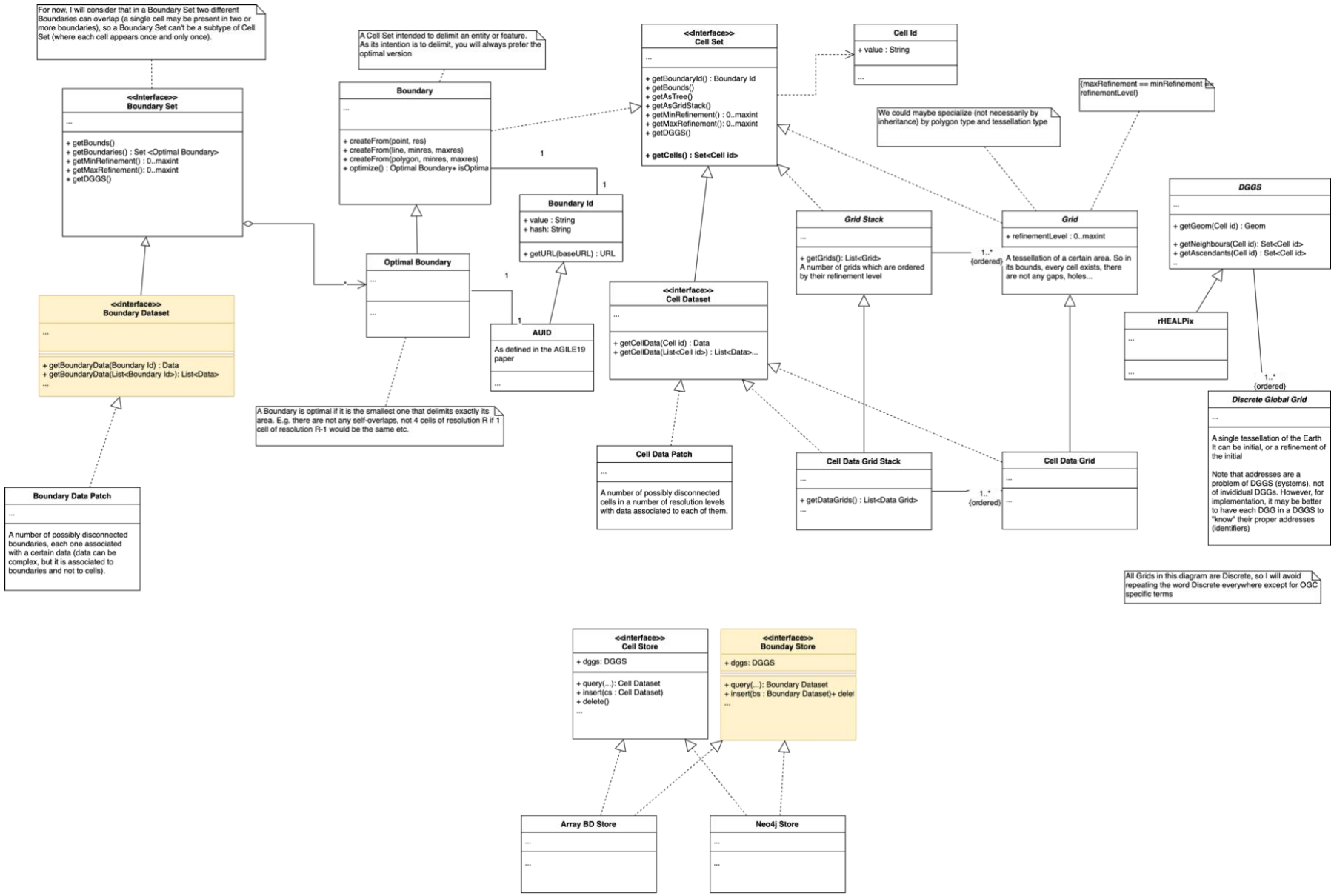

FIGURA 5 - MODELO CONCEPTUAL

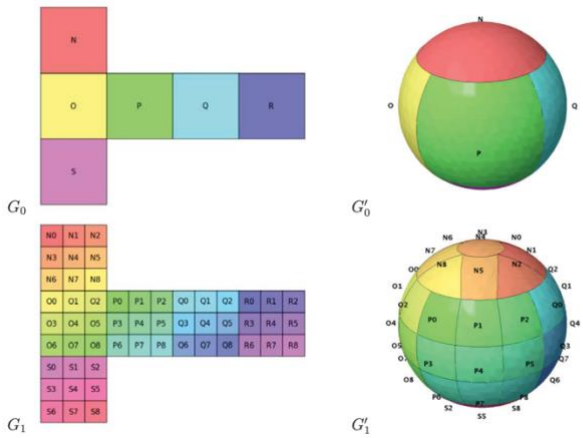

FIGURA 6 – LAS DOS PRIMERAS CUADRÍCULAS PLANAS Y ELIPSOIDALES PARA LA PROYECCIÓN DEL MAPA (0, 0) -RHEALPIX

- **Cell**: celda, objeto fundamental de un DGGS en cada nivel de refinamiento o resolución. Cada una con un identificador único asociado, el Cell Id del modelo conceptual.

- **Cell Set**: conjunto de celdas donde cada celda aparece una única vez. En concreto, conjunto de los Cell Id de las celdas que forman el conjunto. De dicho conjunto se debe poder obtener el nivel mínimo y máximo de refinamiento, el identificador del Boundary, Boundary Id, que forma el conjunto, así como otros elementos como una Grid Stack a partir del conjunto.

- **Boundary Id**: identificador único de un Boundary. Cadena formada a partir de los identificadores de las celdas que forman el Boundary.

- **Boundary**: conjunto de celdas que cubre un área en un DGGS. Es un subtipo de Cell Set. Un Boundary debe poder optimizarse, es decir, obtener el Optimal Boundary correspondiente. Como la intención de un Boundary es delimitar, siempre se va a preferir su versión óptima. También es importante la obtención del bounding box del Boundary, es decir, el área definida por dos longitudes y dos latitudes que cubre el área cubierta por un Boundary. Define los límites de este.

- **Optimal Boundary**: conjunto óptimo de celdas que cubre un área en un DGGS, entendiendo óptimo como el conjunto más pequeño de celdas que cubren exactamente esa área. Un Optimal Boundary debe tener asociado un AUID (Area Unique Identifier), identificador único de un Optimal Boundary formado a partir de los identificadores de las celdas que lo forman [5].

  Para optimizar un Boundary, se recorren cada uno de los niveles de refinamiento existentes en el conjunto buscando un conjunto de celdas que cubran la misma área que una celda de un nivel de refinamiento menor, es decir, que, en un nivel de refinamiento, existan todas las celdas hijas de una celda padre del nivel anterior, de forma que ese

conjunto de celdas pueda ser sustituido por la celda padre (Figura 7). Por ejemplo, las celdas P20, P21, P22, P23, P24, P25, P26, P27 y P28 cubren exactamente la misma área que la celda P2, que es la celda padre de todas ellas, por lo que pueden ser sustituidas por una única celda, consiguiendo así el conjunto óptimo de celdas que cubren un área en un DGGS.

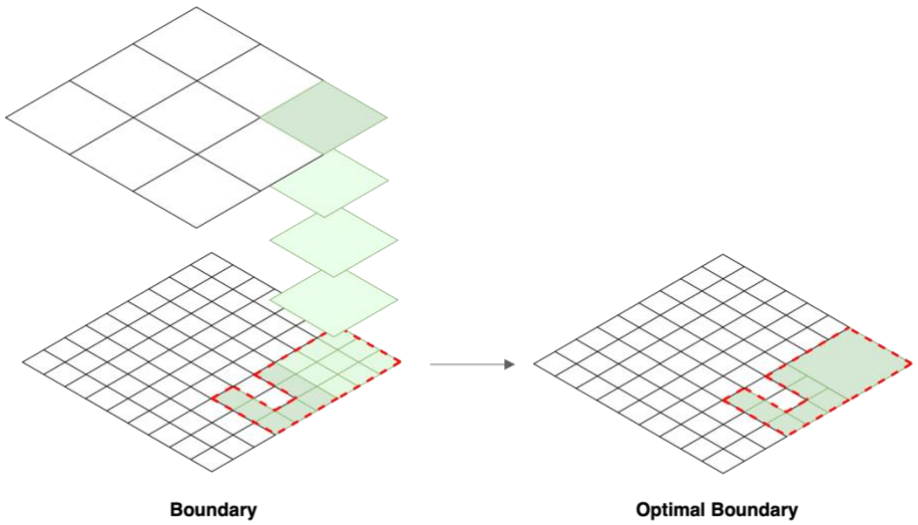

FIGURA 7 - PROCESO DE OPTIMIZACIÓN DE UN BOUNDARY

- **Grid:** teselación de una cierta área, por lo que en sus límites cada Cell existe, no hay vacíos ni agujeros. Es un subtipo de Cell Set, ya que es un conjunto de celdas. Todas las celdas de dicho conjunto se encuentran en el mismo nivel de refinamiento.

- **Grid Stack:** lista de Grids ordenadas por su nivel de refinamiento. Es otro subtipo de Cell Set, ya que es otro conjunto de celdas.

- **Cell DataSet**: conjunto de celdas, Cell Set, en el que cada una tiene asociados unos datos concretos. Se deben poder obtener los datos asociados a una celda dado su identificador, su Cell Id, o a una lista de celdas dada una lista de identificadores.

- **Boundary Set***:* conjunto de Optimal Boundaries. En dicho conjunto, dos Boundaries pueden superponerse (una celda puede estar presente en dos o más Boundaries), por lo que un Boundary Set no puede ser un subtipo de Cell Set. De dicho conjunto se debe poder obtener el nivel mínimo y máximo de refinamiento.

- **Boundary DataSet***:* conjunto de Optimal Boundaries en el que cada uno tiene asociados unos datos concretos. Se deben poder obtener los datos asociados a un Optimal Boundary dado su identificador, o a una lista de Optimal Boundaries dada una lista de identificadores.

- **Cell Store:** almacén de celdas que debe permitir importar, manipular, recuperar y dar persistencia a Cell Datasets.

- **Boundary Store:** almacén de Boundaries que debe permitir importar, manipular, recuperar y dar persistencia a Boundary Datasets.

2.1 Requisitos

2.1.1 Biblioteca

No funcionales

- RNF1: La biblioteca debe estar desarrollada en el lenguaje de programación Python.
- RNF2: El diseño del sistema se debe basar en un modelo conceptual existente.

Funcionales

- RF1: La biblioteca debe permitir representar una **celda** de un DGGS en base al identificador único de la celda.
- RF2: La biblioteca debe permitir representar un **Boundary** a partir de la lista de identificadores de las celdas que lo forman.
- RF3: La biblioteca debe permitir obtener el **Boundary Bounding Box** de un Boundary concreto.
- RF4: La biblioteca debe permitir optimizar un Boundary para obtener el **Optimal Boundary** correspondiente.
- RF5: La biblioteca debe permitir representar un **Cell/Boundary dataset.**
- RF6: La biblioteca debe dar soporte a todas las operaciones de la **API Web**.
- RF7: La biblioteca debe ofrecer una operación para la **transformación** de datos asociados a entidades de un modelo vectorial contenidos en un shapefile en datos asociados a un Boundary.
- RF8: La biblioteca debe ofrecer una operación para la **transformación** de datos asociados a un Boundary, en datos asociados a entidades de un modelo vectorial contenidos en un shapefile.
- RF9: La biblioteca debe ofrecer una operación para la **transformación** de un conjunto de datos asociados a entidades de un modelo vectorial contenidos en un conjunto de shapefiles, en un conjunto de datos asociados a Boundaries contenidos en un Boundary Dataset.
- RF10: La biblioteca debe ofrecer una operación para la **transformación** de un conjunto de datos asociados a Boundaries contenidos en un Boundary Dataset, en un conjunto de datos asociados a entidades de un modelo vectorial contenidos en un conjunto de shapefiles.

- RF11: La biblioteca debe ofrecer una operación para la **transformación** de datos asociados a píxeles de un modelo ráster contenidos en un fichero GeoTIFF, en un conjunto de datos asociados a Celdas contenidos en un Cell Dataset.
- RF12: La biblioteca debe ofrecer una operación para la **transformación** de un conjunto de datos asociados a Celdas contenidos en un Cell Dataset, en datos asociados a píxeles de un modelo ráster contenidos en un fichero GeoTIFF.
- RF13: Las operaciones de transformación deben poder utilizarse a través de una interfaz de línea de comandos (**CLI**).

2.1.2 API Web

Funcionales

- RF14: La API Web debe permitir la **inserción** y **borrado** de Cell/Boundary datasets en la BD, entendiendo un dataset como una lista de identificadores de Celdas/Boundaries asociados cada uno de ellos a datos en formato JSON e incluyendo un identificador único para dicho dataset.
- RF15: La API Web debe permitir la **inserción, modificación** y **borrado** de datos en un Cell/Boundary dataset concreto y existente en la DB, dado el identificador del dataset y una lista de identificadores de Celdas/Boundaries asociados cada uno de ellos a datos en formato JSON.
- RF16: La API Web debe permitir la **recuperación** de todos los Cell/Boundary datasets almacenados en la BD.
- RF17: La API Web debe permitir la **recuperación** de todos los datos de un Cell/Boundary dataset concreto y existente en la BD, dado el identificador del dataset.
- RF18: La API Web debe permitir la **recuperación** de todos los datos asociados a una Celda/Boundary concreto, dado el identificador de la Celda/Boundary.
- RF19: La API Web debe permitir la **recuperación** de los datos asociados a una Celda/Boundary concreto de un Cell/Boundary dataset concreto y existente en la BD, dado el identificador del dataset y el identificador de la Celda/Boundary.
- RF20: La API Web debe permitir la **recuperación** de los datos asociados a las Celdas/Boundaries que intersecten con un polígono dadas sus coordenadas.

3. Diseño de la solución

3.1 Arquitectura

En cuanto a la arquitectura del sistema, se pueden distinguir dos desarrollos principales. En primer lugar, una **biblioteca** de Python que permite importar, manipular, recuperar y dar persistencia a los datos geográficos basados en un DGGS en una base de datos NoSQL de tipo documental, MongoDB. Y, en segundo lugar, un **servidor web** basado en el framework Django REST, que da el soporte de back-end a aplicaciones web o móviles a través de su API (Figura 8). En la sección 3.6 se explican las tecnologías mencionadas.

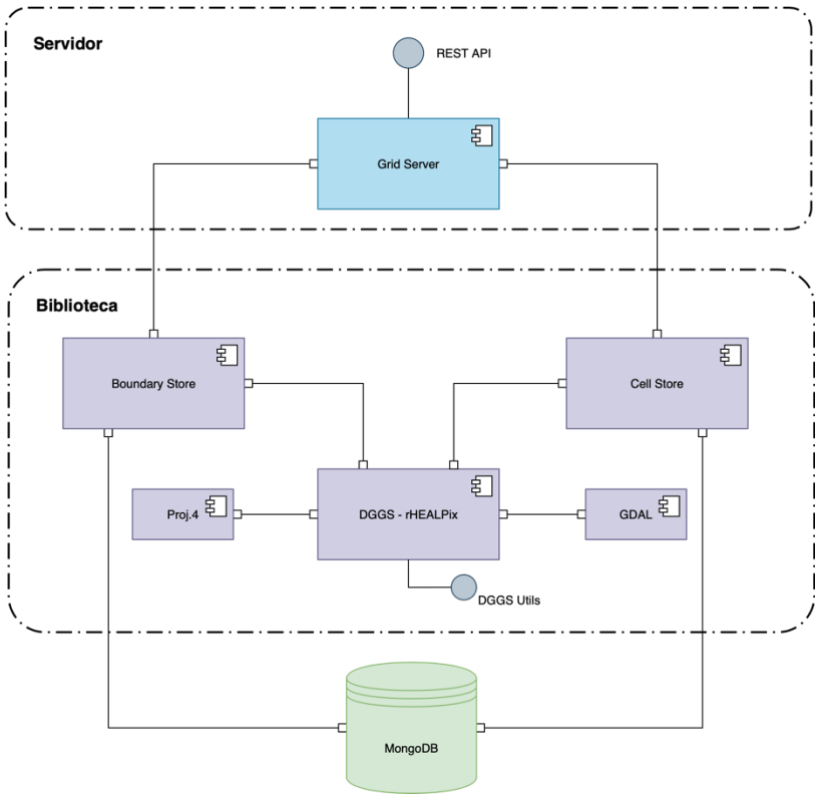

FIGURA 8 - ARQUITECTURA DEL SISTEMA

En cuanto a la **biblioteca** Python, por un lado, incluye la implementación del modelo diseñado, utilizando rHEALPix como DGGS por defecto (componente DGGS-rHEALPix). Dentro de esta implementación destacan servicios como la optimización de un Boundary, que permite obtener un conjunto óptimo de celdas que cubren un área en un DGGS, entendiendo como óptimo el conjunto más pequeño de celdas que cubren exactamente esa área; y la transformación de datos basados en el modelo DGGS hacia formatos más comunes de información geográfica como el modelo vectorial o ráster, y viceversa.

Por otro lado, la biblioteca incluye la implementación de dos almacenes, componentes Boundary Store y Cell Store, que permiten importar, manipular, recuperar y dar persistencia a los datos en una base de datos de MongoDB, apoyando las operaciones expuestas por el API. Estos almacenes hacen uso de la implementación del modelo DGGS anteriormente mencionado.

En cuanto a la **API**, se ha utilizado el framework Django REST para el desarrollo del servidor web (componente Grid Server) que un cliente puede utilizar para importar, recuperar o manipular datos basados en el modelo implementado a través de su API. El servidor hace uso de las operaciones del Boundary y Cell store incluidos en la biblioteca Python desarrollada.

3.2 Modelo de datos

A la hora de hablar del modelo de datos, existe una división derivada de los dos almacenes distintos que se han desarrollado. Por una parte, el almacén de Boundaries, o Boundary store, con el que se da persistencia a Boundary Data Sets, es decir, a conjuntos de Boundaries en los que cada Boundary tiene asociado unos datos concretos, entendiendo un Boundary como un conjunto de Celdas que delimitan un área en un DGGS. Y, por otra parte, el almacén de Celdas, o Cell store, con el que se da persistencia a Cell Data Sets, es decir, a conjuntos de Celdas en los que se asocian datos a cada una de las celdas que los forman.

Por ello, a continuación, se presenta un modelo de entidades y relaciones mediante un diagrama de clases UML, diferenciando lo relacionado con los Boundaries y lo relacionado con las celdas.

3.2.1 Modelo de entidades y relaciones

En la Figura 9 se presenta el diagrama de clases que define el modelo formado por las entidades y relaciones asociadas a un conjunto de Boundaries. A continuación, se definen cada una de las entidades, los atributos que las forman y las relaciones entre ellas.

- La entidad **Boundary**, modeliza el concepto de mismo nombre del diagrama conceptual. Esta entidad cuenta con el atributo **AUID**, que define un identificador único formado a partir de los identificadores de las celdas que forman el Boundary [5] . Cada Boundary tiene asociado un bounding box que define los límites del área en cuestión, de ahí la relación presentada.
- La entidad **BBOX** define el bounding box de un Boundary, es decir, área definida por 4 puntos, superior izquierda, superior derecha, inferior derecha e inferior izquierda, que cubre el área cubierta por un Boundary.
- La entidad **Data** define los datos asociados a un Boundary y cuenta con un atributo que define estos datos en formato JSON.

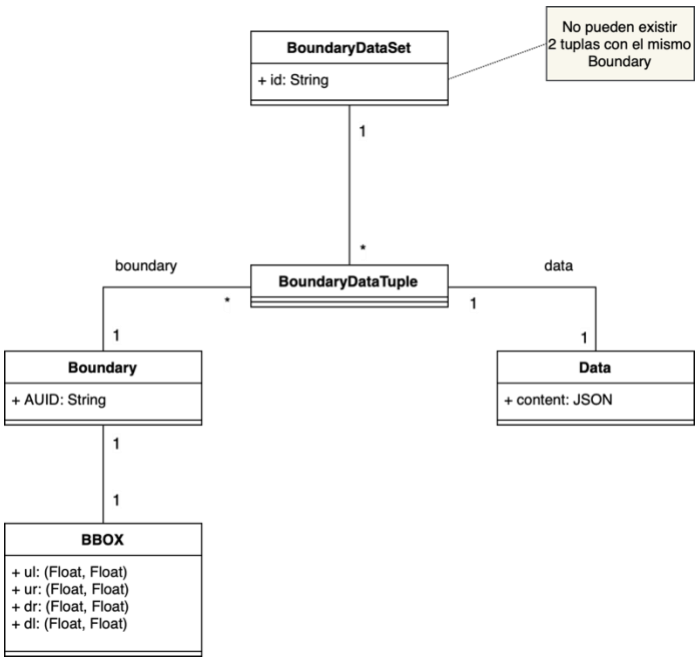

FIGURA 9 – MODELO DE ENTIDADES Y RELACIONES – BOUNDARY STORE

- La entidad **BoundaryDataSet** define un conjunto de Boundaries en el que cada Boundary tiene asociados unos datos concretos, tal y como se define el concepto de mismo nombre del diagrama conceptual. Esto se representa en el modelo mediante una agregación de tuplas Boundary-Data, representadas mediante la entidad **BoundaryDataTuple**, que define la relación entre un Boundary y sus datos. Tal y como se explica en el diagrama, un BoundaryDataSet no puede tener dos tuplas asociadas al mismo Boundary. Además, un BoundaryDataSet cuenta con el atributo **id**, que define el identificador de un conjunto, lo que permite realizar operaciones sobre un conjunto concreto dado su identificador.

En la Figura 10 se presenta el diagrama de clases que define el modelo formado por las entidades y relaciones asociadas a un conjunto de celdas. A continuación, se definen sus entidades, los atributos que las forman y las relaciones entre ellas.

- La entidad **Cell**, modeliza el concepto de mismo nombre del diagrama conceptual. Cada una de las Celdas tiene asociado un identificador único, y este está definido por el atributo **id** de esta entidad.
- La entidad **Data** define los datos asociados a una Celda y cuenta con un atributo que define estos datos en formato JSON.

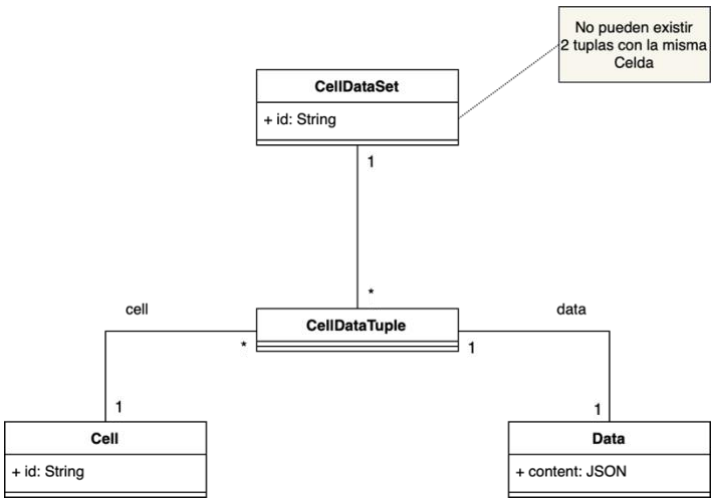

FIGURA 10 – MODELO DE ENTIDADES Y RELACIONES – CELL STORE

- La entidad **CellDataSet** define un conjunto de Celdas en el que cada Celda tiene asociados unos datos concretos, tal y como se define el concepto de mismo nombre del diagrama conceptual. Esto se representa en el modelo mediante una agregación de tuplas Cell-Data, representadas mediante la entidad **CellDataTuple**, que define la relación entre una Celda y sus datos. Tal y como se explica en el diagrama, un BoundaryDataSet no puede tener dos tuplas asociadas a la misma Celda. Además, un CellDataSet cuenta con el atributo **id**, que define el identificador de un conjunto, lo que permite realizar operaciones sobre un conjunto concreto dado su identificador.

3.2.2 Modelo de implementación

Tras el estudio de diferentes alternativas para la persistencia de los datos (BD), se ha decidido usar una base de datos NoSQL orientada a documentos, en concreto, MongoDB.

MongoDB guarda estructuras de datos BSON (una especificación similar a JSON) con un esquema dinámico. Esto implica que las entidades definidas en el modelo de datos se implementan como documentos BSON, y las relaciones entre ellas como referencias entre dichos documentos. En la sección 3.6 se explica esta tecnología, así como las alternativas probadas. A continuación, se describe cómo se implementa el modelo haciendo uso de estas estructuras de datos.

En la Figura 11 pueden observarse los dos documentos que forman la implementación de un Boundary store. Por una parte, el **BoundaryDataSet document**, que representa el conjunto de Boundaries junto con los datos asociados. Está formado únicamente por el campo **_id**, un String que define el identificador del conjunto. Este identificador es usado en el documento de un Boundary para hacer referencia al conjunto al que pertenece, de esta forma, se agrupan los Boundaries y sus datos formando un Boundary Data Set.

Por otra parte, el **Boundary document** representa la tupla Boundary-Data, es decir, la asociación entre un Boundary y unos datos concretos. Este documento cuenta con los siguientes campos:

- El campo **AUID**, un String que representa el identificador único del Boundary.
- El campo **bbox**, un polígono GeoJSON que representa el bounding box del Boundary. Este campo está formado a su vez por otros dos: un campo llamado **type** que especifica el tipo de objeto GeoJSON y un campo llamado **coordinates** que especifica las coordenadas del objeto.
- El campo **data**, que son los datos en formato JSON asociados al Boundary en cuestión.
- El campo **boundary_dataset_id**, identificador del Boundary Data Set al que pertenece el Boundary y datos asociados.

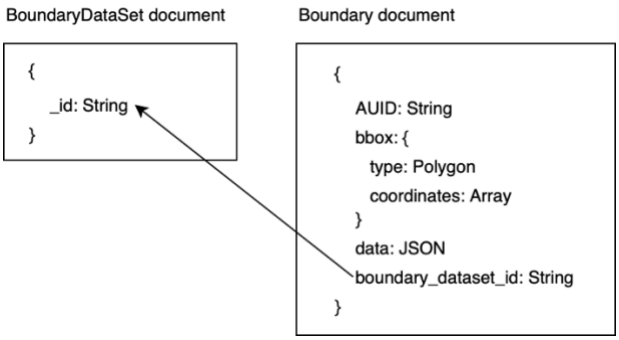

FIGURA 11 – MODELO DE IMPLEMENTACIÓN – DOCUMENTOS DE BOUNDARY STORE

En cuanto a la implementación de un Cell store, en la Figura 12 pueden observarse los dos documentos que la forman. Por una parte, el **CellDataSet document**, que representa el conjunto de Celdas junto con los datos asociados. Está formado únicamente con el campo **_id**, un String que, al igual que en el caso anterior, define el identificador del conjunto.

Por otra parte, el **Cell document** representa la tupla Cell-Data, es decir, la asociación entre una Celda y unos datos concretos. Este documento cuenta con los siguientes campos:

- El campo **CellID**, un String que representa el identificador único de la Celda.
- El campo **data**, que son los datos en formato JSON asociados a la Celda en cuestión.
- El campo **cell_dataset_id**, identificador del Cell Data Set al que pertenece la Celda y datos asociados.

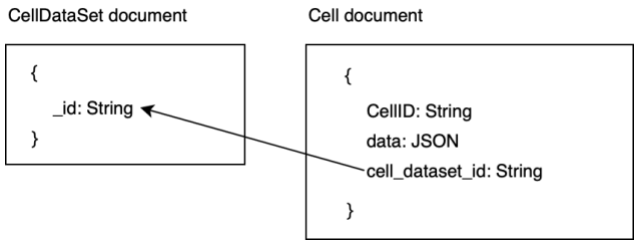

FIGURA 12 – MODELO DE IMPLEMENTACIÓN – DOCUMENTOS DE CELL STORE

3.3 Paquetes y clases

A continuación, se presenta un diagrama UML con los paquetes principales del sistema, pudiendo diferenciar 2 paquetes principales, coincidiendo con la división del desarrollo ya mencionada. El paquete **dggs**, que incluye la biblioteca de Python que permite importar, manipular, recuperar y dar persistencia a los datos basados en un DGGS, y el paquete **api_dggs**, que incluye la implementación de la API que da el soporte de back-end a aplicaciones web o móviles.

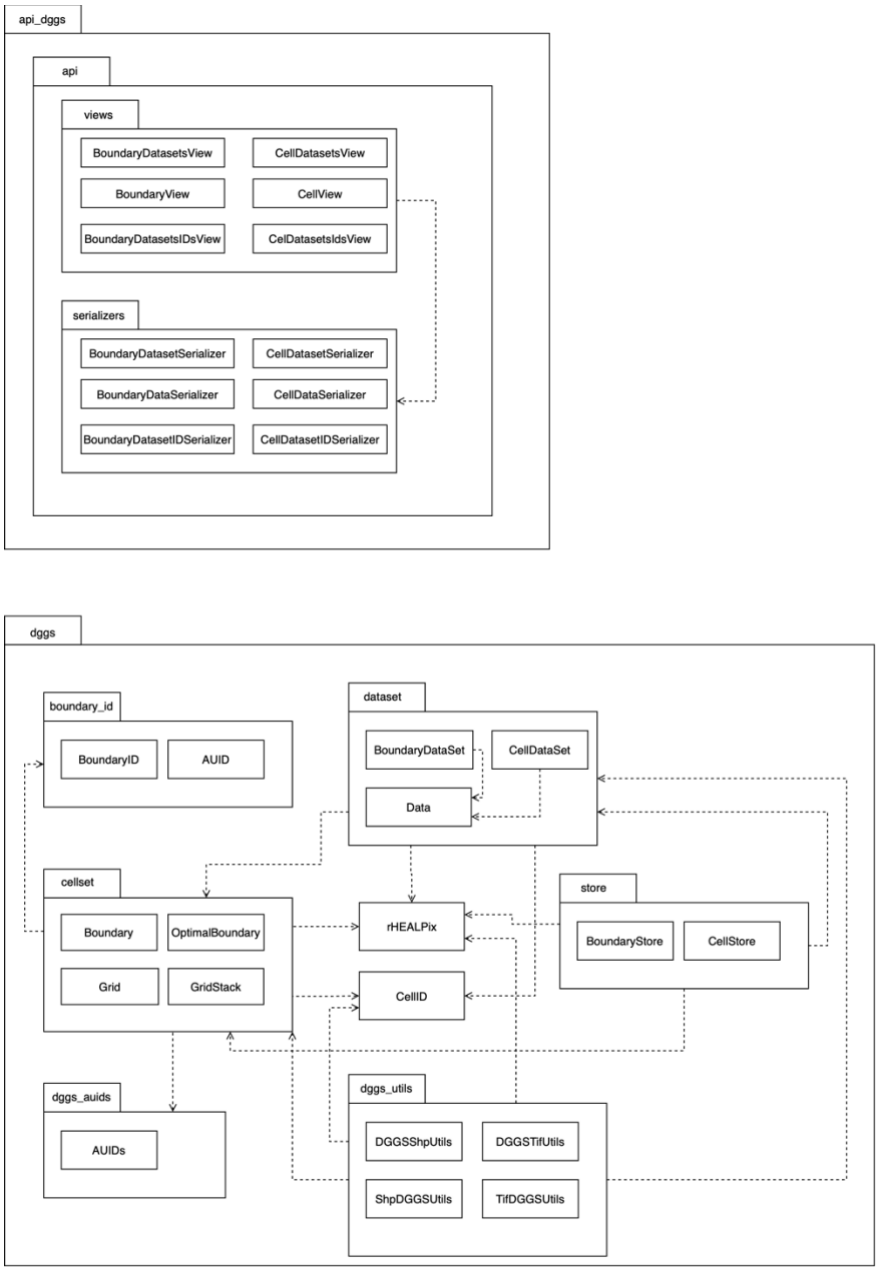

FIGURA 13 – DIAGRAMA DE PAQUETES DEL SISTEMA

En la Figura 13 pueden observarse los dos paquetes principales mencionados junto con los subpaquetes y clases que los forman. A continuación, se describen los paquetes más destacables del sistema y las dependencias que existen entre ellos.

Paquete dggs

Incluye la biblioteca Python que permite importar, manipular, recuperar y dar persistencia a los datos basados en un DGGS. Este paquete está formado por los siguientes subpaquetes y clases:

- Clase **rHEALPix:** en ella se implementa el DGGS rHEALPix**.**
- Clase **CellID:** representa el identificador de una celda.
- Paquete **Boundary_ID:** Agrupa las clases que representan identificadores de un Boundary, es decir, **BoundaryID** y **AUID**.
- Paquete **CellSet:** Agrupa las clases que representan un conjunto de celdas. Estas son, **CellSet, Boundary**, **OptimalBoundary**, **Grid**, y **GridStack**. Ya que todas son conjuntos de celdas, de todos estos conjuntos puede obtenerse un identificador único, es decir, un BoundaryID, por ello este paquete depende del paquete **boundary_id**. También existen dependencias con las clases esenciales rHEALPix y CellID.
- Paquete **Dataset:** Agrupa las clases que representan un conjunto de datos asociados a Boundaries o Celdas, **BoundaryDataSet** y **CellDataSet**, así como la clase que representa esos datos, **Data**. Este paquete, en concreto la clase BoundaryDataSet**,** depende del paquete **CellSet**, ya que en dicha clase, un Boundary es el objeto fundamental. También existen dependencias con las clases esenciales rHEALPix y CellID.
- Paquete **Store:** Este paquete recoge las clases que definen los dos almacenes, el almacén de Boundaries, **BoundaryStore**, y el almacén de celdas, **CellStore**. Estos almacenes trabajan con datasets, tanto de Boundaries como de Celdas, de ahí la dependencia con el paquete **DataSet.** También existen una dependencia con la clase fundamental rHEALPix.
- Paquete **Dggs_auids:** Contiene la clase con las operaciones para generar los AUIDs a partir de un conjunto de identificadores de celdas y viceversa, así como otras utilidades relacionadas con los identificadores, desarrolladas por el grupo grupo IAAA del I3A[3].
- Paquete **Dggs_utils:** Agrupa las clases que permiten realizar transformaciones de datos basados en un DGGS a formatos más habituales. En concreto, las clases **DGGSShpUtils** y **ShpDGGSUtils**, y las clases **DGGSTifUtils** y **TifDGGSUtils**. Se pueden observar dependencias con las clases esenciales rHEALPix y CellID, así como con los paquetes **CellSet** y **DataSet**, debido a la conversión desde y hace esos tipos.

[3] https://github.com/IAAA-Lab/dggs-auids

Paquete api_dggs

Incluye la implementación de la API que da el soporte de back-end a aplicaciones web o móviles. Este paquete está formado por los siguientes subpaquetes y clases:

- Paquete **Api.views:** Agrupa las vistas basadas en clases que se corresponden con los métodos HTTP utilizados para CRUD. Las clases **BoundaryDatasetsView**, **BoundaryView** y **BoundaryDatasetsIDsView** contienen las vistas que controlan la inserción, consulta, modificación y borrado de Boundaries Data Sets. Las clases **CellDatasetsView**, **CellView** y **CellDatasetsIDsView** contienen las vistas que controlan la inserción, consulta, modificación y borrado de Cell Data Sets. Estas vistas dependen de los serializadores y deserializadores agrupados en el paquete **api.serializers.**
- Paquete **Api.serializers:** Agrupa los serializadores que convierten objetos Python al formato de datos JSON (serialización) y viceversa (deserialización). Las clases **BoundaryDatasetSerializer**, **BoundaryDataSeriealizer** y **BoundaryDatasetIDSerializer** contienen los serializadores y deserializadores de los objetos Boundary, Data y BoundaryDataSet. Las clases **CellDatasetSerializer** y **CellDataSeriealizer** contienen los serializadores y deserializadores de los objetos Cell, Data y CellDataSet.

3.3.1 Diagrama de clases

El apartado anterior se ha centrado en presentar y describir los paquetes más relevantes del sistema y las clases que agrupan estos paquetes. En este, se detallan las principales clases existentes, las relaciones que existen entre ellas, y las operaciones y los atributos más relevantes (Figura 14).

- **rHEALPix:** clase que implementa el DGGS rHEALPix. Entre lo atributos con los que cuenta esta clase destacan el atributo **N_side**, entero, mínimo 2, de modo que cada celda tiene N_side x N_side celdas hijas; **north_square**, entero entre 0 y 3 que indica la posición del cuadrado polar norte; **south_square**, entero entre 0 y 3 que indica la posición del cuadrado del polo sur; y **max_area**, área, en metros cuadrados, de las celdas de la rejilla elipsoidal más pequeña. Esta clase cuenta con métodos que permiten obtener información de las celdas como su ancho, su fila y columna, así como sus coordenadas proyectadas o geodésicas, o la obtención de una celda dadas las coordenadas de un punto. En el Anexo 1 se detalla cada uno de los métodos de esta clase.

- **CellID:** clase que representa el identificador de una celda. Tiene un único atributo ***value*** que es la cadena del identificador. Cuenta con una operación ***getRefinement()*** que devuelve el nivel de refinamiento de la celda. Por ejemplo, la celda con identificador N01 tendría un nivel de refinamiento igual a 2, la celda P tendría un nivel de refinamiento igual a 0.

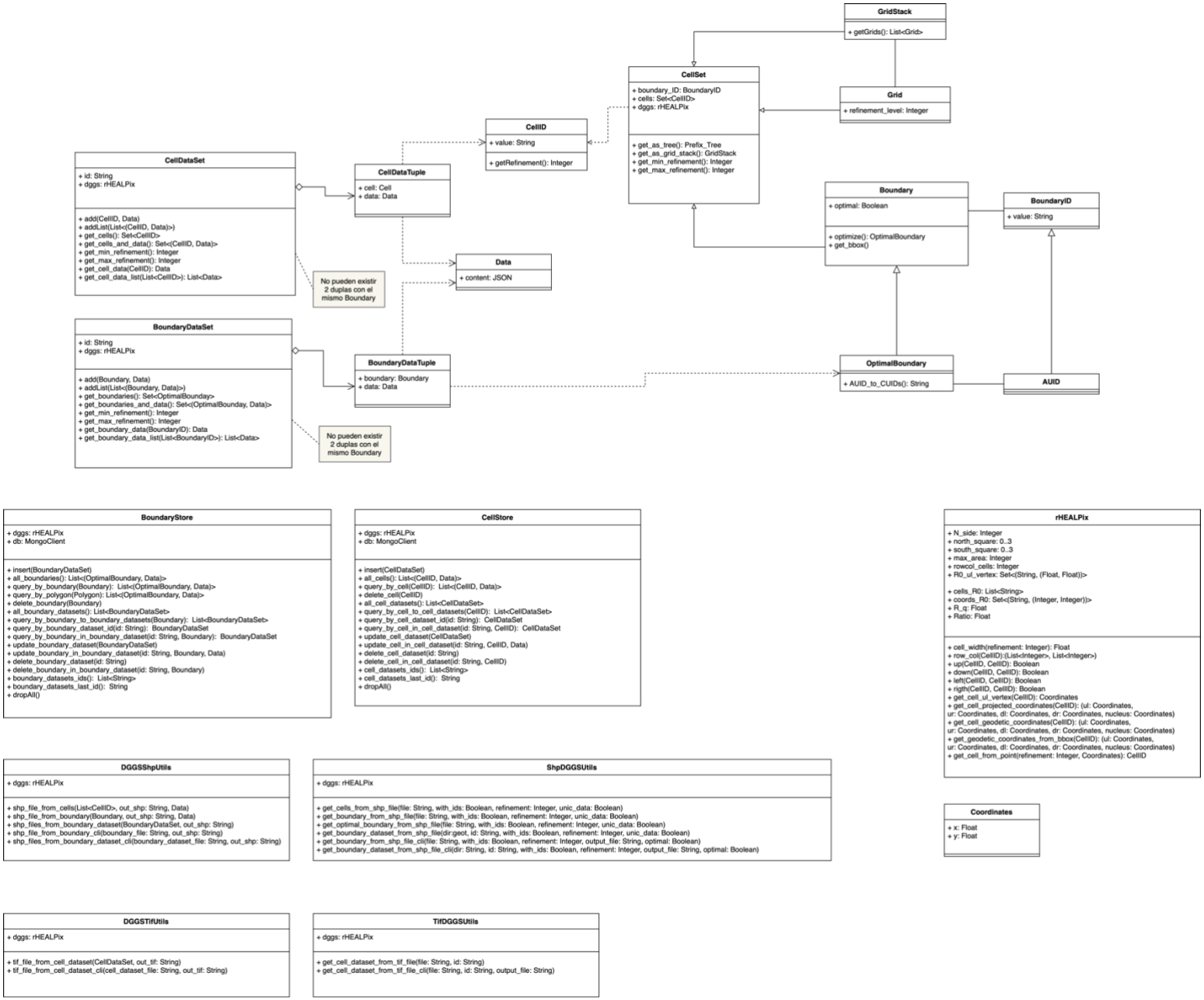

FIGURA 14 - DIAGRAMA DE CLASES

- **BoundaryID**: representa el identificador único de un Boundary formado a partir de los identificadores de las celdas que lo forman. Tiene un único atributo ***value*** que es la cadena del identificador.
    - **AUID:** identificador único de un Optimal Boundary formado a partir de los identificadores de las celdas que lo forman tal y como se define en el **AGILE19 paper**. Tiene un único atributo ***value*** que es la cadena del identificador
- **CellSet:** clase que representa un conjunto de celdas. El atributo ***cells*** es el conjunto de identificadores de celdas (CellID) que forman el CellSet, el atributo ***boundary_ID*** es la cadena con el identificador único del conjunto, formada por la concatenación de los identificadores de celdas, y el atributo ***dggs***, que hace referencia al dggs utilizado, en este caso rHEALPix. Los métodos de esta clase permiten obtener información del conjunto como su nivel máximo y mínimo de refinamiento, así como obtener otras formas de representación del mismo, como por ejemplo un trie o una Grid Stack. En el Anexo 1 se detalla cada uno de los métodos de esta clase.
    - **Boundary:** conjunto de Celdas que delimitan un área en un DGGS. Especialización de la clase CellSet, por lo que hereda todos sus atributos y operaciones, a los que se añade el atributo ***optimal***, booleano que indica si el Boundary es óptimo o no, y las operaciones ***optimize(),*** que optimiza el Boundary en cuestión, devolviendo un OptimalBoundary, y la operación ***get_bbox()***, que devuelve las coordenadas geodésicas del bbox (bounding box) del Boundary (inferior izquierda, inferior derecha, superior derecha, superior izquierda).
        - **OptimalBoundary:** conjunto óptimo de Celdas que cubre un área en un DGGS, entendiendo óptimo como el conjunto más pequeño de celdas que cubren exactamente esa área. Especialización de la clase Boundary, por lo que hereda todos sus atributos y operaciones, pero en este caso, el identificador boundary_id es del tipo **AUID**, ya que debe ser óptimo. A las operaciones que hereda se añade la operación ***AUID_to_CUIDs()***, que devuelve el identificador como concatenación simple de identificadores de celda, es decir, como un BoundaryID.
    - **Grid:** una teselación de una cierta área, por lo que en sus límites cada celda existe, no hay vacíos ni agujeros. Es otra especialización de la clase CellSet, y, en este caso, todas las Celdas del conjunto tienen el mismo nivel de refinamiento, por lo que se añade el atributo ***refinement_level,*** que es el nivel de refinamiento del conjunto o Grid.
    - **GridStack:** una serie de Grids que están ordenadas por su nivel de refinamiento.
- **BoundaryDataSet***:* conjunto de Boundaries en el que cada Boundary tiene asociados unos datos concretos. Cuenta con 3 atributos: **id**, que define el identificador del conjunto, lo que permite realizar operaciones sobre un conjunto concreto dado su identificador; ***dggs***, que hace referencia al dggs utilizado, en este caso rHEALPix; y el conjunto de Boundaries junto con los datos asociados a cada uno de ellos, **boundary_data_set**. Esta clase cuenta con métodos que permiten obtener información

del conjunto como su nivel máximo y mínimo de refinamiento, así como la inserción de Boundaries en el mismo o, por ejemplo, la obtención de un Boundary y sus datos dado el BoundaryID. En el Anexo 1 se detalla cada uno de estos métodos.

- **CellDataSet**: conjunto de Celdas en el que cada Celda tiene asociados unos datos concretos. Cuenta con 3 atributos: **id**, que define el identificador del conjunto, lo que permite realizar operaciones sobre un conjunto concreto dado su identificador; ***dggs***, que hace referencia al dggs utilizado, en este caso rHEALPix; y el conjunto de Celdas junto con los datos asociados a cada una de ellas, **cell_data_set**. Esta clase cuenta con métodos que permiten obtener información del conjunto como su nivel máximo y mínimo de refinamiento, así como la inserción de celdas en el mismo o, por ejemplo, la obtención de una celda y sus datos dado el CellID. En el Anexo 1 se detalla cada uno de estos métodos.

Todas las clases anteriores se localizan en el **componente DGGS-rHEALPix**. En la siguiente sección, Componentes y conectores, se describe dicho componente.

- **BoundaryStore:** clase que representa el almacén de Boundaries. Cuenta con 2 atributos, el atributo ***dggs***, que hace referencia al dggs utilizado, en este caso rHEALPix, y el atributo ***db***, que es la base de datos en la que se da persistencia, en este caso, un cliente de MongoDB. En cuanto a los métodos con los que cuenta esta clase, implementan las operaciones de inserción, consulta, modificación y borrado de datos en el almacén. En el Anexo 1 se detalla cada uno de estos métodos. Esta clase se localiza en el **componente Boundary Store**, que expone a través de su **interfaz** los métodos de esta para ser utilizados por el servidor. En la siguiente sección Componentes y conectores se describe dicho componente.

- **CellStore:** clase que representa el almacén de Celdas. Cuenta con 2 atributos, el atributo ***dggs***, que hace referencia al dggs utilizado, en este caso rHEALPix; y el atributo ***db***, que es la base de datos en la que se da persistencia, en este caso, un cliente de MongoDB. En cuanto a los métodos con las que cuenta esta clase, implementan las operaciones de inserción, consulta, modificación y borrado de datos en el almacén. En el Anexo 1 se detalla cada uno de estos métodos. Esta clase se localiza en el **componente Cell Store**, que expone a través de su **interfaz** los métodos de esta para ser utilizados por el servidor. En la siguiente sección Componentes y conectores se describe dicho componente.

- **DGGSShpUtils:** clase que contiene las operaciones para transformar datos asociados a un Boundary, en datos asociados a entidades de un modelo vectorial contenidos en un shapefile; así como, un conjunto datos asociados a Boundaries contenidos en un Boundary Datase,t en un conjunto de datos asociados a entidades de un modelo vectorial contenidos en un conjunto de shapefiles. Destaca el atributo ***dggs***, que hace referencia al dggs utilizado, en este caso rHEALPix. En el Anexo 1 se detalla cada uno de sus métodos. Esta clase se localiza en el **componente DGGS-rHEALPix**, que expone a

través de su **interfaz** los métodos de esta para ser utilizados por otros. En la siguiente sección Componentes y conectores se describe dicho componente.

- **ShpDGGSUtils:** clase que contiene las operaciones para transformar datos asociados a entidades de un modelo vectorial contenidos en un shapefile, en datos asociados a un Boundary; así como, un conjunto de datos asociados a entidades de un modelo vectorial contenidos en un conjunto de shapefiles, en un conjunto de datos asociados a Boundaries contenidos en un Boundary Dataset. Destaca el atributo ***dggs***, que hace referencia al dggs utilizado, en este caso rHEALPix. En el Anexo 1 se detalla cada uno de sus métodos. Esta clase se localiza en el **componente DGGS-rHEALPix**, que expone a través de su **interfaz** los métodos de esta para ser utilizados por otros. En la siguiente sección, Componentes y conectores, se describe dicho componente.
- **DGGSTifUtils:** clase que contiene las operaciones para transformar datos asociados a celdas contenidos en un Cell Dataset, en datos asociados a píxeles de un modelo ráster contenidos en un fichero GeoTIFF. Destaca el atributo ***dggs***, que hace referencia al dggs utilizado, en este caso rHEALPix. En el Anexo 1 se detalla cada uno de sus métodos. Esta clase se localiza en el **componente DGGS-rHEALPix**, que expone a través de su **interfaz** los métodos de esta para ser utilizados por otros. En la siguiente sección, Componentes y conectores, se describe dicho componente.
- **TifDGGSUtils:** clase que contiene las operaciones para transformar datos asociados a píxeles de un modelo ráster contenidos en un fichero GeoTIFF, en un conjunto datos asociados a celdas contenidos en un Cell Dataset. Destaca el atributo ***dggs***, que hace referencia al dggs utilizado, en este caso rHEALPix. En el Anexo 1 se detalla cada uno de sus métodos. Esta clase se localiza en el **componente DGGS-rHEALPix**, que expone a través de su **interfaz** los métodos de esta para ser utilizados por otros. En la siguiente sección, Componentes y conectores, se describe dicho componente.

3.4 Componentes y conectores

En la Figura 15 se presenta un diagrama UML con los componentes principales del sistema y sus conectores. A continuación, se describe cada componente, qué permite cada conector, y se documentan las interfaces más relevantes del sistema.

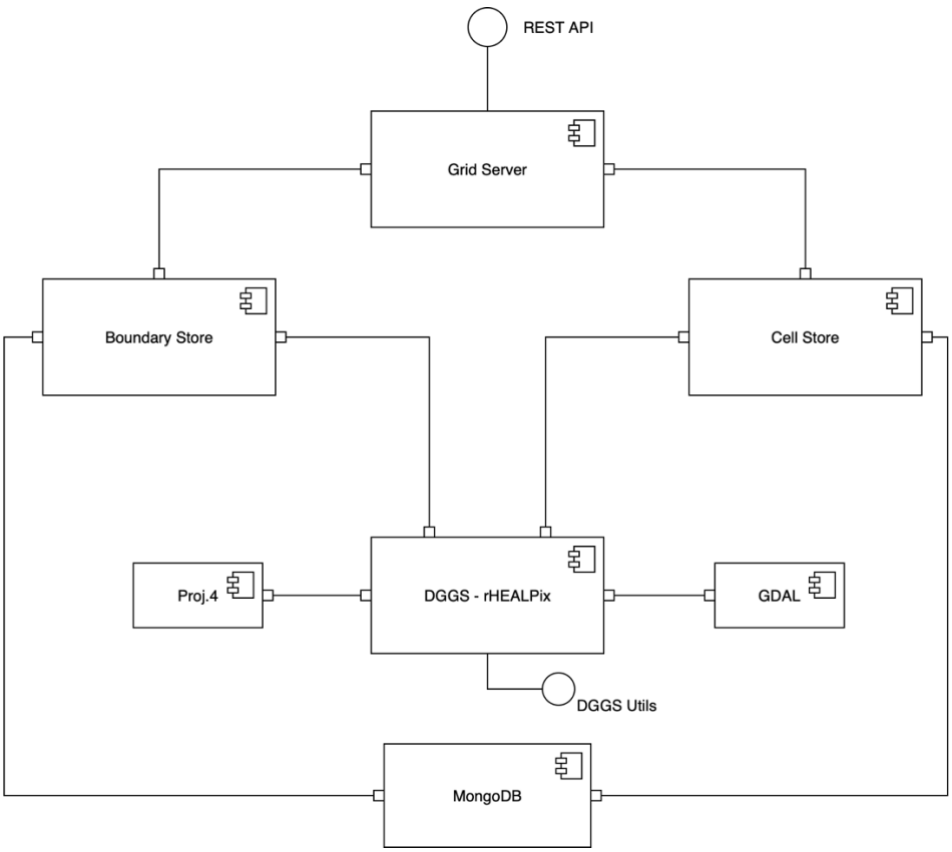

FIGURA 15 - DIAGRAMA DE COMPONENTES Y CONECTORES DEL SISTEMA

- **Grid Server**: servidor desarrollado con el framework Django REST que expone una API, descrita en el Anexo 3, que permite a un cliente web o móvil importar, recuperar o manipular datos basados en el modelo de DGGS implementado. El servidor hace uso de las operaciones del Boundary y Cell Store incluidos en la biblioteca Python desarrollada.

- **Boundary Store**: almacén de Boundaries con el que se da persistencia, sobre una base de datos MongoDB, a Boundary Data Sets, es decir, a conjuntos de Boundaries en los que cada Boundary tiene asociado unos datos concretos.

  Tal y como ya se ha explicado en la sección anterior, este componente expone a través de su **interfaz** las operaciones de la clase BoundaryStore, de la que hace uso el **componente Grid Server** y sobre la que se sustenta la API que este expone, de ahí el conector entre estos dos componentes en el diagrama. La descripción por tanto de las operaciones con las que cuenta la interfaz de este componente se detallan en el Anexo 1, en la descripción de los métodos de la clase BoundaryStore.

A la hora de insertar un BoundaryDataSet, tal y como se explica en el Modelo de datos, cada Boundary perteneciente al conjunto se inserta junto con su **bounding box**, de forma que puedan realizarse operaciones como la de búsqueda mediante **intersección** de un polígono. La operación **insert(BoundaryDataSet),** descrita en el Anexo 1, calcula dicho bounding box, antes de insertar el documento de MongoDB en la base de datos. Para realizar este cálculo, este componente utiliza la operación ofrecida por el **componente DGGS-rHEALPix**: **boundary.get_bbox()**. Además, a la hora de realizar búsquedas mediante un identificador de un Boundary, dado que en la base de datos se guarda el AUID de cada uno de ellos, se optimiza el Boundary, si no es un OptimalBoundary, antes de realizarla. Para ello, este componente hace uso de la operación **boundary.optimize()**, ofrecida también por el **componente DGGS-rHEALPix**. El uso de estas operaciones y de los constructores de las clases Boundary, OptimalBoundary, AUID, Data y BoundaryDatSet para la creación de objetos de estos tipos se representa en el diagrama mediante el conector entre Boundary Store y DGGS-rHEALPix.

El almacén se implementa sobre una base de datos MongoDB, por ello, este componente utiliza el driver ***pymongo*** [4] para trabajar con dicha base. La comunicación entre el Boundary Store y la base de datos MongoDB a través de dicho driver, se ve reflejada en el diagrama mediante un conector.

- **Cell Store**: almacén de Celdas con el que se da persistencia, sobre una base de datos MongoDB, a Cell Data Sets, es decir, a conjuntos de Celdas en los que se asocian datos a cada una de las celdas que los forman.

  Al igual que el componente anterior, este componente expone a través de su **interfaz** las operaciones de la clase CellStore, de la que hace uso el **componente Grid Server** y sobre la que se sustenta la API que este expone, de ahí el conector entre estos dos componentes en el diagrama. La descripción por tanto de las operaciones con las que cuenta la interfaz de este componente se detallan en el Anexo 1, en la descripción de los métodos de la clase CellStore.

  El uso los constructores de las clases CellID, Data y CellDatSet para la creación de objetos de estos tipos se representa en el diagrama mediante el conector entre Cell Store y DGGS-rHEALPix.

  El almacén se implementa sobre una base de datos MongoDB, por ello, este componente utiliza el driver ***pymongo*** para trabajar con dicha base. La comunicación entre el Cell Store y la base de datos MongoDB a través de dicho driver se ve reflejada en el diagrama mediante un conector.

- **DGGS-rHEALPix**: implementación del modelo DGGS diseñado, utilizando rHEALPix como DGGS por defecto. Permite a los **componentes BoundaryStore y CellStore** la creación de objetos de tipo Boundary, OptimalBoundary, AUID, CellID, Data, BoundaryDatSet, así como el uso de servicios como obtención del bounding box de un Boundary mediante la operación **get_bbox()** o la optimización y obtención de un OptimalBoundary mediante

[4] https://pymongo.readthedocs.io/en/stable/

la operación **optimize()**, operaciones de la clase Boundary descrita en la sección anterior.

Además, este componente ofrece una **interfaz** para la realización de transformaciones de datos basados en un DGGS a otros formatos más habituales, en concreto, Shapefile y GeoTiff, y viceversa. A través de esta interfaz, tal y como ya se ha explicado en la sección anterior, se exponen las operaciones de las clases **DGGSShpUtils, ShpDGGSUtils, DGGSTifUtils y TifDGGSUtils,** descritas en el Anexo 1.

- **Proj.4**: biblioteca de proyecciones cartográficas y transformaciones de coordenadas. Ofrece una interfaz Python utilizada por el componente DGGS-rHEALPix.
- **GDAL**: biblioteca de traductores para formatos de datos geoespaciales ráster y vectoriales. Ofrece una interfaz Python utilizada por el componente DGGS-rHEALPix.
- **MongoDB**: base de datos sobre la que se implementan los dos almacenes descritos anteriormente. Como ya se ha explicado, se utiliza el driver *pymongo* para trabajar con ella. De entre las operaciones que ofrece dicho driver, se han utilizado las siguientes:
    - **insert_one():** permite insertar un documento único (un BoundaryDataSet o un CellDataSet).
    - **create_index():** crea un índice en una colección.
    - **find():** permite consultar la base de datos.
    - **update_many:** actualiza uno o más documentos que coincidan con el filtro.
    - **delete_many():** elimina uno o más documentos que coincidan con el filtro.

3.4.1 Documentación de la dinámica del sistema

Como ejemplo interesante de funcionamiento del sistema que involucra la interacción entre varios componentes de este, en la Figura 16, se presenta un diagrama que muestra de forma resumida el proceso de inserción de un BoundaryDataSet en la base de datos.

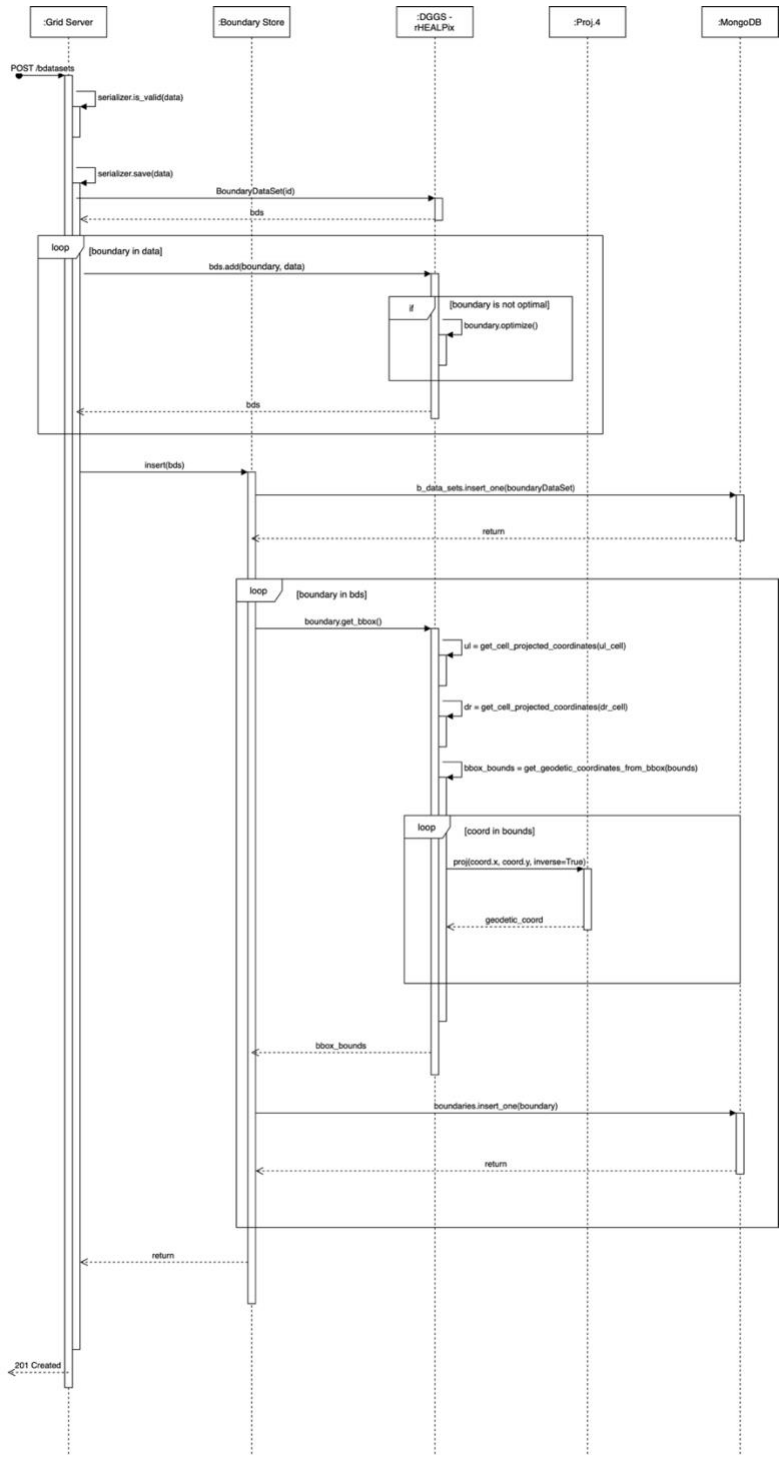

FIGURA 16 - DIAGRAMA DE SECUENCIA. INSERCIÓN DE UN BOUNDARYDATASET

El proceso comienza con una petición POST al Grid server en la que el BoundaryDataSet a insertar es pasado en el cuerpo de la petición en formato JSON. La vista recibe los datos, estos son serializados y validados. Se construye el objeto BoundaryDataSet, y por cada Boundary del conjunto, se realiza una llamada a la función add() de este objeto, con el Boundary y los datos asociados a este pasados como parámetro. Esta función comprueba si el Boundary es óptimo, si no lo es, lo optimiza obteniendo el Optimal Boundary correspondiente. Un vez el objeto BoundaryDataSet tiene todo el conjunto, se pasa como parámetro a la función *insert()* del BoundaryStore.

La función *insert()del BoundaryStore*, en primer lugar, inserta un documento en la colección de Boundary Data Sets con su identificador. En segundo lugar, por cada uno de los Boundaries del conjunto, calcula su bounding box para incluirlo en el documento a insertar en la colección de Boundaries. Se calculan los vértices superior izquierdo e inferior derecho del bounding box a partir las celdas superior izquierda e inferior derecha del conjunto. Una vez obtenidos los vértices, se convierten sus coordenadas, de proyectadas (proyección rHEALPix) a geodésicas (longitud y latitud sobre el elipsoide WGS84). Se devuelve al BoundaryStore el vértice inferior izquierdo, inferior derecho, superior derecho y superior izquierdo del bounding box.

Se inserta un documento en la colección de Boundaries con su AUID, bbox, datos asociados y el identificador del BoundaryDataSet al que pertenece. Una vez insertados todos los Boundaries del BoundaryDataSet, el Grid server devuelve 201 al cliente.

3.5 Distribución

En la Figura 17 se presenta un diagrama UML de despliegue del sistema. En él puede observarse que la base de datos MongoDB se ha desplegado en un contenedor Docker[5]. Un contenedor es una unidad de software estándar que empaqueta el código y todas sus dependencias para que la aplicación se ejecute de manera rápida y fiable de un entorno a otro. Los contenedores son una abstracción en la capa de la aplicación que agrupa el código y las dependencias juntas. Se pueden ejecutar varios contenedores en la misma máquina y compartir el núcleo del sistema operativo con otros contenedores, cada uno de los cuales se ejecuta como procesos aislados en el espacio del usuario.

Para realizar este despliegue mediante Docker, se ha creado un Dockerfile que permite a Docker construir la imagen de MongoDB de forma automática. En él se describe la configuración del contenedor de MongoDB, con un puerto externo 27017 que será usado por el servidor y un puerto interno 27017 que en este caso no es usado, y sus volúmenes montados, en este caso uno en /data/db, donde se mantiene los datos de configuración. El servidor se comunica con la base de datos mediante el driver pymongo, que ofrece un conjunto de herramientas para interactuar con MongoDB a través de Python.

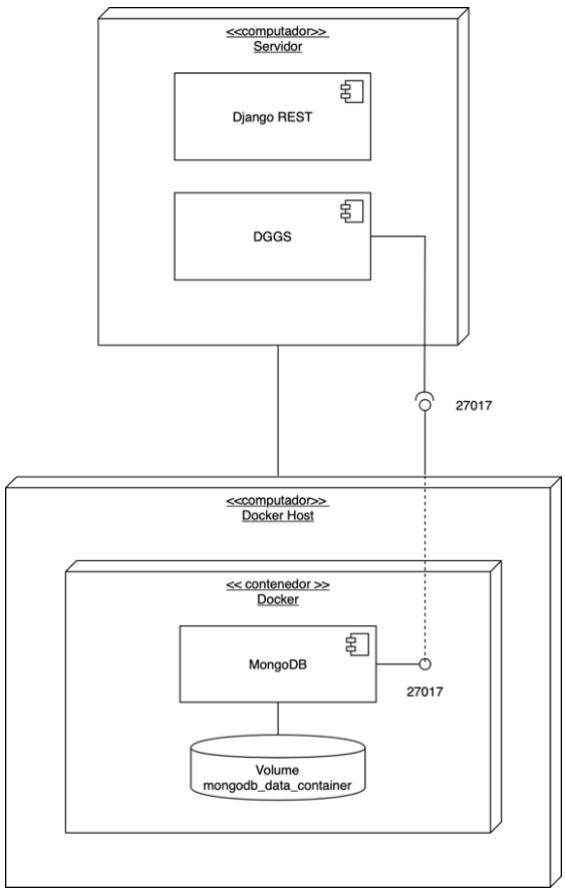

FIGURA 17 - DIAGRAMA DE DESPLIEGUE

[5] https://www.docker.com/

3.6 Implementación

A continuación, se van a explicar algunos aspectos interesantes de la implementación del sistema como las tecnologías o librerías que se han utilizado.

3.6.1 Python

Para el desarrollo de este trabajo se ha utilizado **Python**[6], un lenguaje de programación interpretado cuya filosofía hace hincapié en la legibilidad de su código. Se trata de un lenguaje de programación multiparadigma, ya que soporta orientación a objetos, programación imperativa y, en menor medida, programación funcional. Es un lenguaje interpretado, dinámico y multiplataforma. Es administrado por la Python Software Foundation. Posee una licencia de código abierto, denominada Python Software Foundation License. Entre las ventajas de utilizar Python frente a otros lenguajes destacan la gran cantidad de **bibliotecas**, lo que favorece la productividad a la hora de programar; la facilidad de uso de las **estructuras de datos**, Python ofrece también la opción de tipado dinámico de datos de alto nivel que reduce la longitud del código; es de **código abierto**, Python se desarrolla bajo una licencia de código abierto aprobada por OSI, que lo hace libre de usar y distribuir, incluso con fines comerciales; es **extensible,** Python se puede extender a otros lenguajes; y es fácil de **comprender** y **codificar**, debido a que no es un lenguaje tan detallado, leer Python es muy parecido a leer en inglés, lo que hace también que se requiera menos codificación con respecto a otros lenguajes.

3.6.2 MongoDB

Al inicio de este trabajo se estudiaron diferentes alternativas para la persistencia de los datos (BD) para seleccionar la más adecuada. Entre estas alternativas destacan las bases de datos orientadas a documentos, a grafos, clave-valor, orientadas a columnas y a arrays. En cuanto a las orientadas a grafos, se han hecho pruebas con Neo4J, pero el hecho de que sea un modelo muy flexible puede hacer que los datasets se expandan demasiado, debido a la facilidad de añadir nuevas relaciones. En relación con las clave-valor, se ha probado Riak, pero el hecho de que algunos Boundaries pueden estar ligados a distintos datos en distintos Boundary Datasets hace más complicado una implementación en este tipo de base de datos con respecto a las demás. Se contempló la posibilidad de usar Cassandra, como base de datos orientada a columnas, incluso el uso de una base de datos SQL. En cuanto a las orientadas a arrays, se analizó TileDB, un motor diseñado en torno a arrays multidimensionales que permite almacenar y acceder a matrices densas (p. Ej., Imágenes de satélite), matrices dispersas (p. Ej., LiDAR, genómica), dataframes (cualquier dato en forma tabular) y valores-clave. Pero al final se decidió usar MongoDB para dar persistencia al modelo.

MongoDB [7]es un sistema de base de datos NoSQL, orientado a documentos y de código abierto. MongoDB guarda estructuras de datos BSON (una especificación similar a JSON) con un esquema dinámico, haciendo que la integración de los datos en ciertas aplicaciones sea más fácil y rápida. Entre las razones por las que se ha decidido utilizar este tipo de base de datos destacan

[6] https://www.python.org/
[7] https://www.mongodb.com/es

la **flexibilidad**, ya que permite que los datos estructurados, semiestructurados y no estructurados se puedan almacenar juntos, sin necesidad de una conversión previa. Además, que los datos se almacenen en estructuras parecidas a un JSON hace que el flujo de datos dentro de la aplicación no tenga mayores cambios en la estructura de datos; la **escalabilidad**, MongoDB, al ser una base de datos distribuida puede escalar no solamente de forma vertical (CPU y RAM) si no que también de forma horizontal (creando más nodos); buena sintaxis para hacer **consultas**, cuenta con múltiples operadores que permiten crear consultas con poco código, además cuenta con las agregaciones que permiten realizar operaciones entre múltiples colecciones; **alta disponibilidad**, MongoDB permite tener clúster distribuidos, lo que mejora la velocidad de consulta al disminuir la latencia que existe entre el clúster de base de datos y el servicio que ejecuta la query; y que es de **código abierto**.

3.6.3 Django REST

Django REST framework [8]es un kit de herramientas potente y flexible para crear API web. Alguna de las razones por las que se ha seleccionado este framework para el desarrollo de esta son la opción de tener una **API navegable** desde el navegador (Figura 18), lo que facilita mucho la realización de pruebas; la **serialización** de datos a partir de fuentes de datos ORM o no ORM; muy buena **documentación** y amplia comunidad al ser **open source**; y muy **personalizable**.

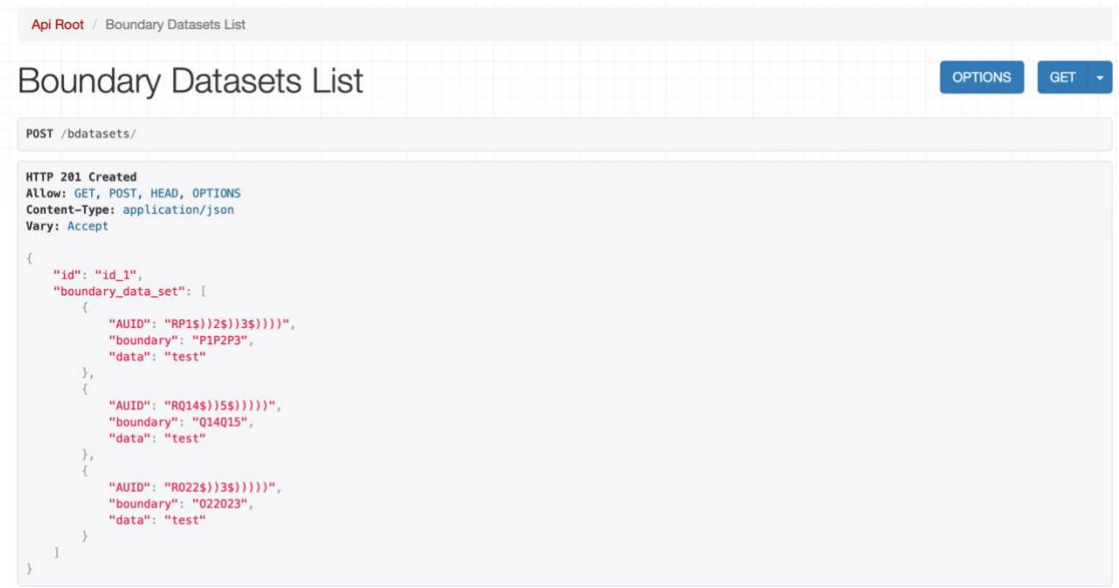

FIGURA 18 - API NAVEGABLE

3.6.4 Proj.4

La conversión de coordenadas proyectadas (proyección rHEALPix) a geodésicas (longitud y latitud sobre el elipsoide WGS84), y viceversa, es necesaria en numerosas operaciones de la bilbioteca desarrollada. Para realizar dicha conversión se ha utilizado **Proj4**[9], que es una biblioteca que proporciona métodos para transformar entre sistemas de referencia de

[8] https://www.django-rest-framework.org/
[9] https://proj.org

coordenadas diferentes. Sus características principales son que proporciona el punto de transformación de un sistema de referencia de coordenadas a otro, que incluye transformación entre datums (datos de referencia), y la gran cantidad de clases de proyección compatibles. En concreto se ha utilizado **pyproj**[10], una interfaz de Python para PROJ, para ejecutar una proyección rHEALPix en un modelo elipsoidal WGS84 y viceversa.

3.6.5 GDAL

A la hora de transformar datos asociados a píxeles de un modelo ráster contenidos en un fichero GeoTIFF en un conjunto datos asociados a Celdas contenidos en un Cell Dataset, es necesaria también una conversión de coordenadas a rHEALPix. Para ello se ha utilizado **GDAL**[11], que es una biblioteca de traducción o transformación para formatos de datos geoespaciales ráster y vectoriales que la Open Source Geospatial Foundation publica bajo una licencia de código abierto estilo X / MIT. En concreto se ha utilizado **gdalwarp**[12]**,** una utilidad de mosaico de imágenes, reproyección y deformación. El programa puede reproyectar a cualquier proyección soportada.

[10] https://pypi.org/project/pyproj/
[11] https://gdal.org
[12] https://gdal.org/programs/gdalwarp.html

3.7 Pruebas

En este trabajo se incluye una colección de tests automáticos en los que se realizan pruebas sobre las operaciones más críticas del sistema, así como pruebas con datos reales proporcionados por GEOT del Dpto. de Geografía y Ord. del Territorio, que han permitido probar las transformaciones de formatos. A continuación, se explican de manera resumida los tests realizados:

- **Boundary** tests

  Conjunto de pruebas realizadas sobre los objetos Boundary. Se incluyen pruebas como la obtención de un Boundary a partir de su identificador y a partir de un conjunto de celdas (***test_boundary_from_boundary_ID(***) y ***test_boundary_from_cells()***), la correcta obtención de un GridStack (***test_grid_stack()***), y del nivel mínimo y máximo de refinamiento (***test_min_refinement(), test_max_refinement()***), así como el correcto funcionamiento de las operaciones de optimización, obtención de las coordenadas de las celdas de un Boundary, u obtención del bbox (***test_optimize_boundary(), test_boundary_projected_coordinates(), test_boundary_geodetic_coordinates() y test_bbox()***).

- **BoundaryDataSet** tests

  Conjunto de pruebas realizadas sobre los objetos BoundaryDataSet. Se incluyen pruebas como la creación de un BoundaryDataSet (***test_boundary_dataset()***), la inclusión de un Boundary o una lista de ellos al conjunto (***test_boundary_dataset_add() y test_boundary_dataset_add_list()***), la obtención de la lista de Boundaries o lista de tuplas Boundary-Data del conjunto (***test_get_boundaries() y test_get_boundaries_and_data()***), la correcta obtención del nivel mínimo y máximo de refinamiento (***test_min_refinement(), test_max_refinement()***), y de los datos a partir de un Boundary o lista de Boundaries (***test_get_boundary_data() y test_get_boundary_data_list()***).

- **BoundaryStore** tests

  Conjunto de pruebas realizadas sobre los objetos BoundaryStore. Se incluyen pruebas de todas las operaciones de inserción, recuperación, modificación y borrado en el almacén.

- **CellDataSet** tests

  Conjunto de pruebas realizadas sobre los objetos CellDataSet. Se incluyen pruebas como la creación de un CellDataSet (***test_cell_dataset()***), la inclusión de una Celda o una lista de ellas al conjunto (***test_cell_dataset_add() y test_cell_dataset_add_list()***), la obtención de la lista de Celdas o lista de tuplas Celda-Data del conjunto (***test_get_cells() y test_get_cells_and_data()***), la correcta obtención del nivel mínimo y máximo de refinamiento (***test_min_refinement(), test_max_refinement()***), y de los datos a partir de una Celda o lista de Celdas (***test_get_cell_data() y test_get_cell_data_list()***).

- **CellStore** tests

  Conjunto de pruebas realizadas sobre los objetos CellStore. Se incluyen pruebas de todas las operaciones de inserción, recuperación, modificación y borrado en el almacén.

- **rHEALPix** tests

  Conjunto de pruebas realizadas sobre las operaciones principales del DGGS rHEALPix. Se incluyen pruebas como el cálculo correcto del ancho de una Celda (***test_cell_width()***), la obtención de la fila y columna de una Celda (***test_rowcol()***), la comprobación de si una Celda está más arriba, abajo, más a la izquierda o a la derecha que otra (***test_up_down() y test_right_left()***), la correcta obtención del vértice superior izquierdo de un Celda (***test_cell_ul_vertex()***), de las coordenadas proyectadas y geodésicas de una Celda (***test_cell_projected_coordinates() y test_cell_geodetic_coordinates()***), y la correcta obtención de una Celda a partir de las coordenadas de un punto (***test_cell_from_point()***).

- Tests de **transformaciones**

  En primer lugar, un conjunto de pruebas para verificar la correcta transformación de un Boundary y BoundaryDataSet en un shapefle y viceversa: (***test_shp_file_from_boundary(), test_shp_files_from_boundary_dataset(), test_get_boundary_from_shp_file() y test_get_boundary_dataset_from_shp_file()***).

  Por otra parte, un conjunto de pruebas para verificar la correcta transformación de un CellDataSet en un fichero GeoTIFF y viceversa: (***test_tif_file_from_cell_dataset() y test_cell_dataset_from_tif_file()***)

Estos tests se han realizado utilizando el módulo **unittest** [13]de Python, un marco de pruebas inspirado originalmente en JUnit y similar a los principales frameworks de pruebas unitarias en otros lenguajes. Admite la automatización de pruebas, códigos de configuración y cierre para pruebas, la agregación de pruebas en colecciones y la independencia de las pruebas del marco de informes.

[13] https://docs.python.org/3/library/unittest.html

4. Gestión del proyecto

En esta sección se detalla cómo se ha llevado a cabo el proyecto, teniendo en cuenta aspectos como la planificación e historia del proyecto, el control de esfuerzos y la gestión de configuraciones.

4.1 Planificación e historia del proyecto

Tal y como se describió en la propuesta del trabajo, se planificaron las siguientes fases:

- Colaborar con los investigadores en el diseño de modelos de datos geográficos DGGS.
- Estudiar alternativas para la persistencia de los datos (BD) y seleccionar una adecuada.
- Diseñar e implementar una biblioteca Python que permita importar, manipular, recuperar y dar persistencia a datos sobre los modelos diseñados y en la BD elegida.
- Diseñar e implementar una API web que dé el soporte de back-end al cliente web que está siendo desarrollado por los otros investigadores.
- Carga de datos reales. Pruebas del sistema.
- Completar la documentación y la memoria del TFG.

A continuación, se detalla la historia del proyecto, que a grandes rasgos coincide con la planificación anterior. El desarrollo del proyecto se extiende desde noviembre de 2019 hasta septiembre de 2020.

Primera fase

Se estableció el objetivo general del trabajo, es decir, avanzar hacia una infraestructura de datos geográficos creados sobre uno o más DGGS. Soportar el almacenamiento (esquema de datos adecuado para DGGS), acceso a través de la red, conversión desde y hacia formatos más habituales de información geográfica, etc.

Como tareas a destacar en esta primera fase se encuentran: familiarizarse con DGGS, familiarizarse con datos geográficos y estudiar alternativas para la persistencia de los datos (BD), y seleccionar una adecuada. Se decidió realizar la implementación de Boundary en una BD NoSQL documental, en concreto, MongoDB. Se consideró el modelo de grafos interesante, pero se vio que era añadir complejidad extra.

Segunda fase

Como objetivo de esta fase se planteó la localización espacial de los Boundaries en la base de datos. Aprovechar el potencial espacial de la BD para poder responder a la query espacial básica: ¿qué Boundaries intersectan con este rectángulo?

Se decidió que lo más sencillo era almacenar el bounding box de cada Boundary en la BD, en el formato de geometría adecuado, indexar esa columna y luego consultar por ella. No tenía sentido convertir el Boundary en geometría y almacenar esa en la BD, dado que se estaría almacenando la geometría dos veces. Además, supondría la conversión de datos discretos a continuos perdiendo una de las ventajas de los DGGS.

Todo esto suponía pasar de id de celda rHEALPix a coordenadas lon/lat (EPSG:4326) y viceversa. Eso es necesario para poder asignar los bounding boxes de los Boundaries en la BD y así poder hacer consultas espaciales de los mismos.

Tercera fase

Una vez resuelto el problema de pasar de id de celda rHEALPix a coordenadas lon/lat (EPSG:4326), se comenzó a implementar el modelo, objetivo de esta tercera fase. Se diseñó e implementó una biblioteca Python para importar, manipular, recuperar y dar persistencia a datos sobre el modelo diseñado y en la BD elegida. En un principio, todo este trabajo se centró en los Boundaries y Boundary Datasets, y su respectivo almacén, ya que era lo que iba a ser utilizado en el proyecto de los investigadores, COLABOTUR.

Se implementaron en Python cada uno de los conceptos del modelo ya presentado, diseñando y ejecutando a su vez un conjunto de pruebas unitarias para cada uno de ellos.

Cuarta fase

En esta cuarta fase, se continuó refinando la implementación del modelo, centrándose esta vez en los Cell Datasets. Además, se diseñó e implementó la API web que da el soporte de back-end al cliente web que estaba siendo desarrollado por los otros investigadores.

Además, se recibieron conjuntos de datos reales de los investigadores GEOT del Dpto. de Geografía y Ord. del Territorio. Esto permitió empezar a hacer pruebas con datos reales. Almacenarlos y recuperarlos, y descubrir en el proceso problemas o necesidades inesperados que pudiesen surgir.

Quinta fase

Esta quinta fase se centró en la implementación de la parte de la biblioteca dedicada a la transformación de los datos basados en el modelo DGGS hacia formatos más comunes de información geográfica como el modelo vectorial o ráster, y viceversa. En concreto, los formatos escogidos fueron Shapefile y GeoTIFF. Se realizaron pruebas con datos reales para comprobar la correcta transformación de estos.

Además, durante esta fase se colaboró en la redacción del artículo sobre COLABOTUR enviado al GEOProcessing 2020. Por otra parte, se comenzó con la redacción de la memoria del trabajo.

Sexta fase

Esta última fase, se centró en la redacción de la memoria del trabajo, así como con tareas de documentación y de corrección de errores.

4.2 Control de esfuerzos

El tiempo dedicado a cada parte del proyecto se ha registrado desde el inicio de este. Para llevar un seguimiento de los esfuerzos se ha usado la herramienta Clockify[14]. Cada tarea registrada en dicha herramienta se ha etiquetado según la fase del desarrollo a la que pertenecía: análisis y diseño, implementación, documentación o pruebas. A continuación, en la Figura 19, se muestra la distribución de horas en función de las fases del desarrollo del proyecto.

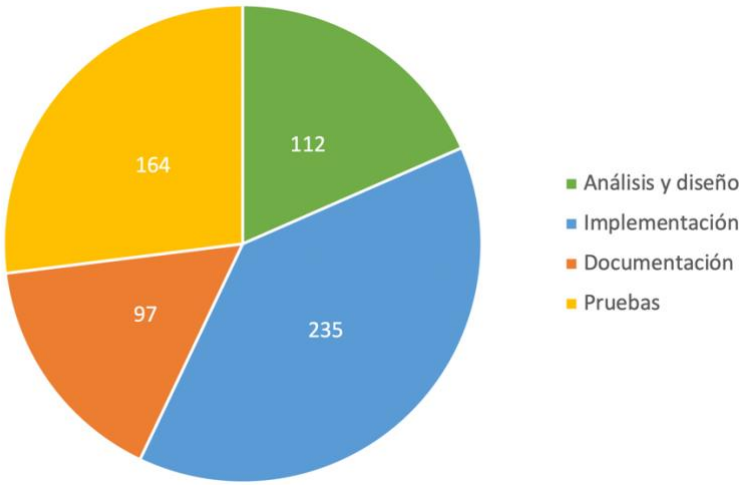

FIGURA 19 - DISTRIBUCIÓN DE HORAS

El tiempo total destinado a la realización del proyecto ha sido de 608 horas. La mayor parte del tiempo se ha invertido en la implementación de la biblioteca y API web, un 38,7%, y en las pruebas realizadas, un 27% del total. El resto del tiempo se ha invertido en el análisis y diseño, un 18,3%, y en la documentación, un 16%.

4.3 Gestión de configuraciones

Durante el trabajo se ha mantenido un control de versiones tanto del código desarrollado, como de la documentación relacionada, para ello se han utilizado las herramientas GitHub y Google Drive.

El código del sistema se encuentra en un repositorio público de GitHub[15] bajo la licencia European Union Public License, una licencia de software libre.

[14] https://clockify.me
[15] https://github.com/IAAA-Lab/grid-server

5. Conclusiones y trabajo futuro

El objetivo de este trabajo ha sido crear una biblioteca para la importación, transformación, almacenamiento, y recuperación de datos geográficos basados en un DGGS, así como la implementación de una API web que permita la integración de la funcionalidad principal en aplicaciones web o móviles.

En este trabajo se han estudiado alternativas de almacenamiento persistente (BD), se ha diseñado e implementado una biblioteca Python que permite importar, manipular, recuperar y dar persistencia a datos sobre los modelos diseñados y en la BD elegida. Se ha diseñado e implementado una API web, que permite a un cliente importar, recuperar o manipular datos basados en el modelo implementado.

La colaboración con investigadores, en concreto, con el grupo IAAA del I3A (Instituto Universitario de Investigación de Ingeniería de Aragón) y GEOT del IUCA (Instituto Universitario de Ciencias Ambientales), es otro aspecto a destacar. Esto ha permitido trabajar con necesidades, requisitos y datos reales, ya que están trabajando en un proyecto que persigue desarrollar aplicaciones para este tipo de sistemas. Estos investigadores han creado un cliente web para la captura de datos sobre DGGS, que utiliza la API web y el almacenamiento en BD diseñados e implementados en este TFG, lo que ha ayudado en su diseño y pruebas. Esta colaboración se ha traducido en la redacción de un artículo conjunto enviado al congreso internacional GEOProcessing 2020 (pendiente de revisión).

Tras la realización del trabajo, se ha visto que una de las líneas de trabajo futuro más claras podría ser realizar una implementación más eficiente en relación con la parte del almacén de Cell Datasets, “almacenamiento ráster" de los DGGS. Almacenarlos como arrays de manera eficiente (BD orientada a arrays, formatos de fichero tipo HDF5...) pero, a la hora de realizar las consultas, extraerlos como si cada celda de estos estuviera indexada por su CellID. Una de las posibilidades sería utilizar TileDB[15], motor diseñado en torno a arrays multidimensionales, siendo esta una de las alternativas para la persistencia de los datos analizadas, que, una vez finalizado este TFG, se ve posiblemente como más adecuada en este caso.

La biblioteca Python implementada permite la transformación de datos basados en un DGGS hacia formatos más comunes de información geográfica como el modelo vectorial o ráster, y viceversa, en concreto, Shapefile y GeoTIFF respectivamente. En relación con esto, otra de las líneas de trabajo futuro sería incorporar más formatos hacia los que realizar estas transformaciones.

Entre otras muchas líneas en las que se podría basar un trabajo futuro, se podría destacar la integración de este sistema en workflows de procesamiento de datos geográficos, de manera que se pudieran aprovechar las virtudes de los DGGS para, por ejemplo, automatizar alguna toma de decisiones que ahora requiere intervención humana, como, por ejemplo, a qué sistema

[15] https://tiledb.com

de coordenadas transformar datos que están en sistemas distintos cuando se quieren combinar en un análisis geográfico.

Con la realización de este trabajo se ha obtenido experiencia en el tratamiento de datos geográficos, en el desarrollo de código Python, así como en el diseño e implementación de sistemas. Se considera la colaboración con investigadores uno de los aspectos más positivos de este trabajo, de gran valor para el futuro profesional.

Referencias

1. Gibb, R. G., & Raichev, A. & Speth, M. 2016. The rHEALPix Discrete Global Grid System. https://raichev.net/files/rhealpix_dggs_preprint.pdf
2. Purss, M. (ed.) 2015. DGGS. OGC Abstract Specifications OGC 15-104r5. http://docs.opengeospatial.org/as/15-104r5/15-104r5.html
3. Víctor Olaya. 2016. Sistemas de Información Geográfica. Un libro libre de Víctor Olaya. https://volaya.github.io/libro-sig/index.html
4. Gibb, R. G. 2016. The rHEALPix Discrete Global Grid System. IOP Conference Series: Earth and Environmental Science. https://iopscience.iop.org/article/10.1088/1755-1315/34/1/012012/pdf
5. Rubén Béjar, Miguel Á. Latre, Francisco J. Lopez-Pellicer, Javier Nogueras-Iso, F. Javier Zarazaga-Soria. 2019. AGILE 2019. On the problem of providing unique identifiers for areas with any shape on Discrete Global Grid Systems. https://agile-online.org/images/conference_2019/documents/short_papers/58_Upload_your_PDF_file.pdf

Anexos

Anexo 1 – Ejemplos de uso de la biblioteca

Stores

BoundaryStore

La clase BoundaryStore representa el almacén de Boundaries con el que se da persistencia, sobre una base de datos MongoDB, a Boundary Data Sets, es decir, a conjuntos de Boundaries en los que cada Boundary tiene asociados unos datos concretos. A continuación, se presentan ejemplos de inserción de un Boundary Dataset, la consulta de un Boundary dado su identificador, el borrado de un Boundary dado su identificador, la consulta de un Boundary DataSet dado su identificador, y la consulta de un Boundary en un Boundary Dataset concreto dados sus identificadores.

```
from dggs.cellset.boundary import Boundary
from dggs.boundary_ID import BoundaryID
from dggs.dataset.boundary_dataset import BoundaryDataSet
from dggs.store.boundary_store import BoundaryStore
from dggs.dataset.data import Data

store = BoundaryStore()
# BoundaryStore por defecto utiliza la configuración de conexión a la base de
datos especificada en el archivo mongodb_config.py .

# Inserción
bds = BoundaryDataSet("id")
boundaries = ['O23P12P34S56', 'P10P11P2', 'N0', 'N8O2P0', 'O6S0S1S2', 'Q']

for boundary in boundaries:
   bds.add(Boundary(boundary_ID=BoundaryID(boundary)), Data("test"))
store.insert(bds)

# Consulta por BoundaryID
stored_boundaries =
store.query_by_boundary((Boundary(boundary_ID=BoundaryID('O23P12P34S56'))))

for boundary in stored_boundaries:
        print(boundary[0]. boundary_ID.value) # BoundaryID
        print(boundary[1].content) # Data

# >> “O23P12P34S56”
# >> “test”


# Borrado por BoundaryID
deleted_boundaries =
store.delete_boundary((Boundary(boundary_ID=BoundaryID('O23P12P34S56'))))
```

```
print(deleted_boundaries)

# >> 1

# Consulta por Boundary Dataset ID
stored_bds = store.query_by_boundary_dataset_id("id")

for bds in stored_bds:
        for boundary in bds.get_boundaries():
                print(boundary[0]. boundary_ID.value) # BoundaryID
                print(boundary[1].content) # Data
# >> “P10P11P2”
# >> “test”

# >> “N0”
# >> “test”

# >> “N8O2P0”
# >> “test”

# >> “O6S0S1S2”
# >> “test”

# >> “Q”
# >> “test”


# Consulta por BoundaryID  en Boundary Dataset concreto
stored_bds = store.query_by_boundary_in_boundary_datasets("id",
(Boundary(boundary_ID=BoundaryID(' P10P11P2'))))

for bds in stored_bds:
        for boundary in bds.get_boundaries():
                print(boundary[0]. boundary_ID.value) # BoundaryID
                print(boundary[1].content) # Data
# >> “P10P11P2”
# >> “test”
```

CellStore

La clase CellStore representa el almacén de celdas con el que se da persistencia, sobre una base de datos MongoDB, a Cell Data Sets, es decir, a conjuntos de celdas en los que cada una tiene asociado unos datos concretos. A continuación, se presentan ejemplos de inserción de un Cell Dataset, la consulta de una celda dado su identificador, el borrado de una celda dado su identificador, la consulta de un Cell DataSet dado su identificador y la consulta de una celda en un Cell Dataset concreto dados sus identificadores.

```
from dggs.cell_ID import CellID
from dggs.dataset.cell_dataset import CellDataSet
from dggs.store.cell_store import CellStore
from dggs.dataset.data import Data

store = CellStore()
# CellStore por defecto utiliza la configuración de conexión a la base de
datos especificada en el archivo mongodb_config.py .

# Inserción
cds = CellDataSet("id")
cells = ['P0', 'P1', 'P2', 'P3', 'P4', 'P5']

for cell in cells:
   cds.add(CellID(cell), Data("test"))
store.insert(cds)

# Consulta por CellID
stored_cells = store.query_by_cell(CellID('P0'))
for cell in stored_cells:
        print(cell[0].value) # CellID
        print(cell[1].content) # Data

# >> “P0”
# >> “test”

# Borrado por CellID
deleted_cells = store.delete_cell(CellID('P0'))
print(deleted_cells)

# >> 1

# Consulta por Cell Dataset ID
stored_cds = store.query_by_cell_dataset_id("id")

for cds in stored_cds:
        for cell in cds.get_cells():
                print(cell[0].value) # CellID
                print(cell[1].content) # Data
# >> “P1”
# >> “test”

# >> “P2”
# >> “test”

# >> “P3”
# >> “test”

# >> “P4”
# >> “test”
```

```
# >> “P5”
# >> “test”


# Consulta por CellID  en Cell Dataset concreto
stored_cds = store.query_by_cell_in_cell_datasets("id", CellID('P1'))

for cds in stored_cds:
        for cell in cds.get_cells():
                print(cell[0].value) # CellID
                print(cell[1].content) # Data
# >> “P1”
# >> “test”
```

Transformaciones

Shapefile

A continuación, se presentan ejemplos de transformación de datos basados en un DGGS hacia un formato vectorial, en concreto, Shapefile, y viceversa. En primer lugar, se muestra directamente el uso de las operaciones de la clase que permiten estas transformaciones. Por otro lado, se muestra cómo se pueden realizar estas transformaciones utilizando la interfaz de línea de comandos (CLI).

```
from dggs.cellset.boundary import Boundary
from dggs.dataset.boundary_dataset import BoundaryDataSet
from dggs.cell_ID import CellID
from dggs.dataset.data import Data
from dggs.dggs_utils.dggs_to_shp_utils import DGGSShpUtils
from dggs.dggs_utils.shp_to_dggs_utils import ShpDGGSUtils

shp_utils = ShpDGGSUtils()
dggs_utils = DGGSShpUtils()

cells = [CellID('P22220720648'), CellID('P22220720656'),
CellID('P22220720657'),CellID('P22220720672'), CellID('P22220720680'),
CellID('P22220720681')]

boundary = Boundary(cells=cells)
data = Data("data_test")

b_dataset = BoundaryDataSet('test_id')
b_dataset.add(boundary, data)

input_file = 'input.shp'
output_file = 'output.shp'

# Shapefile a partir de un Boundary
dggs_utils.shp_file_from_boundary(boundary, output_file)

# Boundary a partir de un Shapefile
boundary, data = shp_utils.get_boundary_from_shp_file(input_file, with_ids
=True)

# Shapefiles a partir de un Boundary dataset
dggs_utils.shp_files_from_boundary_dataset(b_dataset, output_file)

# Boundary dataset a partir de Shapefiles
bds = shp_utils.get_boundary_dataset_from_shp_file('./', 'test_id', with_ids
=True, unic_data=True)
```

```
##########
## CLI ##
##########

# Help DGGS to shp
>> python dggs_to_shp_utils.py -h

#>> -i | --input= -> input json file (defining a boundary or a boundary
dataset)
#>> -t | --type= -> 0 if file defines a boundary or  1 if defines a boundary
dataset
#>> -o | --output= -> output shapefile (.shp)

# Shapefile a partir de un Boundary
>> python dggs_to_shp_utils.py --input=input.json --type=0 --
output=output.shp

# Shapefiles a partir de un Boundary dataset
>> python dggs_to_shp_utils.py --input=input.json --type=1 --
output=output.shp


# Help shp to DGGS
>> python shp_to_dggs_utils.py -h

#>> -f | --file= -> input shapefile (if you want to get a boundary)
#>> -d | --dir= -> directory with shapefiles (if you want to get a boundary
dataset)
#>> --id -> Boundary Dataset identifier
#>> --with_ids -> if in the shapefile there is an id property that indicates
the identifier of the cells
#>> -s | --save= -> if you want to save in a file, the output file (.json)
#>> -o | --optimal= -> include AUID (Optimal Boundary)


# Boundary a partir de un Shapefile
>> python shp_to_dggs_utils.py --file=input.shp --with_ids --save=output.json
--optimal

# Boundary Dataset a partir de Shapefiles
>> python shp_to_dggs_utils.py --dir=input_dir --with_ids --save=output.json
--optimal --id=tests
```

GeoTiff

A continuación, se presentan ejemplos de transformación de datos basados en un DGGS hacia un formato ráster, en concreto, GeoTiff, y viceversa. En primer lugar, se muestra directamente el uso de las operaciones de la clase que permiten estas transformaciones. Por otro lado, se muestra cómo se pueden realizar estas transformaciones utilizando la interfaz de línea de comandos (CLI).

```
from dggs.dggs_utils.dggs_to_tif_utils import DGGSTifUtils
from dggs.dggs_utils.tif_to_dggs_utils import TifDGGSUtils

tif_utils = TifDGGSUtils()
dggs_utils = DGGSTifUtils()

input_file = 'input.tif'
output_file = 'output.tif'

cells = [CellID('N01'), CellID('N02'), CellID('N03')]
data = Data('test')

c_dataset = CellDataSet('test_id')
c_dataset.add(cells[0], data)
c_dataset.add(cells[1], data)
c_dataset.add(cells[2], data)

# GeoTiff a partir de un Cell dataset
dggs_utils.tif_file_from_cell_dataset(c_dataset, self.output_file)

# Cell dataset a partir de un GeoTiff
c_dataset = tif_utils.get_cell_dataset_from_tif_file(input_file, 'test_id')
```

```
##########
## CLI ##
##########

# Help DGGS to tif
>> python dggs_to_tif_utils.py -h

#>> -i | --input= -> input json file (defining a cell dataset)
#>> -o | --output= -> output GeoTiff (.tif)

# GeoTiff a partir de un Cell Dataset
>> python dggs_to_ tif _utils.py --input=input.json --output=output.tif


# Help tif to DGGS
```

```
>> python tif_to_dggs_utils.py -h

#>> -i | --input= -> input GeoTiff
#>> -o | --output= -> if you want to save in a file, the output file (.json)
#>> --id -> Cell Dataset identifier

# Cell Dataset a partir de un GeoTiff
>> python tif_to_dggs_utils.py --file=input.tif --output=output.json
--id=test
```

Anexo 2 – Métodos de las clases en detalle

rHEALPix

- **cell_width(Integer):** devuelve el ancho de una celda cuyo nivel de refinamiento es el pasado como parámetro.
- **rowcol(CellID):** devuelve la fila y la columna de la celda pasada como parámetro.
- **up(CellID, CellID), down(CellID, CellID), left(CellID, CellID) y right(CellID, CellID):** devuelve True si la primera Celda pasada como parámetro está más arriba/abajo/a la izquierda/derecha, que la segunda Celda.
- **get_cell_ul_vertex (CellID):** devuelve las coordenadas proyectadas (proyección rHEALPix) del vértice superior izquierdo de la Celda pasada como parámetro.
- **get_cell_projected_coordinates(CellID):** devuelve las coordenadas proyectadas (proyección rHEALPix) de los vértices y núcleo de la Celda pasada como parámetro.
- **get_cell_geodetic_coordinates(CellID):** devuelve las coordenadas geodésicas (WGS84) de los vértices y núcleo de la Celda pasada como parámetro.
- **get_geodetic_coordinates_from_bbox(bounds):** devuelve las coordenadas geodésicas (WGS84) de los vértices del bbox pasado como parámetro.
- **get_cell_from_point(Integer, point: (Float, Float)):** devuelve la Celda (CellID) que contiene el punto pasado como parámetro para el nivel de refinamiento especificado.

CellSet

- ***get_as_tree(),*** que devuelve un trie o prefix trie que representa el conjunto con los identificadores de las celdas como claves de este.
- ***get_as_grid_stack(),*** que devuelve el conjunto de celdas como una serie de Grids ordenadas por su nivel de refinamiento, es decir, un GridStack.
- ***get_min_refinement(),*** devuelve un entero que representa el mínimo refinamiento del conjunto, es decir, de las Celdas que lo componen, el refinamiento de la Celda con el menor refinamiento.
- ***get_max_refinement(),*** que devuelve un entero que representa el máximo refinamiento del conjunto, es decir, de las Celdas que lo componen, el refinamiento de la Celda con el mayor refinamiento.

BoundaryDataSet

- ***add(Boundary, Data):*** Inserta el par Boundary-Data pasado como parámetro si en el conjunto no existen datos asociados a dicho Boundary.
- ***addList(List<(Boundary, Data)>):*** Inserta la lista de pares Boundary-Data pasada como parámetro. Si en el conjunto existen datos asociados a un Boundary de la lista, no se inserta.
- ***get_boundaries(): Set<OptimalBounday>:*** devuelve el conjunto de OptimalBoundaries que forman el BoundaryDataSet.
- ***get_boundaries_and_data(): Set<(OptimalBounday, Data)>:*** devuelve el conjunto de pares OptimalBoundary-Data que forman el BoundaryDataSet.
- ***get_min_refinement():*** devuelve un entero que representa el mínimo refinamiento del conjunto.
- ***get_max_refinement():*** devuelve un entero que representa el máximo refinamiento del conjunto.
- ***get_boundary_data(BoundaryID):*** dado un identificador de un Boundary (BoundaryID), devuelve, si existe dicho Boundary en el conjunto, los datos asociados a este.
- ***get_boundary_data_list(List<BoundaryID>):*** dada una lista de identificadores de Boundaries (BoundaryID), devuelve, si existen dichos Boundaries en el conjunto, los datos asociados a estos.

CellDataSet

- ***add(CellID, Data):*** Inserta el par CellID-Data pasado como parámetro si en el conjunto no existen datos asociados a dicha Celda.
- ***addList(List<(CellID, Data)>):*** Inserta la lista de pares CellID-Data pasada como parámetro. Si en el conjunto existen datos asociados a una Celda de la lista, no se inserta.
- ***get_cells(): Set<CellID>:*** devuelve el conjunto de Celdas que forman el CellDataSet.
- ***get_cells_and_data(): Set<(CellID, Data)>:*** devuelve el conjunto de pares CellID-Data que forman el CellDataSet.
- ***get_min_refinement():*** devuelve un entero que representa el mínimo refinamiento del conjunto.
- ***get_max_refinement():*** devuelve un entero que representa el máximo refinamiento del conjunto.
- ***get_cell_data(CellID):*** dado un identificador de una Celda (CellID), devuelve, si existe dicha Celda en el conjunto, los datos asociados a esta.
- ***get_cell_data_list(List<CellID>):*** dada una lista de identificadores de Celdas (CellID), devuelve, si existen dichas Celdas en el conjunto, los datos asociados a estas.

BoundaryStore

- **insert(BoundaryDataSet):** operación de inserción de un BoundaryDataSet en el almacén. Tiene como parámetro el BoundaryDataSet a insertar.
- **all_boundaries():** operación que devuelve una lista de tuplas (OptimalBoundary, Data) con todos los Boundaries almacenados y sus datos asociados.
- **query_by_boundary(Boundary):** devuelve una lista de tuplas (OptimalBoundary, Data) con los Boundaries almacenados con identificador igual al del Boundary pasado como parámetro y sus datos asociados.
- **query_by_polygon(Polygon):** devuelve una lista de tuplas (OptimalBoundary, Data) con los Boundaries almacenados cuyo bbox intersecta con el polígono pasado como parámetro y sus datos asociados. El polígono pasado como parámetro es una lista de vértices tal y como especifica el tipo Polygon de MongoDB.
- **delete_boundary(Boundary):** elimina todos los Boundaries almacenados con identificador igual al del Boundary pasado como parámetro y sus datos asociados.
- **all_boundary_datasets():** devuelve una lista de BoundayDataSets con todos los conjuntos de Boundaries almacenados y sus datos.
- **query_by_boundary_to_boundary_datasets(Boundary):** devuelve una lista de BoundayDataSets con todos los conjuntos de Boundaries almacenados en los que existe un Boundary con identificador igual al del Boundary pasado como parámetro.
- **query_by_boundary_dataset_id(id: String):** devuelve, si existe, el BoundayDataSet almacenado cuyo identificador es igual al pasado como parámetro.
- **query_by_boundary_in_boundary_datasets(id: String, Boundary):** devuelve, si existe, el Boundary y sus datos asociados con identificador igual al del Boundary pasado como parámetro, en el BoundayDataSet almacenado cuyo identificador es igual al pasado como parámetro.
- **update_boundary_dataset(BoundaryDataSet):** reemplaza, si existe, el BoundaryDataSet almacenado cuyo identificador es igual al del BoundaryDataSet pasado como parámetro, por el BoundaryDataSet pasado como parámetro.
- **update_boundary_in_boundary_dataset(id: String, Boundary, Data):** actualiza, si existen, los datos asociados al Boundary con identificador igual al del Boundary pasado como parámetro, en el BoundayDataSet almacenado cuyo identificador es igual al pasado como parámetro.
- **delete_boundary_dataset(id: String):** elimina, si existe, el BoundayDataSet almacenado cuyo identificador es igual al pasado como parámetro.
- **delete_boundary_in_boundary_dataset(id: String, Boundary):** elimina, si existen, los datos y el Boundary con identificador igual al del Boundary pasado como parámetro,

en el BoundayDataSet almacenado cuyo identificador es igual al pasado como parámetro.

- **boundary_datasets_ids():** devuelve la lista de identificadores de todos los BoundayDataSets almacenados.
- **boundary_datasets_last_id():** devuelve el identificador del último BoundayDataSet almacenado.
- **dropAll():** elimina todos BoundayDataSets almacenados.

CellStore

- **insert(CellDataSet):** operación de inserción de un CellDataSet en el almacén. Tiene como parámetro el CellDataSet a insertar.
- **all_cells():** operación que devuelve una lista de tuplas (CellID, Data) con todas las Celdas almacenados y sus datos asociados.
- **query_by_cell(CellID):** devuelve una lista de tuplas (CellID, Data) con las Celdas almacenados con identificador al pasado como parámetro y sus datos asociados.
- **delete_cell(CellID):** elimina todas las Celdas almacenadas con identificador igual al pasado como parámetro y sus datos asociados.
- **all_cell_datasets():** devuelve una lista de CellDataSets con todos los conjuntos de Celdas almacenados y sus datos.
- **query_by_cell_to_cell_datasets(CellID):** devuelve una lista de CellDataSets con todos los conjuntos de Celdas almacenados en los que existe una Celda con identificador igual al pasado como parámetro.
- **query_by_cell_dataset_id(id: String):** devuelve, si existe, el CellDataSet almacenado cuyo identificador es igual al pasado como parámetro.
- **query_by_cell_in_cell_datasets(id: String, CellID):** devuelve, si existe, la Celda y sus datos asociados con identificador igual al pasado como parámetro, en el CellDataSet almacenado cuyo identificador es igual al pasado como parámetro.
- **update_cell_dataset(CellDataSet):** reemplaza, si existe, el CellDataSet almacenado cuyo identificador es igual al del CellDataSet pasado como parámetro, por el CellDataSet pasado como parámetro.
- **update_cell_in_cell_dataset(id: String, CellID, Data):** actualiza, si existen, los datos asociados a la Celda con identificador igual al pasado como parámetro, en el CellDataSet almacenado cuyo identificador es igual al pasado como parámetro.
- **delete_cell_dataset(id: String):** elimina, si existe, el CellDataSet almacenado cuyo identificador es igual al pasado como parámetro.
- **delete_cell_in_cell_dataset(id: String, CellID):** elimina, si existen, los datos y la Celda con identificador igual al pasado como parámetro, en el CellDataSet almacenado cuyo identificador es igual al pasado como parámetro.

- **cell_datasets_ids():** devuelve la lista de identificadores de todos los CellDataSets almacenados.
- **cell_datasets_last_id():** devuelve el identificador del último CellDataSet almacenado.
- **dropAll():** elimina todos CellDataSets almacenados.

DGGSShpUtils

- **shp_file_from_cells(List<CellID>, out_shp: String, Data):** genera un shapefile con nombre *out_shp* con una entidad formada por las celdas pasadas como parámetro a la operación. Si el parámetro de tipo Data es distinto a None, la entidad formada por el conjunto de celdas tendrá asociados dichos datos.
- **shp_file_from_boundary(Boundary, out_shp: String, Data):** genera un shapefile con nombre *out_shp* con una entidad formada por el Boundary pasado como parámetro a la operación. Si el parámetro de tipo Data es distinto a None, la entidad formada por el Boundary tendrá asociados dichos datos.
- **shp_files_from_boundary_dataset(BoundaryDataSet, out_shp: String):** genera un shapefile por cada Boundary perteneciente al BoundaryDataSet pasado como parámetro con una entidad formada por dicho Boundary y sus datos asociados.
- **shp_file_from_boundary_cli(boundary_file: String, out_shp: String):** genera un shapefile con nombre *out_shp* con una entidad formada por el Boundary descrito en formato JSON en el fichero pasado como parámetro. Esta operación se utiliza para realizar estas transformaciones a través de una interfaz de línea de comandos (CLI).
- **shp_files_from_boundary_dataset_cli(boundary_dataset_file: String, out_shp: String):** genera un shapefile por cada Boundary perteneciente al BoundaryDataSet (descrito en formato JSON en el fichero pasado como parámetro pasado como parámetro) con una entidad formada por dicho Boundary y sus datos asociados. Esta operación se utiliza para realizar estas transformaciones a través de una interfaz de línea de comandos (CLI).

ShpDGGSUtils

- **get_cells_from_shp_file(file: String, with_ids: Boolean, unic_data: Boolean):** devuelve un conjunto de Celdas a partir de los polígonos (que definen Celdas) que forman la entidad definida en el shapefile pasado como parámetro. El parámetro *with_ids* es un booleano que indica si cada polígono (Celda) definido en el shapefile contiene el identificador de dicha Celda o no, y en caso de que no lo tengan, son calculados. El parámetro *unic_data* es un booleano que indica si todos los polígonos/celdas que forman la entidad en el shapefile tiene asociados los mismos datos o cada celda tiene unos distintos.

- **get_boundary_from_shp_file(file: String, with_ids: Boolean, unic_data: Boolean):** devuelve un Boundary formado por las Celdas a partir de los polígonos (que definen Celdas) que forman la entidad definida en el shapefile pasado como parámetro. El significado de los parámetros *with_ids* y *unic_data* es el explicado anteriormente.
- **get_optimal_boundary_from_shp_file(file: String, with_ids: Boolean, unic_data: Boolean):** devuelve un OptimalBoundary formado por las Celdas a partir de los polígonos (que definen Celdas) que forman la entidad definida en el shapefile pasado como parámetro. El significado de los parámetros *with_ids* y *unic_data* es el explicado anteriormente.
- **get_boundary_dataset_from_shp_file(dir: String, id: String, with_ids: Boolean, unic_data: Boolean):** devuelve un BoundaryDataSet formado por los Boundaries y datos obtenidos a partir de los shapefiles contenidos en el directorio pasado como parámetro. El significado de los parámetros *with_ids* y *unic_data* es el explicado anteriormente. El parámetro *id* indica el identificador del BoundaryDataSet.
- **get_boundary_from_shp_file_cli(file: String, with_ids: Boolean, output_file: String, optimal: Boolean):** genera un fichero en formato JSON con nombre *output_file* o imprime por pantalla si dicho parámetro es None, con un Boundary en formato JSON formado por las Celdas a partir de los polígonos (que definen Celdas) que forman la entidad definida en el shapefile pasado como parámetro. El parámetro optimal es un booleano que indica si en el JSON de salida se debe incluir el AUID del Boundary creado. Esta operación se utiliza para realizar estas transformaciones a través de una interfaz de línea de comandos (CLI).
- **get_boundary_dataset_from_shp_file_cli(dir: String, id: String, with_ids: Boolean, output_file: String, optimal: Boolean):** genera un fichero en formato JSON con nombre *output_file* o imprime por pantalla si dicho parámetro es None, con un BoundaryDataSet en formato JSON formado por formado por los Boundaries y datos obtenidos a partir de los shapefiles contenidos en el directorio pasado como parámetro. El parámetro optimal es un booleano que indica si en el JSON de salida, cada Boundary debe incluir su AUID. El parámetro *id* indica el identificador del BoundaryDataSet. Esta operación se utiliza para realizar estas transformaciones a través de una interfaz de línea de comandos (CLI).

DGGSTifUtils

- **tif_file_from_cell_dataset(CellDataSet, out_tif: String):** genera un GeoTiff con nombre *out_tif* con un píxel por cada celda del CellDataSet pasado como parámetro y con sus datos asociados.
- **tif_file_from_cell_dataset_cli(cell_dataset_file: String, out_tif: String):** genera un GeoTiff con nombre *out_tif* con un píxel por cada celda del CellDataSet descrito en formato JSON en el fichero pasado como parámetro y con sus datos asociados.

TifDGGSUtils

- **get_cell_dataset_from_tif_file(file: String, id: String):** devuelve un CellDataSet formado por las Celdas generadas a partir de cada uno de los píxeles del fichero GeoTiff pasado como parámetro y datos asociados a cada una de ellas. El parámetro *id* indica el identificador del CellDataSet.
- **get_cell_dataset_from_tif_file_cli(file: String, id: String, output_file: String):** genera un fichero en formato JSON con nombre *output_file* o imprime por pantalla si dicho parámetro es None, con un CellDataSet en formato JSON formado por las Celdas generadas a partir de cada uno de los píxeles del fichero GeoTiff pasado como parámetro y datos asociados a cada una de ellas. El parámetro *id* indica el identificador del CellDataSet.

Anexo 3 – API Web (REST) del componente Grid Server (ver Figura 15)

API Resources (Boundary Dataset)

GET /bdatasets

Returns all BoundaryDatasets stored

Example: GET http://example.com/bdatasets

Response body:


```
[
  {
      "id": "id_1",
      "boundary_data_set": [
          {
              "AUID": "RO23$))))P12$)))34$))))S56$)))))",
              "boundary": "O23P12P34S56",
              "data": "test"
          },
          {
              "AUID": "RP10$))1$)))2$))))",
              "boundary": "P10P11P2",
              "data": "test"
          },
          {
              "AUID": "RR1$))2$))))",
              "boundary": "R1R2",
              "data": "test"
          }
```

```
      ]
  },
  {
      "id": "id_2",
      "boundary_data_set": [
          {
              "AUID": "RQ1$))2$))))",
              "boundary": "Q1Q2",
              "data": "test"
          },
          {
              "AUID": "RQ3$))4$))))",
              "boundary": "Q3Q4",
              "data": "test"
          }
      ]
  }
]
```

POST /bdatasets

Insert a BoundaryDataset

Example: POST http://example.com/bdatasets

Request body:

```
{
      "id": "id_1",
      "boundary_data_set": [
```

```
        {
            "boundary": "P1P2P3",
            "data": "test"
        },
        {
            "boundary": "Q14Q15",
            "data": "test"
        },
        {
            "boundary": "O22O23",
            "data": "test"
        }
    ]
}
```


GET /bdatasets/[bdatasets_id]

Returns the BoundaryDataset with that id.

Parameters

Parameter	Parameter Type	Description
bdatasets_id	Path	BoundaryDataset identifier

Example: GET http://example.com/bdatasets/id_1

Response body:


```
[
  {
```

```
      "id": "id_1",
      "boundary_data_set": [
          {
              "AUID": "RO23$))))P12$)))34$))))S56$)))))",
              "boundary": "O23P12P34S56",
              "data": "test"
          },
          {
              "AUID": "RP10$))1$)))2$))))",
              "boundary": "P10P11P12",
              "data": "test"
          },
          {
              "AUID": "RR1$))2$))))",
              "boundary": "R1R2",
              "data": "test"
          }
      ]
  }
]
```


GET /bdatasets/[bdatasets_id]/[boundary_id]

Returns the Boundary with that id along with the Data associated to it, in the BoundaryDataset with that id.

Parameters

Parameter	Parameter Type	Description
bdatasets_id	Path	BoundaryDataset identifier
boundary_id	Path	Boundary identifier (Cell identifier sequence)

Example: GET http://example.com/bdatasets/id_1/P10P11P2

Response body:

```
[
  {
      "id": "id_1",
      "boundary_data_set": [
          {
              "AUID": "RP10$))1$)))2$))))",
              "boundary": "P10P11P2",
              "data": "test"
          },
      ]
  }
]
```

PUT /bdatasets/[bdatasets_id]

Update the BoundaryDataset with that id.

Parameters

Parameter	Parameter Type	Description
bdatasets_id	Path	BoundaryDataset identifier

Example: PUT http://example.com/bdatasets/id_1

Request body:

```
{
      "boundary_data_set": [
          {
              "boundary": "P1P2P3",
              "data": "test"
          },
          {
              "boundary": "Q14Q15",
              "data": "test"
          },
          {
              "boundary": "O22O23",
              "data": "test"
          }
      ]
 }
```

PUT /bdatasets/[bdatasets_id]/[boundary_id]

Update the Boundary with that id, in the BoundaryDataset with that id.

Parameters

Parameter	Parameter Type	Description
bdatasets_id	Path	BoundaryDataset identifier

Parameter	Parameter Type	Description
boundary_id	Path	Boundary identifier (Cell identifier sequence)

Example: PUT http://example.com/bdatasets/id_1/P10P11P2

Request body:

```
{

    "data": "test2"

}
```

DELETE /bdatasets/[bdatasets_id]

Deletes the BoundaryDataset with that id.

Parameters

Parameter	Parameter Type	Description
bdatasets_id	Path	BoundaryDataset identifier

Example: DELETE http://example.com/bdatasets/id_1

DELETE /bdatasets/[bdatasets_id]/[boundary_id]

Deletes the Boundary with that id along with the Data associated to it, in the BoundaryDataset with that id.

Parameters

Parameter	Parameter Type	Description
bdatasets_id	Path	BoundaryDataset identifier
boundary_id	Path	Boundary identifier (Cell identifier sequence)

Example: DELETE http://example.com/bdatasets/id_1/P10P11P2

GET /boundaries

Returns all Boundaries along with the Data associated to them.

Possible parameters

Parameters referring to the vertices of a polygon can be added to filter out those Boundaries that intersect it.

Parameter	Parameter Type	Description
dlx	QueryParam	x coordinate of lower left vertex of the polygon to intersect
dly	QueryParam	y coordinate of lower left vertex of the polygon to intersect
urx	QueryParam	x coordinate of upper right vertex of the polygon to intersect
ury	QueryParam	y coordinate of upper right vertex of the polygon to intersect
Parameter	**Parameter Type**	**Description**
dlx	QueryParam	x coordinate of lower left vertex of the polygon to intersect
dly	QueryParam	y coordinate of lower left vertex of the polygon to intersect
drx	QueryParam	x coordinate of lower right vertex of the polygon to intersect
dry	QueryParam	y coordinate of lower right vertex of the polygon to intersect

Parameter	Parameter Type	Description
urx	QueryParam	x coordinate of upper right vertex of the polygon to intersect
ury	QueryParam	y coordinate of upper right vertex of the polygon to intersect
ulx	QueryParam	x coordinate of upper left vertex of the polygon to intersect
uly	QueryParam	y coordinate of upper left vertex of the polygon to intersect

Example: GET http://example.com/boundaries

Response body:


```
[
    {
        "AUID": "RO23$))))P12$)))34$))))S56$)))))",
        "boundary": "O23P12P34S56",
        "data": "test"
    },
    {
        "AUID": "RP10$))1$)))2$))))",
        "boundary": "P10P11P2",
        "data": "test"
    },
    {
        "AUID": "RR1$))2$))))",
```

```
        "boundary": "R1R2",
        "data": "test"
    },
    {
        "AUID": "RQ1$))2$))))",
        "boundary": "Q1Q2",
        "data": "test"
    },
    {
        "AUID": "RQ3$))4$))))",
        "boundary": "Q3Q4",
        "data": "test"
    }
]
```

Example: GET http://example.com/boundaries/?dlx=-179.9999997096064&dly=12.895313217732834&urx=-90.00000014160271&ury=41.93785365811587

Response body:

```
[
    {
        "AUID": "RO22$))3$)))))",
        "boundary": "O22O23",
        "data": "test"
    }
]
```

GET /boundaries/[boundary_id]

Returns the Boundary (or Boundaries if it exists in different BoundaryDatasets) with that id along with the Data associated to it.

Parameters

Parameter	Parameter Type	Description
boundary_id	Path	Boundary identifier (Cell identifier sequence)

Example: GET http://example.com/boundaries/P1P2P3

Response body:

```
[
    {
        "AUID": "RP1$))2$))3$))))",
        "boundary": "P1P2P3",
        "data": "test"
    },
]
```

DELETE /boundaries/[boundary_id]

Deletes the Boundary (or Boundaries if it exists in different BoundaryDatasets) with that id along with the Data associated to it.

Parameters

Parameter	Parameter Type	Description
boundary_id	Path	Boundary identifier (Cell identifier sequence)

Example: DELETE http://example.com/boundaries/P1P2P3

GET /idsbdatasets

Returns all BoundaryDatasets identifiers

Example: GET http://example.com/idsbdatasets

Response body:

```
[

    {

        "id": "test1",

    },

    {

        "id": "test2",

    },

    {

        "id": "test3",

    },



]
```

GET /idsbdatasets/last

Returns the identifier of the last BoundaryDataset stored

Example: GET http://example.com/idsbdatasets/last

Response body:

```
[

    {

        "id": "test3",
```

```
    }
```

```
]
```

API Resources (Cell Dataset)

GET /cdatasets

Returns all CellDatasets stored

Example: GET http://example.com/cdatasets

Response body:

```
[
  {
      "id": "id_1",
      "cell_data_set": [
          {
              "cellID": "P0",
              "data": "test"
          },
          {
              "cellID": "P1",
              "data": "test"
          },
          {
              "cellID": "P2",
              "data": "test"
```

```
            }
        ]
    },
    {
        "id": "id_2",
        "cell_data_set": [
            {
                "cellID": "S0",
                "data": "test"
            },
            {
                "cellID": "S1",
                "data": "test"
            },
        ]
    },
]
```

POST /cdatasets

Insert a CellDataset

Example: POST http://example.com/cdatasets

Request body:

```
{
        "id": "id_1",
        "cell_data_set": [
            {
```

```
            "cellID": "P0",
            "data": "test"
        },
        {
            "cellID": "P1",
            "data": "test"
        },
        {
            "cellID": "P2",
            "data": "test"
        }
    ]
},
```

GET /cdatasets/[cdatasets_id]

Returns the cellDataset with that id.

Parameters

Parameter	Parameter Type	Description
cdatasets_id	Path	CellDataset identifier

Example: GET http://example.com/cdatasets/id_1

Response body:

```
[
  {
    "id": "id_1",
    "cell_data_set": [
```

```
        {
            "cellID": "P0",
            "data": "test"
        },
        {
            "cellID": "P1",
            "data": "test"
        },
        {
            "cellID": "P2",
            "data": "test"
        }
    ]
  }
]
```


GET /cdatasets/[cdatasets_id]/[cell_id]

Returns the cell with that id along with the Data associated to it, in the cellDataset with that id.

Parameters

Parameter	Parameter Type	Description
cdatasets_id	Path	CellDataset identifier
cell_id	Path	cell identifier (Cell identifier sequence)

Example: GET http://example.com/cdatasets/id_1/P0

Response body:

```
[
  {
      "id": "id_1",
      "cell_data_set": [
          {
              "cellID": "P0",
              "data": "test"
          }
      ]
  },
]
```

PUT /cdatasets/[cdatasets_id]

Update the cellDataset with that id.

Parameters

Parameter	Parameter Type	Description
cdatasets_id	Path	CellDataset identifier

Example: PUT http://example.com/cdatasets/id_1

Request body:

```
{
      "cell_data_set": [
          {
              "cellID": "Q0",
```

```
            "data": "test"
        },
        {
            "cellID": "Q1",
            "data": "test"
        },
        {
            "cellID": "Q2",
            "data": "test"
        }
    ]
}
```

PUT /cdatasets/[cdatasets_id]/[cell_id]

Update the cell with that id, in the CellDataset with that id.

Parameters

Parameter	Parameter Type	Description
cdatasets_id	Path	CellDataset identifier
cell_id	Path	Cell identifier (Cell identifier sequence)

Example: PUT http://example.com/cdatasets/id_1/P0

Request body:

```
{
    {
```

```
          "data": "test2"

      }

}
```

DELETE /cdatasets/[cdatasets_id]

Deletes the CellDataset with that id.

Parameters

Parameter	Parameter Type	Description
cdatasets_id	Path	cellDataset identifier

Example: DELETE http://example.com/cdatasets/id_1

DELETE /cdatasets/[cdatasets_id]/[cell_id]

Deletes the cell with that id along with the Data associated to it, in the CellDataset with that id.

Parameters

Parameter	Parameter Type	Description
cdatasets_id	Path	CellDataset identifier
cell_id	Path	cell identifier (Cell identifier sequence)

Example: DELETE http://example.com/cdatasets/id_1/P0

GET /cells

Returns all cells along with the Data associated to them.

Example: GET http://example.com/cells

Response body:


```
[
    {
          "cellID": "P0",
          "data": "test"
      },
      {
          "cellID": "P1",
          "data": "test"
      },
      {
          "cellID": "P2",
          "data": "test"
      },
      {
          "cellID": "S0",
          "data": "test"
      },
      {
          "cellID": "S1",
          "data": "test"
      },
```

```
]
```

GET /cells/[cell_id]

Returns the cell (or cells if it exists in different CellDatasets) with that id along with the Data associated to it.

Parameters

Parameter	Parameter Type	Description
cell_id	Path	cell identifier (Cell identifier sequence)

Example: GET http://example.com/cells/P0

Response body:

```
[
    {
          "cellID": "P0",
          "data": "test"
     },
]
```

DELETE /cells/[cell_id]

Deletes the cell (or cells if it exists in different CellDatasets) with that id along with the Data associated to it.

Parameters

Parameter	Parameter Type	Description
cell_id	Path	cell identifier (Cell identifier sequence)

Example: DELETE http://example.com/cells/P0

GET /idscdatasets

Returns all CellDatasets identifiers

Example: GET http://example.com/idscdatasets

Response body:

```
[
    {
        "id": "test1",
    },
    {
        "id": "test2",
    },
    {
        "id": "test3",
    },

]
```

GET /idscdatasets/last

Returns the identifier of the last CellDataset stored

Example: GET http://example.com/idscdatasets/last

Response body:

```
[

    {

        "id": "test3",

    }

]
```

Índice de figuras