UNIVERSIDAD DE VALLADOLID

E.T.S.I. TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

Creación de un interfaz de usuario e integración en *GeoServer* de un módulo de *prefetching* de caché de teselas

Autor:
Dña. Alejandra Roldán Mínguez

Tutor:
D. Juan Pablo de Castro Fernández

Valladolid, 26 de junio de 2014

TÍTULO:	**Creación de un interfaz de usuario e integración en *GeoServer* de un módulo de *prefetching* de caché de teselas**
AUTOR:	**Dña. Alejandra Roldán Mínguez**
TUTOR:	**D. Juan Pablo de Castro Fernández**
DEPARTAMENTO:	**Departamento de Teoría de la Señal y Comunicaciones e Ingeniería Telemática**

TRIBUNAL

PRESIDENTE:	**D. Juan Pablo de Castro Fernández**
VOCAL:	**Dña. María Jesús Verdú Pérez**
SECRETARIO:	**Dña. Luisa María Regueras Santos**
SUPLENTE:	**D. Jaime Gómez Gil**
SUPLENTE:	**D. Francisco Merino Caminero**

FECHA:	**Julio de 2014**
CALIFICACIÓN:	

RESUMEN DEL TFG

El trabajo que aquí se presenta tiene por objetivo el desarrollo de una interfaz, integrada con la ya existente de *GeoServer*, a través de la cual poder acceder a los módulos Seeder realizados por Ricardo García, dentro del contexto de datos espaciales. Estos módulos permiten añadir a *GeoServer* la funcionalidad de *prefetch*, es decir, la posibilidad de realizar el cacheo de teselas pero haciendo uso de una estrategia diferente basada en el uso de estadísticas de acceso mediante las cuales estimar la probabilidad de que una tesela sea solicitada por un usuario para ser visualizada. El trabajo realizado da como resultado la creación de una interfaz *web* que facilitará al usuario la tarea de configurar los trabajos necesarios para cachear un conjunto concreto de teselas, y la visualización de su progreso.

ABSTRACT

The objective of the work that it's presented in this document is to develop an interface, integrated with the already exiting for GeoServer, through which to access the *Seeder* modules made by Ricardo García, within the context of spatial data. These modules allow to add to *GeoServer* the prefetch functionality, in other words, the possibility of caching tiles but using a different strategy based on the use of access statistics by which estimate the probability that a tile is going to be requested by a user for display. The work results in the creation of a web interface that facilitates the task of configuring the necessary job to cache a specific set of tiles, and display its progress.

PALABRAS CLAVE

Cacheo, capas, estadísticas, *feature*, *GeoServer*, interfaz, *Java*, *prefetch*, *Seeder*, tesela

KEYWORDS

Layer, statics, *feature*, *GeoServer*, interface, *Java*, *prefetch*, *Seeder*, tile

Agradecer a mi familia por haberme ayudado a llegar hasta aquí y a Juan Pablo de Castro por darme esta oportunidad

Contenidos

Índice de Figuras

Índice de Códigos

Capítulo 1

Introducción

1.1. Motivación

El siguiente Trabajo Fin de Grado en Ingeniería de Tecnologías de Telecomunicación detalla el desarrollo de una serie de componentes de interfaz *web* que amplía la ya existente interfaz de *GeoServer* para permitir la utilización de un conjunto de módulos llamados *Seeder-Prefetch* y *Seeder-Stats* con el fin de que el usuario pueda configurar dinámicamente el funcionamiento de los mismos.

La misión general de estos módulos es respectivamente, captar las estadísticas de acceso a las diferentes teselas de un mapa (*Seeder-Stats*) y a partir de ellas establecer estrategias para mantener en caché aquellas con mayor probabilidad de ser solicitadas por el usuario (*Seeder-Prefetch*).

La noción que se tiene hoy en día del almacenamiento en caché consiste en que aquellos elementos más recientemente accedidos se mantienen en caché, hasta que pasa un determinado tiempo o se llega a la capacidad máxima de caché, cuando se procede a vaciar esta. El módulo *Seeder-Prefetch* modifica la estrategia que se sigue para determinar qué elementos (teselas de mapa en este caso) deben guardarse en caché. Como se explicará más adelante, este módulo aprovecha los resultados del módulo *Seeder-Stats* para establecer la estrategia de cacheo.

La interfaz que se presenta a continuación permitirá al usuario decidir cómo guardar las estadísticas de acceso de cada una de las teselas definidas y configurar los trabajos de *prefetch* así como lanzarlos o detenerlos.

1.2. Objetivos

Los objetivos principales desarrollados en este trabajo fin de grado son:

- **Analizar la interfaz de *GeoServer*.** Nuestro primer objetivo va a ser analizar la programación de la interfaz *web* actual de *GeoServer* para decidir dónde situar y cómo proceder con la ampliación que deseamos realizar sobre ella.
- **Ampliar la interfaz de *GeoServer*.** Una vez que ya hemos analizado como *GeoServer* genera su interfaz *web*, ayudándonos en su estructura añadiremos nuevos apartados. Estos apartados permitirán el uso de los módulos *Seeder* de *IDELab* por parte del usuario de *GeoServer*. De manera genérica estos apartados incluirán formularios en los que el usuario configurará como deben funcionar estos módulos.
- **Enlazar las configuraciones con los módulos *Seeder*.** Una vez que ya hemos recogido de la interfaz *web* las configuraciones concretas de usuario tenemos que lanzar los trabajos correspondientes con estos datos. Para ello deberemos realizar una serie de modificaciones en los módulos de *Seeder* para que estos trabajen conforme se desea.

Capítulo 2

Estado del arte

2.1. *GeoServer*

GeoServer [1] es un servidor de código abierto escrito en *Java* que permite a los usuarios intercambiar y editar datos geoespaciales. Publica datos de las principales fuentes de datos usando estándares abiertos.

GeoServer es desarrollado, probado y respaldado por un grupo diverso de personas y organizaciones de todo el mundo.

Es la implementación de referencia de los estándares *"Open Geospatial Consortiumal"* (OGC), *"Web Feature Service"* (WFS) y *"Web Coverage Service"* (WCS), además también implementa las especificaciones *"Web Map Service"* (WMS). Hoy en día *GeoServer* es el elemento principal de la *"Geospatial Web"*.

2.1.1. Open Geospatial Consortium (OGC)

El *OGC* está formado por una serie de organizaciones tanto públicas como privadas que participan para desarrollar estándares de interfaz dentro de los Sistemas de Información Geográfica y la *World Wide Web*.

El objetivo principal de la organización consiste en servir de punto común para la colaboración entre desarrolladores y usuarios de los servicios y productos de los datos espaciales.

Entre las especificaciones publicadas por el *OGC* se pueden destacar:

- *GML – Geogrhaphic Markup Language*. Lenguaje de marcado geográfico.
- *WFS – Web Feature Service*. Servicio que permite interactuar con los mapas servidos por el estándar *WMS*.
- *WMS – Web Map Service*. Servicio que proporciona mapas de datos referenciados espacialmente.

- *WCS – Web Coverage Service*. Servicio de datos tipo *raster*. Permite realizar peticiones de cobertura geográfica.
- *CSW - Catalog Service Web*. Servicio de catálogo.

2.1.2. Características de GeoServer

GeoServer presenta multitud de características identificadoras:

- Compatible con las especificaciones *WMS*, *WFS* y *WCS*.
- Implementación de *WPS*.
- Facilidad de uso mediante interfaz *web*.
- Soporte para formatos de entrada *PostGIS*, *Shapefile*, *ArcSED*, *DB2*, *Oracle*, *VPF*, *MySQL*, *MapInfo* y *WFS* en cascada.
- Soporte para formatos de salida *JPEG*, *GIF*, *PNG*, *PDF*, *SVG* y *KML*.
- Excelente soporte para *Google Earth*.
- Habilidad para publicar información de *GeoServer* en *Google Maps*.
- Integración con *GeoWebCache*.
- Etc.

2.2. *GeoWebCache*

GeoWebCache [2] es una aplicación *web* escrita en *Java* (código abierto) utilizada para cachear teselas (*tiles*) de los mapas desde una gran variedad de orígenes como *WMS*. Implementa varios servicios de interfaz (*WMS-C*, *WMTS*, *TMS*, *Google Maps KML*, *Virtual Earth*) para acelerar y optimizar la entrega de las imágenes de los mapas.

El objetivo principal que persigue *GeoWebCache* es guardar en caché las imágenes de los mapas, o teselas, previamente solicitadas al servidor correspondiente con el objetivo de acelerar una posible nueva entrada a esa tesela. Actúa como *proxy* entre el cliente (por ejemplo *Google Maps*) y el servidor (en nuestro caso *GeoServer*). Cada vez que un cliente solicita una tesela para su visualización *GeoWebCache* intercepta esta petición y antes de ser enviada al servidor comprueba si tiene almacenada en caché la tesela solicitada para devolverla directamente (lo cual también implica mayor rapidez). En caso contrario ya se procedería a enviar la petición al servidor correspondiente. Por lo que si un usuario visita con muchas frecuencia las mismas teselas de un mapa estas se encontrarán almacenadas en caché y serán servidas sin necesidad de recurrir al servidor y por tanto con menos retraso.

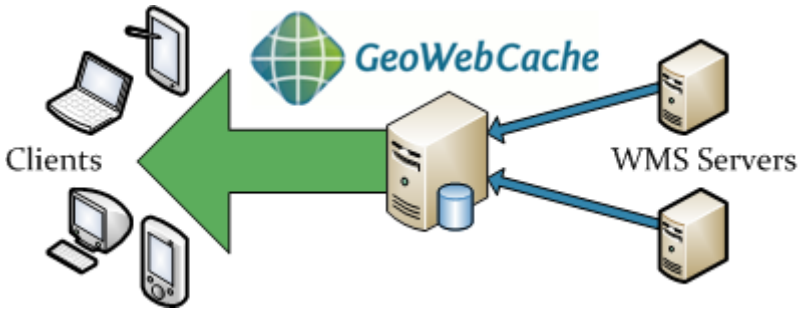

Figura 1. Esquema de funcionamiento de GeoWebCache. Información extraída de la página web oficial de GeoWebCache.

2.3. ¿Cómo gestiona *GeoServer* el cacheo de teselas?

La implementación actual del cacheo de teselas en *GeoServer* se basa en la utilización previamente mencionada de *GeoWebCache*. Podríamos considerar que el cacheo de teselas es una parte de *GeoServer* poco desarrollada hasta el momento, por ejemplo podemos notar este aspecto en cuanto que la página de la interfaz que se encarga de este apartado todavía se considera en desarrollo y tiene un aspecto pobre bastante alejado del diseño del resto de la interfaz *web*.

GeoServer almacena en el ordenador del usuario (en concreto en el directorio de datos que el usuario destine a *GeoServer* que por defecto se encuentra en la carpeta de instalación del programa y se llama *data-dir/gwc*) las teselas previamente cargadas a través de la interfaz *web*, por ejemplo, en el apartado *"Capas de teselas"* seleccionando del desplegable *"Previsualización"* de la correspondiente capa, el formato de salida de la tesela cacheada y el *GridSet* de trabajo.

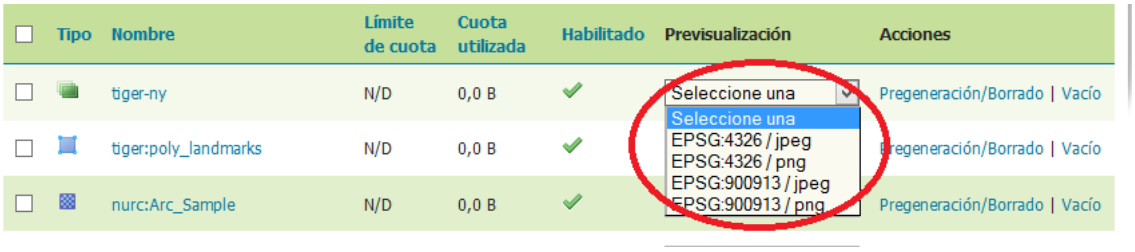

Figura 2. Ejemplo previsualización de capas

GeoServer nos abrirá una nueva página con un mapa en el que se visualizará la correspondiente capa pudiendo navegar a lo largo de ella como en cualquier otro mapa, por ejemplo en *Google Maps*. Según el usuario va navegando por el mapa desplazándose a derecha, izquierda, arriba o abajo o aumentando el *zoom*, las distintas teselas cargadas se guardan en el directorio previamente mencionado.

GeoServer también proporciona otra forma de guardar estas teselas de forma directa sin necesidad de tener que acceder al mapa y navegar por él. Se accede a esta característica mediante el enlace "*Pregeneración/Borrado"* que aparece justo al lado del desplegable antes utilizado. Este enlace nos lleva a una nueva página (la mencionada anteriormente que no está

del todo desarrollada) que permite al usuario decidir que teselas quiere generar (desde y hasta que *zoom*, dentro de que área específica del mapa (*Bounding Box*), *GridSet*, generar desde cero o solo generar las teselas que faltan).

El comportamiento que queremos añadir se podría ver como parecido a este último que implementa *GeoServer* pero con algún detalle a mayores como se explicará más adelante.

2.4. Conceptos

A continuación se describen los conceptos que se utilizarán en el desarrollo del trabajo, relacionados con datos espaciales, *GeoServer* y *GeoWebCache*.

- **Tesela**

Con este término nos referimos a cada uno de los "cuadraditos" que permiten definir un mapa completo y sobre los que realizaremos las tareas de cacheo.

Según aumentamos el nivel de *zoom* el área comprendida por las teselas en relación con el total del mapa será más pequeña pero con mayor detalle.

- **Capa**

En *GeoServer* este término hace referencia a los datos de tipo *vector* o *raster* que contienen una serie de fenómenos (*features*) geográficos. Representa cada uno de los fenómenos que deben ser mostrados en un mapa, por ejemplo una capa que defina las carreteras que existen dentro de un encuadre concreto.

En muchas ocasiones lo que nos interesa en los mapas es ver varios fenómenos simultáneamente sobre el mapa para hacernos una idea más realista. En este caso se recurre a los llamados grupos de capas, conjuntos de capas que se definen sobre la misma zona del mapa y que permiten que se puedan visualizar simultáneamente varios fenómenos, por ejemplo ver a la vez sobre el mapa información sobre carreteras y castillos.

- **Sistema Espacial de Referencia (Spatial reference system, *SRS*) y Sistema de Coordenadas de Referencia (Coordinate reference system, *CRS*)**

El Sistema Espacial de Referencia es un modelo general de referencia usado para localizar entidades geográficas. Un sistema de coordenadas de referencia define como se referencian los datos espaciales en la superficie de la Tierra.

A la hora de especificar una información geográfica es necesario especificar el *CRS* o *SRS* correspondiente, en caso contrario no se sabrá como localizar los datos en el mapa. Importante es tener en cuenta que cada *CRS* utiliza unas unidades distintas para especificar la información. Las unidades básicas son grados métricos, grados sexagesimales y radianes.

Los diferentes tipos de *SRS* se identifican usando un identificador llamado *SRID* (*Spatial Reference System*) incluyendo un código *EPSG*, por ejemplo *EPSG:4326*.

❖ Unidades

Las unidades que se utilizan en sistemas de datos espaciales son:

- Grados métricos.
- Grados sexagesimales.
- Radianes.

❖ Feature Source

Nos referimos con el nombre de *Feature Source* al origen de fenómenos geográficos, como por ejemplo pueden ser carreteras, castillos… cualquier concepto que nos permita agrupar un conjunto de fenómenos dentro de un mismo aspecto. Es decir, podríamos tener un *feature source* denominado "carreteras" o "roads" en el que almacenaríamos la información espacial de una serie de carreteras, otro *feature source* para los castillos y así sucesivamente con todos las clases de fenómenos que quisiéramos identificar.

❖ Tipos de datos

Las capas se organizan en dos tipos de datos: *raster* y *vector*, dos formatos que difieren en cómo almacenan la información espacial:

- *Vector*: la información de los *features sources* se almacena como puntos, líneas o polígonos especificados a partir de un conjunto de coordenadas XY
- *Raster*: la información de los *features sources* se representa mediante celdas. Cada celda tiene distintos valores y todas las celdas con el mismo valor representan una característica específica.

❖ *GridSet*

Un *GridSet* define un sistema de referencia espacial, el *Bounding Box*, una lista de niveles de *zoom* (resolución) y dimensiones de las teselas.

Las peticiones de teselas se realizan tomando un *GridSet* como referencia: solo las teselas que cumplen con los "requisitos" especificados en el *GridSet* (encuadre, *zoom* y dimensiones) serán almacenadas en caché.

Los *GridSet* se definen sobre un *CRS*, el cual identifica las unidades en que se medirán las coordenadas del *GridSet* (métricas, grados sexagesimales o radianes).

❖ *Bounding Box*

El *Bounding Box* (encuadre) delimita una porción concreta del mapa. Viene definido sobre un plano XY mediante cuatro coordenadas (*minX*, *maxX*, *minY*, *maxY*) que generan un cuadrado o un rectángulo. Se puede expresar en coordenadas métricas, grados sexagesimales o radianes.

Un ejemplo típico de definición de encuadre lo podemos encontrar en la declaración de las capas de *GeoServer*: a la hora de crear una nueva capa debemos determinar la extensión de la

capa. Las unidades en que se debe especificar el *Bounding Box* dependerán del *SRC* que se utilice para definir la capa.

Podemos distinguir dos tipos de encuadre:

- Nativo: límites de los datos proyectados en el *SRS* nativo.
- Lat/Lon: límites expresados en el estándar lat/lon (grados sexagesimales).

Se puede apreciar la diferencia entre ambos encuadres: el primero se expresa en las unidades del *SRS* en que definimos la capa y el segundo expresa el mismo encuadre pero en las unidades que de manera usual se usan para identificar los puntos sobre el globo terrestre.

Para nuestro nuevo módulo usaremos encuadres para limitar un área concreta del mapa sobre la que queremos realizar las tareas de cacheo según las estadísticas de acceso a las teselas que caigan en esta área.

Figura 3. Captura de imagen de la página de edición de datos de una capa en GeoServer

❖ Resolución o *zoom*

Simplemente indica el nivel de detalle con el que se está consultando un mapa. Dependiendo de la resolución las teselas contendrán información más o menos detallada. La norma general es a más resolución, más cerca se está consultando el mapa. El número total de niveles de *zoom* depende del *GridSet* en el que se haya definido el conjunto de datos consultados.

El *zoom* se puede ver de manera más sencilla como el hecho de acercar o alejar un mapa para verlo más en detalle o más en general.

- ***Buffer***

Este término se utilizará a la hora de realizar las tareas de *prefetch* y con el queremos indicar que respecto a la localización concreta de los fenómenos en el mapa, sobre cuanta porción de mapa alrededor de ellos se desea realizar también el análisis. Por ejemplo si tenemos un conjunto de carreteras e indicamos un *buffer* de 10Km significa que el análisis se realizará sobre las carreteras especificadas más un margen de 10Km hacia ambos lados de las carreteras.

Se detallará más adelante pero con este parámetro habrá que tener cuidado porque cuanto mayor sea la resolución con la que se hagan los análisis, las teselas contendrán menos kilómetros cuadrados de área y puede que los *buffers* que se especifiquen abarquen más allá de lo que abarca la propia tesela y pueda dar lugar a confusión al usuario.

El *buffer* deberá ser procesado en las mismas unidades de coordenadas que el *GridSet* que se usa para analizar el mapa, para simplificar el trabajo al usuario, a este se le pedirá el *buffer* en metros y un algoritmo lo transformará a las unidades que correspondan.

- **Formato de imagen**

Cuando el usuario mande generar un cacheo, las teselas generadas se guardarán como una imagen en el ordenador. Llegados a este punto las imágenes se pueden guardar en dos formatos: *jpeg* y *png*, permitiéndole al usuario escoger entre ambos.

Tanto en la implementación de cacheo de *GeoServer* ya explicada como en la que se añadirá con los nuevos módulos, está presente la opción de escoger entre estos dos formatos

- **Bases de datos**

Denominaremos bases de datos al lugar donde se almacenarán las estadísticas de las teselas visitadas. En *GeoServer* se le denominará almacén de datos. De entre todos los posibles tipos de bases de datos, en nuestra solución se manejarán dos: *H2* y *PostGIS*. La primera se corresponde a un tipo de base de datos embebida que está presente en la instalación de *GeoServer*. La segunda se corresponde a una extensión de la base de datos *PostgreSQL* que permite el manejo de datos espaciales. Esta última requiere de su instalación en la máquina correspondiente.

2.5. Módulos *Seeder*

Este trabajo tiene por objetivo generar una interfaz *web* que permita al usuario utilizar los módulos *Seeder* realizados por Ricardo García dentro de su trabajo de Tesis Doctoral. A continuación se indica brevemente el trabajo que realizan estos módulos.

Información extraída de la Tesis Doctoral de Ricardo García Martín (Anexo A, apartado A.3. Extensión del prototipo de caché de teselas de *GeoWebCache*).

GeoWebCache está desarrollado utilizando la herramienta de gestión de proyectos *Maven* y la herramienta de desarrollo *Spring*.

El proyecto *GeoWebCache* se compone de un proyecto padre y varios módulos hijos, a los que se han añadido estos tres nuevos módulos:

- `gwc-stats`: almacenamiento de las estadísticas de acceso a las teselas.
- `gwc-featurecatalog`: catálogo de fenómenos geográficos.
- `gwc-prefetch`: calcula la solución *OLS* que mejor ajusta los fenómenos del catálogo con las estadísticas almacenadas.

Todos ellos pertenecientes a un módulo padre llamado `gwc-idelab` que es una réplica del módulo *GeoWebCache* original, sustituyendo sus módulos hijos por sus respectivas dependencias de *Maven*, y añadiendo estos tres módulos nuevos.

En la Figura 4 se muestra la estructura general consistente en la estructura del proyecto de *GeoServer* a la que se han añadido los tres nuevos módulos definidos

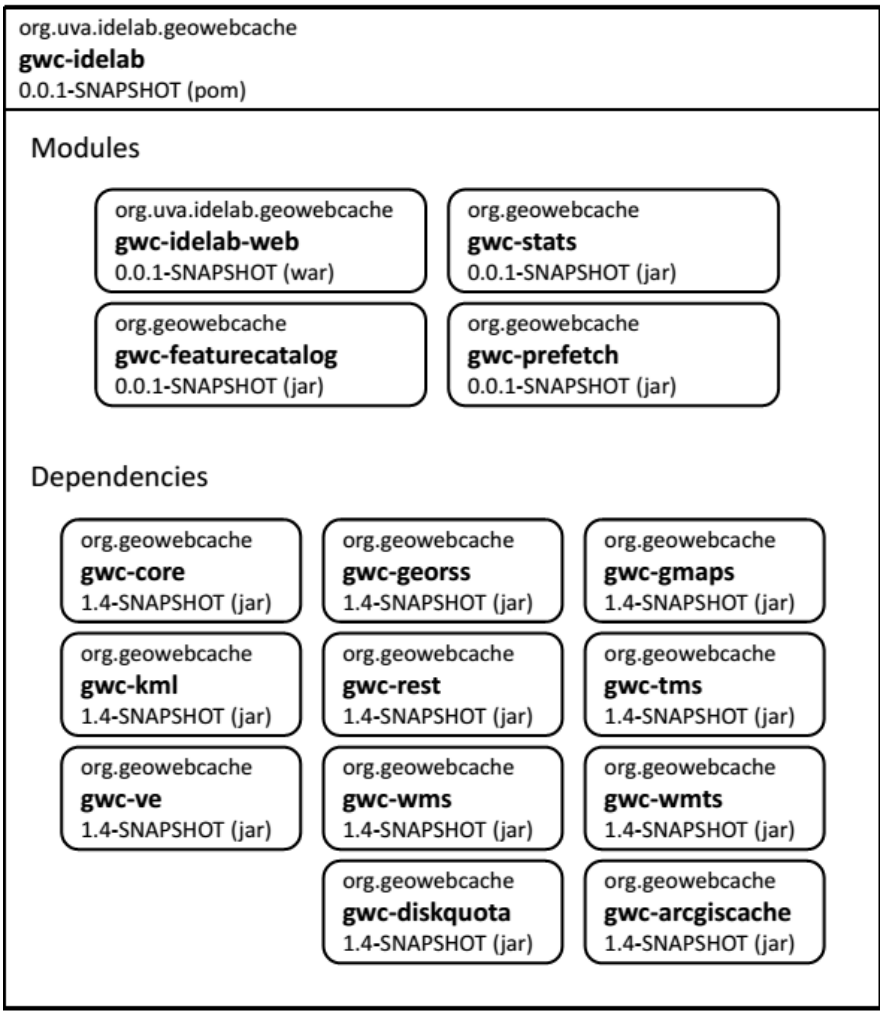

Figura 4. Organización del proyecto Maven para la solución propuesta por Ricardo García (UVa 2013)

2.5.1. Módulo `gwc-stats`

Este módulo da la posibilidad de almacenar metadatos de las teselas: tiempo de creación, espacio de almacenamiento en disco, tiempo del último acceso, tiempo de modificación y número de acceso.

En caso de estar habilitado, el componente `statsStore` asocia un escuchador (`tileStatsListener`) a las capas dadas de alta. De esta forma, cada petición recibida por una capa desencadena una llamada al escuchador indicando qué tesela ha sido solicitada. A su vez, este actualiza la base de datos en que se almacenan las estadísticas (`statsStore`), insertando la información relativa a esta tesela si es la primera vez que se pide, o actualizando su contador y tiempo de último acceso, en caso contrario. Para el almacenamiento de estadísticas se utilizan los `dataStore` de *GeoTools*. Se han implementado soluciones para utilizar como bases de datos *H2* (en memoria con soporte para índices espaciales mediante *HatBox*) y *PostgresSQL* (con la extensión espacial *PostGIS*, en este caso es necesario especificar los parámetros de acceso a la base de datos.

En el índice de estadísticas puede configurarse el rango de niveles de resolución que se quiere registrar.

2.5.2. Módulo `gwc-featurecatalog`

Este módulo ofrece un catálogo al que pueden añadirse colecciones arbitrarias de fenómenos geográficos. Las colecciones de fenómenos (*Feature Collection*) pueden obtenerse realizando una consulta a una fuente de fenómenos (*Feature Source*).

El catálogo de fenómenos consiste en un mapa que contiene las colecciones de fenómenos indexadas por nombre.

2.5.3. Módulo `gwc-prefetch`

Este módulo ofrece la funcionalidad de realizar una precarga de teselas en base al catálogo de fenómenos descrito en la sección anterior y a las estadísticas almacenadas.

Se ha desarrollado un componente `tilePrefetchingJobBean` (renombrado en la versión final del módulo tras la realización de este trabajo) en el que se configuran los parámetros de la tarea de precarga. La terna formada por el identificador de capa, *GridSet* y formato identifican la capa concreta que se desea cachear. Se puede seleccionar la zona geográfica a cachear (por defecto toda la capa) y el nivel de resolución deseado. Durante la navegación del usuario por el mapa es probable que no sólo visite las zonas atravesadas por fenómenos de interés, sino también las zonas adyacentes. Por ello, se aplica un *buffer* configurable alrededor de las geometrías.

Tanto las estadísticas de las peticiones como los fenómenos del catálogo se rasterizan usando el *GridSet* definido a la escala a cachear. Para el rasterizado de las estadísticas se utiliza el contador de peticiones, mientras que para el de los fenómenos se puede seleccionar el valor de cualquier atributo asociado al fenómeno, o un valor binario (0/1) que indica la presencia/ausencia del fenómeno en cada tesela.

La rasterización de las estadísticas da lugar a una matriz B de tamaño m x n donde m y n son el número de teselas que abarca la zona a cachear, en los ejes horizontal y vertical, respectivamente. La rasterización de las l colecciones da lugar a l matrices A_i también de tamaño m x n Mediante un algoritmo matemático explicado en la tesis doctoral antes indicada se obtiene una solución *OLS*, un vector que define unos coeficientes que se corresponden con los pesos de las distintas colecciones de fenómenos, a partir de los cuales se obtiene un valor que se asigna a una tesela y que se puede interpretar como la estimación de accesos a ella.

2.5.4. Módulo `gwc-seeder-web`

Con el actual trabajo se añadirá un nuevo módulo, `gwc-seeder-web`, en el que se definirán todos los aspectos relativos a la interfaz *web*, como se detalla en el Capítulo 4 de esta memoria

Capítulo 3

Análisis del problema

En este capítulo se analizará la necesidad de la realización de este trabajo así como los requisitos que deben satisfacerse una vez se haya completado.

3.1. Introducción

Cómo ya se ha explicado previamente, este trabajo tiene por objetivo la realización de una interfaz *web* que permita manipular y configurar dinámicamente los módulos *Seeder* de Ricardo García.

El objetivo final es añadir los módulos *Seeder* al proyecto de *GeoServer* y que estén plenamente operativos, para ello se busca ampliar la interfaz *web* de *GeoServer* para permitir el acceso a la configuración de la nueva funcionalidad.

El funcionamiento básico de los módulos de *Seeder* consta de los siguientes pasos:

- En la base de datos correspondiente se almacenan las estadísticas de acceso a las teselas de una capa (`gwc-stats`).
- Una vez recopiladas las estadísticas se lanzan los trabajos de *prefetch* (`gwc-prefetch`):
    - En primer lugar se analizan las estadísticas y los *features sources* a los que pertenece la capa y se ordenan las teselas que se deben traer a caché en función de su probabilidad de ser solicitadas al visualizar un mapa.
    - Una vez obtenida la lista ordenada de las teselas, se lanzan las tareas que se encargarán de traer a caché estas teselas en el orden en que se listan.
- Para lanzar estos trabajos se deben especificar una serie de parámetros:
    - Base de datos en la que se quiere almacenar las estadísticas.
    - *Gridset*, *Bounding box*, resolución, formato de salida de la imagen.
    - Parámetros de configuración del programador que controla la ejecución de las tareas de *prefetch*.

Hasta el momento del desarrollo de este trabajo, los módulos antes mencionados funcionaban inicializados mediante código (mediante el marco de desarrollo *Spring* de *Java* para ser más

exactos) y por lo tanto no podían ser utilizados por los usuarios de forma cómoda ya que se requeriría tener conocimientos de programación para poder modificar los parámetros de lanzamiento de las tareas, por otro lado cada vez que se necesitará hacer una modificación habría que volver a compilar el nuevo código.

Este trabajo busca desarrollar un método para poder utilizar las funcionalidades que ofrecen los módulos *Seeder* de manera dinámica, con modificaciones periódicas de los parámetros, permitiendo utilizar las mismas funcionalidades de distintas maneras de forma paralela, es decir, permitir al usuario el lanzamiento de tareas con distintos parámetros para distintas capas, en paralelo.

3.2. Requisitos del trabajo

Todo el trabajo desarrollado deberá ir destinado a ampliar la interfaz *web* de *GeoServer* a la que se añadirán una serie de secciones que permitan al usuario configurar el lanzamiento de las tereas de *prefetch* sobre una capa o grupo de capas.

Cada una de estas nuevas secciones deberá permitir al usuario:

- Configurar el/los *feature/s source/s* al que pertenece la capa/grupo de capas. Se le deberá permitir al usuario seleccionar entre capas publicadas o escoger directamente desde un *feature source* elementos sin necesidad de estar publicados.
- Seleccionar la base de datos en la que el usuario quiere guardar las estadísticas de acceso a cada capa/grupo de capas. Se le mostrará al usuario una lista con todas las bases de datos configuradas en la aplicación de *GeoServer* que sean compatibles con el módulo `gwc-stats`: *H2* y *PostGIS*. El usuario deberá poder seleccionar para capa una base de datos distinta si quisiera.
- Configurar las tareas de *prefetch* seleccionando todos los parámetros necesarios. Apartado de gran interés porque se deberá indicar al usuario toda la información que se necesita para poder lanzar los trabajos.
  Se deberá permitir al usuario escoger los valores de los parámetros de entre los posibles valores en función de los datos de publicación de la capa/grupo de capas.
  Se permitirá al usuario crear un calendario simple de lanzamiento de las tareas indicando cuantas veces quiere repetir el análisis de las estadísticas y posterior trabajo de *prefetch*, así como establecer cada cuanto tiempo realizarlo.
- Visualizar las tareas lanzadas y obtener información sobre su progreso de la forma más completa posible. Un aspecto importante es mostrar al usuario información de los posibles errores que se hayan podido producir a la hora de ejecutar las tareas de *prefetch* para que los tenga en cuenta a la hora de reintentarlo.

Todos los apartados deberán ser lo más claros posibles con indicaciones sobre posibles limitaciones o datos de interés si es posible. Además en aquellos casos en los que el usuario tenga que introducir datos se le ofrecerán valores por defecto que puedan ayudarle a rellenar el formulario.

En caso que el usuario no introduzca datos obligatorios se le deberá avisar de aquellos campos que no ha rellenado.

Si el usuario introduce datos incorrectos (valores no posibles) en alguno de los campos se le deberá avisar indicándole cual ha sido el error y que posibles valores deberá tomar esa variable. Errores típicos que el usuario se puede encontrar son:

- Introducir letras (*string*) en un campo que solo admite números o al revés.
- Introducir números cuyos valores no estén dentro de los rangos esperados, por ejemplo un número negativo cuando se espera un positivo, un valor cero cuando se necesita un valor diferente de cero.
- Cualquier otro error que pueda causar problemas en la ejecución de las tareas de *prefetch*.

También se deberá avisar al usuario de cualquier otro error que se produzca o que pueda causar problemas en la ejecución de las tareas de *prefetch*.

Siguiendo con la estructura de *GeoServer*, se deberá permitir al usuario guardar sus configuraciones, llamémoslas *Seeder*, para que puedan ser cargadas en cualquier momento incluso al reinicializar *GeoServer*, si por algún motivo este tuvo que ser parado, para que el usuario no tenga que volver a introducir todos los datos.

Un requisito que vamos a imponer es que toda esta funcionalidad solo esté disponible si para la capa/grupo de capas en cuestión el usuario ha habilitado lo que se denomina *"Cacheado de Teselas"* (*Tile Caching*) ya que las funcionalidades de esta nueva sección necesitan de la información que se configure sobre este otro apartado.

La interfaz resultante deberá guardar el mismo estilo que la *web* original de *GeoServer* y presentar un aspecto visual atractivo y cómodo para el usuario. Será preferible que se permita que el usuario rellene los campos del formulario seleccionando entre los valores posibles de las variables siempre que sea posible, mediante desplegables en los que se muestren las distintas opciones entre las que escoger.

Capítulo 4

Diseño de la solución

4.1. Introducción

Los pasos que se han seguido en el diseño de la solución han sido:

En primer lugar se ha analizado la interfaz de *GeoServer*. La interfaz *web* que se pretende diseñar debe ampliar la interfaz de *GeoServer* para añadirle las funcionalidades de los módulos *Seeder*. Por ello, el primer paso a dar fue analizar cómo se ha estructurado y diseñado la interfaz de *GeoServer* para ayudarnos de los elementos ya existentes para realizar nuestro propósito, así como mantener una estructura coherente con lo ya implementado.

Una vez sabido cómo debe ser la estructura de la interfaz, se procedió a utilizar elementos de *GeoServer* ya existentes o aprovecharlos como modelo para diseñar elementos nuevos. En este punto se generó un formulario que permite al usuario escoger la configuración que desea aplicar, así como acceder a las nuevas funcionalidades *Seeder*. Se deberá comprobar que se rellenan los campos obligatorios así como que los valores introducidos son correctos.

Por último, se realizaron las modificaciones oportunas sobre los módulos *Seeder* para adaptarlos a la interfaz y ser accedidos a través de ella.

4.2. Estructura de la interfaz *web* de *GeoServer*

Lo primero a destacar es que la interfaz de *GeoServer* no se implementa mediante código *HTML* si no mediante código *Java* y el *framework* de *Apache Wickets* [3], es decir, en un fichero *HTML* se define brevemente la estructura de la página, mediante *wickets* se hace referencia a unos objetos *Java* en los que se define qué es lo que debe aparecer en la página y que operaciones se deben hacer sobre la página.

4.2.1. Página de edición de capas/grupos de capas

Nuestro diseño requiere añadir un formulario que permita modificar las características de configuración de una capa o de un grupo de capas, por lo que el primer paso es analizar cómo se genera cada las páginas de las interfaz de *GeoServer* destinadas a la edición de los parámetros de una capa o un grupo de capas. Lo primero destacable es que el diseño de ambas páginas es distinto.

Si empezamos analizando la página de edición de capas, esta presenta el aspecto que se puede apreciar en la Figura 5.

Figura 5. Presentación de la página de edición de una capa. La información se agrupa en pestañas

La estructura de esta página consta de una serie de pestañas (*tabs*) en las que se agrupan las configuraciones a aplicar a la capa según el aspecto al que se refieren.

Vista esta configuración vamos a añadir nuestro formulario de configuración en otra pestaña independiente para separarla del resto de aspectos.

Para definir las distintas pestañas, *GeoServer* utiliza las clases denominadas `LayerEditTabPanelInfo` y `LayerEditTabPanel`. Extendiendo estas clases se añade una nueva pestaña a la página de edición de una capa, donde se debe especificar o crear el editor que se va a usar para definir el contenido de la pestaña.

En la estructura de *GeoServer* se aprecia que los editores se definen de la siguiente manera.

```
class nombreEditor extends FormComponentPanel<nombreInfo>{}
```

Donde `nombreEditor` correspondería con el nombre que le vamos a dar al editor para ser usado al llamar desde la definición de la pestaña que se usa para definir su contenido, y `nombreInfo` al nombre de la interfaz en la que se definen los datos que conforman el formulario mediante los métodos de `get()` y `set()` para obtenerlos o establecerlos y que serán usados por la aplicación para guardarlos o extraerlos de los ficheros en que se guarda la configuración. Por otro lado esa interfaz deberá tener una implementación en la correspondiente clase (con un nombre similar para relacionarlos), y también necesitará un modelo de datos. En *GeoServer* encontramos un ejemplo en los tres elementos que se

denominan, respectivamente: `GeoServerLayerInfo`, `GeoServerLayerInfoImpl` y `GeoServerLayerInfoModel`, y son el modelo de datos utilizados para implementar el formulario de la pestaña denominada *"Cacheado de teselas"* y que usaremos como base para nuestra solución.

Analizando la página de edición de un grupo de capas, en este caso, las distintas secciones de configuración aparecen todas en la misma página, como se puede apreciar en la Figura 6, agrupadas por aspecto.

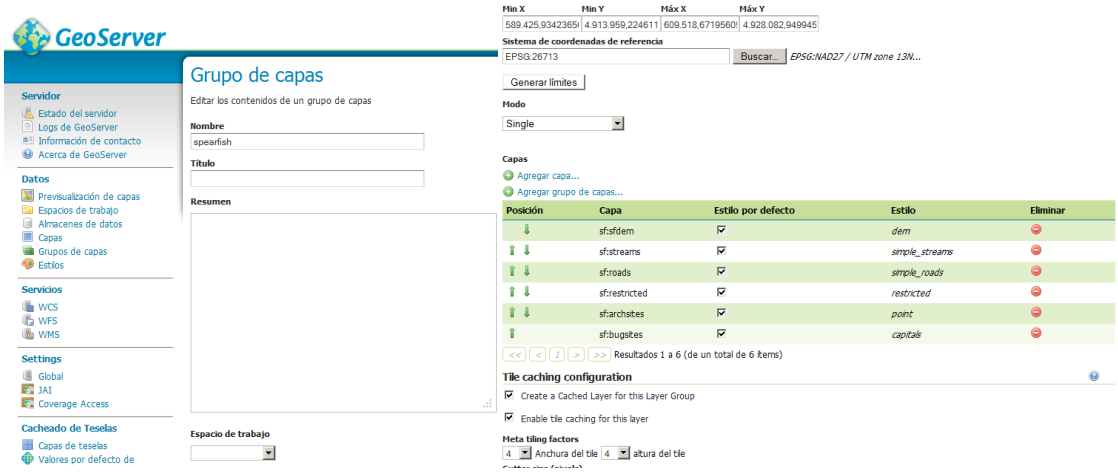

Figura 6. Ejemplos de presentación de la página de edición de un grupo de capas. La información se muestra en una misma página grupada por secciones

Para añadir las distintas secciones se hace uso de la clase abstracta `LayerGroupConfigurationPanel`, donde de igual manera que antes se define el editor a usar para completar el correspondiente panel de configuración.

En nuestro caso se verá que hemos usado el mismo editor para la página de edición de capas y grupo de capas, por lo que tanto solo hemos tenido que definir un único editor y llamarlo desde el creador de pestaña para la edición de capas y el creador del panel de configuración de la página de edición de grupos de capas. Después habrá que tener en cuenta este aspecto para tratar los datos que manejará el editor en función de que nos refiramos a una capa o un grupo de capas.

En cada uno de estos "creadores" se define el modelo de los datos a manejar por el editor, distinguiendo por un lado los datos de definición genérica de la capa o el grupo de capas y los datos concretos de la configuración *Seeder* si ya existen o si no se ha creado ninguna configuración de este tipo cargar, en caso de que existieran, los valores por defecto para inicializar el formulario con algún parámetro coherente.

Para que *GeoServer* detecte que se han definido estos nuevos elementos (pestañas o paneles de configuración) se deben inicializar los objetos correspondientes mediante *Spring* de la siguiente manera:

Para definir la pestaña de la página de edición de capas:

```
<bean id="layerCacheEditTabPanelInfo"
class="org.geoserver.gwc.web.layer.LayerEditCacheOptionsTabPanelInfo">
  <property name="id" value="layerCacheEditTabPanelInfo" />
  <property name="titleKey" value="LayerCacheOptionsTabPanel.title" />
  <property name="descriptionKey"
         value="LayerCacheOptionsTabPanel.shortDescriptio"/>
  <property name="componentClass"
         value="org.geoserver.gwc.web.layer.LayerCacheOptionsTabPanel" />
  <property name="order" value="200" />
</bean>
```

Dónde:

- En el campo "`id`" se indica el nombre que le queremos dar a la *bean* en la que definimos la pestaña.
- En el campo "`class`" especificamos la clase en la que hemos definido el modelo de datos que se maneja en la pestaña. Es una clase que debe extender la clase abstracta `LayerEditTabPanelInfo`.
- En la propiedad "`id`" se indica el nombre con el que nos referiremos a la pestaña. Se suele escoger igual al nombre de la *bean*.
- Se debe especificar el título (propiedad "`titleKey`") que va a presentar la pestaña al usuario y su descripción (propiedad "`descriptionKey`"), bien de manera directa en el fichero de *beans* o bien haciendo referencia un fichero *properties* y especificando aquí el nombre de la propiedad que almacena el título o descripción.
- En la propiedad "`componentClass`" se especifica el nombre de la clase en la que hemos definido la pestaña, debe extender la clase `LayerEditTabPanel`.
- En la propiedad "`order`" se especifica el orden de preferencia a la hora de mostrar la pestaña, es decir, cómo ordenarla en relación al resto de pestañas, en nuestro caso queremos que aparezca la última, por lo que habrá que poner un número mayor al número que presente cómo orden la última pestaña de *GeoServer*.

Para definir el panel de configuración de la página de edición de un grupo de capas:

```
<bean id="layerGroupCacheEditPanelInfo"
class="org.geoserver.web.publish.LayerGroupConfigurationPanelInfo">
  <property name="id" value="layerGroupCacheEditTabPanelInfo" />
  <property name="titleKey" value="LayerGroupCacheOptionsPanel.title" />
  <property name="descriptionKey"
       value="LayerGroupCacheOptionsPanel.shortDescription" />
  <property name="componentClass"
         value="org.geoserver.gwc.web.layer.LayerGroupCacheOptionsPanel"/>
</bean>
```

Dónde:

- En el campo "`id`" se indica el nombre que le damos a la *bean*.
- En la propiedad "`id`" indicamos el nombre con el que se va a conocer el panel dentro de la aplicación.
- Se debe especificar el título (propiedad "`titleKey`")que va a presentar la pestaña al usuario y su descripción (propiedad "`descriptionKey`"), bien de manera directa en el fichero de *beans* o bien haciendo referencia un fichero *properties* y especificando aquí el nombre de la propiedad que almacena el título o descripción.

- En la propiedad "`componentClass`" especificamos la clase en la que se define el nuevo panel de configuración. Debe extender la clase `LayerGroupConfigurationPanel`.

4.2.2. Estructura de *GeoWebCache*

Una vez que ya sabemos cómo tenemos que definir los nuevos componentes que queremos añadir a la interfaz, nos queda analizar cómo nos permite *GeoServer* guardar la información capturada del formulario en un fichero y después recuperarla de él para mostrarla en el formulario rellenado.

Se observa que los datos que después se guardarán o recuperarán de fichero son aquellos que se han definido en la interfaz anteriormente comentada. Cuando el editor definido hereda de la clase `FormComponentPanel<nombreInterfaz>`, implica que los datos que el formulario va a manejar son los que se definen en esa interfaz (mediante métodos `get()` y `set()`).

Lo siguiente observado es que la funcionalidad que se pretende añadir se asemeja a la que presenta *GeoWebCache* dentro de *GeoServer*, ya que presenta al usuario un formulario en la página de edición de capas/grupo de capas (sección *"Cacheado de teselas"*) y permite guardar la información introducida en un fichero, cargarla y borrarla. Por lo que el siguiente paso va a ser analizar como esa estructura permite manejar estas tareas.

El elemento principal es un objeto llamado `GWC` (inicializado mediante *Spring*) que representa el núcleo central de la aplicación *GeoWebCache* sobre *GeoServe*r, objeto que se encarga en última instancia de todas las tareas relacionadas con *GeoWebCache*.

Por otro lado, se identifican tres elementos (`GWCConfig`, `GWCConfigPersister` y `GWCInitializer`, inicializados por *Spring*) que permiten manejar los valores por defecto de los formularios.

El primero de estos elementos permite definir los valores por defecto de los datos del formulario. El segundo permite cargar y guardar esa información por defecto en un fichero, especificado en el mismo elemento. El último elemento, por un lado crea un directorio (cuyo nombre se especifica en el propio elemento) donde almacenar los ficheros de configuración *GeoWebCache*. Por otro lado, permite inicializar la configuración *GeoWebCache* de cada una de las capas/grupos de capas siempre y cuando exista el fichero definido por el elemento `GWCConfig`.

Estos tres elementos solo entran en funcionamiento cuando se inicializa *GeoServer*. Primero se comprueba si no existe el fichero de valor por defecto, si no existe se crea, en caso contrario se deduce que ya se ha inicializado este apartado y no se realiza ninguna acción más. Si el fichero no existe, aparte de crearlo, también crea el directorio donde almacenar las configuraciones concretas para cada capa/grupo de capas y a continuación se crea un fichero asociado a cada capa/grupo de capas con los valores de configuración por defecto.

Los siguientes elementos permiten realizar operaciones de manipulación sobre los datos del formulario para guardarlos en los ficheros correspondientes (creando los ficheros si no existen

o modificando la información ya existente), borrar los ficheros, cargar los datos desde fichero al formulario y más operaciones para permitir el trabajo de *GeoWebCache*:

- Interfaz `TileLayerCatalog` y su implementación `DefaultTileLayerCatalog`.
- Clase `GeoServerTileLayer`.
- Clase `LegacyTileLayerInfoLoader`.
- Y los tres elementos comentados anteriormente: `GeoServerLayerInfo`, `GeoServerLayerInfoImpl` y `GeoServerLayerInfoModel`.

Se puede observar como todos estos elementos comparten la palabra "*tile*" en su nombre, debido a que la estructura que se está analizando maneja información para la configuración de teselas (*tiles*). *GeoServer* manejará la información de ficheros como un catálogo compuesto por una serie de *"tiles"* cada una con una configuración especifica.

4.2.3. Añadir elementos a las páginas

Una vez analizado cómo podemos añadir los formularios para que el usuario cree su configuración y cómo manejar esa información dentro del catálogo de configuración, nos queda ver cómo realizar la presentación del formulario, es decir, como podemos completar el editor con elementos que guarden un aspecto visual parecido con el resto de la página de *GeoServer*.

Para ello se ha ido analizando los distintos tipos de paneles, cuadros, listas… que aparecen en el resto de páginas de la interfaz y se ha buscado la implementación en código de los elementos que después necesitaremos en nuestra parte de la interfaz. De entre todos ellos destacamos elementos básicos, como *checkbox* o cuadros de texto, que se implementan de manera sencilla con unas pocas líneas de código, o paneles más complejos, como tablas con columnas de detalle en las que se pueden añadir elementos, quitarlos u ordenarlos, implementados en ficheros *Java* fuera del editor. *GeoServer* permite definir paneles extendiendo la clase genérica `Panel` u otras más concretas si el panel va a tener una estructura muy específica.

Como se verá en la implementación de nuestra solución, se van a definir los distintos paneles extendiendo la clase `Panel` y se va a mantener la misma estructura en todos ellos:

- Definir los elementos a mostrar en el panel. Normalmente se pasarán como parámetro.
- Definir las columnas que van a formar la tabla.
- Definir cómo se debe completar cada una de las columnas para cada uno de los elementos.
- En caso de que fuera necesario, definir elementos externos a la tabla, lo más típico, enlacen que permitan añadir elementos a la tabla, por ejemplo mostrando una ventana emergente con otro panel.
- Definir como se debe añadir el nuevo elemento elegido a la tabla.

Por último recalcar que cada uno de los paneles o editores debe ir acompañado de un fichero *HTML* con el mismo nombre que el fichero *Java* donde se define la estructura del panel o editor mediante *wickets* cada uno de los elementos que hay que añadir (en el orden en que deben aparecer) indicando el id con el que se ha definido ese panel o elemento en el fichero *Java*.

4.2.4. Listener de eventos

Un elemento interesante de *GeoServer* que nos va a interesar son los *Listeners* de eventos. Los *Listeners* son objetos que heredan de la clase `CatalogListener`, en el caso que vamos a analizar, y cuyo trabajo es escuchar cada uno de los eventos que pueden ocurrir en el catálogo de *tile layers* (añadir un *tile*, borrarlo, modificarlo…) y realizar las tareas programadas para cuando ocurra alguno de estos eventos.

4.2.5. Concepto de catálogo

El módulo *GWC* consta de un catálogo de objetos que estará formado por lo que se denomina *tile layers*, es decir, los elementos de configuración de cada una de las capas relacionados con el cacheado de teselas. Cada uno de estos elementos tienen un nombre que se denomina identificador de la capa y es siempre el mismo para cada capa además de ser único. En nuestra solución también implementaremos un catálogo, que estará formado por la configuración *Seeder* (lo denominaremos objeto `GeoServerSeederLayerInfo`) asociado a cada una de las capas, y los identificadores para cada uno será el identificador relativo a la capa, y por lo tanto el mismo identificador que usamos para distinguir la configuración *tile chaching* de esa capa en el catálogo *GWC*.

4.3. Interfaz *GWC-Seeder IDELab*

Pasamos ahora a describir la estructura generada para implementar la interfaz *GWC-Seeder* de *IDELab*.

4.3.1. Estructura central

En primer lugar describiremos la estructura central que gobierna nuestra solución.

4.3.1.1. Objeto `Seeder`

Consideramos este objeto *Java* como el elemento central de la estructura de nuestra solución. Vamos a analizar sus principales características.

Su estructura es similar al objeto `GWC`. Se inicializa con *Spring* mediante el siguiente código:

```
<bean id="seederFacade"
      class="org.uva.idelab.geoserver.seeder.Seeder"
      depends-on="geoWebCacheExtensions">
      <constructor-arg ref="seederGeoServervConfigPersister" />
      <constructor-arg ref="seederConfiguration" />
      <constructor-arg ref="gwcStorageBroker" />
      <constructor-arg ref="gwcTLDispatcher" />
      <constructor-arg ref="gwcGridSetBroker" />
      <constructor-arg ref="gwcTileBreeder" />
      <constructor-arg ref="DiskQuotaMonitor" />
      <constructor-arg ref="dispatcher" />
      <constructor-arg ref="catalog" />
      <constructor-arg ref="gwcDefaultStorageFinder" />
      <constructor-arg ref="resourceLoader" />
</bean>
```

Código 1. Código para inicializar la fachada de Seeder

En él se indican los elementos que van a actuar como atributos del objeto, indicando que este objeto se inicializa mediante constructor. Son básicamente los mismos que presenta el objeto *GWC* añadiendo a mayores la referencia al objeto `seederConfiguration` que se verá más adelante.

A mayores contiene una serie de métodos que podemos clasificar en dos categorías:

- Aquellos que mantienen una estructura análoga al fichero *GWC*. Muchos de ellos no se utilizan en esta implementación, pero se ha dejado su implementación por si en un futuro hiciera falta utilizar algo similar.
- Aquellos métodos que nos permiten realizar tareas propias de la interfaz que estamos diseñando y que se irán enumerando y describiendo a medida que se necesiten junto con los ficheros *Java* que necesitan de ellos.

4.3.1.2. Objeto `SeederConfiguration`

Objeto bastante parecido al anterior en el que hemos ido agrupando una serie de funciones del código original de *GWC* que nos servían para nuestro módulo con las modificaciones pertinentes para adaptarlas a nuestros propósitos.

```
<bean id="GeoSeverSeederLayerCatalog"
class="org.uva.idelab.geoserver.seeder.DefaultSeederLayerCatalog">
      <constructor-arg ref="resourceLoader" />
      <constructor-arg ref="gwcXmlConfig" />
</bean>
```

Código 2. Código Spring para la inicialización del objeto SeederConfiguration

4.3.1.3. Inicialización del módulo

Los objetos `SeederInitializer`, `SeederConfig` y `SeederConfigPersister` permiten inicializar la estructura *Seeder*.

El primero de ellos, `SeederInitializer`, es una clase que implementa la interfaz `GeoServerInitializer`, inicializada mediante *Spring*:

```
<bean id="seederInitializer"
class="org.uva.idelab.geoserver.seeder.SeederInitializer">
	<constructor-arg ref="seederGeoServervConfigPersister" />
	<constructor-arg ref="rawCatalog" />
	<constructor-arg ref="GeoSeverSeederLayerCatalog" />
</bean>
```

Código 3. Código Spring de inicialización del objeto SeederInitializer

Crea un objeto de tipo `.xml` en el directorio de datos configurado para *GeoServer* con la configuración por defecto especificada.

El objeto `SeederConfigPersister` también se inicializa mediante *Spring*:

```
<bean id="seederGeoServervConfigPersister"
class="org.uva.idelab.geoserver.seeder.SeederConfigPersister">
	<constructor-arg ref="xstreamPersisterFactory" />
	<constructor-arg ref="resourceLoader">
		<description>
			GeoServer's Seeder resource loader to locate the
			root configuration
			directory where to store gwc-seeder-config.xml
		</description>
	</constructor-arg>
</bean>
```

Código 4. Código Spring de inicialización del objeto SeederConfigPersister

Permite cargar y guardar información del fichero previamente indicado.

El último objeto, `SeederConfig`, permite definir e inicializar los valores que conforman la configuración por defecto.

No se va analizar en detalle cada uno de estos ficheros, porque su estructura es idéntica a la original de *GWC* (objetos `GWCInitialize`, `GWCConfig` y `GWCConfigPersister`) con las adaptaciones correspondientes para su correcto funcionamiento en la nueva estructura.

Vamos a describir como es el funcionamiento básico:

En el objeto `SeederConfig` se especifican una serie de atributos y valores por defecto. Al inicializar el módulo, el objeto `SeederInitializer` busca en el directorio de datos el fichero especificado, en nuestro caso `gwc-seeder-config.xml`. Si no lo encuentra lo crea con la configuración por defecto codificada.

`SeederConfigPersister` permite cargar la información guardada en este fichero o modificarla. En la interfaz de *GeoServer* se permite al usuario modificar la configuración del cacheado de teselas y por ello en este módulo se implementa esa funcionalidad, aunque en nuestra interfaz no estará implementado.

El objetivo de este elemento es, que al inicializar el módulo (se entiende por inicializar el que no exista el fichero `gwc-seeder-config.xml`), crea este fichero y un directorio dentro del directorio de datos (objeto `DefaultSeederLayerCatalog` explicado posteriormente) y para todas las capas que tengan activo el cacheado de teselas (aprovechamos la funcionalidad ya implementada por *GWC*) crea un archivo de configuración *Seeder* con la configuración por defecto especificada en ese fichero.

Esto tiene bastante utilidad en *GWC* y el cacheado de teselas porque cuando se inicializa el módulo *GWC* crea un fichero de configuración *"Tile Caching"* para todas las capas que tengan activado el cacheo utilizando como valores por defecto la configuración del fichero `gwc-gs.xml` creado mediante la estructura anteriormente comentada.

En nuestra solución no tiene mucha utilidad este fichero porque la configuración por defecto para cada capa depende de muchos parámetros y es poco uniforme al contrario que en *GWC*, por lo que hemos optado por no introducir prácticamente parámetros como configuración por defecto.

```
<GeoServerSeederConfig>
  <version>1.0.0</version>
  <cacheLayersByDefault>true</cacheLayersByDefault>
</GeoServerSeederConfig>
```

Código 5. Ejemplo de fichero de configuración por defecto gwc-seeder-config.xml

4.3.1.3. Manejo del catálogo de Seeder

Los objetos `DefaultSeederLayerCatalog`, `LegacySeederLayerInfo` y `SeederLayerInfoUtil` permiten manejar y realizar operaciones con el catálogo *Seeder*.

Este catálogo es entendido como una colección de capas para las cuales se ha especificado una configuración *Seeder*. Es decir, que podemos ver este catálogo como el conjunto de ficheros `.xml` que definen la configuración *Seeder* concreta de cada capa.

En el objeto `DefaultSeederLayerCatalog` se define cual debe ser el directorio dentro del directorio de datos en que se debe almacenar el fichero configuración *Seeder* `.xml` de cada capa y la creación de estos ficheros al inicializar el módulo *Seeder*. Además permite modificar estos ficheros según el usuario va interactuando con la interfaz y definir alias para los parámetros. Con los alias se puede cambiar el nombre que aparecerá en el fichero para recoger el usuario y también como se le debe "dar la vuelta" al nombre de los parámetros para procesarlos.

```
1  private void configureXstream(XStream xs) {
2        xs.alias("SeederConfiguration",GeoServerSeederLayerInfoImpl.class);
3        xs.aliasAttribute(GeoServerSeederLayerInfoImpl.class,
4                    "featureCatalog", "FeatureCatalog");
5        xs.aliasAttribute(GeoServerSeederLayerInfoImpl.class,
6              "prefetchingJobConfiguration", "PrefetchingJobConfiguration");
```

```
        xs.alias("FeatureSourceEntry", AbstractFSEntry.class);
        xs.alias("PrefetchingJob", PrefetchJobConfigurationEntry.class);
        xs.aliasAttribute(PrefetchJobConfigurationEntry.class, "bbox",
                "BoundingBox");
        xs.alias("LayerEntry", LayerEntry.class);
        xs.alias("FeatureSourceEntry", FeatureSourceEntry.class);
        xs.registerLocalConverter(ReferencedEnvelope.class, "crs", new
                XStreamPersister.CRSConverter());
}
```

Código 6. Método para definir alias del objeto `DefaultSeederLayerCatalog`

```
<SeederConfiguration>
  <id>LayerInfoImpl--570ae188:124761b8d78:-7fc8</id>
  <name>sf:bugsites</name>
  <enabled>true</enabled>
  <FeatureCatalog>
    <LayerEntry>
      <layerId>LayerInfoImpl--570ae188:124761b8d78:-7fc6</layerId>
    </LayerEntry>
  </FeatureCatalog>
  <PrefetchingJobConfiguration>
    <PrefetchingJob>
      <jobName>a</jobName>
      <gridsetName>EPSG:4326</gridsetName>
      <BoundingBox>
        <minx>-103.86796131703647</minx>
        <maxx>-103.63773523234195</maxx>
        <miny>44.373938816704396</miny>
        <maxy>44.43418821380063</maxy>
        <crs
class="org.geotools.referencing.crs.DefaultGeographicCRS">GEOGCS[&quot;W
GS 84&quot;, &#xd;
  DATUM[&quot;World Geodetic System 1984&quot;, &#xd;
    SPHEROID[&quot;WGS 84&quot;, 6378137.0, 298.257223563,
AUTHORITY[&quot;EPSG&quot;,&quot;7030&quot;]], &#xd;
    AUTHORITY[&quot;EPSG&quot;,&quot;6326&quot;]], &#xd;
  PRIMEM[&quot;Greenwich&quot;, 0.0,
AUTHORITY[&quot;EPSG&quot;,&quot;8901&quot;]], &#xd;
  UNIT[&quot;degree&quot;, 0.01745329251994295], &#xd;
  AXIS[&quot;Geodetic longitude&quot;, EAST], &#xd;
  AXIS[&quot;Geodetic latitude&quot;, NORTH], &#xd;
  AUTHORITY[&quot;EPSG&quot;,&quot;4326&quot;]]</crs>
      </BoundingBox>
      <resolution>13</resolution>
      <buffer>10000.0</buffer>
      <startDelay>0</startDelay>
      <startDelayUnit>msecs</startDelayUnit>
      <repeatCount>1</repeatCount>
      <repeatIntervalUnit>msecs</repeatIntervalUnit>
      <format>image/png</format>
    </PrefetchingJob>
  </PrefetchingJobConfiguration>
  <storeId>DataStoreInfoImpl-e6ec3ec:14224d72e55:-8000</storeId>
</SeederConfiguration>
```

Código 7. Ejemplo de fichero de configuración `Seeder`

Con el código 10 y el ejemplo del código 11 observamos el funcionamiento de los alias:

- En la línea 2 del código 10 se indica que los objetos de clase `GeoServerSeederLayerInfoImpl` se deben especificar con el nombre `SeederConfiguration` como se ve en la primera línea del fichero de configuración de ejemplo de la código 11.
- En la líneas 3-4 del código 10 se indica que al atributo `featureCatalog` de los objetos de clase `GeoServerSeederLayerInfoImpl` se debe nombrar como `FeatureCatalog`, como se aprecia en la línea 5 del fichero de configuración del código 11
- De manera similar se aprecian otras definiciones de alias.

De esta manera si el usuario revisa estos ficheros de configuración verá nombres más amigables que le permitirán relacionar rápidamente la información almacenada.

Los objetos `LegacySeederLayerInfo` y `SeederLayerInfoUtil` definen otras operaciones básicas de manejo de la información almacenada en el catálogo. Se dice que manejan unidades de información `GeoServerSeederLayerInfo`, que es como vamos a denominar al conjunto de datos que definen la configuración *Seeder* que el usuario especifica para cada capa. Es decir, es la forma de nombrar a cada uno de los ficheros `.xml` del módulo *Seeder* y por tanto a cada uno de los elementos del catálogo *Seeder*.

4.3.1.4. Objeto CatalogLayerEventListenerForSeeder

Esta clase define un *listener* (“escuchador”) sobre el catálogo de capas (“*layers*”) de *GWC*. Este objeto está constantemente pendiente de los cambios que puedan acaecer en el catálogo y responder antes ellos.

Para ello se definen una serie de métodos estándar que permiten codificar que acción se debe realizar ante un determinado cambio en el catálogo. Los cambios más usuales son añadir un *tile layer* al catálogo, modificar su información o borrar el *tile layer*.

En nuestra solución solo hemos necesitado manejar el evento de borrar un *tile layer*. Como se explicará en el funcionamiento global de la interfaz, será requisito para poder generar y aplicar una configuración *Seeder* a una capa es que esta tenga guardada una configuración sobre el cacheo de teselas, es decir, que en el catálogo de *GWC* exista un *tile layer* asociado a esa capa. Por tanto si se elimina del catálogo este *tile layer* le vamos a requerir a la aplicación que también elimine el elemento *Seeder* correspondiente a la misma capa del catálogo Seeder que estamos implementando.

Este *listener* se inicializa al cargar *GeoServer* de manera automática al implementar la interfaz `CatalogListener`:

El método implementado para manejar el evento de borrado es:

```
1  public void handleRemoveEvent(CatalogRemoveEvent event) throws
2                CatalogException {
3        CatalogInfo obj = event.getSource();
4
5        if (!(obj instanceof LayerInfo || obj instanceof LayerGroupInfo)) {
6              return;
7        }
```

```
8           if (!mediator.containsLayer(obj.getId())) {
9                   return;
10          }
11          String prefixedName = null;
12
13          if (obj instanceof LayerGroupInfo) {
14                  LayerGroupInfo lgInfo = (LayerGroupInfo) obj;
15                  prefixedName = tileLayerName(lgInfo);
16          } else if (obj instanceof LayerInfo) {
17                  LayerInfo layerInfo = (LayerInfo) obj;
18                  prefixedName = tileLayerName(layerInfo);
19          }
20
21          if (null != prefixedName) {
22                  // notify the layer has been removed
23                  mediator.removeSeederFile(prefixedName);
24                  Map<String, IndexDAO> indexMap = Seeder.get().getIndexMap();
25                  indexMap.remove(prefixedName);
26                  log.log(Level.INFO, "Remove seeder information for layer: " +
27                          obj.getId());
28          }
29  }
```

Código 8. Método para manejar eventos de borrado del objeto `CatalogLayerEventListenerForSeeder`

Cuando el usuario a través de la interfaz de *GeoServer* elimina un *tile layer*, todos los *Listeners* configurados se pondrán en funcionamiento y lanzarán sus funciones denominadas `handleRemoveEvent()` pasándole el evento que ha tenido lugar.

En esta implementación saltará la función indicada en el código anterior. El evento constará del objeto sobre el que ha tenido lugar el correspondiente cambio. Este objeto será del tipo de información manejada en el catálogo.

Para nuestra definición, esta información puede ser de dos tipos en función de que la tesela sea una capa o un grupo de capas, por lo que debemos hacer una diferenciación, cómo aparece en las líneas 13-19, para obtener el objeto concreto.

Las líneas 5-10 tienen por objetivo asegurar que el objeto sobre el que ha tenido lugar el evento es de los dos tipos de objetos (capa o grupo de capas) que también tomamos como objeto de referencia en el catálogo *Seeder*, y por otro lado que esa capa también está agregada al catálogo *Seeder*, mediante la función `containsLayer()` del objeto *Seeder* la cual, pasándole un identificador, busca si existe algún elemento en el catálogo *Seeder* que tenga ese identificador (cada capa tiene un identificador único y siempre es el mismo). La estructura de este método es similar a la que presenta *GWC* por lo que no se entra en más detalle.

Una vez obtenido el nombre del *tile layer* que se ha borrado le indicamos al objeto mediador `Seeder` que borre le fichero de *Seeder* (y por tanto el elemento del catálogo *Seeder*) relacionado con el *tile layer* del mismo nombre.

Las líneas 24 y 25 están relacionadas con la interacción de la interfaz con los módulos *prefetch* y *stats* por lo que se analizará en la correspondiente sección. Brevemente indicar que cada capa tendrá asociado un objeto `IndexDAO` en el que se define la base de datos en que se guardarán y recogerán las estadísticas de acceso asociada a ella, y que solo se debería tener en cuenta si esa capa tiene guardada una configuración *Seeder*. La interfaz recoge todas las

correspondencias entre una determinada capa y su base de datos asociada y en este punto borramos esa información relativa a la capa en cuestión.

Por último indicar que al implementar la interfaz `CatalogListener`, nuestro *listener* debería implementar todos los métodos que en ella se definen para manejar los eventos básicos. De todos ellos solo nos interesa el asociado al evento de borrar, debemos implementar los restantes pero con un simple *return* para que no haga nada simplemente deje pasar el evento.

4.3.1.5. Objetos `GeoServerSeederLayerInfo` y `GeoServerSeederLayerInfoImpl`

`GeoServerSeederLayerInfo` es una interfaz y `GeoServerSeederLayerInfoImpl` es su implementación. Ambos objetos se utilizan para definir los parámetros que van a formar parte de la configuración *Seeder*.

```
1  public interface GeoServerSeederLayerInfo extends Serializable, Cloneable {
2
3          public abstract String getId();
4
5          public abstract void setId(String id);
6
7          public abstract String getName();
8
9          public abstract void setName(String name);
10
11         public abstract void setEnabled(boolean enabled);
12
13         public abstract boolean isEnabled();
14
15         public abstract GeoServerSeederLayerInfo clone();
16
17         public abstract List<AbstractFSEntry> getFeatureSource();
18
19         public abstract void setFeatureSource(List<AbstractFSEntry>
20                featureSource);
21
22         public abstract List<PrefetchJobConfigurationEntry>
23                getJobConfiguration();
24
25         public abstract void setJobConfiguration(
26                List<PrefetchJobConfigurationEntry> gridsetConfiguration);
27
28         public abstract String getStore();
29
30         public abstract void setStore(String storeId);
31 }
```

Código 9. Código de la interfaz `GeoServerSeederLayerInfo`

En la interfaz simplemente se recogen los parámetros mediante los métodos `get()` y `set()` que nos permiten, respectivamente, obtener y establecer su valor. Es obligatorio, para cada parámetro que queramos recoger en el catálogo o ficheros de configuración, especificar sus métodos `get()` y `set()` porque en caso contrario, en función de cual falte, no se podrá guardar el valor o cargarlo.

En la implementación se define en primer lugar una serie de atributos que representan los parámetros de la configuración:

```
1  public class GeoServerSeederLayerInfoImpl implements Serializable,
2  GeoServerSeederLayerInfo {
3
4          /** serialVersionUID */
5          private static final long serialVersionUID = 8277055420849712230L;
6
7          private static final Logger LOGGER =
8                  Logging.getLogger(GeoServerSeederLayerInfoImpl.class);
9
10         private String id;
11
12         private String name;
13
14         private boolean enabled;
15
16         private List<AbstractFSEntry> featureCatalog;
17
18         private List<PrefetchJobConfigurationEntry>
19                 prefetchingJobConfiguration;
20
21          private String storeId;
```

Código 10. Definición de atributos en la clase `GeoServerSeederLayerInfoImpl`

Los atributos a su vez pueden ser otros objetos.

A continuación se implementarán los métodos `get()` y `set()` para estos atributos:

```
22          public List<AbstractFSEntry> getFeatureSource() {
23                  return featureCatalog;
24          }
25
26          public void setFeatureSource(List<AbstractFSEntry> featureSource) {
27                  this.featureCatalog = featureSource;
28          }
29
30          public List<PrefetchJobConfigurationEntry> getJobConfiguration() {
31                  return prefetchingJobConfiguration;
32          }
33
34          public void setJobConfiguration(List<PrefetchJobConfigurationEntry>
35                  gridsetConfiguration) {
36                  this.prefetchingJobConfiguration = gridsetConfiguration;
37          }
38
39          public String getId() {
40                  return id;
41          }
42
43          public void setId(String id) {
44                  this.id = id;
45          }
46
47          public String getName() {
48                  return name;
49          }
50
51          public void setName(String name) {
52                  this.name = name;
53          }
```

```
54
55          public void setEnabled(boolean enabled) {
56                  this.enabled = enabled;
57          }
58
59          public boolean isEnabled() {
60                  return enabled;
61          }
62
63          public String getStore() {
64                  return storeId;
65          }
66
67          public void setStore(String storeId) {
68                  this.storeId = storeId;
69          }
```

Código 11. Definición de métodos `get()` y `set()` en la clase `GeoServerSeederLayerInfoImpl`

Según se puede apreciar en estos objetos, la información que se almacenará en el catalogo *Seeder* o lo que es lo mismo, lo que se verá en los ficheros `.xml` del directorio de datos de *GeoServer* es:

- `id`: identificador de la capa/grupo de capas. Este identificador es generado automáticamente por *GeoServer* y depende de la capa/grupo de capas. Cada capa/grupo de capas tiene asociado un identificador único, de tal manera que sabiendo solo el identificador de la capa/grupo de capas podemos obtener el resto de parámetros.
- `name`: nombre de la capa/grupo de capas.
- `enabled`: parámetro de tipo booleano. Valor `true` especificará que los datos se han validado, `false` en caso contrario.
- Lista de objetos `AbstractFSEntry`: esta lista permite especificar los distintos *Features Source* a los que pertenece la capa. Estos pueden ser de dos tipos: una capa ya publicada o un *Feature Source* (origen de fenómenos) propiamente dicho.

Se deberá tener en cuenta que las entradas de esta lista podrán ser de dos tipos, los que se acaban de enumerar, cada uno de ellos quedaría caracterizado por unos parámetros. En el catálogo (o fichero) no se almacena toda la información sobre la capa o *Feature Source* sino un identificador o algo similar que permita obtener a partir de él, el resto de la información. En el caso de capas publicadas se almacena su identificador de capa (`layerId`), mientras que para el caso de *Features Sources* se almacena el identificador del almacén (`storeId`). En ambos casos también se almacenará lo que se ha denominado `filterCQL` para realizar tareas de filtrado sobre el conjunto de fenómenos.

Vistas estas consideraciones, la forma por la que se ha optado para implementar este objeto es: se ha definido una clase abstracta `AbstractFSEntry` que recoge los parámetros comunes a ambos tipos de entradas (`filterCQL`) y los métodos comunes:

```
1  abstract public class AbstractFSEntry implements Serializable {
2          String filterCQL = "";
3
4          public abstract String getUserFriendlyName();
5
```

```
6           public String getFilterCQL() {
7                   return filterCQL;
8           }
9
10          public void setFilterCQL(String filterCQL) {
11                  this.filterCQL = filterCQL;
12          }
13
14          public abstract boolean warning();
15
16          public abstract String getEntryId();
17
18          public abstract String getfsName();
19  }
```

Código 12. AbstractFSEntry

Se han implementado dos clases (FeatureSourceEntry y LayerEntry) que extienden esta clase y que representan cada uno de los dos tipos de entradas que se manejan. En cada caso se implementan los métodos de la clase abstracta de la mejor manera para adaptarlo al tipo de datos que se maneja así como los métodos y parámetros que son propios de cada clase.

```
1   public class FeatureSourceEntry extends AbstractFSEntry {
2
3           String featureSourceName;
4           String storeId;
5
6           public FeatureSourceEntry(String stId, String fsName) {
7                   setFeatureSource(stId, fsName);
8           }
9
10          public void setFeatureSource(String stId, String fsName) {
11                  storeId = stId;
12                  featureSourceName = fsName;
13          }
14
15          public String getFeatureSource() {
16
17                  CoverageStoreInfo csInfo = GeoServerApplication.get()
18                .getCatalog().getCoverageStore(storeId);
19                  DataStoreInfo dsInfo = GeoServerApplication.get()
20                .getCatalog().getDataStore(storeId);
21
22                  String storeName = null;
23                  if (csInfo != null) {
24                          storeName = csInfo.getName();
25                  } else {
26                          storeName = dsInfo.getName();
27                  }
28                  String displayName = storeName + ":" + featureSourceName;
29                  return displayName;
30
31          }
32
33          @Override
34          public String getUserFriendlyName () {
35                  return "Feature Source: " + this.getFeatureSource();
36          }
37
38          @Override
39          public boolean warning() {
40
41                  boolean warning = false;
```

```
                if (GWC.get().getCatalog().getDataStore(storeId) != null)
        {
                        warning = false;
                } else
                        warning = true;
                return warning;

        }

        public String getEntryId() {
                return storeId;
        }

        @Override
        public String getfsName() {
                return this.featureSourceName;
        }

}
```

Código 13. FeatureSourceEntry

```
public class LayerEntry extends AbstractFSEntry {

        String layerId;
        String featureSourceName;

        public LayerEntry(PublishedInfo layer) {
                setLayer(layer);
        }

        public PublishedInfo getLayer() {
                if (layerId != null) {
                        return GeoServerApplication.get()
                       .getCatalog().getLayer(layerId);
                } else
                        return null;
        }

        public void setLayer(PublishedInfo publishedInfo) {
                layerId = publishedInfo.getId();
        }

        @Override
        public String getUserFriendlyName() {
                return "Layer: " + getLayer().prefixedName();
        }

        public boolean warning() {

                LayerInfo layerInfo = GWC.get().getCatalog()
               .getLayer(layerId);
                boolean warning = false;
                if (layerInfo.getResource() instanceof FeatureTypeInfo) {
                        warning = false;
                } else {
                        warning = true;
                }
                return warning;
        }

        public String getEntryId() {
                return layerId;
        }

        @Override
```

```
45          public String getfsName() {
46                  return "";
47          }
48
49  }
```

Código 14. LayerEntry

Los métodos `setFeatureSourceEntry()` y `setLayerEntry()` permiten inicializar un nuevo objeto del tipo correspondiente. El resto de métodos están relacionados con el manejo de estos objetos en el panel de selección de *Features Sources* del formulario de la configuración *Seeder* y se analizarán cuando se explique la estructura de este panel.

- Lista de objetos `PrefetchJobConfigurationEntry`: esta lista almacena una lista de configuraciones de trabajos de *prefetch* con todos sus parámetros. Cada trabajo de *prefetch* queda caracterizado por los siguientes parámetros:
    - `jobName`: nombre que el usuario ha dado al trabajo.
    - `gridsetName`: nombre del *Gridset* sobre el que se lanza el trabajo.
    - bbox: *Bounding Box* sobre el que se lanza el trabajo, es decir, de todo el globo terrestre, el área concreta al que se refiere el trabajo. Ver Sección 2.4. Conceptos.
    - `resolution`: *zoom* sobre el que se realiza el trabajo, es decir, cuanto de detalle se analiza. Cuanto mayor sea la resolución, más cerca se estará analizando el mapa. Ver Sección 2.4. Conceptos.
    - `buffer`: porción del mapa sobre el que se extiende el estudio. Ver Sección 2.4. Conceptos
    - `startDelay`: retardo con el que empezará el trabajo. Es decir el tiempo que transcurrirá desde que el usuario lanza el trabajo, pulsa el botón para lanzar el trabajo, hasta que se inicia.
    - `startDelayUnit`: con respecto al parámetro anterior, permite especificar las unidades de tiempo en que se mide.
    - `repeatCount`: número de veces que se va a repetir el trabajo. 0 para repetir indefinidamente, 1 para no repetir (una sola vez) y un número mayor que la unidad para un número de repeticiones concreto.
    - `repeatInterval`: intervalo de tiempo entre repeticiones en caso de que el parámetro anterior tenga un valor mayor que la unidad.
    - `repeatIntervalUnit`: unidades de tiempo en que se mide el parámetro anterior.
    - `format`: formato de las imágenes a analizar en el trabajo.

La estructura de este objeto se especifica a continuación, pero simplemente consiste en la declaración de los anteriores parámetros como atributos y los métodos `get()` y `set()` para inicializar las entradas del panel de configuración de trabajos de *prefetch* del formulario de configuración *Seeder*.

```
public class PrefetchJobConfigurationEntry implements Serializable {

        String jobName;
        String gridsetName;
        ReferencedEnvelope bbox;
```

```
int resolution;
double buffer;
int startDelay;
String startDelayUnit;
int repeatCount;
int repeatInterval;
String repeatIntervalUnit;
String format;

public PrefetchJobConfigurationEntry(String gridSubSetName) {
	this.setEntry(gridSubSetName);
}

public void setEntry(String gridsetName) {
	this.gridsetName = gridsetName;
}

public String getGridsetName() {
	return gridsetName;
}

public void setBbox(ReferencedEnvelope re) {
	this.bbox = re;
}

public ReferencedEnvelope getBbox() {
	return bbox;
}

public int getResolution() {
	return resolution;
}

public void setResolution(int resolution) {
	this.resolution = resolution;
}

public double getBuffer() {
	return buffer;
}

public void setBuffer(double buffer) {
	this.buffer = buffer;
}

public int getStartDelay() {
	return startDelay;
}

public void setStartDelay(int startDelay) {
	this.startDelay = startDelay;
}

public int getRepeatCount() {
	return repeatCount;
}

public void setRepeatCount(int repeatCount) {
	this.repeatCount = repeatCount;
}

public int getRepeatInterval() {
	return repeatInterval;
}

public void setRepeatInterval(int repeatInterval) {
```

```
                this.repeatInterval = repeatInterval;
        }

        public String getFormat() {
                return format;
        }

        public void setFormat(String format) {
                this.format = format;
        }

        public String getStartDelayUnit() {
                return startDelayUnit;
        }

        public void setStartDelayUnit(String startDelayUnit) {
                this.startDelayUnit = startDelayUnit;
        }

        public String getRepeatIntervalUnit() {
                return repeatIntervalUnit;
        }

        public void setRepeatIntervalUnit(String repeatIntervalUnit) {
                this.repeatIntervalUnit = repeatIntervalUnit;
        }

        public void setMaxX(Double maxx) {
                ReferencedEnvelope re = new ReferencedEnvelope(maxx,
                this.bbox.getMinX(), this.bbox.getMaxY(), this.bbox.getMinY(),
                this.bbox.getCoordinateReferenceSystem());
                this.bbox = re;
        }

        public void setMinX(Double = new
        ReferencedEnvelope(this.bbox.getMaxX(), minx,
                this.bbox.getMaxY(), this.bbox.getMinY(),
                this.bbox.getCoordinateReferenceSystem());
                this.bbox = re;
        }

        public void setMaxY(Double maxy) {
                ReferencedEnvelope re = new
                ReferencedEnvelope(this.bbox.getMaxX(),
                this.bbox.getMinX(), maxy, this.bbox.getMinY(),
                this.bbox.getCoordinateReferenceSystem());
                this.bbox = re;
        }

        public void setMinY(Double miny) {
                ReferencedEnvelope re = new
                ReferencedEnvelope(this.bbox.getMaxX(),
                this.bbox.getMinX(), this.bbox.getMaxY(), miny,
                this.bbox.getCoordinateReferenceSystem());
                this.bbox = re;
        }

        public String getJobName() {
                return jobName;
        }

        public void setJobName(String jobName) {
                this.jobName = jobName;
        }
}
```

Código 15. PrefetchJobConfigurationEntry

- `storeId`: identificador de la base de datos que se asocia a esta capa/grupo de capas para almacenar las estadísticas de acceso.

4.3.2. Editor del formulario

En este apartado se describe la definición del formulario que aparecerá en la interfaz de usuario.

4.3.2.1. Definición del formulario para la página de edición de capas

Siguiendo la estructura que presenta *GeoServer*, el nuevo formulario se añadirá como una nueva pestaña (*tab*) a continuación de las ya existentes. A ello contribuyen los siguientes objetos:

4.3.2.1.1. Objeto `SeederCacheOptionsTabPanel`

Este objeto se define como una contribución a la página de edición de capas para establecer las opciones, en este caso, de *Seeder Caching* en una nueva pestaña. La estructura de este objeto es análoga a la del objeto `LayerEditCacheOptionsTabPanel` del módulo `web-gwc` con alguna modificación para adaptarlo a nuestro propósito.

```
1  public class SeederCacheOptionsTabPanel extends LayerEditTabPanel {
2
3          private static final long serialVersionUID = 1L;
4
5          private GeoServerSeederLayerEditor editor;
6
7          public SeederCacheOptionsTabPanel(String id, IModel<LayerInfo>
8          layerModel, IModel<GeoServerSeederLayerInfo> seederLayerModel) {
9                  super(id, seederLayerModel);
10                if(SeederConfiguration.isLayerExposable(layerModel.getObject()))
11                {
12                        editor = new GeoServerSeederLayerEditor(
13                       "seederLayerEditor", layerModel, seederLayerModel);
14                        add(editor);
15                 } else {
16                        add(new Label("seederLayerEditor",
17                       new ResourceModel("geometryLessLabel")));
18                 }
19         }
20
21         @Override
22         public void save() {
23                if (editor != null) {
24                        editor.save();
25                }
26         }
27 }
```

Código 16. `SeederCacheOptionsTabPanel`

En él se define un nuevo editor del tipo `GeoServerSeederLayerEditor` (explicado posteriormente), al que se le da el nombre "`seederLayerEditor`" con el que hacerle referencia, y se le pasan como parámetros un modelo de datos sobre la capa que se está editando y otro modelo de datos sobre la configuración *Seeder* de esta capa. Si la capa no

admite características geométricas (lo que se denomina *geometry column*) lanza un mensaje de aviso como se indica en la líneas 16-17.

4.3.2.1.2. Objeto `SeederEditCacheOptionsTabPanelInfo`.

La estructura de este objeto es análoga a la del objeto `LayerEditCacheOptionsTabPanelInfo` del módulo `web-gwc` con alguna modificación para adaptarlo a nuestro propósito.

```
1  public class SeederEditCacheOptionsTabPanelInfo extends LayerEditTabPanelInfo
2  {
3
4          private static final long serialVersionUID = 7917940832781227130L;
5
6          @Override
7          public GeoServerSeederLayerInfoModel createOwnModel(final IModel<?
8          extends ResourceInfo> resourceModel, final IModel<LayerInfo>
9          layerModel, final boolean isNew) {
10
11                 LayerInfo layerInfo = layerModel.getObject();
12
13                 final GWC mediator = GWC.get();
14
15                 final GeoServerTileLayer tileLayer = isNew ? null :
16                                                mediator.getTileLayer(layerInfo);
17
18                 final Seeder seederInstance = Seeder.get();
19                 final SeederConfig seederConfig = seederInstance.getConfig();
20
21
22                 GeoServerSeederLayerInfo seederLayerInfo = isNew ? null :
23                 seederInstance.getSeederLayerInfo(layerInfo);
24
25                 if (isNew || seederLayerInfo == null) {
26
27                        final SeederConfig saneDefaults =
28                               seederConfig.saneConfig();
29                        seederLayerInfo =
30                        SeederLayerInfoUtil.loadOrCreate(layerInfo,
31                        saneDefaults);
32
33                 } else {
34
35                        GeoServerSeederLayerInfo info =
36                               seederInstance.getSeederLayerInfo(layerInfo);
37                        seederLayerInfo = info.clone();
38                 }
39                 return new GeoServerSeederLayerInfoModel(seederLayerInfo,
40                        isNew);
41          }
42   }
```

Código 17. `SeederEditCacheOptionsTabPanelInfo`

En primer lugar obtiene la configuración *Seeder* para esta capa del catálogo de configuración *Seeder*, es decir buscará un objeto de tipo `GeoServerSeederLayerInfo` con identificador igual al de la capa que se esté editando. Puede ser nula en caso de que no exista (líneas 25-37).

Se obtiene esta información mediante el método `getSeederLayerInfo(layerInfo)` del objeto `Seeder`:

```
public GeoServerSeederLayerInfo getSeederLayerInfo(LayerInfo
layerInfo) {
	return
	this.seederConfiguration.getTileLayerInfoByName(
			layerInfo.prefixedName());
}
```

Código 18. Método `getSeederLayerInfo()` del objeto `Seeder`

Este a su vez hace uso de un método del objeto `SeederConfiguration` para obtener la información de la capa en forma de objeto `LayerInfo` buscando a partir del nombre completo de la capa. La estructura de este método es análoga a la del método del mismo nombre del objeto `CatalogConfiguration` del módulo `GWC` y no se entra en más detalle.

Una vez obtenida del catálogo la configuración *Seeder* de esta capa se inicializa con ella una instancia de `GeoServerSeederLayerInfo` para dar valor a los distintos parámetros que forman el formulario para que el usuario pueda visualizarlos y/o modificarlos (líneas 34-36 del código 17).

En caso de que no se encuentre configuración para esta capa, es decir que no exista, se genera una configuración con valores iniciales por defecto (líneas 26-30 del código 17) a partir de la información por defecto que se establece en el objeto `SeederConfig` de manera análoga a cómo se obtiene los valores por defecto en la pestaña de *Tile Caching*.

Para que aparezca en la interfaz el *tab* que acabamos de definir, este objeto requiere ser inicializado mediante *Spring* tal como se explicó en la Sección 4.2.1.

```
<bean id="seederCacheEditTabPanelInfo"
class="org.uva.idelab.geoserver.seeder.
SeederEditCacheOptionsTabPanelInfo">
	<property name="id" value="seederCacheEditTabPanelInfo" />
	<property name="titleKey"
		value="SeederCacheOptionsTabPanel.title" />
	<property name="descriptionKey"
		value="SeederCacheOptionsTabPanel.shortDescription" />
	<property name="componentClass"
		value="org.uva.idelab.geoserver.seeder.
		SeederCacheOptionsTabPanel" />
	<property name="order" value="900" />
</bean>
```

Código 19. Inicialización objeto `SeederEditCacheOptionsTabPanelInfo` mediante *Spring*

Se completa la configuración del *tab* mediante el siguiente fichero *HTML* (`SeederCacheOptionsTabPanel.html`) en el que simplemente se define que el contenido de esta página *web* es contenido definido mediante *Java* y la tecnología *wickets* a partir del nombre que se da al elemento a mostrar (`seederLayerEditor`):

```
<html xmlns:wicket="http://wicket.apache.org/">
<head></head>
<body>
	<wicket:panel>
		<div wicket:id="seederLayerEditor"></div>
	</wicket:panel>
</body>
</html>
```

Código 20. Fichero `SeederCacheOptionsTabPanel.html`

El fichero *HTML* debe tener el mismo nombre que el fichero *Java* en que se defina la estructura del documento *web*.

4.3.2.1.3. Objeto GeoServerSeederLayerInfoModel

Permite definir el modelo de datos que se va a utilizar en el formulario. Estructura idéntica al objeto `GeoServerTileLayerInfoModel` del módulo `web-gwc`.

```
public class GeoServerSeederLayerInfoModel extends
Model<GeoServerSeederLayerInfo> {
	private static final long serialVersionUID=2246174669786551903L;

	private final boolean isNew;

	public GeoServerSeederLayerInfoModel(GeoServerSeederLayerInfo
		info, boolean isNew) {
		super(info);
			this.isNew = isNew;
	}

	public boolean isNew() {
		return isNew;
	}
}
```

Código 21. `GeoServerSeederLayerInfoModel`

4.3.2.2. Definición formulario para la página de edición de grupos de capas.

Siguiendo la estructura que presenta *GeoServer*, el nuevo formulario se añadirá como un nuevo panel a continuación de los ya existentes. A ello contribuye el siguiente fichero al que se añade el modelo de datos del formulario (Sección 4.3.2.1.3).

4.3.2.2.1. Objeto `LayerGroupSeederOptionsPanel`

Permite definir un nuevo panel a añadir a la página de edición de grupos de capas. Estructura análoga al objeto `LayerEditCacheOptionsTabPanelInfo` del módulo `web-gwc`.

```
1  public class LayerGroupSeederOptionsPanel extends
2  LayerGroupConfigurationPanel {
3
4  	private static final long serialVersionUID = -8651034825347320139L;
5
6  	private GeoServerSeederLayerEditor editor;
7
8  	public LayerGroupSeederOptionsPanel(final String id,
9  		IModel<LayerGroupInfo> layerGroupModel) throws Exception {
10 		super(id, layerGroupModel);
11
12 		final LayerGroupInfo layerGroupInfo = getLayerGroupInfo();
13 		final boolean isNew = layerGroupInfo.getId() == null;
14
15 		final Seeder seederInstance = Seeder.get();
16 		final SeederConfig seederConfig = seederInstance.getConfig();
17
18 		GeoServerSeederLayerInfo seederLayerInfo = isNew ? null :
19 			seederInstance.getSeederLayerGroupInfo(layerGroupInfo);
```

```
20                  if (isNew || seederLayerInfo == null) {
21
22                          final SeederConfig saneDefaults =
23                                seederConfig.saneConfig();
24                          seederLayerInfo =
25                                SeederLayerInfoUtil.loadOrCreate(layerGroupInfo,
26                                saneDefaults);
27                  }
28
29                  GeoServerSeederLayerInfoModel seederLayerModel = new
30                        GeoServerSeederLayerInfoModel(seederLayerInfo, isNew);
31
32                editor = new GeoServerSeederLayerEditor("seederLayerEditor",
33                        layerGroupModel, seederLayerModel);
34                  add(editor);
35          }
36
37          @Override
38          public void save() {
39                  editor.save();
40          }
41
42  }
```

Código 22. LayerGroupSeederOptionsPanel

De manera análoga a lo expuesto en la sección 4.3.2.1.2, en primer lugar se busca en el catálogo de configuraciones *Seeder* si existe una configuración para el grupo de capas en cuestión (línea 12), en caso afirmativo se inicializa el formulario con esos valores (líneas 18-19), en caso contrario se cargan los valores por defecto (líneas 21-28). A partir de estos datos se crea un modelo de datos (`GeoServerSeederLayerInfoModel`, líneas 30-31).

Para que este panel que acabamos de definir aparezca en la página de edición de grupo de capas, este objeto se debe inicializar mediante *Spring* como ya se explicó en la Sección 4.2.1:

```
<bean id="LayerGroupSeederEditPanelInfo"
class="org.geoserver.web.publish.LayerGroupConfigurationPanelInfo">
        <property name="id" value="LayerGroupSeederEditTabPanelInfo" />
        <property name="titleKey"
                value="LayerGroupSeederOptionsPanel.title" />
        <property name="descriptionKey"
                value="LayerGroupSeederOptionsPanel.shortDescription" />
        <property name="componentClass"
                value="org.uva.idelab.geoserver.seeder.
                        LayerGroupSeederOptionsPanel" />
</bean>
```

Código 23. Inicialización del objeto `LayerGroupConfigurationPanelInfo` mediante *Spring*

Se completa la configuración con el fichero *HTML* correspondiente (`LayerGroupSeederOptionsPanel.html`). Simplemente en él se define un elemento *wicket* que es el editor del formulario que se define en la siguiente sección, a través del nombre (`seederLayerEditor`) que se le ha dado al crear el editor:

```
<html xmlns:wicket="http://wicket.apache.org/">
<head></head>
<body>
        <wicket:panel>
                <div wicket:id="seederLayerEditor"></div>
        </wicket:panel>
</body></html>
```

Código 24. Fichero `LayerGroupSeederOptionsPanel.html`

El fichero *HTML* debe tener el mismo nombre que el fichero *Java* en que se defina la estructura del documento *web*.

4.3.2.3. Definición del contenido del formulario. Objeto `GeoServerSeederLayerEditor`

En este objeto se define el contenido que tendrá el editor. Este objeto extiende la clase `FormComponentPanel<GeoServerSeederLayerInfo>`. De esta manera se le indica el conjunto de datos que deberá manejar el editor, básicamente definidos a partir de los métodos `get()` y `set()` de la interfaz `GeoServerSeederLayerInfo` ya que en caso que algún dato que queramos manejar en el formulario no esté definido convenientemente con un método `get()` y `set()` el editor no sabrá cómo trabajar con él, como ya se explicó en la sección 4.3.1.5.

```
class GeoServerSeederLayerEditor extends
       FormComponentPanel<GeoServerSeederLayerInfo> {…}
```

Código 25. Extracto `GeoServerSeederLayerEditor` I

Se pueden distinguir varias partes:

Definición de los distintos elementos que aparecerán en el formulario (tanto elementos "simples" como llamadas a otros subeditores o paneles para manejar los parámetros de tipo objeto) y definición de otros parámetros no pertenecientes a la configuración pero necesarios para manejar el formulario.

```
private static final Logger LOGGER =
Logging.getLogger(GeoServerSeederLayerEditor.class);

private static final long serialVersionUID = 7870938096047218989L;

private final FormComponent<Boolean> existTileCaching;

private final FormComponent<Boolean> enabled;

private final FeatureSourcePanel featureSourcePanel;

private final PrefetchJobConfigurationPanel
                                    prefetchJobConfigurationPanel;

private final StoreStatsPanel storePanel;

private final TaskPanel taskPanel;

/**
 * Container for {@link #configs}
 */
private final WebMarkupContainer container;

/**
 * Container for everything but {@link #existTileCaching}
 */
private final WebMarkupContainer configs;

private IModel<? extends CatalogInfo> layerModel;
```

Código 26. Extracto `GeoServerSeederLayerEditor` II

¿Qué definimos con cada uno de estos parámetros?

- `existTileCaching`: nos permite habilitar la creación de la configuración *Seeder*. Un requisito para el uso de esta funcionalidad es que ya exista configuración *Tile Caching* (es decir un objeto de tipo *Tile Layer* en el catálogo *Tile Layers*). Este booleano nos va a permitir mostrar o no el formulario de configuración *Seeder* en función de que se haya generado la configuración anterior ya que alguno de los parámetros de nuestra configuración dependen de ella.
- `enabled`: nos permite habilitar la comprobación de los valores introducidos.
- `featureSourcePanel`: nos permite añadir un panel que representará el objeto de tipo `AbstractFSEntry` (Sección 4.3.1.5) que forma parte de la configuración. Es un panel implementado fuera del editor. Este panel permitirá al usuario seleccionar el *Feature Source* al que quiere hacer pertenecer la capa. En la sección correspondiente a este panel (Sección 4.3.3.1) se explicará su estructura y funcionamiento.
- `prefetchJobConfigurationPanel`: nos permite añadir un panel que representará los distintos objetos de tipo `PrefetchJobConfigurationEntry` (Sección 4.3.1.5) que formarán parte de la configuración *Seeder*. Este panel permitirá al usuario definir la configuración de los distintos trabajos de *prefetch* relativos a esta capa así como lanzarlos. En la sección correspondiente a este panel (Sección 4.3.3.2) se explicará su estructura y funcionamiento.
- `storePanel`: nos permite añadir un panel en el que el usuario seleccionará la base de datos en la que quiere almacenar las estadísticas de acceso relativas a esta capa. En la sección correspondiente a este panel (Sección 4.3.3.3) se explicara su estructura y funcionamiento.
- `taskPanel`: nos permite mostrar al usuario un panel en el que pueda visualizar el estado y más información relativa a los trabajos lanzados sobre esta capa. Contiene solo información de interés al usuario pero no la obtiene de la configuración *Seeder* de la capa, ni se guarda con ella. (Sección 4.3.3.4)
- `layerModel`: nos permite definir el modelo de datos de la capa sobre la que se está realizando la modificación.
- `container, configs`: nos van a permitir definir la interfaz *web* del formulario.

Definición de los elementos del formulario en el constructor de la clase. Este constructor necesita que se le pasen como argumentos los distintos modelos de los objetos sobre los que trabaja para manejar correctamente el formulario. Como componente de tipo “panel” necesita también un identificador que permitirá identificar al panel a la hora de definirlo en el fichero *HTML* desde el que se hace la llamada a este editor.

En este constructor se definirán cada uno de los elementos que deberán aparecer en el formulario definiendo correctamente la forma de aparición (*checkbox*, selección a partir de algún tipo de lista de posibilidades, a partir de otro panel, campos de texto simples o de autocompletado…).

Además puede definirse que unos elementos del formulario solo aparezcan si otros tienen un valor concreto usando el elemento *`WebMarkupContainer`* de *Wickets* para definir los elementos que dependen del valor del “principal” y el método `onUpdate()` que permite indicar como se tiene que actualizar la página en caso de que un determinado atributo tenga un valor concreto.

También se pueden añadir otros elementos que no pertenezcan al propio formulario, es decir, que no pertenezcan a la configuración, como ventanas emergentes mostrando mensajes de aviso (en esta interfaz está implementado a nivel de los subpaneles).

Cada uno de los elementos añadidos tendrá un `id` que permita identificarlos a la hora de instanciar la página desde el fichero *HTML*.

Analizamos paso a paso la estructura de este constructor:

- **Inicialización del constructor**. Estructura idéntica a la presentada en el editor `GeoServerTileLayerEditor`. Simplemente se inicializan los distintos elementos que se van a necesitar y diferencia si el elemento que se está modificando es una capa o un grupo de capas. Se obtiene el objeto `GeoServerSeederLayerInfo` asociado a esta capa/grupo de capas a partir del modelo de datos que se recibe como parámetro

```
public GeoServerSeederLayerEditor(final String id, final IModel<? extends
CatalogInfo> layerModel, final IModel<GeoServerSeederLayerInfo>
seederLayerModel) {
		super(id);
		checkArgument(seederLayerModel instanceof
				GeoServerSeederLayerInfoModel);
		this.layerModel = layerModel;
		setModel(seederLayerModel);

		final IModel<String> createSeederLayerLabelModel;

		final CatalogInfo info = layerModel.getObject();
		final GeoServerSeederLayerInfo seederLayerInfo =
				seederLayerModel.getObject();

		final GWC gwc = GWC.get();

		if (info instanceof LayerInfo) {
			createSeederLayerLabelModel = new
				ResourceModel("enableSeederForLayer");
			ResourceInfo resource = ((LayerInfo) info).getResource();
			resource = ModificationProxy.unwrap(resource);
			originalLayerName = resource.getPrefixedName();
		} else if (info instanceof LayerGroupInfo) {
			createSeederLayerLabelModel = new
				ResourceModel("createSeederLayerForLayerGroup");
			LayerGroupInfo lgi =
				ModificationProxy.unwrap((LayerGroupInfo) info);
			originalLayerName = tileLayerName(lgi);
		} else {
			throw new IllegalArgumentException("Provided model does
				not target a LayerInfo nor a LayerGroupInfo: " +
				info);
		}
```

Código 27. Extracto `GeoServerSeederLayerEditor` III

Solo mostramos el formulario si se ha guardado previamente una configuración *Tile Caching* para esta misma capa. Es decir, si el usuario no ha guardado o ha borrado previamente la configuración *Tile Caching* para esta capa, no se le mostrará el formulario, sino un aviso indicándole porque no puede acceder a esta funcionalidad. Si el usuario guarda una configuración *Tile Caching* para la capa se genera una configuración *Seeder* con los valores por defecto para esta capa, aunque no se permitió acceder al formulario.

Se implementa esta disyuntiva como una casilla de verificación (*checkbox*) aunque nunca se mostrará en pantalla y por lo tanto el usuario no podrá seleccionarlo/deseleccionarlo. Su valor depende de si existe o no configuración *Tile Caching* guardada.

```
boolean doCreateSeederConfig;

if (gwc.hasTileLayer(info)) {
       doCreateSeederConfig = true;
} else {
       doCreateSeederConfig = false;
}
add(existTileCaching = new CheckBox("enableSeeder", new
              Model<Boolean>(doCreateSeederConfig)));

existTileCaching.add(new AttributeModifier("title", true,
              new ResourceModel("enableSeeder.title")));
```

Código 28. Extracto `GeoServerSeederLayerEditor` IV

- Se define el **contenedor del editor/formulario** (Código 29) para después ir añadiéndole los distintos elementos del formulario. Después de definir cualquier elemento para añadirlo a la interfaz *web* es necesaria la sentencia `config.add(nombre_elemento)`, donde `nombre_elemento` será el nombre del objeto con el que hemos definido el elemento de la interfaz *web*.

```
container = new WebMarkupContainer("container");
container.setOutputMarkupId(true);
add(container);

configs = new WebMarkupContainer("configs");
configs.setOutputMarkupId(true);
container.add(configs);
```

Código 29. Extracto `GeoServerSeederLayerEditor` V

Añadimos los distintos elementos que conforman el formulario. La estructura básica es inicializar el objeto creado en la cabecera del editor como un elemento del tipo correspondiente (*checkbox*, selectores, paneles externos), añadirle los atributos correspondientes, y añadirlo a la interfaz (`configs.add()`).

- Habilitar la comprobación de parámetros introducidos:

```
add(enabled = new CheckBox("enabled", new
       PropertyModel<Boolean>(getModel(), "enabled")));
enabled.add(new AttributeModifier("title", true, new
       ResourceModel("enabled.title")));
configs.add(enabled);
```

Código 30. Extracto `GeoServerSeederLayerEditor` VI

- **Tabla de *Features Sources***. Esta tabla permite al usuario seleccionar el *Feature Source* al que pertenece la capa. Como se verá más adelante, para construir este panel necesitamos pasarle como parámetros una lista de elementos a mostrar en el panel. Para ello definimos una lista de elementos `AbstractFSEntry` (Sección 4.3.1.5) que representan los objetos que mostramos en este panel y la forma en que se almacena la información de *Features Source* en el catálogo *Seeder*. Intentamos obtener del objeto `GeoServerSeederLayerInfo` una lista de *Feature Sources* ya inicializada mediante el método `get()` correspondiente. En caso de que exista, inicializamos la lista recientemente creada añadiéndole las entradas ya guardadas. En caso contrario

se deja vacía. Por cuestiones que se verán después, el panel también necesita el identificador de la capa que se está modificando, el cual se obtiene mediante las sentencias 9-12. Con todos los datos creamos el nuevo panel y los añadimos.

```
1  List<AbstractFSEntry> lista = new ArrayList<AbstractFSEntry>();
2
3  if (seederLayerInfo.getFeatureSource() != null) {
4         for (int i = 0; i < seederLayerInfo.getFeatureSource().size(); i++) {
5               lista.add(seederLayerInfo.getFeatureSource().get(i));
6         }
7  }
8
9  String layerId = null;
10 if (info.getId() != null) {
11        layerId = info.getId();
12 }
13
14 add(featureSourcePanel = new FeatureSourcePanel("featureSourcePanel", lista,
15 layerId));
16 configs. add(featureSourcePanel);
```

Código 31. Extracto `GeoServerSeederLayerEditor` VII

- **Panel de configuración de los trabajos** (Código 32). De manera análoga procedemos a añadir un nuevo panel externo. En este caso este necesita recibir como parámetros la lista de *GridSet* definimos en la configuración *Tile Caching* para esta capa, la lista de trabajos ya guardados (es decir, entradas de esta tabla ya guardadas) y el identificador de la capa que se está modificando.

Haremos uso de los elementos que nos proporciona *GWC* para obtener la lista de *GridSet* asociados a esta capa, para ello deberemos obtener el objeto `LayerInfo` o `LayerGroupInfo`, en función de qué estemos modificando, a partir de él obtener el elemento *Tile Layer* (configuración *Tile Caching*) asociado y por último mediante el método `getGridSubsets()` obtener el conjunto de los *GridSet* configurados. La única diferencia con lo que necesitamos es que nos devuelve un conjunto de datos de tipo *set* (elementos sin ordenar) y queremos una lista (elementos ordenados), simplemente la línea 22 nos permite definir un *array* a partir de un conjunto tipo *set*. La particularidad de esta lista es que nunca estará vacía porque para poder llegar a este panel se necesita tener guardada una configuración *Tile Caching* y para que esta se pueda guardar se requiere seleccionar al menos un *GridSet*. La lista de trabajos ya guardados se obtiene mediante el método `get()` correspondiente a partir del objeto `GeoServerSeederLayerInfo` obtenido a partir del modelo de datos del editor.

```
1  Set<String> gridsetSet = null;
2  List<String> gridsetList = null;
3  LayerInfo layerInfo;
4  LayerGroupInfo layerGroupInfo;
5
6  String layerName = null;
7
8  if (info.getId() != null && gwc.hasTileLayer(info)) {
9         layerInfo = gwc.getLayerInfoById(info.getId());
10        layerGroupInfo = gwc.getLayerGroupById(info.getId());
11        GeoServerTileLayer tileLayer;
12        if (layerInfo != null) {
13              tileLayer = gwc.getTileLayer(layerInfo);
14        } else {
15              tileLayer = gwc.getTileLayer(layerGroupInfo);
16        }
17
18        layerName = tileLayer.getName();
```

```
19          gridsetSet = tileLayer.getGridSubsets();
20
21          gridsetList = new ArrayList(gridsetSet);
22
23      }
24
25      List<PrefetchJobConfigurationEntry> jobList =
26                              seederLayerInfo.getJobConfiguration();
27      add(prefetchJobConfigurationPanel = new
28                      PrefetchJobConfigurationPanel("prefetchJobConfigurationPanel",
29      gridsetList, jobList, layerId));
30      configs.add(prefetchJobConfigurationPanel);
```

Código 32. Extracto `GeoServerSeederLayerEditor` VIII

- **Panel de selección de base de datos** (Código 33). Este panel permite al usuario seleccionar la base de datos en la que quiere que se almacenen las estadísticas de acceso a esta capa concreta. Distintas capas pueden tener distinto valor en este apartado, a elección del usuario. Este panel requiere recibir como parámetro el identificador de la base de datos (denominado *store*) seleccionado y el identificador de la capa que se está modificando.

Las líneas 2-4 tienen por objetivo borrar la asociación entre esta capa y la base de datos correspondiente para que no se sigan guardando estadísticas en ella. Una vez que el usuario ha guardado una configuración (requisito para poder hacerlo entre otras cosas será indicar un *store*) se crea una asociación entre la capa y la base de datos y se empiezan a recoger estadísticas. Estas líneas están destinadas a borrar esa asociación si por cualquier motivo del fichero de configuración (catálogo *Seeder*) se ha borrado, por ejemplo manualmente, el identificador de la base de datos, para evitar almacenar estadísticas en el lugar erróneo.

```
1   String storeId = seederLayerInfo.getStore();
2   if (storeId == null && layerId != null) {
3           Seeder.get().removeItemIndexMap(layerId);
4   }
5
6   add(storePanel = new StoreStatsPanel("storePanel", storeId, layerId));
7   configs.add(storePanel);
```

Código 33. Extracto `GeoServerSeederLayerEditor` IX

- **Panel de visualización de tareas lanzadas** (Código 34). Este panel permite al usuario visualizar las tareas de *prefetch* lanzadas relativas a esta capa. Necesita recibir como parámetro el nombre la capa, para de todas las múltiples tareas que se hayan lanzado mostrar solo las asociadas a esta capa concreta.

```
add(taskPanel = new TaskPanel("taskPanel", layerName));
configs.add(taskPanel)
```

Código 34. Extracto `GeoServerSeederLayerEditor` X

Las últimas líneas del constructor permiten ocultar o mostrar el formulario en función del valor del booleano `existTileCaching`.

```
1   configs.setVisible(existTileCaching.getModelObject());
2
3   existTileCaching.add(new OnChangeAjaxBehavior() {
4           private static final long serialVersionUID = 1L;
5
6           @Override
7           protected void onUpdate(AjaxRequestTarget target) {
8                   final boolean existTileLayer =
```

```
                        existTileCaching.getModelObject().booleanValue();

                        if (existTileLayer) {
                                updateConfigsVisibility(target);
                        }

        }
        });


        private void updateConfigsVisibility(AjaxRequestTarget target) {
        final boolean createTileLayer =
                existTileCaching.getModelObject().booleanValue();
                configs.setVisible(createTileLayer);
                target.addComponent(container);
        }
```

Código 35. Extracto GeoServerSeederLayerEditor XI

- Definición de un método `convertInput()` que se encarga de procesar cada uno de los valores introducidos en el formulario y crear un objeto (en este caso `GeoServerSeederLayerInfo`) con los valores concretos del formulario antes de que se guarde en el catálogo (y por tanto se cree el fichero en el directorio de datos). Entre otras cosas este método nos permite validar los datos introducidos. Este método se ejecuta al cambiar de pestaña o al darle a la opción de guardar el formulario.

```
protected void convertInput() {

        Seeder seeder = Seeder.get();
        final boolean enableSeederConfig =
                seeder.getConfig().isCacheLayersByDefault();

        GWC gwc = GWC.get();
        final CatalogInfo layer = layerModel.getObject();
        final boolean tileLayerExists = gwc.hasTileLayer(layer);

        GeoServerSeederLayerInfo seederLayerInfo = getModelObject();

        if (enableSeederConfig && tileLayerExists) {
                enabled.processInput();

                if (enabled.getModelObject().booleanValue()) {
                        for (int i = 0; i < prefetchJobConfigurationPanel.
                        getEntries().size(); i++) {
                                if (prefetchJobConfigurationPanel.getEntries()
                                .get(i).getStartDelay() < 0) {
                                        error("Start Delay must be positive");
                                                return;
                                }
                                if (prefetchJobConfigurationPanel.getEntries()
                                .get(i).getRepeatCount() < 0) {
                                        error("Repeat count must be positive: "
                                                + "0 = Repeat Indefinetily , "
                                                + "1 = No repetition , "
                                                + "> 1 = Especific Repetition "
                                                + "Count");

                                        return;
                                }
                                if (prefetchJobConfigurationPanel.getEntries()
                                .get(i).getRepeatCount() > 1 &&
                                prefetchJobConfigurationPanel.getEntries().get(i).
                                getRepeatInterval() == 0) {
```

```
38                                      error("With repetition, the interval "
39                                              + "cannot be null");
40                                              return;
41                                      }
42                                      if (prefetchJobConfigurationPanel.getEntries()
43                                      .get(i).getRepeatInterval() < 0) {
44                                              error("Repeat interval cannot "
45                                              + "be negative");
46                                              return;
47                                      }
48                                      if (prefetchJobConfigurationPanel.getEntries()
49                                      .get(i).getBuffer() < 0) {
50                                              error("Buffer must be positive");
51                                              return;
52                                      }
53                                      if (prefetchJobConfigurationPanel.getEntries()
54                                      .get(i).getJobName() == null) {
55                                              error("All job configurations "
56                                              + "must have a Job Name");
57
58                                      return;
59                                      }
60                                      if (prefetchJobConfigurationPanel.getEntries()
61                                      .get(i).getFormat() == null) {
62                                              error("Select a image format "
63                                              + "for job configuration");
64                                              return;
65                                      }
66                              }
67
68                              if (featureSourcePanel.getEntries().isEmpty()) {
69                                      error("Feature Source list cannot be empty");
70                                      return;
71                              }
72
73                              if (storePanel.getEntries().isEmpty()) {
74                                      error("Select a Data Store to store stats");
75                                      return;
76                              }
77
78                              String storeId =
79                              storePanel.getEntries().get(0).getStoreInfoId();
80                              boolean correct = seeder.validateStore(storeId);
81                              if (!correct) {
82                                      error("DataStore is not JDBC");
83                              }
84                      }
85              seederLayerInfo.setFeatureSource(featureSourcePanel.getEntries());
86
87              seederLayerInfo.setJobConfiguration(prefetchJobConfigurationPanel
88                      .getEntries());
89                      List<StoreEntry> list = storePanel.getEntries();
90                      if (list != null && !list.isEmpty()) {
91                              // en esta lista solo habrá un elemento.
92                              seederLayerInfo.setStore(list.get(0).getStoreInfoId());
93                      }
94                      seederLayerInfo.setId(layerModel.getObject().getId());
95                      setConvertedInput(seederLayerInfo);
96              } else {
97                      seederLayerInfo.setId(null);
98                      setConvertedInput(seederLayerInfo);
99              }
100             setModelObject(seederLayerInfo);
101     }
```

Código 36. Extracto GeoServerSeederLayerEditor XII

En las líneas 3-11 se preparan los datos que se tienen que validar. Solo se procesará la información de este formulario en caso de que esté activo, es decir, que si el usuario accede a la pestaña de configuración *Seeder* y no puede rellenar el formulario porque no ha guardado previamente una configuración *Tile Caching*, al cambiar de pestaña o darle a guardar al formulario (al darle a guardar desde cualquier pestaña se procesan todas las pestañas y la información de cada una de ellas se almacena correspondientemente) no deberá validar si los datos introducidos son o no son correctos o si se han rellenado campos obligatorios. Para ello utilizamos el booleano `tileLayerExists`.

Otro requisito para procesar el formulario será que esté activada la funcionalidad de almacenar configuraciones *Seeder* (booleano `enableSeederConfig`). Realmente este requisito se podría omitir porque por defecto este booleano siempre tiene valor `true`, y el usuario no puede modificarlo de ninguna manera (es un valor por defecto establecido en `SeederConfig`).

En caso que se cumplan estos dos requisitos se accede a las sentencias incluidas dentro de la estructura de control de flujo `if` de la línea 13. Mediante la línea 16 obtengo el valor introducido en la casilla de verificación de *"Validación de los valores introducidos"*, será `true` si se ha seleccionado la casilla o `false` en caso contrario. Coherentemente si se ha seleccionado esta opción se validarán los datos introducidos, en caso contrario nos saltaremos esta funcionalidad.

Suponemos que el usuario ha seleccionado esta opción, por lo que debemos validar que se han rellenado los campo obligatorios (prácticamente todos los que aparecen en el formulario) y que los datos introducidos se adaptan a los valores esperados o válidos. En concreto:

- No es obligatorio configurar trabajos de *prefetch*, pero en caso de iniciar una configuración, se deberán rellenar todos los campos del correspondiente panel. Además como algunos de ellos son muy precisos, se requerirá que su contenido se amolde a unos ciertos patrones:
    - El retardo inicial (*Start Delay*) deberá ser una cantidad positiva o nula porque representa un instante temporal. Líneas 19-23.
    - El número de repeticiones (*Repeat Count*) deberá ser un número positivo mayor que la unidad para un número concreto de repeticiones, 0 para repetir indefinidamente y 1 para no repetir. Líneas 24-32.
    - En caso de que en el campo anterior se indique un número concreto de repeticiones (número de repeticiones mayor que la unidad), se deberá especificar el intervalo de tiempo que tiene transcurrir entre repeticiones. Además, como en el caso del retardo inicial, este parámetro deberá ser un número positivo o nulo, en la práctica es ineficiente que sea cero. Ya se abordarán más adelante conceptos sobre cómo tratar cada uno de estos parámetros y que significan. Líneas 34-48
    - El *buffer* debe ser una cantidad positiva. Líneas 48-52.
    - Se debe especificar un nombre para el trabajo, para poder diferenciarlo de otros. Líneas 53-58.
    - Se debe especificar un formato de imagen de entre los ofrecidos.

- Es obligatorio que la lista de *Features Source* a los que pertenece la capa tenga al menos un elemento. Se verá que se ha impuesto otra condición que se comprueba más adelante.
- De igual manera se deberá haber seleccionado una, y solo una base de datos, es decir que el panel de selección de bases de datos deberá tener una entrada. Se verá que no es problema el hecho de que pueda tener más de una entrada, porque si el usuario selecciona una base de datos y después selecciona otra, se borra la anterior y se añade la nueva. Así que solo queda por comprobar que al menos se ha seleccionado una.

Requisito sobre esta base de datos, para poder manejar la infraestructura de los módulos *Seeder-Stats* y *Seeder-Prefetch*, es que sean se tipo *JDBC*. El siguiente paso será comprobar este hecho para la base de datos que haya seleccionado el usuario. Para ello se hace uso del método `validateStore()` del objeto central `Seeder` al que se le debe pasar como parámetro el identificador de la base de datos seleccionada. Este método tiene la siguiente estructura:

```
public boolean validateStore(String storeId) {
		boolean correct = false;
		StoreInfo storeInfo = GWC.get().getCatalog()
			.getStore(storeId, StoreInfo.class);
		if (storeInfo != null) {
			DataStoreInfo dataInfo = (DataStoreInfo) storeInfo;
			DataAccess<? extends
				FeatureType, ? extends Feature> store;
			try {
				store = dataInfo.getDataStore(null);
				DataStore dataStore = (DataStore) store;
				if (dataStore instanceof JDBCDataStore) {
					correct = true;
				} else
					correct = false;
			} catch (IOException e) {
			logger.log(Level.WARNING, "Unable to find "
				+ "Data Store for the selected "
				+ "store: " + storeInfo.getName(), e);
			}

			return correct;
		} else {
			logger.log(Level.WARNING, "Unable to find "
				+ "Data Store for store: " + storeId);

			return false;
		}
}
```

Código 37. Método `validateStore()` del objeto `Seeder`

A partir del identificador de base de datos que recibe como parámetro busca la información de la base de datos (lo que se denomina `StoreInfo`). Primero comprobamos que realmente existe una base de datos con ese identificador. Si no existe el objeto `StoreInfo` tendrá un valor `null`. Si no tiene valor `null`, se transforma el `StoreInfo` en un tipo más concreto que es `DataStoreInfo` (hay distintos tipo de `StoreInfo`, solo el subtipo `DataStoreInfo` puede ser de tipo *JDBC* por lo que nos centramos en él). Intentamos manipular esta transformación y comprobar si es o no de tipo `JDBCDataStore`. En caso de que haya algún problema es que no

es del tipo que queremos y retornamos un `false`, es decir, la base de datos seleccionada no se ajusta a lo requerido, en caso de que no haya habido ningún problema retornamos `true`, es decir, la base de datos se ajusta a lo requerido. Este booleano es el que manejamos, de vuelta al editor, en el método `convertInput()` para validar el identificador de base de datos seleccionada.

Si en alguno de las validaciones anteriores ha habido algún problema se abandonará el procesado de los datos (sentencia `return`) y se mostrará al usuario un mensaje de error en la parte superior del formulario en color rojo indicándole que es lo que ha fallado. No se le permitirá cambiar de pestaña o guardar el formulario hasta que haya solucionado estos problemas

El siguiente paso es preparar los datos que se han introducido para poder añadirlos a la base de datos (líneas 85-93).

Las líneas 94-95 y 97-98 realizan el mismo proceso, pero diferenciando el modelo de datos tratado, las primeras están dentro de la estructura `if` que se encarga de validar los datos introducidos y una vez procesados los añade al modelo de datos junto con la línea 100, mientras que las líneas 97-98 realizan la misma tarea pero cuando no se ha procesado el formulario y por ello trabajarán con un conjunto de datos distinto.

- Definición de un método `save()` en el que se indica que operaciones hay que llevar a cabo cuando en la página se pulse el botón correspondiente de enviar formulario. En nuestro caso las operaciones a realizar serán crear un nuevo elemento a almacenar en el catálogo *Seeder* a partir de la información introducida, comprobando si en función de los valores hay que "trucar" algún campo, comprobar que se debe guardar este objeto (en nuestro caso solo deberá guardarse si ya existe un objeto *Tile Caching* de *GeoServer* para la misma capa/grupo de capas) y crear un nuevo objeto o modificarlo en caso de que ya exista.

```
1  public void save() {
2
3          final GWC gwc = GWC.get();
4          final Seeder seeder = Seeder.get();
5          final CatalogInfo layer = layerModel.getObject();
6
7          final GeoServerSeederLayerInfo seederLayerInfo =
8                           getModelObject();
9          final boolean tileLayerExists = gwc.hasTileLayer(layer);
10         String tileLayerName = seederLayerInfo.getName();
11         Preconditions.checkState(layer.getId() != null);
12         seederLayerInfo.setId(layer.getId());
13
14         final String name;
15         final GridSetBroker gridsets = gwc.getGridSetBroker();
16
17         GeoServerSeederLayer tileLayer;
18         if (layer instanceof LayerGroupInfo) {
19                 LayerGroupInfo groupInfo = (LayerGroupInfo) layer;
20                 name = tileLayerName(groupInfo);
21                 tileLayer = new GeoServerSeederLayer(groupInfo,
22                                 gridsets,
23                                 seederLayerInfo);
24         } else {
25                 LayerInfo layerInfo = (LayerInfo) layer;
26                 name = tileLayerName(layerInfo);
```

```
27                  tileLayer = new GeoServerSeederLayer(layerInfo,
28                          gridsets, seederLayerInfo);
29          }
30
31          CatalogInfo info = layerModel.getObject();
32          Set<String> gridsetSet = null;
33          List gridsetList = null;
34          List<PrefetchJobConfigurationEntry> gsList = null;
35
36          LayerInfo layerInfo = gwc.getLayerInfoById(info.getId());
37          LayerGroupInfo layerGroupInfo =
38                          gwc.getLayerGroupById(info.getId());
39
40          seederLayerInfo.setJobConfiguration(
41                          prefetchJobConfigurationPanel.getEntries());
42
43          if (gwc.hasTileLayer(layer)) {
44
45                  if (layerInfo != null) {
46                          gridsetSet =
47                          gwc.getTileLayer(layerInfo).getGridSubsets();
48                  } else {
49                          gridsetSet = gwc.getTileLayer(layerGroupInfo)
50                              .getGridSubsets();
51                  }
52                          gridsetList = new ArrayList(gridsetSet);
53          }
54
55          gsList = new ArrayList<PrefetchJobConfigurationEntry>();
56
57          if (seederLayerInfo.getJobConfiguration() != null) {
58                  for (int i = 0; i < seederLayerInfo.getJobConfiguration()
59                          .size(); i++) {
60                              gsList.add(seederLayerInfo
61                                  .getJobConfiguration().get(i));
62                          }
63          }
64
65          if (seederLayerInfo.getJobConfiguration() != null) {
66                  for (int i = 0; i < seederLayerInfo.getJobConfiguration()
67                      .size(); i++) {
68                          if(!gridsetList.contains(seederLayerInfo
69                              .getJobConfiguration().get(i).getGridsetName())) {
70
71                                  seederLayerInfo.getJobConfiguration().remove(i
72                              );
73                                  }
74                          }
75          }
76
77          seederLayerInfo.setName(name);
78          final boolean existsSeederConfig =
79                      seeder.containsLayer(tileLayer.getId());
80
81          if (tileLayerExists) {
82                  if (existsSeederConfig) {
83                          seeder.save(tileLayer);
84                  } else {
85                          seeder.add(tileLayer);
86                  }
87          } else {
88                  seeder.removeSeederFile(tileLayerName);
89          }
90
91          if (seederLayerInfo.getStore() != null) {
92                  Seeder.get().createDataBase(seederLayerInfo.getStore(),
```

```
93                              tileLayer);
94                      }
95              }
```

Código 38. Extracto `GeoServerSeederLayerEditor` XIII. Método `save()`

La estructura de este método es análoga a la del método `save()` que, por ejemplo, podemos encontrarnos en el editor de la pestaña *Tile Caching*. Se ha añadido una funcionalidad que consiste en comprobar que todos los trabajos de *prefetch* configurados están asociados a un *GridSet* que está seleccionado para esta capa en la configuración *Tile Caching* (líneas 65-75). El supuesto en el que se basa este añadido es la posibilidad de que el usuario haya definido un cierto trabajo de *prefetch* asociado a un *GridSet* concreto seleccionado de entre las posibilidades, que se verá que serán los distintos *GridSet* que el usuario haya seleccionado en la configuración *Tile Caching*, pero posteriormente haya deseleccionado ese *GridSet* dentro de esta misma configuración. En este caso se procede a borrar todos los trabajos que no estén asociados a *GridSet* configurados en el *Tile Caching*. Visto de otra forma, de todos los trabajos que puedan aparecer en el panel de configuración de trabajos de *prefetch*, solo se guarda en el catalogo (fichero) los que se refieran a *GridSet* seleccionados en la pestaña de *Tile Caching* en el momento de pulsar el botón de guardar/enviar formulario.

Las líneas 81-85 tienen por objetivo modificar o añadir la configuración *Seeder* en función de que ya existiera o no previamente. Accede a esta disyuntiva siempre que exista una configuración *Tile Caching*, en caso contrario se procede a eliminar el fichero de configuración *Seeder* asociado en caso de que exista (supuesto que una vez creada configuración *Tile Caching* y *Seeder*, el usuario elimina la configuración *Tile Caching*, automáticamente se elimina también la configuración *Seeder* de su catálogo). Los métodos `save()`, `add()` y `removeSeederFile()` presentan una implementación análoga a los métodos que utiliza *GWC* para realizar las mismas tareas, por lo que no se entra en más detalles en cuanto a su implementación.

Por último, líneas 91-94, se inicializa/reinicializa el módulo de recogida de estadísticas (*Seeder-Stats*) para esta capa en función del identificador de base de datos seleccionado por el usuario. Este método recibe como parámetros el identificador de la base de datos seleccionado y la capa/grupo de capas (*Tile Layer*) sobre la que se quiere lanzar el módulo de recogida de estadísticas:

```
1   public void createDataBase(String storeId, TileLayer layer) {
2
3           Logger.log(Level.INFO, "Accessing Data Base for layer: " +
4           layer.getName());
5
6           StoreInfo storeInfo = GWC.get().getCatalog().getStore(storeId,
7                 StoreInfo.class);
8           DataStoreInfo dataInfo = (DataStoreInfo) storeInfo;
9           DataAccess<? extends FeatureType, ? extends Feature> store = null;
10          try {
11                  store = dataInfo.getDataStore(null);
12          } catch (IOException e) {
13                  logger.log(Level.WARNING, "Unable to find Data Store for "
14                          + "store: " + storeInfo.getName(), e);
15          }
16          DataStore dataStore = (DataStore) store;
17          if (dataStore instanceof JDBCDataStore) {
18          JDBCDataStore jdbc = (JDBCDataStore) dataStore;
```

```
19          DataSource dataSource = jdbc.getDataSource();
20          String table = AbstractIndexDAO.TABLE_NAME;
21          IndexDAO index;
22
23          if (dataInfo.getType().equals("PostGIS") ||
24                 dataInfo.getType().equals("H2")) {
25                 if (dataInfo.getType().equals("PostGIS")) {
26                        PostgresIndexDAO pgindex = new PostgresIndexDAO();
27                        pgindex.setDataSource(dataSource);
28                        pgindex.setTable(table);
29                        try {
30                               pgindex.init();
31                        } catch (SQLException e1) {
32                        logger.log(Level.WARNING, "Error accesing Data Base: " +
33                               dataInfo.getName() + " when creating " + table
34                               + " table", e1);
35                        }
36
37                        index = pgindex;
38                 } else {
39                        H2IndexDAO h2index = new H2IndexDAO();
40                        h2index.setDataSource(dataSource);
41                        h2index.setTable(table);
42
43                       try {
44                               h2index.init();
45                        } catch (SQLException e2) {
46                        logger.log(Level.WARNING, "Error accesing Data Base: "
47                               + dataInfo.getName() + " when creating " + table
48                               + " table", e2);
49
50                        }
51
52                        index = h2index;
53
54                 }
55                 this.indexMap.put(layer.getId(), index);
56
57                 logger.log(Level.INFO, "Initiating Stats Module for layer: "
58                        + layer.getName());
59          } else
60                 logger.log(Level.WARNING, "Not supported JDBC Data Store");
61          } else
62                 logger.log(Level.WARNING, "Selected store is not JDBC");
63          }
```

Código 39. Método `createDataBase()` de objeto `Seeder`

A partir del identificador de la base de datos (*store*) seleccionado para la capa se obtiene su `dataStore`. En principio este debería ser siempre de tipo *JDBC* porque la interfaz de usuario lo comprueba antes de llegar a este punto, tal y como se ha explicado anteriormente. Para evitar que el usuario cambie manualmente los ficheros de configuración e introduzca un valor para el identificador de la base de datos que no cumpla la condición, se vuelve a comprobar que la base de datos indicada es de tipo *JDBC*.

En el caso que la base de datos sea de tipo *JDBC* se debe hacer otra particularización, es decir, dentro de las bases de datos de tipo *JDBC* hay a su vez otra serie de subtipos. El módulo *Seeder-Stats* solo puede trabajar con bases de datos *H2* o *PostGIS*. Para poder lanzar este módulo se necesita crear lo que se denominará *Index* o Índice (Sección 4.4.2.3), e inicializarlo correctamente en función del tipo de base de datos, por lo que a mayores se realiza una nueva comprobación sobre la base de datos y en función del resultado inicializar el índice como de

tipo *H2* o *PostGIS* (líneas 24-60). Si no es de ninguno de estos dos tipos se lanza un error indicando que la base de datos seleccionada no se acomoda a ningún tipo de los soportados por la aplicación.

Una vez creado el índice *(index*), se registra. El registro del *index* se realiza añadiéndole a un mapa llamado `indexMap` del objeto `Seeder`. Este mapa se encarga de almacenar las correspondencias entre capas y los índices a las bases de datos en que se recogen sus estadísticas de acceso, para poder saber en cualquier momento para qué capas se tiene configurada la base de datos y el acceso a ella. En caso de haber modificado la configuración de la base de datos de una capa, no pasa nada, porque al intentar añadir al mapa `indexMap` una pareja clave-valor con el mismo valor de clave, sustituye la anterior, por lo que se elimina la anterior conexión a la base de datos y se crea una nueva.

En el apartado correspondiente (Sección 4.4.2) se analizará cómo se lleva a cabo la conexión y establecimiento de la base de datos y cómo esta se inicializa.

Para terminar la sección relativa al editor del formulario de configuración *Seeder*, se especifica el fichero *HTML* necesario (`GeoServerSeederLayerEditor.html`) para definir la estructura que presentará cuando se presente en un navegador *web*. Como se utiliza la tecnología de *wickets*, simplemente se declaran los elementos y se indica en cada uno de ellos el identificador que se haya dado al elemento cuando se declaró en el fichero *Java*. Se debe tener en cuenta que no sirve de nada el orden en que se definen los elementos del formulario en el fichero *Java*, el orden los impone la definición de elementos en el fichero *HTML*, no es mala práctica seguir el mismo orden para facilitar la comprensión de código.

```
<html xmlns:wicket="http://wicket.apache.org/">
<head></head>
<body>
<wicket:panel>
<fieldset>
<legend>
<span>
       <wicket:message key="GeoServerSeederLayerEditor.title">Seeder caching
       configuration</wicket:message>
</span>
<a href="#" class="help-link"></a>
</legend>

<wicket:message key="GeoServerSeederLayerEditor.messageAdvice">
Advice</wicket:message>

<br />

<wicket:message key="GeoServerSeederLayerEditor.messageAdviceSave">
Advice</wicket:message>

<ul>
<li wicket:id="container">
       <div wicket:id="configs">
              <ul>
                     <li><input id="tileCacheEnabled" wicket:id="enabled"
                     class="field checkbox" type="checkbox" />
                     <label for="tileCacheEnabled" class="choice">
                     <wicket:message key="enabled">Enable Seeder
                     caching</wicket:message>
                     </label></li>
```

```
                        <div wicket:id="featureSourcePanel"></div>
                        <br />
                        <div wicket:id="prefetchJobConfigurationPanel"></div>
                        <br /><br />
                        <div wicket:id="storePanel"></div>
                        <br />
                        <div wicket:id="taskPanel"></div>
                    </ul>
                </div>
            </li>
            </ul>
            </fieldset>
            </wicket:panel>
            </body>
            </html>
```

Código 40. Fichero `GeoServerSeederLayerEditor.html`

El fichero *HTML* debe tener el mismo nombre que el fichero *Java* en que se defina la estructura del documento *web*.

4.3.3. Paneles externos

En esta sección se procede a explicar la estructura y funcionamiento de los paneles externos de los que se hace uso en el formulario previamente presentado. En él encontraremos cuatro paneles distintos. El primero de ellos permite al usuario definir el conjunto de *Features Source* al que pertenece la capa/grupo de capas que se está editando (Ver Sección 4.3.3.1.), el segundo permite al usuario configurar el/los trabajos de *prefetch* relacionados con la capa (Ver Sección 4.3.3.2), el tercero permite al usuario seleccionar la base de datos en que se va a almacenar las estadísticas de acceso relativas a esta capa/grupo de capas (Ver Sección 4.3.3.3) y el cuarto panel permite al usuario visualizar el estado de las tareas lanzadas en relación a esta capa (Ver Sección 4.3.3.4).

4.3.3.1. Panel de *Features Source*

Permite al usuario seleccionar el conjunto de *Features Source* al que pertenece la capa/grupo de capas en cuestión. Es decir, el conjunto de fenómenos geográficos, como por ejemplo pudiera ser carreteras, castillos, iglesias… Simplemente consiste en una tabla cuyas entradas son las distintas opciones seleccionadas.

En la Sección 2.4 – Conceptos, se explicó brevemente lo que se denominaba *Feature Source* y también los distintos tipos de datos que se manejaban en *GeoServer* (*vector* y *raster*). Esta implementación solo permitirá seleccionar al usuario *Features Source* que contengan datos de tipo *vector*, porque solo se tiene programada la manipulación de este tipo de *Features* en el módulo *Seeder – Prefetch*. Realmente al usuario se le va a permitir seleccionar cualquier tipo de *Feature Source* pero si no es de tipo *vector* se le mostrará un mensaje de advertencia avisándole que no podrá usar esa selección en un posible trabajo de *prefetch*.

A la hora de analizar este (y cualquier otro panel) hay que distinguir varios conceptos: por un lado el propio panel y por otro las entradas del panel, o mejor dicho, el formato de las entradas del panel. Además este primer panel es un poco más complejo que el resto porque

maneja distintos tipos de entrada en función de que el *Feature Source* que seleccione el usuario sea una capa ya publicada o un origen de fenómenos propiamente dicho.

Otro aspecto a tener en cuenta, es que un panel puede depender a su vez de otro panel, la dependencia más típica (y además la implementada) es que las entradas del panel no se completan manualmente si no que el usuario selecciona de entre unas opciones que aparecen en otro panel, normalmente presentado como una ventana emergente.

Analizamos primero las **entradas** a este panel. Ya se vio la estructura concreta de estas en la sección 4.3.1.5 en la que se analizaron brevemente los objetos que formaban parte de la configuración *Seeder*. En concreto se vio que uno de los elementos de esta configuración era una lista de objeto `AbstractFSEntry`. Este objeto se implementa como una clase abstracta, ya que como se ha indicado previamente, las entradas a este panel pueden ser de dos tipos y en función de ello la información para completarlas se obtendrá de distinta manera, por lo que lo más sencillo es definir un objeto para cada tipo de entrada y hacer que hereden de la clase abstracta `AbstractFSEntry`. A medida que avance el estudio del panel, se explicará el funcionamiento de los métodos de cada uno de estos objetos en un contexto más concreto.

La información para cada una de las entradas que debe presentar esta panel en pantalla consiste en el nombre del elemento, el cual además nos permitirá diferenciar si es una capa (*Layer*) o un origen de fenómenos (*Feature Source*), una advertencia en caso de que la selección no se adapte al requisito de emplear datos de tipo *vector* y una casilla en que el usuario pueda indicar, si así lo quiere, un filtro *CQL* para seleccionar dentro del *Feature Source*. A mayores incluiremos una opción de eliminar la entrada de la tabla y una opción para ordenar las entradas, aunque en la práctica es indiferente su orden.

A mayores, este panel contará con dos enlaces a ventanas emergentes que muestran al usuario una lista de donde seleccionar las entradas que quiere añadir a la tabla. Un enlace a una lista de capas ya publicadas y otro enlace a un panel de selección de orígenes de fenómenos previamente configurados.

El código que permite generar el panel se estructura de la siguiente manera (objeto `FeatureSourcePanel`):

```
public class FeatureSourcePanel extends Panel {…}
```

Código 41. Extracto `FeatureSourcePanel` I

En primer lugar se definen los elementos que compondrán el panel:

```
ModalWindow popupWindow;
SeederEntryProvider entryProvider;
GeoServerTablePanel<AbstractFSEntry> layerTable;
List<AbstractFSEntry> items;
```

Código 42. Extracto `FeatureSourcePanel` II

- `popupWindow`: ventana emergente para los paneles de selección.
- `entryProvider`: clase estática definida en el propio objeto que permite especificar el formato que tendrán las entradas. Visto de forma más coloquial, las columnas que conformarán la tabla, el orden en que deben aparecer, su título…

- `layerTable`: la tabla que conforma el panel.
- `items`: lista de entradas de la tabla.

A continuación se define el constructor del objeto:

```
public FeatureSourcePanel(String id,  List<AbstractFSEntry> lista, String layerId) {…}
```

Código 43. Extracto `FeatureSourcePanel` III

En él se definen en primer lugar la lista *items*, es decir, las entradas:

```
super(id);

Seeder.get().addFeatureList(layerId, this);
items = new ArrayList<AbstractFSEntry>();
items = lista;
```

Código 44. Extracto `FeatureSourcePanel` IV

La segunda línea añade este panel al registro de paneles que lleva el objeto *Seeder*, para facilitar el acceso a él desde cualquier otro punto del formulario. Este aspecto es útil, sobre todo, cuando se lanza un trabajo de *prefetch* desde el segundo panel (Ver Sección 4.3.3.2) ya que se accederá a este panel cargado "en memoria" y se obtendrán de él los parámetros que se necesiten para lanzar las tareas correspondientes, no obligando así al usuario a tener que guardar cualquier cambio realizado en la configuración y tener que volver a acceder a esta página para lanzar el trabajo. Este método tiene el siguiente aspecto:

```
public void addFeatureList(String layerId, FeatureSourcePanel entry) {
        this.featureList.put(layerId, entry);
}
```

Código 45. Extracto `FeatureSourcePanel` V

El objeto `Seeder` tendrá definido un atributo llamado `feautureList` que consistirá en un mapa en el que se van guardando correspondencias entre capas/grupos de capas y el panel (objeto de tipo `FeatureSourcePanel`) asociado. Cada vez que se modifique el panel, se volverá a lanzar el constructor, pero no hay ningún problema con "duplicados" porque este registro es un mapa, cuando se vuelva a guardar una nueva pareja con la misma clave, borrará la anterior.

Las líneas 3 y 4 permiten inicializar la lista de *items*, y darle como valor el conjunto de entradas pasado como parámetros. Recordar que cuando se llama a este panel desde el editor del formulario (Ver Sección 4.3.2) se creaba una lista de entradas, en primer lugar vacía, y después, si el usuario ya había guardado una configuración *Seeder*, la inicializaba recuperando los valores guardados en el catálogo.

A continuación se define la propia tabla:

```
entryProvider = new SeederEntryProvider(items);
add(layerTable = new GeoServerTablePanel<AbstractFSEntry>("layers",
entryProvider) {

        @Override
        protected Component getComponentForProperty(String id, IModel
        itemModel, Property<AbstractFSEntry> property) {
                Component result;
                if (property == SeederEntryProvider.LAYER) {
```

```
                        result = layerLink(id, itemModel);
                } else if (property == SeederEntryProvider.FILTER_CQL) {
                        result = filterField(id, itemModel);
                } else if (property == SeederEntryProvider.WARNING) {
                        result = warningLink(id, itemModel);
                } else if (property == SeederEntryProvider.REMOVE) {
                        result = removeLink(id, itemModel);
                } else if (property == SeederEntryProvider.POSITION) {
                        result = positionPanel(id, itemModel);
                } else {
                        result = null;
                }
                return result;
        }
}.setFilterable(false));
layerTable.setItemReuseStrategy(new DefaultItemReuseStrategy());
layerTable.setOutputMarkupId(true);
```

Código 46. Extracto `FeatureSourcePanel` VI

Se instancia un objeto de tipo `SeederEntryProvider` (explicado posteriormente) y a partir de él se crea una tabla. La estructura `if-else` múltiple que se aprecia permite indicar que debe aparecer en la celda de la tabla correspondiente en función de la propiedad a mostrar. Más coloquialmente, que información se debe mostrar para la entrada correspondiente en cada columna.

Para cada una de las entradas de la tabla, se irá sucesivamente recorriendo cada una de las columnas definidas en `SeederEntryProvider`, mediante la estructura `if-else` se lanzará un método u otro (se analizan a continuación) para extraer la información que se debe mostrar en la correspondiente celda.

El siguiente paso que se ha tomado ha sido definir los dos enlaces que dan lugar a las ventanas emergentes en que se seleccionan las entradas a añadir a la tabla. Se añade una ventana emergente:

```
add(popupWindow = new ModalWindow("popup"));
```

Código 47. Extracto `FeatureSourcePanel` VII

Y dos enlaces. Explicamos el primero, porque el segundo es totalmente análogo:

```
add(new AjaxLink("addLayer") {
        @Override
        public void onClick(AjaxRequestTarget target) {
                popupWindow.setInitialHeight(375);
                popupWindow.setInitialWidth(525);
                        popupWindow.setTitle(new
                                ParamResourceModel("chooseLayer", this));
                popupWindow.setContent(new
                                LayerListPanel(popupWindow.getContentId()) {
                        @Override
                        protected void handleLayer(LayerInfo layer,
                         AjaxRequestTarget target) {
                                popupWindow.close(target);

                         boolean alreadyExist = false;
                        for (int i = 0; i < items.size(); i++) {
                                if (items.get(i).getEntryId().
                                                equals(layer.getId())) {
                                                alreadyExist = true;
```

```
                                        break;
                                }
                        }

                        if (!alreadyExist) {
                                entryProvider.getItems().add(new
                                        LayerEntry(layer));
                        }
                target.addComponent(layerTable);
                }
        });

        popupWindow.show(target);
    }
});
```

Código 48. Extracto `FeatureSourcePanel` VIII

Este primer enlace permite al usuario seleccionar una capa ya publicada. El método `onClick()` definido dentro del enlace, permite indicar que se debe realizar cuando el usuario pulse el enlace. En este caso lanzará una ventana emergente (en la cuarta y quinta línea se especifican su tamaño inicial y el titulo).

La octava línea especifica el contenido que tendrá la ventana emergente. En este caso se ha hecho uso de una panel ya implementado por *GeoServer* (`LayerListPanel()`), que muestra una lista de capas con sus opciones más relevantes y permite seleccionar una pinchando sobre su nombre. El método `handleLayer()` se ejecuta una vez que en la lista anterior se seleccionar una opción (sobreescribe un método sin implementación del panel `LayerListPanel`). Este método comprueba si ya se ha añadido esta capa a la lista, en caso negativo la añade. La línea `popupWindow.show(target)` permite "recargar" el panel.

El segundo enlace:

```
add(new AjaxLink("addFromFeatureSource") {
@Override
public void onClick(AjaxRequestTarget target) {
        popupWindow.setInitialHeight(375);
        popupWindow.setInitialWidth(525);
        popupWindow.setTitle(new
                        ParamResourceModel("chooseFeatureSource", this));
        popupWindow.setContent(new
                DataStoreListPanel(popupWindow.getContentId()) {
        protected void handleFeatureSource(String storeId, String
                        fsName, AjaxRequestTarget target) {
                        popupWindow.close(target);
                        boolean alreadyExist = false;
                        for (int i = 0; i < items.size(); i++) {
                                if (items.get(i) instanceof
                                FeatureSourceEntry && items.get(i)
                                .getfsName().equals(fsName)) {
                                        alreadyExist = true;
                                        break;
                                }
                        }

                        if (!alreadyExist) {
                                entryProvider.getItems().add(new
                                FeatureSourceEntry(storeId, fsName));
```

```
                                }

                                        target.addComponent(layerTable);
                        }
                });

                popupWindow.show(target);
        }
});
```

Código 49. Extracto `FeatureSourcePanel` IX

Es idéntico. La única diferencia está en que el contenido de la ventana emergente está definido mediante otro panel, `DataStoreListPanel`, que muestra al usuario un selector de origen de fenómenos (lo que se denomina almacén) y después una lista de *Features Sources* presente en ese almacén escogido, donde el usuario puede seleccionar la opción deseada. Este panel "emergente" forma parte de la solución planteada (`DataStoreListPanel`), pero su estructura es totalmente análoga a la que presenta la página de creación de nueva capa que está implementada en el objeto `NewLayerPage`. Para definir la tabla que aparecerá al seleccionar un almacén de datos se necesita un objeto denominado `DataStoreListPanelProvider` cuya estructura es análoga al objeto `NewLayerPageProvider` adaptándolo a nuestras necesidades. Una diferencia reseñable es la acción a realizar una vez que se pulsa el correspondiente enlace de realizar acción sobre un *Feature Source*, en este caso se querrá obtener la información sobre ese *Feature* para poder añadirlo a la tabla. Como el código para implementar este panel es análogo a código de *GeoServer* no se específica aquí, para poder consultarlo se puede recurrir al código fuente del trabajo. Además este panel lleva asociado un fichero *HTML* (`DataStoreListPanel.html`) en el que se define la estructura de este panel, de manera similar a como se define la estructura de la página de creación de nueva capa (`NewLayerPage.html`):

```
<html>
<body>
<wicket:panel>
        <form id="selector" wicket:id="selector">
                <wicket:message key="addAFeatureFrom">Add a feature from
                </wicket:message>
                        <select wicket:id="storesDropDown"></select>
        </form>

        <div wicket:id="selectFeaturesContainer">
                <div wicket:id="selectFeatures">

                <p>
                <wicket:message key="listOfResourcesContained">Here is a list of
                resources contained in the store</wicket:message>
                '<span wicket:id="storeName"></span>'.
                <wicket:message key="clickOnTheLayerYouWishToConfigure">Click on
                the feature you wish to select.</wicket:message>
                </p>

                <div wicket:id="features"></div>

                </div>
        </div>

</wicket:panel>
</body></html>
```

Código 50. Fichero `DataStoreListPanel.html`

Una vez que ya hemos completado la definición "visual" del panel, procedemos a indicar el proveedor (`SeederEntryProvider`) que define las columnas de la tabla y que ya se ha mencionado previamente:

```
	static class SeederEntryProvider extends
			GeoServerDataProvider<AbstractFSEntry> {

			public static Property<AbstractFSEntry> LAYER = new
				PropertyPlaceholder<AbstractFSEntry>("element");

			public static Property<AbstractFSEntry> FILTER_CQL = new
				PropertyPlaceholder<AbstractFSEntry>("filterCQL");

			public static Property<AbstractFSEntry> WARNING = new
				PropertyPlaceholder<AbstractFSEntry>("");

			public static Property<AbstractFSEntry> REMOVE = new
				PropertyPlaceholder<AbstractFSEntry>("remove");

			public static Property<AbstractFSEntry> POSITION = new
				PropertyPlaceholder<AbstractFSEntry>("position");

			static List PROPERTIES = Arrays.asList(POSITION, LAYER, WARNING,
				FILTER_CQL, REMOVE);

			List<AbstractFSEntry> items;

			public SeederEntryProvider(List<AbstractFSEntry> items) {
				this.items = items;
			}

			@Override
			protected List<AbstractFSEntry> getItems() {
				return items;
			}

			@Override
			protected List<Property<AbstractFSEntry>> getProperties() {
				return PROPERTIES;
			}

	}
```

Código 51. Extracto `FeatureSourcePanel` X

También permite obtener una lista con las propiedades (columnas) de la tabla y las entradas que la forman.

El siguiente método también permite obtener las entradas de la tabla. Por ejemplo, este método se usó para obtener la lista de elementos seleccionados en el método `save()` del editor del formulario.

```
	public List<AbstractFSEntry> getEntries() {
			return items;
	}
```

Código 52. Extracto `FeatureSourcePanel` XI

A continuación se definen los métodos definidos anteriormente para indicar como se debe completar cada celda de la tabla:

- ¿Cómo debe completar el campo de nombre?:

```
Component layerLink(String id, IModel itemModel) {
	AbstractFSEntry entry = (AbstractFSEntry) itemModel.getObject();
	return new Label(id, entry.getUserFriendlyName());
}
```

Código 53. Extracto `FeatureSourcePanel` XII

A partir de la entrada (`itemModel.getObject()`) se lanza el método `getUserFriendlyName()` que permite obtener un nombre amigable para el usuario. En caso de que la entrada sea una capa ya publicada:

```
public String getUserFriendlyName() {
	return "Layer: " + getLayer().prefixedName();
}
```

Código 54. Extracto `FeatureSourcePanel` XIII

A partir del objeto capa (*layer*) de *GeoServer* hago uso de sus utilidades para obtener su nombre completo, es decir, almacén y nombre representativo. Se añade a mayores "Layer: " para que el usuario reconozca de qué tipo de entrada se trata.

Si la entrada es un *Feature Source*:

```
public String getUserFriendlyName() {
	return "Feature Source: " + this.getFeatureSource();
}



public String getFeatureSource() {

	CoverageStoreInfo csInfo = GeoServerApplication.get()
	.getCatalog().getCoverageStore(storeId);
	DataStoreInfo dsInfo = GeoServerApplication.get()
	.getCatalog().getDataStore(storeId);

	String storeName = null;
	if (csInfo != null) {
		storeName = csInfo.getName();
	} else {
		storeName = dsInfo.getName();
	}
	String displayName = storeName + ":" + featureSourceName;
	return displayName;

}
```

Código 55. Extracto `FeatureSourcePanel` XIV

Hacemos uso de las utilidades de *GeoServer* para obtener el nombre el *FeatureSource* seleccionado. Añadimos "Feature Source: " al nombre para que el usuario identifique el tipo de entrada.

- ¿Es esta entrada válida?

```
Component warningLink(String id, IModel itemModel) {
	AbstractFSEntry entry = (AbstractFSEntry) itemModel.getObject();
	final CatalogIconFactory icons = CatalogIconFactory.get();
```

```
            boolean warning = entry.warning();
            Fragment f = new Fragment(id,"warning",FeatureSourcePanel.this);

            String message = "";
            ResourceReference storeIcon;
            if (warning) {
                  message = "Not be used in seed";
                  storeIcon = new ResourceReference(getClass(),
                        "../../../img/icons/silk/error.png");
            } else
                  storeIcon = new ResourceReference(getClass(),
                        "../../../img/icons/silk/accept.png");

            f.add(new Label("warning", message));
            f.add(new Image("storeIcon", storeIcon));

            return f;
      }
```

Código 56. Extracto `FeatureSourcePanel` XV

Comprobamos si la entrada seleccionada se refiere a un origen de fenómenos con tipos de datos *vector*. Para ello accede al método `warning()` del objeto entrada el cual devolverá `true` si todo es correcto o `false` si la entrada no se ajusta a los requisitos. En el primer caso se muestra un icono dando el visto bueno (un *tick* verde), en el segundo se muestra un mensaje de aviso (*"Not be used in seed"*) y un icono de advertencia.

El método `warning()` cuando la entrada es una capa publicada:

```
      public boolean warning() {

            LayerInfo layerInfo = GWC.get().getCatalog().getLayer(layerId);
            boolean warning = false;
            if (layerInfo.getResource() instanceof FeatureTypeInfo) {
                  warning = false;
            } else {
                  warning = true;
            }
            return warning;
      }
```

Código 57. Extracto `FeatureSourcePanel` XVI

Hace uso de las utilidades de *GeoServer* para obtener el `LayerInfo` asociado a esta capa y a partir de él comprobar si es de tipo `FeatureTypeInfo` (tipo de origen de datos de tipo *vector*).

De manera análoga, si la entrada es de tipo *Feature Source*:

```
      public boolean warning() {

            boolean warning = false;
            if (GWC.get().getCatalog().getDataStore(storeId) != null) {
                  warning = false;
            } else
                  warning = true;
            return warning;

      }
```

Código 58. Extracto `FeatureSourcePanel` XVII

Hace uso de las utilidades de *GeoServer* para acceder la información del origen de datos correspondientes. En concreto comprueba si existe un almacén de tipo *Data Store* (tipo de almacenes que contienen orígenes de fenómenos de tipo *vector*) con el identificador de almacén asociado al *Feature Source* seleccionado.

- Entrada de texto para el filtro:

```
Component filterField(String id, IModel itemModel) {
	Fragment f = new Fragment(id, "text", FeatureSourcePanel.this);
	IModel<String> model = new PropertyModel<String>(itemModel,
	"filterCQL");
	Component text = new TextField<String>("text", model);

	f.add(text);
	return f;
}
```

Código 59. Extracto `FeatureSourcePanel` XVIII

Simplemente se muestra una entrada de texto para que el usuario introduzca el filtro que desea aplicar.

- ¿Cómo borrar una entrada de la tabla?

```
1  Component removeLink(String id, IModel itemModel) {
2  	final AbstractFSEntry entry = (AbstractFSEntry) itemModel.getObject();
3
4  		ImageAjaxLink link = new ImageAjaxLink(id, new
5  		ResourceReference(getClass(),
6  		"../../../img/icons/silk/delete.png")) {
7  			@Override
8  			protected void onClick(AjaxRequestTarget target) {
9
10 				items.remove(entry);
11 				target.addComponent(layerTable);
12 			}
13 		};
14 		link.getImage().add(new AttributeModifier("alt", true, new
15 	ParamResourceModel("AbstractLayerGroupPage.th.remove", link)));
16 		return link;
17
18 	}
19
```

Código 60. Extracto `FeatureSourcePanel` XIX

El usuario dispondrá de un enlace (implementado mediante un icono) que al ser pulsado (método `onClick()` línea 9) elimina de la lista de entradas (`items`) la seleccionada.

- ¿Cómo cambiar la posición de las entradas?

```
Component positionPanel(String id, IModel itemModel) {
	return new PositionPanel(id, (AbstractFSEntry) itemModel.getObject());
}

class PositionPanel extends Panel {

	AbstractFSEntry entry;
	private ImageAjaxLink upLink;
```

```
	private ImageAjaxLink downLink;

	public PositionPanel(String id, final AbstractFSEntry entry) {
		super(id);
		this.entry = entry;
		this.setOutputMarkupId(true);

		upLink = new ImageAjaxLink("up", new
			ResourceReference(getClass(),
			"../../../img/icons/silk/arrow_up.png")) {

			@Override
			protected void onClick(AjaxRequestTarget target) {
				int index =
					items.indexOf(PositionPanel.this.entry);
				items.remove(index);
				items.add(Math.max(0, index - 1),
					PositionPanel.this.entry);
				target.addComponent(layerTable);
				target.addComponent(this);
				target.addComponent(downLink);
				target.addComponent(upLink);
			}

			@Override
			protected void onComponentTag(ComponentTag tag) {
				if (items.indexOf(entry) == 0) {
					tag.put("style", "visibility:hidden");
				} else {
					tag.put("style", "visibility:visible");
				}
			}
		};
		upLink.getImage().add(new AttributeModifier("alt", true, new
			ParamResourceModel("up", upLink)));
		upLink.setOutputMarkupId(true);
		add(upLink);

		downLink = new ImageAjaxLink("down", new
			ResourceReference(getClass(),
			"../../../img/icons/silk/arrow_down.png")) {

			@Override
			protected void onClick(AjaxRequestTarget target) {
				int index =
					items.indexOf(PositionPanel.this.entry);
				items.remove(index);
				items.add(Math.min(items.size(), index + 1),
					PositionPanel.this.entry);
				target.addComponent(layerTable);
				target.addComponent(this);
				target.addComponent(downLink);
				target.addComponent(upLink);
			}

			@Override
			protected void onComponentTag(ComponentTag tag) {
				if (items.indexOf(entry) == items.size() - 1) {
					tag.put("style", "visibility:hidden");
				} else {
					tag. put("style", "visibility:visible");
				}
			}
		};
		downLink.getImage().add(new AttributeModifier("alt", true, new
			ParamResourceModel("down", downLink)));
```

```
                downLink.setOutputMarkupId(true);
                add(downLink);
        }
}
```

Código 61. Extracto `FeatureSourcePanel` XX

Idéntico a las implementaciones que *GeoServer* hace de este aspecto. Al lado de cada entrada aparecerán unas flechas (hacia arriba y hacia abajo) al pulsar en ellas, respectivamente, se desplazará la correspondiente entrada hacia arriba o hacia abajo. Además se asegura que para la primera entrada solo aparece la opción de desplazar hacia abajo y para la última solo la opción de desplazar hacia arriba.

Para finalizar con este panel, se presenta el fichero *HTML* que permite definir la estructura y organización del panel (`FeatureSourcePanel.html`):

```
<html>
<body>
<wicket:panel>
        <ul>

                <li><label><wicket:message key="featureSource">Feature
                        Source</wicket:message></label></li>

                <li><a class="add-link" wicket:id="addLayer"><wicket:message
                        key="addLayer">Add Layer...</wicket:message></a></li>

                <li><a class="add-link" wicket:id="addFromFeatureSource">
                <wicket:message key="addFromFeatureSource">Add From Feature
                        Source...</wicket:message></a></li>

                <li>
                        <div wicket:id="layers"></div>
                </li>

        </ul>

        <div wicket:id="popup"></div>

</wicket:panel>

<wicket:fragment wicket:id="text">
        <input type="text" wicket:id="text" />
</wicket:fragment>

<wicket:fragment wicket:id="warning">
         <img wicket:id="storeIcon" />
         <text wicket:id="warning"/>
</wicket:fragment>

</body>
</html>
```

Código 62. Fichero `FeatureSourcePanel.html`

Es importante el nombre que se le da al *wicket* ya que este debe corresponder con el nombre que se ha dado al elemento *Java* en que se define el contenido.

El siguiente fichero *HTML* permite definir los elementos de posición del panel (`FeatureSourcePanel$PositionPanel.html`):

```
<html>
<body>
	<wicket:panel>
		<span wicket:id="up"></span>
		<span wicket:id="down"></span>
	</wicket:panel>
</body>
</html>
```

Código 63. Fichero `FeatureSourcePanel$PositionPanel.html`

4.3.3.2. Panel de configuración de trabajos de *prefetch*

Este panel permite al usuario configurar los trabajos de *prefetch*. En la sección 4.3.1.5 ya se indicaron los parámetros que formaban esta configuración y por lo tanto van a formar parte de este panel. Consistirá en una tabla donde cada una de sus entradas representa un trabajo distinto.

La forma de manejar este panel es la siguiente: en primer lugar el usuario añadirá un nuevo trabajo seleccionando un *GridSet* entre los que están configurados para esta capa (configuración de la pestaña *Tile Caching*, deben guardarse los cambios realizados en esta configuración antes de añadir un trabajo). Una vez seleccionado el *GridSet*, el usuario dispondrá de un formulario donde rellenar el resto de parámetros. Una vez completado puede borrar este trabajo, o lanzarlo como *Seed* o *ReSeed*. La diferencia entre estas dos formas de lanzar el trabajo consiste en que lanzando el trabajo como *Seed* solo se traen a caché las teselas que falten, por lo tanto es una operación que potencialmente tardará menos que la segunda, que consiste en obviar cualquier tesela que ya se haya podido almacenar en caché y traer todas de nuevo.

Este panel estará formado por una o más entradas de tipo `PrefecthJobConfigurationEntry`, explicadas ya en la Sección 4.3.1.5.

En cuanto a la estructura de este panel,

```
public class PrefetchJobConfigurationPanel extends Panel {}
```

Código 64. Extracto `PrefetchJobConfigurationPanel` I

será un poco más compleja.

En primer lugar definimos como atributos los elementos que utilizará el panel:

```
private static Logger log =
	Logging.getLogger(PrefetchJobConfigurationPanel.class);

public static final String JOB_STATUS_WAITING = "WAITING";
public static final String JOB_STATUS_RUNNING = "RUNNING";
public static final String JOB_STATUS_ENDED = "ENDED";
public static final String JOB_STATUS_ERROR = "ERROR";

ModalWindow popupWindow;
JobConfigurationEntryProvider entryProvider;
GeoServerTablePanel<PrefetchJobConfigurationEntry> gridsetTable;
```

```
List<PrefetchJobConfigurationEntry> items;
String layerId;
GeoServerDialog adviceDialog;
GeoServerDialog deleteDialog;

int bufferElect;
```

Código 65. Extracto `PrefetchJobConfigurationPanel` II

De manera similar al panel de *Features Sources*, se definen:

- `popupWindow`: ventana emergente para añadir un nuevo trabajo.
- `entryProvider`: el proveedor de entradas de la tabla.
- `gridsetTable`: la propia tabla.
- `items`: la lista de entradas.
- `layerId`: el identificador de la capa/grupo de capas.
- `adviceDialog`, `deleteDialog`: mensajes de aviso emergentes.

A continuación se define el constructor del panel en que se declarará como se debe completar cada una de las columnas de la tabla para cada entrada y un enlace hacia una ventana emergente con el contenido de esta y acciones al realizar como consecuencia de otras acciones en la ventana emergente.

```
public PrefetchJobConfigurationPanel(String id, final List<String>
gridsetList, List<PrefetchJobConfigurationEntry> jobList, final String
layerId) {…}
```

Código 66. Extracto `PrefetchJobConfigurationPanel` III

Primero inicializamos los datos del panel con los argumentos del constructor. Recibiremos la lista de *GridSet* asociados a la capa en cuestión, la lista de trabajos ya configurados (obtenidos ambos de los objetos `GeoServerSeederLayerInfo` del catálogo *Seeder* en el editor del formulario, Sección 4.3.2), y el identificador de la capa.

```
super(id);
items = new ArrayList<PrefetchJobConfigurationEntry>();
this.layerId = layerId;

if (jobList != null) {
        items = jobList;
}
```

Código 67. Extracto `PrefetchJobConfigurationEntry` IV

Si la lista de trabajos ya configurados no está vacía, inicializamos la lista de entradas con ellas. En caso contrario la lista de entradas estará vacía.

A continuación, se añade la tabla que compone el panel y se indica cómo debe completar cada una sus columnas para cada entrada haciendo uso del proveedor de entradas `JobConfigurationEntryProvider`:

```
entryProvider = new JobConfigurationEntryProvider(items);
add(gridsetTable = new GeoServerTablePanel<PrefetchJobConfigurationEntry>
("entry", entryProvider) {
        @Override
        protected Component getComponentForProperty(String id, IModel
        itemModel, Property<PrefetchJobConfigurationEntry> property) {
```

```
                Component result = null;
                if (property == JobConfigurationEntryProvider.JOBNAME) {
                        result = jobNameLink(id, itemModel);
                } else if (property == JobConfigurationEntryProvider.GRIDSET) {
                        result = gridsetLink(id, itemModel);
                } else if (property == JobConfigurationEntryProvider.BBOX) {
                        result = bboxLink(id, itemModel);
                } else if (property == JobConfigurationEntryProvider.RESOLUTION)
                {
                        result = resolutionLink(id, itemModel);
                } else if (property == JobConfigurationEntryProvider.BUFFER) {
                        result = bufferLink(id, itemModel);
                } else if (property == JobConfigurationEntryProvider.
                START_DELAY) {
                        result = delayLink(id, itemModel);
                } else if (property == JobConfigurationEntryProvider.
                REPEAT_COUNT) {
                        result = repeatLink(id, itemModel);
                } else if (property == JobConfigurationEntryProvider.
                REPEAT_INTERVAL) {
                        result = intervalLink(id, itemModel);
                } else if (property == JobConfigurationEntryProvider.FORMAT) {
                        result = formatLink(id, itemModel);
                } else if (property == JobConfigurationEntryProvider.ACTION) {
                        result = actionLink(id, itemModel, new
                ParamResourceModel("seed", this));
                } else {
                        result = null;
                }

                return result;
        }
};.setFilterable(false));
gridsetTable.setItemReuseStrategy(new DefaultItemReuseStrategy());
gridsetTable.setOutputMarkupId(true);
```

Código 68. Extracto `PrefetchJobConfigurationPanel` V

Mediante la estructura `if-else` múltiple se indica cómo se debe ir completando la tabla para cada entrada, es decir, que información se debe mostrar en función de la columna a completar.

El proveedor de entradas `JobConfigurationEntryProvider` se implementa como una clase estática en el mismo objeto (fuera del constructor):

```
static class JobConfigurationEntryProvider extends
GeoServerDataProvider<PrefetchJobConfigurationEntry> {
        public static Property<PrefetchJobConfigurationEntry> JOBNAME = new
        PropertyPlaceholder<PrefetchJobConfigurationEntry>("JobName");

        public static Property<PrefetchJobConfigurationEntry> GRIDSET = new
        PropertyPlaceholder<PrefetchJobConfigurationEntry>("Gridset");

        public static Property<PrefetchJobConfigurationEntry> BBOX = new
        PropertyPlaceholder<PrefetchJobConfigurationEntry>("BoundingBox");

        public static Property<PrefetchJobConfigurationEntry> RESOLUTION = new
        PropertyPlaceholder<PrefetchJobConfigurationEntry>("Resolution");

        public static Property<PrefetchJobConfigurationEntry> BUFFER = new
        PropertyPlaceholder<PrefetchJobConfigurationEntry>("Buffer(m)");

        public static Property<PrefetchJobConfigurationEntry> START_DELAY = new
        PropertyPlaceholder<PrefetchJobConfigurationEntry>("StartDelay");
```

```
        public static Property<PrefetchJobConfigurationEntry> REPEAT_COUNT =
        new PropertyPlaceholder<PrefetchJobConfigurationEntry>("RepeatCount");

        public static Property<PrefetchJobConfigurationEntry> REPEAT_INTERVAL =
        new PropertyPlaceholder<PrefetchJobConfigurationEntry>(
        "RepeatInterval");

        public static Property<PrefetchJobConfigurationEntry> FORMAT = new
        PropertyPlaceholder<PrefetchJobConfigurationEntry>("Format");

        public static Property<PrefetchJobConfigurationEntry> ACTION = new
        PropertyPlaceholder<PrefetchJobConfigurationEntry>("Action");

        static List PROPERTIES = Arrays
                        .asList(JOBNAME, GRIDSET, BBOX, RESOLUTION, BUFFER,
                START_DELAY, REPEAT_COUNT, REPEAT_INTERVAL, FORMAT, ACTION);

        List<PrefetchJobConfigurationEntry> items;

        public JobConfigurationEntryProvider
        (List<PrefetchJobConfigurationEntry> items) {
                this.items = items;
        }

        @Override
        protected List<PrefetchJobConfigurationEntry> getItems() {
                return items;
        }

        @Override
        protected List<Property<PrefetchJobConfigurationEntry>> getProperties()
        {
                return PROPERTIES;
        }

}
```

Código 69. Extracto `PrefetchJobConfigurationPanel` VI

Igual que en el caso del panel de selección de *Features Sources*, nos permite definir las columnas de que va a constar la tabla y el orden en que deben aparecer, así como su título.

El siguiente paso será definir el enlace que nos permite añadir un nuevo trabajo:

```
1   add(popupWindow = new ModalWindow("popup"));
2           add(new AjaxLink("addJob") {
3                   @Override
4                   public void onClick(AjaxRequestTarget target) {
5                   popupWindow.setInitialHeight(200);
6                   popupWindow.setInitialWidth(525);
7                   popupWindow.setTitle(new ParamResourceModel("addJob", this));
8                   popupWindow.setContent(new
9                   GridsetChoicesPanel(popupWindow.getContentId(), gridsetList) {
10                          @Override
11                          protected void handleSelection(String gridsetName,
12                          AjaxRequestTarget target) {
13                                  popupWindow.close(target);
14                                  PrefetchJobConfigurationEntry entry =
15                                  createDefaultGridsetConfigurationEntry(gridsetName
16                                  , layerId);
17                                  entryProvider.getItems().add(entry);
18                                  target.addComponent(gridsetTable);
19                          }
20                  });
```

```
21                  popupWindow.show(target);
22                  }
23  });
```

Código 70. Extracto `PrefetchJobConfigurationPanel` VII

Este enlace, al ser pulsado, lanzará una ventana emergente cuyo contenido se establece mediante el método `setContent()`, es decir, se establece que el contenido será un panel denominado `GridsetChoicesPanel` (analizado a continuación) que presentará al usuario una lista de los *GridSet* configurados para esta capa y permite seleccionar uno. Una vez escogido un *GridSet* de esta lista se lanza el método `handleSelection()` (líneas 11-19) que crea una nueva entrada en la tabla, es decir, un nuevo trabajo. Esta entrada consistirá en un mini-formulario en que el usuario deberá introducir sus datos de configuración. Para ayudar al usuario, se verá, que al cargar un nuevo trabajo, la mayor parte de los campos están rellenados con valores por defecto seleccionados a partir de la información de la capa. De esto se encarga el método `createDefaultGridsetConfigurationEntry()` (se recomienda consultar el código fuente para la lectura de su implementación). Una vez que este método nos ha devuelto una entrada inicializada, la añadimos a la lista de entradas y se recarga el panel.

Antes de seguir analizando el resto del constructor analizamos brevemente el panel `GridsetChoicesPanel` indicado anteriormente. No se entra mucho en detalle ya que la estructura es exactamente la misma que la del resto de paneles que estamos viendo. En primer lugar se definen los elementos del panel (tabla, lista de entradas, líneas 5-9), a continuación en el constructor de la clase se define la tabla y como se deben completar las columnas para cada entrada (líneas 15-46). Ya fuera del constructor se define el proveedor de entradas para definir las columnas de la tabla. Recordar que el método `handleSelection()` no tiene implementación, porque se sobre implementa en el constructor del panel. Este panel consta tan solo de dos columnas por entrada. En la primera se muestra el nombre del *GridSet* y la segunda consiste en un enlace que al ser pulsado lanza el método `handleSelection()` seleccionado el *GridSet* correspondiente para el resto de tareas.

```
1   public class GridsetChoicesPanel extends Panel {
2
3           private static Logger log = Logging.getLogger(StoreStatsPanel.class);
4
5           StoreStatsEntryProvider gridsetProvider;
6           StoreProvider storeProvider = new StoreProvider();
7
8           GeoServerTablePanel<String> table;
9           List<String> items;
10
11          public GridsetChoicesPanel(String id,List<String> gridsetList) {
12                  super(id);
13                  items = gridsetList;
14
15                  gridsetProvider = new StoreStatsEntryProvider(items);
16                  add(table = new GeoServerTablePanel<String>("entry",
17                  gridsetProvider) {
18
19                          @Override
20                          protected Component getComponentForProperty(String id,
21                          IModel itemModel, Property<String> property) {
22                                  final CatalogIconFactory icons =
23                                  CatalogIconFactory.get();
24                                          Component result = null;
25                                          if (property ==
```

```
26                                      StoreStatsEntryProvider.GRIDSET) {
27                                              result = gridsetLink(id,
28                                              itemModel);
29                                      } else if (property ==
30                                      StoreStatsEntryProvider.ACTION) {
31                                              result = resourceChooserLink(id,
32                                              itemModel,
33                                              new ParamResourceModel("select",
34                                              this));
35                                      } else {
36                                              result = null;
37                                      }
38
39                                      return result;
40                              }
41                      }.setFilterable(false));
42                      table.setItemReuseStrategy(new
43                              DefaultItemReuseStrategy());
44                      table.setOutputMarkupId(true);
45
46              }
47
48              public List<String> getEntries() {
49                      return items;
50              }
51
52              SimpleAjaxLink resourceChooserLink(String id, IModel itemModel,
53              IModel label) {
54                      return new SimpleAjaxLink(id, itemModel, label) {
55
56                      @Override
57                      protected void onClick(AjaxRequestTarget target) {
58
59                      String gridsetName = (String) getDefaultModelObject();
60                      handleSelection(gridsetName, target);
61
62                      }
63
64              };
65      }
66
67      Component gridsetLink(String id, IModel itemModel) {
68              String entry = (String) itemModel.getObject();
69              return new Label(id, entry.toString());
70      }
71
72      static class StoreStatsEntryProvider extends
73      GeoServerDataProvider<String> {
74
75              public static Property<String> GRIDSET = new
76              PropertyPlaceholder<String>("Gridset");
77
78
79              public static Property<String> ACTION = new
80              PropertyPlaceholder<String>("Action");
81
82              static List PROPERTIES = Arrays.asList(GRIDSET, ACTION);
83
84              List<String> items;
85
86              public StoreStatsEntryProvider(List<String> items) {
87                      this.items = items;
88              }
89
90              @Override
91              protected List<String> getItems() {
```

```
92                      return items;
93              }
94
95              @Override
96              protected List<Property<String>> getProperties() {
97                      return PROPERTIES;
98              }
99
100     }
101
102     protected void handleSelection(String gridsetName, AjaxRequestTarget
103      target) {}
104 }
```

Código 71. Objeto `GridsetChoicesPanel`

El fichero `GridsetChoicesPanel.html` permite definir la estructura que presentará, utilizando estos objetos *Java*, el panel de selección de *GridSet*:

```
<html>
<body>
<wicket:panel>
        <ul>
                <li>
                        <div wicket:id="entry"></div>
                </li>
        </ul>
        <div wicket:id="popup"></div>
</wicket:panel>
</body>
</html>
```

Código 72. Fichero `GridsetChoicesPanel.html`

Por último definimos dos diálogos de aviso:

```
add(adviceDialog = new GeoServerDialog("adviceDialog"));
adviceDialog.setInitialWidth(500);
adviceDialog.setInitialHeight(150);

add(deleteDialog = new GeoServerDialog("deleteDialog"));
deleteDialog.setInitialWidth(500);
deleteDialog.setInitialHeight(75);
```

Código 73. Extracto `PrefetchJobConfigurationPanel` VIII

Con los métodos `setInitialWidth()` y `setInitialHeight()` se establece el tamaño inicial de la ventana emergente en que consiste el aviso. Más adelante se verá cual es la utilización de estas ventanas emergentes.

De esta manera terminamos el constructor del panel.

Definimos el método mediante el cual se puede obtener la lista de entradas de la tabla:

```
public List<PrefetchJobConfigurationEntry> getEntries() {
        return items;
}
```

Código 74. Extracto `PrefetchJobConfigurationPanel` IX

El siguiente paso va a ser centrarnos en el conjunto de métodos que permiten componer la información que debe aparecer en cada celda de la tabla. Para este panel las celdas contendrán elementos de formulario para que el usuario introduzca datos o seleccione opciones. Se analizarán en el orden en que irán apareciendo.

- **Nombre del trabajo**: en este apartado el usuario introducirá el nombre del trabajo con el objetivo de diferenciarlo del resto de trabajos. Esta campo no se inicializa mediante el método `createDefaultGridsetConfigurationEntry()`, pero es obligatorio rellenarlo.

Se implementa mediante una caja de texto (`TextField`). Aparecerá en este y también en el resto de celdas que analicemos, por lo que explicamos aquí el significado de las líneas 13-23. En caso de que el contenido de esta caja de texto cambie, se lanza el método `onUpdate()`, con el objetivo de permitir que el usuario pueda lanzar un trabajo sin guardar su configuración en el catálogo (es decir, pulsar el botón de guardar o enviar el formulario general). De esta manera se establece, en este caso, como nombre del trabajo para esta entrada, el que el usuario haya introducido, para después poder acceder a él.

```
1  Component jobNameLink(String id, IModel itemModel) {
2          final PrefetchJobConfigurationEntry entry =
3                 (PrefetchJobConfigurationEntry) itemModel.getObject();
4
5          Fragment f = new Fragment(id, "jobName",
6                 PrefetchJobConfigurationPanel.this);
7
8          IModel<String> model = new PropertyModel<String>(itemModel,
9                 "jobName");
10         final Component jobName = new TextField<String>("jobName", model);
11         f.add(jobName);
12
13         jobName.add(new AjaxFormComponentUpdatingBehavior("onChange") {
14                @Override
15                protected void onUpdate(AjaxRequestTarget target) {
16
17                        String nameString =
18                              jobName.getDefaultModelObjectAsString();
19                       entry.setJobName(nameString);
20                       target.addComponent(jobName);
21
22                       }
23                });
24                return f;
25 }
```

Código 75. Extracto `PrefetchJobConfigurationPanel` X

El parámetro "`jobName`" permite asociar esta caja de texto con el atributo del mismo nombre del objeto `PrefetchingJobConfigurationEntry`.

- ***GridSet* seleccionado para el trabajo**: simplemente es una línea de texto en la que se muestra el nombre del *GridSet* seleccionado en el panel emergente de adicción de un nuevo trabajo:

```
Component gridsetLink(String id, IModel itemModel) {
       PrefetchJobConfigurationEntry entry = (PrefetchJobConfigurationEntry)
              itemModel.getObject();
       return new Label(id, entry.getGridsetName());
}
```

Código 76. Extracto `PrefetchJobConfigurationPanel` XI

- **Encuadre o *Bounding Box***. El encuadre (Ver sección 2.4. Conceptos) se especifica mediante cuatro coordenadas `maxX`, `minX`, `maxY` y `minY`. Para cada entrada esta celda consistirá en cuatro cajas de texto para que el usuario especifique estas cuatro coordenadas.

En este caso las cajas de texto se implementan mediante el objeto de *GeoServer* `DecimalTextField` ya que los datos introducidos solo pueden ser números admitiendo decimales, de esta manera si se introduce una cadena no numérica no la hará caso. En caso de intentar guardar la información se encargará automáticamente de lanzar un mensaje de error si el dato introducido no es numérico. Si se lanza un trabajo con un parámetro de *Bounding Box* no numérico se coge como parámetro el anterior.

Además el propio *GeoServer* se encarga de "generar" un *Bounding Box* correcto si el introducido tiene inconsistencias, el ejemplo típico, que se haya dado un valor mínimo para un eje mayor que el máximo, en este caso el valor máximo se pondrá a cero. Estos efectos solo se computan a la hora de procesar el trabajo, no se aprecia en el formulario.

```
Component bboxLink(String id, IModel itemModel) {
        final PrefetchJobConfigurationEntry entry =
        (PrefetchJobConfigurationEntry) itemModel.getObject();

        Fragment f = new Fragment(id, "bbox",
                PrefetchJobConfigurationPanel.this);
        IModel<Double> modelMaxX = new PropertyModel<Double>(itemModel,
                "bbox.maxx");
        IModel<Double> modelMinX = new PropertyModel<Double>(itemModel,
                "bbox.minx");

        IModel<Double> modelMaxY = new PropertyModel<Double>(itemModel,
                "bbox.maxy");
        IModel<Double> modelMinY = new PropertyModel<Double>(itemModel,
                "bbox.miny");

        final DecimalTextField maxx = new DecimalTextField("maxx", modelMaxX);
        final DecimalTextField minx = new DecimalTextField("minx", modelMinX);
        final DecimalTextField maxy = new DecimalTextField("maxy", modelMaxY);
        final DecimalTextField miny = new DecimalTextField("miny", modelMinY);

        f.add(maxx);
        f.add(minx);
        f.add(maxy);
        f.add(miny);

        f.add(new Label("labelmaxx", "Max X"));
        f.add(new Label("labelminx", "Min X"));
        f.add(new Label("labelmaxy", "Max Y"));
        f.add(new Label("labelminy", "Min Y"));

        maxx.add(new AjaxFormComponentUpdatingBehavior("onChange") {

                @Override
                protected void onUpdate(AjaxRequestTarget target) {
                        String maxxString =
                                maxx.getDefaultModelObject().toString();
                        double maxxDouble = Double.parseDouble(maxxString);
                        entry.setMaxX(maxxDouble);
                        target.addComponent(maxx);

                }
        });
```

```
		minx.add(new AjaxFormComponentUpdatingBehavior("onChange") {

			@Override
			protected void onUpdate(AjaxRequestTarget target) {
				String minxString =
					minx.getDefaultModelObject().toString();
				double minxDouble = Double.parseDouble(minxString);
				entry.setMinX(minxDouble);
				target.addComponent(minx);

			}
		});

		maxy.add(new AjaxFormComponentUpdatingBehavior("onChange") {

			@Override
			protected void onUpdate(AjaxRequestTarget target) {
				String maxyString =
					maxy.getDefaultModelObject().toString();
				double maxyDouble = Double.parseDouble(maxyString);
				entry.setMaxY(maxyDouble);
				target.addComponent(maxy);
			}
		});

		miny.add(new AjaxFormComponentUpdatingBehavior("onChange") {

			@Override
			protected void onUpdate(AjaxRequestTarget target) {
				String minyString =
					miny.getDefaultModelObject().toString();
				double minyDouble = Double.parseDouble(minyString);
				entry.setMinY(minyDouble);
				target.addComponent(miny);
			}
		});
		return f;
	}
```

Código 77. Extracto `PrefetchJobConfigurationPanel` XII

Se puede apreciar que cada una de las cuatro coordenadas del encuadre se tratan por separado.

Cuando se actualiza el valor de cada coordenada el método `onUpdate()`, coge el valor de la caja de texto. Todo parámetro cogido de una caja de texto, independientemente del tipo, es una cadena de texto (*string*), como en este caso el valor queremos que se trate como un decimal (*double*) el método `parseDouble()` nos permite transformar el parámetro en formato *string* a formato *double* y establecerlo como parámetro de la entrada.

La sentencia `target.addComponent()` permite añadir de nuevo el elemento a la tabla, básicamente actualizarlo. La principal función de esta línea será expresar correctamente el valor introducido, por ejemplo, dependiendo del idioma de la interfaz si se introduce 0.1 el punto (“.”) no se identifica como coma decimal por ello al introducir 0.1 en una caja de texto y salir de ella, automáticamente el valor que aparecerá será 1. Si se introduce 0,1 la interfaz sí lo identificará como una coma decimal y al actualizarlo lo dejará igual.

Este funcionamiento permite que el usuario sepa cual son los valores que realmente va a utilizar la interfaz y no haya ninguna confusión. Sobre todo el problema puede venir del idioma

en que se esté utilizando la interfaz debido al diferente tratamiento de la coma (“,”) y el punto (“.”), por ejemplo, entre la notación inglesa y la española. En la notación inglesa 10,000 (coma) corresponde a diez mil, lo que en la notación española se representa con 10.000 (punto). En la notación española 10,000 (coma) corresponde a diez con tres ceros decimales.

Este campo se encuentra inicializado por defecto por medio del método `createDefaultGridsetConfigurationEntry()`. En concreto se utiliza como encuadre por defecto el encuadre Lat/Lot configurado para la capa en la pestaña “Datos”.

Los parámetros “`bbox.maxx`”, “`bbox.minx`”, “`bbox.maxy`” y “`bbox.miny`” permiten asociar las correspondientes cajas de texto con las cuatro componentes del atributo `bbox` del objeto `PrefetchingJobConfigurationEntry`.

- **Resolución o *zoom***: es un valor numérico mediante el cual el usuario especifica cuanto de cerca quiere analizar el mapa. Cuanto mayor sea su valor, más cerca. Se implementa mediante un selector desplegable (`DropDownChoice`) que maneja números enteros.

```
1   Component resolutionLink(String id, IModel itemModel) {
2           final PrefetchJobConfigurationEntry entry =
3           (PrefetchJobConfigurationEntry) itemModel.getObject();
4           GWC gwc = GWC.get();
5           List<GridSet> availableGridSet = gwc.getGridSetBroker().getGridSets();
6           int numLevel = 0;
7           for (int i = 0; i < availableGridSet.size(); i++) {
8                   if (availableGridSet.get(i).getName()
9                   .equals(entry.getGridsetName())) {
10                          numLevel = availableGridSet.get(i).getNumLevels();
11                  }
12          }
13          List<Integer> availableResolution = new ArrayList<Integer>();
14          for (int i = 0; i < numLevel; i++) {
15                  availableResolution.add(i);
16          }
17          Fragment f = new Fragment(id, "resolution",
18                  PrefetchJobConfigurationPanel.this);
19          IModel<Integer> model = new PropertyModel<Integer>(itemModel,
20                  "resolution");
21          final Component resolution = new DropDownChoice<Integer>("resolution",
22                  model, availableResolution);
23          f.add(resolution);
24
25          resolution.add(new AjaxFormComponentUpdatingBehavior("onChange") {
26
27                  @Override
28                  protected void onUpdate(AjaxRequestTarget target) {
29
30                  String resolutionString =
31                          resolution.getDefaultModelObject().toString();
32                  int resolutionInt = Integer.parseInt(resolutionString);
33                                  entry.setResolution(resolutionInt);
34                          }
35                  });
36                  return f;
37      }
```

Código 78. Extracto `PrefetchJobConfigurationPanel` XIII

El conjunto de números entre los que puede elegir el usuario dependerá del *GridSet* seleccionado, es decir, la resolución dependerá del *GridSet*. Las líneas 7-12 permiten obtener

el número de niveles de zoom asociados al *GridSet* seleccionado para este trabajo. Las líneas 13-16 permiten crear la lista de opciones entre las que puede seleccionar el usuario. Esta lista estará conformada por los números enteros desde 0 hasta el número de niveles que especifique el *GridSet*.

El resto del comportamiento es similar a lo explicado para los elementos anteriores.

Este campo se encuentra inicializado mediante el método `createDefaultGridsetConfigurationEntry()` a un valor 13.

El parámetro "`resolution`" permite asociar esta caja de texto con el atributo del mismo nombre del objeto `PrefetchingJobConfigurationEntry`.

- ***Buffer***. Ya en la sección 2.4. Conceptos se explicó que íbamos a entender por *buffer*: exceso con respecto a la localización concreta sobre la que también se quiere realizar el análisis. El usuario introducirá el *buffer* en metros, tal y como se le indica en la interfaz. Después será el módulo *Seeder-Prefetch* el que se encargue de transformarlo a las unidades de medición concretas del *GridSet* correspondiente.

```
Component bufferLink(String id, IModel itemModel) {
	final PrefetchJobConfigurationEntry entry =
		(PrefetchJobConfigurationEntry) itemModel.getObject();
	Fragment f = new Fragment(id, "buffer",
		PrefetchJobConfigurationPanel.this);
	IModel<Double> model = new PropertyModel<Double>(itemModel, "buffer");
	final Component buffer = new DecimalTextField("buffer", model);
	f.add(buffer);

	buffer.add(new AjaxFormComponentUpdatingBehavior("onChange") {

		@Override
		protected void onUpdate(AjaxRequestTarget target) {
			String bufferString =
				buffer.getDefaultModelObject().toString();
			Double bufferDouble = Double.parseDouble(bufferString);
			entry.setBuffer(bufferDouble);
			target.addComponent(buffer);

		}
	});
	return f;
}
```

Código 79. Extracto `PrefetchJobConfigurationPanel` XIV

También se deberá tener en cuenta el objetivo de la sentencia `target.addComponent()` para que el usuario sepa realmente que dato ha introducido (recordar los problemas con la coma o el punto según el idioma de la interfaz).

Este campo se encuentra inicializado por defecto mediante el método `createDefaultGridsetConfigurationEntry()` a un valor de 10000 metros.

El parámetro "`buffer`" permite asociar esta caja de texto con el atributo del mismo nombre del objeto `PrefetchingJobConfigurationEntry`.

- **Retardo inicial**: especifica el tiempo que deberá transcurrir desde que se pulsa el botón de lanzar trabajo hasta que este se inicializa. Esta celda se implementa mediante una caja de texto de valor numéricos (`TextField<Integer>`) y un selector desplegable (`DropDownChoices`) de tipo *string* (cadenas de texto) en que el usuario deberá especificar las unidades en que se debe medir el número introducido en la caja de texto previa. Si el valor introducido en la caja de texto no es un valor numérico entero tomará como valor válido para lanzar un trabajo, el último entero introducido.

```
Component delayLink(String id, IModel itemModel) {
	final PrefetchJobConfigurationEntry entry =
		(PrefetchJobConfigurationEntry) itemModel.getObject();

	Fragment f = new Fragment(id, "delay",
		PrefetchJobConfigurationPanel.this);
	IModel<Integer> model = new PropertyModel<Integer>(itemModel,
		"startDelay");
	final Component delay = new TextField<Integer>("delay", model);
	f.add(delay);

	List<String> availableUnit = new ArrayList<String>();
	availableUnit.add("msecs");
	availableUnit.add("seconds");
	availableUnit.add("minutes");
	availableUnit.add("hours");
	availableUnit.add("days");
	availableUnit.add("weeks");
	availableUnit.add("months");
	availableUnit.add("years");

	IModel<String> modelUnit = new PropertyModel<String>(itemModel,
		"startDelayUnit");
	final Component startDelayUnit = new DropDownChoice<String>
		("startDelayUnit", modelUnit, availableUnit);
	f.add(startDelayUnit);

	delay.add(new AjaxFormComponentUpdatingBehavior("onChange") {

		@Override
		protected void onUpdate(AjaxRequestTarget target) {

			String delayString =
				delay.getDefaultModelObject().toString();
			int delayInt = Integer.parseInt(delayString);
			entry.setStartDelay(delayInt);
			target.addComponent(delay);

		}
	});

	startDelayUnit.add(new AjaxFormComponentUpdatingBehavior("onChange") {

		@Override
		protected void onUpdate(AjaxRequestTarget target) {
			String startDelayUnitString =
				startDelayUnit.getDefaultModelObject().toString();
			entry.setStartDelayUnit(startDelayUnitString);
		}
	});
	return f;
}
```

Código 80. Extracto `PrefetchJobConfigurationPanel` XV

La diferencia con los componentes anteriores es que la creación de la lista de opciones del selector de unidades.

Estos campos se encuentran inicializados por defecto mediante el método `createDefaultGridsetConfigurationEntry()` a un valor de 0 y milisegundos respectivamente.

Los parámetros "`startDelay`" y "`startDelayUnit`" permiten asociar la caja de texto y la selección del selector desplegable con los atributos del mismo nombre del objeto `PrefetchingJobConfigurationEntry`.

- **Repetición**: número de veces que deberá repetirse la tarea de *prefetch*. Se ha establecido un valor 0 para repetir indefinidamente, 1 para no repetir y una cantidad mayor que uno para indicar un número concreto de repeticiones.

En primer lugar se muestra el usuario un mensaje para avisarle que si quiere que se repita indefinidamente deberá introducir un 0. Esta celda se implementa mediante una caja de texto de valores enteros. Si el usuario introduce un número que no es entero, lo ignorará y tomará como valor válido el último entero introducido.

```
Component repeatLink(String id, IModel itemModel) {
	final PrefetchJobConfigurationEntry entry =
		(PrefetchJobConfigurationEntry) itemModel.getObject();

	Fragment f = new Fragment(id, "repeat",
		PrefetchJobConfigurationPanel.this);
	IModel<Integer> model = new PropertyModel<Integer>(itemModel,
		"repeatCount");
	final Component repeat = new TextField<Integer>("repeat", model);

	f.add(repeat);

	repeat.add(new AjaxFormComponentUpdatingBehavior("onChange") {

		@Override
		protected void onUpdate(AjaxRequestTarget target) {
			String repeatString =
				repeat.getDefaultModelObject().toString();
			int repeatInt = Integer.parseInt(repeatString);
			entry.setRepeatCount(repeatInt);
			target.addComponent(repeat);
		}
	});
	return f;
}
```

Código 81. Extracto `PrefetchJobConfigurationPanel` XVI

Este campo se encuentra inicializado por defecto mediante el método `createDefaultGridsetConfigurationEntry()` a un valor de 1 (sin repetición).

El parámetro "`repeat`" permite asociar la caja de texto el atributo del mismo nombre del objeto `PrefetchingJobConfigurationEntry`.

- **Intervalo de repetición**: en caso de que el usuario indique un valor distinto de 1 en el número de repeticiones del análisis, es decir, el análisis deba repetirse al menos una vez,

también deberá especificar el tiempo entre análisis, entendido como el tiempo que transcurre entre que se lanza una repetición y la siguiente, y no entre que termina una y empieza la siguiente. Por ello el usuario deberá tener cuidado de poner un tiempo demasiado pequeño que haga que los análisis se solapen, aunque el módulo *Seeder-Prefetch* se encargará de no lanzar una repetición si hay otra en ejecución, como se verá más adelante.

De manera análoga a la especificación del retardo inicial, la celda constará de una caja de texto para dar un valor numérico entero (igual que antes si el valor introducido no es entero, se coge como valor el último entero introducido) y un selector desplegable para escoger las unidades en que se mide esa cantidad.

```
Component intervalLink(String id, IModel itemModel) {
	final PrefetchJobConfigurationEntry entry =
		(PrefetchJobConfigurationEntry) itemModel.getObject();

	Fragment f = new Fragment(id, "interval",
		PrefetchJobConfigurationPanel.this);
	IModel<Integer> model = new PropertyModel<Integer>(itemModel,
		"repeatInterval");
	final Component interval = new TextField<Integer>("interval", model);
	f.add(interval);

	List<String> availableUnit = new ArrayList<String>();
	availableUnit.add("msecs");
	availableUnit.add("seconds");
	availableUnit.add("minutes");
	availableUnit.add("hours");
	availableUnit.add("days");
	availableUnit.add("weeks");
	availableUnit.add("months");
	availableUnit.add("years");

	IModel<String> modelUnit = new PropertyModel<String>(itemModel,
		"repeatIntervalUnit");
	final Component intervalUnit = new DropDownChoice<String>
		("intervalUnit", modelUnit, availableUnit);
	f.add(intervalUnit);

	interval.add(new AjaxFormComponentUpdatingBehavior("onChange") {

		@Override
		protected void onUpdate(AjaxRequestTarget target) {
			String intervalString =
				interval.getDefaultModelObject().toString();
			int intervalInt = Integer.parseInt(intervalString);
			entry.setRepeatInterval(intervalInt);
			target.addComponent(interval);
		}
	});

	intervalUnit.add(new AjaxFormComponentUpdatingBehavior("onChange") {

		@Override
		protected void onUpdate(AjaxRequestTarget target) {
			String intervalUnitString =
				intervalUnit.getDefaultModelObject().toString();
			entry.setRepeatIntervalUnit(intervalUnitString);
		}
	});
	return f;
}
```

Código 82. Extracto `PrefetchJobConfigurationPanel` XVII

Estos campos se encuentran inicializados por defecto mediante el método `createDefaultGridsetConfigurationEntry()` a un valor de 5000 y milisegundos respectivamente.

Los parámetros "`repeatInterval`" y "`repeatIntervalUnit`" permiten asociar la caja de texto y la selección del selector desplegable con los atributos del mismo nombre del objeto `PrefetchingJobConfigurationEntry`.

- **Formato de la imagen**: especifica el formato de las imágenes que se deben utilizar en el análisis. Es decir, el análisis se basa en acceder a la base de datos y trabajar sobre las estadísticas de acceso a una determinada capa (la que estamos tratando), cuando ha sido accedida o visualizada. Para poder visualizarlas el usuario accede mediante una pareja de valores *GridSet*-formato de la imagen. El formato se especifica para indicar en que formato de imagen (*jpg* o *png*) se guardarán las teselas visitadas en el ordenador del usuario. Al lanzar un trabajo de análisis permitiremos al usuario que diferencie entre accesos a la capa especificando uno u otro formato de imagen.

Esta celda se implementa mediante un selector desplegable en que se da a elegir al usuario entre las opciones disponibles. Estas se obtienen haciendo uso de las herramientas de *GeoServer*:

```
Component formatLink(String id, IModel itemModel) {
        final PrefetchJobConfigurationEntry entry =
                (PrefetchJobConfigurationEntry) itemModel.getObject();
        LayerInfo layerInfo = GWC.get().getCatalog().getLayer(layerId);
        LayerGroupInfo layerGroupInfo =
                GWC.get().getCatalog().getLayerGroup(layerId);
        TileLayer tileLayer;
        if (layerInfo != null) {
                tileLayer = GWC.get().getTileLayer(layerInfo);
        } else
                tileLayer = GWC.get().getTileLayer(layerGroupInfo);
        List<MimeType> mimeTypeList = tileLayer.getMimeTypes();
        List<String> formatList = new ArrayList<String>();
        for (int i = 0; i < mimeTypeList.size(); i++) {
                String format = mimeTypeList.get(i).getFormat();
                formatList.add(format);
        }

        Fragment f = new Fragment(id, "format",
                PrefetchJobConfigurationPanel.this);
        IModel<String> model = new PropertyModel<String>(itemModel, "format");
        final Component format = new DropDownChoice<String>("format", model,
                formatList);
        f.add(format);

        format.add(new AjaxFormComponentUpdatingBehavior("onChange") {

                @Override
                protected void onUpdate(AjaxRequestTarget target) {
                        String formatString =
                                format.getDefaultModelObject().toString();
                        entry.setFormat(formatString);
                }
        });
        return f;
}
```

Código 83. Extracto `PrefetchJobConfigurationPanel` XVIII

Este campo no se encuentra inicializado por defecto. El parámetro "`format`" permite asociar la caja de texto el atributo del mismo nombre del objeto `PrefetchingJobConfigurationEntry`.

- La última columna de cada entrada permite realizar **acciones** con esta configuración: borrarla, lanzar el trabajo como *Seed* y lanzar el trabajo como *ReSeed*. Ya se indicó al inicio de esta sección la diferencia entre lanzar el trabajo como *Seed* o como *ReSeed*. Para realizar cada una de estas acciones se implementa un enlace distinto. El enlace de borrado se implementa mediante un icono (líneas 8-49), y los enlaces a los lanzamientos de tareas como enlaces de texto (línea 52 para *Seed* y línea 53 para *ReSeed*).

```
1  Component actionLink(String id, IModel itemModel, IModel label) {
2         final PrefetchJobConfigurationEntry entry =
3                 (PrefetchJobConfigurationEntry) itemModel.getObject();
4
5         Fragment f = new Fragment(id, "action",
6         PrefetchJobConfigurationPanel.this);
7
8         ImageAjaxLink link = new ImageAjaxLink("delete", new
9          ResourceReference(getClass(), "../../../img/icons/silk/delete.png")) {
10               @Override
11               protected void onClick(AjaxRequestTarget target) {
12                      deleteDialog.setTitle(new Model<String>("Confirm removal"
13                      + " of Job Configuration?"));
14               deleteDialog.showOkCancel(target, new
15                       GeoServerDialog.DialogDelegate() {
16                              private static final long serialVersionUID = 1L;
17
18                              @Override
19                              protected Component getContents(final String id) {
20                                     return new Label(id, "Are you sure to "
21                                     + "remove Job Configuration '"
22                                     + entry.getJobName() + "'?");
23                              }
24
25                              @Override
26                              protected boolean onSubmit(final AjaxRequestTarget
27                               target, final Component contents) {
28                                             items.remove(entry);
29                                             return true;
30                              }
31
32                              @Override
33                              public void onClose(final AjaxRequestTarget
34                               target) {
35                                     target.addComponent(gridsetTable);
36                              }
37
38                              @Override
39                              protected boolean onCancel(final AjaxRequestTarget
40                               target) {
41                                     final boolean closeWindow = true;
42                                     return closeWindow;
43                                     }
44                              });
45
46                      }
47               };
48               link.getImage().add(new AttributeModifier("alt", true, new
49                ParamResourceModel("AbstractLayerGroupPage.th.remove", link)));
50
51               f.add(link);
```

```
52                  f.add(seedLink("seed", itemModel, label, false));
53                  f.add(seedLink("reseed", itemModel, label, true));
54
55                  return f;
56          }
```

Código 84. Extracto PrefetchJobConfigurationPanel XIX

En caso de que el usuario pulse sobre el icono de borrar se lanza el método onClick() (líneas 11-47). Esto provocará que salte un dialogo de confirmación que preguntará al usuario si está de acuerdo con borrar esta entrada de la tabla. Aquí es donde se usa el objeto deleteDialog() definido en el constructor. En caso de pulsar sobre el enlace de confirmación ("*Ok*") borrará la entrada de la tabla y de la lista de entradas (líneas 26-30). En caso de pulsar sobre el enlace de cancelación ("*Cancel*") se cierra la ventana emergente y no realiza ninguna tarea más.

Por otro lado, las acciones a realizar al pulsar sobre los enlaces *Seed/ReSeed* se definen en la función seedLink() (líneas 52-53). La misma para ambos enlaces, para diferenciar entre uno y otro está el último parámetro que valdrá false para la tarea de *Seed* y true para la tarea de *ReSeed*.

Una vez ya completada la explicación de la implementación de los valores de las celdas, se procede a explicar la estructura del método seedLink() que se acaba de enumerar. Es una función un tanto compleja: implementa un enlace de texto que al ser pulsado coge todos los parámetros introducidos en el "mini-formulario" de la entrada correspondiente así como otros parámetros del formulario también necesarios, comprueba que todos los valores introducidos son correctos (se han rellenado campo obligatorios, los datos introducidos se adaptan a los requisitos…). Una vez pasada esta fase, se programa el calendario de lanzamiento de las tareas, haciendo uso del servicio *Quartz* que permite crear programadores y asignarles trabajos.

Vamos a ir analizándolo poco a poco:

```
SimpleAjaxLink seedLink(String id, final IModel itemModel, IModel label, final
boolean reseed) {…}
```

Código 85. Extracto PrefetchJobConfigurationPanel XXI

En primer lugar implementamos el nombre del enlace, aquí es donde tiene sentido el booleano indicado antes que diferenciaba los enlaces:

```
final String cadena;
if (reseed) {
        cadena = "re";
} else {
        cadena = "";
}
final IModel<String> labelModel = new ResourceModel(cadena + "seed");
```

Código 86. Extracto PrefetchJobConfigurationPanel XXII

Ahora implementamos el enlace:

```
return new SimpleAjaxLink(id, itemModel, labelModel) {
        @Override
        protected void onClick(AjaxRequestTarget target) {…}
};
```

Código 87. Extracto `PrefetchJobConfigurationPanel` XXIII

En el método `onClick()` definiremos todas las acciones que deben realizar al pulsar el correspondiente enlace. Las primeras líneas obtienen la fecha en que se lanzó el trabajo, medido en el número de segundos transcurridos desde el 1 de enero de 1970 (línea 1). También obtenemos la entrada sobre la que se lanzó el trabajo, para de ella poder obtener los parámetros necesarios para lanzar la tarea (líneas 3-4). Por último obtenemos el *Tile Layer* asociado a la capa/grupo de capas sobre la que se lanza el trabajo:

```
1  Date date = new Date(System.currentTimeMillis());
2
3  PrefetchJobConfigurationEntry entry = (PrefetchJobConfigurationEntry)
4  itemModel.getObject();
5
6  LayerInfo layerInfo = GWC.get().getCatalog().getLayer(layerId);
7  LayerGroupInfo layerGroupInfo = GWC.get().getCatalog().getLayerGroup(layerId);
8  TileLayer tileLayer;
9  if (layerInfo != null) {
10         tileLayer = GWC.get().getTileLayer(layerInfo);
11 } else
12         tileLayer = GWC.get().getTileLayer(layerGroupInfo);
```

Código 88. Extracto `PrefetchJobConfigurationPanel` XXIV

A continuación se procede a evaluar que los valores introducidos son correctos, en caso de error lanzará un mensaje de advertencia. Aquí es donde se usa el objeto `adviceDialog()` que se definió en el constructor, para lanzar una ventana emergente donde mostrar al usuario un mensaje de error personalizado al error encontrado. La sentencia `return` permite terminar la ejecución de código en caso de error.

- Se comprueba que se ha dado un nombre al trabajo que se quiere lanzar:

```
if (entry.getJobName() == null || entry.getJobName().equals("")) {
        adviceDialog.setTitle(new Model<String>("Info"));
        IModel<String> modelHeading = new Model<String>("Incorrect
        Parameters");
        IModel<String> modelMessages = new Model<String>("Enter a name for the
        job");
        adviceDialog.showInfo(target, modelHeading, modelMessages);
        log.warning("Incorrect value: not job name");
        return;
}

log.info("(" + tileLayer.getName() + ":" + entry.getGridsetName() + ":" +
entry.getFormat() + "). Checking Prefetching Job Configuration values...");
```

Código 89. Extracto `PrefecthJobConfigurationPanel` XV

- Se comprueba si se ha seleccionado una base de datos para las estadísticas y se ha configurado correctamente para esta capa/grupo de capas. El usuario deberá tener en cuenta que para que se use la base de datos seleccionada en la tabla correspondiente deberá haber

guardado la configuración *Seeder* ya que si no se intentará usar la última guardada, en caso de no haberse guardado ninguna, salta un error:

```
Map<String, IndexDAO> indexMap = Seeder.get().getIndexMap();
if (!indexMap.containsKey(layerId)) {
	adviceDialog.setTitle(new Model<String>("Info"));
	IModel<String> modelHeading = new Model<String>("Incorrect " +
		"Parameters");
	IModel<String> modelMessages = new Model<String>(
		"There is not a data base configurated for this layer." +
		"Please save configuration before seed");
	adviceDialog.showInfo(target, modelHeading, modelMessages);
	log.warning("Incorrect value: not data base configurated");
	return;
}
```

Código 90. Extracto `PrefetchJobConfigurationPanel` XXVI

Aquí se hace uso de un mapa, que se verá posteriormente, en que se guardan correspondencias entre cada capa/grupo de capas y su conexión a la base de datos seleccionada. Si nunca se ha configurado una base de datos para esta capa, este registro (mapa) no contendrá una pareja de elementos cuya clave sea el identificador de la capa/grupo de capas en cuestión.

```
public Map<String, IndexDAO> getIndexMap() {
	return indexMap;
}
```

Código 91. Método `getIndexMap()` del objeto `Seeder`

- Se comprueba si se ha seleccionado al menos un *Feature Source* válido (tipo *vector*) tal y como se explicó en la Sección 4.3.3.1.

```
FeatureCatalog entryList = Seeder.get().constructFeatureCatalog(layerId);
if (entryList.getMap().isEmpty()) {
	adviceDialog.setTitle(new Model<String>("Info"));
	IModel<String> modelHeading = new Model<String>("Incorrect " +
		"Parameters");
	IModel<String> modelMessages = new Model<String>("Feature Source " +
		"List must contain at least a correct feature source");
	adviceDialog.showInfo(target, modelHeading, modelMessages);
	log.warning("Incorrect value: empty Feature list");
	return;
}
```

Código 92. Extracto `PrefetchJobConfigurationPanel` XXVII

El método `constructFeatureCatalog()` del objeto `Seeder` permite generar un objeto de tipo `FeatureCatalog` (Ver Sección 4.4.1) en el que se almacenan todos los *Features Source* seleccionados por el usuario, los que almacenan datos de tipo *vector*; tiene la siguiente estructura:

```
public FeatureCatalog constructFeatureCatalog(String layerId) {

	List<AbstractFSEntry> entryList =
	Seeder.get().getFeatureList().get(layerId).getEntries();
	Map<String, FeatureCollection> map = new HashMap<String,
	FeatureCollection>();
	for (int i = 0; i < entryList.size(); i++) {
		AbstractFSEntry entry = entryList.get(i);
```

```
                String entryId = entry.getEntryId();
                if (entry instanceof LayerEntry) {

                        LayerInfo layerInfo =
                        GWC.get().getCatalog().getLayer(entryId);
                        Type type = layerInfo.getType();

                        if (type.toString().equals("VECTOR")) {
                                if (layerInfo.getResource() instanceof
                                FeatureTypeInfo) {
                                        FeatureTypeInfo resource =
                                        (FeatureTypeInfo) layerInfo.getResource();
                                        try {
                                        FeatureSource fs =
                                        resource.getFeatureSource(null, null);
                                        FeatureCollection fc = fs.getFeatures();
                                        String key = fs.getName().getURI();
                                        map.put(key, fc);
                                        } catch (IOException e) {
                                                logger.log(Level.WARNING, "Unable
                                                to create Feature Catalog", e);
                                        }

                                }

                        }

                } else {

                        String fsName = entry.getfsName();
                        FeatureTypeInfo fti =
                        GWC.get().getCatalog().getFeatureTypeByName(fsName);
                        if (fti != null) {

                                try {
                                FeatureSource fs = fti.getFeatureSource(null,
                                null);
                                FeatureCollection fc = fs.getFeatures();
                                String key = fs.getName().getURI();
                                map.put(key, fc);
                                } catch (IOException e) {
                                        logger.log(Level.WARNING, "Unable to create
                                        Feature Catalog", e);
                                }
                        }

                }

        }

        FeatureCatalog featureCatalog = new FeatureCatalog();
        featureCatalog.setMap(map);

        return featureCatalog;
    }
```

Código 93. Extracto `PrefetchJobConfigurationPanel` XXVIII

Mediante las primeras líneas se obtienen los *Features Sources* que el usuario ha seleccionado para esta capa. Para ello se busca en la lista de paneles `FeatureSourcePanel` registrados el correspondiente a esta capa/grupo de capas. El objeto `FeatureCatalog` consiste en un mapa en que se van almacenando correspondencias entre nombre identificador de un *Feature*

Source y el propio *Feature Source*. Se inicializa un mapa con estas consideraciones, inicialmente vacío.

De este panel se obtiene cada una de las entradas. Como se explicó en la sección 4.3.3.1 las entradas a esta tabla pueden ser de dos tipos `LayerEntry` y `FeatureSourceEntry`. La sentencia `if-else` nos permite realizar un procesamiento distinto en función del tipo de entrada. En caso de que sea de tipo `LayerEntry` se obtiene, mediante las utilidades de *GeoServer,* el `layerInfo` asociado a esta capa y a partir de él, su tipo, en forma de un *string*. Si este es igual a *"VECTOR"* los datos son de tipo *vector*, el siguiente paso es comprobar que el origen de datos en que se encuentra esta capa es de tipo `FeatureSourceInfo` y a partir de añado al mapa el nombre (key) del *Feature Source* y el propio *Feature Source* al que pertenece esta capa publicada.

En caso de que la entrada sea de tipo `FeatureSourceEntry`, se trata de obtener del catálogo de *GeoServer* un objeto de tipo `FeatureSourceInfo` a partir del identificador almacenado en la entrada. Si el objeto devuelto no es nulo, significa que el *Feature Source* seleccionado se adapta al tipo que de orígenes de fenómenos que se permite, así que se añade al mapa.

Una vez completado el mapa, se instancia un objeto de tipo `FeatureCatalog` y se le añade este mapa.

De vuelta al panel de configuración de trabajos de *Prefetch*, si este catálogo devuelto está vacío se lanza un mensaje de error al usuario.

- Se comprueba si el valor del retardo inicial es positivo:

```
if (entry.getStartDelay() < 0) {
        adviceDialog. setTitle(new Model<String>("Info"));
        IModel<String> modelHeading = new Model<String>("Incorrect " +
        "Parameters");
        IModel<String> modelMessages = new Model<String>("Start Delay must " +
        "be a positive number");
        adviceDialog.showInfo(target, modelHeading, modelMessages);
        log.warning("Incorrect value: Start Delay must be a positive number");
        return;
}
```

Código 94. Extracto `PrefetchJobConfigurationPanel` XXIX

- Se comprueba el valor introducido en el campo de número de repeticiones. Este solo puede contener valores 0 para repetir indefinidamente, 1 para no repetir y un valor mayor a la unidad para un número concreto de repeticiones. En primer lugar se ha comprobado que se ha introducido alguno de estos tres valores. Si se ha introducido un valor de los correctos, si este valor es mayor que la unidad o igual a cero, el usuario debe dar valor obligatorio al campo intervalo de repetición y este deberá tener un valor mayor que cero. En caso de no requerir repetición ese campo se obvia.

```
if (entry.getRepeatCount() < 0) {
        adviceDialog.setTitle(new Model<String>("Info"));
        IModel<String> modelHeading = new Model<String>("Incorrect " +
        "Parameters");
        IModel<String> modelMessage = new Model<String>(
        "<html><p>Repeat count cannot be negative:<br/> &nbsp0 for repeat
        indefinitely<br/> &nbsp1 for no repetition<br/> &nbsp1 for a
```

```
            especific repetition</p></html>");

            adviceDialog.showInfo(target, modelHeading, modelMessage);

            log.warning("Incorrect value: Repeat count cannot be negative");
            return;
        } else if ((entry.getRepeatCount()== 0 || entry.getRepeatCount() > 1 ) &&
        entry.getRepeatInterval() <= 0) {
            adviceDialog.setTitle(new Model<String>("Info"));
            IModel<String> modelHeading = new Model<String>("Incorrect " +
            "Parameters");
            IModel<String> modelMessages = new Model<String>("Repeat interval " +
            "must be higher than zero");
            adviceDialog.showInfo(target, modelHeading, modelMessages);
            log.warning("Incorrect value: Repeat interval must be higher than " +
            "zero");
            return;
        }
```

Código 95. Extracto `PrefetchJobConfigurationPanel` XXX

El código *HTML* que se aprecia a la hora de mostrar el mensaje de error al usuario simplemente busca dar una forma de presentación más agradable al usuario.

- Se comprueba que el encuadre (*Bounding Box*) tenga definidas las cuatro coordenadas:

```
    if (entry.getBbox() == null) {
            adviceDialog.setTitle(new Model<String>("Info"));
            IModel<String> modelHeading = new Model<String>("Incorrect Bounding " +
            "Box");
            IModel<String> modelMessages = new Model<String>("Select a Bounding " +
            "Box");
            adviceDialog.showInfo(target, modelHeading, modelMessages);
            log.warning("Incorrect value: Incorrect Bounding Box");
            return;
    }
```

Código 96. Extracto `PrefetchJobConfigurationPanel` XXXI

- Se comprueba que el *buffer* introducido es positivo:

```
    if (entry.getBuffer() < 0) {
            adviceDialog.setTitle(new Model<String>("Info"));
            IModel<String> modelHeading = new Model<String>("Incorrect Buffer");
            IModel<String> modelMessages = new Model<String>("Buffer must be " +
            "positive");
            adviceDialog. showInfo(target, modelHeading, modelMessages);
            log.warning("Incorrect value: Incorrect Buffer");
            return;
    }
```

Código 97. Extracto `PrefetchJobConfigurationPanel` XXXII

- Se comprueba que no se haya lanzado un trabajo con el mismo nombre:

```
    List<PrefetchTask> ptaskList = Seeder.get()
            .getPrefetchTasks(tileLayer. getName());

    for (int i = 0; i < ptaskList.size(); i++) {
            PrefetchTask ptask = ptaskList.get(i);
            if (ptask.getJobName().equals(entry.getJobName())) {
                    adviceDialog.setTitle(new Model<String>("Info"));
                    IModel<String> modelHeading = new Model<String>("Another Job " +
                    "Running");
```

```
                IModel<String> modelMessages = new Model<String>(
                "Already exists a Prefetch Task with the same job name. Wait " +
                "for it to finish or cancel it to continue");
                adviceDialog.showInfo(target, modelHeading, modelMessages);
                log.warning("Incorrect value: existing job name");
                return;
            }
        }
```

Código 98. Extracto `PrefetchJobConfigurationPanel` XXXIII

El método del objeto `Seeder` `getPrefetchTask()` permite obtener las tareas de *prefetch* en ejecución de una determinada capa cuyo nombre recibe como parámetro. Tiene la siguiente estructura:

```
    public List<PrefetchTask> getPrefetchTasks(String layerName) {
        List<GWCTask> items = new ArrayList<GWCTask>();

        Iterator<GWCTask> iter = tileBreeder.getRunningAndPendingTasks();

        while (iter.hasNext()) {
            GWCTask task = iter.next();
            if (layerName.equals(task.getLayerName())) {
                items.add(task);
            }
        }
        List<PrefetchTask> pList = new ArrayList<PrefetchTask>();
        for (int i = 0; i < items.size(); i++) {
            if (items.get(i) instanceof PrefetchTask) {
                PrefetchTask task = (PrefetchTask) items.get(i);
                pList.add(task);
            }
        }
        return pList;
    }
```

Código 99. Método `getPrefetchTasks()` del objeto `Seeder`

Las tareas de *prefetch* se registran en un registro de *GWC*. Este código es similar al que implementa *GeoServer* para poder acceder a las tareas de `GWCTask`. Las tareas de tipo `PrefetchTask` son un subtipo de ellas. El bucle `while` permite obtener todas las tareas de tipo `GWCTask` relacionadas con esta capa. El bucle `for` permite quedarnos solo con las que sean `PrefecthTask` y añadirlas a una lista. El método devuelve esta lista.

De vuelta al panel, este coge esta lista y busca si en ella hay alguna tarea cuyo nombre (ya se verá que es uno de los parámetros que caracterizarán a las tareas `PrefetchTask`) es igual al nombre del trabajo que queremos lanzar. En caso afirmativo se lanza un mensaje de error y se finaliza la ejecución de código.

Llegados a este punto se han superado todas las comprobaciones. El siguiente paso será transformar los parámetros retardo inicial e intervalo entre repeticiones de las unidades de tiempo en que el usuario las haya expresado a milisegundos, la unidad de tiempo que utiliza el servicio *Quartz*:

```
    long startDelayMS = timeInMiliSeconds(entry.getStartDelayUnit(),
        entry.getStartDelay());
    long repeatIntervalMS = timeInMiliSeconds(entry.getRepeatIntervalUnit(),
        entry.getRepeatInterval());
```

Código 100. Extracto `PrefetchJobConfigurationPanel` XXXIV

El método que permite esta transformación es el siguiente:

```
public long timeInMiliSeconds(String unit, int value) {
	long valueLong = (long) value;
	long timeMS = 0;
	if (unit.equals("years")) {
		timeMS = (long) (valueLong * 365 * 24 * 60 * 60 * 1000);
	} else if (unit.equals("months")) {
		timeMS = (long) valueLong * 30 * 24 * 60 * 60 * 1000;
	} else if (unit.equals("weeks")) {
		timeMS = (long) (valueLong * 7 * 24 * 60 * 60 * 1000);
	} else if (unit.equals("days")) {
		timeMS = (long) (valueLong * 24 * 60 * 60 * 1000);
	} else if (unit.equals("hours")) {
		timeMS = (long) (valueLong * 60 * 60 * 1000);
	} else if (unit.equals("minutes")) {
		timeMS = (long) (valueLong * 60 * 1000);
	} else if (unit.equals("seconds")) {
		timeMS = (long) (valueLong * 1000);
	} else
		timeMS = (long) valueLong;

	return timeMS;
}
```

Código 101. Extracto `PrefetchJobConfigurationPanel` XXXV

Recibe la cantidad que debe transformar y la unidad en que está expresada. La sentencia `if-else` múltiple permite diferenciar según unidad origen y transformarlo a milisegundos con las expresiones típicas. Se considera que los meses tienen todos 30 días.

A continuación se procede a programar el conjunto de tareas necesarias para lanzar el trabajo requerido por el usuario.

En primer lugar se define un objeto de la clase *trigger* de *Quartz* que se va a encargar de lanzar los trabajos. Un *trigger* simplemente es un lanzador de tareas.

```
SimpleTrigger trigger = new SimpleTrigger();
trigger.setName(entry.getJobName());
trigger.setStartTime(new Date(System.currentTimeMillis() +
	startDelayMS));
if (entry.getRepeatCount() > 1) {
	trigger.setRepeatCount(entry.getRepeatCount() - 1);
	trigger.setRepeatInterval(repeatIntervalMS);
}
if (entry.getRepeatCount() == 0) {
	trigger.setRepeatCount(-1);
	trigger.setRepeatInterval(repeatIntervalMS);
}
```

Código 102. Extracto `PrefetchJobConfigurationPanel` XXXVI

Los parámetros básicos que caracterizan un *trigger* son el nombre del trabajo lanzado, en este caso se le da el nombre que ha escogido el usuario, y el instante de tiempo en que debe lanzarse la tarea por primera vez, en este caso a partir del momento en que el usuario pulsa el enlace pasado el tiempo especificado en el parámetro retardo inicial. Este último parámetro debe especificarse como la cantidad de milisegundos transcurridos desde 1 de enero de 1970 en que debe lanzarse la tarea. Para ello se usa la función `System.currentTimeMillis()` que permite obtener la cantidad de milisegundos transcurridos entre esa fecha anterior y el

instante en que se lanza esta función. Se le suma a mayores los milisegundos de retardo que quiere el usuario para establecer un momento "futuro" concreto.

Existen otra serie de parámetros pero son opcionales en función de cómo deba trabajar el *trigger*. Estos son el número de veces que debe repetir la tarea. En nuestro caso, esta funcionalidad deberá implementarse siempre que el campo repeticiones que el usuario puede rellenar contenga una cantidad superior a 1. En caso de establecer este parámetro también se debe especificar el intervalo entre repeticiones, obtenido a partir del parámetro que el usuario puede configurar. Este intervalo debe especificarse en milisegundos.

Por otro lado, el usuario puede haber configurado una repetición infinita de los trabajos. En este caso el atributo `repeatCount` del objeto *trigger* deberá tener un valor -1. También se debe especificar la cantidad de milisegundos entre repeticiones.

El siguiente paso es configurar los detalles del trabajo que debe lanzar con esas configuraciones el *trigger*. Para ello se usan los objetos de la clase `JobDetail` de *Quartz*. Se debe establecer el nombre del trabajo y la clase a la que pertenece el trabajo, es decir, la clase *Java* en la que se define el objeto que se corresponde con el trabajo. O visto de otra forma, el objeto *Java* en el que se definen las distintas funciones que deben ejecutarse, `FeatureBasedPrefetchingJob` (ver Sección 4.4.2) en este caso.

```
JobDetail job = new JobDetail();
try {
	job.setName(entry.getJobName());
	job.setJobClass(FeatureBasedPrefetchingJob.class);
	JobDataMap jdm = Seeder.get().newJDM(layerId, entry, date, format,
		reseed);
	job.setJobDataMap(jdm);
} catch (Exception e) {
	adviceDialog.setTitle(new Model<String>("Info"));
	IModel<String> modelHeading = new Model<String>("Error creating job");
	IModel<String> modelMessages = new Model<String>(
	"Unable to create job '" + entry.getJobName() + "'" + e.getMessage());
	adviceDialog.showInfo(target, modelHeading, modelMessages);
	log.info("Unable to create job '" + entry.getJobName() + "'");
	return;
}
```

Código 103. Extracto `PrefetchJobConfigurationPanel` XXXVII

Mediante un objeto de la clase `JobDataMap` se definen los atributos que deberán pasarse al trabajo. Es decir, para poder ejecutarse, ese objeto requiere de una serie de parámetros, por ejemplo, los configurados por el usuario. Este objeto consiste en un mapa en que se almacenan parejas nombre-valor con los distintos datos necesarios para después recuperarlos y no tener que pasárselos como parámetros en la llamada a la función, porque no existe ninguna llamada a la función explícita al ser una tarea programada. En el método `newJDM()` del objeto `Seeder` se define la creación de este mapa de detalles:

```
1  public JobDataMap newJDM(String layerID, PrefetchJobConfigurationEntry gce,
2          Date date, String format, boolean reseed) throws IOException {
3
4          LayerInfo layerInfo = GWC.get().getLayerInfoById(layerID);
5          LayerGroupInfo layerGroupInfo = GWC.get().getLayerGroupById(layerID);
6
7          GeoServerTileLayer tileLayer;
```

```
8           GeoServerSeederLayerInfo seederLayerInfo;
9           if (layerInfo != null) {
10                  tileLayer = GWC.get().getTileLayer(layerInfo);
11                  seederLayerInfo = getSeederLayerInfo(layerInfo);
12          } else {
13                  tileLayer = GWC.get().getTileLayer(layerGroupInfo);
14                  seederLayerInfo = getSeederLayerGroupInfo(layerGroupInfo);
15          }
16
17          String storeId = seederLayerInfo.getStore();
18          StoreInfo storeInfo = GWC.get().getCatalog().getStore(storeId,
19                  StoreInfo.class);
20          DataStoreInfo dataInfo = (DataStoreInfo) storeInfo;
21          DataAccess<? extends FeatureType, ? extends Feature> ds =
22                  dataInfo.getDataStore(null);
23          DataStore store = (DataStore) ds;
24          JDBCDataStore jdbc = null;
25          if (store instanceof JDBCDataStore) {
26                  jdbc = (JDBCDataStore) store;
27          } else {
28                  throw new IOException("Data Store is not JDBC.");
29          }
30
31          String layerName = tileLayer.getName();
32          JobDataMap jdm = new JobDataMap();
33
34          jdm.put("jobName", gce.getJobName());
35          jdm.put("layerId", layerName);
36          jdm.put("gridSetId", gce.getGridsetName());
37          jdm.put("format", format);
38          jdm.put("tileLayerDispatcher", tld);
39          jdm.put("storageBroker", storageBroker);
40          jdm.put("featureCatalog", constructFeatureCatalog(layerID));
41          jdm.put("dataStore", jdbc);
42          jdm.put("bbox", gce.getBbox());
43          jdm.put("res", gce.getResolution());
44          jdm.put("buffer", gce.getBuffer());
45          jdm.put("reseed", reseed);
46          jdm.put("tileBreeder", tileBreeder);
47          jdm.put("registry", this);
48          jdm.put("store", storeId);
49          jdm.put("initTime", date);
50
51          return jdm;
52
53  }
```

Código 104. Método `newJDM()` del objeto `Seeder`

Este método recibe como parámetros el identificador de la capa/grupo de capas para la que se lanzó el trabajo, el conjunto de configuraciones del panel que rellenó el usuario, la fecha en que se lanzó la tarea, el formato de imagen escogido para hacer el *prefetch* y si la tarea debe lanzarse como *Seed* o *ReSeed*.

A partir del identificador de la capa/grupo de capas se obtiene el objeto `TileLayer` de *GWC* asociado y a partir de él, el objeto `SeederLayerInfo` (líneas 4-8). Mediante la línea 17 se obtiene a partir de este último objeto el `storeId` que el usuario guardó por última vez para la capa. Es requisito al cambiarse la base de datos para las estadísticas que el usuario guarde los cambios, ya que en caso contrario no se genera conexión con esa base de datos y llegados a este punto se lanza el resto del trabajo con la última conexión de base de datos configurada. A partir de este identificador se obtiene el objeto `DataStore` y si es de tipo *JDBC* (se supone que

se ha comprobado reiterativamente que es de tipo *JDBC* pero partimos del supuesto que el usuario puede modificar manualmente los ficheros de configuración y hacer caer la aplicación) obtener el *Feature Source* corresponde a la base de datos en que se guardan las estadísticas de este capa/grupo de capas (líneas 18-29).

Las líneas 34-49 permiten, a partir de los valores recién definidos o los pasados como parámetro, añadir al mapa de detalles de trabajo parejas nombre-valor mediante el método `put()` clásico de todo mapa.

De vuelta al panel de configuración de trabajos, solo queda, una vez configurado el *trigger* y el trabajo a lanzar, el programador que se encargará de hacer que el *trigger* lance, según el calendario con que se ha configurado, el trabajo definido en el objeto `JobDetail`.

```
Scheduler scheduler;
try {
	scheduler = new StdSchedulerFactory().getScheduler();
	scheduler.start();
	scheduler.scheduleJob(job, trigger);
} catch (SchedulerException e) {
	log.info("Already exist a Job: '" + entry.getJobName() + "'");
	adviceDialog.setTitle(new Model<String>("Info"));
	IModel<String> modelHeading = new Model<String>(
		"This job already exist");
	IModel<String> modelMessages = new Model<String>(
		"Already exist a Prefetch Job with these name");
	adviceDialog.showInfo(target, modelHeading, modelMessages);
	return;
}

log.info("Done");
log.info("Starting prefetch job '" + entry.getJobName() + "'");
```

Código 105. Extracto `PrefetchJobConfigurationPanel` XXVIII

Llegados aquí se completa la definición *Java* del panel de configuración de trabajos de *prefetch*.

Para finalizar se muestra el fichero `PrefetchJobConfigurationPanel.html` en que se define la verdadera estructura y orden que tendrá este panel:

```
<html>
<body>
<wicket:panel>
<ul>
	<li><label> <wicket:message
	key="prefetchingJobConfiguration">Prefetching Job
	Configuration</wicket:message>
	</label></li>

	<li><a class="add-link" wicket:id="addJob"><wicket:message
	key="addJob">Add Job</wicket:message>
	</a></li>

	<li>
		<div wicket:id="entry"></div>
	</li>

	<div wicket:id="adviceDialog"></div>
	<div wicket:id="deleteDialog"></div>
```

```
</ul>
<div wicket:id="popup"></div>
</wicket:panel>

<wicket:fragment wicket:id="jobName">
	<input type="text" style="width: 100%" wicket:id="jobName" />
</wicket:fragment>

<wicket:fragment wicket:id="buffer">
	<input type="text" style="width: 100%" wicket:id="buffer" />
</wicket:fragment>

<wicket:fragment wicket:id="resolution">
	<select class="select" id="resolution" wicket:id="resolution"></select>
</wicket:fragment>

<wicket:fragment wicket:id="bbox">
	<label wicket:id="labelminx">minX</label>
	<input type="text" style="width: 100%" wicket:id="minx" />
	<br />
	<label wicket:id="labelmaxx">maxX</label>
	<input type="text" style="width: 100%" wicket:id="maxx" />
	<br />
	<label wicket:id="labelminy">minY</label>
	<input type="text" style="width: 100%" wicket:id="miny" />
	<br />
	<label wicket:id="labelmaxy">maxY</label>
	<input type="text" style="width: 100%" wicket:id="maxy" />
</wicket:fragment>

<wicket:fragment wicket:id="delay">
	<input type="text" style="width: 100%" wicket:id="delay" />
	<br />
	<select class="select" id="startDelayUnit" style="width: 100%"
		wicket:id="startDelayUnit"></select>
</wicket:fragment>

<wicket:fragment wicket:id="repeat">
	<wicket:message key="repeatAdvice">0 for Repeat
	Indefinitely</wicket:message>
	<input type="text" style="width: 100%" wicket:id="repeat" />
</wicket:fragment>

<wicket:fragment wicket:id="interval">
	<input type="text" style="width: 100%" wicket:id="interval" />
	<br />
	<select class="select" id="intervalUnit" style="width: 100%"
		wicket:id="intervalUnit"></select>
</wicket:fragment>

<wicket:fragment wicket:id="format">
	<select class="select" id="format" style="width: 100%"
		wicket:id="format"></select>
</wicket:fragment>

<wicket:fragment wicket:id="action">
	<div wicket:id="delete" align="center"/>
	<div style="white-space: nowrap;" align="center">
		<a target="_blank" wicket:id="seed"></a> <br /> <a
		target="_blank"
		wicket:id="reseed" align="center"></a> <br />
	</div>
</wicket:fragment>
</body>
</html>
```

Código 106. Fichero PrefetchJobConfigurationPanel.html

4.3.3.3. Panel de selección de la base de datos

Este panel permite al usuario seleccionar la base de datos en la que guardar y extraer las estadísticas de acceso a esta capa/grupo de capas. Consiste en una tabla en que se muestra la selección del usuario. Para elegir la base de datos el usuario dispone de un enlace que al ser pulsado lanza una ventana emergente con todas las bases de datos que el usuario puede elegir, en concreto solo muestra las bases de datos para las que se tiene implementación: *H2* y *PostGIS*, aunque como se ha visto en repetidas ocasiones, se comprueba reiterativamente que las bases de datos seleccionadas se ajusta a formato *JDBC* y dentro de este tipo, *H2* o *PostGIS* para evitar que un usuario trate de engañar a la aplicación y establecer conexión con una base de datos especificada no usando la interfaz *web* que se está presentando.

En primer lugar presentamos la estructura de las entradas que conforman esta tabla. Se definen en el objeto `StoreEntry`:

```
public class StoreEntry implements Serializable {

	String storeInfoId;

	public StoreEntry(String storeInfoId) {
		this.storeInfoId = storeInfoId;
	}

	public String getStoreInfoId() {
		return storeInfoId;
	}

	public void setStoreInfoId(String storeInfoId) {
		this.storeInfoId = storeInfoId;
	}

}
```

Código 107. Objeto `StoreEntry`

Esta tabla solo contendrá una entrada ya que no se permite al usuario seleccionar más de una base de datos para cada capa/grupo de capas. Esta entrada muestra un conjunto de información relativa a esta base de datos seleccionada, pero sobre ella solo se define el identificador del *store* creado en *GeoServer* y *GWC*, porque el resto de información se obtiene a parte de este identificador y las herramientas de *GeoServer/GWC*.

A continuación se procede explicar la estructura de definición del panel `StoreStatsPanel`.

```
public class StoreStatsPanel extends Panel {…}
```

Código 108. Extracto `StoreStatsPanel` I

En primer lugar se definen los elementos que formarán parte del panel:

```
private static Logger log = Logging.getLogger(StoreStatsPanel.class);

ModalWindow popupWindow;
StoreStatsEntryProvider entryStatsProvider;

GeoServerTablePanel<StoreEntry> storeTable;
List<StoreEntry> store;
String layerId;
```

Código 109. Extracto `StoreStatsPanel` II

Cuyo significado es:

- `popupWindow`: ventana emergente en que se mostrará el panel donde el usuario podrá elegir la base de datos.
- `entryStatsProvider`: proveedor de entradas de la tabla.
- `storeTable`: la propia tabla.
- `store`: lista de entradas de la tabla. En este caso la única entrada que puede existir.
- `layerId`: identificador de la capa/grupo de capas.

A continuación se define el constructor del panel en el que se declarará cómo se debe completar cada una de las columnas de la tabla para cada una de las entradas y el enlace que permite lanzar la ventana emergente donde el usuario seleccionará la base de datos (la entrada a completar).

```
public StoreStatsPanel(String id, String storeId, String layerId) {…}
```

Código 110. Extracto `StoreStatsPanel` III

En primer lugar se inicializan los datos del panel con los argumentos del constructor. Este recibe el identificador de la base de datos (parámetro nulo en caso que nunca se haya guardado una) y el identificador de la capa/grupo de capas:

```
1  Seeder.get().addStorePanel(layerId, this);
2  this.layerId = layerId;
3
4  store = new ArrayList<StoreEntry>();
5
6  if (storeId != null) {
7
8         StoreInfo storeInfo = GWC.get().getCatalog().getStore(storeId,
9                StoreInfo.class);
10        if (storeInfo != null) {
11               store.add(new StoreEntry(storeId));
12        } else {
13               Seeder.get().removeItemIndexMap(layerId);
14        }
15
16 }
```

Código 111. Extracto `StoreStatsPanel` IV

La primera línea del código 111 registra este panel (asociado a esta capa) en el objeto `Seeder` para poder acceder a él rápidamente desde cualquier otro punto de la aplicación:

```
public void addStorePanel(String layerId, StoreStatsPanel panel) {
       this.storeList.put(layerId, panel);
}
```

Código 112. Método `addStorePanel()` del objeto Seeder

En la línea 4 del código 111 se puede ver la inicialización de la lista de entradas de la tabla. Para seguir con la estructura del resto de tablas-paneles de la interfaz, se define el conjunto de datos que maneja el panel como una lista de entradas, aunque en este caso esta lista solo contendrá un elemento.

Si el identificador de *store* recibido como parámetro no es nulo (porque la tabla ya estaba inicializada) se accede al objeto `StoreInfo` asociado a este identificador utilizando para ello las herramientas de *GWC*. A continuación se comprueba que existe un objeto `StoreInfo` con ese identificador. En caso afirmativo, se añade una nueva entrada a la tabla para él. En caso contrario, se procede a borrar toda la información relativa a ese identificador porque significa que, posteriormente a que el usuario escogiera esta base de datos para sus propósitos de almacenamiento de estadísticas, la base de datos ha sido borrada de la aplicación. Este es el objetivo del método `removeItemIndexMap()` del objeto `Seeder` que se encarga de borrar del mapa en el que se almacenan los accesos a las bases de datos para cada capa, el relativo a esta capa (a través de su identificador de capa).

```
public void removeItemIndexMap(String layerId) {
	this.indexMap.remove(layerId);
}
```

Código 113. Método `removeItemIdexMap()` del objeto `Seeder`

Ya se ha hablado alguna vez de este mapa `indexMap`. Es uno de los atributos del objeto `Seeder` y permite poder acceder al índice (`index`) que se ha configurado para el acceso a la base de datos correspondiente para cada una de las capas. Es decir cada capa (siempre que se haya configurado) tendrá asociado un índice que se puede ver como un acceso o conexión a la base de datos en que se recogen sus estadísticas. Este mapa permite mantener una relación de estas asociaciones para, en cualquier punto de la aplicación, acceder a la información de acceso a la base de datos.

El siguiente paso es añadir la tabla que conforma el panel e indicarle como debe completar cada una de sus columnas para cada una de las entradas haciendo uso del proveedor de entradas `StoreStatsEntryProvider`:

```
entryStatsProvider = new StoreStatsEntryProvider(store);
add(storeTable = new GeoServerTablePanel<StoreEntry>("entry",
entryStatsProvider) {
	@Override
	protected Component getComponentForProperty(String id, IModel
	itemModel, Property<StoreEntry> property) {
		Component result = null;

		if (property == StoreStatsEntryProvider.DATATYPE) {
			result = dataTypeLink(id, itemModel);
		} else if (property == StoreStatsEntryProvider.WORKSPACE) {
			result = workspaceLink(id, itemModel);
		} else if (property == StoreStatsEntryProvider.STORENAME) {
			result = storeNameLink(id, itemModel);
		} else if (property == StoreStatsEntryProvider.TYPE) {
			result = typeLink(id, itemModel);
		} else if (property == StoreStatsEntryProvider.ENABLED) {
			result = enabledLink(id, itemModel);
		} else {
			result = null;
		}

		return result;
	}
}.setFilterable(false));
storeTable.setItemReuseStrategy(new DefaultItemReuseStrategy());
storeTable.setOutputMarkupId(true);
```

Código 114. Extracto `StoreStatsPanel` V

Mediante la estructura `if-else` se indica cómo se debe ir completando la tabla, es decir, que información se debe mostrar en cada columna. Más adelante se explicarán los distintos métodos que ahí se presentan.

El proveedor de entradas `StoreStatsEntryProvider` se implementa como una clase estática en el mismo objeto aunque fuera del constructor:

```
static class StoreStatsEntryProvider extends GeoServerDataProvider<StoreEntry>
{

	public static Property<StoreEntry> DATATYPE = new
		PropertyPlaceholder<StoreEntry>("DataType");

	public static Property<StoreEntry> WORKSPACE = new
		PropertyPlaceholder<StoreEntry>("Workspace");

	public static Property<StoreEntry> STORENAME = new
		PropertyPlaceholder<StoreEntry>("StoreName");

	public static Property<StoreEntry> TYPE = new
		PropertyPlaceholder<StoreEntry>("Type");

	public static Property<StoreEntry> ENABLED = new
		PropertyPlaceholder<StoreEntry>("Enabled");

	static List PROPERTIES = Arrays.asList(DATATYPE, WORKSPACE, STORENAME,
		TYPE, ENABLED);

	List<StoreEntry> items;

	public StoreStatsEntryProvider(List<StoreEntry> items) {
		this.items = items;
	}

	@Override
	protected List<StoreEntry> getItems() {
		return items;
	}

	@Override
	protected List<Property<StoreEntry>> getProperties() {
		return PROPERTIES;
	}

}
```

Código 115. Extracto `StoreStatsPanel` VI

Como casos anteriores, esta clase nos permite definir las columnas de que va a constar la tabla y el orden en que deben aparecer, así como su título.

Por último se define en enlace que nos permite lanzar la ventana emergente donde seleccionar la base de datos:

```
1  add(popupWindow = new ModalWindow("popup"));
2  add(new AjaxLink("selectStore") {
3  	@Override
4  	public void onClick(AjaxRequestTarget target) {
5  		popupWindow.setInitialHeight(375);
6  		popupWindow.setInitialWidth(525);
7  		popupWindow.setTitle(new ParamResourceModel("chooseStore",
8  			this));
```

```
9                   popupWindow.setContent(new
10                  StoreChoicesPanel(popupWindow.getContentId()) {
11                          @Override
12                          protected void handleSelection(StoreInfo storeInfo,
13                          AjaxRequestTarget target) {
14                                  popupWindow.close(target);
15                                  store.clear();
16                                  entryStatsProvider.getItems().add(new
17                                  StoreEntry(storeInfo.getId()));
18
19                                  target.addComponent(storeTable);
20                          }
21                  });
22                  popupWindow.show(target);
23          }
24  });
25  popupWindow.show(target);
26
```

Código 116. Extracto `StoreStatsPanel` VII

Como en casos anteriores, se establece el tamaño inicial y el título de la ventana. En la líneas 9-21 se define que el contenido de esta ventana viene establecido por el objeto `StoreChoicesPanel` explicado a continuación. Sobre este objeto se sobrescribe el método `handleSelection()` para definir qué acciones se deben realizar una vez que el usuario haya pulsado el correspondiente botón de enlace (de selección en este caso) sobre la página (panel en este caso) que se muestre en esa ventana emergente. Se ha indicado repetidas veces que este panel/tabla solo puede tener una entrada, por lo que cuando el usuario seleccione una base de datos del panel emergente, habiendo una ya seleccionada en el panel principal, se deberá borrar la selección anterior e incluir la nueva. Este es el propósito de las líneas 14-19: en primer lugar se borra la lista de entradas y luego se añade una nueva con los nuevos datos escogidos por el usuario y se actualiza el panel.

El contenido de este panel emergente es análogo al resto de paneles. La diferencia está en el tipo de entradas y por tanto datos que maneja. Las entradas son objetos `StoreInfo` (ya definido por *GWC*):

- Primero se definen los elementos que formarán parte del panel:

```
private static Logger log = Logging.getLogger(StoreStatsPanel.class);

ModalWindow popupWindow;
StoreStatsEntryProvider entryProvider;
GeoServerTablePanel<StoreInfo> table;
List<StoreInfo> items;
```

Código 117. Extracto `StoreChoicesPanel` I

Con definiciones análogas al resto de paneles.

- Se define el constructor del panel con idéntica estructura a cualquier otro constructor de los explicados en este trabajo.

```
1   public StoreChoicesPanel(String id) {
2           super(id);
3           items = new ArrayList<StoreInfo>();
```

```
4   List<StoreInfo> storeList = new ArrayList<StoreInfo>();
5
6           storeList = GWC.get().getCatalog().getStores(StoreInfo.class);
7           for (int i = 0; i < storeList.size(); i++) {
8                   if (storeList.get(i).getType() != null && storeList.get(i)
9                           instanceof DataStoreInfo) {
10                          if (storeList.get(i).getType().equals("PostGIS") ||
11                                  storeList.get(i).getType().equals("H2")) {
12                                          items.add(storeList.get(i));
13                          }
14                  }
15          }
16
17          add(popupWindow = new ModalWindow("popup"));
18
19          entryProvider = new StoreStatsEntryProvider(items);
20          add(table = new GeoServerTablePanel<StoreInfo>("entry",
21                  entryProvider) {
22                  @Override
23                  protected Component getComponentForProperty(String id, IModel
24                          itemModel, Property<StoreInfo> property) {
25                  Component result = null;
26                  if (property == StoreStatsEntryProvider.DATATYPE) {
27                          result = dataTypeLink(id, itemModel);
28                  } else if (property == StoreStatsEntryProvider.WORKSPACE)
29                     {
30                          result = workspaceLink(id, itemModel);
31                  } else if (property == StoreStatsEntryProvider.STORENAME)
32                     {
33                          result = storeNameLink(id, itemModel);
34                  } else if (property == StoreStatsEntryProvider.TYPE) {
35                          result = typeLink(id, itemModel);
36                  } else if (property == StoreStatsEntryProvider.ENABLED) {
37                          result = enabledLink(id, itemModel);
38                  } else if (property == StoreStatsEntryProvider.ACTION) {
39                          result = resourceChooserLink(id, itemModel, new
40                                  ParamResourceModel("select", this));
41                  } else {
42                          result = null;
43                  }
44
45                  return result;
46                  }
47          }.setFilterable(false));
48          table.setItemReuseStrategy(new DefaultItemReuseStrategy());
49          table.setOutputMarkupId(true);
50  }
51
```

Código 118. Extracto `StoreChoicesPanel` II

Lo único reseñable a explicar es cómo completar las distintas entradas a esta tabla. Estará compuesta por toda la información pertinente relativa a las bases de datos que se encuentren configuradas en la aplicación y que sean de tipo bien *H2*, bien *PostGIS*. Mediante las herramientas de *GWC* se obtienen todos los *stores* configurados y de todos ellos solo se añaden a la lista de entradas los que cumplan el requisito anterior (líneas 9-15).

El proveedor de entradas tendrá la forma:

```
static class StoreStatsEntryProvider extends GeoServerDataProvider<StoreInfo>
{
        public static Property<StoreInfo> DATATYPE = new
        PropertyPlaceholder<StoreInfo>("DataType");
```

```
		public static Property<StoreInfo> WORKSPACE = new
		PropertyPlaceholder<StoreInfo>("Workspace");

		public static Property<StoreInfo> STORENAME = new
		PropertyPlaceholder<StoreInfo>("StoreName");

		public static Property<StoreInfo> TYPE = new
		PropertyPlaceholder<StoreInfo>("Type");

		public static Property<StoreInfo> ENABLED = new
		PropertyPlaceholder<StoreInfo>("Enabled");

		public static Property<StoreInfo> ACTION = new
		PropertyPlaceholder<StoreInfo>("Action");

		static List PROPERTIES = Arrays.asList(DATATYPE, WORKSPACE, STORENAME,
		TYPE, ENABLED, ACTION);

		List<StoreInfo> items;

		public StoreStatsEntryProvider(List<StoreInfo> items) {
			this.items = items;
		}

		@Override
		protected List<StoreInfo> getItems() {
			return items;
		}

		@Override
		protected List<Property<StoreInfo>> getProperties() {
			return PROPERTIES;
		}

	}

	protected void handleSelection(StoreInfo storeInfo, AjaxRequestTarget target)
	{}
```

Código 119. Extracto `StoreChoicesPanel` III

Para terminar con el panel emergente se muestra el fichero `StoreChoicesPanel.html` en que se define la estructura que presentará en pantalla:

```
	<html>
	<body>
	<wicket:panel>
		<ul>
			<li>
				<div wicket:id="entry"></div>
			</li>

		</ul>

		<div wicket:id="popup"></div>

	</wicket:panel>

	<wicket:fragment wicket:id="iconFragment">
		<img wicket:id="storeIcon" />
	</wicket:fragment>
	</body></html>
```

Código 120. Fichero `StoreChoicesPanel.html`

Los métodos que se usan para completar la información de cada entrada del panel `StoreChoicesPanel` hacen uso de los métodos de *GeoServer/GWC* para el objeto `StoreInfo` y presentan una estructura totalmente análoga (aunque con alguna modificación para adaptarlo a nuestros propósitos) al panel de la sección Almacenes de datos de *GeoServer* denominado `StorePanel`, que permite seleccionar un almacén de datos para acceder a su configuración. Se presentan a continuación estos métodos sin entrar en más detalles:

```
Component dataTypeLink(String id, IModel itemModel) {
	final CatalogIconFactory icons = CatalogIconFactory.get();
	StoreInfo entry = (StoreInfo) itemModel.getObject();
	ResourceReference storeIcon = icons.getStoreIcon(entry);

	Fragment f = new Fragment(id, "iconFragment", StoreChoicesPanel.this);
	f.add(new Image("storeIcon", storeIcon));

	return f;
}

Component workspaceLink(String id, IModel itemModel) {
	StoreInfo entry = (StoreInfo) itemModel.getObject();

	return new Label(id, entry.getWorkspace().getName());
}

Component storeNameLink(String id, IModel itemModel) {
	final CatalogIconFactory icons = CatalogIconFactory.get();
	StoreInfo entry = (StoreInfo) itemModel.getObject();

	return new Label(id, entry.getName());
}

Component typeLink(String id, IModel itemModel) {
	final CatalogIconFactory icons = CatalogIconFactory.get();
	StoreInfo entry = (StoreInfo) itemModel.getObject();

	String type = entry.getType();

	try {
		ResourcePool resourcePool =
			GWC.get().getCatalog().getResourcePool();
		if (entry instanceof DataStoreInfo) {
			DataStoreInfo dsInfo = (DataStoreInfo) entry;
			DataAccessFactory factory =
			resourcePool.getDataStoreFactory(dsInfo);
			if (factory != null) {
				type = factory.getDisplayName();
			}
		} else if (entry instanceof CoverageStoreInfo) {
			Format format =
			resourcePool.getGridCoverageFormat((CoverageStoreInfo)
			entry);
			if (format != null) {
				type = format.getName();
			}
		}
	} catch (Exception e) {
		// fine, we tried
	}

	return new Label(id, type);
}

Component enabledLink(String id, IModel itemModel) {
```

```
            final CatalogIconFactory icons = CatalogIconFactory.get();
            StoreInfo entry = (StoreInfo) itemModel.getObject();

            ResourceReference enabledIcon;
            if (entry.isEnabled()) {
                  enabledIcon = icons.getEnabledIcon();
            } else {
                  enabledIcon = icons.getDisabledIcon();
            }
            Fragment f = new Fragment(id, "iconFragment", StoreChoicesPanel.this);
            f.add(new Image("storeIcon", enabledIcon));
            return f;
      }
```

Código 121. Extracto `StoreChoicesPanel` IV

Pero de todos ellos el más interesante es el siguiente:

```
      SimpleAjaxLink resourceChooserLink(String id, IModel itemModel, IModel label)
      {
            return new SimpleAjaxLink(id, itemModel, label) {
                  @Override
                  protected void onClick(AjaxRequestTarget target) {
                        StoreInfo storeInfo = (StoreInfo)
                        getDefaultModelObject();
                        handleSelection(storeInfo, target);
                  }
            };
      }
```

Código 122. Extracto `StoreChoicesPanel` V

Que implementa el enlace que al ser pulsado por el usuario selecciona la entrada correspondiente para ser añadida a la tabla principal. Al pulsar el enlace (método `onClick()`) lanza el método `handleSelection()` sobre-implementado en el constructor del panel principal explicado previamente.

El último paso, relativo al panel `StoreStatsPanel`, es explicar la implementación de los distintos métodos que se encargar de mostrar la información de la entrada:

- **Tipo de dato**: se especifica mediante un icono que representa si los datos que se almacenan en esa base de datos son de tipo *vector* o de tipo *raster*, y está directamente relacionado con el tipo de base de datos. Debido a que la implementación actual de este trabajo solo se centra en los datos de tipo *vector*, el icono que aparecerá se corresponderá con el que *GeoServer* utiliza para representarlos:

```
      Component dataTypeLink(String id, IModel itemModel) {
            final CatalogIconFactory icons = CatalogIconFactory.get();
            StoreEntry entry = (StoreEntry) itemModel.getObject();
            StoreInfo storeInfo =
            GWC.get().getCatalog().getStore(entry.getStoreInfoId(),
            StoreInfo.class);
            ResourceReference storeIcon = icons.getStoreIcon(storeInfo);

            Fragment f = new Fragment(id, "iconFragment", StoreStatsPanel.this);
            f.add(new Image("storeIcon", storeIcon));

            return f;
      }
```

Código 123. Extracto `StoreStatsPanel` VIII

Para obtener este icono se hace uso de las herramientas de *GWC*. Una de ellas permite obtener el icono de tipo de datos correspondiente a partir del objeto `StoreInfo` asociado al almacén.

- **Espacio de trabajo (*workspace*)**: en esta celda se indica el nombre del espacio en que se ha definido el almacén:

```
Component workspaceLink(String id, IModel itemModel) {
          StoreEntry entry = (StoreEntry) itemModel.getObject();
          StoreInfo storeInfo =
          GWC.get().getCatalog().getStore(entry.getStoreInfoId(),
          StoreInfo.class);
          return new Label(id, storeInfo.getWorkspace().getName());
    }
```

Código 124. Extracto `StoreStatsPanel` IX

Consiste en una etiqueta con el nombre del *workspace* obtenido mediante las herramientas de *GWC* a partir del objeto `StoreInfo` asociado a la entrada.

- **Nombre del almacén**: nombre que el usuario da a la base de datos al configurarla:

```
Component storeNameLink(String id, IModel itemModel) {
          StoreEntry entry = (StoreEntry) itemModel.getObject();
          StoreInfo storeInfo =
          GWC.get().getCatalog().getStore(entry.getStoreInfoId(),
          StoreInfo.class);
          return new Label(id, storeInfo.getName());
    }
```

Código 125. Extracto `StoreStatsPanel` X

Se representa como una etiqueta cuyo valor se obtiene a partir de las herramientas de *GWC* y el `StoreInfo` asociado a la entrada.

- **Tipo**: en este apartado se especifica el tipo de base de datos. Teniendo en cuenta todas las limitaciones anteriores, solo hay dos valores posibles: *H2* y *PostGIS*.

```
Component typeLink(String id, IModel itemModel) {
          StoreEntry entry = (StoreEntry) itemModel.getObject();
          StoreInfo storeInfo =
          GWC.get().getCatalog().getStore(entry.getStoreInfoId(),
          StoreInfo.class);
          String type = storeInfo.getType();

          try {
                ResourcePool resourcePool =
                      GWC.get().getCatalog().getResourcePool();
                if (storeInfo instanceof DataStoreInfo) {
                      DataStoreInfo dsInfo = (DataStoreInfo) storeInfo;
                      DataAccessFactory factory =
                            resourcePool.getDataStoreFactory(dsInfo);
                      if (factory != null) {
                            type = factory.getDisplayName();
                      }
                } else if (storeInfo instanceof CoverageStoreInfo) {
                      Format format = resourcePool
                      .getGridCoverageFormat((CoverageStoreInfo) storeInfo);
                      if (format != null) {
                            type = format.getName();
```

```
                        }
                }
        } catch (Exception e) {
                // fine, we tried
        }

        return new Label(id, type);
    }
```

Código 126. Extracto `StoreStatsPanel` XI

Se obtiene haciendo uso de las herramientas de *GWC*.

- **Habilitado**: mediante un icono se representan si la aplicación ha podido comprobar que el almacén o base de datos se encuentra habilitado:

```
Component enabledLink(String id, IModel itemModel) {
        final CatalogIconFactory icons = CatalogIconFactory.get();
        StoreEntry entry = (StoreEntry) itemModel.getObject();
        StoreInfo storeInfo =
                GWC.get().getCatalog().getStore(entry.getStoreInfoId(),
                StoreInfo.class);
        ResourceReference enabledIcon;
        if (storeInfo.isEnabled()) {
                enabledIcon = icons.getEnabledIcon();
        } else {
                enabledIcon = icons.getDisabledIcon();
        }
        Fragment f = new Fragment(id, "iconFragment", StoreStatsPanel.this);
        f.add(new Image("storeIcon", enabledIcon));
        return f;
    }
```

Código 127. Extracto `StoreStatsPanel` XII

Se obtiene haciendo uso de las herramientas de *GWC*.

Se ha utilizado reiterativamente la palabra base de datos asociada al termino `StoreInfo`. La denominación en español de este término es *Almacén de Datos*. En nuestro caso, los almacenes de datos que el usuario va a poder utilizar dentro del contexto del módulo *Seeder* son bases de datos, bien embebidas (*H2*) o externas (*PostGIS*) que el usuario deberá haber configurado previamente y por ello se ha optado por utilizar este otro término, pero cualquiera de los mencionados, en el fondo, hacen referencia al mismo concepto.

El fichero *HTML* que define la estructura que este panel presentará en pantalla se denomina igual que el fichero *Java* en que se define el panel, `StoreStatsPanel.html`, y tiene el siguiente contenido:

```
    <html>
    <body>
    <wicket:panel>
            <fieldset>
                    <legend>
                    <span><wicket:message key="StoreStatsPanel.title">Store Stats
                    Configuration</wicket:message></span>
                    </legend>
                    <ul>
                            <li><label><wicket:message key="advice">Save
                            Configuration</wicket:message></label></li>
```

```
                        <li><a class="add-link" wicket:id="selectStore">
                        <wicket:message key="selectStore">Select
                        Store</wicket:message></a></li>

                        <li>
                                <div wicket:id="entry"></div>
                        </li>

                </ul>

                <div wicket:id="popup"></div>
        </fieldset>
</wicket:panel>

<wicket:fragment wicket:id="iconFragment">
        <img wicket:id="storeIcon" />
</wicket:fragment>

</body>
</html>
```

Código 128. Fichero `StoreStatsPanel.html`

4.3.3.4. Panel de visualización de tareas lanzadas

Este panel permite al usuario visualizar las tareas lanzadas relativas a esta capa/grupo de capas. La información mostrada para cada tarea es:

- El **identificador de tarea**.
- El ***GridSet*** para el que se lanzó la tarea.
- El **formato de imagen** que se seleccionó.
- El ***status*** de la tarea, es decir, en qué estado se encuentra. Se representa mediante un icono que al ser pulsado lanza una ventana emergente con información sobre el progreso o explicación de los errores. Posibles valores:
    - En ejecución.
    - En espera.
    - Completado.
    - Error.
- **Información** relativa a la tarea. Se representa mediante una imagen-enlace que al ser pulsada despliega una ventana emergente con toda la información relativa a la tarea lanzada:
    - Tipo de tarea: análisis, *prefetch*, *seed* o *truncate* (estas dos últimas son tareas definidas por *GWC*).
    - Nombre del trabajo.
    - Nombre de la base de datos.
    - Nombre de la capa/grupo de capas.
    - *GridSet*.
    - Formato de imagen.
    - *Bounding Box*.
    - *Buffer*.
    - Resolución.
    - Booleano que permite comprobar si la tarea se lanzó como *Seed* o *ReSeed*.

    - Una primera imagen en que se representan las teselas definidas para esta capa dentro de los límites establecidos por el usuario (encuadre, *buffer*, resolución)
    - Una segunda imagen en que se representan las teselas que concordando en *GridSet* y formato, han sido visitadas dentro del encuadre y resolución especificados por el usuario.
- **Tipo de tarea**: *Seed*, *ReSeed* o análisis.
- **Barra de estado** del progreso de la tarea y especificación en tanto por ciento.
- **Estimación del tiempo** transcurrido y tiempo restante para la finalización de la tarea.
- **Enlaces** que permiten eliminar una tarea ya completada de la lista o parar una ejecución.
- **Enlaces** que permiten eliminar todas las estadísticas de acceso relativas a la tarea correspondiente.

En la sección *"Manual de Usuario"* se analizará cómo debe interpretar el usuario cada uno de estos apartados y ejemplos típicos. En este apartado solo se analizará cómo se ha generado este panel y completado su información.

Este panel estará compuesto por una serie de entradas, cada uno de ellas se corresponde con una tarea. Las entradas a esta tabla se modelan como objetos `TaskEntry` que tienen la siguiente estructura:

```
public class TaskEntry implements Serializable {

	String taskId;
	String layerName;
	String gridset;
	String format;
	String status;
	String type;
	Long tilesTotal;
	Long tilesDone;
	Long timeRemaining;
	Long timeSpent;
	String statusMessage;
	String jobName;
	String infoMessage;
	boolean stats;
	String storeId;

	public TaskEntry(String jobName, String taskId, String
	layerName, String gridset, String format, String status, String
	type, Long tilesTotal, Long tilesDone, Long timeRemaining, Long
	timeSpent, String statusMessage, String infoMessage, boolean stats,
	String storeId) {
		this.taskId = taskId;
		this.layerName = layerName;
		this.gridset = gridset;
		this.format = format;
		this.status = status;
		this.type = type;
		this.tilesTotal = tilesTotal;
		this.tilesDone = tilesDone;
		this.timeRemaining = timeRemaining;
		this.timeSpent = timeSpent;
		this.statusMessage = statusMessage;
		this.jobName = jobName;
		this.infoMessage = infoMessage;
```

```
		this.stats=stats;
		this.storeId = storeId;
	}

	public Long getTimeRemaining() {
		return timeRemaining;
	}

	public void setTimeRemaining(Long timeRemaining) {
		this.timeRemaining = timeRemaining;
	}

	public Long getTimeSpent() {
		return timeSpent;
	}

	public void setTimeSpent(Long timeSpent) {
		this.timeSpent = timeSpent;
	}

	public String getTaskId() {
		return taskId;
	}

	public void setTaskId(String taskId) {
		this.taskId = taskId;
	}

	public String getLayerName() {
		return layerName;
	}

	public void setLayerName(String layerName) {
		this.layerName = layerName;
	}

	public String getStatus() {
		return status;
	}

	public void setStatus(String status) {
		this.status = status;
	}

	public String getType() {
		return type;
	}

	public void setType(String type) {
		this.type = type;
	}

	public Long getTilesTotal() {
		return tilesTotal;
	}

	public void setTilesTotal(Long tilesTotal) {
		this.tilesTotal = tilesTotal;
	}

	public Long getTilesDone() {
		return tilesDone;
	}

	public void setTilesDone(Long tilesDone) {
		this.tilesDone = tilesDone;
```

```
	}

	public String getGridset() {
		return gridset;
	}

	public void setGridset(String gridset) {
		this.gridset = gridset;
	}

	public String getFormat() {
		return format;
	}

	public void setFormat(String format) {
		this.format = format;
	}

	public String getStatusMessage() {
		return statusMessage;
	}

	public void setStatusMessage(String statusMessage) {
		this.statusMessage = statusMessage;
	}

	public String getJobName() {
		return jobName;
	}

	public void setJobName(String jobName) {
		this.jobName = jobName;
	}

	public String getInfoMessage() {
		return infoMessage;
	}

	public void setInfoMessage(String infoMessage) {
		this.infoMessage = infoMessage;
	}

	public boolean isStats() {
		return stats;
	}

	public void setStats(boolean stats) {
		this.stats = stats;
	}

	public String getStoreId() {
		return storeId;
	}

	public void setStoreId(String storeId) {
		this.storeId = storeId;
	}

}
```

Código 129. Objeto `TaskEntry`

La definición de este objeto consiste en una serie de atributos, con su correspondiente constructor y métodos `get()` y `set()`. ¿Qué representa cada uno de los atributos?

- `taskId`: identificador de la tarea.
- `layerName`: nombre de la capa sobre la que se lanzó la tarea.
- `gridset`: *GridSet* sobre el que se lanzó la tarea.
- `format`: formato de imagen seleccionado por el usuario.
- `status`: estado de la tarea.
- `type`: tipo de tarea.
- `tilesTotal`: número total de teselas que la tarea debe traer a caché.
- `tilesDone`: número total de teselas ya traídas a caché.
- `timeRemaining`: estimación del tiempo restante.
- `timeSpent`: tiempo transcurrido desde que se inició el trabajo.
- `statusMensage`: mensaje asociado al estado (*status*) de la tarea.
- `jobName`: nombre de la tarea.
- `infoMessage`: mensaje de información relativo al trabajo lanzado.
- `stats`: booleano que permite diferenciar las tareas definidas en nuestro trabajo con las de *GWC*, está relacionado con el enlace que permite eliminar todas las estadísticas de acceso relativas a esta capa de la base de datos correspondiente. Este es un concepto solo asociado a las tareas de *prefetch* y de análisis que se definen en esta nueva implementación, por lo que no tiene sentido lanzarlo sobre las tareas de *GWC*. Este booleano permitirá comprobar que tarea se está manejando y lanzar un mensaje de error si el usuario quiere purgar una base de datos para una tarea para la cual no es posible. De esta manera valdrá `true` para las tareas que permiten purgar base de datos y `false` en caso contrario.
- `storeId`: identificador del almacén (base de datos) de donde esta tarea extrae las estadísticas para sus propósitos.

Pasamos a analizar el panel.

```
public class TaskPanel extends Panel {…}
```

Código 130. Extracto `TaskPanel` I

En primer lugar se definen los elementos que formarán parte del panel:

```
private static Logger log = Logging.getLogger(TaskPanel.class);

TaskEntryProvider entryProvider;
GeoServerTablePanel<TaskEntry> taskTable;
List<TaskEntry> items;
String layerName;
GeoServerDialog infoDialog;
GeoServerDialog statusDialog;
GeoServerDialog adviceDialog;
GeoServerDialog deleteDialog;
```

Código 131. Extracto `TaskPanel` II

Cuyo significado es el siguiente:

- `entryProvider`: proveedor de las entradas de la tabla.
- `taskTable`: la propia tabla.
- `items`: lista de entradas.

- `layerName`: nombre de la capa para la que se muestras las tareas.
- `infoDialog`: diálogo emergente en que se muestra la información de la tarea, lanzado tras pulsar una imagen-enlace.
- `statusDialog`: diálogo emergente en que se muestran mensajes relativos al estado de la tarea, lanzado tras pulsar una imagen-enlace.
- `adviceDialog`: mensaje de aviso utilizado para advertir al usuario que la funcionaldad de purgar la base de datos no está operativa en las tareas de *GWC*.
- `deleteDialog`: diálogo emergente que solicita confirmación al usuario para borrar las estadísticas de acceso relacionadas con una tarea.

El constructor de la clase tiene la misma estructura que cualquiera de los explicados anteriormente y solo nos detendremos en los aspectos relevantes:

```
	public TaskPanel(String id, final String layerName) {
		super(id);
		this.layerName = layerName;
		items = new ArrayList<TaskEntry>();
		if (layerName != null) {
			getTasks(layerName);
		}

		add(popupWindow = new ModalWindow("popup"));

		entryProvider = new TaskEntryProvider(items);
		add(taskTable = new GeoServerTablePanel<TaskEntry>("entry",
		entryProvider) {

			@Override
			protected Component getComponentForProperty(String id, IModel
			itemModel, Property<TaskEntry> property) {
				Component result;
				if (property == TaskEntryProvider.ID) {
					result = taskIdLink(id, itemModel);
				} else if (property == TaskEntryProvider.GRIDSET) {
					result = gridsetLink(id, itemModel);
				} else if (property == TaskEntryProvider.FORMAT) {
					result = formatLink(id, itemModel);
				} else if (property == TaskEntryProvider.STATUS) {
					result = statusLink(id, itemModel);
				} else if (property == TaskEntryProvider.INFO) {
					result = infoLink(id, itemModel);
				} else if (property == TaskEntryProvider.TYPE) {
					result = typeLink(id, itemModel);
				} else if (property == TaskEntryProvider.COMPLETEDBAR) {
					result = completedBarLink(id, itemModel);
				} else if (property == TaskEntryProvider.COMPLETEDSTATUS)
				{
					result = completedStatusLink(id, itemModel);
				} else if (property == TaskEntryProvider.TIMEBAR) {
					result = timeBarLink(id, itemModel);
				} else if (property == TaskEntryProvider.TIMEREMAINING) {
					result = timeRemainingLink(id, itemModel);
				} else if (property == TaskEntryProvider.ACTION) {
					result = actionLink(id, itemModel);
				} else {
					result = null;
				}

				return result;
			}
		}.setFilterable(false));
```

```
		taskTable.setItemReuseStrategy(new DefaultItemReuseStrategy());
		taskTable.setOutputMarkupId(true);

		add(new AjaxLink("refreshList") {
			@Override
			public void onClick(AjaxRequestTarget target) {
				log.info("Refreshing Task List");

				items.clear();
				getTasks(layerName);
				target.addComponent(taskTable);
			}
		});

		add(infoDialog = new GeoServerDialog("infoDialog"));
		infoDialog.setInitialWidth(500);
		infoDialog.setInitialHeight(300);

		add(statusDialog = new GeoServerDialog("statusDialog"));
		statusDialog.setInitialWidth(500);
		statusDialog.setInitialHeight(150);

		add(adviceDialog = new GeoServerDialog("adviceDialog"));
		adviceDialog.setInitialWidth(500);
		adviceDialog.setInitialHeight(130);

		add(deleteDialog = new GeoServerDialog("deleteDialog"));
		deleteDialog.setInitialWidth(500);
		deleteDialog.setInitialHeight(85);
	}
```

Código 132. Extracto `TaskPanel` III

¿Cómo se inicializa la lista de entradas? Este es quizás el aspecto más importante de este panel, por ello se ha reservado una función para ello, `getTasks()`:

```
	public void getTasks(String layerName) {…}
```

Código 133. Extracto `TaskPanel` IV

Lanzada esta función el primer paso es, mediante la siguiente línea, obtener todas las tareas `GWCTask` a partir del nombre de la capa/grupo de capas recibido como parámetro.

```
	List<GWCTask> listGWCTask = Seeder.get().getGWCTasks(layerName);
```

Código 134. Extracto `TaskPanel` V

Para ello hace uso del método `getGWCTasks()` del objeto `Seeder` que presenta la siguiente estructura:

```
	public List<GWCTask> getGWCTasks(String layerName) {

		List<GWCTask> items = new ArrayList<GWCTask>();

		Iterator<GWCTask> iter = tileBreeder.getRunningAndPendingTasks();
		collectGWTTasks(layerName, items, iter);
		iter = this.GWCTasksRegistry.iterator();
		collectGWTTasks(layerName, items, iter);

		return items;
	}
```

```
private void collectGWCTasks(String layerName, List<GWCTask> items,
Iterator<GWCTask> iter) {
	while (iter.hasNext()) {
		GWCTask task = iter.next();
		if (layerName.equals(task.getLayerName()) &&
		!items.contains(task)) {
			items.add(task);
		}
	}
}
```

Código 135. Métodos `getGWCTask()` y `collectGWCTasks()` del objeto `Seeder`

Este método hace uso de las herramientas de *GWC* para obtener todas las tareas lanzadas en la aplicación. El método `collectGWCTasks()` del mismo objeto `Seeder`, recibe el nombre de la capa para la que se está buscando sus tareas en ejecución, el iterador que se obtuvo mediante *GWC* o a partir del registro de tareas `GWCTasksRegistry` del objeto `Seeder`, que contiene todas las tareas lanzadas y la lista de tareas de esta capa vacía que inicializó el método anterior y que este se encargará de rellenar. Para ello escoge solamente aquellas tareas que están asociadas a la capa con nombre igual al nombre que se recibe como parámetro siempre y cuando no esté ya añadida. En la lista `GWCTasksRegistry` se van añadiendo todos los trabajos `PrefetchTask` cuando se lanzan para evitar problema de que *GWC* no los maneje correctamente.

De vuelta el método `getTasks()` del panel, se analiza cada una de estas tareas seleccionadas para crear una entrada de `TaskEntry`. Las tareas de esta lista van a ser principalmente de tres tipos, que son los que se consideran a la hora de mostrar información en el panel: `SeedTask`, `TruncateTask` y `PrefetchTask`. Las dos primeras son de *GWC*, la última ha sido añadida mediante el módulo *Seeder* para el que este trabajo esta implementado la interfaz gráfica para su funcionamiento.

Para cada una de estas tareas:

```
for (int i = 0; i < listGWCTask.size(); i++) {
	GWCTask task = listGWCTask.get(i);
	…
}
```

Código 136. Extracto `TaskPanel` VI

se deben obtener los parámetros necesarios para completar una entrada. Estas tareas contendrá una serie de parámetros comunes que se pueden obtener de la siguiente manera haciendo uso de las herramientas que el objeto `GWCTask` de *GWC* presenta, otra serie de parámetros se inicializan vacíos y después se completan.

```
Long taskId = task.getTaskId();
String layerNameTask = task.getLayerName();
String status = task.getState().toString();
String type = task.getType().toString();
Long tilesTotal = task.getTilesTotal();
Long tilesDone = task.getTilesDone();
Long timeRemaining = task.getTimeRemaining();
Long timeSpent = task.getTimeSpent();
String gridSetID = null;
String format = null;
String jobName = "";
```

```
String statusMessage = "";
String infoMessage = "";
boolean stats = false;
String storeId = "";
```

Código 137. Extracto `TaskPanel` VII

Llegados a este punto se debe diferenciar entre las tres tareas para poder extraer de ellas los valores de *GridSet* y formato, lo cual no es sencillo, pero se usará reflexión para solventarlo:

```
Class<? extends GWCTask> taskClass = task.getClass();
```

Código 138. Extracto `TaskPanel` VIII

Mediante una estructura `if`, se analizan los tres casos posibles. Por ejemplo, empezando con la clase `SeedTask`:

```
if (taskClass.getName().equals("org.geowebcache.seed.SeedTask")) {
	try {
		Field fieldIter = taskClass.getDeclaredField("trIter");
		fieldIter.setAccessible(true);
		Object value = fieldIter.get(task);
		if (value instanceof TileRangeIterator) {
			TileRangeIterator tileIterator =
				(TileRangeIterator) value;
			gridSetID = tileIterator.getTileRange().getGridSetId();
			format = tileIterator.getTileRange()
				.getMimeType().getFormat();
			int zoomStart = tileIterator
				.getTileRange().getZoomStart();
			int zoomStop = tileIterator.getTileRange().getZoomStop();

			boolean reseed;
			if (type.equals("RESEED")) {
				reseed = true;
			} else
				reseed = false;
			 statusMessage = "Running Seed Task.";
			infoMessage =
			"<html><p>SEED TASK<br/> &nbspLayer: "
			+ layerName + "<br/> &nbspGridSet: "
			+ gridSetID + "<br/> &nbspFormat: " + format
			+ "<br/> &nbspZoom Start: " + zoomStart
			+ "<br/> &nbspZoom Stop : " + zoomStop
			+ "<br/> &nbspReesed: " + reseed
			+ "</p></html>";

		}
	} catch (NoSuchFieldException e) {
		log.warning(e.getLocalizedMessage());
	} catch (SecurityException e) {
		log.warning(e.getLocalizedMessage());
	} catch (IllegalArgumentException e) {
		log.warning(e.getLocalizedMessage());
	} catch (IllegalAccessException e) {
		log.warning(e.getLocalizedMessage());
	}

}
```

Código 139. Extracto `TaskPanel` XIX

Para extraer estos dos valores necesarios se usan las herramientas de *GWC* (líneas 3-14). Por otro lado, se busca si la tarea se lanzó para realizar *Seed* o *ReSeed* (líneas 16-20). Las líneas 21-29 tienen por objetivo generar los mensajes de *status* e información que se mostrarán al usuario asociados a esta tarea. Sería deseable que estos se generasen en la propia tarea y no en este punto, pero como esta está definida por *GWC*, no podemos modificar su implementación. Todo esto se ha situado dentro de una estructura `try-catch` debido a las exigencias de las herramientas de *GWC* que se mencionaron previamente.

De manera análoga se obtienen los valores de los parámetros para las tareas `TruncateTask`:

```
if (taskClass.getName().equals("org.geowebcache.seed.TruncateTask")) {
	try {
		Field fieldIter = taskClass.getDeclaredField("tr");
		fieldIter.setAccessible(true);
		Object value = fieldIter.get(task);
		if (value instanceof TileRange) {
		TileRange tileRange = (TileRange) value;
		gridSetID = tileRange.getGridSetId();
		format = tileRange.getMimeType().getFormat();
		int zoomStart = tileRange.getZoomStart();
		int zoomStop = tileRange.getZoomStop();

		statusMessage = "Running Truncate Task.";
		infoMessage = "<html><p>TRUNCATE TASK<br/> &nbspLayer: "
		+ layerName + "<br/> &nbspGridSet: " + gridSetID
		+ "<br/> &nbspFormat: " + format
		+ "<br/> &nbspZoom Start: " + zoomStart
		+ "<br/> &nbspZoom Stop : "
		+ zoomStop + "</p></html>";

		}
	} catch (NoSuchFieldException e) {
		Log.warning(e.getLocalizedMessage());
	} catch (SecurityException e) {
		Log.warning(e.getLocalizedMessage());
	} catch (IllegalArgumentException e) {
		Log.warning(e.getLocalizedMessage());
	} catch (IllegalAccessException e) {
		Log.warning(e.getLocalizedMessage());
	}

}
```

Código 140. Extracto `TaskPanel` X

Y las tareas `PrefetchTask`:

```
if (task instanceof PrefetchTask) {
	PrefetchTask ptask = (PrefetchTask) task;
	gridSetID = ptask.getGridSubset().getName();
	format = ptask.getMimeType().getFormat();
	jobName = ptask.getJobName();
	statusMessage = ptask.getStatusMessage();
	infoMessage = ptask.getInfoMessage();
	stats = true;
	storeId = ptask.getStoreId();
}
```

Código 141. Extracto `TaskPanel` XI

Una vez obtenidos los datos necesarios, se crea la entrada `TaskEntry` y se añade a la lista:

```
TaskEntry entry = new TaskEntry(jobName, taskId.toString(), layerNameTask,
        gridSetID, format, status, type, tilesTotal, tilesDone,
        timeRemaining, timeSpent, statusMessage, infoMessage, stats, storeId);
items.add(entry);
```

Código 142. Extracto `TaskPanel` XII

Este último tipo de tareas son propuestas por esta solución por lo que se pueden modificar para que devuelvan de manera sencilla los parámetros que se necesitan. Por ello se puede ver que en este último caso no se generan los mensajes si no que se obtienen del objeto que representa la tarea y los métodos no obligan a utilizar una estructura `try-catch`.

Existe otro tipo de tareas que serán reflejadas en este panel. Se trata de las tareas de análisis previas a las tareas `PrefetchTask` que se encargan de recopilar los datos necesarios para esta tarea. Para obtenerlas se utiliza el método `getPrefetchingJobBeans()` del objeto `Seeder`.

```
JobRegistry seeder = Seeder.get();
List<PrefetchingJob> jobs = seeder.getPrefetchingJobBeans();
```

Código 143. Extracto `TaskPanel` XII

Que se implementa de la siguiente manera:

```
public List<PrefetchingJob> getPrefetchingJobBeans() {
        return this.prefetchingJobList;
}
```

Código 144. Método `getPrefetchingJobBeans()` del objeto `Seeder`

Simplemente devuelve la lista de tareas de análisis que se crea en el objeto `Seeder`. Cada vez que se lanza una tarea de análisis (`PrefetchingJob` explicado en la Sección 4.4.2) se accede a esta lista y se añade un nuevo elemento, para después poder recuperar todas las tareas para completar el panel. El objeto `Seeder` implementa la interfaz `JobRegistry` que forma parte del módulo *Seeder-Prefecth*. Esta definición permite construir un puente entre el módulo *Seeder-Prefetch* y el módulo *Seeder-Interfaz Web*. De esta manera desde el módulo *Seeder-Prefetch* se puede acceder al módulo *Seeder-Interfaz Web* y viceversa. El objetivo principal será poder acceder a la lista de tareas de análisis lanzadas, y después de forma particular a cada una de ellas.

Una vez cargada la lista, a partir de cada una de sus elementos se crean las entradas correspondientes siempre y cuando se correspondan con la capa para la que se están visualizando las tareas:

```
1  for (PrefetchingJob prefetchingJobBean : jobs) {
2          String layerN = prefetchingJobBean.getLayerId();
3          String gridsetName = prefetchingJobBean.getGridSetId();
4          String state = prefetchingJobBean.getStatus();
5          String format = prefetchingJobBean.getFormat();
6          String jobName = prefetchingJobBean.getJobName();
7          long timeSpent = 0;
8          long timeRemaining = 0;
9          long done = prefetchingJobBean.getStep();
10         long total = 100;
11         long actualTime = (new Date(System.currentTimeMillis())).getTime();
12         long initWaitTime = 0;
13         long finishWaitTime;
14         String statusMessage = prefetchingJobBean.getStatusMessage();
```

```
15          String infoMessage = prefetchingJobBean.getInfoMessage();
16          boolean stats = true;
17          String storeId = prefetchingJobBean.getStoreId();
18          if (prefetchingJobBean.getLayerId().equals(layerName)) {
19                 if (state.equals(PrefetchingJob.JOB_STATUS_WAITING)) {
20                        initWaitTime =
21                               prefetchingJobBean.getPreviousFireTimeInMSecond();
22                        finishWaitTime =
23                               prefetchingJobBean.getNextFireTimeInMSecond();
24                        actualTime =
25                               (new Date(System.currentTimeMillis())).getTime();
26
27                        timeSpent = (actualTime - initWaitTime) / 1000;
28                        timeRemaining = (finishWaitTime - actualTime) / 1000;
29
30                 } else {
31                 timeSpent = prefetchingJobBean.getTimeSpent() / 1000;
32                 timeRemaining = (prefetchingJobBean.getTotalTime() -
33                               prefetchingJobBean.getTimeSpent()) / 1000;
34                 }
35
36                 TaskEntry entry = new TaskEntry(jobName, "0", layerN,
37                        gridsetName, format, state, "ANALYSIS", total, done,
38                        timeRemaining, timeSpent,statusMessage, infoMessage,
39                        stats, storeId);
40                        items.add(entry);
41
42          }
43
44   }
```

Código 145. Extracto `TaskPanel` XIII

Los objetos `PrefetchingJob` están preparados para poder devolver cada uno de los parámetros que se necesiten para mostrar la información correspondiente al usuario.

En el caso de las tareas de análisis el número de teselas totales o realizadas se debe interpretar como el número de pasos. Es decir, se verá que las tareas de análisis están compuestas de una serie de pasos que se reparten el 100% del total del procesamiento requerido. En lugar de indicar el número de teselas totales y ya completadas, se indicará el tanto por ciento de ejecución de la tarea, por ello el total se define como 100 y las teselas hechas, el tanto por ciento completado.

Hay que diferenciar que las tareas de análisis pueden estar ejecutándose o en espera a que se lance la siguiente repetición si así fuera. Por ello la estructura `if-else` (líneas 19-34 del código 145) permite diferenciar estos dos casos. En la Sección 4.4.2 se explicará que devuelve cada uno de los métodos que se aquí se utilizan para calcular el tiempo restante y el tiempo empleado que se necesitan para crear la entrada. En concreto, se analizará que se entenderá por estos tiempos en cada caso y como se procede a su estimación. Lo que si se indica, es que estos tiempos devueltos se miden en milisegundos, y para inicializar las entradas se debe expresar en segundos, por ello se divide entre 1000.

En las líneas 36-40 del código 145 se inicializa una nueva entrada `TaskEntry` a partir de estos datos.

Diferenciar que las tareas GWCTask tienen asociado un identificador único que proporciona *GWC* de manera que no pueda haber datos tareas con el mismo id. A todas las tareas de análisis se les asigna el identificador 0, que nunca puede ser asignado a una tarea *GWC*.

Para finalizar con el constructor se indica la clase estática que actúa como proveedor de entradas definiendo la estructura de las columnas de la tabla:

```
	static class TaskEntryProvider extends GeoServerDataProvider<TaskEntry> {

		public static Property<TaskEntry> ID = new
			PropertyPlaceholder<TaskEntry>("Id");

		public static Property<TaskEntry> GRIDSET = new
			PropertyPlaceholder<TaskEntry>("Gridset");

		public static Property<TaskEntry> FORMAT = new
			PropertyPlaceholder<TaskEntry>("Format");

		public static Property<TaskEntry> STATUS = new
			PropertyPlaceholder<TaskEntry>("Status");

		public static Property<TaskEntry> INFO = new
			PropertyPlaceholder<TaskEntry>("Info");

		public static Property<TaskEntry> TYPE = new
			PropertyPlaceholder<TaskEntry>("Type");

		public static Property<TaskEntry> COMPLETEDBAR = new
			PropertyPlaceholder<TaskEntry>("Completed");

		public static Property<TaskEntry> COMPLETEDSTATUS = new
			PropertyPlaceholder<TaskEntry>("");

		public static Property<TaskEntry> TIMEBAR = new
			PropertyPlaceholder<TaskEntry>("Time");

		public static Property<TaskEntry> TIMEREMAINING = new
			PropertyPlaceholder<TaskEntry>("");

		public static Property<TaskEntry> ACTION = new
			PropertyPlaceholder<TaskEntry>("Action");

		static List PROPERTIES = Arrays
			.asList(ID, GRIDSET, FORMAT, STATUS, INFO, TYPE,
			COMPLETEDBAR, COMPLETEDSTATUS, TIMEBAR, TIMEREMAINING, ACTION);

		List<TaskEntry> items;

		public TaskEntryProvider(List<TaskEntry> items) {
			this.items = items;
		}

		@Override
		protected List<TaskEntry> getItems() {
			return items;
		}

		@Override
		protected List<Property<TaskEntry>> getProperties() {
			return PROPERTIES;
		}
	}
```

Código 146. Extracto TaskPanel XIV

El siguiente paso es explicar cómo se debe completar la información de cada columna:

- **Identificador de tarea**:

```
Component taskIdLink(String id, IModel itemModel) {
	TaskEntry entry = (TaskEntry) itemModel.getObject();
	return new Label(id, entry.getTaskId().toString());
}
```

Código 147. Extracto TaskPanel XV

- **Nombre del *GridSet***:

```
Component gridsetLink(String id, IModel itemModel) {
	TaskEntry entry = (TaskEntry) itemModel.getObject();
	return new Label(id, entry.getGridset() == null ? "" :
	entry.getGridset());
}
```

Código 148. Extracto TaskPanel XVI

- **Formato de imagen**:

```
Component formatLink(String id, IModel itemModel) {
	TaskEntry entry = (TaskEntry) itemModel.getObject();
	return new Label(id, entry.getFormat() == null ? "" :
		entry.getFormat());
}
```

Código 149. Extracto TaskPanel XVII

Estas celdas se completan directamente utilizando el método que permite obtener el parámetro correspondiente a partir del objeto TaskEntry asociado.

- **Estado de la tarea**: se implementa mediante un icono que al ser pulsado lanza una ventana emergente con el mensaje de estado asociado. Además este icono es ilustrativo del estado al que representa. Por ejemplo, si se quiere indicar que el estado de la tarea es *"Completado"* aparecerá como icono un *tick* de aprobación, si se quiere indicar que ha habido algún error, se mostrará un icono de advertencia… En la sección dedicada al *"Manual de Usuario*" se mostrará una descripción de los distintos iconos que se puede encontrar el usuario y su significado:

```
1  Component statusLink(String id, IModel itemModel) {
2  	final TaskEntry entry = (TaskEntry) itemModel.getObject();
3
4  	Fragment f = new Fragment(id, "status", TaskPanel.this);
5
6  	ResourceReference icon;
7
8  	String title = "";
9  	String heading = "";
10 	if (entry.getStatus().equals(PrefetchingJob.JOB_STATUS_ERROR)) {
11 		icon = new ResourceReference(getClass(),
12 			"../../../img/icons/silk/error.png");
13 		title = "Error";
14 		heading = "Error in analysis";
15 	} else if (entry.getStatus().equals(PrefetchingJob.JOB_STATUS_WAITING))
16 	{
```

```
17                  icon = new ResourceReference(getClass(),
18                          "../../../img/icons/silk/arrow_refresh.png");
19                  title = "Waiting";
20                  heading = "Waiting for a new analysis";
21          } else if (entry.getStatus().equals(PrefetchingJob.JOB_STATUS_RUNNING)
22                  || entry.getStatus().equals(GWCTask.STATE.RUNNING)) {
23                  icon = new ResourceReference(getClass(),
24                          "../../../img/icons/silk/bullet_go.png");
25                  title = "Running";
26                  heading = "Task in execution";
27          } else if (entry.getStatus().equals(PrefetchingJob.JOB_STATUS_ENDED) ||
28          entry.getStatus().toString().equals(GWCTask.STATE.DONE.toString())) {
29                  icon = new ResourceReference(getClass(),
30                          "../../../img/icons/silk/tick.png");
31                  title = "Ended";
32                  heading = "Task completed";
33          } else {
34                  icon = new ResourceReference(getClass(),
35                          "../../../img/icons/silk/bullet_blue.png");
36                  title = entry.getStatus().toString();
37                  heading = entry.getStatus().toString();
38          }
39
40          final String adviceTitle = title;
41          final String messageHeading = heading;
42
43          ImageAjaxLink statusLink = new ImageAjaxLink("statusLink", icon) {
44
45                  @Override
46                  protected void onClick(AjaxRequestTarget target) {
47                          String message = "";
48                          if (entry.getStatusMessage() != null) {
49                                  message = entry.getStatusMessage();
50                          } else
51                                  message = "Information not available";
52                          statusDialog.setTitle(new Model<String>(adviceTitle));
53                          IModel<String> modelHeading = new
54                                  Model<String>(messageHeading);
55                          IModel<String> modelMessages = new
56                                  Model<String>(message);
57                          statusDialog.showInfo(target, modelHeading,
58                                  modelMessages);
59                  }
60          };
61
62          f.add(statusLink);
63          return f;
64  }
```

Código 150. Extracto `TaskPanel` XVIII

Las líneas 10-38 del código 150 permiten cargar un icono u otro en función del estado en que se encuentre la tarea. Los estados se definen como constantes del elemento `PrefetchingJob` definido en el *Modulo-Prefetch*. Se debe tener en cuenta que:

- Las tareas *GWC* pueden estar en ejecución o completadas (líneas 21-32 del código 150).
- Las tareas de análisis pueden estar en espera o en ejecución (líneas 15-26 del código 150).
- Se ha implementado un estado por defecto, para cualquier otro estado no contemplado como típico (líneas 33-38 del código 150) que muestra un icono neutro.

A parte del icono, creamos un título (`title`) y una cabecera (`heading`) que asociaremos a la ventana emergente y que están relacionados con el estado. Una vez cargado el icono se implementa con él un enlace. Al ser pulsado debe lanzar una ventana emergente (`statusDialog` definido anteriormente) en que se muestre el mensaje de estado asociado a esta entrada y guardado como parte del objeto `TaskEntry`.

- **Información de la tarea**: se implementa mediante un icono que al ser pulsado lanza una ventana emergente en que se muestra un mensaje informativo sobre la tarea con la estructura indicada al inicio de esta sección:

```
1  Component infoLink(String id, IModel itemModel) {
2          final TaskEntry entry = (TaskEntry) itemModel.getObject();
3
4          Fragment f = new Fragment(id, "info", TaskPanel.this);
5
6          ResourceReference icon;
7          icon = new ResourceReference(getClass(),
8                  "../../../img/icons/silk/information.png");
9          String title = "Info";
10         String heading = "Task Information";
11
12         final String adviceTitle = title;
13         final String messageHeading = heading;
14
15         ImageAjaxLink infoLink = new ImageAjaxLink("infoLink", icon) {
16
17                 @Override
18                 protected void onClick(AjaxRequestTarget target) {
19                         String message = "";
20                         if (entry.getStatusMessage() != null) {
21                                 message = entry.getInfoMessage();
22                         } else
23                                 message = "Information not available";
24                         infoDialog.setTitle(new Model<String>(adviceTitle));
25                         IModel<String> modelHeading = new
26                                 Model<String>(messageHeading);
27                         IModel<String> modelMessages = new
28                                 Model<String>(message);
29                         infoDialog.showInfo(target, modelHeading, modelMessages);
30                                 return;
31
32                 }
33         };
34
35         f.add(infoLink);
36
37         return f;
38
39 }
```

Código 151. Extracto `TaskPanel` XIX

Se carga un icono representativo del concepto de información y se añade un título y una cabecera a la ventana emergente (líneas 6-10 del código 151). Con esta información se crea una imagen-enlace que al ser pulsada (líneas 15-33 del código 151) abre una ventana emergente donde se carga el mensaje de información que tenga asociada esta tarea en caso de que tenga (línea 21) o uno por defecto si no tiene (línea 23)

- **Tipo de tarea**: directamente a partir del objeto `TaskEntry` obtenemos si la tarea es de *Seed*, *Truncate*, *Prefecth* o Análisis (*Analysis*).

```
Component typeLink(String id, IModel itemModel) {
          TaskEntry entry = (TaskEntry) itemModel.getObject();
          return new Label(id, entry.getType());

}
```

Código 152. Extracto `TaskPanel` XX

- **Barra de estado**: se implementa como una barra típica de progreso que se va llenando a medida que avanza el proceso:

```
1  Component completedBarLink(String id, IModel itemModel) {
2         TaskEntry entry = (TaskEntry) itemModel.getObject();
3         Fragment f = new Fragment(id, "completed", TaskPanel.this);
4
5          final IModel<Number> limitModel = new
6                 Model<Number>(entry.getTilesTotal());
7         final IModel<Number> usedModel = new
8                 Model<Number>(entry.getTilesDone());
9
10        StatusTaskBar statusBar = new StatusTaskBar("statusBarCompleted",
11                limitModel, usedModel);
12        f.add(statusBar);
13
14        if (entry.getTaskId().equals("0") &&
15         entry.getStatus().equals(PrefetchingJob.JOB_STATUS_WAITING)) {
16                return new Label(id, "Waiting for new analysis to start");
17        } else
18                return f;
19 }
```

Código 153. Extracto `TaskPanel` XXI

Se crea haciendo uso del objeto `StatusTaskBar` que se presentará a continuación. Su implementación es similar a la que utiliza *GeoServer*. Para generar la barra de progreso necesita el valor límite y el valor realizado (líneas 4-11 del código 153).

```
public class StatusTaskBar extends Panel {

        private static final long serialVersionUID = 1L;

        @SuppressWarnings("deprecation")
        public StatusTaskBar(final String id, final IModel<Number> limitModel,
        final IModel<Number> progressModel) {
                super(id);
                setOutputMarkupId(true);
                add(HeaderContributor.forCss(StatusTaskBar.class,
                        "statusTask.css"));

                WebMarkupContainer usageBar = new
                        WebMarkupContainer("statusBarProgress");
                WebMarkupContainer excessBar = new
                        WebMarkupContainer("statusBarExcess");

                final double limit = limitModel.getObject().doubleValue();
                final double used = progressModel.getObject().doubleValue();
                final double excess = used - limit;

                int usedPercentage;
                int excessPercentage;
```

```
                final int progressWidth = 80;// progress bar with, i.e. 100%

                if (excess > 0) {
                        excessPercentage = (int) Math.round((excess *
                                progressWidth) / used);
                        usedPercentage = progressWidth - excessPercentage;
                } else {
                        usedPercentage = (int) Math.round(used * progressWidth /
                                limit);
                        excessPercentage = 0;
                }

                usageBar.add(new AttributeModifier("style", true, new
                        Model<String>("width: " + usedPercentage
                        + "px; border-left: inherit;")));

                String redStyle = "width: " + excessPercentage + "px; left: " +
                        (usedPercentage) + "px;";

                excessBar.add(new AttributeModifier("style", true, new
                Model<String>(redStyle)));

                add(usageBar);
                add(excessBar);
        }
}
```

Código 154. Objeto StatusTaskBar

Tiene asociado el siguiente fichero *HTML* (`StatusTaskBar.html`) en que se define su estructura:

```
<html xmlns:wicket="http://wicket.apache.org/">
<body>
<wicket:panel>
 <div class="status_bar">
  <span class="statusBarProgress" wicket:id="statusBarProgress"></span>
  <span class="statusBarExcess"   wicket:id="statusBarExcess"></span>
 </div>
</wicket:panel>
</body>
</html>
```

Código 155. Fichero `StatusTaskBar.html`

Su estilo se especifica mediante Hojas de Estilo en Cascada (`statusTask.css`):

```
.status_bar {
        width: 80px;
        height: 10px;
        float: left;
        margin: 4px;
        background-color: #e8e6e6;
        border: 1px solid #555;
        -moz-border-radius: 5px;
        -webkit-border-radius: 5px;
        border-radius: 5px;
}

.statusBarProgress {
        position: absolute;
        z-index: 3;
        background-color: green;
        height: 10px;
```

```
		width: 50px;
	}

	.statusBarExcess {
		position: absolute;
		z-index: 4;
		background-color: red;
		height: 10px;
		width: 50px;
		left: 50px;
	}
```

Código 156. Fichero `StatusTaskBar.css`

En concreto, aparecerá una barra vertical con bordes redondeados en que una línea horizontal verde del mismo grosor que la barra irá poco a poco cubriendo toda la barra. Cuando esté cubierta por completo, significará que se ha completado el 100% del progreso.

Para las tareas de análisis, en el caso en que estén en estado de espera a iniciar una nueva ejecución, no se muestra una barra de progreso, simplemente la frase "*Esperando el inicio de un nuevo análisis*" (línea 16 del código 153).

- **Frase explicativa del progreso**: junto a esta barra aparecerá indicado el tanto por ciento de progreso junto con el número total de teselas almacenadas y las totales a procesar. En caso de un análisis mostrará el tanto por ciento de ejecución, junto con el número de pasos realizados sobre un total de 100 tal y como se indicó anteriormente.

```
Component completedStatusLink(String id, IModel itemModel) {
		TaskEntry entry = (TaskEntry) itemModel.getObject();

		if (entry.getTaskId().equals("0") &&
		entry.getStatus().equals(PrefetchingJob.JOB_STATUS_WAITING)) {
			return new Label(id, "");
		} else {
			long porcentage;

			if (entry.getTilesTotal() != 0) {
				porcentage = 100 * entry.getTilesDone() /
					entry.getTilesTotal();
			} else {
				porcentage = 0;
			}
			return new Label(id, porcentage + "% (" + entry.getTilesDone()
			+ " / " + entry.getTilesTotal() + ")");

		}

	}
```

Código 157. Extracto `TaskPanel` XXII

En el caso de una tarea de análisis que este en estado de espera no se muestra nada, se deja vacía esta celda.

- **Barra de tiempo transcurrido**: se implementa exactamente igual que la barra de progreso pero sirve para indicar una estimación del tiempo transcurrido con respecto al tiempo total que se cree que tardará en ejecutarse la tarea, ya que es imposible saber con

exactitud cuándo tiempo se necesitará ya que depende de muchos factores, entre ellos la carga de trabajo a la que esté sometida la máquina en que se realiza la tarea.

```
1  Component timeBarLink(String id, IModel itemModel) {
2          TaskEntry entry = (TaskEntry) itemModel.getObject();
3          Fragment f = new Fragment(id, "time", TaskPanel.this);
4
5          Long timeTotal = entry.getTimeRemaining() + entry.getTimeSpent();
6
7          final IModel<Number> limitModel;
8          final IModel<Number> usedModel;
9
10         if (entry.getTimeRemaining() == -1 || entry.getTimeSpent() == -1) {
11                 limitModel = new Model<Number>(0);
12                 usedModel = new Model<Number>(0);
13         } else {
14                 limitModel = new Model<Number>(timeTotal);
15                 usedModel = new Model<Number>(entry.getTimeSpent());
16         }
17
18         StatusTaskBar statusBar = new StatusTaskBar("statusBarTime",
19                 limitModel, usedModel);
20
21         f.add(statusBar);
22         if (entry.getStatus().equals(PrefetchingJob.JOB_STATUS_ERROR) ||
23         entry.getStatus().equals(PrefetchingJob.JOB_STATUS_ENDED) ||
24         entry.getStatus().equals(GWCTask.STATE.DONE.toString())) {
25                 return new Label(id, "");
26         } else
27                 return f;
28 }
```

Código 158. Extracto TaskPanel XXIII

Los valores temporales se obtienen a partir de los valores que se almacenan en la entrada al ser creada, tal y como se explicó anteriormente. Cuando *GWC* todavía no ha estimado cuanto tiempo ha transcurrido o queda para terminar la tarea le da un valor -1. Por ello las líneas 13-16 del código 158 tienen por objetivo generar una barra sin inicializar progreso para evitar problemas. En caso de que la tarea haya finalizado debido a un error o se haya completado no se mostrará esta barra. Sí se mostrará para tareas de análisis en estado de espera, indicando el tiempo que falta para que se inicie el siguiente análisis.

- **Información temporal**: al lado de la barra de progreso temporal se indica el tiempo restante como un *string*:

```
Component timeRemainingLink(String id, IModel itemModel) {
        TaskEntry entry = (TaskEntry) itemModel.getObject();
        String timeRemaining = toTimeString(entry.getTimeRemaining());

        if (entry.getStatus().equals(PrefetchingJob.JOB_STATUS_ERROR) ||
        entry.getStatus().equals(PrefetchingJob.JOB_STATUS_ENDED) ||
        entry.getStatus().equals(GWCTask.STATE.DONE.toString())) {
                return new Label(id, "");
        } else if (entry.getTimeSpent() == 0 || entry.getTimeSpent() == -1 ||
        entry.getTimeRemaining() == -1) {
                return new Label(id, "Estimating...");
        } else
                return new Label(id, timeRemaining + " remaining");
}
```

Código 159. Extracto TaskPanel XXIV

En caso que la tarea se haya parado debido a un error o haya finalizado no se muestra nada. En caso que no se hayan estimado todavía los valores temporales se muestra el mensaje "*Estimando*". En el resto de casos se muestra el tiempo restante.

Se puede saber si todavía no se ha estimado los valores temporales, si el parámetro tiempo transcurrido (`timeSpent`) para las tareas de análisis (`PrefetchingJob`) tiene valor 0 y para las tareas *GWC* los parámetros tiempo transcurrido (`timeSpent`) y restante (`timeRemaining`) de los objetos `SeedTask`, `TruncateTask` o `PrefetchTask` vale -1.

- **Enlaces de acción**: en este punto se representan varios enlaces:
    - Eliminar una entrada correspondiente a una tarea ya finalizada. Una vez que las tareas ya se han completado, permanecen en el panel para que el usuario pueda visualizarlas. Si quiere eliminarlas de este panel puede emplear este enlace.
    - Parar una tarea en ejecución: pulsando este enlace el usuario puede cancelar una tarea que se esté ejecutando. Se deberá tener en cuenta que si se cancela una tarea de análisis en ejecución, se para este análisis pero no las sucesivas repeticiones. Si se cancela un análisis en espera se cancela toda la programación de esta tarea.
    - Purgar la base de datos: este enlace permite eliminar todas las estadísticas de acceso asociadas a esta entrada, es decir, todas las estadísticas que esta tarea utilizaría para realizar las tareas de análisis. Como se ha explicado anteriormente, solo se puede realizar esta operación si las entradas corresponden a tareas de análisis y *Prefecth*, en definitiva, las tareas implementadas mediante los módulo estudiados en este trabajo.

El siguiente código implementa estos enlaces:

```
Component actionLink(String id, IModel itemModel) {

	final TaskEntry entry = (TaskEntry) itemModel.getObject();

	Fragment f = new Fragment(id, "action", TaskPanel.this);

	ParamResourceModel model;
	if (entry.getStatus().equals(PrefetchingJob.JOB_STATUS_RUNNING) ||
	entry.getStatus().equals(GWCTask.STATE.RUNNING)) {
		model = new ParamResourceModel("kill", this);
	} else
		model = new ParamResourceModel("remove", this);

	f.add(killLink("kill", itemModel, model));

	f.add(purgeLink("purge", itemModel, new ParamResourceModel("purge",
		this)));

	return f;
}
```

Código 160. Extracto `TaskPanel` XXV

Los enlaces de eliminar y parar se implementan juntos, en función de que la tarea se esté ejecutando o esté completada se muestra uno y el otro:

```
1  SimpleAjaxLink killLink(String id, IModel itemModel, IModel label) {
2         return new SimpleAjaxLink(id, itemModel, label) {
3
4                 @Override
5                 protected void onClick(AjaxRequestTarget target) {
6
7                         TaskEntry entry = (TaskEntry) getDefaultModelObject();
8
9                         if (!entry.getTaskId().equals("0")) {
10                                // tareas GWC
11                        Long taskId = Long.parseLong(entry.getTaskId());
12                                Seeder.get().killTask(taskId);
13                                Seeder.get().reinitializeGWCTask(taskId);
14                        } else {
15                                if (entry.getStatus().equals(
16                                 PrefetchingJob.JOB_STATUS_RUNNING)) {
17                                Seeder.get().cancelPrefetchingJob(
18                                        entry.getJobName());
19                                } else if (entry.getStatus().equals(
20                                 PrefetchingJob.JOB_STATUS_WAITING))
21                                 Seeder.get().unschedulerPrefetchingJob(
22                                        entry.getJobName());
23                                 Seeder.get().reinitializePrefetchingJob(
24                                        entry.getJobName());
25                                } else {
26                                 Seeder.get().reinitializePrefetchingJob(
27                                 entry.getJobName());
28                                }
29                        }
30
31                        items.clear();
32                        getTasks(layerName);
33                        target.addComponent(taskTable);
34
35                }
36
37        };
38 }
```

Código 161. Extracto TaskPanel XXVI

Al clicar sobre este enlace primero se evalúa si la tarea es de tipo GWCTask o de análisis, tiendo en cuenta que todas las tareas de análisis tienen el identificador 0. En el primer caso (líneas 9-13 del código 161) se obtiene el identificador. A partir de él se lanza el método killTask() del objeto Seeder:

```
public void killTask(Long taskId) {
        tileBreeder.terminateGWCTask(taskId);
}
```

Código 162. Método killTask() del objeto Seeder

Este hace uso de las herramientas de *GWC* para cancelar la tarea GWCTask asociada a este identificador.

El método reinitializeGWCTask():

```
public void reinitializeGWCTask(long id) {
        GWCTask found = findGWCTask(id);
        removeGWCTask(found);
}

public GWCTask findGWCTask(long id) {
        GWCTask found = null;
```

```
		for (int i = 0; i < GWCTasksRegistry.size(); i++) {
			GWCTask task = GWCTasksRegistry.get(i);
			if (task.getTaskId() == id) {
				found = task;
				break;
			}
		}
		return found;
	}

	public void removeGWCTask(GWCTask found) {
		if (found != null)
			GWCTasksRegistry.remove(found);
	}
```

Código 163. Métodos `reinitializeGWCTask()`, `findGWCTask()` y `removeGWCTask()` del objeto `Seeder`

permite en primer lugar buscar el objeto `GWCTask`, es decir la tarea, que encaja con este identificador y después con el método `removeGWCTask()` borra esta tarea del registro de tareas `GWCTakskRegistry()`. La cadena de métodos que se ejecutan al llamar a `reinitializeGWCTask()` está principalmente pensado para borrar las tareas `PrefetchTask`, porque las otras dos de *GWC* se borran perfectamente con el método `killTask()`.

Para las tareas de análisis, si la tarea está ejecutándose, se cancela el análisis pero no la programación de repeticiones (líneas 15-18 del código 161), lo cual permite realizar el método `cancelPrefetchingJob()` del objeto `Seeder`:

```
	public void cancelPrefetchingJob(String jobName) {
		for (int i = 0; i < prefetchingJobList.size(); i++) {
			PrefetchingJob job = prefetchingJobList.get(i);


			if (job.getJobName().equals(jobName)) {
				job.setCancelRequested(true);
			}

		}
	}
```

Código 164. Método `cancelPrefetchingJob()` del objeto `Seeder`

Este busca el objeto `PrefetchingJob` asociado a esta tarea y establece su cancelación. Se verá posteriormente, que mientras este objeto está ejecutando sus funciones, comprueba de manera periódica si entre la anterior verificación y esta se ha llamado al método `setCancelRequested()` con valor `true`, porque significará que se deberá cancelar. Se verá en detalle en la correspondiente sección.

Si está en estado de espera se desprograma la tarea. El método `unschedulerPrefetchingJob()` del objeto `Seeder` hace uso de las herramientas de *Quartz* para llevar a cabo este objetivo:

```
	public void unschedulerPrefetchingJob(String jobName) {

		StdSchedulerFactory factory = new StdSchedulerFactory();
		try {
			Scheduler s = factory.getScheduler();
```

```
                s.unscheduleJob(jobName, null);
            } catch (SchedulerException e) {
                logger.warning("Unable to unscheduler job: " + jobName);
            }
        }
```

Código 165. Método `unschedulerPrefetchingJob()` del objeto `Seeder`

La tarea de análisis puede haberse cancelado por haber encontrado algún problema a la hora de ejecutarse. En esta situación, ya se habrá desprogramado todo (Sección 4.4.2) porque si al lanzar una vez la tarea da error, seguirá dándolo con mucha probabilidad, si se vuelve a lanzar siguiendo el calendario programado. En el panel aparecerá solo información sobre la tarea errónea aunque esta ya no existirá.

Llegados a este punto el usuario lo que querrá será eliminar esta información del panel, así que solo queda eliminar este trabajo de la lista de trabajos. En caso de que la tarea se haya completado estamos en el mismo supuesto que el caso anterior, en el panel solo aparece información sobre una tarea que ya no existe. Ambos casos están recogidos en las líneas 25-28 del código 161. El método `reinitializePrefetchingJob()` permite borrar esta entrada de la lista de entradas de la tabla.

Una vez realizada la actualización correspondiente, se limpia la tabla y se vuelven a cargar las tareas de la capa/grupo de capas en ejecución restantes, para actualizar el contenido del panel.

```
        items.clear();
        getTasks(layerName);
        target.addComponent(taskTable);
```

Código 166. Extracto `TaskPanel` XXVII

Por otro lado se implementa el enlace para purgar la base de datos:

```
1   SimpleAjaxLink purgeLink(String id, IModel itemModel, IModel label) {
2   return new SimpleAjaxLink(id, itemModel, label) {
3
4           @Override
5           protected void onClick(AjaxRequestTarget target) {
6
7                   final TaskEntry entry = (TaskEntry)
8                   getDefaultModelObject();
9                   if (!entry.isStats()) {
10                          adviceDialog.setTitle(new Model<String>("Error"));
11                          IModel<String> modelHeading = new
12                                  Model<String>("Not available");
13                          IModel<String> modelMessages = new
14                                  Model<String>("This operation is "
15                                                  + "not available for this task");
16
17                          adviceDialog.showInfo(target, modelHeading,
18                          modelMessages);
19                  } else {
20                          Map<String, IndexDAO> indexMap =
21                                  Seeder.get().getIndexMap();
22                          final String layerId =
23                                  GWC.get().getTileLayerByName(layerName).getId();
24                          if (!indexMap.containsKey(layerId)) {
25                                  adviceDialog.setTitle(new
26                                          Model<String>("Info"));
```

```
27                                  IModel<String> modelHeading = new
28                                          Model<String>("Cannot Purge Data Base");
29                                  IModel<String> modelMessages = new
30                                          Model<String>("There is not index for "
31                                                  + "this layer in any Data Base");
32                                  adviceDialog.showInfo(target, modelHeading,
33                                          modelMessages);
34                                  log.warning("Cannot Purge Data Base"
35                                          + ". No index definition "
36                                          + "for this layer in any Data Base");
37                                  return;
38
39                          } else {
40                                  deleteDialog.setTitle(new
41                                  Model<String>("Confirm removal of Data Base"
42                                          + " Information?"));
43                                  deleteDialog.showOkCancel(target, new
44                                  GeoServerDialog.DialogDelegate() {
45                                  private static final long
46                                          serialVersionUID = 1L;
47
48                                  @Override
49                                  protected Component
50                                  getContents(final String id) {
51                                          String storeId = entry.getStoreId();
52                                          StoreInfo storeInfo =
53                                          GWC.get().getCatalog().getStore(
54                                          storeId, StoreInfo.class);
55                                          String message = "Data Base asociated "
56                                                  + "with this task is : '"
57                                                  + storeInfo.getName()
58                                                  + "'. Are you sure to purge it? "
59                                                  + "This will completely remove "
60                                                  + "the Prefetching Information "
61                                                  + "related with Job: '"
62                                                  + entry.getJobName() + "'.";
63                                          return new Label(id, message);
64                                          }
65
66                                  @Override
67                                  protected boolean onSubmit(final AjaxRequestTarget
68                                  target, final Component contents) {
69                                          Seeder.get().purgeDB(entry);
70                                          return true;
71                                  }
72
73                                  @Override
74                                  public void onClose(final AjaxRequestTarget
75                                  target) {
76
77                                          target.addComponent(taskTable);
78
79                                  }
80
81                                  @Override
82                                  protected boolean onCancel(final AjaxRequestTarget
83                                  target) {
84                                          final boolean closeWindow = true;
85                                          return closeWindow;
86                                  }
87                          });
88                  }
89
90                  }
91
92          }
```

```
93
94        };
95    }
```

Código 167. Extracto `TaskPanel` XXVIII

En caso de que la entrada sea de tipo `SeedTask` o `TruncateTask`, esta funcionalidad no está implementada, porque no tiene sentido. Para no proceder con el resto de funciones, se comprueba el tipo de tarea sobre la que se ha lanzado este enlace. Para ello, al principio del todo, se definió un atributo booleano llamado `stats` que tendría valor `true` si la tarea fuera `PrefetchTask` o `PrefetchingJob`. Si este booleano tiene valor `false`, se lanza un mensaje de aviso al usuario para indicarle que no puede proceder con esta funcionalidad.

En caso contrario se sigue con el procedimiento. Se obtiene el acceso a la base de datos asociada a esta capa/grupo de capas mediante el mapa `indexMap` del objeto `Seeder` que en anteriores ocasiones se ha enumerado, aquí se puede ver un ejemplo de utilización de este mapa y por qué es necesario. De otra manera sería más complicado acceder a la conexión a la base de datos de una capa concreta para mandar realizar las tareas oportunas, borrar entradas en este caso.

En ese mapa buscamos si existe alguna pareja clave-valor con clave igual al identificador de la capa/grupo de capas. En caso de que por algún motivo, esta búsqueda resulte fallida, se lanza un mensaje de aviso al usuario. En caso contrario, se muestra al usuario un mensaje de confirmación, para que afirme que realmente quiere proceder con el borrado de las estadísticas. Si el cliente cancela o cierra la ventana no se realiza ninguna acción más. En caso de que confirme su acción se lanza el método `purgeDB()` del método `Seeder`:

```
public void purgeDB(TaskEntry entry) {
        String storeId = entry.getStoreId();

        StoreInfo storeInfo = GWC.get().getCatalog().getStore(storeId,
                StoreInfo.class);
        if (storeInfo != null) {
                String layerName = entry.getLayerName();
                String layerId =
                        GWC.get().getTileLayerByName(layerName).getId();
                IndexDAO index = indexMap.get(layerId);
                String jobName = entry.getJobName();

                BoundingBox reqBounds = null;
                PrefetchingJob pjb = findPrefetchingJob(jobName);
                ReferencedEnvelope bbox = pjb.getBbox();
                if (bbox != null) {
                        reqBounds = new BoundingBox(bbox.getMinX(),
                        bbox.getMinY(), bbox.getMaxX(), bbox.getMaxY());
                }
                TileLayer tileLayer = null;
                try {
                        tileLayer = tld.getTileLayer(layerName);
                } catch (GeoWebCacheException e) {
                        logger.warning("Layer {" + layerName
                        + "} does not exist");
                        return;
                }

                GridSubset gridSubset =
                        tileLayer.getGridSubset(pjb.getGridSetId());
```

```
                    if (gridSubset == null) {
                           logger.warning("Layer {" + layerName
                           + "} does not contain GridSet {" + pjb.getGridSetId()
                           + "}");
                           return;
                    }

                    long[] coverageIntersection =
                    gridSubset.getCoverageIntersection(pjb.getRes(), reqBounds);

                    long minx = coverageIntersection[0];
                    long miny = coverageIntersection[1];
                    long maxx = coverageIntersection[2];
                    long maxy = coverageIntersection[3];

                    Map<String, Object> map = new HashMap<String, Object>();
                    map.put("layerName", layerName);
                    map.put("gridset", entry.getGridset());
                    map.put("format", entry.getFormat());

                    if (storeInfo.getType().equals("PostGIS")) {
                           PostgresIndexDAO pIndex = (PostgresIndexDAO) index;
                           try {
                                  pIndex.deleteEntry(layerName, pjb.getGridSetId(),
                                  pjb.getFormat(), maxx, minx, maxy, miny,
                                  pjb.getRes());
                                  logger.warning("Data Base purged");

                           } catch (CannotGetJdbcConnectionException e) {
                                  logger.warning("Cannot purge Data Base");
                           }
                    } else if (storeInfo.getType().equals("H2")) {
                           H2IndexDAO h2Index = (H2IndexDAO) index;
                           try {
                                  h2Index.deleteEntry(layerName, pjb.getGridSetId(),
                                  pjb.getFormat(), maxx, minx, maxy, miny,
                                  pjb.getRes());
                                  logger.warning("Data Base purged");

                           } catch (CannotGetJdbcConnectionException e) {
                                  logger.warning("Cannot purge Data Base");
                           }
                    }
             } else {

                    logger.warning("Cannot purge Data Base because it seems not to
                    exists");

             }

      }
```

Código 168. Método purgeDB() del objeto Seeder

Este método debe preparar una consulta hacia la base de datos muy concreta, ya que debe mandar borrar todas aquellas entradas que esta tarea utilizaría para seleccionar que teselas debe traer a caché, en función de parámetros como nombre de la capa, *GridSet*, formato de imagen, encuadre, resolución… La información concreta que se almacena en la base de datos al visualizar una tesela, se analiza en la correspondiente Sección 4.4.2.

Este método recibirá como parámetros la entrada completa del panel. A partir de ella obtiene todos los parámetros que se necesitan para crear esa consulta. Para ello se hace uso de métodos ya explicados anteriormente. Un método no presentado todavía es

`findPrefetchingJob()`, el cual se usa para encontrar la tarea de análisis que se corresponde con el nombre del trabajo lanzado para poder obtener los parámetros que condicionan la consulta, a partir de la lista en la que se registran todos los trabajos que se lanzan. Para cada uno de ellos comprueba el nombre y se queda con aquel único para el que hay coincidencia:

```
public PrefetchingJob findPrefetchingJob(String jobName) {
	PrefetchingJob found = null;
	for (int i = 0; i < prefetchingJobList.size(); i++) {
		PrefetchingJob job = prefetchingJobList.get(i);
		if (job.getJobName().equals(jobName)) {
			found = job;
			break;
		}
	}
	return found;
}
```

Código 169. Método `findPrefetchingJob()` del objeto `Seeder`

Obtenido el objeto `PrefetchingJob` que representa el trabajo, se obtiene de él los parámetros que se usaron para lanzar la tarea, porque se corresponden con los parámetros que se necesitan para crear la consulta condicional. Una vez obtenidos accedemos al objeto del índice (Ver Sección 4.4.2) y mandamos ejecutar el método `deleteEntry()` pasándole los parámetros que se necesitan para generar la consulta requerida. Los conceptos relativos se analizan en detalle en la sección correspondiente, ya que pertenecen a los módulos *Seeder-Stats* y *Seeder-Prefetch* no creados en este trabajo, si no anteriormente, son la base sobre la que se ha implementado la interfaz. En la sección dedicada a ellos se explicará brevemente la funcionalidad de cada uno de ellos así como las modificaciones que se hayan podido realizar para adecuarlos a la interfaz que se acaba de explicar.

Para finalizar con este panel, se muestra el fichero *HTML* `TaskPanel.html` en que se define su estructura:

```
<html>
<body>
<wicket:panel>
	<fieldset>
		<legend>
			<span><wicket:messagekey="TaskPanel.title">Tasks
			</wicket:message></span>
		</legend>

		<ul>
			<li><label><wicket:message key="taskPanel">Task
			Panel</wicket:message>
			</label></li>
			<li>
				<div wicket:id="entry"></div>
				<div wicket:id="infoDialog"></div>
				<div wicket:id="statusDialog"></div>
				<div wicket:id="adviceDialog"></div>
				<div wicket:id="deleteDialog"></div>
			</li>
			<li><a wicket:id="refreshList"><wicket:message
			key="refreshList">Refresh List</wicket:message>
			</a></li>
		</ul>

		<div wicket:id="popup"></div>
```

```
		</fieldset>
	</wicket:panel>

	<wicket:fragment wicket:id="completed">
		<div wicket:id="statusBarCompleted"></div>
	</wicket:fragment>

	<wicket:fragment wicket:id="time">
		<div wicket:id="statusBarTime"></div>
	</wicket:fragment>

	<wicket:fragment wicket:id="status">
		<div wicket:id="statusLink" />
	</wicket:fragment>

	<wicket:fragment wicket:id="info">
		<div wicket:id="infoLink" />
	</wicket:fragment>

	<wicket:fragment wicket:id="action">
		<div style="white-space: nowrap;">
			<a target="_blank" wicket:id="kill"></a> <br />
			<a target="_blank" wicket:id="purge" align="center"></a> <br />
		</div>
	</wicket:fragment>

	</body>
	</html>
```

Código 170. Fichero `TaskPanel.html`

4.3.4. Definición externa de recursos

Si se ha analizado el contenido de los extractos de código anteriores, se habrá podido apreciar que en muchas ocasiones, los mensajes, títulos y demás que puedan aparecer en la interfaz no se definen en el propio fichero *Java*, si no que se encuentran definidos en ficheros `propierties`. El principal objetivo que se puede destacar de estos ficheros es el poder crear ficheros para distintos idiomas y expresar la frase, mensaje, título en el correspondiente idioma. Cuando el navegador acceda a la interfaz, de todos los ficheros `propierties` usará el relacionado con el idioma del navegador, en caso de que exista, en caso contrario cogería el idioma por defecto.

Objetos como `ResourceModel` y `ParamResourceModel` acceden al fichero concreto que esté manejando el navegador y buscan la frase que necesitan para completar los elementos. Para este trabajo, el fichero `GeoServerApplication.properties` para el idioma por defecto (inglés) tiene la siguiente estructura:

```
#PANEL DE FEATURES SOURCES
FeatureSourcePanel.addLayer                = Add Layer (workspace : layer)
FeatureSourcePanel.addFromFeatureSource    = Add Feature Source (store : feature)
FeatureSourcePanel.featureSource           = Feature Source
FeatureSourcePanel.chooseLayer             = Choose new layer
FeatureSourcePanel.chooseFeatureSource     = Choose Layer From Feature Source
FeatureSourcePanel.up                      = Move layer up
FeatureSourcePanel.down                    = Move layer down

#PANEL DE SELECCIÓN DE STORE
StoreChoicesPanel.select                   = Select
```

```
StoreChoicesPanel.th.DataType              = Data Type
StoreChoicesPanel.th.StoreName             = Store Name

#PANEL DE SELECCIÓN DE GRIDSET
GridsetChoicesPanel.select                 =Select

#PANEL DE VISUALIZACIÓN DEL STORE ESCOGIDO
StoreStatsPanel.selectStore         = Select Store
StoreStatsPanel.title               = Store Stats Configuration
StoreStatsPanel.chooseStore         = Choose store for stats
StoreStatsPanel.th.DataType         = Data Type
StoreStatsPanel.th.StoreName        = Store Name
StoreStatsPanel.advice              = Save configuration to create correct access to
                                    selected store. The data base used for stats and
                                    prefetch is the last saved correctly.


#PANEL DE CONFIGURACIÓN DE GRIDSET
PrefetchJobConfigurationPanel.addJob       = Add Job
PrefetchJobConfigurationPanel.title        = Gridset Configuration
PrefetchJobConfigurationPanel.prefetchingJobConfiguration   = Prefeching Job
                                                              Configuration
PrefetchJobConfigurationPanel.gridsetConfigurationAdvice = You must save
                                    configuration to apply changes before seed
PrefetchJobConfigurationPanel.th.BoundingBox      = Bounding Box
PrefetchJobConfigurationPanel.th.StartDelay       = Start Delay
PrefetchJobConfigurationPanel.th.RepeatCount      = Repeat Count
PrefetchJobConfigurationPanel.th.RepeatInterval   = Repeat Interval
PrefetchJobConfigurationPanel.th.JobName          = Job Name
PrefetchJobConfigurationPanel.delete.title        = Remove Job Configuration

PrefetchJobConfigurationPanel.seed                = Seed
PrefetchJobConfigurationPanel.reseed              = Reseed

PrefetchJobConfigurationPanel.repeatAdvice        = 0 for Repeat Indefinitely


#PANEL DE VISTA DE TEREAS EJECUTÁNDOSE
TaskPanel.title             = Tasks
TaskPanel.taskPanel         = List of currently executing tasks for this layer/group:
TaskPanel.kill              = Kill task
TaskPanel.refreshList       = Refresh List
TaskPanel.purge             = Purge DB
TaskPanel.remove            = Remove

#SUBPANEL PARA ELEGIR DESDE DATA STORES
#Mensajes del panel
DataStoreListPanel.clickOnTheLayerYouWishToConfigure     = Click on the feature you
                                                           wish to select
DataStoreListPanel.listOfResourcesContained              = Here is a list of
                                                 resources contained in the store

#Link para escoger
DataStoreListPanel.select     = Select

#Titulos tabla
DataStoreListPanel.th.name     = Feature name
DataStoreListPanel.th.action   = Action

#Mensaje seleccion de dropdown
addAFeatureFrom      = Add feature from

#EDITOR TAB CAPA/GRUPO CAPAS PARA SEEDER CACHING
#titulo de la página dentro del tab
GeoServerSeederLayerEditor.title    = Seeder caching configuration
```

```
#mensajes de aviso
GeoServerSeederLayerEditor.messageAdvice   = Seeder Configuration only can be modify
if there is saved a Tile Caching Configuration for the same layer/layer group.
GeoServerSeederLayerEditor.messageAdviceSave = It should save at least once this
configuration to apply the changes to the catalog. The values shown are aids to the
user for quick setup, but does not mean they are saved. To check the settings saved
go to data_dir/seeder-config in the installation folder of GeoServer.

#habilitar la configuración
GeoServerSeederLayerEditor.enableSeederForLayer    = Enable Seeder Caching
                                                     Configuration for this layer
GeoServerSeederLayerEditor.enableSeeder.title      = Check this option for enabled
                                               Seeder Caching Configuration for this layer

#chechbox para habilitar/deshabilitar el uso de la configuración seeder
GeoServerSeederLayerEditor.enabled  = Enable validation of parameters for a Prefecth
Job
GeoServerSeederLayerEditor.enabled.title = Used this option if you would like to
check if the parameters are correct for a prefecth job, before to save them.

#DEFINICIÓN DEL TAB SEEDER CACHING PARA LAYER
SeederCacheOptionsTabPanel.title                   = Seeder Caching
SeederCacheOptionsTabPanel.shortDescription        = Configure seeder caching options
                                                     for the layer/group
SeederCacheOptionsTabPanel.geometryLessLabel       = Seeder caching is not possible
                                                     for layers with no geometry column

#DEFINICIÓN DEL PANEL SEEDER CACHING PARA LAYER GROUP
LayerGroupSeederOptionsPanel.title                 = Seeder Caching
LayerGroupSeederOptionsPanel.shortDescription      = Configure seeder caching options
                                                     for the layer group
```

Código 171. Fichero `GeoServerApplication.propierties`

En él se puede apreciar que la forma de definir estos atributos textuales es mediante el nombre del objeto *Java* en que aparecerán, seguido del nombre con que se llamarán desde los objetos `ResourceModel` y `ParamResourceModel`. Por ejemplo:

```
existTileCaching.add(new AttributeModifier("title", true, new
                ResourceModel("enableSeeder.title")));
```

Código 172. Ejemplo uso de recursos externos I

Esta línea está extraída del fichero `GeoServerSeederLayerEditor.java`. En ella se añade a un elemento de tipo *checkbox* la frase explicativa que aparecerá a su lado. En concreto, se está indicando que deberá crear un objeto `ResourceModel` y completarlo con la sentencia definida en el fichero `GeoServerApplication.properties` con nombre `GeoServerLayerEditor.enableSeeder.title`.

También se usan estos objetos para definir estructuras textuales de los ficheros *HTML*. Por ejemplo:

```
<wicket:message key="featureSource">Feature Source</wicket:message>
```

Código 173. Ejemplo uso de recursos externos II

Esta línea está extraída del fichero `FeatureSourcePanel.html`. En ella se define mediante *wickets* un mensaje y se indica como valor para el atributo `key`, `featureSource`. Esto implica que el navegador deberá buscar en el fichero `GeoServerApplication.propierties` una

propiedad con el nombre `FeatureSourcePanel.featureSource` y establecerla como valor de este mensaje. En caso de que no se encuentre mostrará el texto incluido entre las etiquetas del elemento *HTML*.

Las propiedades del tipo `NombrePanel.th.Titulo` permiten especificar un título para las columnas de la tabla que compone el panel. El valor de *"Titulo"* es el que se especifica en el elemento `PropertyPlaceholder`. Por ejemplo:

```
public static Property<StoreInfo> DATATYPE = new
       PropertyPlaceholder<StoreInfo>("DataType");
```

Código 174. Ejemplo uso de recursos externos III

Esta línea esta extraída del fichero `StoreChoicesPanel.java`. En ella se define una columna para la tabla que compone el panel del mismo nombre. Para cambiar mediante una propiedad externa el título de esta columna, dentro del fichero `GeoServerApplication.properties` deberá existir una sentencia de la forma:

```
StoreChoicesPanel.th.DataType = Tipo de dato
```

Código 175. Ejemplo uso de recursos externos IV

Para terminar con esta sección, indicar que en este trabajo se ha generado una interfaz en español a través del siguiente fichero, `GeoServerApplication_es.propierties`:

```
#PANEL DE FEATURES SOURCES
FeatureSourcePanel.addLayer                = Añadir Capa (espacio de trabajo : capa)
FeatureSourcePanel.addFromFeatureSource    = Añadir Feature Source (almacén :
                                           fenómeno)
FeatureSourcePanel.featureSource           = Feature Source
FeatureSourcePanel.chooseLayer             = Escoger nueva capa
FeatureSourcePanel.chooseFeatureSource     = Escoger nueva capa desde origen del
fenómeno
FeatureSourcePanel.up                      = Mover capa arriba
FeatureSourcePanel.down                    = Mover capa abajo

#pongo los títulos de la tabla en español
FeatureSourcePanel.th.position             = posición
FeatureSourcePanel.th.element              = elemento
FeatureSourcePanel.th.filterCQL            = filtroCQL
FeatureSourcePanel.th.remove               = eliminar

#PANEL DE SELECCIÓN DE STORE
StoreChoicesPanel.select                   = Seleccionar
StoreChoicesPanel.th.DataType              = Tipo de dato
StoreChoicesPanel.th.StoreName             = Nombre almacén
StoreChoicesPanel.th.Workspace             = Espacio de trabajo
StoreChoicesPanel.th.Type                  = Tipo
StoreChoicesPanel.th.Enabled               = Habilitado
StoreChoicesPanel.th.Action                = Acción

#PANEL DE SELECCIÓN DE GRIDSET
GridsetChoicesPanel.select                 =Seleccionar
GridsetChociesPanel.th.Action              = Acción

#PANEL DE VISUALIZACIÓN DEL STORE ESCOGIDO
StoreStatsPanel.selectStore                = Seleccionar Almacén de Datos
StoreStatsPanel.title          = Configuración del almacenamiento de las estadísticas
StoreStatsPanel.chooseStore                = Escoger almacén para las estadísticas
```

```
StoreStatsPanel.advice                    = Guardar configuración para crear acceso
correcto a la base de datos. La base de datos utilizada para almacenar estadísticas y
trabajos de prefecth será la última guardada correctamente.
#pongo los títulos de la tabla en español
StoreStatsPanel.th.DataType               = Tipo de dato
StoreStatsPanel.th.StoreName              = Nombre almacén
StoreStatsPanel.th.Workspace              = Espacio de trabajo
StoreStatsPanel.th.Type                   = Tipo
StoreStatsPanel.th.Enabled                = Habilitado

#PANEL DE CONFIGURACIÓN DE GRIDSET
PrefetchJobConfigurationPanel.addJob      = Añadir trabajo
PrefetchJobConfigurationPanel.prefetchingJobConfiguration   = Configuración de
                                                            Trabajos de Prefetch

#pongo los títulos de la tabla en español
PrefetchJobConfigurationPanel.th.JobName          = Nombre Trabajo
PrefetchJobConfigurationPanel.th.BoundingBox      = Encuadre
PrefetchJobConfigurationPanel.th.Resolution       = Resolución
PrefetchJobConfigurationPanel.th.StartDelay       = Retardo inicial
PrefetchJobConfigurationPanel.th.RepeatCount      = Repetición
PrefetchJobConfigurationPanel.th.Enable           = Habilitar
PrefetchJobConfigurationPanel.th.Action           = Acción
PrefetchJobConfigurationPanel.th.Unit             = Unidad
PrefetchJobConfigurationPanel.th.Format           = Formato
PrefetchJobConfigurationPanel.th.RepeatInterval   = Intervalo de repetición
PrefetchJobConfigurationPanel.repeatAdvice        = 0 para repetir indefinidamente


#PANEL DE TAREAS EJECUÁNDOSE
TaskPanel.title             = Tareas
TaskPanel.taskPanel         = Lista de tareas actualmente en ejecución
TaskPanel.kill              = Parar tarea
TaskPanel.refreshList       = Actualizar lista

#pongo los títulos de la tabla en español
TaskPanel.th.Layer          = Capa
TaskPanel.th.Type           = Tipo
TaskPanel.th.Completed      = Completado
TaskPanel.th.Time           = Tiempo
TaskPanel.th.Action         = Acción
TaskPanel.th.Format         = Formato
TaskPanel.purge             = Purgar BD
TaskPanel.remove            = Eliminar
TaskPanel.kill              = Parar

#SUBPANEL PARA ELEGIR DESDE DATA STORES
#Mensajes del panel
DataStoreListPanel.clickOnTheLayerYouWishToConfigure     = Pulsa en el feature que
                                                         quieres elegir
DataStoreListPanel.listOfResourcesContained              = Aquí se muestra una lista
                                          de los recursos contenidos en el almacén

#Link para escoger
DataStoreListPanel.select          = Seleccionar

#Titulos tabla
DataStoreListPanel.th.name         = Nombre feature
DataStoreListPanel.th.action       = Acción

#Mensaje selección de dropdown
addAFeatureFrom                    = Añadir feature desde

#EDITOR TAB CAPA/GRUPO CAPAS PARA SEEDER CACHING
#titulo de la página dentro del tab
GeoServerSeederLayerEditor.title    = Seeder caching configuration
```

```
#mensajes de aviso
GeoServerSeederLayerEditor.messageAdvice = Solo se puede modificar la configuración
de seeder una vez que se ha guardado una configuración de Tile Caching.
GeoServerSeederLayerEditor.messageAdviceSave = Se deberá guardar al menos una vez
esta configuraciÓn para que realmente se aplique al catálogo. La primera vez que se
accede a la página los valores se generán por defecto, pero no están guardados. Para
comprobar la configuración guardada acuda al directorio data_dir/seeder-config dentro
de la carpeta de instalación de GeoServer

#habilitar la configuración
GeoServerSeederLayerEditor.enableSeederForLayer    = Habilitar Configuración Seeder
                                                     Caching para esta capa
GeoServerSeederLayerEditor.enableSeeder.title      = Escoger esta opción para
                              habilitar la configuración Seeder Caching para esta capa

#chechbox para habilitar/deshabilitar el uso de la configuración seeder
GeoServerSeederLayerEditor.enabled  = Habilitar seeder caching para esta capa/grupo
GeoServerSeederLayerEditor.enabled.title = Si no se selecciona, esta configuración no
                                    será aplicada

#DEFINICIÓN DEL TAB SEEDER CACHING PARA LAYER
SeederCacheOptionsTabPanel.title                   = Seeder Caching
SeederCacheOptionsTabPanel.shortDescription        = Configurar opciones de Seeder
                                                     caching para capa

#DEFINICIÓN DEL PANEL SEEDER CACHING PARA LAYER GROUP
LayerGroupSeederOptionsPanel.title                 = Seeder Caching
LayerGroupSeederOptionsPanel.shortDescription      = Configurar opciones de Seeder
                                                     caching para grupo de capas
```

Código 176. Fichero `GeoServerApplication_es.propierties`

En la definición del nombre del fichero se puede ver cómo definir distintos ficheros `propierties` por idioma. El sufijo *"_es"* a continuación del nombre base permite indicar al navegador que este es el conjunto de propiedades a utilizar si está configurado en español. Por ejemplo, otras notaciones que se utilizarían para otros idiomas seria *"_fr"* para francés, *"_en_us"* para inglés americano, *"_de"* para alemán…

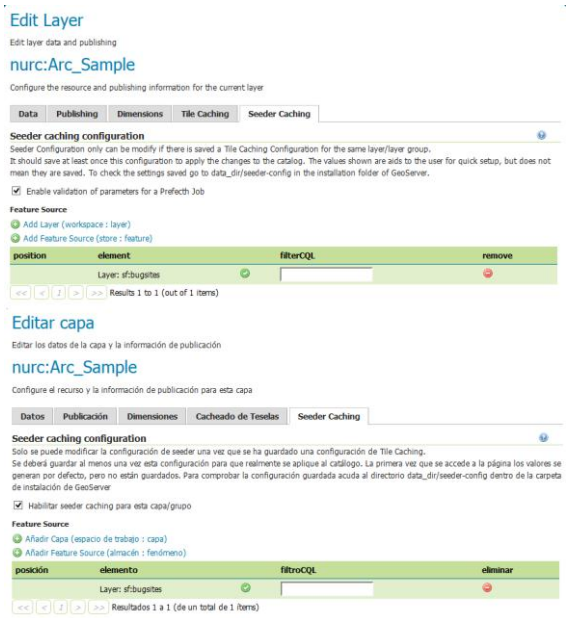

Figura 7. Interfaces en inglés y español

En las imágenes de la Figura 7 puede verse un extracto de la interfaz que se diseñó, cuando el navegador está configurado en español y cuando está configurado en otro idioma o si no existiera la implementación en español.

La interfaz en inglés se encuentra totalmente implementada. Algunos de los mensajes de error o aviso que se puede encontrar el usuario solo están implementados en inglés por lo que el *"Manual de Usuario"* que se presenta en el Capítulo 5, se explica en base a la interfaz en inglés.

4.4. Enlazar la interfaz con los módulos *Seeder*

Con la interfaz ya definida, lo siguiente es enlazarla con los módulos ya existentes de *Seeder*. La interfaz va a permitir al usuario especificar los parámetros a aplicar a la hora de lanzar un trabajo de *seed* sobre la capa / grupo de capas correspondiente, así como lanzarlo. Una característica a destacar es que puede lanzar un trabajo con una configuración de *Seeder* concreta sin necesidad de que esta haya sido guardada, lo cual puede ser ventajoso si el usuario normalmente lanza un trabajo con una configuración concreta pero en un instante determinado quiere lanzar el trabajo con distintos parámetros, ya que no perderá los valores usuales si no guarda los nuevos valores. La única restricción que se le impone al usuario es que debe guardar previamente una configuración en que se especifique la base de datos de la que se quiere recoger las estadísticas para los análisis porque se requiere preparar con anticipación un acceso correcto a esa base de datos. Las restricciones de uso que se consideran más importantes y que el usuario debe conocer para una correcta manipulación de la interfaz han sido incluidas mediante mensajes.

El comportamiento que se ha implementado consiste en que al clicar un enlace denominado *"Seed"* se crea una programación mediante la cual se lanzará el trabajo o tarea según el usuario haya especificado. Para ello se hace uso del servicio *Quartz* que permite crear programadores y asignarles un trabajo a realizar (con los parámetros necesarios para realizar ese trabajo) y un lanzador del trabajo, denominado *trigger* (con el calendario de lanzamiento del trabajo).

Uno de los paneles añadidos a la interfaz de usuario se encargará de mostrar al usuario todos los tipos de trabajos o tareas que se están ejecutando y que están relacionados con esta capa / grupo de capas. No solo se mostrarán los trabajos de tipo *Seeder* añadidos a mayores con este módulo, sino también los que ya se realizan en *GeoServer*. De esta manera el usuario podrá visualizar los trabajos lanzados e información completa sobre el estado en que se encuentran.

El siguiente paso en nuestro trabajo es enlazar la interfaz con los módulos *Seeder-Prefetch*, *Seeder-Stats* y *Seeder-Feature Catalog* para permitir realizar todas las tareas enumeradas previamente (Sección 2.5 Módulos *Seeder*).

Ya se ha definido previamente el uso de elementos de estos módulos dentro de la interfaz, el objetivo de este apartado es definir cuál es el funcionamiento e implementación de esos elementos y como se pueden "comunicar" con la interfaz, en definitiva como podemos enlazar la configuración que el usuario introduce con un trabajo que se lanza justo en el momento en

que el usuario lo desea. Ya se ha explicado el uso del servicio *Quartz* que permite lanzar un determinado trabajo siguiendo un calendario concreto, simplemente a partir de ahora se definirá cómo es el trabajo que se lanza (en la sección 4.3.3.2 ya se explicó cómo generar un calendario con los datos recogidos del usuario). Además se explicará el funcionamiento del módulo de recogida de estadísticas (*Seeder-Stats*), el cual actúa de forma transparente, es decir, no ofrece resultados directos al usuario ni esté sabrá cuando se lanza, a menos que consulte la documentación.

Los módulos que a continuación se detallan, como se indicó en la presentación, no forman parte directa de este trabajo. El objetivo principal de este trabajo, es definir una interfaz *web*, en concreto ampliar la interfaz *web* ya existente de *GeoServer* para añadirle la funcionalidad que permiten estos módulos. Estos fueron testeados de manera local, inicializados manualmente con valores base para comprobar su correcto funcionamiento, pero para poder usarlos de manera general desde la interfaz *web* que se ha descrito anteriormente, han sido necesarias una serie de modificaciones.

En este apartado se define de manera general la estructura de estos módulos (también se indicó en la sección 2.5), así como las modificaciones principales que se han debido realizar sobre ellos, pero sin entrar en detalle en cómo se implementaron para realizar las tareas de *prefetch* y de recogida de estadísticas. En caso de querer obtener detalles sobre estos conceptos se deberá recurrir a la Tesis Doctoral de Ricardo García.

4.4.1. Módulo *Seeder-Feature Catalog*

Formalmente denominado `gwc-featurecatalog`, este módulo tan solo recoge la definición de objetos `FeatureCatalog`, los cuales son usados para generar un catálogo de *Features Sources* en que se almacenan en forma de mapa con valores clave-valor las distintas colecciones de fenómenos que el usuario ha seleccionado. Yendo al significado concreto que se le da, el conjunto de colecciones de fenómeno a las que el usuario quiere hacer pertenecer una capa concreta. El mapa antes indicado recoge parejas nombre de la colección – colección de fenómenos, esto último especificado mediante los objetos `FeatureCollection` de *GeoServer*:

```
public class FeatureCatalog {

	Map<String,FeatureCollection> map = null;
	public Map<String, FeatureCollection> getMap() {
		return map;
	}
	public void setMap(Map<String, FeatureCollection> map) {
		this.map = map;
	}
}
```

Código 177. Objeto `FeatureCatalog` del módulo `gwc-featurecatalog`

4.4.2. Módulo *Seeder-Stats*

Formalmente denominado `gwc-stats`, se encarga de almacenar las estadísticas de acceso a una determinada capa. El funcionamiento básico de este módulo consiste en que al inicializarse permite crear para cada una de las capas que tengan definida una base de datos en la configuración *Seeder*, la conexión con ella. Por otro lado, a cada capa definida en la aplicación, esté o no publicada, añade un escuchador `TileStatsListener` que será el que se encargue de manejar los eventos cada vez que la correspondiente capa esté siendo accedida, es decir, será en el que se encargue de lanzar las tareas oportunas para almacenar las estadísticas de acceso en la base de datos, siempre y cuando la capa tenga configurado un acceso a base de datos, para lo cual deberá estar publicada y haberse guardado una configuración *Seeder* con el parámetro identificador de base de datos.

La información que se almacena se compone de:

- Nombre de la capa.
- *GridSet* con el que se está visualizando la capa.
- Formato de imagen con que se está visualizando. El usuario debe tener en cuenta que cuando se visualizan capas mediante el visor de *GeoServer*, estas se guardan en el directorio de datos de instalación de la aplicación, el formato de imagen hace referencia al formato en que se guardan estas teselas en la máquina del usuario.
- Identificador de las teselas visualizadas. Si el usuario está visualizando la capa con mucha resolución, no todas las teselas entrarán en pantalla, y por ello no todas estarán siendo accedidas. Lo mismo ocurre si la capa a visualizar tiene un tamaño relativamente grande.
- Resolución de vista de la capa.
- Fechas de último acceso y última modificación.
- Geometría de la tesela.

4.4.2.1. Enlazado *Seeder-Stats* con Interfaz *Web*

Para poder enlazar el módulo de recogida de estadísticas `gwc-stats` y la interfaz *web*, que se ha presentado en el Capítulo 4, se ha definido un objeto, realmente una interfaz, que actúa a modo de "puente" entre ambos:

```
public interface StatsMediator {

	public String getLayerStoreId(String layerId);

	public boolean containsLayer(String layerId);

	public boolean containsStoreId(CatalogInfo info);

	public void createDataBase(String storeId, TileLayer layer);

	public boolean validateStore(String storeId);

	public Map<String, IndexDAO> getIndexMap();

}
```

Código 178. Objeto `StatsMediator`

Esta interfaz recoge, por supuesto sin implementación, los métodos del objeto central `Seeder` de la interfaz *web*, que se necesitará usar el módulo de estadísticas. Los métodos que aquí se enumeran ya fueron descritos anteriormente, porque también se necesitan en la interfaz *web*, por lo que no se entrará en más detalle, simplemente se recordará su función. Si se analiza en detalle la definición del objeto `Seeder`, este se presenta como una implementación de esta interfaz.

4.4.2.2. Inicialización del módulo

Este módulo se inicializa junto con la aplicación, para lo cual se debe programar mediante *Spring*:

```
<bean id="statsStore" class="org.uva.idelab.geowebcache.stats.StatsStore"
       init-method="startUp">
       <property name="tileLayerDispatcher" ref="gwcTLDispatcher" />
       <property name="enabled" value="true" />
       <property name="mediator" ref="seederFacade" />
       <property name="catalog" ref="catalog" />

</bean>
```

Código 179. Inicialización del módulo de estadísticas mediante *Spring*

El objeto `StatsStore` actuará de elemento central en este módulo. Los atributos que aquí se definen permiten asociarle elementos ya inicializados mediante *GeoServer*. Realmente se inicializan simultáneamente, pero cuando *Spring* ve que para inicializar un objeto requiere de otro, primero inicializa los objetos sin dependencias y después los que de ellos dependen. No se pueden crear dependencias circulares, porque si no, la aplicación no se iniciaría nunca. El parámetro `enabled` permite, simplemente, habilitar el módulo. Se puede apreciar que al inicializar el objeto `StatsStore` se debe recurrir al método `startUp()` y los atributos necesarios se inicializan mediante métodos `set()`.

Con respecto a la implementación original que tenía este objeto antes de adaptarlo para su uso mediante interfaz de usuario, se han realizado unas ligeras modificaciones que han tenido por objetivo:

- Permitir enlazarlo con la interfaz *web*, para lo cual se define como propiedad del objeto el elemento que actúa como puente entre los dos módulos.
- Asociar a este módulo un catálogo de eventos para manejar las ocasiones en que se añade una nueva capa con el objetivo de crear la conexión con la base de datos correspondiente. En principio se podría haber llevado a cabo esta tarea en el escuchador de eventos de la interfaz *web*, pero por similitud de conceptos se decidió realizarlo en este módulo.
- Permitir inicializar correctamente el módulo para adaptarlo a las nuevas necesidades. Para ello se ha modificado el método `startUp()` presentando ahora la siguiente estructura:

```
public void startUp() {

       catalogLayerEventListener = new
                     CatalogLayerEventListenerForStats(this.mediator,
              catalog);
```

```
            this.catalog.addListener(catalogLayerEventListener);

            if (enabled) {
                  Iterable<TileLayer> allLayers =
                              tileLayerDispatcher.getLayerList();
                  this.tileStatsListener = new TileStatsListener();
                  tileStatsListener.setMediator(mediator);
                  for (TileLayer layer : allLayers) {
                        layer.addLayerListener(tileStatsListener);
                        String storeId =
                              this.mediator.getLayerStoreId(layer.getId());
                        if (storeId != null) {
                              if (mediator.validateStore(storeId)) {
                                          mediator.createDataBase(storeId,
            layer);
                              }

                        }
                  }
            }
      }
```

Código 180. Método `startUp()` del objeto `StatsStore`

Como ya se ha explicado brevemente, crea un nuevo escuchador de eventos del catálogo de tipo `CatalogLayerEventListenerForStats`, el cual que se explicará posteriormente, y lo añade al catálogo de escuchadores de *GeoServer*, por ello se ha debido añadir este, cómo parámetro del objeto `StatsStore`.

A continuación, siempre y cuando el módulo este habilitado, que lo estará por defecto, obtiene todas las capas (publicadas o no) de *GeoServer* y les añade un nuevo escuchador, en este caso el *listener* `tileStatsListener()` que se mencionó anteriormente y que se encargará de lanzar las tareas que permiten almacenar las estadísticas de acceso según se va visualizando una capa. Se verá después porque se debe pasar la referencia del objeto mediador, que a su vez hace referencia al objeto central `Seeder` de la interfaz *web*.

Una vez añadido el escuchador, se comprueba para cada capa si tiene asociado un identificador de base de datos, es decir, se accede al catálogo *Seeder* en que se almacenan todas las capas para las que se guardó una configuración *Seeder* junto con esta configuración. Se accede a la configuración y se obtiene el parámetro identificador de *store*, que apunta a un almacén concreto de los definidos en la aplicación. Si todo es correcto, este identificador tendrá un valor concreto, así que el siguiente paso es comprobar que se corresponde con un identificador de base de datos válida teniendo en cuenta todas las consideraciones que se han indicado en la sección correspondiente a la base de datos: es de tipo *JDBC* y dentro de este tipo, *H2* o *PostGIS* (el método `validateStore()` fue explicado en la Sección 4.3.2.3). Si esta comprobación también es correcta se indica al objeto `Seeder` que cree la conexión para esta base de datos (en método `createDataBase()` fue explicado en la Sección 4.3.2.3). Entonces aquí se aprecia la necesidad del objeto `StatsMediator` para acceder al objeto `Seeder`, no solo por la implementación concreta de los métodos que puede ahorrar líneas de código, si no por el acceso a los datos concretos que esté manejando el objeto `Seeder`.

Se han definidos dos nuevos escuchadores o *Listeners*. El primero de ellos es `CatalogLayerEventListenerForStats`:

```
public class CatalogLayerEventListenerForStats implements CatalogListener {

	private static Logger log =
		Logging.getLogger(CatalogLayerEventListenerForStats.class);

	private final StatsMediator seeder;

	private final Catalog catalog;

	public CatalogLayerEventListenerForStats(final StatsMediator seeder,
	Catalog catalog) {
		this.seeder = seeder;
		this.catalog = catalog;
	}

	public void handleAddEvent(CatalogAddEvent event) throws
	CatalogException {

		CatalogInfo obj = event.getSource();

		if (!(obj instanceof LayerInfo || obj instanceof
		LayerGroupInfo)) {
			return;
		}
		if (GWC.get().getTileLayer(obj) == null) {
			return;
		} else if (!seeder.containsLayer(obj.getId())) {
			return;
		} else if (!seeder.containsStoreId(obj)) {
			return;
		} else {
			String storeId = seeder.getLayerStoreId(obj.getId());
			seeder.createDataBase(storeId,
				GWC.get().getTileLayer(obj));
		}

	}

	public void handleModifyEvent(CatalogModifyEvent event) throws
	CatalogException {
		return;
	}

	public void handlePostModifyEvent(CatalogPostModifyEvent event) throws
	CatalogException {
		return;
	}

	public void handleRemoveEvent(CatalogRemoveEvent event) throws
	CatalogException {
		return;
	}

	public void reloaded() {
		return;
	}

}
```

Código 181. Objeto `CatalogLayerEventListenerForStats`

Cada vez que se añade una nueva capa al catálogo de capas de *GeoServer* se lanza el método `handleAddEvent()`. Este, en primer lugar comprueba que existe, para la capa añadida, *tile layer* y *seeder layer*, es decir, objetos asociados a las configuraciones *Tile Caching* y *Seeder*

Caching. Además se comprueba que se ha indicado una base de datos. En principio, si el usuario sigue los pasos definidos en la interfaz *web*, estas tres comprobaciones serías exitosas siempre, pero por precaución ante posibles cambios no contemplados, se sigue esta secuencia de validaciones. Superadas todas las comprobaciones se inicializa la conexión con la base de datos (método `createDataBase()` Sección 4.3.2.3). El método `containsStoreId()` permite comprobar si el objeto `seederLayerInfo` asociado a la capa/grupo de capas tiene asociado un identificador de almacén de datos, o lo que es lo mismo, si el usuario ha seleccionado una base de datos:

```
	public boolean containsStoreId(CatalogInfo info) {
		boolean exists = false;
		GeoServerSeederLayerInfo seederLayerInfo = null;
		if (info instanceof LayerInfo) {
			LayerInfo layerInfo = (LayerInfo) info;
			seederLayerInfo = getSeederLayerInfo(layerInfo);
		} else if (info instanceof LayerGroupInfo) {
			LayerGroupInfo layerGroupInfo = (LayerGroupInfo) info;
			seederLayerInfo = getSeederLayerGroupInfo(layerGroupInfo);
		}

		if (seederLayerInfo != null) {
			String storeId = seederLayerInfo.getStore();
			if (storeId != null) {
				exists = true;
			}
		}
		return exists;

	}
```

Código 182. Método `containsStoreId()` del objeto `Seeder`

El segundo escuchador es `TileStatsListener`:

```
	public class TileStatsListener implements TileLayerListener {

		private static Log log = LogFactory.getLog(TileStatsListener.class);

		IndexDAO index;
		String layerName;
		StatsMediator mediator;

		public IndexDAO getIndex() {
			return index;
		}

		public void setIndex(IndexDAO index) {
			this.index = index;
		}

		public TileStatsListener() {

		}

		public void tileRequested(TileLayer layer, ConveyorTile tile) {

			Map<String, IndexDAO> indexMap = mediator.getIndexMap();
			if (indexMap.containsKey(layer.getId())) {
				index = indexMap.get(layer.getId());
				log.warn("tileRequested for layer: " + layer.getName());
				index.hitTile(tile);
			}
```

```
		}

		public String getLayerName() {
			return layerName;
		}

		public void setLayerName(String layerName) {
			this.layerName = layerName;
		}

		public StatsMediator getMediator() {
			return mediator;
		}

		public void setMediator(StatsMediator mediator) {
			this.mediator = mediator;
		}

	}
```

Código 183. Objeto `TileStatsListener`

De él, lo más interesante es el método `tileRequested()`. Cada vez que una tesela de una capa es visualizada, si esta capa está asociada a este *listener* se ejecuta este método. Recordar que todas las capas definidas en *GeoServer* (publicadas o no) se asocian a este escuchador al inicializar la aplicación mediante el método `startUp()` del objeto `StatsStore`. El método recibirá como parámetros el objeto `TileLayer` asociado a la capa visualizada y el objeto `ConveyorTile` asociado a la tesela.

Una vez ejecutado el método, en primer lugar mediante el mediador `StatsMediator` accede al objeto `Seeder` para obtener de él el mapa de registro de conexiones a bases de datos. En concreto buscará en este mapa si existe una conexión para la capa que se está visualizando. En caso afirmativo obtiene esa conexión, el elemento `index` que se obtiene se analizará más adelante, pero permite lanzar operaciones sobre la base de datos. En este caso concreto se lanza el método `hitTile()` con el que se procede a añadir o actualizar la información de acceso relativa a esta tesela.

4.4.2.3. Definición de índices (*index*) de bases de datos

Los índices o *index* de bases de datos se entienden como los objetos *Java* en que se definen las conexiones y operaciones que se lanzarán sobre la correspondiente base de datos. Se ha visto reiterativamente que el usuario escoge una base de datos en la que por un lado quiere recoger las estadísticas de acceso a la capa/grupo de capas correspondiente y por otro indica de donde extraer las estadísticas para proceder con los trabajos de *prefetch*. Cuando el usuario guarda esta configuración se llama a un método `createDataBase()` en que tras verificar que la base de datos (o genéricamente almacén de datos) es del tipo adecuado (*H2* o *PostGIS*) procede a crear un índice mediante el objeto `PostgresIndexDAO` o `H2IndexDao`, en función del tipo de base de datos seleccionada.

Estos dos son los objetos en que se define la implementación concreta que permite acceder a ese tipo de base de datos de manera correcta. Cada base de datos utiliza distintas sentencias para poder realizar las tareas básicas de inserción, modificación o borrado de entradas, por lo que la principal diferencia entre estos dos objetos, será la creación de estas sentencias.

Principalmente tienen la misma estructura, sobre todo, porque son implementación de una misma clase abstracta. La estructura exacta de esta jerarquía de objetos comienza en una interfaz denominada `IndexDAO`, que ya se ha visto nombrada en alguna ocasión anterior, en la que se definen todos los métodos que deberán contener los objetos que se utilizan para implementar conexiones a una base de datos. La clase abstracta `AbstractIndexDAO` implementa la interfaz anterior. Se encarga de implementar todos aquellos métodos que van a compartir todos los objetos que de ella hereden, es decir, los objetos `H2IndexDAO` y `PostgresIndexDao` extienden esta clase abstracta y por tanto deben implementar todos los métodos de la interfaz `IndexDAO`. Todos aquellos que tendrían la misma implementación en ambos objetos, se implementan en la clase abstracta `AbstractIndexDAO`. Aquellos que difieren deben implementarse en el objeto concreto.

Dentro de la gran cantidad de métodos que podemos encontrar en estos objetos, se presentan aquellos que se han enumerado en alguna ocasión o han debido sufrir alguna modificación para su uso a través de la interfaz de usuario.

Cuando el usuario seleccione una base de datos, sobre ella se **creará una nueva tabla** denominada `idelabstats` (tal y como se configura en el objeto `AbstractIndexDAO`) en la que se irá almacenando las estadísticas. El primer paso de inicialización del objeto `index` (método `init()`) será crear esa base de datos. Aquí llega el primer conflicto porque la sintaxis de la sentencia de creación será distinta en función de sobre qué tipo de base de datos se esté realizando:

- En *H2* (`H2IndexDAO`):

```
public void init() throws SQLException {


        String sql = "CREATE TABLE IF NOT EXISTS \"" + table
        + "\" ( \"id\" bigint(20) NOT NULL auto_increment PRIMARY KEY, "
        + "\"layer\" varchar(25) not null, \"gridset\" varchar(25) "
        + "not null, \"blob_format\" varchar(25) not null, \"x\" bigint "
        + "not null, \"y\" bigint not null, \"res\" bigint not null, "
        + "\"hits\" bigint, \"blob_size\" bigint, \"last_access\" bigint,"
        + " \"last_update\" bigint, \"version\" bigint, \"bbox\" geometry, "
        + "UNIQUE KEY(\"layer\", \"gridset\", \"blob_format\", \"x\", "
        + "\"y\", \"res\"));";
        getSimpleJdbcTemplate().update(sql, new HashMap());

        String sql2 = "CALL AddGeometryColumn(null, '" + table
                + "', 'bbox', 4326, 'POLYGON', 2)";
        try {
                Connection conn = getConnection();
                Statement statement = conn.createStatement();
                statement.execute(sql2);
        } catch (Exception e) {
                Log.log(Level.INFO, "Table \"" + table
                        + "\" already exists, not be create", e.getStackTrace());

        }

}
```

Código 184. Método `init()` del objeto `H2IndexDAO`

- En *PostGIS* (`PostgresIndexDAO`):

```
public void init() throws SQLException {


	String sql = "CREATE TABLE \""+ table
	+ "\" ( ID  SERIAL PRIMARY KEY, \"layer\" varchar(25) not null , "
	+ "\"gridset\" varchar(25) not null, \"blob_format\" varchar(25) "
	+ "not null, \"x\" bigint not null, \"y\" bigint not null, \"res\" "
	+ "bigint not null, \"hits\" bigint, \"blob_size\" bigint, "
	+ "\"last_access\" bigint, \"last_update\" bigint, \"version\" "
	+ "bigint, UNIQUE(\"layer\", \"gridset\", \"blob_format\", \"x\""
	+ ", \"y\", \"res\"));";
	Statement statement = null;
	String sql2 = "SELECT AddGeometryColumn('" + table
		+ "', 'bbox', 4326, 'POLYGON', 2 );";
	try {
		getSimpleJdbcTemplate().update(sql, new HashMap());
		Connection conn = getConnection();
		statement = conn.createStatement();
		statement.execute(sql2);

	} catch (DataAccessException e1) {
		log.log(Level.INFO, "Table \"" + table
			+ "\" already exists, not be create");
	}

}
```

Código 185. Método `init()` del objeto `PostgresIndexDAO`

El elemento `table` tiene valor `idelabstats` tal y como se inicializa en `AbstractIndexDAO`. Si al intentar crear la tabla en la base de datos, esta ya existe, lanzará un mensaje de error que no colapsará la aplicación debido a que está controlado mediante una estructura `try-catch`. La diferencia entre los dos tipos de base de datos está básicamente en cómo crear una columna de datos espaciales, requerimiento principal para nuestros propósitos.

En cualquiera de las dos sentencias de creación se aprecia que la clave que permitirá distinguir entradas está compuesta por el nombre de la capa, el *GridSet*, formato de imagen, los parámetros "*x*" e "*y*" de ordenación de las teselas y la resolución.

El método `hitTile()` se lanza cada vez que se el usuario visualiza una tesela de una capa (evento captado el escuchador `TileStatsListener`):

```
public TileStats hitTile(ConveyorTile tile, int weight) {

	long lastaccess = System.currentTimeMillis();

	TileStats ts = null;

	boolean flag = true;

	int numRetries = 0;

	while(flag && numRetries<MAX_RETRIES)
	{
		numRetries++;
		flag = false;

		try {
			ts = getTileStats(tile);
```

```
                        if(ts==null) { // INSERT
                                ts = insert(tile);
                        } else { // UPDATE
                                ts.setNumHits(ts.getNumHits()+weight);
                                ts.setLastAccessTime(lastaccess);
                                ts.setBlobSize(tile.getStorageObject()
                                        .getBlobSize());
                                if(!(update(ts)>0)) {
                                        logger.warn("Update block");
                                        flag = true;
                                }
                        }
                } catch (DataAccessException e) {
                        logger.warn(e);
                        flag = true;
                }

                if(flag) {
                        try {
                                logger.warn("Tile "+tile
                                +" is blocked. Num retries: "+numRetries);
                                Random rand = new
                                        Random(System.currentTimeMillis());
                                Thread.sleep((long)
                                        (1000+1000*rand.nextDouble()));
                        } catch (InterruptedException e1) {
                                e1.printStackTrace();
                        }
                }
        }

        return ts;
    }
```

Código 186. Método `hitTile()` del objeto `abstractIndexDAO`

Este método comprueba si la tesela ya había sido o no visitada previamente, o visto de otra forma, si ya se habían recogido estadísticas de acceso. En caso positivo se procede a actualizar esa información (`update()`), en caso negativo, introduce la nueva información en la base de datos (`insert()`).

El método `update()` permite actualizar la información de acceso a una tesela que ya había sido visitada previamente, se implementa igual para los dos tipos de base de datos:

```
    public int update(TileStats tileStats) {

        String sql = "UPDATE \""+ table +"\" SET \"hits\"=:hits, "
                + "\"last_access\"=:lastAccess, \"last_update\"=:lastUpdate,"
                + " \"version\"=:new_version WHERE \"layer\"=:layer AND "
                + "\"gridset\"=:gridset AND \"blob_format\"=:blobFormat "
                + "AND \"x\"=:x AND \"y\"=:y AND \"res\"=:res AND "
                + "\"version\"=:old_version";

        long lastaccess = System.currentTimeMillis();

        Map<String,Object> parameters = new HashMap<String,Object>();
        parameters.put("layer", tileStats.getLayer());
        parameters.put("gridset", tileStats.getGridSet());
        parameters.put("blobFormat", tileStats.getBlobFormat());
        parameters.put("x", tileStats.getX());
        parameters.put("y", tileStats.getY());
        parameters.put("res", tileStats.getZ());
        parameters.put("hits", tileStats.getNumHits());
```

```
            parameters.put("lastAccess", lastaccess);
            parameters.put("lastUpdate", tileStats.getLastUpdateTime());
            parameters.put("old_version", tileStats.getVersion());
            parameters.put("new_version", tileStats.getVersion()+1);

        return getSimpleJdbcTemplate().update(sql, parameters);
    }
```

Código 187. Método `update()` del objeto `abstractIndexDAO`

El método `insert()` permite añadir nuevas entradas a la base de datos en caso de que nunca se hayan registrado acceso a esta base de datos, o se hayan borrado. De igual manera que para inicializar la tabla, se deberá hacer distinción en función del tipo de base de datos porque la sintaxis será distinta:

- En *H2* (`H2IndexDAO`):

```
    public TileStats insert(ConveyorTile tile) {

            String sql = "INSERT INTO \"" + table
            + "\" (\"id\",\"layer\", \"gridset\", \"blob_format\", "
            + "\"x\", \"y\", \"res\", \"hits\", \"blob_size\", "
            + "\"last_access\", \"last_update\", \"bbox\", \"version\") "
            + " VALUES (null,:layer, :gridset, :blobFormat, :x, :y, "
            + ":res, :hits, :blobSize, :lastAccess, :lastUpdate, :bbox,"
            + " :version)";
            Map<String, Object> parameters = getParametersForTile(tile);
            parameters.put("hits", new Long(1));
            parameters.put("lastAccess", System.currentTimeMillis());
            parameters.put("lastUpdate", System.currentTimeMillis());
            parameters.put("version", 0);

            getSimpleJdbcTemplate().update(sql, parameters);
            return createTileStats(parameters);
    }
```

Código 188. Método `insert()` del objeto `H2IndexDAO`

- En *PostGIS* (`PostgresIndexDAO`).

```
    public TileStats insert(ConveyorTile tile) {

            Map<String, Object> parameters = getParametersForTile(tile);
            String sql = "INSERT INTO \"" + table
            + "\" (\"layer\", \"gridset\", \"blob_format\", \"x\","
            + " \"y\", \"res\", \"hits\", \"blob_size\", \"last_access\","
            + " \"last_update\", BBOX, \"version\") "
            + " VALUES (:layer, :gridset, :blobFormat, :x, :y, :res, "
            + ":hits, :blobSize, :lastAccess, :lastUpdate, GeometryFromText('"
            + parameters.get("bbox") + "',4326), :version)";
            parameters.put("hits", new Long(1));
            parameters.put("lastAccess", System.currentTimeMillis());
            parameters.put("lastUpdate", System.currentTimeMillis());
            parameters.put("version", 0);
            try {
                    // postgres 8
                    getSimpleJdbcTemplate().update(sql, parameters);
            } catch (DataAccessException e) {
                    // postgres 9
                    String sql1 = "INSERT INTO \"" + table
                    + "\" (\"layer\", \"gridset\", \"blob_format\", \"x\", \"y\","
                    + " \"res\", \"hits\", \"blob_size\", \"last_access\", "
                    + "\"last_update\", BBOX, \"version\") "
```

```
                + " VALUES (:layer, :gridset, :blobFormat, :x, :y, :res, "
                + ":hits, :blobSize, :lastAccess, :lastUpdate, "
                + "ST_GeomFromText('"+ parameters.get("bbox") + "',4326),"
                + " :version)";
                getSimpleJdbcTemplate().update(sql1, parameters);

        }
        return createTileStats(parameters);
    }
```

Código 189. Método `insert()` del objeto `PostgreIndexDAO`

En el caso de *PostGIS*, la estructura `try-catch` tiene por objetivo manejar distintas versiones de base de datos *PostGres*, porque incluso dentro de un mismo tipo de base de datos, según va evolucionando la aplicación, se van cambiando conceptos. En este caso concreto se ha implementado la aplicación para las versiones 8 y 9 de *PostGres*, la diferencia fundamental está en cómo insertar los datos geométricos: en la versión 8 se utiliza el método `GeometryFromText()` y en la versión 9, `ST_GeomFromText()`. Son detalles pequeños que pueden hacer fallar la aplicación si no se tienen en cuenta. Claramente, si se utiliza una base de datos, aunque sea *PostGIS*, que no se adapte a estas versiones, con mucha probabilidad fallará, porque de momento no se ha implementado el módulo para manejar las posibles modificaciones que halla en la sintaxis de las sentencias *SQL*. Indicar que *PostGIS* es un tipo especial de base de datos *PostGres*, en concreto, aquel que permite el manejo de datos espaciales.

El método `getTileStats()` permite obtener de la base de datos aquella entrada cuyos parámetros coinciden concretamente con los de la tesela para la que se está verificando si se debe actualizar o insertar información. Existen distintas "versiones" de este método en función de que la búsqueda tenga que ser más concreta o no.

El método `deleteEntry()` se utiliza en la opción de purgar la base de datos y permite borrar un conjunto concreto de estadísticas. Ya se indicó que esta funcionalidad estaba asociada a las tareas de *prefetch* lanzadas por el usuario y tenía por objetivo borrar de la base de datos correspondiente todas aquellas estadísticas (por tanto entradas) que el trabajo concreto usará para generar sus resultados (nombre de capa, *GridSet*, formato, resolución… con que se lanza el trabajo). Por lo tanto se trata de una sentencia de borrado muy concreta que se implementa:

- En *H2* (`H2IndexDAO`):

```
public void deleteEntry(String layerName, String gridset, String format, long
xmax, long xmin, long ymax, long ymin, int res) {

        String sql = "DELETE FROM \"" + table
                + "\" WHERE \"layer\"=:layer AND \"gridset\"=:gridset AND"
                + " \"blob_format\"=:blobFormat AND \"x\">=:xmin AND "
                + "\"x\"<=:xmax AND \"y\">=:ymin AND \"y\"<=:ymax AND "
                + "\"res\"=:res";
        Map<String, Object> parameters = new HashMap<String, Object>();
        parameters.put("layer", layerName);
        parameters.put("gridset", gridset);
        parameters.put("blobFormat", format);

        parameters.put("xmin", xmin);
        parameters.put("xmax", xmax);
```

```
            parameters.put("ymin", ymin);
            parameters.put("ymax", ymax);

            parameters.put("res", res);

            getSimpleJdbcTemplate().update(sql, parameters);

        }
```

Código 190. Método `deleteEntry()` del objeto `H2IndexDAO`

- En *PostGIS* (`PostgresIndexDAO`):

```
    public void deleteEntry(String layerName, String gridset, String format, long
    xmax, long xmin, long ymax, long ymin, int res) {

            String sql = "DELETE FROM \"" + table
                    + "\" WHERE \"layer\"=:layer AND \"gridset\"=:gridset "
                    + "AND \"blob_format\"=:blobFormat AND \"x\">=:xmin AND "
                    + "\"x\"<=:xmax AND \"y\">=:ymin AND \"y\"<=:ymax AND "
                    + "\"res\"=:res";
            Map<String, Object> parameters = new HashMap<String, Object>();
            parameters.put("layer", layerName);
            parameters.put("gridset", gridset);
            parameters.put("blobFormat", format);

            parameters.put("xmin", xmin);
            parameters.put("xmax", xmax);
            parameters.put("ymin", ymin);
            parameters.put("ymax", ymax);

            parameters.put("res", res);

            getSimpleJdbcTemplate().update(sql, parameters);

    }
```

Código 191. Método `deleteEntry()` del objeto `PostgresIndexDAO`

Aunque en principio la implementación de ambos métodos es idéntica, se ha separado en los objetos correspondientes, por si en una futura actualización de los estándares de *SQL* de los distintos tipo de base de datos se modifica la sintaxis requerida para llevar a cabo estas operaciones, como se ha visto ocurre en otros contextos.

4.4.3. Módulo *Seeder-Prefetch*

Formalmente denominado `gwc-prefetch`, se encarga de las tareas de *prefetch*, es decir, a partir de las estadísticas de acceso traer a caché aquellas teselas con mayor probabilidad de ser solicitadas. Dentro de esta funcionalidad se pueden distinguir dos tareas básicas que ya han sido enumeradas previamente:

- ✓ Tareas de análisis: denominaremos tareas de análisis al conjunto de funciones ejecutadas previamente al lanzamiento de la propia tarea de *prefetch*. Este conjunto de funciones tienen por objetivo, a partir de los datos configurados por el usuario y las estadísticas de acceso que también dependen de esos datos de usuario, preparar la lista de teselas con mayor probabilidad de ser de nuevo solicitadas, aplicando una

serie de algoritmos que se obtuvieron como solución en la tesis doctoral de Ricardo García.

- ✓ Tareas de *prefetch*: denominaremos tareas de *prefetch* a aquella funcionalidad que se encarga de almacenar en el ordenador de usuario las teselas con mayor probabilidad de ser solicitadas según le indique el resultado de las tareas de análisis.

El último paso de una tarea de análisis, será lanzar una tarea de *prefetch*.

4.4.3.1. Enlazado *Seeder-Prefetch* con Interfaz *Web*

Para poder enlazar el módulo de tareas de *prefetch* `gwc-prefetch` y la interfaz *web*, que se ha presentado en el Capítulo 4, se ha definido un objeto, realmente una interfaz, que actúa a modo de "puente" entre ambos:

```
public interface JobRegistry {

	public abstract void registerGWCTask(GWCTask gwcTask);

	public abstract List<PrefetchingJob> getPrefetchingJobBeans();

	public abstract List<PrefetchTask> getPrefetchTasks(String layerName);

	public void reinitializePrefetchingJob(String jobName);

	public void unschedulerPrefetchingJob(String jobName);

	public File getBaseDirectory();

	public abstract void registerPrefetchingJob(PrefetchingJob job);

	public abstract void removePrefetchingJob(PrefetchingJob found);

	public abstract PrefetchingJob findPrefetchingJob(String jobName);

}
```

Código 192. Objeto `JobRegistry`

Esta interfaz recoge, por supuesto sin implementación, los métodos del objeto central `Seeder` de la interfaz *web*, que necesitará usar el módulo de *prefetch*. Los métodos que aquí se enumeran se explicarán a medida que se necesiten. Si se analiza en detalle la definición del objeto `Seeder`, este se presenta como una implementación de esta interfaz.

4.4.3.2. Tareas de análisis

Las tareas de análisis se encargan de analizar todos los datos recogidos del usuario junto con las estadísticas de acceso que concuerdan con los datos de usuario y genera una lista ordenada con las teselas que se deben almacenar en la máquina de usuario.

La base sobre la que se plantean las tareas de análisis consiste en definir una clase abstracta que hereden, y por tanto de la que sean subclases, todas aquellas distintas tareas de análisis que se quieran distinguir. Esto quiere decir: en la implementación del módulo que existía con anterioridad a este trabajo, se presentó una tarea de análisis que permitía manejar *Features Source*, por lo tanto solo se distingue un tipo de tareas de análisis. Este trabajo ha optado por presentar una estructura ligeramente más general de tal manera que en un futuro se puedan

aplicar o generar tareas de análisis que manejen otros conceptos. Por ello se ha creado una clase abstracta, `PrefetchingJob`, en la que se reúnen todos los atributos y métodos, que se considera en este momento todas las tareas de análisis compartirían, para después en el objeto correspondiente definir las posibles diferencias que pueda haber entre ellas. En este trabajo se ha generado el objeto `FeatureBasedPrefetchingJob`, que ya se había enumerado en alguna ocasión anterior, en el que se definen las particularidades de las tareas de análisis referidas a *Features*.

De esta forma, para este trabajo se puede considerar que los objetos `PrefetchingJob` y `FeatureBasedPrefetchingJob` son el mismo, porque la única implementación que puede tener el primero, es el segundo. Por ello se presentará la estructura del segundo como si toda la implementación estuviera en él mismo, aunque verdaderamente parte de ella se encuentre en `PrefetchingJob` y la herede.

El objeto `PrefetchingJob` (y por tanto `FeatureBasedPrefetchingJob`) extiende la clase `QuartzJobBean`, es decir hace uso del servicio *Quartz*. Esto significa que este servicio puede lanzarlo siguiendo, por un ejemplo, un calendario de ejecuciones.

Como se vio en la Sección 4.3.3.2, a partir de parte de los datos introducidos por el usuario se programa un calendario de lanzamientos de un determinado objeto *Java*, en concreto la clase *Java* que se especificó en el atributo `jobClass` del objeto `JobDetail` (`FeatureBasedPrefetchingJob.class` en este caso). De esta forma, el servicio *Quartz* se encargará de lanzar este objeto, visto como un conjunto de sentencias a ejecutar, según el calendario que se especificó.

Ahora se analiza como es el trabajo que realiza este objeto:

En primer lugar se definen todos los atributos necesarios para realizar la tarea de análisis:

- ✓ Atributos que se han considerado comunes a todas las tareas de análisis y se heredan de la clase abstracta `PrefetchingJob`. Se definen por categorías:
  - Los siguientes atributos permiten especificar los cuatro estados en que se puede encontrar una tarea de análisis y sobre todo tienen por objetivo poder utilizarlos como parámetro, por ejemplo, para realizar estructuras de control de flujo en el panel de visualización de tareas de la interfaz *web*. De esta manera si se cambia el nombre identificador de un estado, ese cambio se hace efectivo en todo el código.

```
public static final String JOB_STATUS_WAITING = "WAITING";
public static final String JOB_STATUS_RUNNING = "RUNNING";
public static final String JOB_STATUS_ENDED = "ENDED";
public static final String JOB_STATUS_ERROR = "ERROR";
```

Código 193. Definición atributos `PrefetchingJob` I

  - Los siguientes atributos permiten definir el tanto por ciento que se estima que un conjunto de sentencias o bloque de ejecución tardará en ejecutarse con respecto al tiempo total de ejecución de una tarea de análisis.

```
protected static final int INIT_JOB_PERCENT = 1;
protected static final int LAUNCHING_TASK_PERCENT = 1;
protected static final int RASTERIZING_FEATURES_PERCENT = 20;
protected static final int GET_FEATURES_PERCENT = 10;
protected static final int RASTERINZING_TARGET_PERCENT = 4;
protected static final int SOLVER_PERCENT = 50;
protected static final int SORT_PRIORITIES_PERCENT = 4;
protected static final int ASSIGN_PRIORITIES_PERCENT = 10;
```

Código 194. Definición atributos `PrefetchingJob` II

- Parámetros necesarios para lanzar una tarea de análisis concreta y que dependen de los datos definidos por el usuario. Los nombres son directamente intuitivos sobre el concepto al que se refieren y coinciden con los que se configuraron en el mapa de detalle de trabajo del *trigger*.

```
protected String layerName;
protected String gridSetId;
protected String format;
protected TileLayerDispatcher tileLayerDispatcher;
protected FeatureSource tileStatsFeatureSource;
protected ReferencedEnvelope bbox;
protected String status = JOB_STATUS_WAITING;
protected double buffer;
protected int res;
protected boolean reseed;
protected String storeId;
protected String jobName;
```

Código 195. Definición atributos `PrefetchingJob` III

- El siguiente parámetro son permite manejar el tanto por ciento superado de la tarea.

```
protected long step = 0;
```

Código 196. Definición atributos `PrefetchingJob` IV

- El siguiente booleano nos permite comprobar si el usuario ha mandado cancelar la tarea de análisis (valor `true`) y proceder a su cancelación efectiva.

```
volatile boolean cancelRequested = false;
```

Código 197. Definición atributos `PrefetchingJob` V

- El siguiente conjunto de parámetros permite controlar las cuestiones temporales de la tarea, que se requieren para mostrar al usuario estimaciones sobre tiempo transcurrido y tiempo restante de la tarea.

```
protected long initTimeInMSecond;
long actualTimeInMSecond;
long timeSpent;
long totalTime;
long nextFireTimeInMSecond;
long previousFireTimeInMSecond;
```

Código 198. Definición atributos `PrefetchingJob` VI

- Los siguientes tres parámetros permiten especificar los mensajes de estado e información que se muestran al usuario en el panel de visualización de tareas.

```
protected String statusMessage;
protected String infoMessage;
protected String mainMessage;
```

Código 199. Definición atributos PrefetchingJob VII

- Este parámetro permite manejar el número de veces que una tarea de análisis se ha debido cancelar porque ya había otra ejecutándose. Se verá a continuación.

```
protected int errorFires;
```

Código 200. Definición atributos PrefetchingJob VIII

- Otros parámetros necesarios:

```
protected TileBreeder tileBreeder;
protected StorageBroker storageBroker;
protected JobRegistry registry;
```

Código 201. Definición atributos PrefetchingJob IX

✓ Atributos propios para el caso de tareas de análisis sobre *Features* definidos en el objeto FeatureBasedPrefetchingJob:

```
FeatureCatalog featureCatalog;
FeatureRasterizer rasterizer = null;
```

Código 202. Definición atributos FeatureBasedPrefetchingJob

Al contrario de otros objetos, estos que se lanzan mediante *Quartz* se "inicializan" mediante el método executeInternal(), ya que este es el método que se lanzará cada vez que el servicio *Quartz* tenga que "ejecutar" la clase definida. Para el caso de FeatureBasedPrefetchingJob tiene la siguiente estructura:

```
protected void executeInternal(JobExecutionContext context) throws
JobExecutionException {…}
```

Código 203. Método executeInternal() del objeto FeatureBasedPrefetchingJob I

El parámetro que se recibe, context, permite obtener todos los parámetros relacionados con la programación de la tarea, como puede ser el conjunto de parámetros con que se configuró para lanzarla (JobDetail) o el instante temporal en que se lanzará la siguiente vez.

El contenido de este método esta englobado dentro de una estructura try-catch:

```
try {
        …
} catch (JobExecutionException je) {
        updateStatus(context);
        throw new JobExecutionException("Cancelled by the user.");
} catch (Exception e) {

        this.status = JOB_STATUS_ERROR;
        this.setStatusMessage(e.getLocalizedMessage());
        registry.unschedulerPrefetchingJob(jobName);
        e.printStackTrace();

}
```

Código 204. Método executeInternal() del objeto FeatureBasedPrefetchingJob II

Los posibles errores que se manejan mediante las sentencias `catch` son:

✓ El usuario ha cancelado una tarea de análisis en ejecución, pero no ha desprogramado la tarea completa. En ese caso se manda lanzar una excepción propia del servicio *Quartz*, `JobExecutionException()` que se caracteriza por terminar el lanzamiento concreto de la tarea, pero no el resto de lanzamientos programados. El método `updateStatus()` permite actualizar el estado de la tarea de análisis:

```
public void updateStatus(JobExecutionContext context) {
	Date nextFireTime = context.getNextFireTime();

	if (context.getNextFireTime() != null) {

		if (nextFireTime.compareTo(new Date(System.currentTimeMillis()))
		< 0) {
			SimpleTrigger trigger = (SimpleTrigger)
				context.getTrigger();
			long repeatInterval = trigger.getRepeatInterval();
			long nextTime = nextFireTime.getTime()
					+ errorFires * repeatInterval;
			if (trigger.getFinalFireTime() != null && nextTime >
			trigger.getFinalFireTime().getTime()) {
				nextFireTimeInMSecond = (-1);

			} else
				nextFireTimeInMSecond = nextTime;
		} else
			nextFireTimeInMSecond = nextFireTime.getTime();
	}

	previousFireTimeInMSecond = initTimeInMSecond;

	if (nextFireTime == null || nextFireTimeInMSecond == -1) {
		this.status = JOB_STATUS_ENDED;
		this.statusMessage = "Analysis completed for job " + jobName;
		this.timeSpent = this.totalTime;
		this.step = 100;
	} else {
		this.statusMessage = "Next run: " + nextFireTime;
		this.status = JOB_STATUS_WAITING;
	}
}
```

Código 205. Método `updateStatus()` del objeto `FeatureBasedPrefetchingJob`

En este método se juega con los parámetros de programación de lanzamiento de la tarea como son el instante temporal en que se lanzó por última vez o cuando se volverá a lanzar. El principal objetivo que tiene este método es actualizar la información de estado de la tarea, en concreto los instantes de lanzamiento del siguiente análisis, para su visualización en el panel *Task Panel*.

Si existe siguiente estado de lanzamiento se actualiza la información relativa a posterior y siguiente momento en que se lanzará la tarea y se indica que se está esperado el comienzo de un nuevo análisis. En caso contrario, se indica que ya se han completado todas las tareas de análisis programadas.

✓ Ha ocurrido algún error a la hora de ejecutar la tarea y se procede a desprogramarla entera, porque se considera que si algo ha fallado ahora, volverá a fallar de nuevo. Para ello se accede al método del objeto `Seeder` de la interfaz *web* `unschedulerPrefetchingJob()` a

través del "puente" creado. Este se encarga de desprogramar el *scheduler* definido para esta tarea. Recordar los *scheduler* se identifican a través del nombre que se dio al trabajo, en este caso el parámetro `jobName` que introduce el usuario:

```
public void unschedulerPrefetchingJob(String jobName) {
	StdSchedulerFactory factory = new StdSchedulerFactory();
	try {
		Scheduler s = factory.getScheduler();
		s.unscheduleJob(jobName, null);

	} catch (SchedulerException e) {
		logger.warning("Unable to unscheduler job: " + jobName);
	}
}
```

Código 206. Método `unschedulerPrefetchingJob()` del objeto `Seeder`

Dentro de la estructura `try` se define la funcionalidad de la tarea de análisis:

- ✓ Se inicializan los parámetros de la tarea:

```
Date initDate = new Date(System.currentTimeMillis());
initTimeInMSecond = initDate.getTime();
this.status = JOB_STATUS_RUNNING;

configureJob(context);

this.featureCatalog = (FeatureCatalog)
	context.getJobDetail().getJobDataMap().get("featureCatalog");
String typeMessage = "<html><p>ANALYSIS JOB<br/>";
this.mainMessage = " &nbspJob Name : " + jobName
	+ "<br/> &nbspStats Store : " + storeName
	+ "<br/> &nbspLayer: " + layerName
	+ "<br/> &nbspGridSet: " + gridSetId
	+ "<br/> &nbspFormat: " + format
	+ "<br/> &nbspBounding Box: " + bbox
	+ "<br/> &nbspBuffer: " + buffer
	+ "<br/> &nbspResolution: " + res
	+ "<br/> &nbspReseed: " + reseed + "</p>";
String cierreHtml = "</html>";
String initMsg = typeMessage + mainMessage;

this.infoMessage = initMsg + cierreHtml;

String runningMessage = "Running Analysis Job '" + jobName + "'";
this.statusMessage = runningMessage;
```

Código 207. Método `executeInternal()` del objeto `FeatureBasedPrefetchingJob` III

Los mensajes que se mostrarán al usuario se configuran en formato *HTML*. Todos aquellos parámetros que son comunes a distintos tipos de análisis se configuran en el método `configureJob()`, que se encontrará implementado en `PrefetchingJob`:

```
protected void configureJob(JobExecutionContext context) throws IOException,
TransformException, FactoryException {
	JobDetail jd = context.getJobDetail();
	JobDataMap jdm = jd.getJobDataMap();
	this.layerName = jdm.getString("layerId");
	this.gridSetId = jdm.getString("gridSetId");
	this.format = jdm.getString("format");
	this.tileLayerDispatcher = (TileLayerDispatcher)
		jdm.get("tileLayerDispatcher");
```

```
            this.storageBroker = (StorageBroker) jdm.get("storageBroker");
            this.jobName = jdm.getString("jobName");
            ReferencedEnvelope refenv = (ReferencedEnvelope) jdm.get("bbox");
            this.buffer = jdm.getDouble("buffer");
            this.res = jdm.getInt("res");
            this.reseed = jdm.getBoolean("reseed");
            this.tileBreeder = (TileBreeder) jdm.get("tileBreeder");
            this.registry = (JobRegistry) jdm.get("registry");
            this.storeId = jdm.getString("store");
            this.errorFires = 0;
            JDBCDataStore jdbc = (JDBCDataStore) jdm.get("dataStore");
            this.tileStatsFeatureSource =
                    jdbc.getFeatureSource(AbstractIndexDAO.TABLE_NAME);

            GridSetBroker gridSetBroker = GWC.get().getGridSetBroker();
            GridSet gridSet = gridSetBroker.get(this.gridSetId);
            SRS srs = gridSet.getSrs();
            CoordinateReferenceSystem gridSetCrs = CRS.decode("EPSG:"
                    + srs.getNumber(), true);
            this.bbox = refenv.transform(gridSetCrs, true);
    }
```

Código 208. Método `configureJob()` del objeto `PrefetchingJob`

A partir del parámetro `context` se obtiene el detalle del trabajo lanzado y de él el mapa de parejas clave-valor en que se almacenaron los valores necesarios.

✓ El primer paso que se realiza en una tarea de análisis es comprobar que no existe ninguna otra tarea de análisis sobre el mismo trabajo (con mismo `jobName`) ejecutándose, ya que es coherente no ejecutar dos veces en paralelo lo mismo. Así se trata de evitar problemas si el usuario configura intervalos entre tiempos insuficientes para que termine una tarea de análisis:

```
    step = 0;
    if (registry != null) {

            List<PrefetchingJob> jobList = registry.getPrefetchingJobBeans();

            for (int i = 0; i < jobList.size(); i++) {
                PrefetchingJob job = jobList.get(i);
                if (job.getJobName().equals(jobName) &&
                job.getStatus().equals(JOB_STATUS_RUNNING)) {
                        job.setErrorFires(job.getErrorFires() + 1);
                        log.info("Another Analysis Running for layer {"
                                + layerName + "}, gridset {" + gridSetId + "}");
                        throw new JobExecutionException("Another job running");
                }
            }

            registry.reinitializePrefetchingJob(jobName);
            log.info("Starting prefetching job bean for layer {" + layerName
                    + "}, gridset {" + gridSetId + "}");

            registry.registerPrefetchingJob(this);

    }
```

Código 209. Método `executeInternal()` del objeto `FeatureBasedPrefetchingJob` IV

El método `getPrefetchingJobBean()` del objeto `Seeder`, al que se accede a través del puente `registry`, permite obtener la lista de tarea de análisis que se están ejecutado. Sobre esta lista

busco si alguna de sus entradas contiene un trabajo con el mismo nombre del que estoy intentando lanzar y que se encuentre en estado de ejecución (`running`). En caso afirmativo lanza una excepción que es capturada por la primera sentencia `catch` de las mencionadas anteriormente. El método `setErrorFires()` permite aumentar en un unidad el número de veces que se ha intentado lanzar un análisis y ya había otro ejecutándose para poder manejar los tiempos de lanzamiento del siguiente análisis válido, una vez termine el que se está ejecutando, tal y cómo se vio brevemente en el método `updateStatus()`.

En caso contrario, se sigue con la ejecución del objeto. Al iniciar una nueva instancia de objeto, borro cualquier referencia que hubiera al anterior trabajo con el mismo nombre y vuelvo a registrarlo, mediante los métodos del objeto `Seeder` al que se accede mediante el "puente" `registry`:

Reinicialización de la lista de tareas de análisis:

```
public void reinitializePrefetchingJob(String jobName) {
	PrefetchingJob found = findPrefetchingJob(jobName);
	removePrefetchingJob(found);
}
```

Código 210. Método `reinitializePrefetchinJob()` del objeto `Seeder`

En ella buscamos la entrada que se refiere a este mismo trabajo.

```
public PrefetchingJob findPrefetchingJob(String jobName) {
	PrefetchingJob found = null;
	for (int i = 0; i < prefetchingJobList.size(); i++) {
		PrefetchingJob job = prefetchingJobList.get(i);
		if (job.getJobName().equals(jobName)) {
			found = job;
			break;
		}
	}
	return found;
}
```

Código 211. Método `findPrefetchingJob()` del objeto `Seeder`

Y la borramos.

```
public void removePrefetchingJob(PrefetchingJob found) {
	if (found != null)
		prefetchingJobList.remove(found);
}
```

Código 212. Método `removePrefetchingJob()` del objeto `Seeder`

Se vuelve a añadir la nueva repetición a esa lista.

```
public void registerPrefetchingJob(PrefetchingJob job) {
	this.prefetchingJobList.add(job);
}
```

Código 213. Método `registerPrefetchingJob()` del objeto `Seeder`

Esta es la implementación por la que se ha optado a la hora de registrar las tareas de análisis en ejecución: cuando se lanza una nueva tarea de análisis, se borra el registro de la anterior

que permaneció para poder indicarle al usuario cuanto faltaba para lanzar la siguiente tarea de análisis y se vuelve a añadir.

✓ Se actualiza el mensaje de estado:

```
mainMessage = " &nbspJob Name : " + jobName
        + "<br/> &nbspStats Store : " + storeName
        + "<br/> &nbspLayer: " + layerName
        + "<br/> &nbspGridSet: " + gridSetId
        + "<br/> &nbspFormat: " + format
        + "<br/> &nbspBounding Box: " + bbox
        + "<br/> &nbspBuffer: " + buffer
        + "<br/> &nbspResolution: " + res
        + "<br/> &nbspReseed: " + reseed + "</p>";

this.infoMessage = typeMessage + mainMessage + cierreHtml;
```

Código 214. Método `executeInternal()` del objeto `FeatureBasedPrefetchingJob` V

✓ Las siguientes líneas están extraídas directamente del código generado con anterioridad a este trabajo para realizar el cometido concreto de la tarea de análisis:

```
TileLayer tileLayer = null;

try {
        tileLayer = tileLayerDispatcher.getTileLayer(layerName);
} catch (GeoWebCacheException e) {
        log.error("Layer {" + layerName + "} does not exist", e);
        return;
}

GridSubset gridSubset = tileLayer.getGridSubset(gridSetId);

if (gridSubset == null) {
        log.error("Layer {" + layerName + "} does not contain GridSet {"
                + gridSetId + "}");
        return;
}

MimeType mimeType = null;
if (format == null) {
        mimeType = tileLayer.getMimeTypes().get(0);
} else {
        try {
                mimeType = MimeType.createFromFormat(format);
        } catch (MimeException e4) {
                e4.printStackTrace();
        }
}

BoundingBox reqBounds = null;

if (bbox != null) {
        reqBounds = new BoundingBox(bbox.getMinX(), bbox.getMinY(),
                bbox.getMaxX(), bbox.getMaxY());
} else {
        log.warn("No bounding box specified. The whole coverage will be used");
        reqBounds = gridSubset.getCoverageBounds(res);
}

long[] coverageIntersection = gridSubset.getCoverageIntersection(res,
        reqBounds);

long minx = coverageIntersection[0];
```

```
long miny = coverageIntersection[1];
long maxx = coverageIntersection[2];
long maxy = coverageIntersection[3];

long width = maxx - minx + 1;
long height = maxy - miny + 1;

if (width < 1 || height < 1) {
	log.warn("Specified bounding box does not intersect the available área”
		+ “for this layer");
	return;
}

Collection<FeatureCollection> col = featureCatalog.getMap().values();

if (col.size() == 0) {
	log.warn("Empty feature catalog");
	return;
}

step(INIT_JOB_PERCENT);
```

Código 215. Método `executeInternal()` del objeto `FeatureBasedPrefetchingJob` VI

Se puede destacar el uso del método `step()` que permite actualizar el tanto por ciento de ejecución de la tarea en función de la porción que correspondía al trabajo realizado entre la anterior llamada a este función y la actual. Es una manera intuitiva de calcular el progreso.

```
protected void step(float increase) throws JobExecutionException {

	Date actualDate = new Date(System.currentTimeMillis());
	actualTimeInMSecond = actualDate.getTime();
	timeSpent = actualTimeInMSecond - initTimeInMSecond;
	this.step += increase;
	totalTime = timeSpent * 100 / step;

	if (isCancelRequested()) {
		throw new JobExecutionException("Cancelled by the user.");

	}
}
```

Código 216. Método `step()` del objeto `FeatureBasedPrefetchingJob`

Este método además permite actualizar el tiempo restante (una estimación) y comprobar si mientras se completaba el correspondiente bloque de ejecución el usuario ha cancelado la tarea de análisis, para proceder a su cancelación efectiva. Se realiza una estimación del tiempo total que requerirá la tarea mediante una simple regla de tres: sabiendo el tiempo que se ha requerido para ejecutar un determinado tanto por ciento, se puede calcular de forma aproximada cuando tiempo supondrá calcular el 100%.

El siguiente bloque se encarga de generar el *Buffering* y la *Rasterización* de los datos, así como guardar las imágenes resultantes del raster en la máquina del usuario:

```
File baseDir = registry.getBaseDirectory();
File wwwSubDir = new File(baseDir, "www");
File seederSubDir = new File(wwwSubDir, "Seeder");
```

```
String[] nameParts = layerName.split(":");
String ws = nameParts[0];
File wsSubDir = new File(seederSubDir, ws);
String layer = "";
File jobSubDir;
if (nameParts.length == 2) {
	layer = nameParts[1];
	File layerSubDir = new File(wsSubDir, layer);
	jobSubDir = new File(layerSubDir, jobName);
} else
	jobSubDir = new File(wsSubDir, jobName);
if (!jobSubDir.exists()) {
	jobSubDir.mkdirs();
}
//
// Buffering & Rasterization
//
List<Raster> rasterFeaturesList = rasterize(width, height, col,
nameParts, ws, layer, jobSubDir);

this.infoMessage = typeMessage + mainMessage + cierreHtml;

FeatureCollection tileStatsFeatureCollection = null;

try {
	tileStatsFeatureCollection = tileStatsFeatureSource.getFeatures(
	CQL.toFilter("\"res\" = " + res + " and \"layer\" = '"
	+ layerName + "' and gridset = '" + gridSetId
	+ "' and \"blob_format\" = '" + format + "'"));
	step(GET_FEATURES_PERCENT);

} catch (CQLException e) {
	log.error(e);
	return;
} catch (IOException e) {
	log.error(e);
	return;
}
Raster rasterTarget = null;
try {
	File rasterLogImage = new File(jobSubDir, "stats.png");
	rasterTarget = getRaster(calculateBufferedFeatures
		(tileStatsFeatureCollection), "hits", rasterLogImage, null);
	step(RASTERINZING_TARGET_PERCENT);
	String src;
	if (nameParts.length == 2) {
		src = "http://localhost:8080/geoserver/www/Seeder/"
			+ ws + "/" + layer + "/" + jobName + "/stats.png";
	} else {
		src = "http://localhost:8080/geoserver/www/Seeder/"
			+ ws + "/" + jobName + "/stats.png";
	}
	String imageMessage = "<p> Access Stats </p><a href=\"" + src
		+ "\" target=\"new\"><img src=\"" + src
		+ "\" alt=\"Raster\" width=\"150\"/></a>";
	report(imageMessage);

	this.infoMessage = typeMessage + mainMessage + cierreHtml;

} catch (FeatureRasterizerException e) {
	log.error("Error while rasterizing target");
	return;
}

List<Tile> prefetchingTileList = new ArrayList<Tile>();
```

Código 217. Método `executeInternal()` del objeto `FeatureBasedPrefetchingJob` VII

El método `rasterize()` ha sido modificado para permitir mostrar al usuario las imágenes obtenidas como parte del mensaje de información del panel de visualización de tareas. Para ello se ha considerado que solo el directorio *www* del directorio de datos de la aplicación *GeoServer* es accesible desde la *web*.

```
protected List<Raster> rasterize(long width, long height,
Collection<FeatureCollection> col, String[] nameParts, String ws, String
layer, File jobSubDir) throws Exception {
	rasterizer = new FeatureRasterizer((int) height, (int) width);

	List<Raster> rasterFeaturesList = new ArrayList<Raster>();

	for (FeatureCollection featureCol : col) {
		try {
			CoordinateReferenceSystem targetCRS =
				bbox.getCoordinateReferenceSystem();
			FeatureType schema = featureCol.getSchema();
			CoordinateReferenceSystem featureCRS =
				schema.getCoordinateReferenceSystem();

			Hints.putSystemDefault(Hints.FORCE_LONGITUDE_FIRST_AXIS_O
				RDER, Boolean.TRUE);
			MathTransform transform =
			CRS.findMathTransform(featureCRS, targetCRS, true);

			File featureLogImage = new File(jobSubDir,
			featureCol.getSchema().getName().getLocalPart()
			+ ".png");

			FeatureCollection buffered =
			calculateBufferedFeatures(featureCol);

			rasterFeaturesList.add(getRaster(buffered, "1",
			featureLogImage, transform));

			String src;
			if (nameParts.length == 2) {
				src =
				"http://localhost:8080/geoserver/www/Seeder/"
				+ ws + "/" + layer + "/" + jobName + "/" +
				                +
				featureCol.getSchema().getName().getLocalPart()
				+ ".png";
			} else {
				src =
				"http://localhost:8080/geoserver/www/Seeder/"
				+ ws + "/" + jobName + "/" +
				featureCol.getSchema().getName().getLocalPart()
				+ ".png";
			}

			String imageMessage = "<p>" +
				featureCol.getSchema().getName().getLocalPart()
				+ "</p><a href=\"" + src
				+ "\" target=\"new\"><img src=\"" + src
				+ "\" width=\"150\"/></a><br/><br/>";

			report(imageMessage);

		} catch  (FeatureRasterizerException e) {
			log.error("Error while rasterizing feature "
			+ featureCol.getID());
		}
		step(RASTERIZING_FEATURES_PERCENT / col.size());
```

```
            }
            return rasterFeaturesList;
      }
```

Código 218. Método `rasterize()` del objeto `FeatureBasedPrefetchingJob`

A continuación se computa la solución *OLS*, se asignan prioridades a las teselas y se orden de mayor a menor prioridad, para ser traídas a caché en este orden.

```
      //
      // Compute OLS solution
      //
      Raster y_hat_raster = computePredictionModel(rasterTarget,
            rasterFeaturesList);

      //
      // Assign priorities to tiles and order in descending order of
      // priority
      //
      int totalTiles = y_hat_raster.getWidth() * y_hat_raster.getHeight();
      float priorStep = ASSIGN_PRIORITIES_PERCENT / totalTiles;

      for (int i = 0; i < y_hat_raster.getWidth(); i++) {
            for (int j = 0; j < y_hat_raster.getHeight(); j++) {
                  prefetchingTileList.add(new Tile(minx + i, miny + j, res,
                  y_hat_raster.getSampleDouble(i, j, 0)));
                  step(priorStep);
            }
      }
      Collections.sort(prefetchingTileList, Collections.reverseOrder());
      step(SORT_PRIORITIES_PERCENT);
```

Código 219. Método `executeInternal()` del objeto `FeatureBasedPrefetchingJob` VIII

Llegados a este punto queda lanzar la tarea de *prefetch* (`PrefetchTask`). Se aprovecha para registrarla en el objeto `Seeder` y así poder acceder a ella desde la interfaz *web*. Por último se actualiza el estado de la tarea de análisis mediante el método `updateStatus()`.

```
      //
      // Create and launch prefetching task (using the task support of
      // GWC, so
      // this can be queries through the rest interface)
      //

      boolean doFilterUpdate = false;
      GWCTask gwcTask = new PrefetchTask(storageBroker, tileLayer,
            gridSubset, mimeType, reseed, doFilterUpdate,
            prefetchingTileList, jobName, mainMessage, storeId);

      tileBreeder.dispatchTasks(new GWCTask[] { gwcTask });
      registry.registerGWCTask(gwcTask);
      step(LAUNCHING_TASK_PERCENT);

      updateStatus(context);
```

Código 220. Método `executeInternal()` del objeto `FeatureBasedPrefetchingJob` IX

El registro de la tarea `PrefetchTask` se realiza mediante el método `registerGWCTask()` del objeto `Seeder` de la interfaz *web*:

```
public void registerGWCTask(GWCTask gwcTask) {
                if (!this.GWCTasksRegistry.contains(gwcTask))
                this.GWCTasksRegistry.add(gwcTask);
}
```

Código 221. Método `registerGWCTask()` del objeto `Seeder`

No se entra en detalle sobre los conceptos de las tareas de análisis que aquí se hayan podido presentar ya que su definición no es el objetivo de este trabajo. Para obtener la implementación completa de todos los métodos de los objetos `PrefetchingJob` y `FeatureBasedPrefetchingJob`, junto con comentarios se deberá recurrir al código fuente de este trabajo.

4.4.3.3. Tareas de *prefetch*

Las tareas de *prefetch* se definen en el objeto `PrefetchTask` que se añade como un subtipo de las tareas `GWCTask` de *GWC*. La implementación de este objeto se hereda directamente del código disponible antes de este trabajo.

Las únicas modificaciones que se han realizado han tenido por objetivo definir los mensajes de estado e información que se deben mostrar al usuario en el panel de visualización para facilitar su obtención.

En los dos métodos que se presentan a continuación definen la funcionalidad principal de las tareas de *prefetch*.

```
public PrefetchTask(StorageBroker sb, TileLayer tl, GridSubset gridSubset,
MimeType mimeType, boolean reseed, boolean doFilterUpdate, List<Tile>
prefetchingTileList, String jobName, String infoMessage, String storeId) {

        this.storageBroker = sb;
        this.tl = tl;
        this.reseed = reseed;
        this.doFilterUpdate = doFilterUpdate;
        this.prefetchingTileList = prefetchingTileList;
        this.gridSubset = gridSubset;
        this.mimeType = mimeType;
        this.jobName = jobName;
        this.storeId = storeId;

        this.statusMessage = "Running Prefetch Task for Job '"
                + jobName + "'.";

        String typeMessage = "<html><p>PREFETCH TASK<br/>";
        this.infoMessage = typeMessage + infoMessage;
        tileFailureRetryCount = 0;
        tileFailureRetryWaitTime = 100;
        totalFailuresBeforeAborting = 10000;
        sharedFailureCounter = new AtomicLong();

        if (reseed) {
                super.parsedType = GWCTask.TYPE.RESEED;
        } else {
                super.parsedType = GWCTask.TYPE.SEED;
        }
        super.layerName = tl.getName();
```

```
        super.state = GWCTask.STATE.READY;
    }

    protected void doActionInternal() throws GeoWebCacheException,
    InterruptedException {

        super.state = GWCTask.STATE.RUNNING;

        // Lower the priority of the thread
        Thread.currentThread().setPriority((java.lang.Thread.NORM_PRIORITY +
                java.lang.Thread.MIN_PRIORITY) / 2);

        checkInterrupted();

        // approximate thread creation time
        final long START_TIME = System.currentTimeMillis();

        final String layerName = tl.getName();
        log.info(Thread.currentThread().getName() + " begins seeding layer : "
                + layerName);

        checkInterrupted();

        super.tilesTotal = prefetchingTileList.size();

        final int metaTilingFactorX = tl.getMetaTilingFactors()[0];
        final int metaTilingFactorY = tl.getMetaTilingFactors()[1];

        final boolean tryCache = !reseed;

        checkInterrupted();

        this.tilesDone = 0;
        long seedCalls = 0;

        for (int i = 0; i < prefetchingTileList.size(); i++) {
                long[] gridLoc = prefetchingTileList.get(i).getGridLocation();
                if (gridLoc != null && this.terminate == false) {
                        checkInterrupted();
                        Map<String, String> fullParameters = null;

                        ConveyorTile tile = new ConveyorTile(storageBroker,
                        layerName, gridSubset.getName(), gridLoc, mimeType,
                        fullParameters, null, null);

                        tile. setTileLayer(tl);

                        for (int fetchAttempt = 0; fetchAttempt <=
                        tileFailureRetryCount; fetchAttempt++) {

                                try {
                                        checkInterrupted();
                                        tl.seedTile(tile, tryCache);

                                        break;// success, let it go
                                } catch (Exception e) {
                                        if (tileFailureRetryCount == 0) {
                                                if (e instanceof
                                                GeoWebCacheException) {
                                                throw (GeoWebCacheException) e;
                                                }
                                                throw new GeoWebCacheException(e);
                                        }

                                        long sharedFailureCount =
```

```
                                sharedFailureCounter.incrementAndGet();
                                if (sharedFailureCount >=
                                totalFailuresBeforeAborting) {
                                        log.info("Aborting seed thread "
                                        + Thread.currentThread().getName()
                        + ". Error count reached configured maximum of "
                                        + totalFailuresBeforeAborting);
                                        super.state = GWCTask.STATE.DEAD;
                                        return;
                                }
                                String logMsg = "Seed failed at "
                                        + tile.toString() + " after "
                                        + (fetchAttempt + 1) + " of "
                                        + (tileFailureRetryCount + 1)
                                        + " attempts.";
                                if (fetchAttempt < tileFailureRetryCount) {
                                        log.debug(logMsg);
                                        if (tileFailureRetryWaitTime > 0) {
                                                log.trace("Waiting "
                                                + tileFailureRetryWaitTime
                                                + " before trying again");

                                                Thread.sleep
                                                (tileFailureRetryCount);
                                        }
                                } else {
                                        log.info(logMsg + " Skipping and"
                                        + " continuing with next "
                                        + "tile. Original error: "
                                        + e.getMessage());
                                }
                        }
                }

                if (log.isTraceEnabled()) {
                        log.trace(Thread.currentThread().getName()
                        + " seeded " + Arrays.toString(gridLoc));
                }


                final long tilesCompletedByThisThread = seedCalls;

                updateStatusInfo(tl, tilesCompletedByThisThread,
                        START_TIME);

                checkInterrupted();
                seedCalls++;
        }
}

if (this.terminate) {
        log.info("Job on " + Thread.currentThread().getName()
                + " was terminated after " + this.tilesDone + " tiles");
} else {
        log.info(Thread.currentThread().getName()
        + " completed (re)seeding layer " + layerName + " after "
        + this.tilesDone + " tiles and "
        + this.timeSpent + " seconds.");
}

checkInterrupted();
if (threadOffset == 0 && doFilterUpdate) {
        runFilterUpdates(gridSubset.getName());
}

super.state = GWCTask. STATE.DONE;
```

```
            this.statusMessage = "Prefetch Task for Job '" + jobName
                    + "' finished.";
            this.tilesDone = tilesTotal;
        }
```

Código 222. Objeto `PrefetchTask`

Para consultar la implementación concreta de los módulos gwc-featurecatalog, `gwc-stats` y `gwc-prefetch`, se insta al lector a consultar el código fuente de este trabajo.

Capítulo 5

Manual de Usuario

La presente sección se corresponde con una guía en la que se detalla los pasos que deberá seguir el usuario para instalar, utilizar y comprender correctamente la interfaz *web* previamente descrita. Se muestra el tutorial para la versión en inglés de la página *web*.

5.1. Instalación

Para integrar los módulos *Seeder* en *GeoServer* se debe copiar los ficheros `.jar` donde se definen, que se proporcionan junto a este documento, en el directorio *webapps**geoserver**WEB-INF**lib* del directorio de instalación de *GeoServer*.

Para una correcta instalación, entre otros, deben instalarse también los siguientes módulos, disponibles en *Internet*:

- *Quartz* 1.6.3.
- *Commons-Math* 2.0.
- *Spring Batch Infraestructure*.

a parte de los propios módulos que ya vienen instalados por defecto con *GeoServer*, prestando especial atención a los módulos para las bases de datos *H2* y *PostGres*.

Para un correcto funcionamiento es requisito indispensable que la base de datos *PostGres* tenga perfectamente configurada la extensión *PostGIS*. En caso contrario la aplicación fallará. En este sentido no es suficiente con instalar solo una base de datos *PostGres*, se debe configurar correctamente con el *plug-in* para *PostGIS*.

Una vez instalados los módulos anteriormente indicados, se puede lanzar la aplicación mediante el ejecutable *Start GeoServer*. Si todo es correcto, debe poderse acceder a la página de *GeoServer* y acceder a cualquiera de las funcionalidad de los módulos *Seeder-IDELab*, tal y como se describe en las secciones posteriores.

5.2. Acceso al formulario de configuración Seeder

Una vez lanzada la página de *GeoServer*, accesible por defecto en la dirección `localhost:8080/geoserver/web`, se accede al menú *"Layers"* o de la sección *"Data"* del panel izquierdo. Una vez dentro del menú se selecciona una capa de las disponibles, tal y como se muestra en la Figura 8.

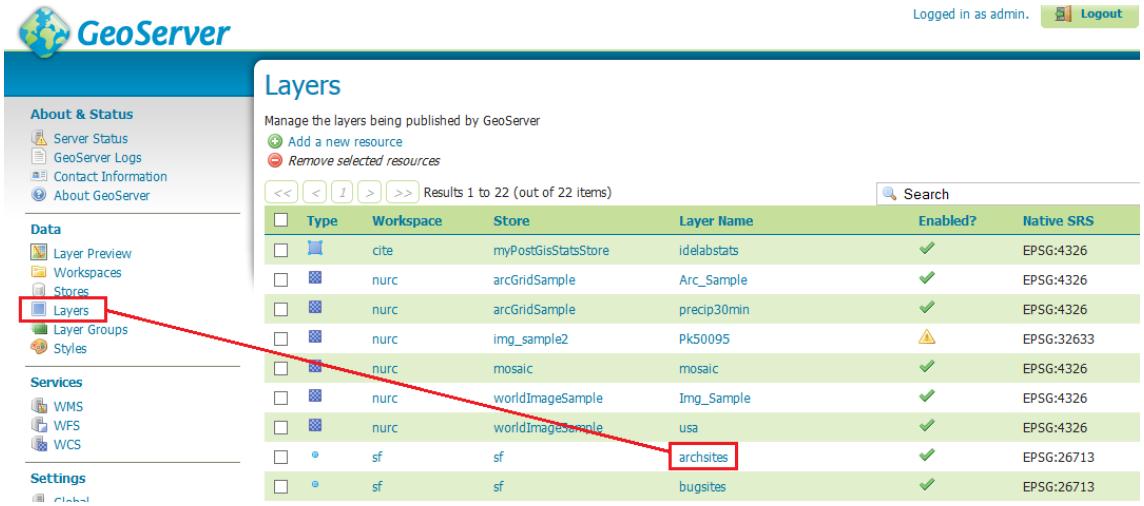

Figura 8. Página de acceso a capas ("*layers*")

Tras pulsar el correspondiente enlace, se accede a la página de edición de capas. En ella se puede apreciar que está estructurada en pestañas (*tabs*) tal y como se ha mencionado repetidas veces. De entre todas las pestaña, se selecciona la denomina *"Seeder Caching"* y se visualizará en pantalla el formulario de configuración *Seeder*, tal y como se muestra en la Figura 9.

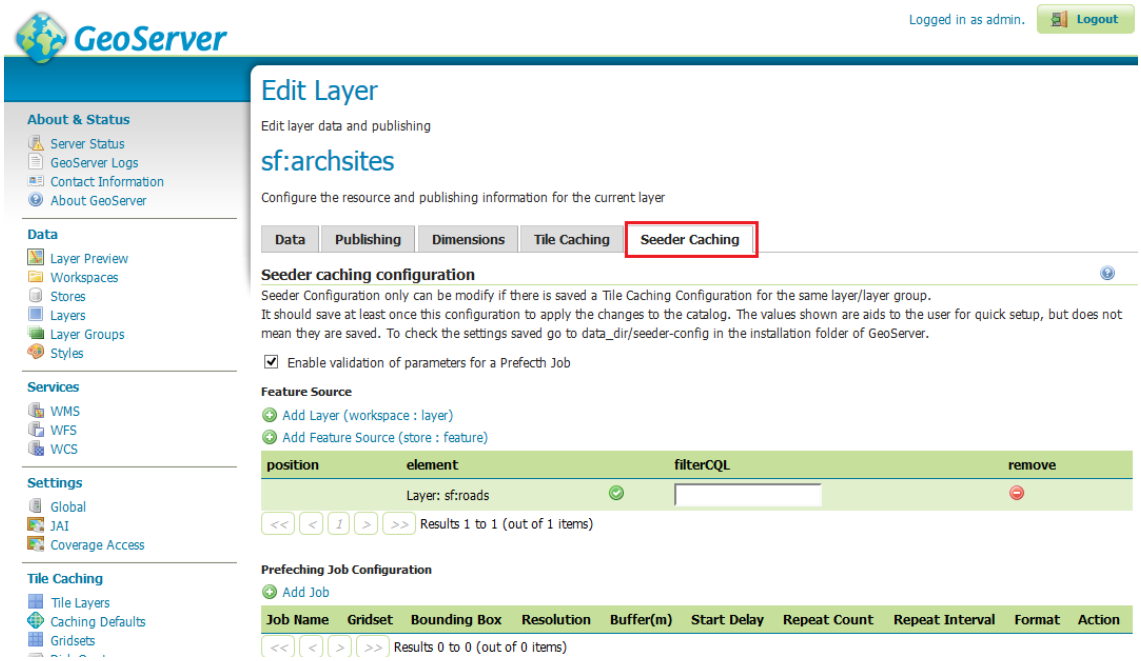

Figura 9. Página de modificación de capas. Pestaña *"Seeder Caching"*

En caso de querer acceder al formulario para un grupo de capas, el usuario debe seleccionar el menú *"Layer Groups"* de la sección *"Data"* del panel izquierdo. A continuación seleccionará un grupo de capas concreto, como se indica en la Figura 10.

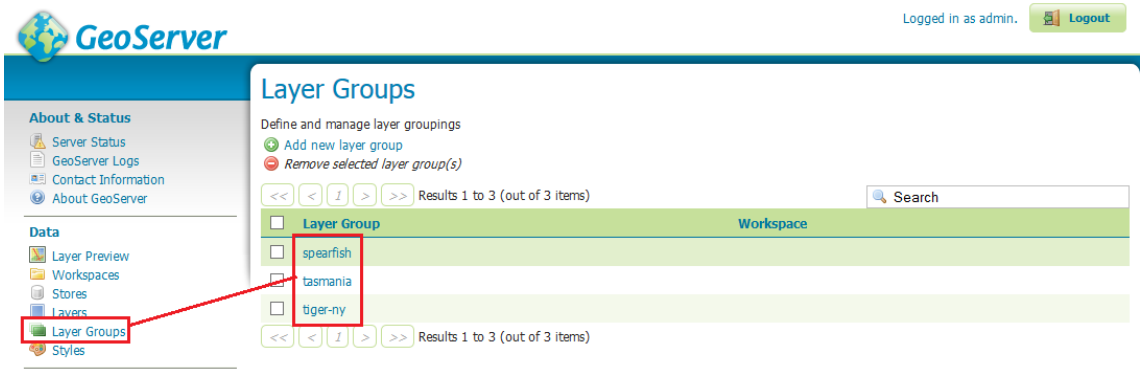

Figura 10. Página de acceso a grupo de capas

Tras ello, se accederá a una página en que deberá navegar para buscar donde se encuentra el formulario, por defecto, será el último panel de esta página.

A partir de ahora el manual de usuario se centra en el formulario dentro de la página de edición de una capa. En el caso de la página de edición de grupos de capas es totalmente análogo.

5.3. Panel de selección de *Features Sources*

El panel de selección de *Features Source* tiene por objetivo permitir y facilitar al usuario indicar a que orígenes de fenómenos pertenece la capa que se está editando. Este panel presenta un aspecto similar al mostrado en la Figura 11.

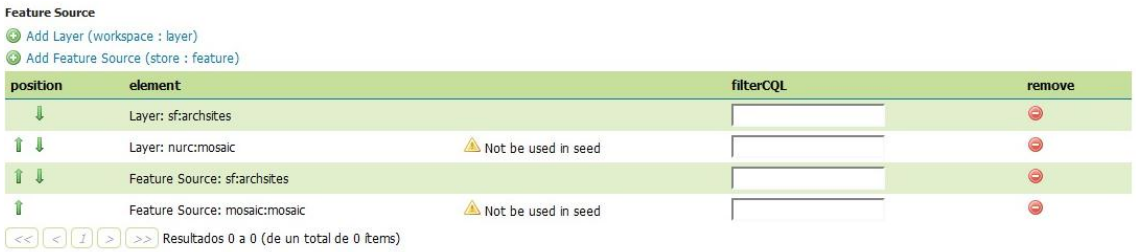

Figura 11. Panel de selección de *Features Sources*

Este panel está compuesto por cinco columnas:

- La primera permite al usuario modificar la posición de los elementos de la tabla.
- La segunda muestra el nombre del elemento añadido al panel distinguiendo si el *Feature Source* añadido es una capa publicada o un origen de fenómenos propiamente dicho.

- La tercera columna muestra un mensaje de aviso para aquellas entradas que representan *Features Sources* que no se corresponden con datos de tipo *vector* (Ver sección 4.3.3.1) porque no podrán ser usados en trabajos de *Prefetch*.
- El cuarto campo corresponde con un filtro a aplicar dentro del *Feature Source*.
- La quinta columna permite eliminar una entrada de la tabla pulsando un icono-enlace.

Inicialmente este panel se encontrará vacío, pero el usuario puede añadirle entradas mediante los enlaces que se aprecian antes del panel. El primero de ellos (*"Add Layer"*) se utiliza para agregar un origen de fenómenos que se corresponde con una capa que ya ha sido publicada. Tras pulsar este enlace se abre una ventana emergente, como la mostrada en la Figura 12.

En ella aparecen todas las capas publicadas en *GeoServer*. El usuario puede seleccionar una pulsando sobre su nombre. Tras seleccionarla, esta aparecerá con toda la información pertinente en el panel.

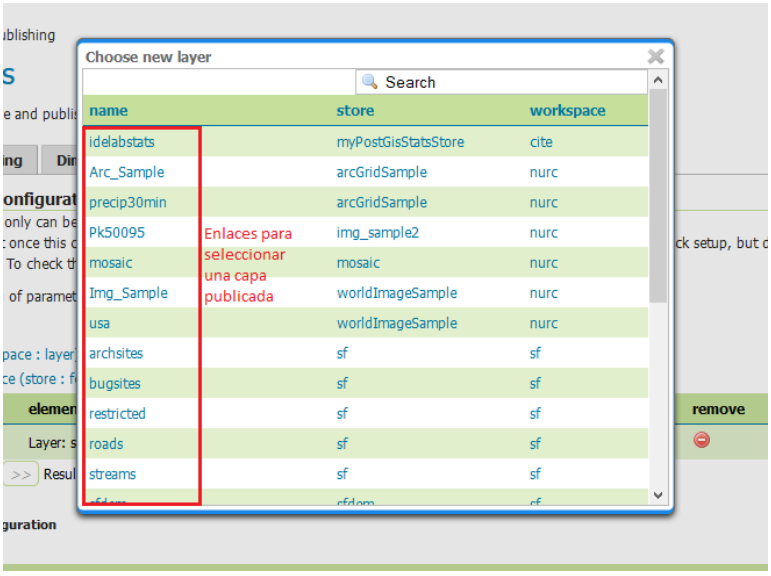

Figura 12. Ventana emergente de selección de capa publicada como *Feature Source*

El segundo enlace (*"Add Feature Source"*) permite añadir un *Feature Source* propiamente dicho. Al pulsar sobre este enlace, se lanza otra ventana emergente, que en primer lugar solicitará al usuario seleccionar un almacén de datos (mediante un desplegable de opciones) y después un origen de fenómenos (mediante un panel con las opciones disponibles). La opción escogida se selecciona pulsando el enlace *"Select"* correspondiente (Figura 13).

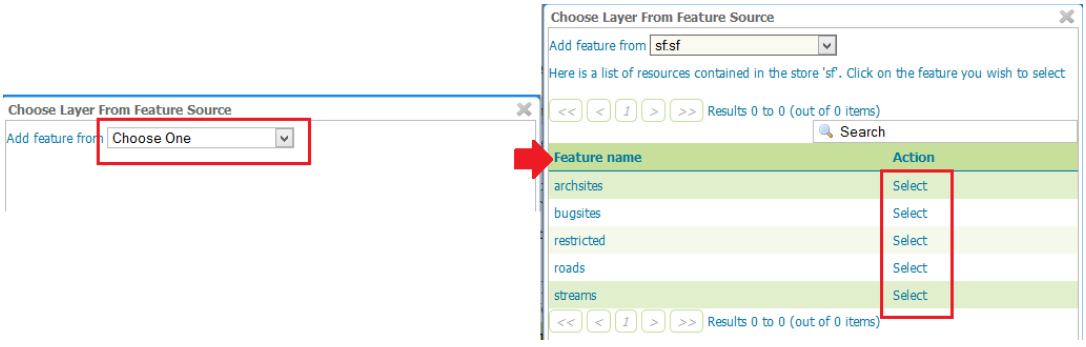

Figura 13. Ventana emergente de selección de *Feature Source*

Una vez el usuario ha seleccionado los *Features Sources* correspondientes, puede modificar su posición (aunque en la práctica no tiene funcionalidad) o borrarlos del panel si así lo desea. Además, en caso de que este tipo de *Features Sources* no sea válido para realizar un trabajo de *Prefetch* se mostrará un mensaje de aviso (Figura 14).

Figura 14. Panel de selección de *Feature Source* II

Por otro lado, no se puede guardar una configuración *Seeder* o intentar cambiar de pestaña si este panel está vacío, en caso de intentarlo aparecerá un mensaje de aviso en la parte superior de la página (Figura 15).

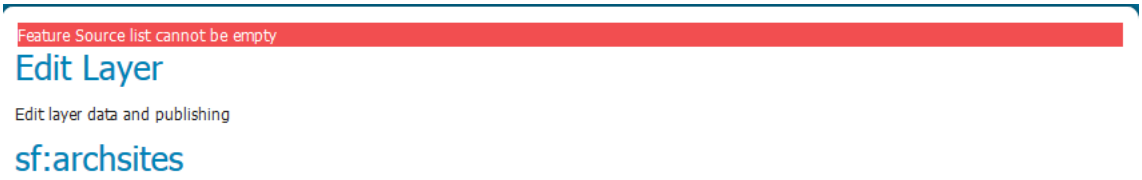

Figura 15. Mensaje de error I: *Feature Source* vacío

5.4. Panel de configuración de trabajos de *Prefetch*

Este panel permite al usuario indicar los parámetros de lanzamientos de un trabajo de *prefetch*. Presenta un aspecto similar al mostrado en la Figura 16.

El significado de las columnas es el siguiente:

- En la primera columna, *"JobName"*, se especifica el nombre del trabajo a lanzar, para diferenciarlos del resto. Campo obligatorio.
- La segunda columna indica el nombre del *GridSet* escogido para añadir el trabajo.
- En el campo *"Bounding Box"* se debe especificar el encuadre en que se quiere centrar el trabajo. Su valor está inicializado por defecto por la aplicación al añadir una nueva entrada a partir del encuadre de definición de la capa, pudiendo haber sufrido alguna transformación de unidades para adaptarlo a las unidades del *GridSet* seleccionado.
- En los campos *"Resolution"* y *"Buffer(m)"* el usuario especificará la resolución y el *buffer* (medido en metros) con que se quiere lanzar el usuario. Para más detalles en cuanto a estos conceptos ver Sección 2.4.
- Los siguientes tres campos permiten configurar el modo de lanzamiento de la tarea. En concreto, cuando debe lanzarse, cuantas veces debe repetirse y cuál es el intervalo temporal entre comienzo de repeticiones.
- En la columna *"Format"* el usuario especificará el formato de imágenes de las teselas que quiere analizar.

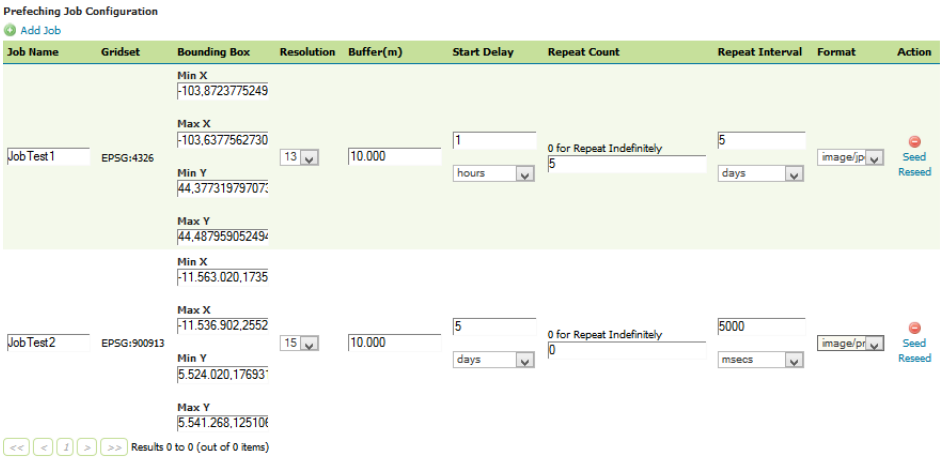

Figura 16. Panel de configuración de trabajos de *Prefetch*

- Por último, en la columna *"Actions"* se implementan tres enlaces que permiten, respectivamente, borrar una entrada de la tabla, lanzar una tarea de *Prefetch* de tipo *Seed* (traer a caché todas las teselas de nuevo) o de tipo *ReSeed* (traer a caché solo las teselas que faltan).

Si el usuario quiere añadir un nuevo trabajo debe pulsar el enlace *"Add Job"*. Tras ello se presentará una ventana emergente en que el usuario debe elegir entre las distintas posibilidades de *GridSet* configurados para esta capa, pulsando sobre el enlace *"Select"* correspondiente (Figura 17).

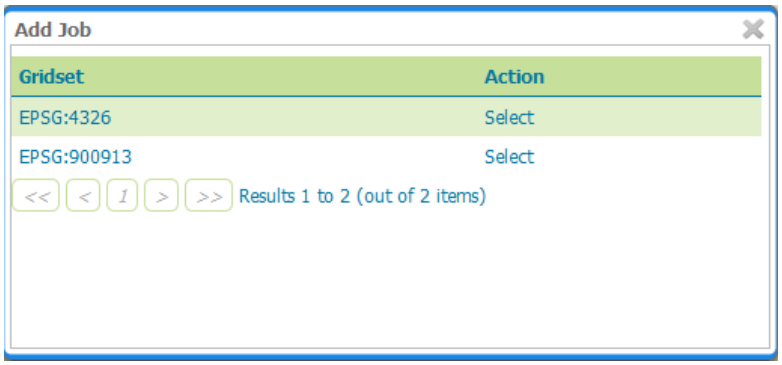

Figura 17. Ventana emergente para añadir un nuevo trabajo de *Prefetch*

Tras hacerlo, se añadirá una nueva entrada en la tabla, en la que, siempre que se pueda, aparecerán completados por defecto la mayor parte de los campo (excepto "*JobName*" y "*Format*"), para facilitar al usuario el manejo de los valores. Especialmente ventajosa es la inicialización del campo *"Bounding Box"*, ya que este se inicializa con los valores que definen el encuadre de la capa correspondiente, aunque transformados a las unidades métricas del *GridSet* seleccionado, tal y como se indicó anteriormente. Aunque estos campos estén inicializados por defecto, el usuario puede modificarlos sin problemas. En caso de que no sean correctos, se le indicará cual es el error.

A la hora de guardar una configuración *Seeder* o intentar cambiar de pestaña, en la parte superior de la página se mostrarán los siguientes mensajes de aviso o error:

- En caso de que el retardo especificado no sea positivo, Figura 18.

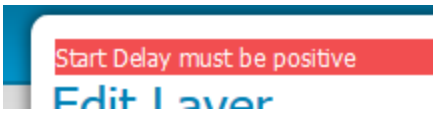

Figura 18. Mensaje de error II: Retardo Inicial

- En caso de que el número de repeticiones no sea coherente, Figura 19.

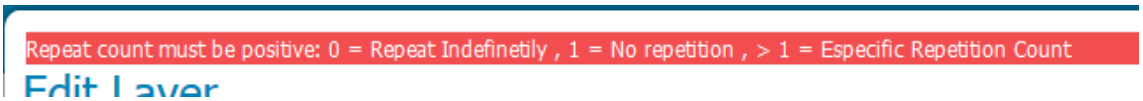

Figura 19. Mensaje de error III: Número de repeticiones

- En caso de que se haya especificado que se deben realizar repeticiones del trabajo, no se haya especificado el intervalo entre repeticiones o no sea coherente, Figuras 20 y 21.

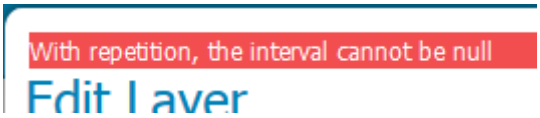

Figura 20. Mensaje de error IV: Intervalo de repetición

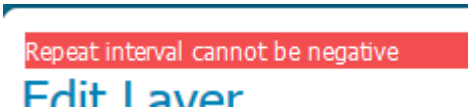

Figura 21. Mensaje de error V: Intervalo de repetición

- En caso de que el *buffer* especificado sea negativo, Figura 22.

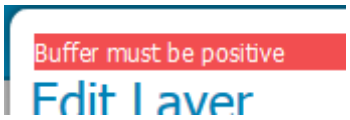

Figura 22. Mensaje de error VI: *Buffer*

- En caso de que no se haya configurado un nombre para el trabajo, Figura 23.

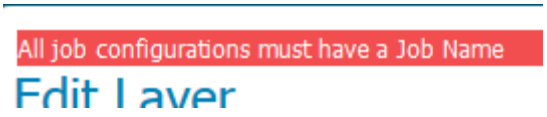

Figura 23. Mensaje de error VII: Nombre de trabajo

- En caso de que no se haya especificado un formato de imagen, Figura 24.

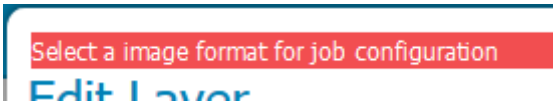

Figura 24. Mensaje de error VIII: Formato de imagen

Una vez añadida una entrada al panel:

- Puede borrarse.
- Lanzarse un trabajo como *Seed*.
- Lanzarse un trabajo como *ReSeed*.

Mediante los enlaces implementados en la columna *"Action"*, como se muestra en la Figura 25.

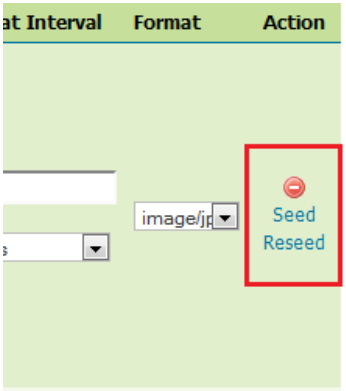

Figura 25. Detalle panel de configuración de trabajos de *Prefetch*. Campo *"Actions"*

Si el usuario selecciona el enlace para borrar una entrada se le solicitará confirmación mediante un diálogo emergente, Figura 26.

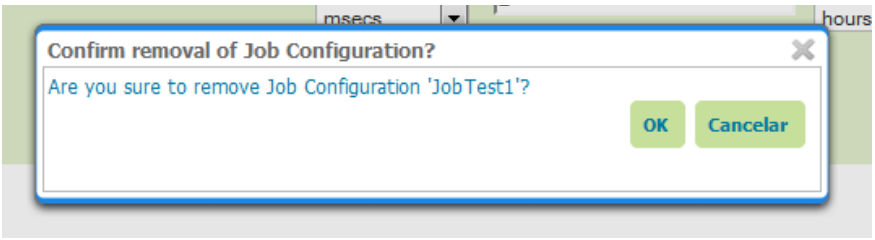

Figura 26. Mensaje confirmación I. *Prefetch Panel*

Si el usuario lanza una tarea de *Seed*/*ReSeed*, se comprobarán todos los datos introducidos en el formulario, y si alguno no es adecuado, lanzará una ventana emergente con el mensaje de error correspondiente:

- El panel de *Features Source* no contiene ninguna entrada válida, Figura 27.

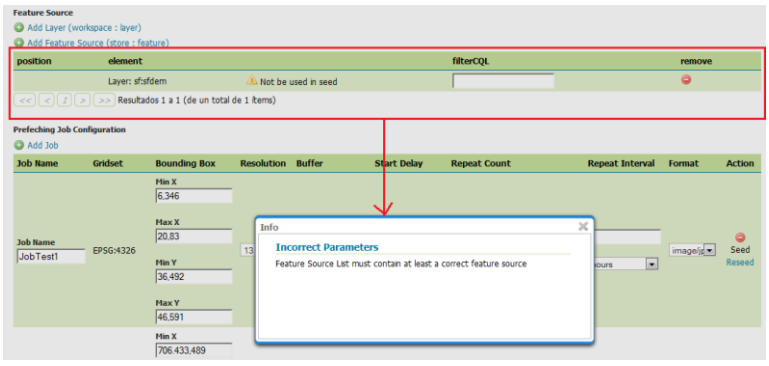

Figura 27. Mensaje de error IX. *Feature Source*

- No se ha especificado un nombre para el trabajo, Figura 28.

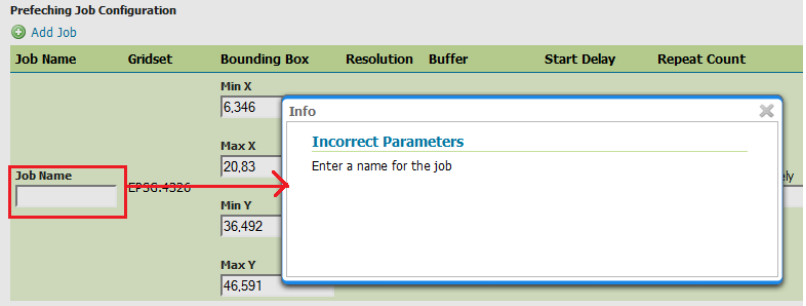

Figura 28. Mensaje de error X: Nombre de trabajo

- Existe otro trabajo con el mismo nombre. Figura 29.

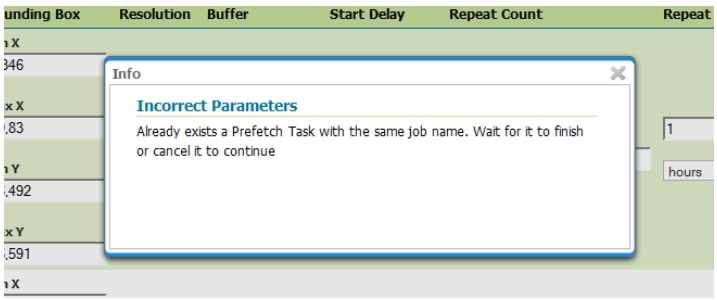

Figura 29. Mensaje de error XI

- El *buffer* introducido no es correcto, es decir, no es una cantidad positiva, Figura 30.

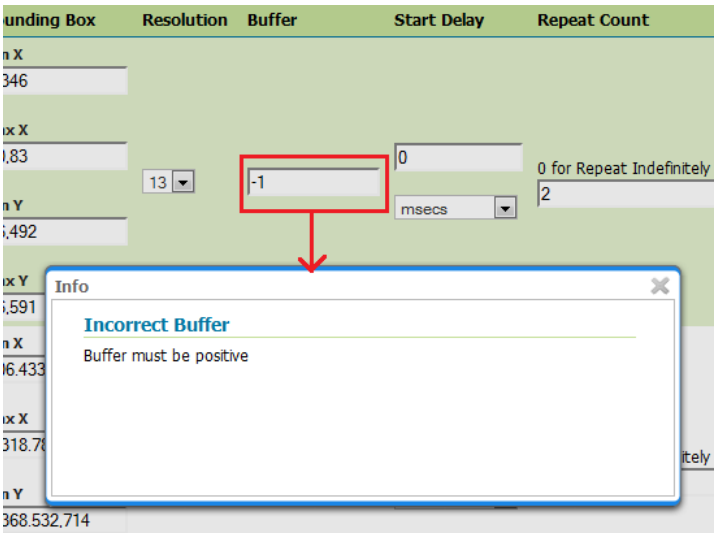

Figura 30. Mensaje de error XII: *Buffer*

- Se ha especificado un retardo temporal negativo, Figura 31.

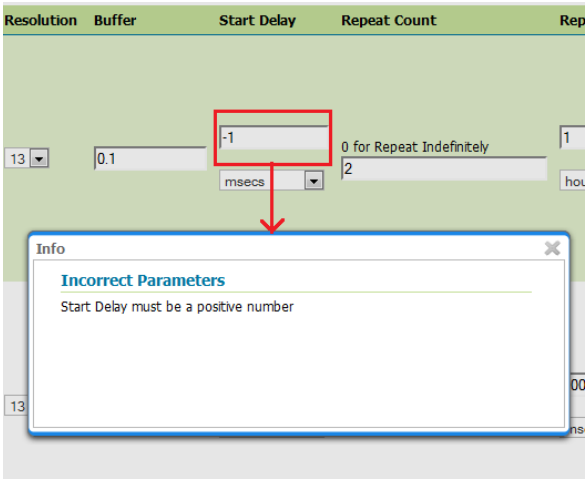

Figura 31. Mensaje de error XIII: Retardo Inicial

- Se ha especifico un número de repeticiones negativo, Figura 32.

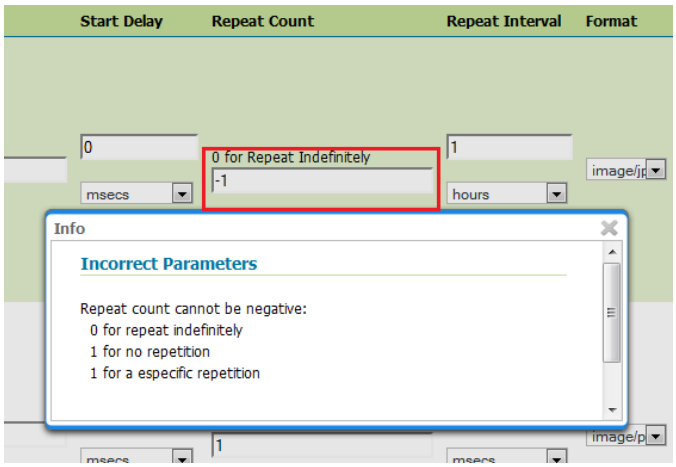

Figura 32. Mensaje de error XIV: Número de repeticiones

- Habiéndose seleccionado la opción de lanzar un número concreto de repeticiones, el intervalo temporal entre repeticiones especificado no es un valor coherente, Figura 33.

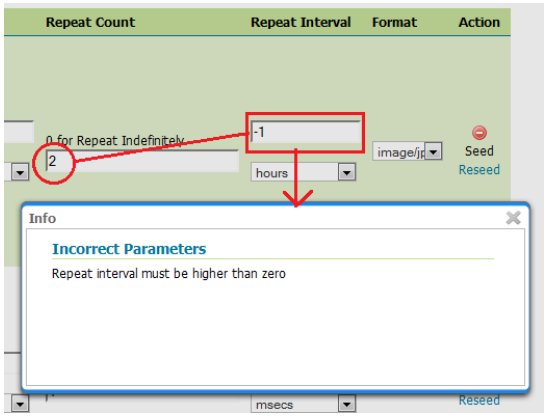

Figura 33. Mensaje de error XV: Intervalo de repetición

- No se ha escogido un formato de imagen, Figura 34.

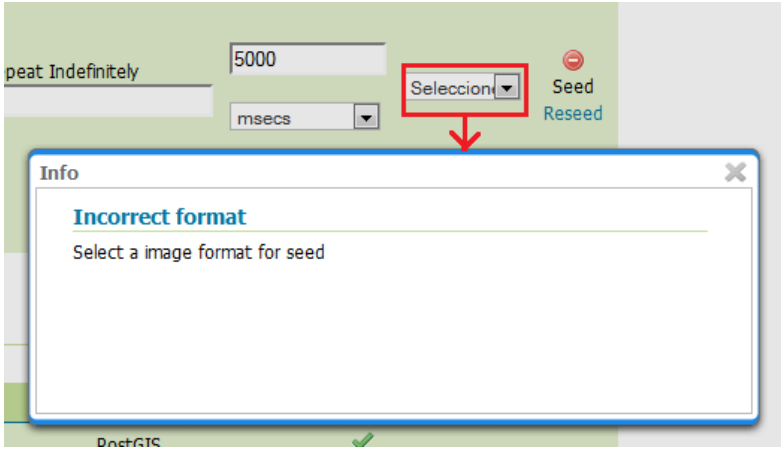

Figura 34. Mensaje de error XVI: Formato de imagen

- No se ha configurado una base de datos para las estadísticas de acceso a esta capa, Figura 35.

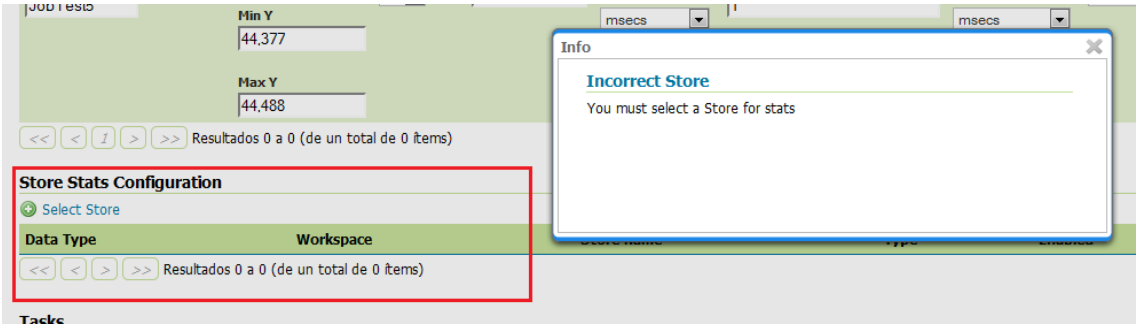

Figura 35. Mensaje de error XVII: Base de datos

En caso de no especificar dato en alguno de estos campos, se tomará el último valor introducido, aunque ya no se encuentre "escrito" en la correspondiente casilla del formulario. Por tanto, excepto los campos *"JobName"* y *"Format"*, el resto siempre tendrán valor, porque al menos han sido inicializados con los valores por defecto que genera la aplicación, y por ello no se implementa el manejo de errores en caso de que estos campos estén vacíos.

Además la aplicación se cuida de que el usuario pueda guardar la configuración con campos vacíos o datos incoherentes, por ejemplo, un carácter no numérico en un campo numérico. En estas situaciones se generarán avisos de error en la parte superior de la página, como los mostrados en la Figura 36.

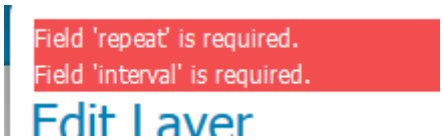

Figura 36. Mensaje de error XVIII

5.5. Panel de selección de base de datos

Este panel permite al usuario seleccionar una base de datos para esta capa con dos propósitos:

- Indicar donde almacenar las estadísticas de acceso a esta capa.
- Indicar de donde extraer las estadísticas de acceso para poder realizar los trabajos de *Prefetch*.

Presenta un aspecto similar al mostrado en la Figura 37.

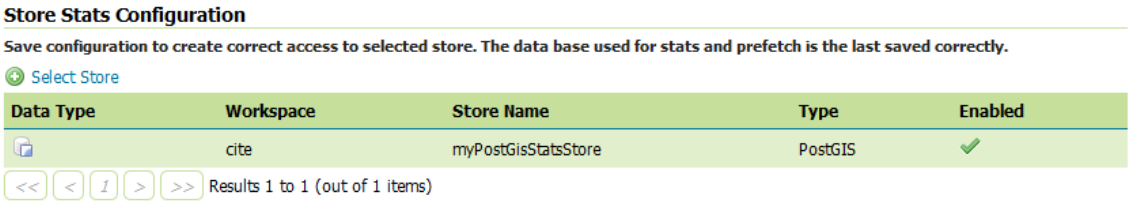

Figura 37. Panel de selección de base de datos

El enlace *"Select Store"* permite al usuario seleccionar una base de datos. En concreto, al ser pulsado, genera una ventana emergente en que se muestran todas las bases de datos que el usuario puede seleccionar para los propósitos de este módulo, es decir, son bien de tipo *H2* o bien de tipo *PostGIS* por las limitaciones impuestas por el módulo presentado en la Sección 4.4.2.

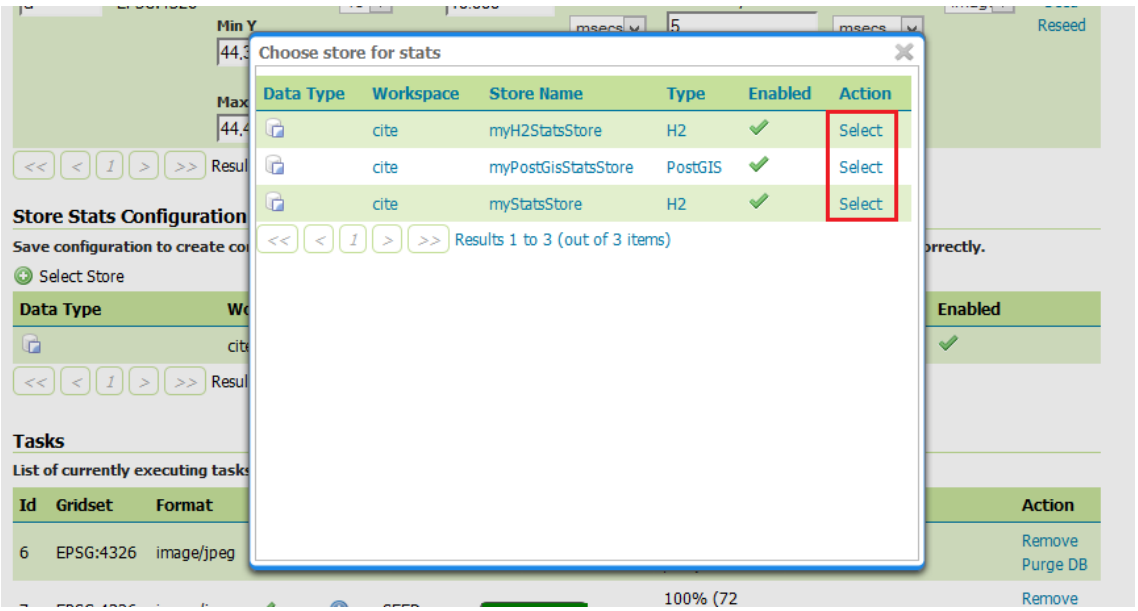

Figura 38. Ventana emergente de selección de base de datos

Para seleccionar una base de datos se debe pulsar el enlace *"Select"* correspondiente, tras ello se añadirá a la tabla.

El usuario no puede eliminar ninguna entrada de esta tabla, ya que cada capa debe tener al menos una base de datos asociada, una misma base de datos puede estar asociada a varias capas/grupos de capas. Si el usuario quiere modificar la base de datos asociada a una capa

deberá seleccionar otra de la lista de bases de datos disponibles. Automáticamente se borra la anterior y se añade la nueva. Estos cambios no son efectivos hasta que el usuario guarda la configuración *Seeder*, como bien advierte el mensaje de aviso que precede a este panel.

5.6. Panel de visualización de tareas lanzadas

Este panel permite al usuario visualizar todas las tareas lanzadas sobre esta capa, tanto las de *GWC* como las que añade el módulo *Seeder*. Presenta un aspecto similar al mostrado en la Figura 39.

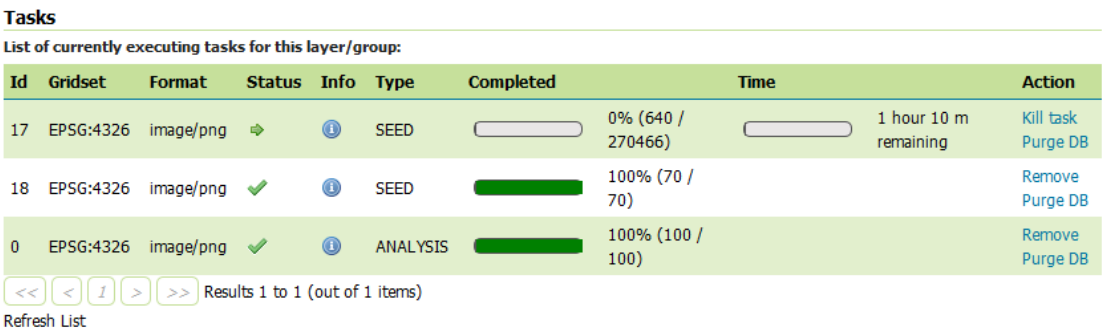

Figura 39. Panel de visualización de tareas lanzadas

El usuario obtendrá información sobre las tareas propias de *GWC* como son *Seed* y *Truncate*, lanzadas desde las utilidades que presenta *GWC* en el apartado *"Tile Caching"* → *"Tile Layers"*. También podrá observar las tareas propias del módulo *Seeder*. Estas son las tareas de análisis y las tareas de *Prefetch*. Cuando un usuario lanza una tarea de *Prefetch*, antes de proceder con ella, se lanza una tarea, que se denomina de análisis, en que se recopilan todos los datos, principalmente las estadísticas de acceso a las teselas para crear un lista ordenada de las teselas que se deben traer a caché.

El usuario puede configurar una tarea de *Prefetch* que se repita múltiples e incluso indefinidamente. Cada una de estas repeticiones constará de una tarea de análisis y una tarea de *Prefetch*.

El enlace inferior *"Refresh List"* permite actualizar la lista.

La mayor parte de los campos de este panel son descriptivos por si mismos:

- El campo *"Status"* se corresponde con una imagen-enlace. El significado de los diferentes iconos que pueden aparecer en él es el siguiente:

  Tarea en ejecución

  Tarea terminada satisfactoriamente

  Error en la tarea

Esperando la inicialización de una tarea de análisis.

Al pulsar sobre cada uno de estos iconos se abre una ventana emergente con información relativa al estado en que se encuentra la tarea. Se pueden ver ejemplos en la Figura 40.

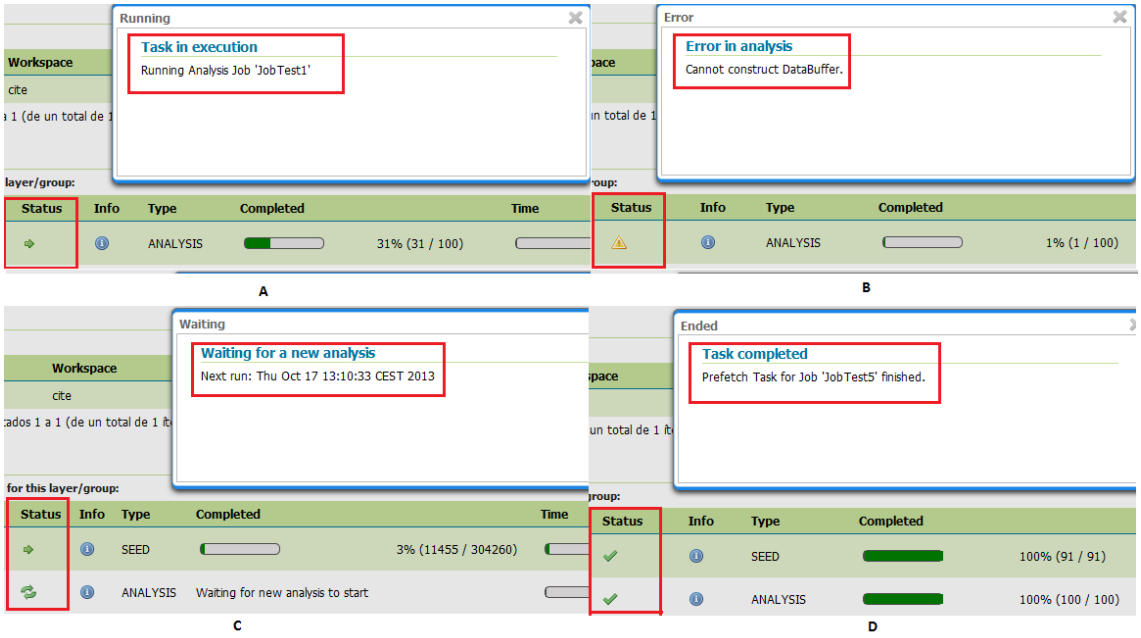

Figura 40. Mensajes de estado: tarea a) en ejecución, b) errónea, c) en espera por un nuevo análisis, d) completada

- El campo *"Info"* se define como otra imagen-enlace que al ser pulsada lanza una ventana emergente con toda la información correspondiente a la tarea. Pueden verse un ejemplo en la Figura 41.

  La información proporcionada al usuario será:

  - Tipo de tarea.
  - Nombre de la tarea (*"Job Name"*).
  - Base de datos de la que se están recogiendo las estadísticas (*"Stats Store"*).
  - Capa sobre la que se ha lanzado la tarea (*"Layer"*).
  - *GridSet* de la tarea.
  - Formato de imagen en que se están guardando las teselas en la máquina de usuario (*"Format"*).
  - Encuadre en que se está realizando el trabajo (*"Bounding Box"*).
  - *Buffer* y resolución del trabajo.
  - Si se lanzó la tarea como *Seed* (traer todas las teselas a caché) o *ReSeed* (traer solo las teselas que falta).
  - La primera imagen que se muestra al usuario representa la capa que se está visualizando. En color blanco, sobre el fondo negro, se especifican las zonas en que está definida la información de la capa, aplicando el valor de *buffer* configurado por el usuario.

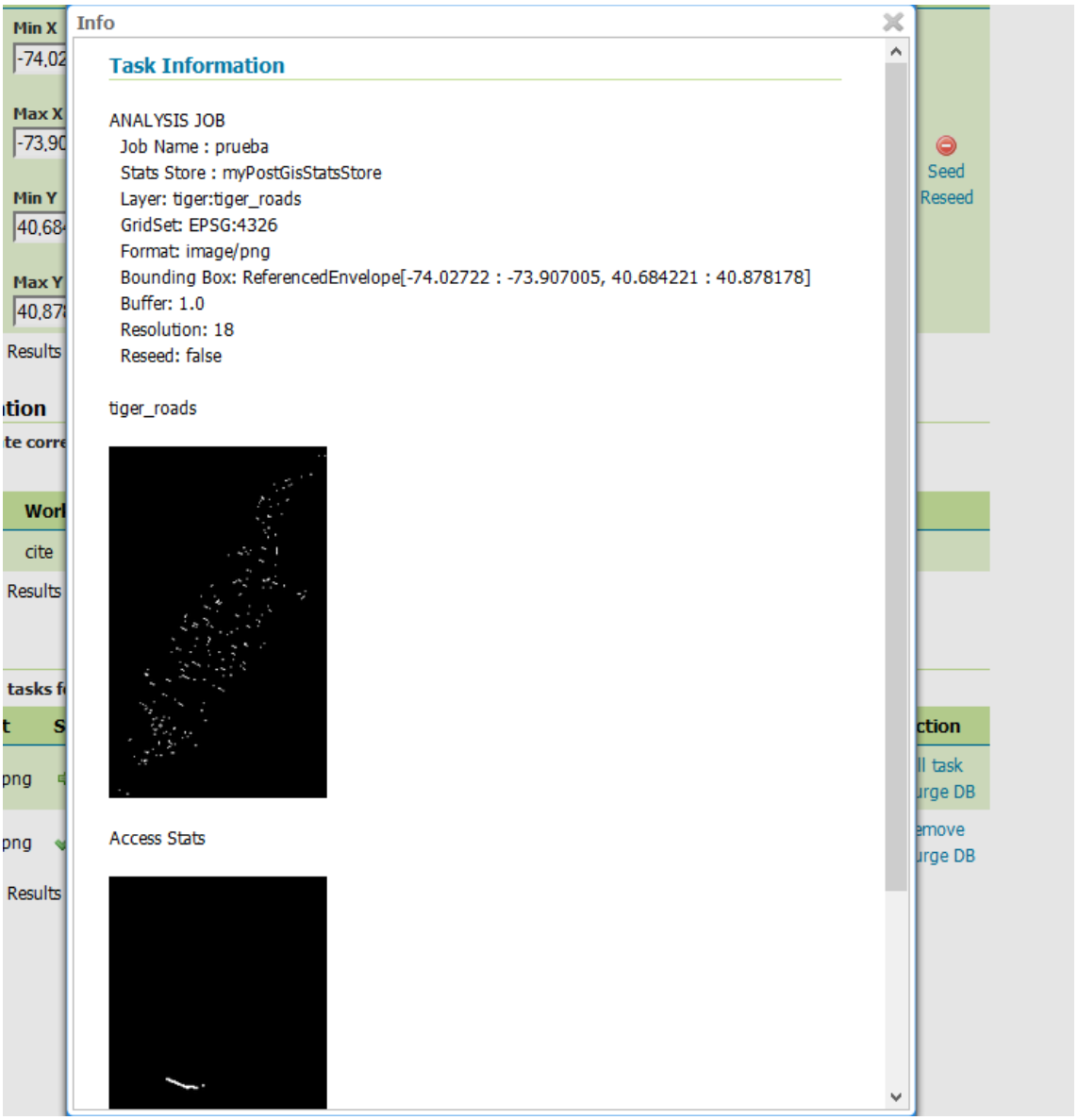

Figura 41. Mensaje de información de una tarea de análisis

- En la segunda imagen se muestra en blanco aquellas teselas que han sido visitadas dentro del encuadre y en función de la resolución, formato y demás que ha especificado el usuario.

Pulsado sobre cualquiera de estas dos imágenes se abre una nueva ventana en el navegador, mostrando la imagen en tamaño original para que el usuario pueda visualizarla en más detalle.

- Los campos *"Completed"* y *"Time"* muestran al usuario el progreso de la tarea en tanto por ciento y el tiempo restante tanto de forma gráfico como de manera textual.

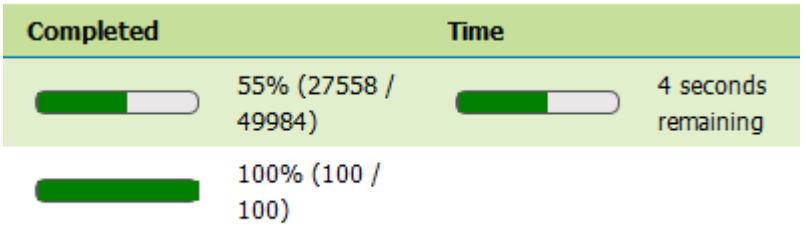

Figura 42. Detalle panel de visualización de tareas

- Los enlaces especificados en *"Action"* permiten realizar acciones sobre las tereas:
  - Si la tarea ya ha finalizado, el enlace *"Remove"* permite eliminar su información del panel. Se considera que una tarea también ha finalizado cuando ha resultado errónea. En este caso el resultado no es satisfactorio, pero a efectos del panel, es igual que una tarea finalizada correctamente, ya no queda rastro de ella en la aplicación y solo se muestra al usuario información sobre lo que se realizó. En este punto se borra esta información.
  - Si la tarea no ha finalizado, el enlace *"Kill Task"* permite cancelar la tarea. Llegados a este punto el usuario deberá tener en cuenta.
    - Para tareas de *Seed* y *Truncate* se cancela la tarea.
    - Para tarea de *Prefetch* también cancela la tarea, aunque no desprograma el conjunto de repeticiones que el usuario haya podido establecer.
    - Para una tarea de análisis en ejecución, se cancela este análisis y por tanto la siguiente tarea de *Prefetch*.
    - Para una tarea de análisis en espera, es decir, en el intervalo entre repeticiones, cancela la programación completa de la tarea, es decir, todas las repeticiones restantes.
  - Tanto si la tarea ha finalizado como si sigue en ejecución/espera el enlace *"PurgeDB"* permite borrar todas las estadísticas de acceso relacionadas con la tarea correspondiente de la base de datos, es decir, todas las estadísticas que se usan para realizar este conjunto de tareas análisis/*Prefetch*. Esta opción, solo está disponible para las tareas de análisis y *Prefetch*, ya que las tareas de *GWC* no se relacionan con conceptos de base de datos. Por ello, si el usuario pulsa este enlace para una tarea de *GWC* se lanzará un mensaje indicando que esta opción no está disponible para este tipo de tareas, tal y como se muestra en la Figura 43.

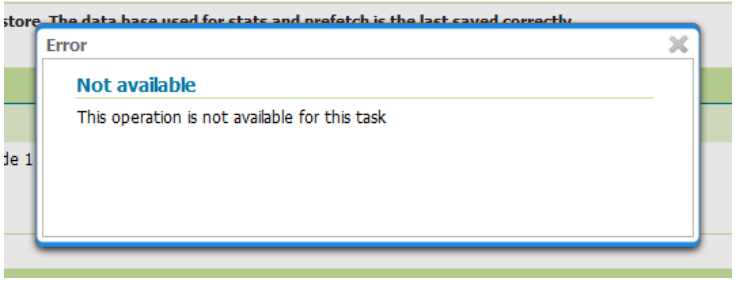

Figura 43. Mensaje de error XIX: Operación no disponible

En caso de lanzar esta operación para una tarea de análisis o *Prefetch*, se solicitará confirmación por parte del usuario, con un diálogo parecido al mostrado en la Figura 44.

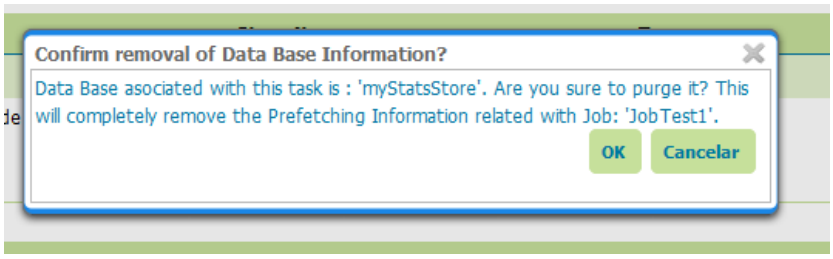

Figura 44. Mensaje de confirmación II. *Task Panel*

Capítulo 6

Conclusiones y Líneas Futuras

Para concluir este trabajo se presentan las conclusiones del mismo así como posibles estudios futuros que tengan por objetivo su mejora.

Hoy en día *GeoServer* proporciona funcionalidades relacionadas con los datos espaciales, en concreto, permite el intercambio y edición de datos espaciales. Consta de una aplicación *web* (*GeoWebCache*) mediante la cual se puede cachear teselas de los mapas visualizados para acelerar el proceso de descarga de un mapa tras haber sido ya visitado. Pero a lo largo de este trabajo se ha podido observar que ese cacheado no se puede considerar del todo eficiente, porque, o bien el usuario debe recorrer los mapas para guardar sus teselas o bien debe seleccionar que teselas concretas quiere guardar tal y como se indicó en la Sección 2.3.

Los módulos `gwc-stats`, `gwc-featurecatalog` y `gwc-prefetch` (ver Sección 2.5) permiten ampliar esta funcionalidad. Más en detalle, permiten traer a caché aquellas teselas que tienen una mayor probabilidad de ser solicitadas por parte del usuario.

El presente trabajo ha tenido por objetivo implementar el módulo `gwc-seeder-web`, el cual tiene por objetivo dotar de interfaz *web* a los módulos anteriores para facilitar el uso de los mismos por parte del usuario, permitiendo simultáneamente su integración en la interfaz de *GeoServer*. Así se ha conseguido llevar a cabo el desarrollo presentado en los capítulos previos, aunque no exento de una serie de limitaciones en su uso, en concreto el usuario se va a encontrar con las siguientes restricciones:

- Solo puede escoger bases de datos de tipo *PostGIS* o *H2* para el almacenamiento de las estadísticas. Pueden aparecer problemas de sintaxis *SQL* si las bases de datos utilizadas tienen versiones distintas a las probadas.
- Puede guardar una configuración sin validar datos si deselecciona la opción correspondiente pero nunca podrá lanzar un trabajo si los valores introducidos no son correctos. En cualquier caso se le avisará de que los valores introducidos no son válidos.

- A pesar de que puede lanzar trabajos con configuraciones no guardadas, antes de lanzar un trabajo (después de una modificación o tras haberlo añadido) se aconseja guardar la configuración.
- Puede haber problemas al lanzar trabajos con *GridSet* diferentes a los *GridSet* de los *features* que se seleccionen, ya que algunas transformaciones de *CRS* pueden ser incorrectas o no posibles de realizar. En caso de error se mostrará un mensaje al usuario, y en este caso se recomienda que intente lanzar un trabajo equivalente con otro *GridSet*.
- A los trabajos lanzados se les deberá poner nombre, no pudiéndose lanzar dos o más trabajos con el mismo nombre.
- No se podrá visualizar un trabajo hasta la primera vez que se lance el análisis previo a los trabajos de *prefetch*, por ello el usuario deberá tener cuidado con el parámetro *"Start Delay"* porque hasta que no finalice este tiempo después de lanzar un trabajo no será consciente de la existencia de ese trabajo programado (no se puede visualizar en el panel de tareas).
- Si el usuario cambia de nombre de capa durante el transcurso de un trabajo, perderá la referencia con él y no podrá visualizarlo, ni realizar ninguna otra tarea que permite la interfaz con los trabajos lanzados.
- El usuario también deberá tener cuidado con cambiar parámetros no pertenecientes a la configuración *Seeder* pero que si le afecten, mientras se está ejecutando un trabajo porque puede causar problemas o errores no contemplados. Por ejemplo, al eliminar un *GridSet* de la configuración *"Tile Layer"* se eliminan todas las configuraciones de trabajos relacionados con ese *GridSet* pero no se manda finalizar los trabajos, por lo que el usuario debería previamente a eliminar un *GridSet* finalizar todos los trabajos relacionados, en caso contrario la aplicación fallará.
- En caso de que ocurra cualquier error a la hora de realizar los trabajos, se mostrará al usuario una advertencia indicando brevemente cual ha sido el problema ocurrido. Para mayor información sobre el error, el usuario deberá acudir a los ficheros de *log* de *GeoServer*.
- La interfaz permitirá purgar las bases de datos a partir de los trabajos realizados. En este caso se entiende purgar por eliminar todas las estadísticas que este un trabajo ha utilizado para realizar el análisis. En caso de que el usuario quiera borrar por completo la base de datos deberá ir capa por capa para cada combinación posible de parámetros de *Prefetch* purgando los datos de cada trabajo o recurrir a cualquier aplicación externa para administrar su base de datos.

Teniendo en cuenta estos aspectos todavía quedaría trabajo por desarrollar para una implementación totalmente completa de la funcionalidad objetivo, entre lo que se podría destacar:

- Permitir el uso de otras bases de datos distintas a *PostGIS* o *H2*. Realmente este trabajo formaría parte de la mejora de los módulos `gwc-stats`, `gwc-featurecatalog` y `gwc-prefetch`.
- En caso de modificar parámetros que afecten a los trabajos de *prefetch* lanzados, realizar los cambios oportunos para mantener la coherencia o compatibilidad. Por

ejemplo, que en caso de cambiar el nombre de una capa no se pierda la referencia con todos los trabajos lanzados para ella, o al eliminar un *GridSet* de una capa, finalizar todos los trabajos relacionados para evitar un fallo de la aplicación.

- Mostrar al usuario mensajes más detallados y entendibles en torno al progreso de una tarea lanzada, especialmente los mensajes de error, para evitar que el usuario tenga que recurrir a los ficheros de *log* de la aplicación para buscar la explicación concreta del error y sobre todo que estos mensajes sean más amigables.
- Antes de lanzar un trabajo, comprobar que la base de datos de la que se van a recoger las estadísticas para determinar que teselas traer a caché, tiene datos relacionados con él, ya que en caso contrario no tiene sentido proceder con su ejecución.
- Manejo de tipos de datos distintos a *vector*, como por ejemplo, *raster*.
- Manejo más detallado y concreto de los cambios de unidades de datos espaciales.
- Uso del campo *CQL* para filtrar dentro de un *Feature* Source.
- Permitir la visualización de los trabajos desde el momento concreto en que se programan y no desde que se ejecutan por primera vez.
- Integrar una pequeña interfaz que permita la administración de las bases de datos empleadas para el almacenamiento de estadísticas para evitar que el usuario tenga que recurrir a herramientas externas.
- Hacer efectivos los cambios del panel de selección de base de datos sin necesidad de guardar el formulario de configuración *Seeder*.

Referencias

[1] Página *web* oficial de *GeoServer*, accesible en http://geoserver.org/. Último acceso 20 de Marzo de 2014

[2] Página *web* oficial de *GeoWebCache*, accesible en http://geowebcache.org/. Último acceso 20 de Marzo de 2014

[3] *Wickets*, *framework* de *Apache*, accesible en http://wicket.apache.org/. Último acceso 20 de Marzo de 2014

Contenido del CD adjunto

- Memoria del trabajo en formato *PDF*.
- Código fuente del trabajo ubicado en el directorio "*Código Fuente"*.
- Módulos del trabajo en formato `.jar` ubicados en el directorio "*Módulos"*.