UNIVERSITAT POLITÈCNICA

DE VALÈNCIA

Escuela Técnica Superior de Ingeniería

Geodésica, Cartográfica y Topográfica

Grado en Ingeniería Geomática y Topografía

PROYECTO DE FINAL DE GRADO

DISEÑO DE UNA IDE, GEOPORTAL Y APLICACIÓN WEB DE DENUNCIAS ADAPTADA A DISPOSITIVOS MÓVILES (TORRENT)

TUTOR:

José Carlos Martínez Llario

AUTOR:

Jose Ángel Hermosilla Rodrigo

Torrent, Julio 2016

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1. OBJETIVO DEL TRABAJO

El presente proyecto tiene varios objetivos.

Se pretende desarrollar para el ayuntamiento de Torrent (Valencia) una infraestructura de datos espaciales (IDE) conforme a los estándares especificados por el Open Geospatial Consortium (OGC), que también será acorde a las especificaciones de la directiva INSPIRE y cuyo producto final y visible para el usuario consistirá en un Geoportal (Página web) que ofrecerá unos servicios y funcionalidades mediante los cuales el usuario podrá interactuar con la Información Geográfica (IG) servida.

Estos servicios se encargarán de facilitar al usuario la **localización**, **visualización** y **descarga** de nuestra información geográfica.

Por otra parte, se implementará una aplicación web, adaptada a dispositivos móviles que consistirá en una pequeña red social para los ciudadanos de Torrent mediante la cual un usuario registrado podrá reportar incidencias, sugerencias, mejoras, etc. de cualquier elemento susceptible de ser espacialmente referenciado.

Además de almacenar la geometría, estos reportes tendrán otro tipo de datos como un título, una descripción, imágenes, etiquetas, etc.

Los datos generados por la aplicación también se incluirán dentro de nuestra IDE y producirán un mapa de incidencias dinámico, el cual nos mostrará las zonas donde más registros se han localizado en dos períodos de tiempo determinados.

La aplicación tendrá los elementos básicos de una red social como pueden ser un sistema de usuarios y autenticación, sistema de notificaciones, sistema de comentarios, etc.

Es por ello que aparecen otros objetivos previos como analizar las tecnologías que existen para desarrollar lo planeado, elegir las más idóneas y profundizar en las elegidas hasta tener el conocimiento necesario para lograr los objetivos iniciales.

Este proyecto pretende mostrar las prácticas a desarrollar para consolidar una IDE y elaborar una aplicación web relacionada con el ámbito de la Geomática, es decir que se alimente de datos geográficos y ofrezca productos derivados de operaciones geoespaciales, apoyándonos en el uso de las nuevas tecnologías abiertas basadas en la colaboración a la comunidad.

La aplicación desarrollada está concebida como una aplicación colaborativa, donde los propios usuarios alimentan la información. Pretende ser un servicio adicional del Geoportal y brindar la posibilidad de realizar denuncias sociales a los ciudadanos de Torrent. También pretende familiarizar al usuario con la Información Geográfica mediante el uso de mapas interactivos, búsquedas geoespaciales, avisos de proximidad, etc.

2. INFRAESTRUCTURA DE DATOS ESPACIALES

2.1. INTRODUCCIÓN

Actualmente entendemos que los recursos cartográficos son o deben ser una fuente de información pública, contrastada y estandarizada.

El control, la producción y el acceso a la cartografía antiguamente recaían en poderes gubernamentales y se utilizaba en labores de defensa, recolección de impuestos, desarrollo, etc.

Con la aparición de la tecnología SIG y la cartografía digital, se produjo un incremento y diversificación enorme del uso y del tipo de aplicaciones que usaban información cartográfica.

Estas tecnologías permitían que así cualquier persona pudiera crear sus propios mapas, gracias a la compartición de recursos cartográficos en los SIG de escritorio alimentados fácilmente con productos derivados de operaciones de análisis espacial, datos GPS, imágenes satélites, etc.

Este avance supuse el fin del modelo del viejo monopolio donde la producción cartográfica caía en mano de pocos organismos y con una serie de productos cartográficos muy acotada.

Este periodo de la tecnología SIG vino caracterizada por (Llario, s.f.):

- La aparición de muchos actores implicados en la distribución y recopilación de datos espaciales.
- La proliferación de aplicaciones SIG, diferentes tipos de productos y formatos digitales.

Lo cual originó la aparición de una serie de nuevos problemas como (Llario, s.f.):

- La duplicación de la información geográfica, ya que los diferentes actores no eran conscientes de que quizás alguien ya había creado la cartografía que estaban tratando de producir debido a la dificultad de descubrir y acceder a toda la información cartográfica existente pero no visible.
- Dificultad para compartir la información tanto por problemas técnicos como la aparición de formatos diferentes como por problemas organizativos al no estar los actores acostumbrados a compartir información ni tampoco los organismos preparados para ello.
- Dificultad para evaluar los productos cartográficos existentes, es decir, su calidad, metadatos, etc. lo cual impedía su uso en muchos casos.

Las infraestructuras de datos espaciales (IDE) aparecieron con la finalidad de resolver estos problemas, entre otros.

En el documento de la agenda 21, plan general aprobado en la conferencia de las Naciones Unidas en Río de Janeiro sobre el Medioambiente y el Desarrollo en el año 2012, en el capítulo sobre "Información para la adopción de decisiones" se menciona expresamente la importancia de los Sistemas de Información Geográfica y la necesidad de armonizar los datos.

En dicho documento se definen las acciones que deben tomar los países y las organizaciones para lograr un desarrollo sostenible, muchas de las cuales están relacionadas directa o indirectamente con la información geográfica.

Todos estos principios vuelven a aparecer en la orden ejecutiva *129061* del gobierno de *EE.UU.* en el año 1994. En esta orden el presidente Bill Clinton crea la llamada *National Spatial Data Infrastructue* (NSDI) de los **EE.UU**.

Esta orden define la creación de repositorios y catálogos de datos (*National Geospatial Data Clearinghouse*), la obligación de acceso público a la información geoespacial, la creación de estándares para alcanzar los objetivos, etc.

Además otro paso muy importante fruto de estas acciones fue la creación en 1994 del OGC (*Open Geospatial Consortium*). Consorcio internacional sin ánimo de lucro que se posicionará como referente mundial en la creación de estándares para lograr la interoperabilidad de la información geográfica (Llario, s.f.).

2.2. DEFINICIÓN DE IDE

Una IDE (Infraestructura de Datos Espaciales), conocida también como Infraestructura de Información Geográfica (IG), es un sistema informático integrado por un conjunto de datos y servicios (descritos a través de sus metadatos) que son gestionados a través de Internet, conforme a estándares que regulan y garantizan la interoperabilidad de sus datos y a acuerdos políticos que permiten que un usuario, a través de un simple navegador, pueda localizar, visualizar, acceder y combinar la IG en función de sus necesidades.

Una IDE se materializa en un Geoportal, es decir, una página web que da acceso a los servicios de la IDE.

2.3. INTEROPERABILIDAD, ORGANISMOS DE ESTANDARIZACIÓN Y NORMAS

Para lograr la interoperabilidad, es decir, crear un sistema escalable, son necesarios ciertas normas y estándares.

2.3.1. INTEROPERABILIDAD E INTEROPERABILIDAD GEOGRÁFICA

La definición de interoperabilidad en un sistema informático aparece en la norma *ISO 19119* como (Llario, s.f.):

"Capacidad para comunicarse, ejecutar programas, o transferir datos entre varias unidades funcionales de forma que se requiera del usuario poco o ningún conocimiento de las características únicas de esas unidades."

La misma norma *ISO 19119* define la "interoperabilidad geográfica" como (Llario, s.f.):

"Capacidad de los sistemas de información de 1) intercambiar libremente toda clase de información espacial sobre La Tierra y sobre los objetos y fenómenos sobre y por debajo de la superficie terrestre; 2) cooperativamente, sobre redes, ejecutar aplicaciones capaces de manipular tal información."

2.3.2. NORMA Y ESTÁNDAR

Las normas y estándares además de necesarias para alcanzar la interoperabilidad, ayudan a reducir costes a la hora de la producción, conceptualización y diseño de un producto.

Una **norma** es todo documento que armoniza aspectos técnicos de un producto, servicio o componente, definido como tal por algún organismo oficial de normalización. Las normas también se conocen como estándar de jure (o iure). El *Comité Técnico 211* de la *ISO* tiene como objetivo el desarrollo normas en el área de la Información Geográfica. Dichas normas se agrupan en un conjunto que se denomina familia *ISO 19100*. (Llario, s.f.)

Un **estándar** es cualquier documento técnico o práctica que sin ser norma, está totalmente aceptado por el uso y cumple una función similar a la de una norma. La diferencia es que un estándar no ha sido definido por un organismo oficial de normalización. También es conocido como estándar de facto. Los estándares referentes a la Información Geográfica suelen ser creados por Consorcios formados por Agencias Cartográficas Nacionales, Universidades y Empresas como el *Open Geospatial Consortium* (OGC). (Llario, s.f.)

Los estándares suelen ser generados más rápidamente para su adaptación al mercado. También puede ser que un estándar de facto termine convirtiéndose en norma. Un ejemplo referente a la información geográfica es el servicio *WMS*, estándar propuesto por el *OGC*, que terminó convirtiéndose tras unos años en la norma *ISO 19128 "Interfaz de servidor de mapas"*, así como otros estándares del *OGC*.

A veces se puede hacer confuso el uso de la palabras norma y estándar. Generalmente siempre nos referimos a un estándar, algunos creados por organismos de normalización y otros no. Discriminamos por norma aquellos estándares que además se rigen por un documento *ISO, CEN, AENOR* o de otro organismo de normalización oficial.

Existen otro tipo de documentos que pueden ser tratados como una **recomendación**, es decir, directrices que no son de obligado cumplimiento pero que tratan de mejorar las prácticas y usos dentro de una comunidad. Por ejemplo, el perfil *NEM* es una recomendación del *IGN*.

Finalmente, las **especificaciones** son una forma de referirse a documentos técnicos que detallan un producto o servicio con toda la información necesaria para su producción. Algunas pueden ser adoptadas como normas o estándares. Un ejemplo es la especificación del lenguaje *ECMAScript 2016* que se rige por el estándar *ECMA-262*, el documento donde el *OGC* detalla el servicio *WFS* es una especificación que además es un estándar creado por el *OGC* y en este caso también llegó a convertirse tras un proceso más largo en norma *ISO (ISO 19128:2005)*.

2.3.3. ORGANISMOS DE ESTANDARIZACIÓN

A continuación se verán los organismos de estandarización más importantes en el ámbito de la Información Geográfica.

2.3.3.1. OPEN GEOSPATIAL CONSORTIUM (OGC)

El *Open Geospatial Consortium* (*OGC*) es un consorcio internacional sin ánimo de lucro originado en 1994 y que consta con cerca de 500 miembros entre organizaciones comerciales, gubernamentales, universidades, pequeñas empresas, etc. (Llario, s.f.)

Los estándares que se desarrollan se basan en consensos voluntarios sin ánimo de lucro, ya que su misión es promover el desarrollo, técnicas y estándares de sistemas abiertos en el ámbito de la Información Geográfica.

El OGC dispone de muchos estándares y especificaciones, algunos correspondientes a servicios y otro a lenguajes o formatos utilizados en la Información Geográfica.

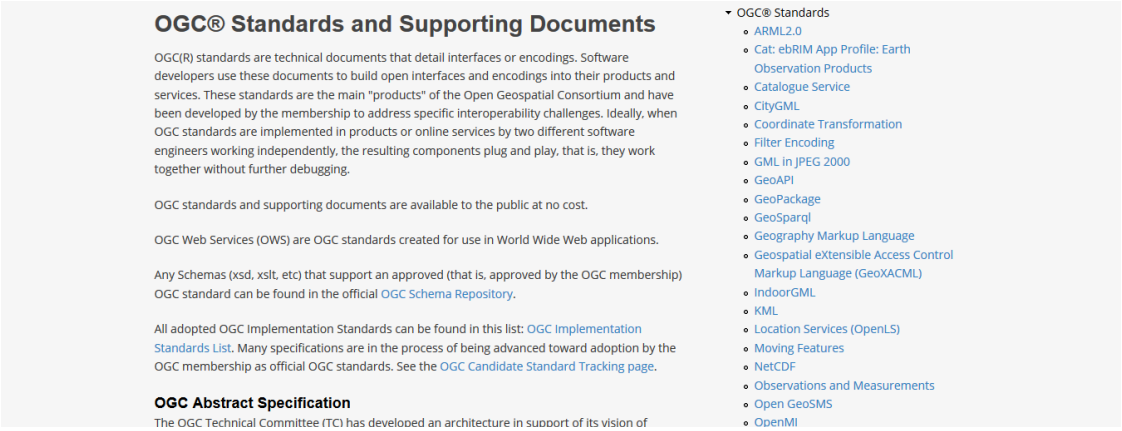


Imagen 1 - Algunos de los estándares del OGC.(http://www.opengeospatial.org/standards).

Además provee test de conformidad para cada uno de sus servicios y estándares, de manera que se puede conocer si un producto que implementa alguna de sus especificaciones es conforme con ellas.

2.3.3.2. LA FAMILIA DE NORMAS ISO 19100

ISO es la organización encargada de elaborar normas internacionales en ámbitos científicos, tecnológicos, etc. Dentro de dicha organización los Comités Técnicos (*Technical Commitee*) son los encargados de elaborar dichas normas.

El proceso de elaboración de una norma *ISO* debe pasar por varias fases antes de considerarse norma internacional final. Las fases son las siguientes (Llario, s.f.):

- Se elabora el documento *Working Draft (WD)* por un grupo del *TC*, el cual es un documento técnico preliminar de un posible documento en el futuro.
- Se elabora el documento *Commitee Draft (CD)*, el cual es un borrador más completo que se distribuye a los miembros del TC para su valoración y posibles sugerencias.
- Se obtiene el *Draft International Standard (DIS)* del proceso anterior y se distribuye a todos los países miembros de *ISO* para votación y comentarios.
- Tras aprobar las modificaciones el documento *DIS* se transforma en *FDIS (Final Draft International Standard)*. Este documento ya se puede utilizar de forma profesional por las compañías, organizaciones, etc.
- El *FDIS* se convierte en *IS (Norma Internacional)*, documento editorialmente corregido y admitido tras una votación final.

El **Comité Técnico 211** (*ISO/TC211*) es el que trabaja en el ámbito de la información geográfica y produce las normas utilizadas en las IDEs, bases de datos espaciales, servicios de IG, etc.

2.4. MARCO LEGAL DE LAS IDES

2.4.1. INSPIRE

La directiva **INSPIRE** proporciona el marco legal de las IDE para los estados miembro de la unión Europea, directiva *2007/2 de 14 de marzo de 2007*. Consiste en una serie de normas de ejecución (reglamentos, documentos técnicos) donde se especifica (Llario, s.f.):

- Las bases para la producción de cartografía, elección de modelos cartográficos adecuados y la estructuración de la IG.
- Los metadatos que debe contener la IG.
- Los servicios que debe ofrecer una IDE y su grado de fiabilidad.
- Los servicios que deben ser gratuitos y las restricciones que puedan tener.
- Que los estados miembros deben tomar medidas para la interoperabilidad de los datos entre sus organismos públicos y entre el resto de países de la UE.
- Que los estados miembros deben presentar informes periódicos para ser evaluados.
- El plazo que tienen los países miembros para implementar todo lo anterior.

Existen diferentes documentos y normas legales dentro de la Unión Europea (Llario, s.f.):

- **Directiva**: Establece un plazo a los estados miembros para la consecución de resultados u objetivos concretos en un plazo determinado.
- **Reglamentos**: De obligado cumplimiento para los estados miembro y sus ciudadanos. Es una norma de aplicación directa, es decir, no hace falta su transposición a Ley.
- **Otros**: Como guías técnicas que muestran detalles de la implementación de los reglamentos.

El documento de la directiva *INSPIRE* consta de 14 páginas, de las cuales 7 son artículos y las demás corresponden los anexos y la introducción. Los anexos de la directiva son tres y muestran tres listas de temáticas donde clasificar la información geográfica.

Imagen 2 - Contenidos de los anexos I, II y III de INSPIRE.

La directiva se puede dividir en (Llario, s.f.):

- **Consideraciones iniciales**: Se comenta el porqué de la directiva y las principales pautas y objetivos generales.
- **Capítulo I: Disposiciones generales.** Define conceptos como Infraestructura de IG, autoridad pública, etc. También se define a qué conjuntos de datos se aplica la directiva.
- **Capítulo II: Metadatos.** Define la obligación de crear metadatos y qué deben contener de forma general. También se definen los plazos de creación y se apunta a la futura creación de las correspondientes normas de ejecución de metadatos.
- **Capítulo III: Interoperabilidad de conjuntos y servicios de datos espaciales.** Hace referencia a la aparición de las normas de ejecución correspondientes y qué deberán contener de forma general para lograr la armonización de los conjuntos y servicios de datos espaciales. Especifica los plazos para alcanzar dicha armonización.
- **Capítulo IV: Servicios de Red.** Establece los servicios mínimos que deberán tener las IDE de los estados miembros y qué servicios serán gratuitos y de pago.
- **Capítulo V: Puesta en común de los datos.** Obliga a los estados miembros a adoptar los acuerdos necesarios para la puesta en común de los conjuntos de datos y servicios entre sus administraciones. También plantea las condiciones para el intercambio de IG con otros estados miembros.
- **Capítulo VI: Coordinación y medidas complementarias.** Obliga a los estados miembros a crear comisiones para gestionar las contribuciones de usuarios, productores, etc. y designar un punto de contacto para la comunicación con Europa.
- **Capítulo VII: Disposiciones finales.** Obliga a los estados miembros a crear informes periódicos sobre la aplicación de la directiva que entregará al parlamento y la comisión Europea.

Tras la aprobación de la directiva se instó a cada país miembro a cumplir esta mediante la transposición a Ley para todos los países miembros. En España dicha transposición es la ley **LISIGE**, *Ley 14/2010 del 5 de Julio del año 2010*.

2.4.2. LISIGE

El marco legal de las IDEs en España, al ser un estado miembro de la UE viene marcado por la directiva INSPIRE y por su transposición a la legislación española *(Ley 14/201* **LISIGE** o "*Ley sobre las Infraestructuras y los Servicios de Información Geográfica de España*"). (Llario, s.f.)

La ley **LISIGE** se basa en todo lo anteriormente nombrado sobre *INSPIRE* y además incluye unos capítulos sobre la IDE de España, el sistema cartográfico nacional, una reestructuración de los anexos, etc.

A continuación se muestra una lista de los puntos complementarios que añade *LISIGE* y como se estructuran (Llario, s.f.):

- **Capítulo I**: Incluye las definiciones del **anexo I** de *INSPIRE* y además define los términos de IDE e IG de forma más específica que *INSPIRE*, define Información Geográfica de Referencia, Datos Temáticos Fundamentales y Datos Temáticos Generales y otros términos.
- **Capítulo II**: Se establecen las competencias del *Consejo Superior Geográfico* (*CSG*), como coordinador y operador de la *IDEE* y punto de contacto con la comisión Europea. Define las funciones del Geoportal de la *IDEE* que implementará los servicios establecidos por *INSPIRE*. Incluye las normas de ejecución de *INSPIRE* sobre los conjuntos y servicios de datos (**Capítulo III** de *INSPIRE*).
- **Capítulo III**: Contiene información sobre los **capítulos II, III y IV** de *INSPIRE*.
- **Capítulo IV**: Define el Geoportal de la IDE de la Administración General del Estado (*IDE AGE*) y sus funciones, así como las funciones del Instituto Geográfico Nacional respecto a la constitución y mantenimiento de dicho Geoportal.
- **Capítulo V**: Se instituye el Sistema Cartográfico Nacional (*Real Decreto 1545/2007*), como marco de coordinación de la actividad cartográfica en España. También regula según las escalas qué organismos deben de producir la cartografía en España.

2.5. COMPONENTES DE UNA IDE

Como se ha comentado anteriormente, una IDE es un sistema informático que incluye:

- Datos geográficos
- Servicios
- Metadatos

2.5.1. DATOS GEOGRÁFICOS

Se entiende como dato geográfico cualquier tipo de fenómeno susceptible de ser geográficamente referenciado. Se pueden clasificar en función del tipo de información que están representando (Llario, s.f.):

- **Datos de Referencia:** Información Geográfica fundamental utilizada como base para referenciar cualquier otro conjunto de datos temáticos.
- **Datos Temáticos**: Datos obtenidos a partir de información geográfica de referencia singularizando o desarrollando algún aspecto concreto de la información contenida, sin añadir información específica adicional.

La directiva *INSPIRE* especifica que se aplicará a los conjuntos de datos espaciales que cumplan las siguientes condiciones:

- Se refieran a una zona sobre la que un Estado miembro tenga y/o ejerza jurisdicción.
- Estén en formato electrónico.
- Obren en poder de alguna de las partes que figuran a continuación:
  - Una autoridad pública, después de ser producidos o recibidos por una autoridad pública, o sean gestionados o actualizados por dicha autoridad y estén comprendidos en el ámbito de sus actividades públicas.
  - Un tercero al que se hubiera facilitado el acceso a la red con arreglo a lo dispuesto en el artículo 12 de la Directiva.
- Traten de uno o más de los temas recogidos en los **Anexos I, II o III**.

2.5.2. METADATOS

El término "*metadatos*" no tiene una definición exacta, pero podemos referirnos a ellos como "*información de los datos*", "*datos sobre los datos*", etc. Los recursos contenidos en una IDE (datos y servicio) deben tener metadatos ya que aportan información alfanumérica sobre estos como el sistema de referencia, fecha de creación, autor de los datos, calidad de los datos, etc.

Los metadatos aportan unos claros beneficios (Llario, s.f.):

- Ayudan a los motores de búsqueda a localizar nuestros recursos.
- Mejoran la gestión de los recursos.
- Define la información de un recurso y por tanto hace posible el entendimiento de este por otras personas.
- Ayudan a compartir recursos con otros organismos.
- Nos dan información acerca del título y la descripción de los datos, fecha de creación y actualización, personas y/u organismos creadores de los recursos, zona geográfica de los datos, proceso de producción cartográfica utilizado, formato, etc.

Existen normas internacionales (*ISO*) que permiten realizar los metadatos de datos, servicios y otros recursos. *ISO (International Organization for Standarization)* es un organismo de estandarización internacional no gubernamental. Las normas *ISO* también tienen su equivalente traducción al Español según el organismo de normalización *AENOR* (normas *UNE*).

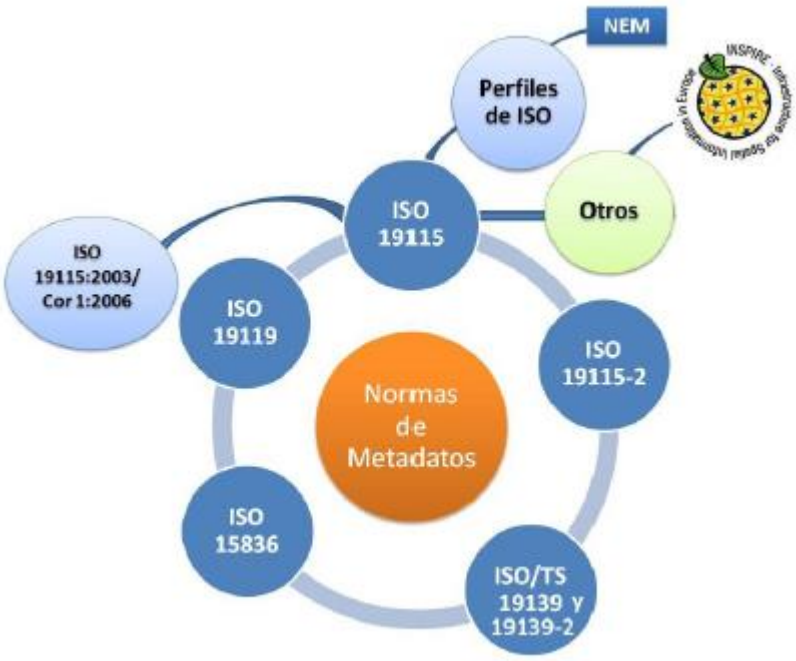


Imagen 3 - Normas de metadatos. (Llario, s.f.)

A continuación se destacan algunos de los aspectos más importantes de las normas anteriores.

2.5.2.1. ISO 19115

La norma **ISO 19115:2003 <<Geographic Information – Metadata>>** fue aprobada en 2003, adoptada como Norma Europea por **CEN/TC287 (EN ISO 19115)** y finalmente traducida como una norma española **(UNE EN ISO 19115).** (Llario, s.f.)

Define el modelo requerido para describir información geográfica y servicios. Proporciona información sobre la identificación, la extensión, la calidad, el modelo espacial y temporal, la referencia espacial y la distribución de los datos geográficos digitales.

Aunque esta norma se aplica para datos geográficos digitales también puede extenderse a distintos tipos de datos no digitales como mapas, cartas y documentos de texto, incluso a datos no geográficos.

Además define (Llario, s.f.):

- Las secciones de metadatos obligatorios y condicionales, entidades de metadatos y elementos de matdatos.
- El conjunto mínimo de metadatos requeridos para el correcto funcionamiento en las aplicaciones de metadatos.
- Los elementos de metadatos opcionales.
- Un método para crear extensiones de metadatos para adaptarse a necesidades más específicas.

Algunas características de la *ISO 19115* son (Llario, s.f.):

- Es una norma compleja, con un total de 409 elementos y 27 listas controladas.
- La norma dispone de un Core (núcleo) que es el número mínimo de elementos a rellenar de forma obligatoria.
- Permite catalogar la información geográfica independientemente del formato.
- Permite catalogar la información geográfica independientemente del nivel de detalle (serie cartográfica, hoja de una serie, capa cartográfica, fenómeno geográfico, colección de fenómenos, servicios, etc.).

El contenido de la norma se presenta dentro de paquetes *UML* y cada paquete tiene su diccionario de datos (Llario, s.f.):

- Cada paquete contiene una o más entidades (*Clases UML*)
- Las entidades contienen elementos (*Atributos de Clase en UML*)
- Las entidades se puede relacionar con una o más entidades.

Algunos inconvenientes de esta norma son (Llario, s.f.):

- Poca existencia de elementos obligatorios.
- Poca claridad en la condición de algunos elementos condicionales.
- Poco detalle en cuanto metadatos sobre información geográfica en formato ráster, malla o de fotogrametría. La **ISO 19115-2** añade nuevos elementos para solventar dicha falta de detalle.

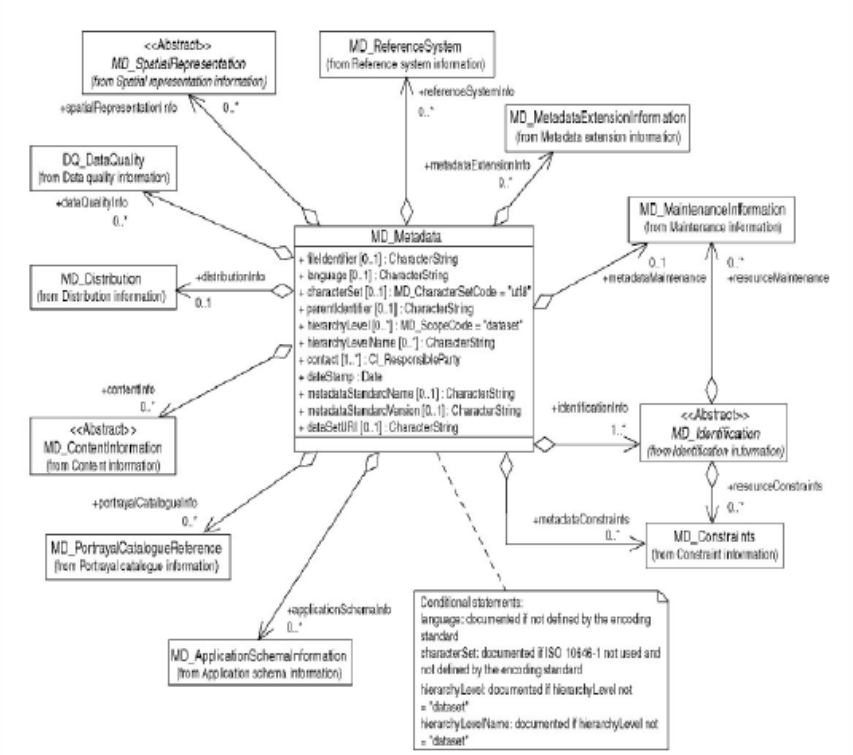


Imagen 4 - Cuadro de la norma ISO 19115:2003. Distribución de paquetes. (Llario, s.f.)

Al ser una norma muy extensa se define un núcleo, el cual debe ser utilizado por los perfiles de esta norma, que contiene una serie de metadatos mínimos requeridos para catalogar un conjunto de datos.

2.5.2.2. ISO 19115-1

Esta norma fue aprobada en 2014 y viene a sustituir a la *ISO 19115:2003* y su corrigendum del 2006. Las principales características son (Llario, s.f.):

- Incluye los metadatos para los servicios de *la ISO 19119*.
- Se eliminan los metadatos referentes a la calidad (DQ_DataQuality) haciendo referencia a la norma **ISO 19157** donde se definen.
- Mantiene grandes cantidades de datos descritos según la norma *19115:2003* y añade nuevos.
- Sirve como repositorio para nuevos metadatos según nuevas especificaciones de productos.
- No crea nuevos elementos obligatorios, para asegurar la compatibilidad con la anterior revisión.

2.5.2.3. ISO 19119-2

La norma ISO 19115-2:2009 **<<Información Geográfica - Metadatos - Extensión para imágenes y datos malla>>** amplía la Norma *ISO 19115:2003* definiendo los elementos de metadatos adicionales necesarios para describir las imágenes geográficas y los datos malla de una forma más específica.

Esta norma se ha desarrollado a partir de la norma ISO *19115:2003*, del Informe Técnico *ISO/TR 19121* y otras normas relacionadas, que contemplan metadatos para imágenes y datos malla. La Norma *ISO 19115-2:2009*, utilizada junto a la *ISO 19115:2003*, permite describir de forma completa los distintos procesos de producción por los que se obtienen imágenes y datos malla y las características de estos datos y contiene el modelo de extensiones de metadatos necesarias para documentar la información sobre la calidad de los datos, su representación espacial, su contenido y la forma de adquisición de la información. (Llario, s.f.)

2.5.2.4. ISO 19119

La norma ISO 19119:2005 **<<información Geográfica - Servicios>>** ha sido desarrollada por el Comité Técnico *ISO/TC 211* con el fin de proporcionar un espacio de trabajo a los desarrolladores para crear aplicaciones que permitan a los usuarios acceder y procesar datos geográficos procedentes de diversas fuentes.

Los servicios son aplicaciones que ofrecen una funcionalidad en la red a través de una interfaz y pueden disponer de diferentes niveles de funcionalidad.

Se utilizará esta norma para realizar los metadatos de los servicios típicos de una IDE como *WMS, WFS, WCS, WMTS*, etc., aunque la norma *ISO 19119* permite realizar metadatos para cualquier servicio (Llario, s.f.):

- Servicios geográficos de interacción humana
- Servicios geográficos de gestión de modelo/información
- Servicios geográficos de gestión de flujos de trabajo/tareas
- Servicios geográficos de procesamiento
- Servicios geográficos de comunicaciones

La norma *ISO 19115-1* contiene los metadatos de los servicios de la norma *ISO 19119*. Pese a esto todavía se sigue usando la *ISO 19119* debido a que la *ISO 19115-1* todavía no se aplica.

2.5.2.5. ISO 19139 E ISO 19139-2

La norma *ISO/TS 19139:2007* define el esquema *XML* de implementación para las normas *ISO 19115 e ISO 19119*. Es decir, define el "lenguaje de intercambio" para almacenar los metadatos de una forma estándar. (Llario, s.f.)

Esta normativa incorpora una codificación basada en esquemas *XML* para describir, validar e intercambiar metadatos relativos a la Información Geográfica, además de cumplir las reglas de codificación definidas por *ISO 19118* y añadir detalles específicos para crear los *XML* derivados de los modelos *UML* definidos por *ISO 19115*.

El lenguaje *XML* es un lenguaje de etiquetas que se utiliza para crear documentos con información estructurada. La sintaxis de los archivos de metadatos sigue esquemas *XML*. El propósito de un esquema es definir los componentes válidos de un documento XML como elementos, atributos, elementos hijos, orden y número de los elementos, tipos de datos, valores por defecto de los elementos, etc.

Más recientemente apareció la norma ISO/TS 19139:2 **"Geographic information --Metadata - XML schema implementation - Part 2: Extensions for imagery and gridded data"** que define el esquema *XML* de implementación para la *ISO 19115-2*.

Esta norma se utiliza para crear *XML* derivados a partir de los Modelos *UML* definidos en las normas *ISO 19115-2*.

2.5.2.6. DUBLIN CORE

La iniciativa *Dublin Core Metadata (DCMI)* es una organización abierta que fomenta la participación en el diseño de los metadatos. El esquema *Dublin Core* es un pequeño conjunto de términos (vocabulario) utilizado para describir tanto recursos web como cualquier recurso físico (libros, CDS, etc.) y en la actualidad es la norma de metadatos más utilizada debido a su simplicidad. (Llario, s.f.)

El conjunto de elementos de metadatos de *Dublin Core* consta únicamente de 15 elementos y se pueden utilizar para describir cualquier objeto de información.

2.5.2.7. PERFIL NEM

Cada país, región u organismo además puede elaborar su propio perfil de metadatos, es decir, un subconjunto de la norma *ISO 19115* que simplifica la norma *ISO* agiliza la creación de los metadatos.

En España se creó el perfil *NEM* (*Núcleo Español de Metadatos*) debido a la necesidad de adaptar la extensa norma internacional sobre los metadatos de la IG (*ISO 19115*).

El perfil *NEM* está formado por los siguientes elementos (Llario, s.f.):

- 22 elementos de la Norma Internacional *ISO 19115* pertenecientes a su núcleo (Core):
  - 7 elementos que son obligatorios y que se recomienda incluir como mínimo.
  - 15 elementos que se dividen en opcionales y condicionales.
- 3 elementos que se encuentran en el estándar *Dublín Core*.
- 3 elementos adicionales pertenecientes a la Norma *ISO 19115*.
- 2 elementos pertenecientes a la Norma *ISO 19115* incluidos en el *NEM* por su uso en la Directiva Europea *Marco del Agua (WFD)*.
- Otros elementos adicionales pertenecientes a la Norma *ISO 19115* que profundizan en la calidad.

En el perfil *NEM* se define la calidad como obligatoria, y en caso de no poder o no tener medidas realizadas de ella, la organización u organismo encargado de crear los metadatos podrá incluir las expresiones "*no disponible*"," *no aplicable*".

Una de las premisas básicas que se tuvieron en cuenta en la definición de la primera versión de *NEM* en 2004 fue considerarlo como un perfil abierto, es decir, estaba sujeto a posibles modificaciones futuras según surgieran documentos normativos o legales relacionados con metadatos, que fuera preciso tener.

2.5.3. SERVICIOS

Los servicios mínimos que debe ofrece una IDE pueden venir establecidos por el marco legal del país en cuestión, como es el caso de España, en donde la directiva INSPIRE establece los servicios mínimos que debe tener una IDE en un país miembro de la UE. (Llario, s.f.)

En caso de que el país no este regido por ningún marco legal o recomendación que establezca estos requisitos de una IDE el sitio más adecuado e internacional que utilizar como base es la asociación "*Global Spatial Data Infrastructure Association*" (*GSDI*).

Se consideran por tanto servicios mínimos de una IDE:

- **Servicios de visualización**

Es el servicio de visualización de mapas que los usuarios utilizarán para visualizar las imágenes de los mapas de forma remota. Algunos de los estándares para la implementación de este tipo de servicios son el *WMS* o el *WMTS*, este último acelera la carga de los mapas en los clientes y disminuye la carga de procesamiento en el servidor.

- **Servicios de localización y descubrimiento**

Mediante este servicio los metadatos de los recursos cartográficos serán descubribles, es decir, serán visibles a través de Internet. Estos recursos de metadatos se llaman catálogos y cualquier buscador de metadatos se podrá conectar a ellos utilizando el estándar *CSW*.

Algunos servicios no obligatorios más utilizados son:

- **Servicios de Nomenclátor o Gazzetter**

El Instituto Geográfico Nacional (*IGN*) de España aconseja la creación de servicio de nomenclátor, aunque veremos que la directiva *INSPIRE* no menciona nada al respecto y muchas IDE en España no incorporan este servicio. Además considerando que su implementación con las actuales herramientas software no es inmediata en este curso no lo vamos a implementar de forma práctica.

Se puede implementar utilizando varios modelos pero quizás la forma más habitual es mediante la utilización de un perfil del estándar *WFS* que se llama *WFS-G (Gazetter).*

- **Servicios de descarga**

Estos servicios deben permitir generar copias de los recursos cartográficos o de partes de ellos. En algunas IDE se implementa simplemente ubicando enlaces a las fuentes de datos originales de los recursos cartográficos por ejemplo ficheros *shape*.

Lo aconsejable es utilizar los estándares del *OGC* que además de permitir la descarga en el caso vectorial (*WFS*) también añaden la posibilidad de editar las geometrías online. Se aconseja utilizar los estándares *WFS* (datos vectoriales) y *WCS* (datos ráster).

- **Servicios de procesamiento**

Permite a la IDE ofrecer servicios de análisis espacial o geoprocesamiento. El estándar del OGC utilizado es el *Web Processing Service* (*WPS*).

En la web oficial del *OGC* se pueden obtener las especificaciones de la implementación de los distintos estándares para los servicios relacionados con la IG. Es la ventaja de estar usando estándares abiertos.

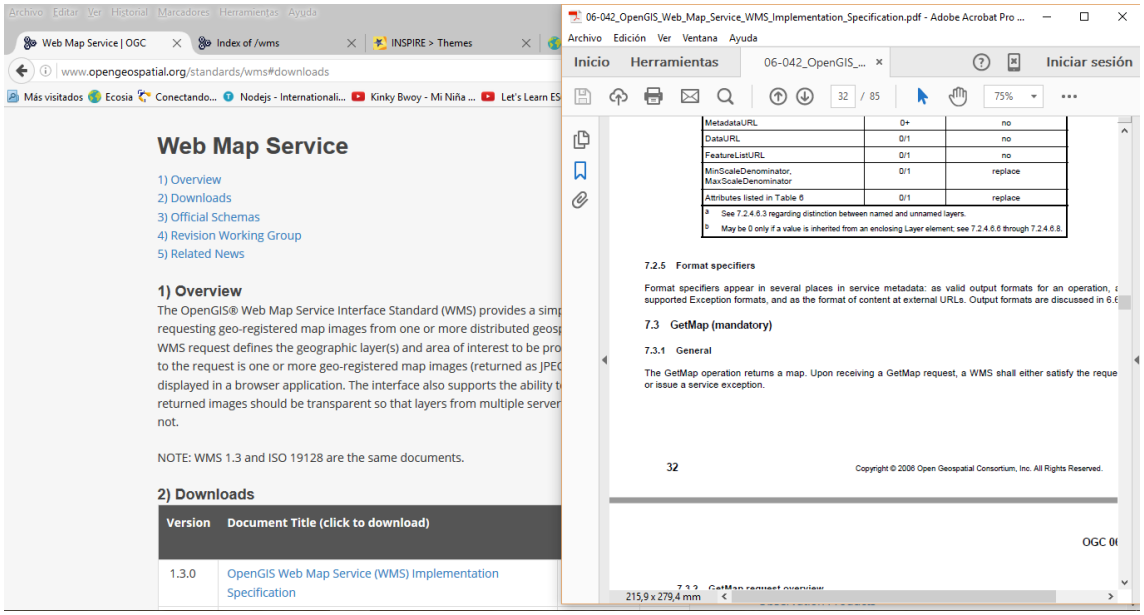


Imagen 5 - Web oficial del OGC en la sección del estándar WMS y a la derecha su correspondiente documento de especificaciones de implementación.

3. SERVIDORES Y APLICACIONES WEB

Un servidor es una aplicación en ejecución (*software*) capaz de atender las peticiones de un cliente y devolverle una respuesta en concordancia. Actualmente el mundo de los servidores y las aplicaciones web está en continuo cambio y evolución. Numerosos *frameworks* están apareciendo para facilitar la tarea de diseñar e implementar un servidor HTTP sobre el que alojar una aplicación web.

Los servidores operan a través de una arquitectura cliente-servidor, la cual se detalla a continuación.

3.1. ARQUITECTURA CLIENTE-SERVIDOR

El modelo o arquitectura cliente-servidor es un sistema distribuido en el cual un cliente solicita un recurso o servicio a un servidor. (Llario, s.f.)

Los clientes y los servidores suelen comunicarse a través de un entorno de red y residen normalmente en sistemas separados.

Algunos ejemplos de aplicaciones que utilizan un modelo cliente-servidor son los gestores de correo electrónico, una aplicación de impresión remota y por supuesto Internet cuando visualizamos por ejemplo cualquier página web.

Los clientes y los servidores intercambian mensajes siguiendo un patrón de petición-respuesta (*request-response*):

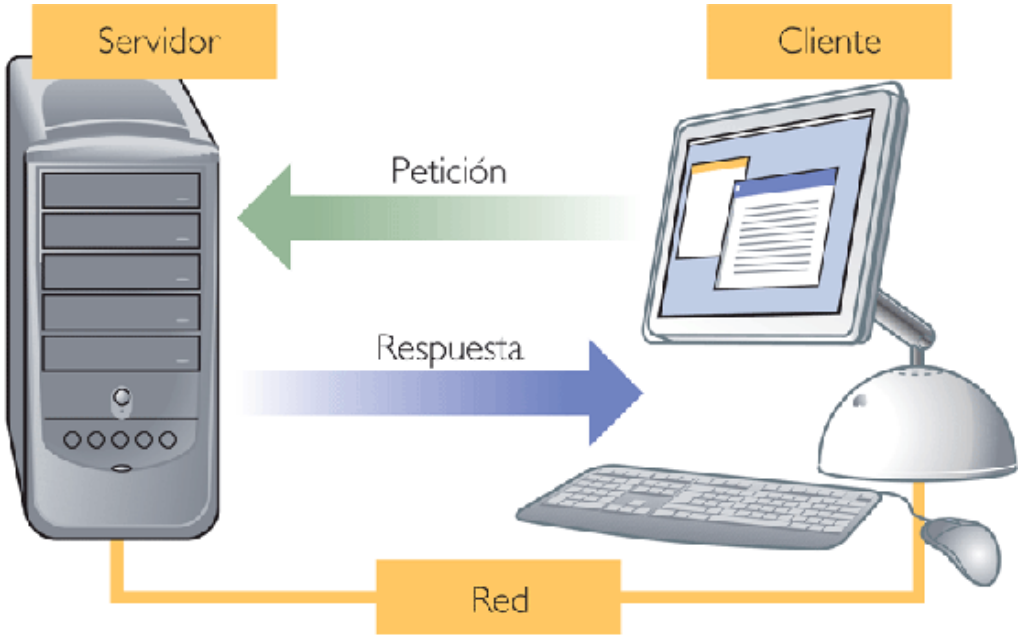


Imagen 6 - Flujo de comunicación en una arquitectura Cliente-Servidor.

Un cliente envía o solicita una petición, el servidor atiende dicha petición y genera una respuesta que devuelve al cliente.

3.1.1. COMUNICACIÓN A TRAVÉS DE LA RED

Para poder realizar esta comunicación entre cliente y servidor se necesita un lenguaje común y se deben seguir unas reglas para que ambos sepan cómo comportarse.

Estos lenguajes y reglas de comunicación están definidos en protocolos de comunicación necesarios para intercambiar datos entre ordenadores. El protocolo más ampliamente utilizado es el *Internet Protocol Suite*, que consiste en la pareja de los protocolos *TCP (Transmission Control Protocol) / IP (Internet Protocol).* Este tipo de protocolos proporcionan una manera fiable de transmitir paquetes de datos sobre la red. (Llario, s.f.)

El protocolo ***TCP/IP*** proporciona una conectividad entre diferentes puntos de la red especificando como los datos deben ser formateados, direccionados, transmitidos y recibidos y ofrece un sistema de cuatro capas, a través de las cuales se comunican los datos de un ordenador a otro desde el nivel más bajo (más cercano al hardware de la red) hasta el nivel más alto (más cercano al software). Estas cuatro capas, los protocoles que se usan en cada una de ellas y su relación con el *modelo OSI* se describen en la siguiente imagen:

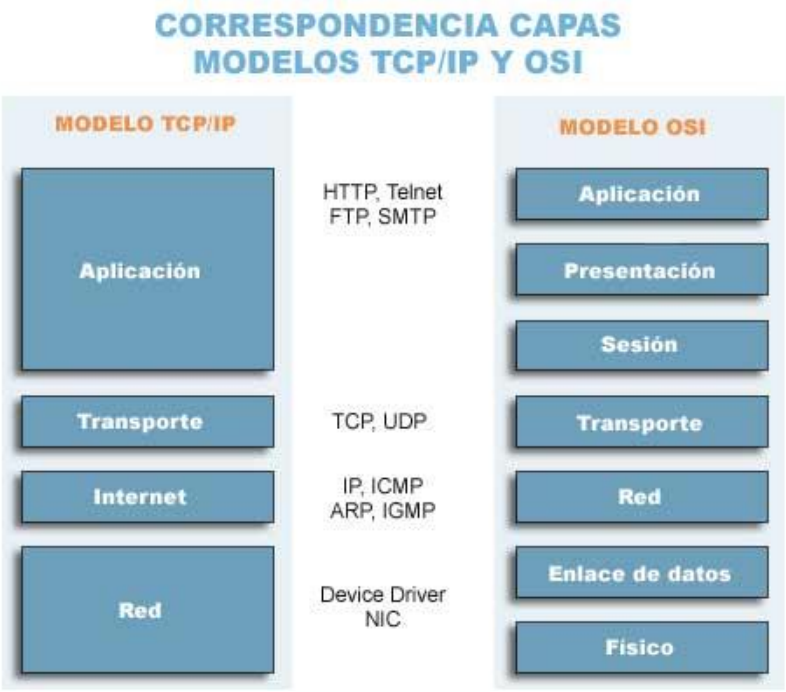
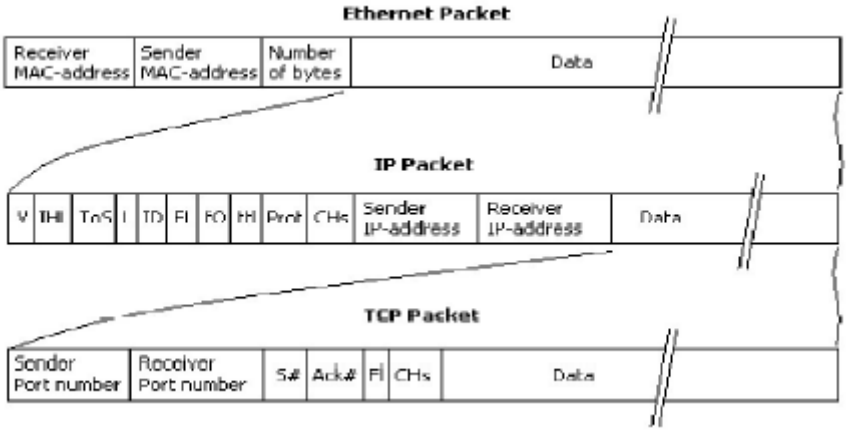


Imagen 7 - Capas del modelo de comunicación TCP/IP con sus protocolos más utilizados, y el modelo OSI.

El modelo de *Interconexión de Sistemas Abiertos* (*Open System Interconnection o modelo OSI*) es un modelo de referencia para los protocolos de la red que se basa en una arquitectura de capas, como en este caso el protocolo *TCP/IP*.

La siguiente figura muestra como el protocolo en el nivel más bajo Ethernet envía información (*data*) entre dos direcciones físicas (*MAC*) de dos dispositivos (*Receiver MAC-address y Sender MAC-address*). La dirección *MAC* es un identificador único de 42 *bits* que se asocia a una tarjeta o dispositivo de red.



En el siguiente nivel se aprecia como en la sección data de *Ethernet* se codifica un paquete de información *IP*, compuesto a su vez por una sección de datos (*data*), algunos indicadores (*V, IHL*, etc.) y las direcciones *IP* origen y destino entre los dispositivos de red.

La dirección IP identifica los dispositivos de red de un equipo como un *router*, una impresora en red, una tarjeta de red, un dispositivo *WIFI*, etc.

El siguiente nivel, el de transporte, codifica el paquete de información *TCP*, compuesto a su vez por una sección de datos (*data*), algunos indicadores y el puerto de origen y destino.

El puerto es un valor numérico entre *0 y 65535* que identifica la conexión entre el cliente y el servidor, es decir identifica la comunicación entre diferentes aplicaciones cliente-servidor.

El siguiente y último nivel es el de aplicación (*application layer*). Es usado por la mayoría de las aplicaciones para proporcionar servicios al usuario o intercambio de datos sobre la red y se apoya en los protocolos de nivel inferior. El protocolo de nivel de aplicación más conocido es el *HTTP (Hyper Text Transjer Protocol o Protocolo de Transferencia de Hipertexto)* que permite el medio para navegar por páginas web. También otros muy conocidos son los protocolos de aplicación *FTP, DNS, SSH, POP, IMAP*, etc. (Llario, s.f.)

- **La dirección IP**

La IP consiste en un número de 32 *bits* formada por 4 octetos (4 números entre O y 255). Esta versión es también conocida como *IPv4*. Existe otra versión originada con el crecimiento masivo de dispositivos conectados a la red para generar un rango más amplio de direcciones *IP*. Por ello se creó otro estándar denominado *IPv6* (128 *bits*).

- Aspecto de una dirección *IPv4*: **192.168.1.14**
- Apecto de una dirección *IPv6* : **2004:9f0d:74x7::9p3a:74f7**

Existen dos tipos de direcciones *IP* (Llario, s.f.):

- **IP estáticas**: la dirección *IP* es fija y no cambia a lo largo del tiempo.
- **IP dinámicas**: la dirección es asignada automáticamente (utilizando generalmente el protocolo *DHCP*). Esta dirección no es permanente y suele cambiar al reconectar internet cada cierto tiempo y es la que normalmente se contrata en los hogares.

3.1.2. TIPOS DE CLIENTES Y SERVIDORES

3.1.2.1. CLIENTE

Un cliente es un componente hardware o software que accede a un servicio disponible por un servidor.

Por ejemplo, los navegadores de internet son clientes que se conectan a servidores web, los cuales devuelven unas páginas web que son visualizadas por estos clientes. (Llario, s.f.)

CLIENTE LIGERO	CLIENTE PESADO
La carga de procesamiento se realiza mayoritariamente por el ordenador central (servidor).	El procesamiento se realiza mayoritariamente en el cliente, la carga de procesamiento no recae al menos totalmente en el servidor.
Ocupan poco espacio y/o requieren un hardware poco potente.	Requisitos en cuanto a hardware más elevados y generalmente un proceso de instalación más elaborado.
Ejemplo: navegador web.	Puede tener capacidad para trabajar offline.
	Ejemplo: QGIS, Google Earth.

La premisa que se toma para discriminar entre un cliente pesado y otro ligero atiende a donde se realiza la carga de procesamiento (Cliente o Servidor). En el caso de los navegadores web, a diferencia de tiempos pasados, son un componente software complicado y que realiza en ocasiones un procesamiento elevado.

3.1.2.2. SERVIDOR

Un servidor responde a las peticiones del cliente a través de la red. Generalmente un servidor puede responder a múltiples clientes y no suele interactuar directamente con el usuario.

La separación entre el cliente y el servidor es lógica, es decir, de funcionalidad. Normalmente ambos están situados en diferentes máquinas aunque no necesariamente.

Algunos ejemplos de servidores son: servidores de archivos, servidores de bases de datos, servidores de correo, servidores web o también llamados servidores *HTTP*. (Llario, s.f.)

3.1.3. ARQUITECTURA CLIENTE-SERVIDOR WEB

La arquitectura cliente-servidor web no es más que una arquitectura cliente-servidor donde los clientes son navegadores web (cliente web) que realizan una serie de peticiones mediante el protocolo *HTTP* a un servidor web.

Algunos de los servidores más conocidos son (Llario, s.f.):

- *Apache HTTP Server* (*Open Source*)
- *Apache Tomcat* (*Open Source*)
- *Intenet Information Services* (*IIS*)

3.1.3.1. URL

URL, Localizador Uniforme de Recursos, también conocida como dirección web, se utiliza particularmente con el protocolo *HTTP* y constituye una referencia a un recurso de Internet que el cliente solicita al servidor.

Las *URLs* pueden ser usadas con distintos protocolos, no únicamente con el protocolo *HTTP* como por ejemplo las direcciones de transferencia de ficheros (*FTP*), correo electrónico (*mailto*), etc.

esquema://dominio: puerto/ruta

- **esquema**: especifica el protocolo, por ejemplo: *http, https, ftp, file, mailto*, etc.
- **dominio**: es el nombre o la IP de la máquina a la cual solicitamos el recurso.
- **puerto**: es opcional, si no se especifica se utiliza el puerto por defecto para el esquema utilizado. Por ejemplo, para http el puerto por defecto sería el *80*.
- **ruta**: camino o ruta al recurso solicitado en el servidor0

.

3.1.3.2. SERVIDORES DNS

Anteriormente se ha visto que mediante el protocolo *TCP/IP* para poder dirigir los datos entre dos ordenadores se necesita especificar la dirección *IP* del origen y del destino.

Cuando utilizamos una URL en un navegador no especificamos la dirección *IP* destino pero la comunicación funciona gracias a los llamados Servidores de nombres de dominio (*Domain Name System*).

Estos servidores, cuyas direcciones quedan establecidas en la configuración de red de nuestro ordenador, se encargan de traducir los nombres de dominio o *hostname* a las direcciones *IP* correspondientes.

Si el servidor *DNS* no funcionara o no estuviera configurado la única forma de poder conectarse a un servidor web sería utilizar directamente la dirección *IP* del servidor web de destino.

3.2. NODEJS Y APLICACIONES WEB

NodeJS es una plataforma diseñada para ejecutar programas escritos en JavaScript. Utiliza el motor V8 de JavaScript de Chrome, posee una arquitectura orientada a eventos con la utilización de un único hilo de ejecución (*Thread*) no bloqueante, lo cual le convierte en una plataforma ligera y eficiente.

Además *NodeJS* posee un manejador de paquetes (*Node Package Manager* o *NPM*) que alberga el mayor ecosistema desarrolladores y librerías de código abierto.

Incluye en su núcleo una gran variedad de módulos (correspondientes al manejo de capas de bajo nivel) como los módulos *http* y *https*, que en concreto ayudan a realizar servidores *HTTP* y *HTTPS*.

Otro módulo importante es el módulo Clúster que contiene una clase llamada *Worker*. *NodeJS*, así como *JavaScript*, funcionan trabajando con un único hilo de ejecución, esto lo hace realmente eficiente para operaciones que no requieran un alto grado de procesamiento por el servidor. En ocasiones, algunos procesos pueden requerir dicho alto grado de procesamiento y la clase *Worker* ayuda a ejecutar diferentes hilos que puedan funcionar concurrentemente y en paralelo. Estos procesos hijo pueden ayudar a aprovechar al máximo el procesador del servidor para que utilice eficientemente sus núcleos.

La lista completa de módulo puede encontrarse en su página oficial (“https://nodejs.org”).

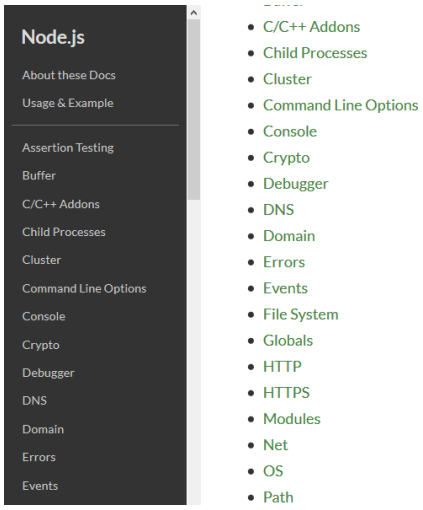


Imagen 8 - Algunos de los módulos del núcleo de NodeJS.

Hay que tener en cuenta que a pesar de la importancia de estos módulos, *NodeJS* para los desarrolladores tendría poca utilidad sin su repositorio *NPM*, ya que en este se alojan módulos de otros desarrolladores que generalmente nos van a facilitar el trabajo y nos van a evitar escribir código que ya está escrito y testeado.

Alguno de los módulos más importantes y famosos dentro de *NPM* es *ExpressJS*, el cual ayuda en la puesta en marcha y la creación de aplicaciones web.

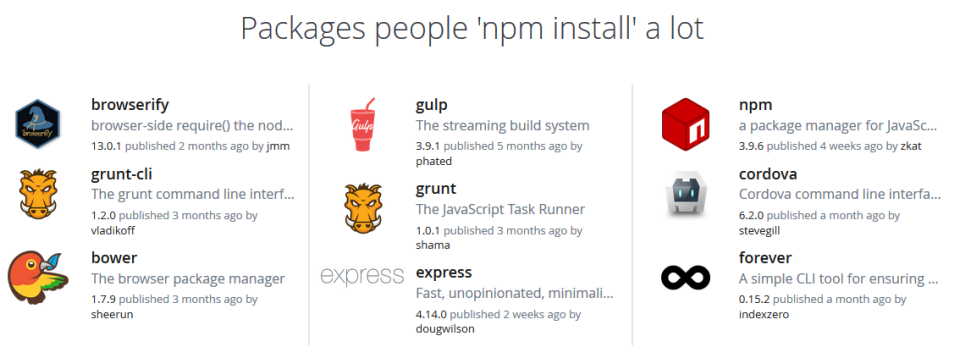


Imagen 9 - Módulos más instalados según la web de NPM. Entre ellos se encuentra el Framework Express.

NodeJS ha tenido un impacto mundial desde que fuera creado en 2009 por Ryan dahl, de hecho ha sido el motivo de que se hayan elaborado y aprobado nuevas especificaciones del lenguaje *JavaScript* tan rápidamente en los últimos años.

Es por tanto lógico pensar que dentro del mundo de las aplicaciones web ha causado una gran repercusión. La idea de escribir código para ambas plataformas (Cliente y Servidor) en el mismo lenguaje es una idea que sedujo a la comunidad de desarrolladores y que parece que avanza a pasos agigantados.

Imagen 10 - Ejemplo de un servidor HTTP sencillo realizado en NodeJS.

Un ejemplo de un sencillo servidor *HTTP* que responde a una petición se muestra en la imagen anterior. En la primera línea de código se define una función que recibe "*req*" y "*res*", dos objetos relacionados con la petición y la respuesta respectivamente. Dentro de la función se especifica una cabecera que especifica el tipo *MIME* de la respuesta como texto plano y finalmente escribe "*Hola TFG*" en el cuerpo de la respuesta. En la segunda utilizamos el método **createServer** del módulo http al que se le pasa la función anterior y se especifica al servidor que escuche en el puerto **8000**. Al ejecutar esta última línea de código se mostrará la información del servidor creado y será accesible desde nuestra máquina mediante su dirección *IP*.

NodeJS es por tanto un entorno realmente potente para la realización de aplicaciones web que generalmente consumen recursos del propio servidor (imágenes, contenido multimedia, etc.) o de otros servicios como una base datos local, servicios externos, etc.

También es útil para el *web scraping,* o lo que es lo mismo, la extracción de información de ciertas partes de un documente HTML, práctica utilizada cuando es necesario obtener la información de una página web y esta no es accesible desde un servicio *JSON* o *XML*.

Otras aplicaciones de *NodeJS* son la generación de scripts o rutinas mediante código JavaScript y ejecutarlas desde el terminal.

En cuanto a *frameworks* o librerías destinadas a implementar la lógica del cliente, actualmente se pueden realizar completas aplicaciones basadas en el patrón **Modelo-Vista-Controlador** alimentadas con información de nuestro servidor. Algunas de las más utilizadas son *AngularJS* que posee un sistema de plantillas basado en rutas, *EmberJS*, etc. Otras destinadas a realizar aplicaciones exportables a código nativo de plataformas como Android, IOs o Windows Phone como *ReactJS*, *Cordova, Ionic*, etc.

No hay que confundirlas con librerías como *jQuery* o *Bootstrap*, que son librerías de diseño que tratan de simplificar operaciones sobre documentos *HTML* mediante código *JavaScript* y *CSS*.

4. ZONA DE ESTUDIO

Torrente (en valenciano y oficialmente Torrent) es una ciudad de la Comunidad Valencia, perteneciente a la provincia de Valencia y situada en el área metropolitana de Valencia en la comarca de la Huerta Oeste.

Posee 80551 habitantes según datos del Instituto Nacional de Estadística (INE) del año 2014, convirtiéndose en el segundo municipio con más población de la provincia de Valencia.

Se encuentra a 9 km de la ciudad de Valencia y unos 15 km del mar y limita al norte con Aldaya, Alacuás y Chirivella, al este con Picaña y Catarroja, al sur con Alcácer y Picasent, y al oeste con Monserrat, Godelleta, Turís y Chiva.

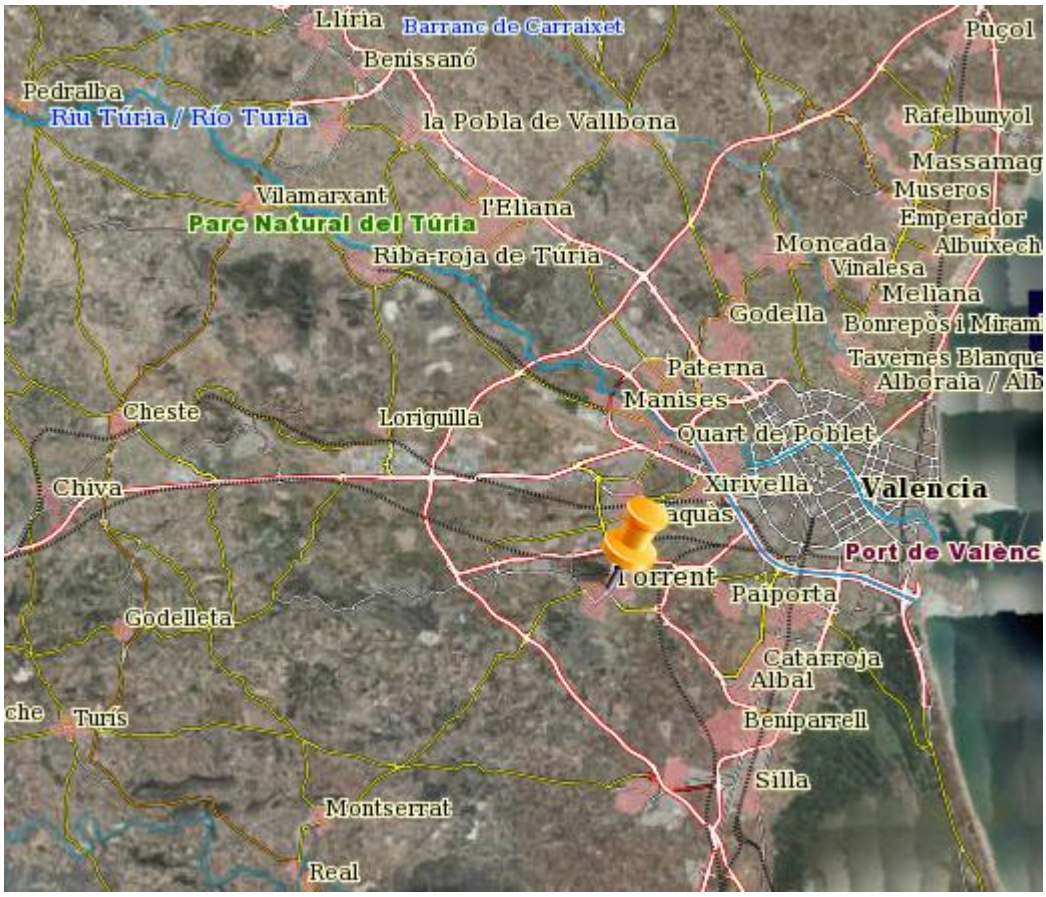


Imagen 11 - Localización del municipio de Torrent.

5. SOFTWARE EMPLEADO, INSTALACIÓN Y PRIMEROS PASOS

Antes de entrar de lleno en la elaboración e implementación de la Infraestructura de datos espaciales y la aplicación cabe destacar el trabajo previo de búsqueda de información y discriminación del software a emplear entre un gran abanico de posibilidades teniendo como único requisito que sea libre, gratuito y a poder ser de código abierto.

En este apartado se describen las principales herramientas software utilizadas, así como la instalación, configuración y los primeros pasos que se han llevado a cabo con las mismas.

5.1. MÁQUINA VIRTUAL Y SISTEMA OPERATIVO

En primer lugar se ha optado por llevar a cabo todo el proceso en una máquina virtual ya que nos permite llevar nuestro sistema operativo completo de una máquina a otra y arrancarlo mediante la instalación de un pequeño software de escritorio. En este caso la elección ha sido *VMWare*.

También se ha barajado la posibilidad de crear una máquina virtual online. Este tipo de hosting como cloud9 nos ofrecen un entorno *Linux* (*Ubuntu Server*) y una pequeña interfaz de usuario para crear proyectos en distintas plataformas (*Node.js, Ruby On Rails, Symphony*,…).

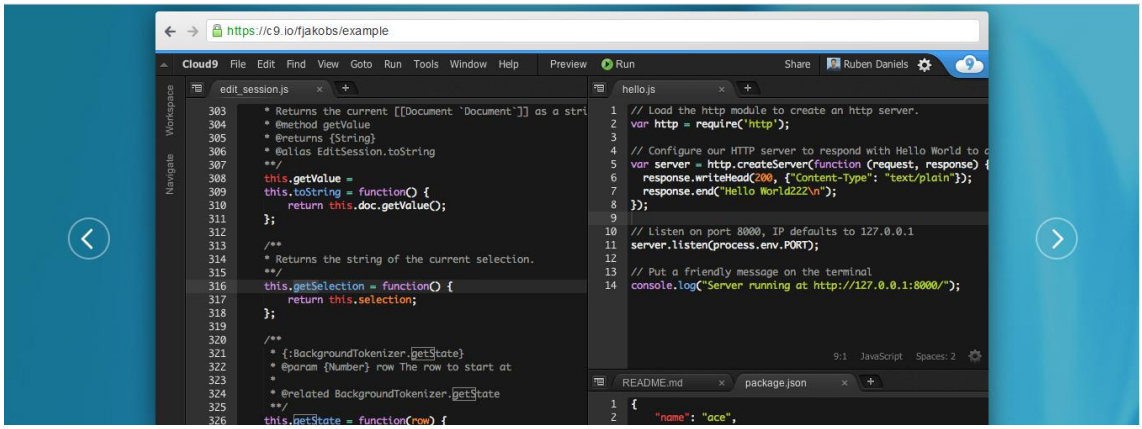


Imagen 12 - Página de Cloud9 - http://c9.io

Ubuntu Server es un sistema operativo que no posee una interfaz de escritorio gráfica si no que se maneja en su totalidad desde un terminal de Linux, también conocido como *bash*. Lo importante de este sistema operativo es que la instalación en disco no llega a superar los 800 *mb* lo que hace que sea realmente útil para ser instalado en máquinas que actúen como servidores. Por el contrario manejar todo el trabajo que supone este proyecto desde el terminal puede resultar una tarea sustancialmente costosa y relativamente complicada. Además, este tipo de servicios online como cloud9 tienen limitaciones dependiendo del tipo de suscripción que tengas.

Así pues, teniendo claro que vamos a utilizar *VMWare*, falta por definir el sistema operativo a utilizar. Se ha optado por utilizar una distribución de *Linux*, en concreto *Ubuntu LTS 15.10*.

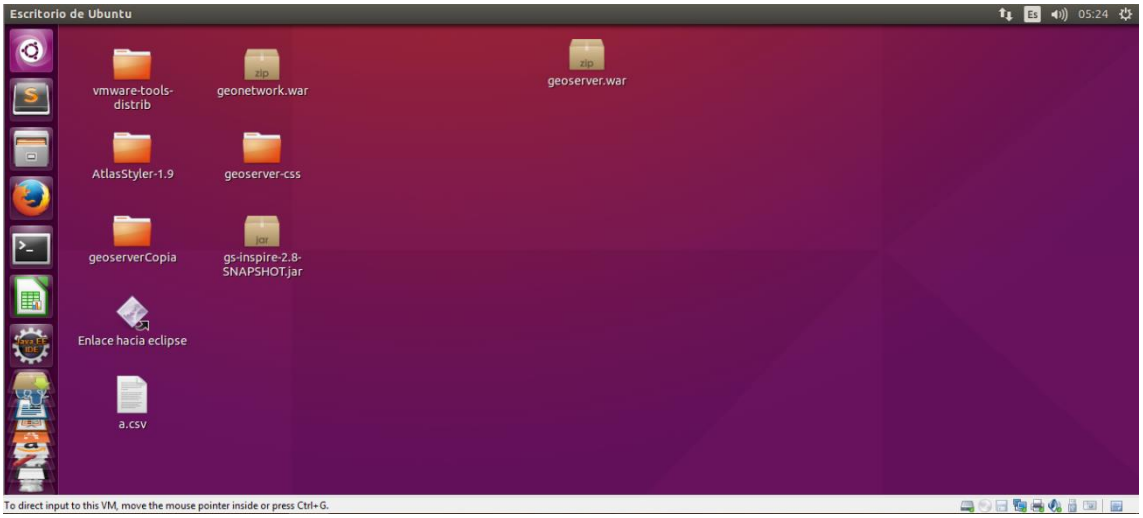


Imagen 13 – Escitorio de Ubuntu ejecutándose en la máquina virtual VMWare Player.

Ubuntu es un sistema operativo basado en *GNU/Linux* y que se distribuye como software libre. Su patrocinador, Canonical, es una compañía británica que ofrece el sistema de manera gratuita y se financia por medio de servicios vinculados al sistema operativo y vendiendo soporte técnico. Además, al mantenerlo libre y gratuito, la empresa es capaz de aprovechar los desarrolladores de la comunidad para mejorar los componentes de su sistema operativo. (Wikipedia, s.f.)

Así pues el proceso a llevar a cabo es descargar e instalar *VMWare* en nuestra máquina, descargar la distribución de Linux escogida, crear una máquina virtual con *VMWare Workstation* e instalar nuestro sistema operativo en ella.

5.2. SISTEMA DE VERSIONES, GIT Y GITHUB

Se llama control de versiones a la gestión de los diversos cambios que se realizan sobre los elementos de algún producto o una configuración del mismo. Una versión, revisión o edición de un producto, es el estado en el que se encuentra el mismo en un momento dado de su desarrollo o modificación. **Git** es un software de control de versiones diseñado pensando en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones cuando éstas tienen un gran número de archivos de código fuente. **GitHub** es un servicio web de host de repositorios *Git*. Al contrario que *Git*, que es una herramienta de línea de comandos, provee una interfaz gráfica web y de escritorio.

Para el control de las versiones de este proyecto y de esta manera ver los cambios que se han hecho entre diferentes actualizaciones se emplea la tecnología *Git*, instalando dicho software en nuestra máquina. También se creará una cuenta en *GitHub* para poder sincronizar nuestro repositorio local con el repositorio remoto que se crea.

Como se puede observar, para el proyecto se han realizado 47 *commits*, es decir, se han hecho modificaciones sobre la rama principal 47 veces, entre los meses de Diciembre (2015) a Junio de 2016.

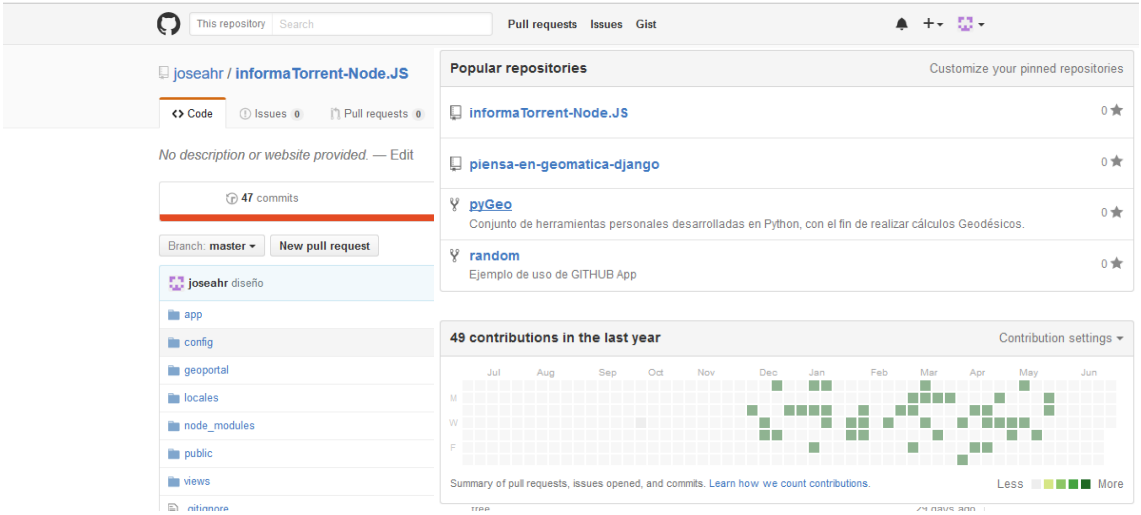


Imagen 14 - Repositorio del código del proyecto en GitHub.

La URL que da acceso al código en GitHub es *https://github.com/joseahr/informaTorrent-Node.JS*.

Una vez creados los repositorios (local y remoto), es necesario asignar el repositorio remoto al repositorio local (El directorio donde se encuentra la aplicación *Node*) mediante el comando "*git remote add origin*".

A partir de este momento ya existe la capacidad de usar la línea de comando de *Git* para añadir nuevas "ramas" y/o cambiar la rama principal con los comandos "*git commit –m*" y "*git push origin master*", etc.

Imagen 15 - Página realizada para mostrar el código del proyecto.

También se ha realizado una pequeña página *HTML* que muestra el árbol de trabajo del proyecto y muestra el código alojado en la rama master del proyecto en *GitHub* del archivo que se elija.

Imagen 16 - Página realizada para mostrar el código del proyecto. Archivo "server.js" abierto en el visor creado.

5.3. POSTGRESQL Y POSTGIS

PostgreSQL es un sistema de gestión de bases de datos relacionales orientado a objetos. Es libre, publicado bajo una licencia PostgreSQL similar a la BSD o MIT. El aspecto más característico de *PostgreSQL* es que posee una alta concurrencia. Mediante un sistema denominado *MVCC* (Acceso concurrente multiversión, por sus siglas en inglés) *PostgreSQL* permite que mientras un proceso escribe en una tabla, otros accedan a la misma tabla sin necesidad de bloqueos. Cada usuario obtiene una visión consistente de lo último a lo que se le hizo *commit*. (Wikipedia, s.f.)

PostGIS es un módulo que añade soporte de objetos geográficos a la base de datos *PostgreSQL*, convirtiéndola en una base de datos espacial para su utilización en Sistema de Información Geográfica. Se publica bajo la Licencia Pública General de *GNU*. Un aspecto que debemos tener en cuenta es que *PostGIS* ha sido certificado en 2006 por el Open Geospatial Consortium (OGC) lo que garantiza la interoperabilidad con otros sistemas también interoperables. Almacena la geometría en formato *WKB* (Well-Known Binary), aunque hasta la versión 1.0 se utilizaba la forma *WKT* (Well-Known Text). (Wikipedia, s.f.)

La instalación de *PostgreSQL* se puede llevar a cabo mediante una búsqueda en el repositorio *APT*, así pues haciendo uso del terminal ejecutamos el siguiente comando:

```
^Cjose@TFG:~$ sudo apt-get install postgresql-9.4 pgadmin3
Leyendo lista de paquetes... Hecho
Creando árbol de dependencias
Leyendo la información de estado... Hecho
pgadmin3 ya está en su versión más reciente.
```

Imagen 17 - Comando para la instalación de PostgreSQL.

Tras completarse la instalación de *PostgreSQL* (y de cualquier software que instalemos) se comprobará que funciona y que por tanto la instalación se completó correctamente. En este caso probaremos que tenemos acceso al *CLI* (Command Line Interface) de *PostgreSQL*:

```
jose@TFG:~$ psql
psql (9.4.5, server 9.4.8)
Type "help" for help.

jose=# \q
jose@TFG:~$
```

Imagen 18 - CLI (Command Line Interface) de PostgreSQL.

Se confirma entones que podemos ejecutar *PostgreSQL* desde el terminal y nos da información de la versión que tenemos instalada.

```
jose@TFG:~$ psql
psql (9.4.5, server 9.4.8)
Type "help" for help.

jose=# CREATE USER jose WITH PASSWORD 'jose'
jose-# CREATE USER geonetwork WITH PASSWORD 'geonetwork'
jose-#
```

Imagen 19 - Creación de usuarios en PostgreSQL.

El primer paso que debemos hacer tras instalar *PostgreSQL* es crear usuarios para la autenticación de la conexión con las bases de datos que creemos. Se crean dos usuarios, uno para acceder a las bases de datos que se crearán para almacenar la cartografía y las tablas de la aplicación y otro para acceder a la base de datos que se creará para *Geonetwork*. Más adelante se detallará.

La instalación de *PostGIS* también la realizamos desde el terminal:

```
jose@TFG:~$ sudo apt-get install -y postgis postgresql-9.4-postgis-2.1
Leyendo lista de paquetes... Hecho
Creando árbol de dependencias
Leyendo la información de estado... Hecho
postgis ya está en su versión más reciente.
postgresql-9.4-postgis-2.1 ya está en su versión más reciente.
fijado postgresql-9.4-postgis-2.1 como instalado manualmente.
```

Imagen 20 - Comando para instalar PostGIS.

Para utilizar *PostGIS* en la base de datos *PostgreSQL* crearemos más adelante la extensión de *PostGIS* con la función *CREATE EXTENSION*, adquiriendo así un carácter espacial heredando todas las funciones Geoespaciales que trae *PostGIS*.

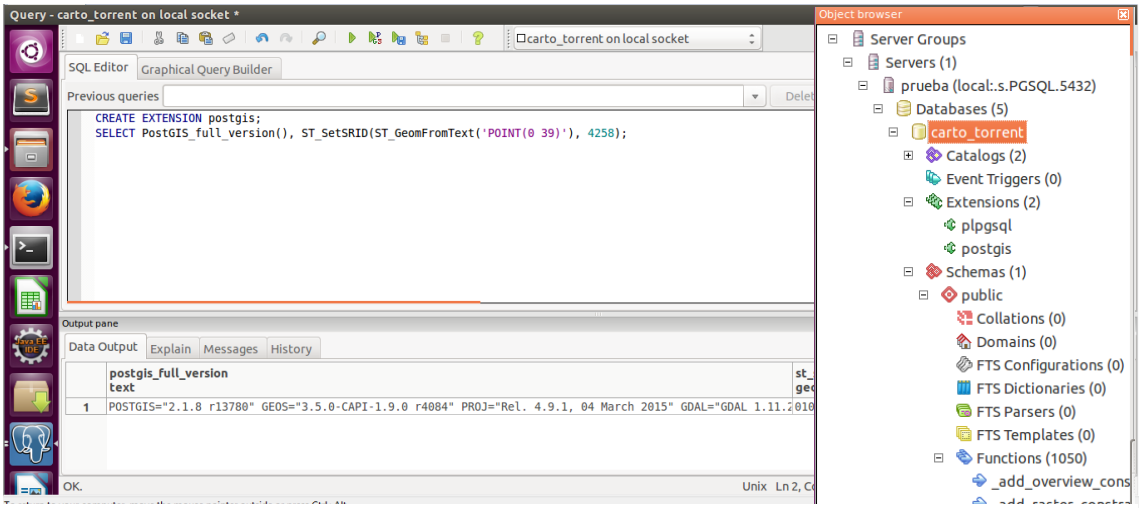


Imagen 21 - Creación de la extensión PostGIS en una base de datos y consultas con funciones de PostGIS. Funciones de PostGIS instaladas (1050).

5.4. APACHE TOMCAT, GEOSERVER Y GEONETWORK

5.4.1. APACHE TOMCAT

Tomcat funciona como un contenedor de *servlets*, es decir, pequeños programas o aplicaciones que se ejecutan en el contexto del navegador web. *Tomcat* implementa las especificaciones de los *servlets* y de *Java Server Pages* (JSP) de Oracle y funciona bajo un servidor *HTTP* de *Apache*. En lo que concierne a este proyecto, la versión de *Tomcat* que se va a instalar es las 7 debido a que se usa la versión 7 de *Java*, y su principal uso es el de servir como contenedor de *Geoserver* y *Geonetowrk*.

La instalación de *Tomcat* también se realiza desde la búsqueda en el repositorio *APT*:

```
jose@TFG:~$ sudo apt-get install tomcat7-docs tomcat7-examples tomcat7-admin
[sudo] password for jose:
Leyendo lista de paquetes... Hecho
Creando árbol de dependencias
Leyendo la información de estado... Hecho
tomcat7-admin ya está en su versión más reciente.
tomcat7-docs ya está en su versión más reciente.
tomcat7-examples ya está en su versión más reciente.
```

Imagen 22 - Comando para instalar Apache Tomcat.

Una vez instalado debemos crear un usuario que tenga el rol necesario para controlar el gestor de aplicaciones "*manager-gui*", para poder instalar *Geoserver* y *Geonetwork* posteriormente. Esto se hace editando el archivo "*tomcat-users.xml*" situado en la carpeta */var/lib/tomcat7/conf*. Como necesitamos permisos de administrador para escribir en el archivo accederemos a él a través del terminal con el comando "*gedit*" para abrirlo con el editor de texto *gedit* y con la opción *sudo* para dar permisos de usuario:

```
sudjose@TFG:/var/lib/tomcat7/conf$ sudo gedit tomcat-users.xml
[sudo] password for jose:
```

Imagen 23 - Comando para acceder a /var/lib/tomcat7/conf/tomcat-users.xml con permisos de administrador.

De este modo se nos abrirá el editor y podremos editar el archivo.

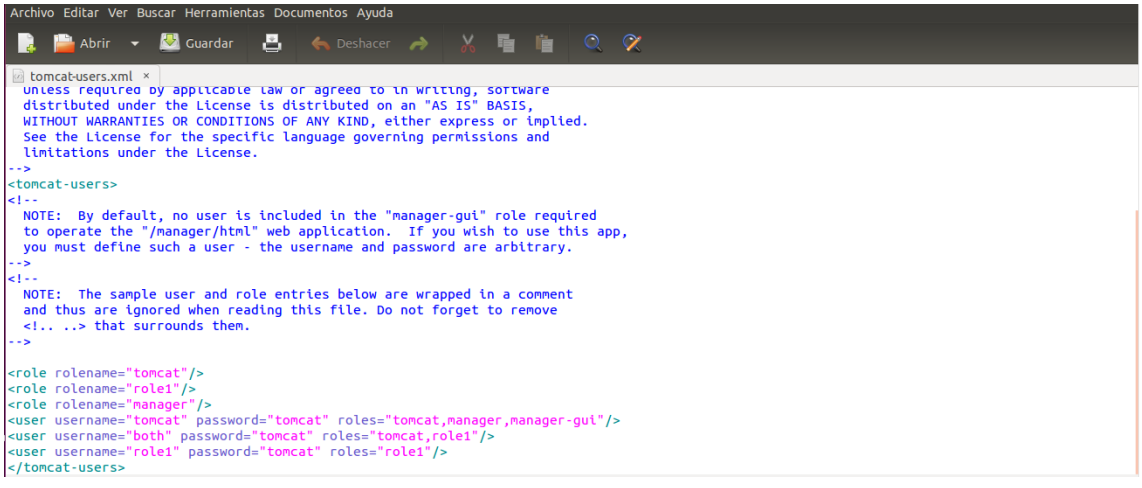


Imagen 24 - Edición del archivo /var/lib/tomcat7/conf/tomcat-users.xml

Para comprobar que *Tomcat* funciona reiniciaremos el servicio.

Tras iniciarse el servicio accedemos mediante nuestra IP y el puerto donde se está ejecutando *Tomcat* y comprobamos que se muestra la página de Inicio de *Tomcat*:

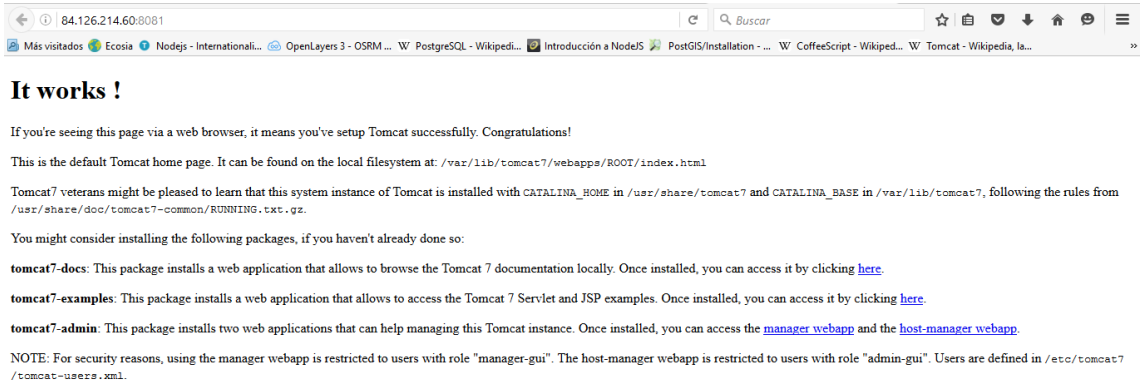


Imagen 25 - Página de Inicio de Tomcat.

5.4.2. GEOSERVER Y GEONETWORK

GeoServer es un servidor de código abierto escrito en Java que permite a los usuarios compartir y editar datos geoespaciales. Diseñado para la interoperabilidad, publica datos de las principales fuentes de datos espaciales usando estándares abiertos. *GeoServer* pretende operar como un nodo a través de una IDE libre y abierta para ofrecer datos geoespaciales.

Entre las principales características de *Geoserver* se pueden citar algunas como (Wikipedia, s.f.):

- Enteramente compatible con las especificaciones *WMS, WCS* y *WFS*, testados por el test de conformidad *CITE* del *OGC*.
- Fácil utilización a través de la herramienta de administración vía web, no es necesario entrar en archivos de configuración grandes y complicados.
- Soporte amplio de formatos de entrada *PostGIS, Shapefile, ArcSDE y Oracle.VFP, MySQL, MapInfo* y *WMS* en cascada también están entre los formatos de entrada soportados, entre otros.
- Soporte de formatos de salida tales como *JPEG, GIF, PNG, SVG, GML, GeoJSON*, etc.
- Soporte completo de *SLD*, como definiciones del usuario (*POST* y *GET*), y como uso de configuración de estilos.
- Soporte para edición de datos de banco de datos individuales a través del protocolo *WFS Transaccional* (*WFS-T*), disponible para todos los formatos de datos.
- Basado en *servlets Java (JEE)*, puede funcionar en cualquier *servlet* contenedor, en este caso *Tomcat* 7.
- Proyectado para ser compatible con extensiones.

Geonetwork es un entorno de gestión de información espacial diseñado para permitir acceso a bases de datos georreferenciadas, productos cartográficos y metadatos de varias fuentes, mejorando el intercambio usando las capacidades de la Internet. Es una aplicación de catálogo que maneja datos espacialmente referenciados, provee excelentes funciones de edición y búsqueda de metadatos. Actualmente numerosas iniciativas de Infraestructuras de datos espaciales alrededor del mundo. (Wikipedia, s.f.)

Además de mapas, servicios y capas geoespaciales, conjuntos de datos no geográficos también pueden ser descritos por el catálogo.

Las versiones instaladas de *Geoserver* y *Geonetwork* son la *2.8 SNAPSHOT* y la *2.10.2* respectivamente.

La instalación de *Geoserver* y *Geonetwork* se realiza a través del gestor de aplicaciones de *Tomcat* al cual accederemos con el usuario y contraseña con los permisos necesarios creado anteriormente:

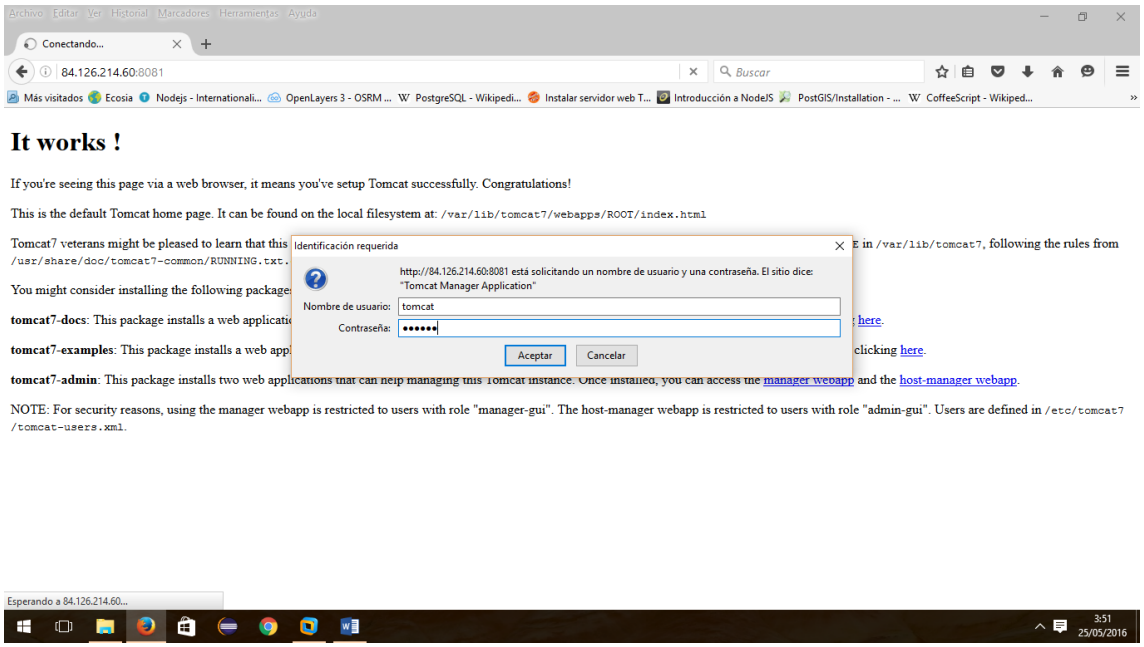


Imagen 26 - Accediendo al "manager-gui" con nuestras credenciales.

El siguiente paso es descargar los archivos "*.war*" (*Web Application Archive*) que son "a grosso modo" las dependencias (Código *Java* compilado en una librería) de la apliaciones web (*Geoserver* y *Geonetwork* en este caso) y sirven como instaladores en *Tomcat*.

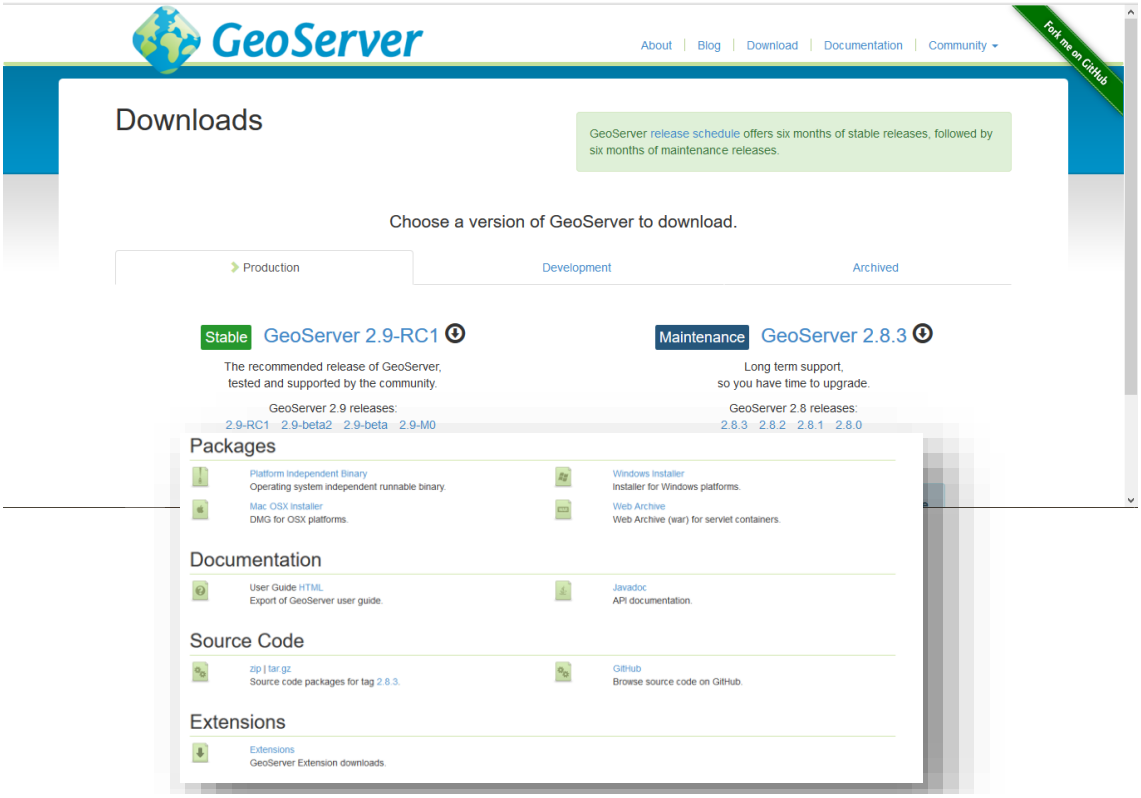


Imagen 27 - Web de Geoserver, formatos de descarga, referencias a la documentación y al código fuente, extensiones.

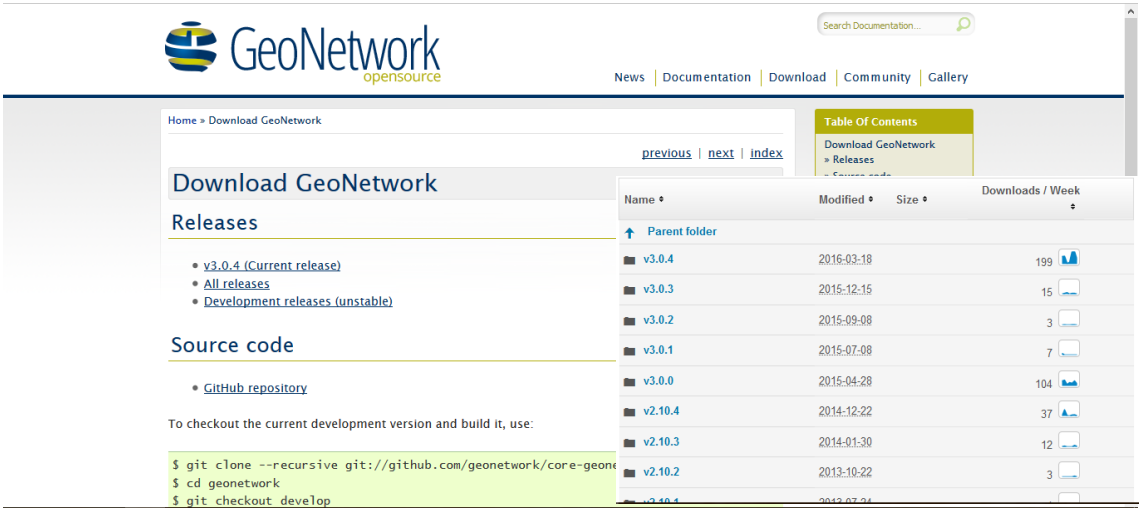


Imagen 28 - Web de Geonetwork, código fuente, versiones disponibles para descargar.

Para poder llevar a cabo la instalación de estos archivos "*.war*" es necesario modificar una variable de entorno en la configuración de *Tomcat* relacionada con la subida de archivos (formularios multiparte). Por defecto *Tomcat* tiene preestablecido un tamaño máximo que los archivos que se suban al servidor no deben superar. En este caso los archivos "*.war*" de *Geoserver* y *Geonetwork* ocupan más que lo que indica el parámetro por defecto, por lo que hay que cambiarlo de forma que podamos subir archivos del tamaño de *Geonetwork*, que es el que más ocupa de los dos. El parámetro coincide con la etiqueta XML "*max-file-size*" dentro del archivo **web.xml**:

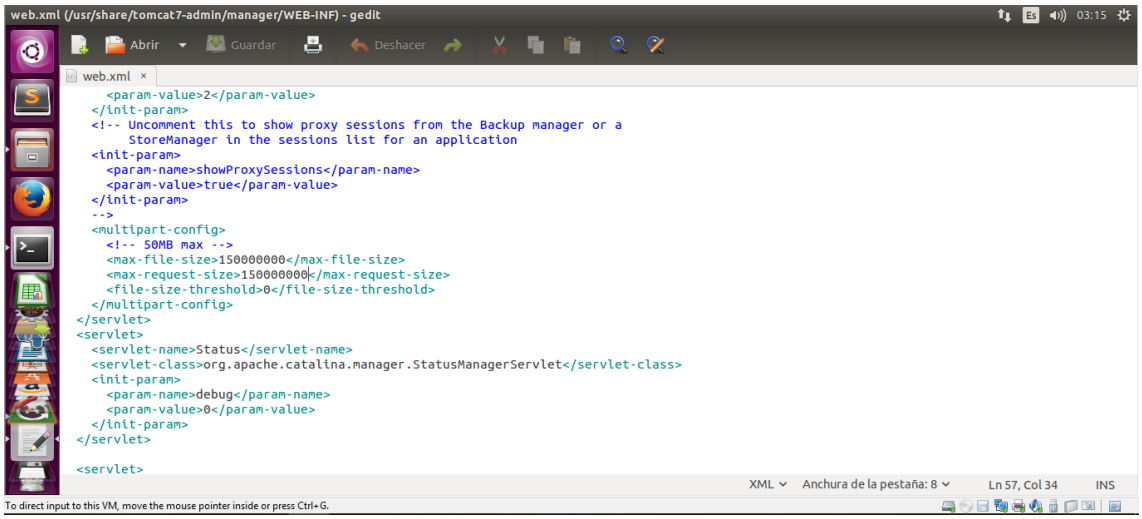


Imagen 29 - Editando parámetro max-file-size.

Posteriormente procederemos a cargar dichos archivos en el gestor de aplicaciones de *Tomcat*:

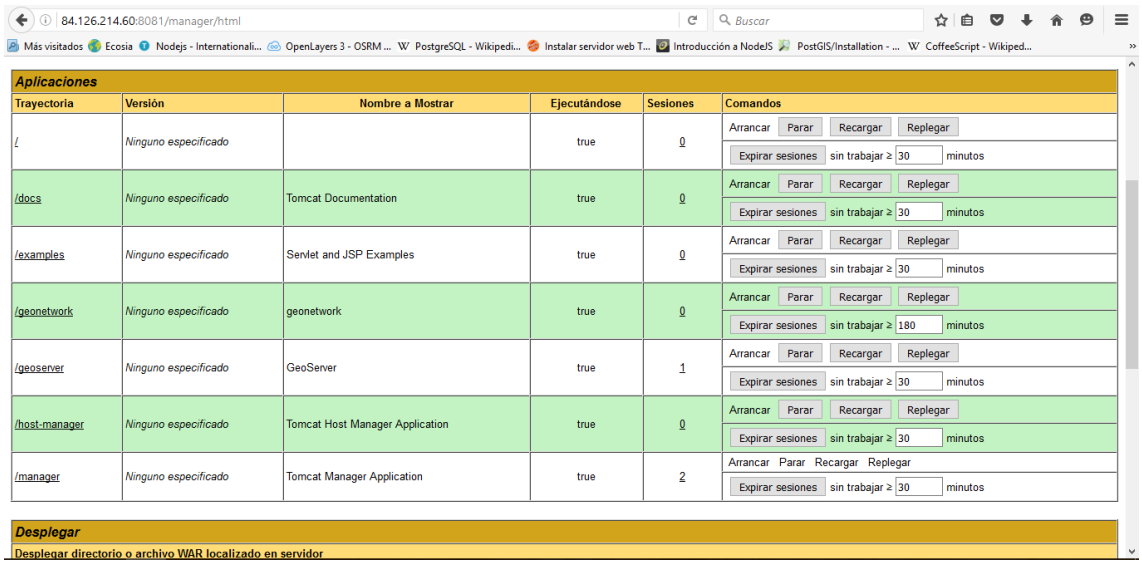


Imagen 30 - Geoserver y Geonetwork instalados en Tomcat.

Una vez finalizado ya tendremos *Geoserver* y *Geonetwork* funcionando en nuestra máquina.

Debido a que la carpeta donde instala la base de datos *Geonetwork* es una carpeta en la que *Tomcat* no tiene permisos de administrador y produce un error, se ha querido cambiar el motor de base de datos de *Geonetwork* que por defecto es *H2*, para que funcione a través de una base de datos *PostgreSQL*. Para ello, usando los parámetros del usuario de *PostgreSQL* creado para *Geonetwork* cambiaremos este parámetro desde el archivo "*config.xml*" situado en la carpeta "*/var/lib/tomcat7/webapps/geonetwork/WEB-INF*".

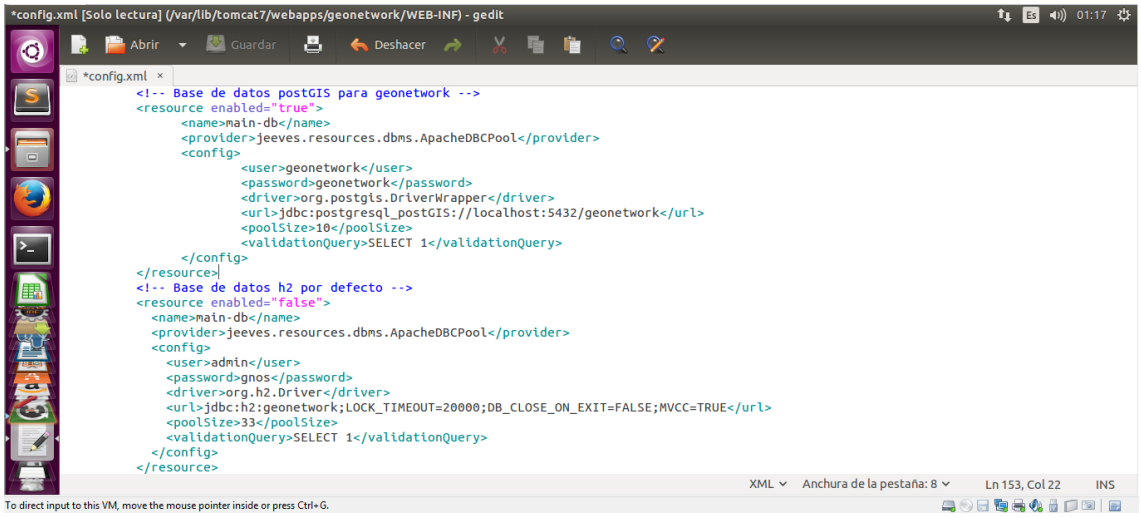


Imagen 31 - Cambio del motor de bases de datos de Geonetwork a PostgreSQL.

Se puede observar que se ha dado una ruta hacia una base de datos *PostgreSQL* que de momento no existe. Tras reiniciar el servicio de *Tomcat*, *Geonetwork* se encargará de crear la base de datos con el nombre que le hemos dado.

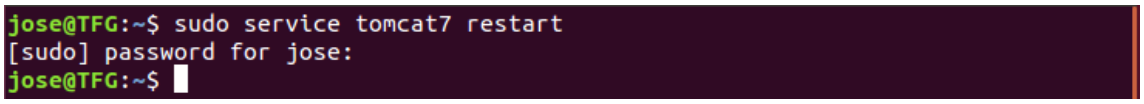


Imagen 32 - Comando para reiniciar el servicio de Tomcat 7.

De esta forma ya no existirá ningún conflicto con los permisos de escritura mencionados anteriormente y *Geonetwork* almacenará toda su información en una base de datos *PostgreSQL*.

A continuación se comprueba la información general de los productos instalados.

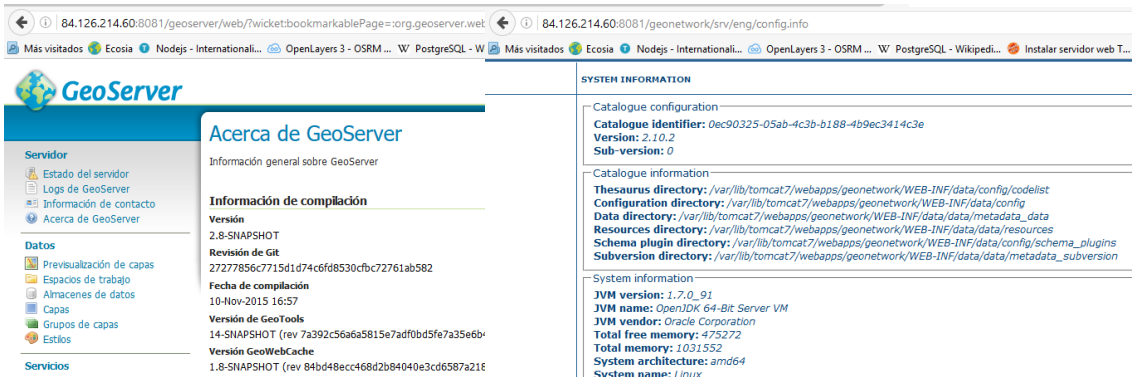


Imagen 33 - Información de las versiones instaladas de Geoserver y Geonetwork.

5.4.3. REDIRECCIÓN DE PUERTOS

Con el fin de servir nuestras aplicaciones por Internet a través del *router* del hogar, se pretende redireccionar un puerto de este para que traslade la petición a un puerto de la máquina virtual. De esta manera se podrán tener los servicios que ejecutemos sirviéndose en la red teniendo como punto de acceso el propio *router* del hogar. Por lo tanto desde este momento hay que definir ciertos parámetros como en qué puertos van a ejecutarse las aplicaciones en la máquina virtual (**puerto privado**) y desde qué puertos vamos a transferir las peticiones a cada aplicación (**puerto público**).

- Para el servidor *NodeJS*, el puerto público y privado en ambos casos será el *3000*.
- Para *Geoserver* y *Geonetwork*, el puerto público y privado en ambos casos será el *8081*, ya que el *8080* por defecto de *Tomcat*, no se puede utilizar como puerto público, ya que es el puerto utilizado para el transporte por el protocolo *HTTP*.

Podemos acceder a la interfaz gráfica del *router* a través de su *IP*, como se muestra en la siguiente imagen:

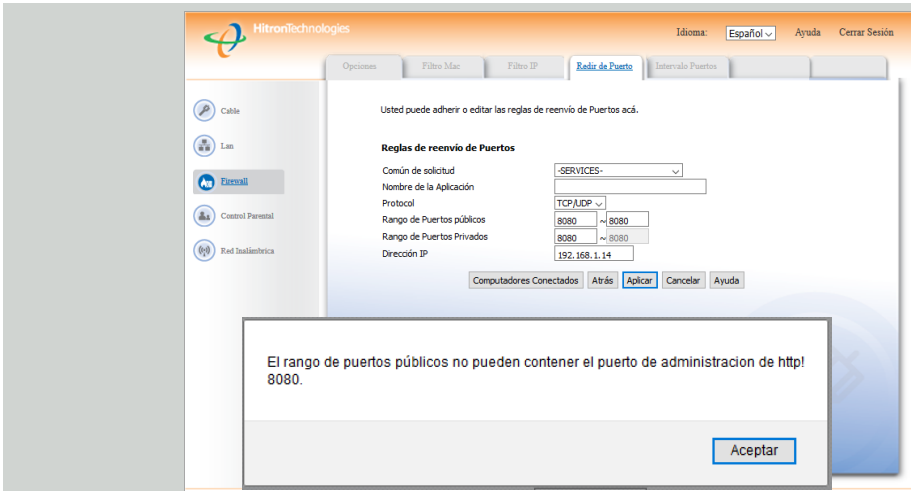


Imagen 35 - Interfaz gráfica del router para la redirección de puertos y error al tratar de asignar el puerto público a 8080.

Esto supone cambiar el puerto por defecto en el que *Tomcat* se ejecuta. Este parámetro se encuentra en el archivo "*server.xml*" del directorio "*/etc/tomcat7*":

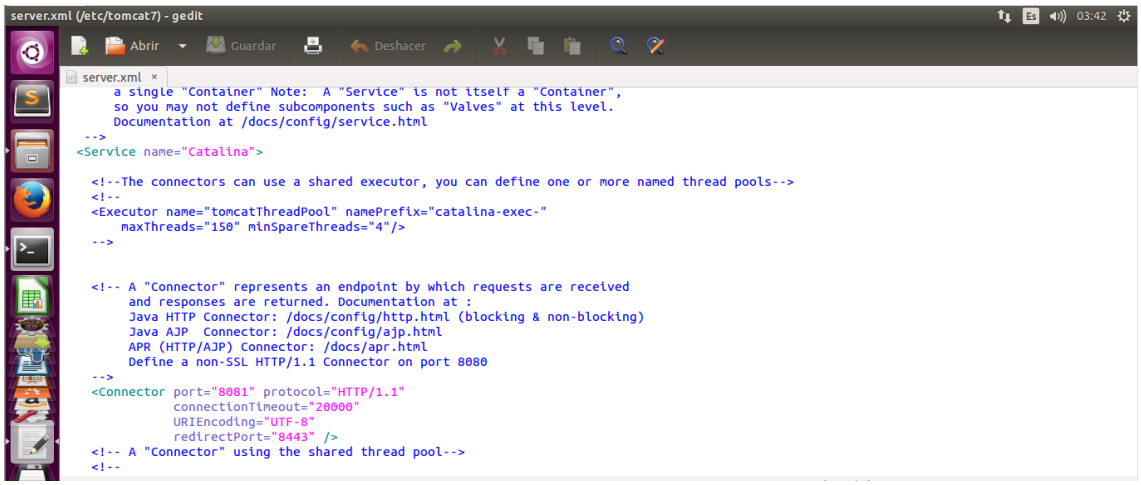


Imagen 34 - Cambiando el puerto por el cual se ejecuta Tomcat y sus aplicaciones.

Una vez reiniciado el servicio de *Tomcat* se puede acceder a *Tomcat* y sus aplicaciones instaladas desde el puerto *8081*.

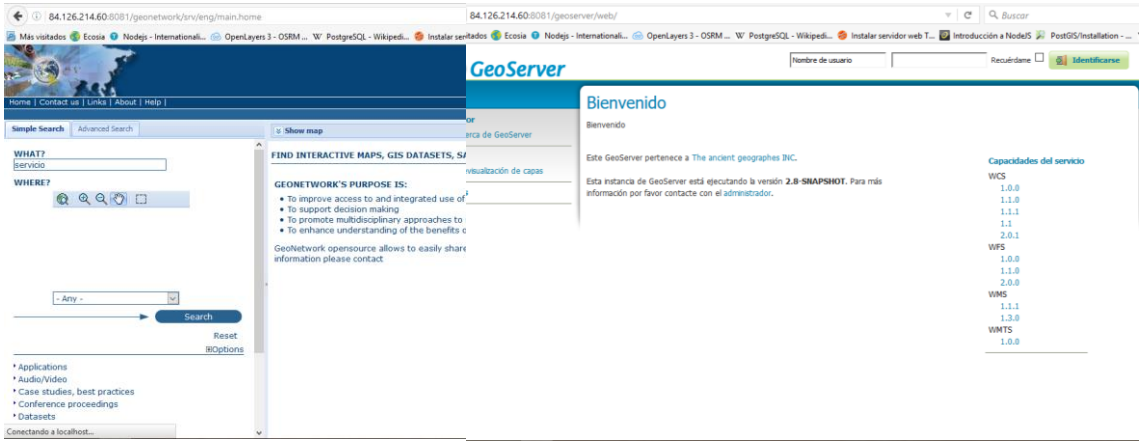


Imagen 36 - Geoserver y Geonetwork accesibles desde el puerto 8081.

5.4.4. CREACIÓN DE UN ESPACIO DE TRABAJO EN GEOSERVER.

Los espacios de trabajo "*workspace*" en *Geoserver* nos sirven para obtener una separación lógica de las capas y brindar una dirección única mediante un espacio de nombres "*namespace*" para cada capa. Es por ello que podríamos tener varias capas con el mismo nombre en distintos espacios de trabajo y mantener una referencia unívoca a cada capa gracias al estar en distintos espacios de trabajo.

Así pues si existe un espacio de trabajo llamado "*mi_espacio*" y una capa dentro de este llamada "*mi_capa*", ésta quedaría unívocamente identificada mediante "*mi_espacio: mi_capa*".

Hay que tener en cuenta que se creará un servicio web con una dirección específica para cada espacio de trabajo, es por ello que en este proyecto solo se utiliza un espacio de trabajo, aunque también sería lógico haber hecho una separación entre las capas cuyo uso es más frecuente en el *Geoportal* y las capas referentes a la aplicación.

Para crear el espacio de trabajo que contendrá las capas y dará acceso a los servicios, entraremos al apartado "Espacios de trabajo" en *Geoserver*. En la parte superior de esta página da la opción para crear un nuevo espacio de trabajo.

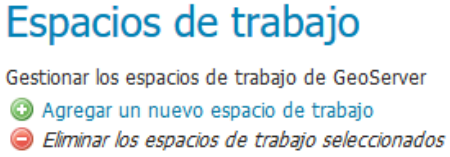


Imagen 37 - Menú que da acceso al formulario de creación de un nuevo espacio de trabajo en Geoserver.

Al hacer *click* sobre esta opción, *Geoserver* nos mostrará un formulario para introducir los datos referentes al espacio de trabajo que se creará.

Al rellenar la información correctamente y aceptar el formulario ya se habrá creado el espacio de trabajo que se utilizará más adelante cuando haya que publicar las capas de los distintos almacenes de trabajo que se crearán.

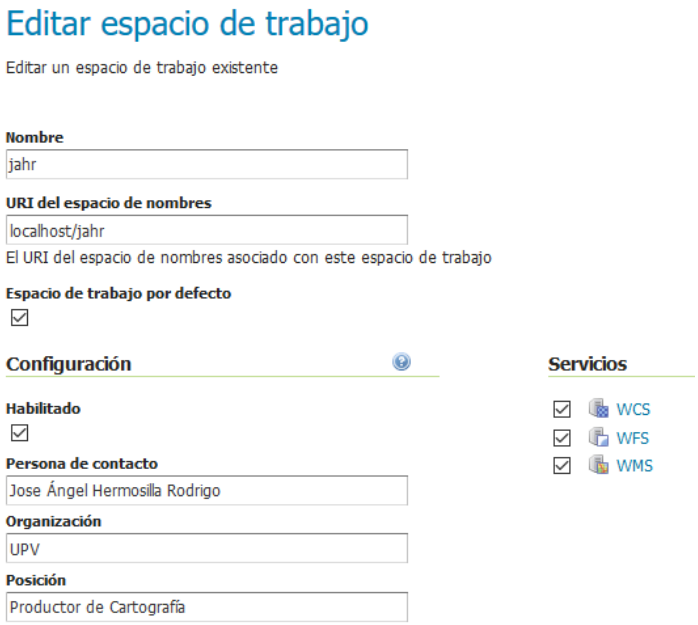


Imagen 38 - Formulario para la creación del Espacio de trabajo.

5.4.5. CONFIGURACIÓN DE LOS SERVICIOS WMS, WFS Y WCS.

A continuación se habilitarán y configurarán los servicios que *WMS*, *WFS* y *WCS* que se usarán para servir la cartografía almacenada en el espacio de trabajo creado anteriormente.

Hay que mencionar que se ha instalado una extensión en *Geoserver* que hace que los documentos de capacidades de los servicio *WMS* y *WFS* añadan ciertos atributos propuestos por *INSPIRE*.

En concreto la información que añade es la siguiente:

- Para **WMS** → lenguaje del documento, URL del servicio de metadatos asociado con el servicio WMS y el tipo de servicio de metadatos.
- Para **WFS** → Añade los mismos atributos que el WMS y además las direcciones a los servicios de metadatos de cada conjunto de datos utilizado en el servicio, si se rellena.

La instalación del *plugin* dará acceso a su configuración para cada servicio en la propia configuración de cada servicio.

Se accederán a cada una de estas configuraciones desde el menú lateral izquierdo.

A continuación se muestran algunos de los parámetros de configuración del servicio *WMS*:

Imagen 39 - Configuración del servicio WMS.

Se puede observar cómo se utiliza el espacio de trabajo creado para este proyecto y la sección de INSPIRE que nos habilita el *plugin*. Toda la información que se pasa en esta sección aparecerá en el documento de capacidades del servicio.

Procederemos de la misma forma para el servicio *WFS*:

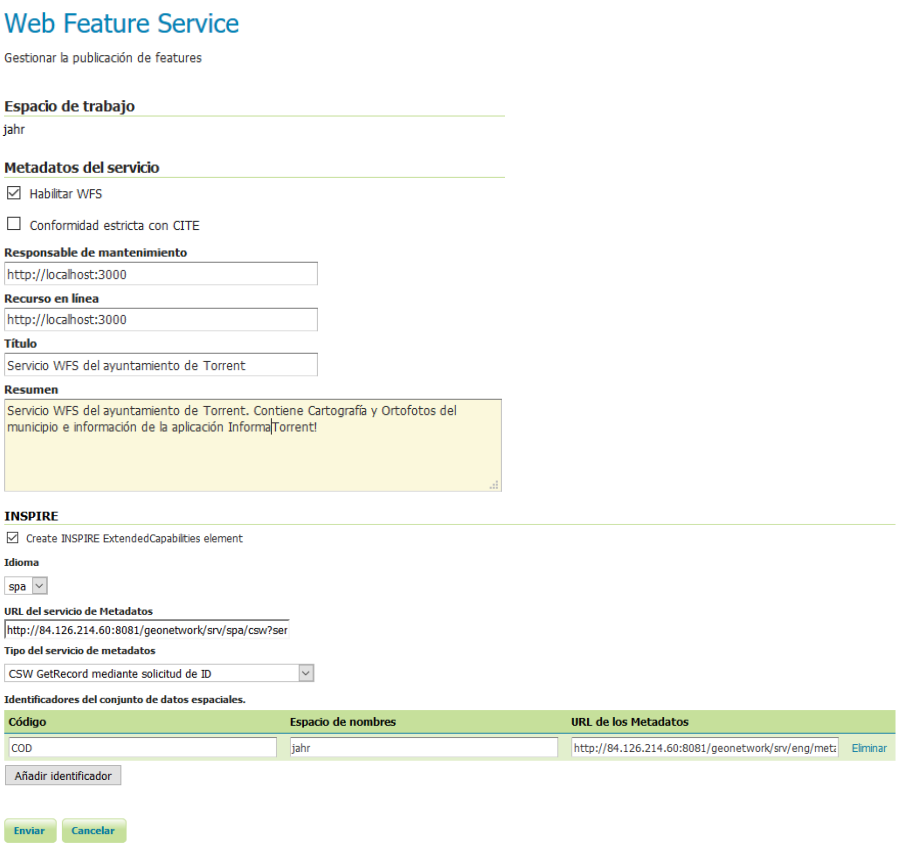


Imagen 40 - Configuración del servicio WFS.

La única excepción en este formulario es la sección de *INSPIRE* que como se puede observar en la imagen da la opción de añadir la dirección hacia el servicio de metadatos asociado con cada conjunto de datos que contenga dicho servicio.

Análogamente para el servicio *WCS*:

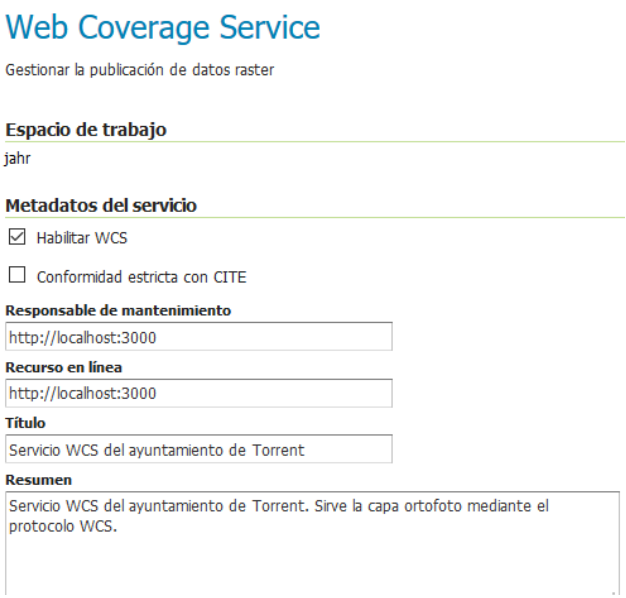


Imagen 41 - Configuración del servicio WCS.

5.5. ECLIPSE Y ENIDE

Eclipse es una plataforma compuesta por un conjunto de herramientas de programación de código abierto multiplataforma para desarrollar lo que el proyecto llama aplicaciones de cliente enriquecido, opuesto a las aplicaciones de cliente ligero basadas en navegadores. Esta plataforma, típicamente ha sido usada para desarrollar entornos de desarrollo integrados (del inglés *IDE*), como el *IDE* de *Java* llamado *Java Development Toolkit* (*JDT*) y el compilador (*ECJ*) que se entrega como parte de Eclipse (y que son usados también para desarrollar el mismo *Eclipse*. (Wikipedia, s.f.)

Enide es un *plug-in* disponible en el *Marketplace* de *Eclipse* que nos proporciona un entorno más centrado en la perspectiva *JavaScript* y en el desarrollo en *NodeJS*. Tiene soporte para escribir en *TypeScript* y *CoffeScript*, este último posee una sintaxis muy parecida a Python y se compila en JavaScript.

Podemos descargar cualquier versión de Eclipse desde su página web con la ventaja de que la instalación consiste en extraer el contenido del "*.tar*" donde se desee y hacer un acceso directo al ejecutable.

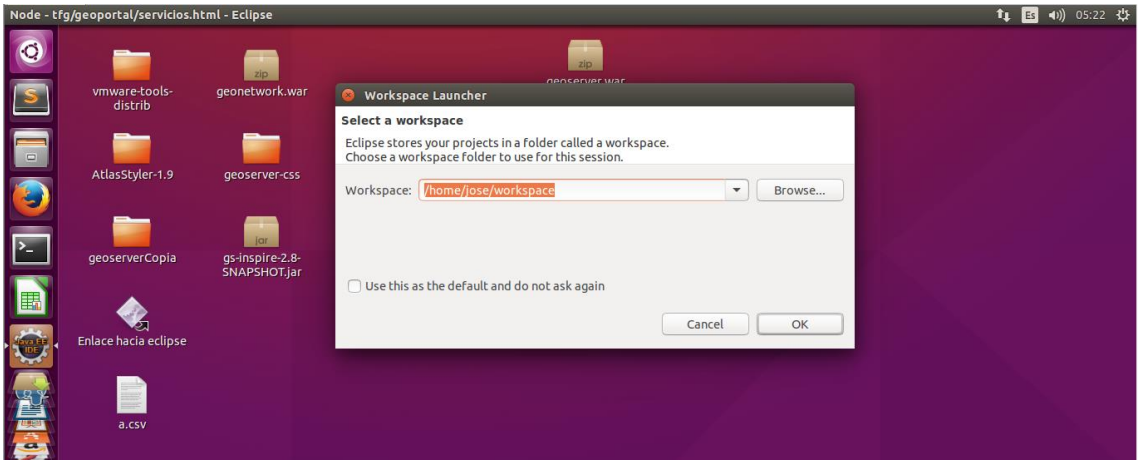


Imagen 42 - Espacio de trabajo en Eclipse.

Al iniciar el programa pedirá que elijamos una carpeta que servirá de espacio de trabajo por defecto para la sesión que iniciemos (Se pueden tener varios espacio de trabajo).

Así pues se creará una carpeta que sirva como contenedora del proyecto a realizar (y de otros posibles proyectos) y se elegirá dicho espacio de trabajo como carpeta madre al iniciar Eclipse.

Enide se instala directamente desde *Eclipse* y sus distintas versiones se pueden encontrar en el "*Eclipse Marker Place*", un repositorio de *plugins* para *Eclipse*.

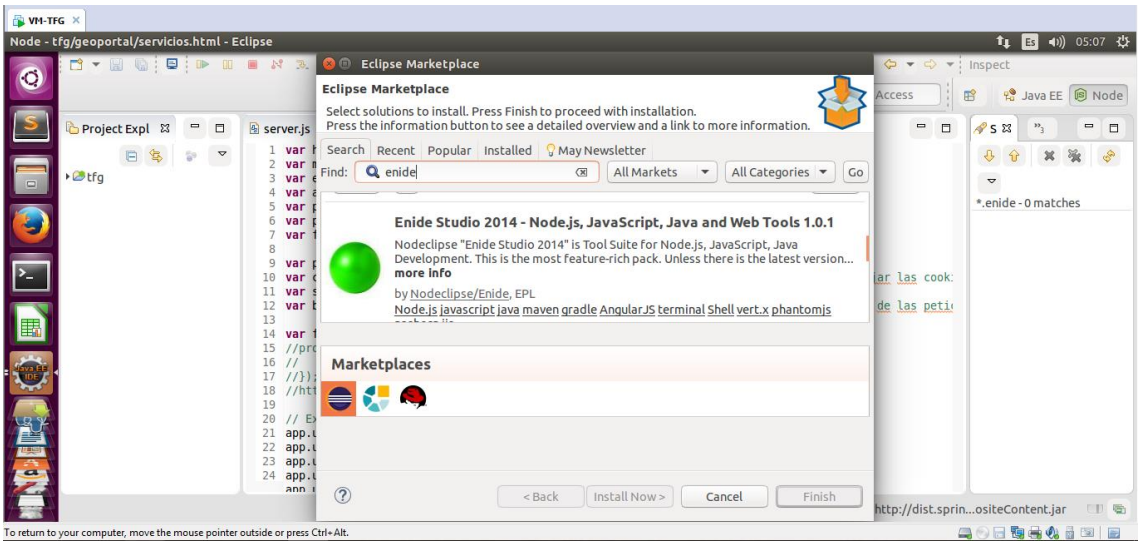


Imagen 43 - Búsqueda de Enide en el "marketplace" de Eclipse.

Como vemos desde la perspectiva de *Node*, una vez instalado *Enide*, podremos crear diversos proyectos de Inicio de *NodeJS*. Estos proyectos de Inicio generalmente conocidos como "*Hola Mundo*" sirven para ilustrar la sintaxis básica del lenguaje de programación en cuestión.

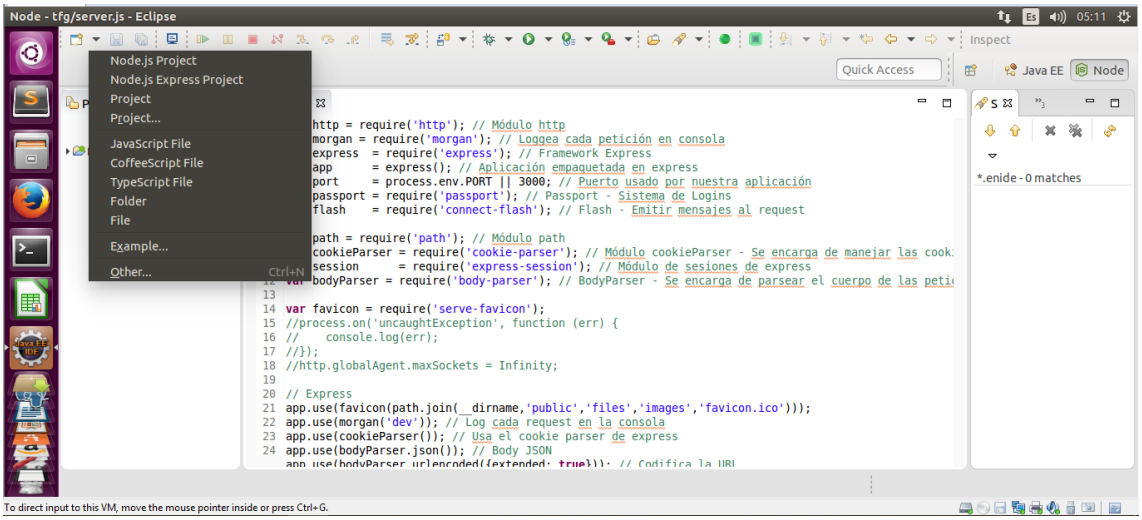


Imagen 44 - Eclipse abierto con la perspectiva Enide. Opciones para iniciar proyectos con Enide.

Se puede observar que nos da la opción de crear un proyecto "*Node.js Express*". *Express* es un *framework* para la realización de aplicaciones web para *NodeJS* del cual se hablará más adelante. Esta opción es muy ventajosa ya que nos crea por defecto una estructura de carpetas y archivos básica para un proyecto con el *framework Express*. Aunque por conveniencia, ya que un proyecto en *NodeJS* no sigue ninguna regla a la hora de estructurarlo, dejaremos el diseño para más adelante y lo haremos manualmente.

5.6. NODEJS, ESPRESSJS, SOCKET.IO Y OTROS MÓDULOS.

NodeJS se programa del lado del servidor, lo que indica que los procesos para el desarrollo de software en "*Node*" se realizan de una manera muy diferente a los de *Javascript* del lado del cliente.

De entre alguno de los conceptos que cambian al estar NodeJS del lado del servidor es el asunto del *"Cross Browser"*, que indica la necesidad en el lado del cliente de hacer código que se interprete bien en todos los navegadores. Cuando trabajamos con *Node* solamente necesitamos preocuparnos de que el código que se escriba se ejecute correctamente en el servidor.

Otro aspecto relevante es la programación asíncrona y la programación orientada a eventos, con la particularidad que los eventos en esta plataforma son orientados a cosas que suceden del lado del servidor (p.e. se ha eliminado un fichero) y no del lado del cliente como por ejemplo el evento *click* del ratón.

Además, *NodeJS* implementa los protocolos de comunicaciones en redes más habituales, de los usados en Internet, como puede ser el *HTTP, DNS, TLS, SSL*, etc. y brinda uno de los más extensos repositorios teniendo en cuenta su corta vida.

- **Instalación de NodeJS**

La instalación de *NodeJS* se realizará a través del terminal, mediante el comando *apt-get* de la siguiente forma:

```
jose@TFG:~$ sudo apt-get update && sudo apt-get install nodejs
```

Imagen 45 - Comando para la instalación de NodeJS.

Este comando, tras autentificarnos con la contraseña de nuestro usuario en el sistema (al haber empleado el prefijo sudo), aplicará actualizaciones y posteriormente decargará e instalará la última versión de *NodeJS* en nuestro sistema operativo.

Tras la ejecución y para verificar que realmente *NodeJS* está instalado, lo comprobaremos de la siguiente forma:

```
jose@TFG:~$ nodejs -v
v5.11.1
jose@TFG:~$ npm -v
3.6.0
jose@TFG:~$
```

Imagen 46 - Comprobación de las versiones instaladas de Node y NPM.

Efectivamente comprobamos que *NodeJS* y *NPM* están correctamente instalados con las versiones *5.11.1* y *3.6.0* respectivamente. *NPM* (*Node Package Manager*) es el manejador de paquetes de *NodeJS*. Si lo deseamos también podemos testear algún código JavaScript desde el terminal ejecutando previamente *NodeJS* para acceder a su Interfaz de línea de comandos (*Command Line Interface* o *CLI*):

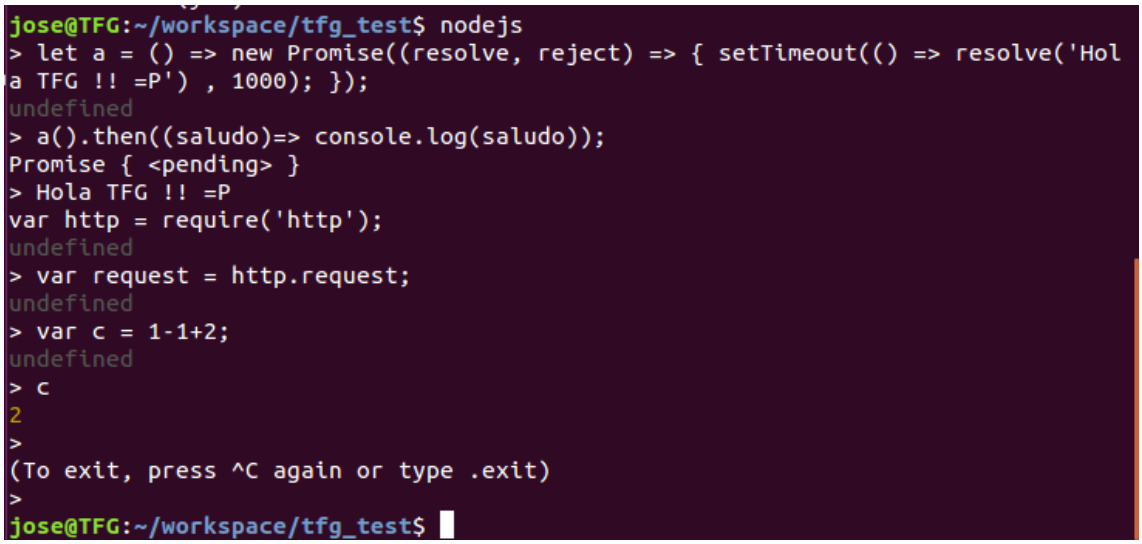


Imagen 47 - Ejecución de código JavaScript en el terminal a través de NpdeJS.

- **Creación del directorio de trabajo para nuestro proyecto**

Tras verificar anteriormente que *NPM* también está instalado, se procederá a crear un directorio de trabajo y crearemos el archivo *package.json* fundamental en cualquier aplicación *NodeJS* mediante el comando "*npm init*". El archivo *package.json* contiene los metadatos de nuestro proyecto y entre esos metadatos se incluyen todas las dependencias (módulos instalados) de nuestro proyecto. Si deseáramos proveer nuestro proyecto a terceros no necesitaría que le pasáramos todas las dependencias ya que con este archivo podríamos instalarlas mediante el uso del comando "*npm install*".

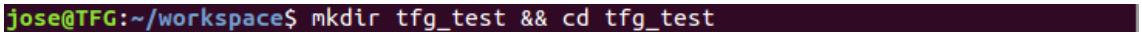


Imagen 48 - Comando para crear un directorio y acceder a él.

- **Instalación de los módulos externos necesarios**

El primer módulo que vamos a instalar, *ExpressJS*, es el más importante ya que nos provee de una base para realizar aplicaciones web cuyos aspectos más destacables son una API para enrutamiento, un *parser* de las peticiones *HTTP*, funciones *middleware* y un motor de vistas. *NodeJS*, en su nucleo posee un módulo llamado "*http*" para crear instancias de un servidor HTTP, pero si quisiéramos desarrollar una aplicación web utilizando solo este módulo y/o solo los módulos del núcleo de *NodeJS*, mucho código de bajo nivel "*low-level code*" como *parsear* las cookies, guardar sesiones, seleccionar el patrón correcto para cada ruta en base a expresiones regulares, etc. tendría que recaer en nuestras manos. En definitiva nos facilita la creación de aplicaciones web o servicios de una forma rápida y segura. De ahí la necesidad de *NPM* y la instalación de módulos que faciliten el trabajo.

Así pues, instalamos el módulo de *ExpressJS* mediante *NPM* utilizando los *flags -g* para hacer una instalación global (De esta forma podremos requerir el módulo *Express* en cualquier proyecto que creemos) y *–save* dentro de la carpeta de nuestro proyecto para incluir esta dependencia en el archivo *package.json* de nuestro proyecto. En ambas se utilizará el decorador *@latest* para instalar las últimas versiones de cada módulo.

```
jose@TFG:~/workspace/tfg_test$ sudo npm install express -g
loadDep:ms → network
```

Imagen 49 - Comando para la instalación de ExpressJS de forma global.

Al final de cada proceso de instalación mostrará dependencias adicionales que se han instalado.

```
jose@TFG:~/workspace/tfg_test$ npm install express --save
tfg_test@1.0.0 /home/jose/workspace/tfg_test
└─┬ express@4.13.4
  ├─┬ accepts@1.2.13
  │ ├─┬ mime-types@2.1.11
  │ │ └── mime-db@1.23.0
  │ └── negotiator@0.5.3
  ├── array-flatten@1.1.1
  ├── content-disposition@0.5.1
  ├── content-type@1.0.2
  ├── cookie@0.1.5
  ├── cookie-signature@1.0.6
  ├─┬ debug@2.2.0
  │ └── ms@0.7.1
  ├── depd@1.1.0
  ├── escape-html@1.0.3
  ├── etag@1.7.0
  ├─┬ finalhandler@0.4.1
  │ └── unpipe@1.0.0
  ├── fresh@0.3.0
  ├── merge-descriptors@1.0.1
  ├── methods@1.1.2
  ├─┬ on-finished@2.3.0
  │ └── ee-first@1.1.1
```

Imagen 50 - Comando para la instalación de ExpressJs de forma local con la opción "--save" para incluir la dependencia en el "package.json". Árbol de carpetas creadas tras la instalación.

También se habrá creado el directorio *node_modules*, directorio donde se almacenará cualquier módulo instalado mediante *NPM* en nuestro directorio de trabajo. Por el momento tendremos todas las dependencias que se han instalado con *ExpressJS*.

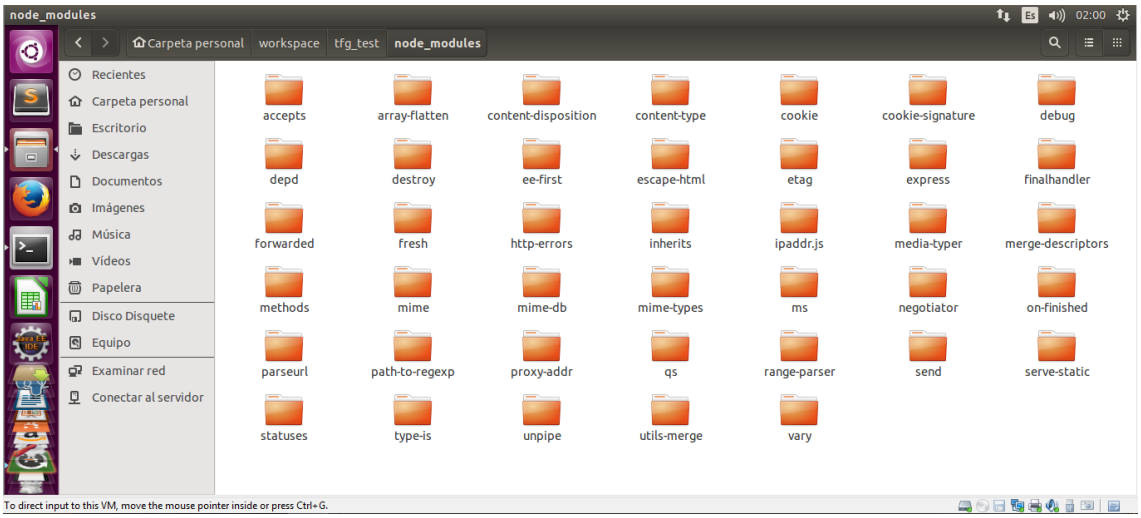


Imagen 51 - Vista de la carpeta "node_modules" creada tras la instalación de ExpressJS.

Otro módulo que merece más detenimiento es *Socket.IO*, el cual nos brinda con una *API* para utilizar *websockets* en nuestro servidor para una comunicación cliente-servidor en tiempo real (También posee de una librería *JavaScript* necesaria en el cliente).

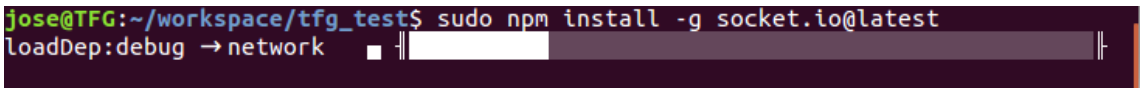


Imagen 52 - Comando para la instalación de Socket.IO.

Para la comunicación con *PostgreSQL* existe un módulo en *NPM* especialmente útil, *pg-promise*. Nos brinda una interfaz que utiliza "*Promise*" para la realización de llamadas asíncronas a la base de datos. "*Promise*" es una clase definida en JavaScript a partir de la especificación *ECMASCRIPT6 o ES6* y que viene a solucionar varios problemas a la hora de escribir código asíncrono en JavaScript como por ejemplo el uso de funciones *callback* anidadas. Con *pg-promise* no nos tenemos que preocupar de hacer las operaciones de "*COMMIT*" tras cada consulta, pues se encarga de iniciar y cerrar las conexiones. Nos brinda de funciones para realizar múltiples tareas sobre una base de datos y varias clases y espacios de nombre para formatear correctamente las consultas y prevenir la inyección *SQL*.

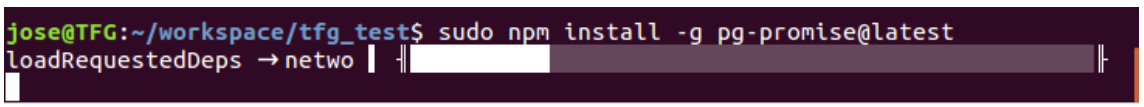


Imagen 53 - Comando para la instalación de PG-Promise.

Otros módulos externos utilizados se detallan en la siguiente tabla:

NOMBRE	DESCRIPCIÓN
VALIDATOR	Sirve para realizar comprobaciones en cadenas de texto.
CONNECT-FLASH	Permite flashear al cliente mensajes a través de una petición.
SERVE-FAVICON	Middleware que nos permite servir el famoso “favicon”.
I18N-2	Middleware que no permite establecer un “locale” y servir la aplicación en el idioma seleccionado.
BCRYPT-NODEJS	Permite crear y comparar cadenas de texto encriptadas.
BLUEBIRD	Wrapper de la clase Promise que añade ciertas funciones a la clase nativa.
PG-MONITOR	Add-on de pg-promise. Permite loguear eventos de la comunicación con la bdd en la consola.
MULTER	Middleware empleado para la gestión de subidas de imágenes.

Tabla 1 - Otros módulos NPM instalados.

5.7. LIBRERÍAS JAVASCRIPT DEL LADO DEL CLIENTE

En el proceso de diseño e implementación de un cliente web es necesario analizar qué componentes son los necesarios para realizar lo que se propone. También debemos de cuidar el diseño y hacer que sea lo más fácil de usar posible para mejorar la experiencia del usuario.

Todo esto lo resuelven algunos librerías que nos proveen componentes (*JavaScript*), estilos (*CSS, Less, Sass*), etc., en definitiva una base para realizar el diseño de las páginas web. Un ejemplo de este tipo de librerías es Bootstrap.

También existen algunas librerías como *jQuery* que ayudan a realizar modificaciones sobre el documento *HTML* o acceder a estos elementos de distintas formas. En este caso *jQuery* viene a acortar con su sintaxis código que puede ser creado igualmente con puro *JavaScript*.

En un escalón por encima están aquellos *frameworks* con los que se pueden realizar completas aplicaciones *MVC* (Modelo Vista Controlador) y que en algunos casos pueden ser exportadas a aplicaciones móviles nativas (*IOs, Android y Windows Phone*) como *Angular, Cordova, Ionic (Cordova + Angular), React.*

En la realización de este proyecto se han usado distintos tipos de librerías JavaScript. Algunas de las usadas pertenecen a proyectos de gente anónima que ofrece sus productos a la comunidad de forma altruista. A continuación se detalla una lista con las librerías más importantes usadas en este proyecto.

NOMBRE	REPOSITORIO
OpenLayers 3	https://github.com/openlayers/ol3
ol3-ext	https://github.com/Viglino/ol3-ext
OL3-AnimatedCluster	https://github.com/Viglino/OL3-AnimatedCluster
ol3-geocoder	https://github.com/jonataswalker/ol3-geocoder
ol3-contextmenu	https://github.com/jonataswalker/ol3-contextmenu
Bootstrap	https://github.com/twbs/bootstrap
bootstrap-select	https://github.com/silviomoreto/bootstrap-select
bootstrap-sidebar	https://github.com/asyraf9/bootstrap-sidebar
bootstrap-tagsinput	https://github.com/bootstrap-tagsinput/bootstrap-tagsinput
bootstrap3-dialog	https://github.com/nakupanda/bootstrap3-dialog
highlight.js	https://github.com/iominh/highlight.js
js-cookie	https://github.com/js-cookie/js-cookie
Multiselect	https://github.com/crlcu/multiselect
jQRangeSlider	https://github.com/ghusse/jQRangeSlider
share-button	https://github.com/carrot/share-button
jquery.appear	https://github.com/bas2k/jquery.appear
Modernizr	https://github.com/Modernizr/Modernizr

Tabla 2 - Librerías JavaScript utilizadas para realizar el cliente.

5.8. LIBRERÍA GDAL

Geospatial Data Abstraction Library (GDAL/OGR) es una librería multiplataforma escrita en *C++* para datos tipo geoespacial, ráster y vectorial.

Está publicada bajo una licencia de código abierto por el *Open Geospatial Consortium*.

Viene con una variedad de utilidades en la línea de comandos para el procesado de datos y es el motor de acceso a los datos para una gran cantidad de aplicaciones como *MapServer, QGIS,* etc. Soporta alrededor de 50 formatos distintos para archivos de tipo ráster y 20 para vectorial.

En el proyecto solo se utilizará para pregenerar las teselas de la ortofoto y evitar así que este proceso lo haga el servidor de mapas (*Geoserver*). (Wikipedia, s.f.)

Imagen 54 - Logo de la librería GDAL.

6. DESARROLLO DE UNA IDE Y GEOPORTAL

En este apartado se describirán los procesos que se han llevado a cabo para el desarrollo de la Infraestructura de datos espaciales y un Geoportal que dará acceso a nuestros contenidos e implementará ciertas funcionalidades con las que el usuario podrá interactuar con la información geográfica almacenada.

6.1. DATOS DE PARTIDA

La cartografía empleada se ha obtenido de diversas fuentes que distribuyen información geográfica de forma libre y gratuita. Los distintos orígenes de datos utilizados son:

- Cartografía Vectorial del proyecto CartoCiudad del IGN (Formato SHP).
- Ortofoto a escala 1:5000 de la Comunidad Valenciana del año 2012 (Formato ECW)
- Cartografía existente en el ayuntamiento de Torrent (WMS)
- Ortofotos del PNOA (WMS)

La fecha de actualización de la cartografía vectorial procedente de CartoCiudad la podemos consultar de una manera rápida desde su propia página web.

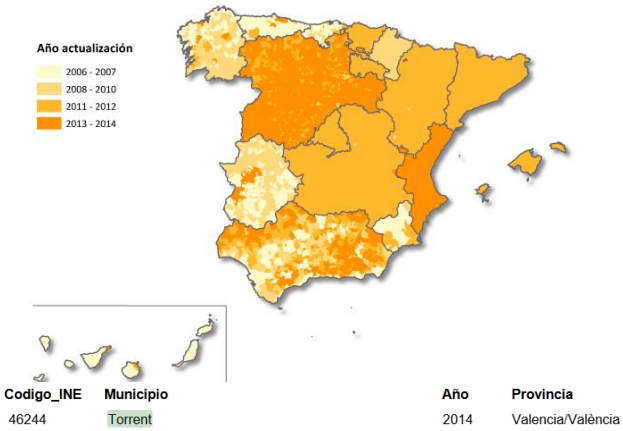


Imagen 55 - Mapa de última fecha de actualización de CartoCiudad. Para Torrent el año 2014.

La cartografía que obtenemos de *CartoCiudad*, proporcionada por el *Centro de Descargas del Centro Nacional de Información Geográfica (CNIG)*, contiene la información para toda la Comunidad Valenciana, es por ello que necesita ser sometida a un proceso de recorte por el municipio, una tarea sencilla con cualquier *SIG* de Escritorio como *QGIS, gvSIG*, etc.

También se podría introducir toda la información en la base de datos *PostGIS* y aplicar las mismas operaciones de recorte sobre nuestras capas mediante consultas espaciales.

En este caso el software elegido por simplicidad ha sido *gvSIG*.

Las capas que se generan tras el proceso de recorte son:

- **manza_pol** → Contiene información acerca de las manzanas catastrales.
- **portal_pk_pun** → Contiene información acerca de los portales de cada manzana catastral.
- **topónimo_pun** → Contiene información relativa a la toponimia.
- **torrent_pol** → Contiene el polígono formado por los límites administrativos de torrent.
- **tramos_vial_lin** → Contiene información acerca de las carreteras, sendas, etc.

La ortofoto se ha conseguido a través de *Terrasit*, la IDE de la Comunidad Valenciana. Para ello es necesario registrarse previamente para acceder a este tipo de contenidos.

Imagen 56 - Página de descargas del Terrasit (IDE de la Comunidad Valenciana).

La ortofoto también ha sido recortada en *gvSIG* y exportada posteriormente a formato *TIF*. Para que la ortofoto funcione con *Geoserver* de una manera óptima es necesario crear mosaicos y pirámides al archivo ráster. Para ello usaremos la librería GDAL que viene con ciertos comandos para realizar este tipo de tareas, entre muchas otras.

El comando utilizado para obtener un archivo *GeoTiff* que contenga mosaicos es el siguiente:

gdal_translate –a_srs EPSG:25830 –of GTiff –co “TILED_YES” entrada.tif salida.tif

La opción *–a_srs* especifica el sistema de referencia de salida, *-of* el formato de salida y *–co* recibe opciones, en este caso que genere mosaicos. Recibe el fichero de entrada y la ruta de salida para el fichero generado.

Las pirámides se generan a partir del fichero de salida de la operación anterior y se utiliza para ello el siguiente comando:

gdaladdo –r average entrada.tif 2 4 8 16 32 64

La opción *–r* especifica un algoritmo de remuestreo, en este caso se indica que calcule la media. Recibe como primer argumento el archivo de entrada y el resto de argumentos son los factores de zoom, los cuales se han calculado mediante potencias de 2 hasta 64.

El resultado de esta operación sobrescribirá el archivo de entrada y será la ortofoto final que emplearemos en *Geoserver*.

También se ha obtenido cartografía que ya existe en el ayuntamiento de Torrent y que es accesible mediante servicios *WMS* y *WFS*. De este servicio solo se utilizarán las capas que añadan más información que la obtenida con lo anteriormente nombrado y por lo tanto se descartarán aquellas que sean redundantes o cuya información sea escueta o intranscendente para los ámbitos del proyecto.

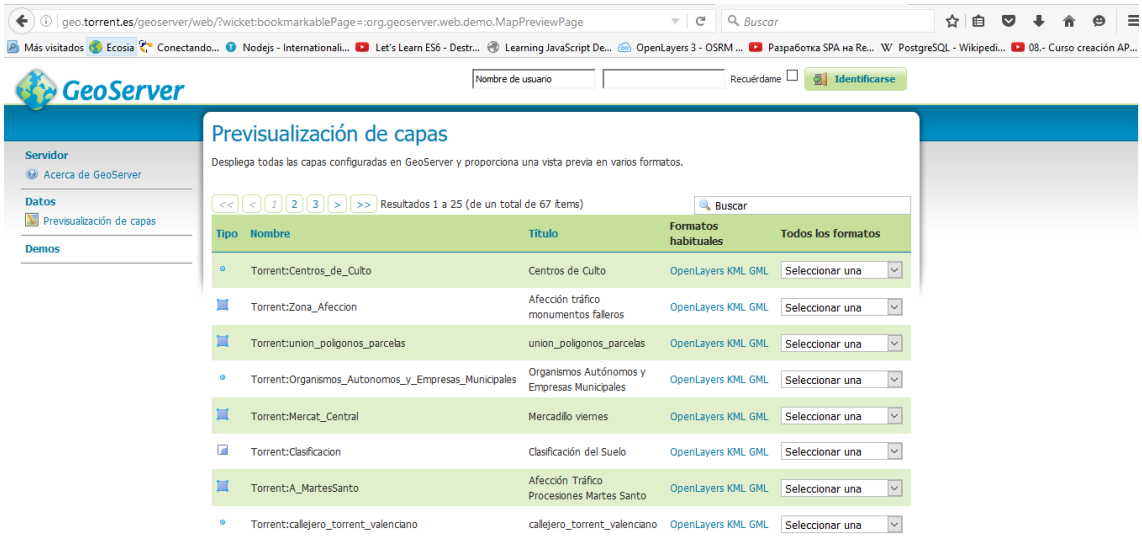


Imagen 57 - Algunas capas del WMS en cascada con el ayuntamiento de Torrent.

Cabe destacar que el *Geoportal* de Torrent no tiene un acceso visible a su servidor de información geográfica, que es *GeoServer*, pero su dirección se puede obtener fácilmente analizando las peticiones que el propio visor de cartografía manda al servidor desde el menú de cualquier navegador (Apartado de Redes).

Otro servicio *WMS* utilizado como servidor en cascada es el *WMS* del *PNOA* (*Plan Nacional de Ortofotogrametría Aérea*) para obtener una ortofoto con mayor resolución.

Una vez procesados los datos de salida, ya están listos para ser añadidos a un almacén de datos en *Geoserver* y publicar las capas. En el caso de la cartografía vectorial, se almacenará previamente en una base de datos *PostGIS* y será *Geoserver* el que se conecte a *PostGIS* para obtener dichas capas. Así pues estaremos usando *PostGIS* como repositorio de las capas vectoriales en lugar de que *Geoserver* obtenga las capas desde una carpeta de nuestro sistema. Tener la cartografía almacenada en una base de datos *PostGIS* hace que podamos utilizar cualquiera de sus funciones sobre ella, servirla a través de la web sin necesidad de otro intermediario y nos ayudará más adelante en la elaboración de ciertas partes de la aplicación.

Para importar los *shapefiles* a *PostgreSQL* es necesario un *plugin* llamado "**shp2pgsql**". Con este *plugin* podemos ejecutar el siguiente comando para importar una capa en formato *.shp* a *PostgrSQL*:

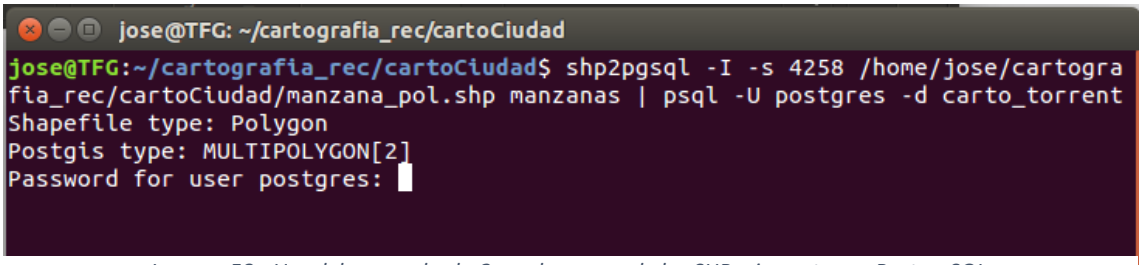


Imagen 58 - Uso del comando shp2pgsql con una de los SHP a importar en PostgreSQL.

El comando anterior, tras autenticarnos con el usuario, creará una tabla llamada manzanas en la base de datos "*carto_torrent*", mencionada en el apartado 4.3., y tomará como archivo de entrada la capa *manza_pol.shp*, la cual especificamos su sistema de referencia, "*EPSG:4258*".

Se repite el comando anterior para cada una de las capas restantes, resultando en 5 tablas creadas en la base de datos "*carto_torrent*".

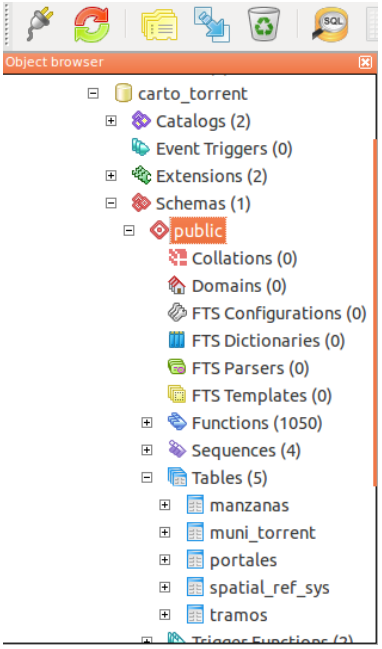


Imagen 59 - Tablas que se han creado tras la importación de los SHP.

Llegado a este punto, todo el trabajo se vuelca sobre *Geoserver*, creando cada almacén de datos y publicando las capas para que estén accesibles desde nuestros servicios, para se accede a la sección "*Almacenes de Datos*" en *Geoserver* y hacemos *click* en la opción "*Agregar nuevo almacén*".

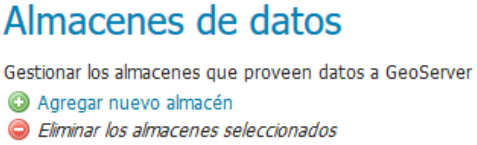


Imagen 60 - Opción para crear un nuevo almacén de datos.

Los almacenes a crear son los siguientes:

- **carto_torrent** → Origen de datos *PostGIS*, se conecta a la base de datos "*carto_torrent*" para obtener las capas.
- **ortofoto** → Orígen de datos Ráster, obtiene la ortofoto de un directorio en nuestro sistema.
- **wms_cascada_orto** → Orígen de datos *WMS*, obtiene la capa ortofoto del servicio WMS del PNOA (IGN).
- **ajuntament_torrent_capas** → Orígen de datos *WMS*, obtiene las capas del servicio WMS del ayuntamiento de Torrent.
- **denuncias** → Orígen de datos *PostGIS*, se conecta a la base de datos "*denuncias*", se explicará en el siguiente aprtado referente al desarrollo de la aplicación.

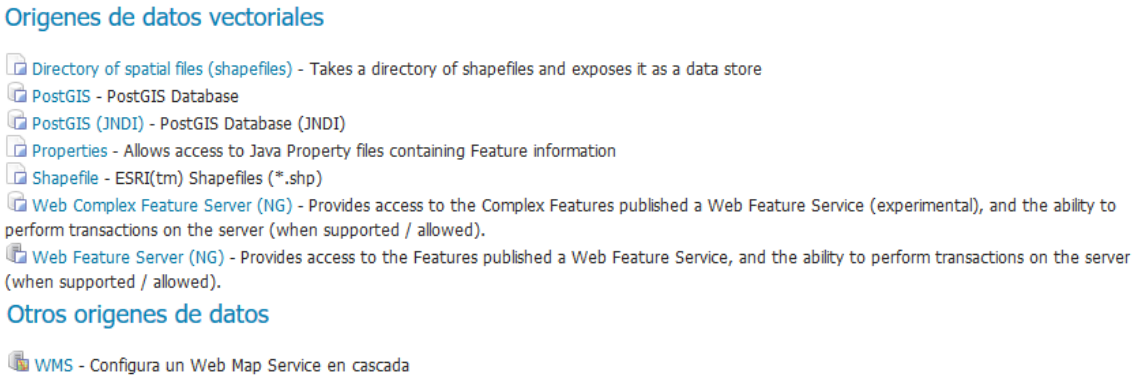


Imagen 61 - Tipos de orígenes de datos que acepta Geoserver para crear el almacén.

La siguiente imagen muestra los parámetros de conexión para los almacenes de datos "*carto_torrent*" y "*ortofoto*" cuyos orígenes de datos son una base de datos *PostGIS* y un archivo ráster, respectivamente. Cabe indicar que todo está siendo asignado al único espacio de trabajo creado "*jahr*".

Imagen 62 - Parámetros de conexión para los almacenes "carto_torrent" y "ortofoto".

Por último se muestra los parámetros de conexión para los almacenes de datos "*wms_cascada_orto*" y "*ajuntament_torrent_capas*", cuyos orígines de datos son servicios *WMS* en cascada.

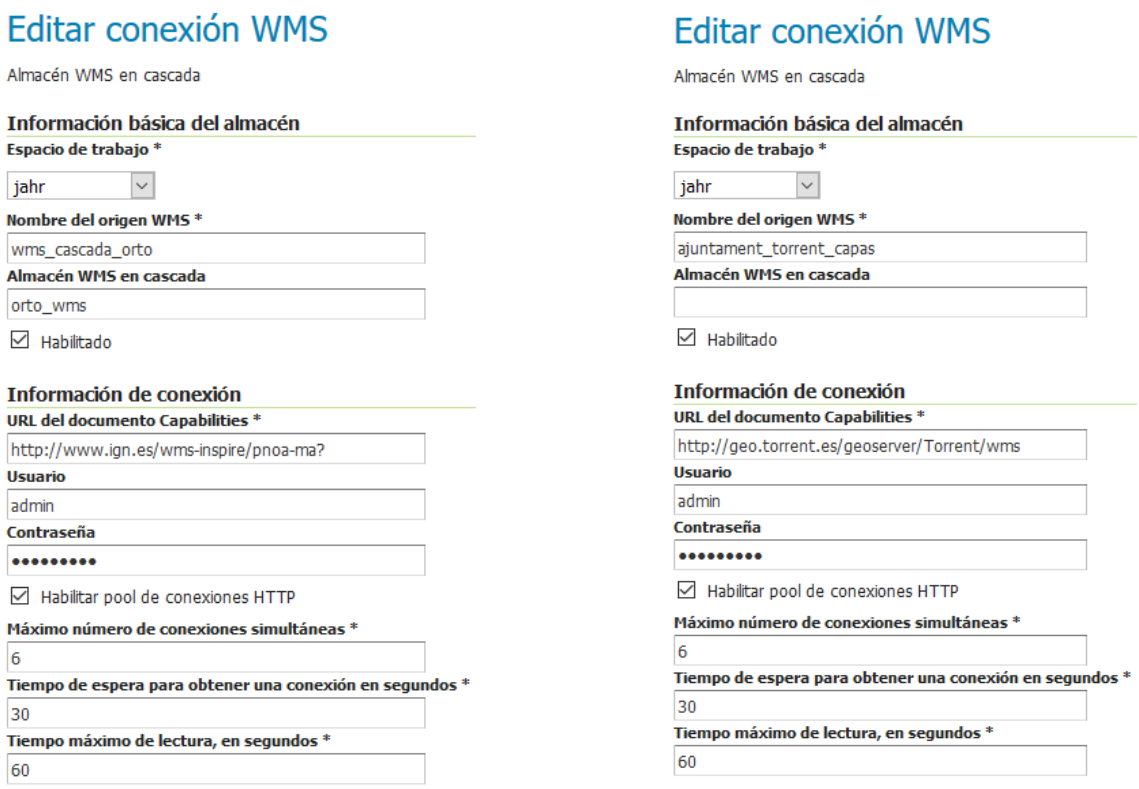


Imagen 63 - Parámetros de conexión para los almacenes "wms_cascada_orto" y "ajuntament_torrent_capas".

6.2. ESTILOS SLD

Para la creación de los estilos, *GeoServer* admite un lenguaje llamado SLD (*Styled Layer Descriptor*) el cual es un lenguaje de marcas que propone el *OGC*.

Alternativamente, se puede descargar un *plugin* de *Geoserver* para escribir los estilos en un lenguaje muy similar al CSS. Este *plugin*, dota a *GeoServer* de diversas funcionalidades nuevas y convertirá el código que escribamos en *SLD*. Se puede descargar desde su página oficial e instalarlo es tan sencillo como volcar el contenido de la descarga en la carpeta *WEB-INF/lib* de *GeoServer* y reiniciar el servicio de *Tomcat*.

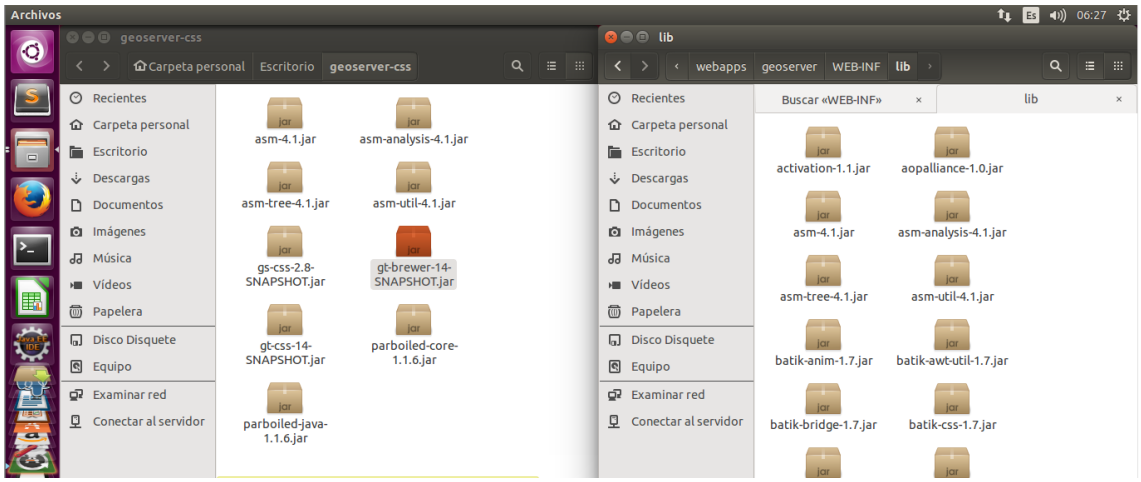


Imagen 64 - A la izquierda el contenido de la descarga y a la derecha la carpeta "lib" de Geoserver donde se almacenarán los plugins que se instalen.

Esto abrirá una opción "*Estilos CSS*" en el menú izquierdo de navegación que llevará a un editor de texto donde podremos generar los estilos, previsualizarlos y ver el código *SLD* generado.

Imagen 65 - Creación de un estilo "CSS" para una capa.

A continuación se muestra una lista detallada de los estilos creados para cada capa.

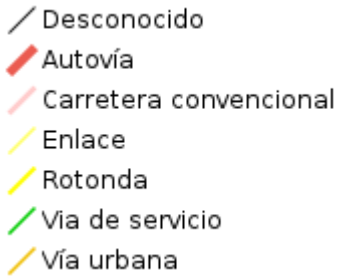
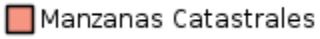
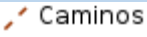
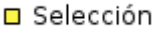
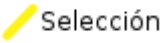
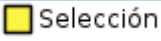
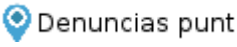
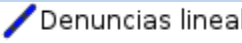
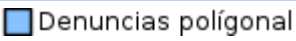
NOMBRE	LEYENDA	PROPIEDADES
Tramos		Estilo para elementos lineales. Muestra un estilo para cada tipo de vía. Excepto las autovías, los demás elementos se mostrarán a partir de las escala 1:5.000. Será el estilo a aplicar en la capa **viales**.
Portales		Estilo para elementos puntuales. Escala máxima para su visualización 1:2000. Será el estilo a aplicar en la capa **portales**.
manzanas		Estilo para elementos poligonales. Será el estilo a aplicar en la capa **manzanas**.
caminos		Estilo para elementos lineales. Será el estilo a aplicar en la capa **caminos**.
vial_nombres		Estilo para elementos lineales. Muestra el texto del atributo nom_vial. Escala máxima para su visualización 1:2000. Será el estilo a aplicar en la capa **viales_nombres**.
muni_torrent		Muestra un estilo según tres rangos de escalas. 1:15.000 y escalas inferiores mostrará el primer estilo. Entre 1:15.000 y 1:10.000 mostrará el tercer estilo y de 1:10.000 a escalas superiores mostrará el segundo estilo. Será el estilo a aplicar en la capa **muni_torrent**.
punto_sel		Estilo para elementos puntuales. Sirve para resaltar un elemento al ser seleccionado.
linea_sel		Estilo para elementos lineales. Sirve para resaltar un elemento al ser seleccionado.
poli_sel		Estilo para elementos poligonales. Sirve para resaltar un elemento al ser seleccionado.
denuncias_puntos		Estilo para elementos puntuales. Será el estilo a aplicar en la capa **denuncias_puntos**.
denuncias_lineas		Estilo para elementos lineales. Será el estilo a aplicar en la capa **denuncias_lineas**.
denuncias_poligonos		Estilo para elementos poligonales. Será el estilo a aplicar en la capa **denuncias_poligonos**.

Tabla 3 - Estilos creados en Geoserver.

6.3. PUBLICACIÓN DE CAPAS

La publicación de las capas en *GeoServer* se hace desde el apartado "*Capas*", donde nos da una opción para añadir una nueva capa.

Lo primero que pregunta *GeoServer* es de qué almacén se quiere agregar una capa y eligiendo un almacén nos muestra todas las capas que contiene dicho almacén.

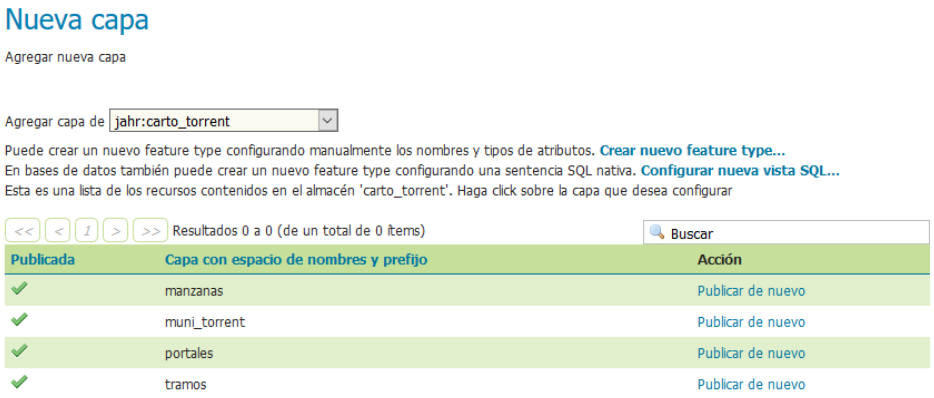


Imagen 66 - Formulario para la creación de una nueva capa.

Como se puede observar en la imagen anterior se puede publicar una capa cuantas veces se desee, siempre que tengan nombres distintos dentro del mismo almacén.

Al hacer *click* sobre "*publicar*" en alguna de las capas, llevará a un formulario en el que pide información para la publicación de la capa. En la siguiente imagen se muestran los parámetros de configuración más importantes para la capa "manzanas":

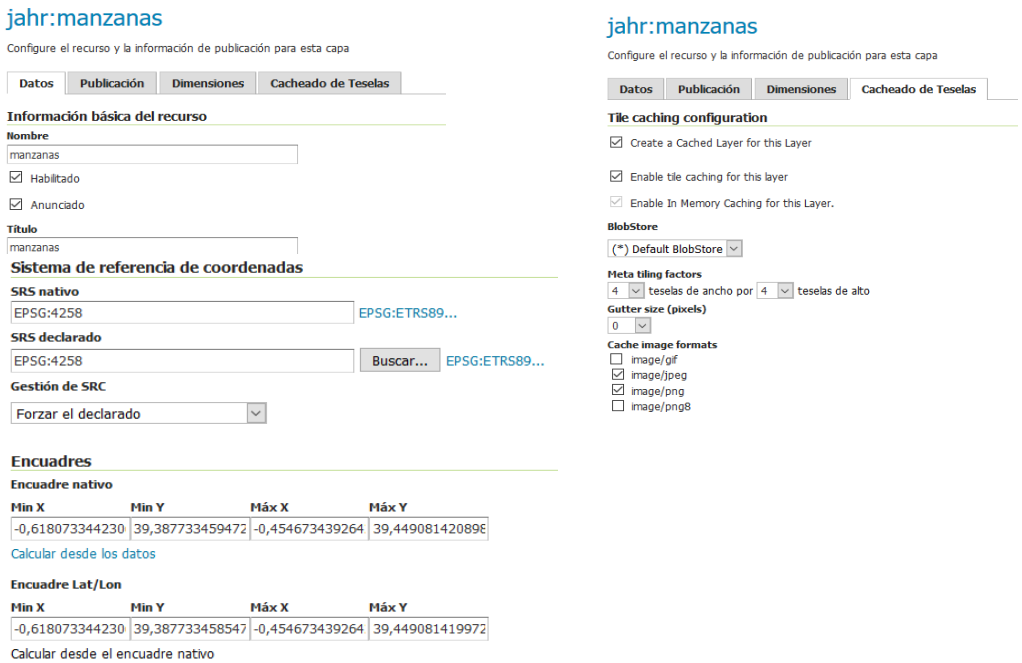


Imagen 67 - Formulario para la publicación de una capa.

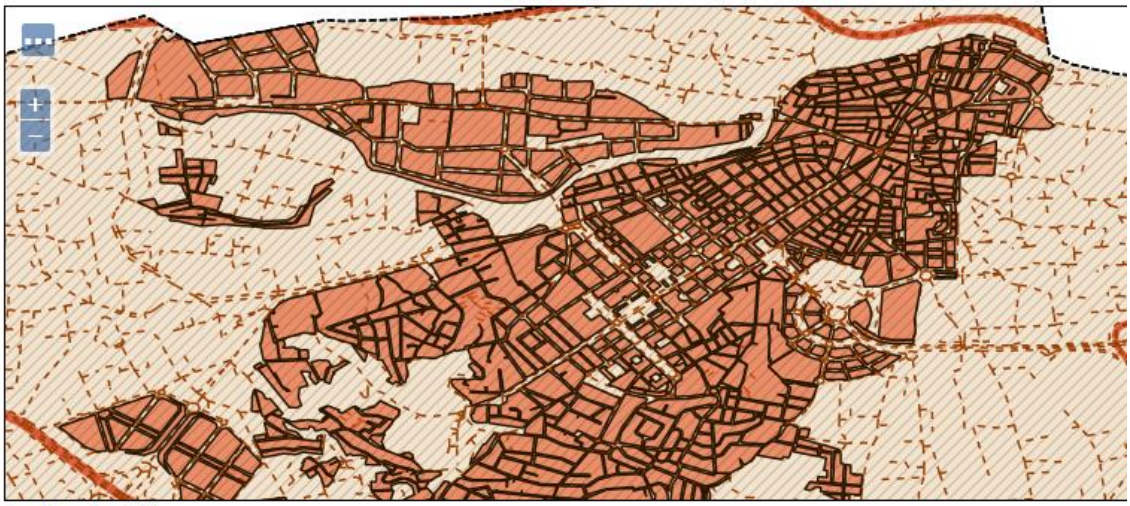
Al aceptar el formulario, si todo está correctamente rellenado, ya estará publicada la capa manzanas.

Este proceso se lleva a cabo con los siguientes almacenes de trabajo y capas:

- Almacén "**carto_torrent**" → Constituirá la cartografía base de la IDE. Capas: portales, muni_torrent, viales, caminos, manzanas y nombres_viales.
- Almacén "**ortofoto**" → Capas: ortofoto.
- Almacén "**wms_cascada_orto**" → Capas: OI.OrthoImageCoverage.
- Almacén "**ajuntament_torrent_capas**" → Complementará la cartografía base. Capas: Parques_Y_Jardines, Farmacias, Policia, TorrentBici, Lugares_Interes, Deportes, etc.

Al publicar una capa asignaremos los estilos creados en el apartado anterior para cada capa, exceptuando las capas de los servidores en cascada, cuyos estilos de capas ya están definidos en el servidor de origen y la ortofoto, que no posee ningún estilo.

Finalmente, se irá al apartado "*Previsualización de capas*" en *GeoServer* y se comprobará que todo funciona como se espera.



Scale = 1 : 34K

caminos

fid	id_tramo	id_vial	ine_via	dgc_via	tipo_via	tipo_v_des	tip_via_in	nom_via	nom_altern	non
caminos.766	4.62440106573E11	4.62440001274E11	4624400067	-998.0	1.0	Vía urbana	CALLE	ALAMEDA REINA DOÑA SOFIA	-997	1/01
caminos.4729	4.6244010665E11	4.62440001288E11	4624403469	-998.0	1.0	Vía urbana	CALLE	CIUDAD DE EL ALTO	-997	1/01

manzanas

Imagen 68 - Previsualización de algunas de las capas publicadas.

6.4. CREACIÓN DE LOS METADATOS

Para llevar a cabo la tarea de realizar los metadatos de los datos y los servicios creados, *CatMDEdit*, el cual es software libre, hace que el trabajo sea más fácil ya que implementa el perfil del *NEM* (*Núcleo Español de Metadatos*), a parte de los perfiles de las normas *ISO* y *Dublin Core* referentes a la información geográfica.

También provee de información de cada uno de los ítems propuestos para su relleno como:

- Saber si un ítem es obligatorio, no obligatorio o condicional y en caso de la última qué condición debe cumplirse para no ser obligatorio según el perfil utilizado.
- Descripción de cada ítem y ejemplos.
- Relleno automático de ciertos ítems.

Utilizando *CatMDEdit*, se crearán los metadatos para cada uno de los servicios creados (*WMS, WFS, WMTS, WCS y CSW*) y para cada uno de los recursos ráster (*Ortofoto recortada*) y vectorial (*Capas recortadas de CartoCiudad*) que se han procesado en el apartado 5.1.

A continuación se muestra una tabla con información de cada uno de los metadatos creados:

Estándar	Id único	Idioma	Título	Tipo	Tema	Responsable
ISO19119	spaupvWMTSWebMapTileService10052016000	español	Servicio web de mapas Teselados del ayuntamiento de Torr…	Servicio		UPV
ISO19119	spaupvwmsWebMapService10052016000	español	Servicio Web de Mapas del Ayuntamiento de Torrent	Servicio	Map access service	UPV
ISO19119	spaupvWFSWebFeatureService10052016000	español	Servicio de Descarga del ayuntamiento de Torrent	Servicio		UPV
ISO19119	spaupvWCSWebCoverageService10052016000	español	Servicio WCS del ayuntamiento de Torrent	Servicio	WMS	UPV
ISO19115	spaupvviales10052016	español	Viales	Conjunto de da…	Transporte	UPV
ISO19115	spaupvtoponimia10052016	español	Etiquetado calles	Conjunto de da…	Localización	UPV
ISO19115	spaupvPortal10052016	español	Número de portales de las edificaciones de Torrent	Conjunto de da…	Localización	UPV
ISO19115	spaupvmunitorrent10052016	español	Municipio de Torrent	Conjunto de da…	Límites	UPV
ISO19115	spaupvManza10052016	español	Manzanas Catastrales de Torrent.	Conjunto de da…	Planeamiento Cat…	UPV
ISO19119	spaupvCSWCatalogueServiceForTheWeb10052016000	español	Servicio de localización del Ayuntamiento de Torrent	Servicio	Europe	UPV

Imagen 69 - Listado de los metadatos creados.

Se usa el perfil *NEM 1.2* y el *NEM 1.2 –S* para el relleno de los metadatos. En el caso de los metadatos de datos cartográficos, podemos importar los archivos a *CatMDEdit* y rellenará ciertos ítems automáticamente.

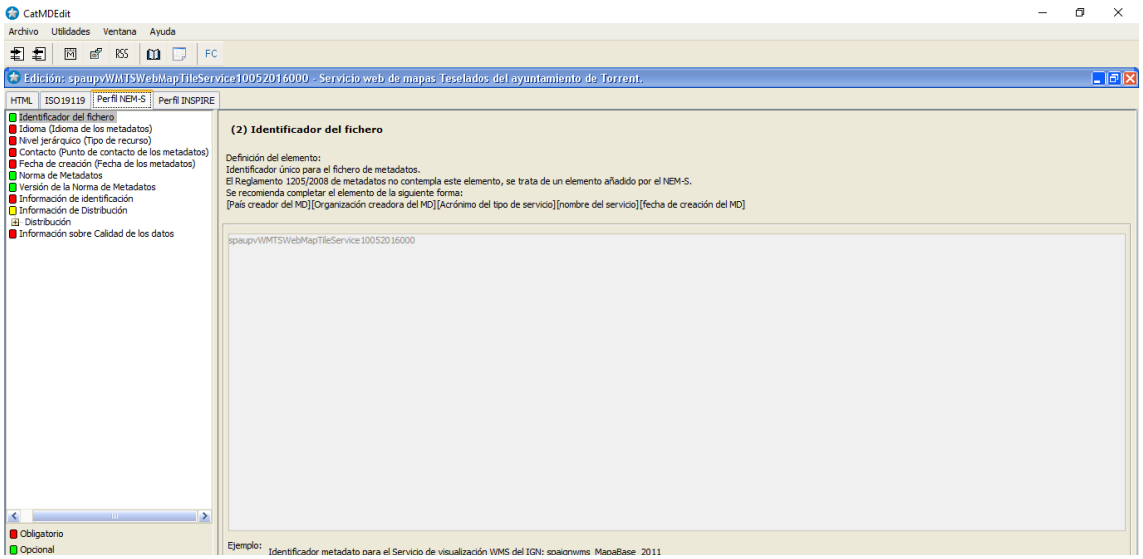


Imagen 70 - Ejemplo del relleno para los metadatos de un servicio.

Una vez creados los metadatos, hay que importarlos al servicio de catálogo para que estén accesibles, para ello se accede a *Geonetwork* como administrador al menú de herramientas de administración y se hace *click* en la sección para importar metadatos, que llevará al siguiente formulario:

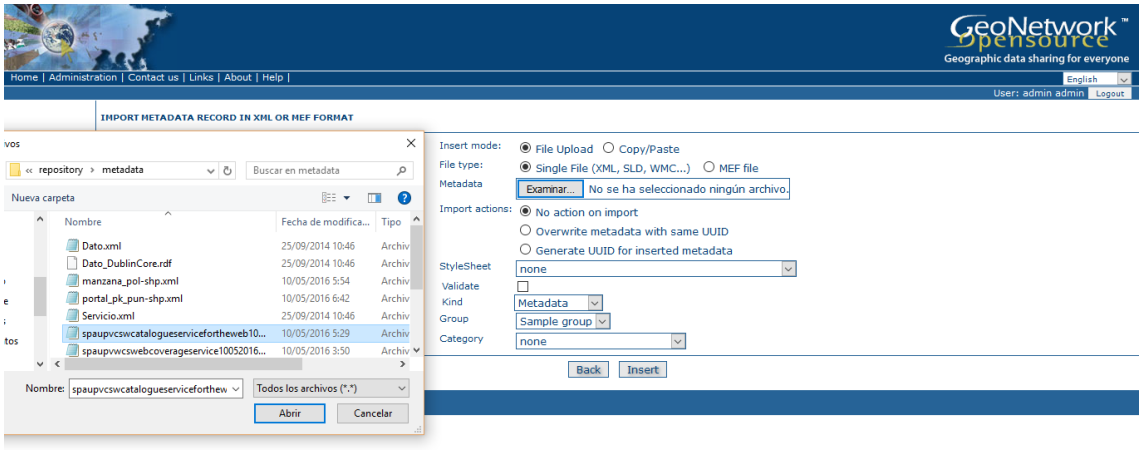


Imagen 71 - Formulario para añadir un nuevo metadato.

Importados todos los metadatos creados ya se tendrá completado el servicio de búsqueda del *Geoportal* y los usuarios podrán acceder a esta instancia de *Geonetwork* para buscar información sobre los recursos y servicios ofrecidos.

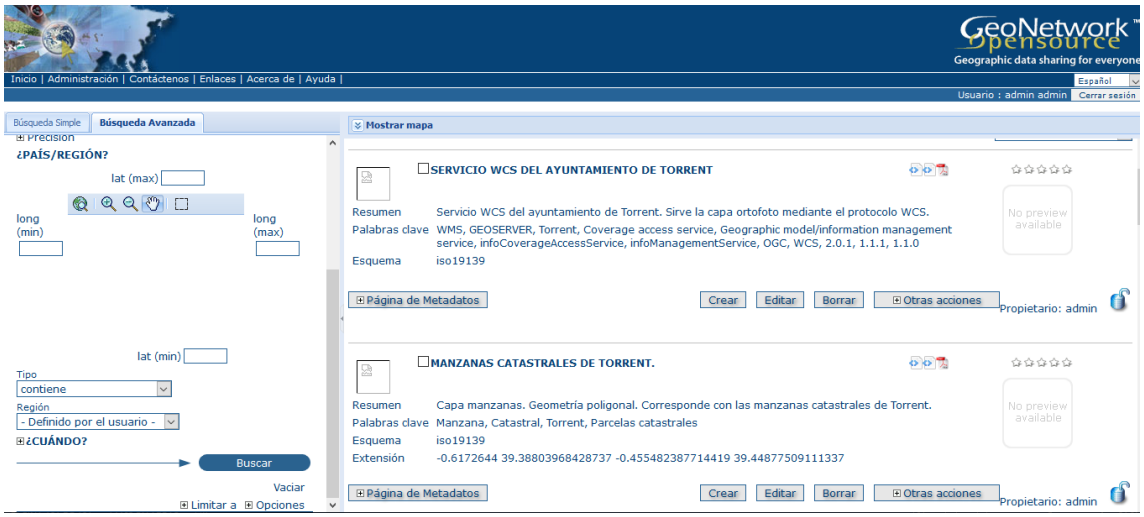


Imagen 72 - Servicio de búsqueda funcionando con los metadatos sirviéndose a través de este.

6.5. ELABORACIÓN DEL GEOPORTAL

La elaboración del Geoportal tiene una parte de desarrollo en *Node*, ya que los documentos HTML del Geoportal se sirven desde la aplicación generada con *Express*. Se usa también el *framework i18n* para servir el Geoportal distintos idiomas (Catellano por defecto, Valenciano e Inglés) y el módulo *pg-promise* para la conexión con la base de datos *PostgreSQL*.

6.5.1. RUTAS DEFINIDAS EN EXPRESS PARA EL GEOPORTAL

Ruta	*Método*	*Descripción*
/	GET	Renderiza la página de bienvenida del geoportal.
/visor	GET	Renderiza una página que muestra un visor con ciertas funcionalidades.
/descargas	GET	Renderiza una página que muestra un visor con la funcionalidad de poder descargar cartografía por áreas seleccionables en el mapa.
/proyecto	GET	Renderiza una página donde se muestra el árbol del código del proyecto, el cual tiene la capacidad de abrir cualquier archivo que contiene nuestro proyecto colgado en el repositorio GitHub.
/xhr	GET	Recibe un URL, la cual se espera que sea hacia un servicio WMS y devuelve el documento de capacidades de dicho servicio.
/info	GET	Recibe el nombre de una de las tablas de Postgres: (manzanas, viales, muni_torrent,etc.) y devuelve el nombre y el tipo de datos de cada una de sus columnas.

Tabla 4 - Rutas definidas para el Geoportal en Express.

Teniendo claro las rutas definidas y para qué sirven, el siguiente y último paso es implementar los documentos HTML que se renderizan a través de las rutas mencionadas anteriormente. Algunos de los controles que aparecen en los mapas del Geoportal sirven también para algunos mapas de la aplicación, por lo que los archivos *JavaScripts* utilizados se explicarán más adelante, ya que se considera que hay elementos en común en la puesta en marcha de los mapas con *OpenLayers* y el código necesario es el mismo.

A continuación se procede a comentar el diseño de los distintos documentos HTML que culminará el trabajo del Geoportal. Para ello utilizaremos el lenguaje *HTML 5, CSS3 y JavaScript*.

6.5.2. DISEÑO DEL GEOPORTAL

- **Página principal**

La página principal contiene un navegador superior fijo para navegar entre las distintas secciones, un banner informativo para captar la atención, una sección de servicios con un listado de preguntas frecuentes y un listado de los servicios con distintas funciones, una sección de descargas y una sección que da acceso a la aplicación. El enlace hacia los metadatos en *Geonetwork* es accesible desde el navegador superior.

Imagen 73 - Página principal. Portada y sección de Servicios.

La sección de descargas contiene un link que lleva al visor de descargas.

Imagen 74 - Página principal. Sección de descargas y aplicación.

- **Visor**

El visor tiene diversas funcionalidades como realizar rutas entre dos puntos utilizando el servicio de *OpenStreet Maps*, añadir un servidor *WMS* externo al mapa, realizar peticiones *GetFeatureInfo* a las capas visibles, seleccionar elementos de las capas según sus atributos mediante consultas *CQL* a *Geoserver* y menús de navegación.

Imagen 75 - Visor. Añadir Servidor WMS externo.

Además contiene los controles comunes a todos los mapas como controles de *Zoom, Mapa overview, Geocoder* y un control de capas.

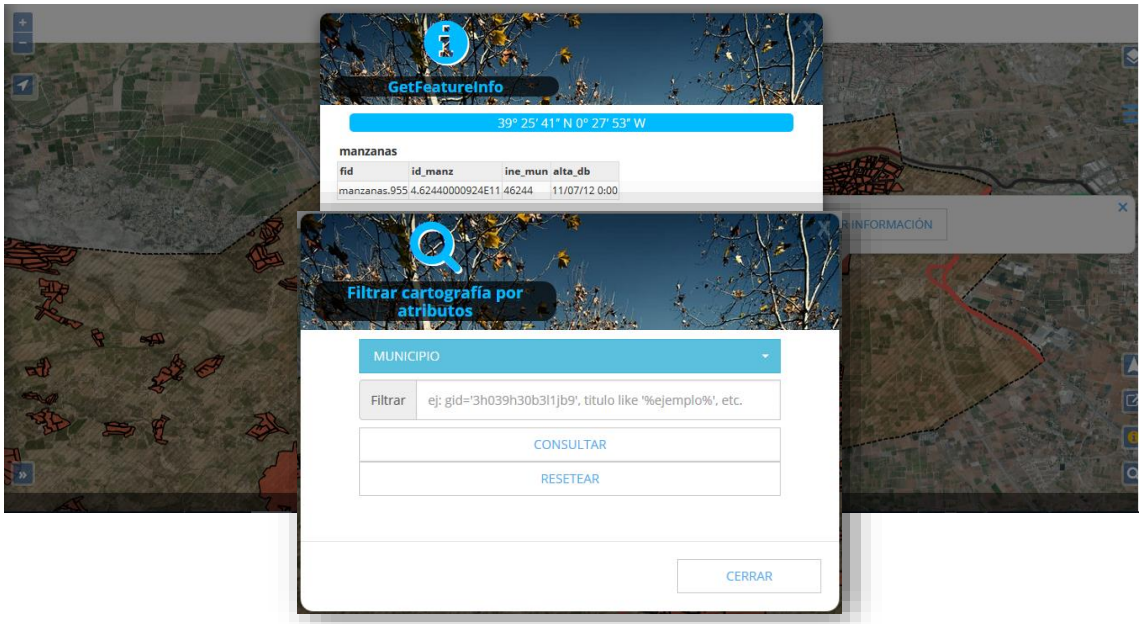


Imagen 76 - Visor. GetFeatureInfo y selección de elementos por atributos.

- **Visor de descargas**

El visor de descargas, como se ha comentado anteriormente, tiene únicamente la funcionalidad de descargas cartografía según el área (rectángulo definido por las coordenadas X e Y mínimas y máximas) dibujada en el mapa. Estas capas son las pertenecientes a la cartografía recortada de *CartoCiudad*, almacenada en *PostgreSQL* y utilizada en *Geoserver* como capas de nuestros servicios. Para poder realizar esta tarea, usaremos el servicio WFS para generar los archivos de descargas dinámicamente. Por lo tanto para realizar una descarga desde el mapa hay que elegir una capa en el menú desplegable, dibujar un "*BBOX*" y elegir el formato de salida de entre los posibles (*SHP, GeoJSON o GML*).

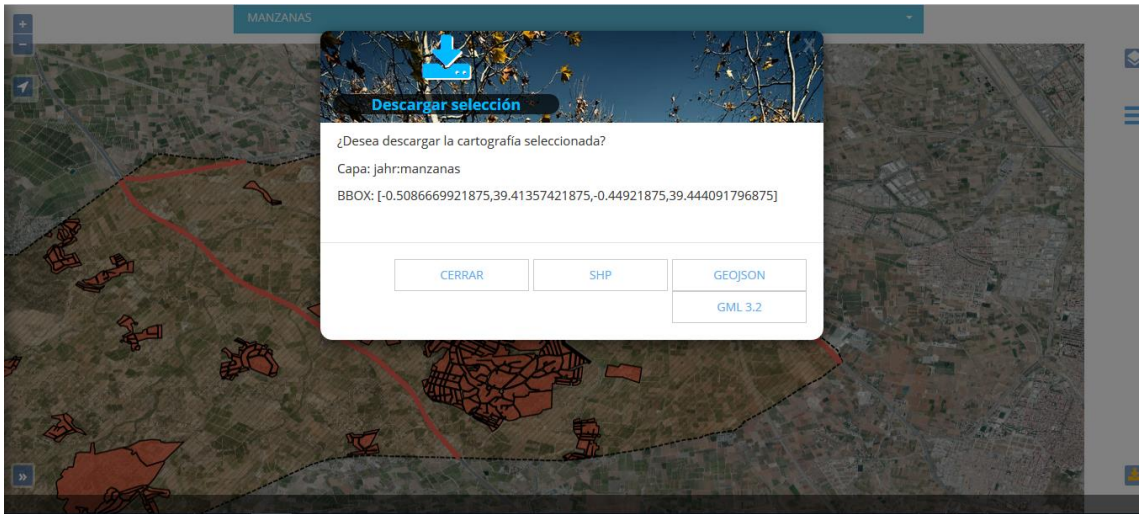


Imagen 77 - Visor de descargas.

7. DESARROLLO DE LA APLICACIÓN

La tarea más ardua y que ha supuesto la mayor inversión de tiempo para llevar a cabo el trabajo propuesto en este documento, ha sido sin duda la elaboración de la aplicación, debido entre otros factores al proceso de constante aprendizaje de los lenguajes necesarios así como la revisión y depuración del código generado.

En este apartado se detallan los aspectos más importantes dentro del desarrollo de la aplicación y se especifica qué contiene y cuál es la función de los archivos más relevantes del código.

7.1. CREACIÓN Y MODELIZACIÓN DE LA BASE DE DATOS EN POSTGRESQL

Este es el primer paso y el más importante dentro de la creación de una aplicación que consuma recursos de una base de datos, ya que se definen los objetos a través de sus propiedades y las relaciones entre ellos, lo que se conoce en bases de datos como *data modelling*.

- **Creación de la base de datos**

En primer lugar se debe crear una base de datos en *PostgreSQL* para almacenar nuestras tablas, una tarea sencilla desde *pgAdmin III*, un cliente de *PostgreSQL* (Todo este proceso puede llevarse a cabo mediante comandos en el *bash* utilizando el comando *psql*).

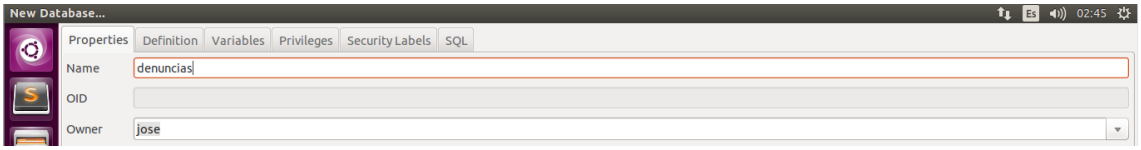


Imagen 78 - Creación de la base de datos "denuncias" que almacenará las tablas para la aplicación.

Realmente no hubiera hecho falta crear una nueva base de datos, ya que se podrían almacenar las tablas en la base de datos creada anteriormente para almacenar la cartografía de Torrent "*carto_torrent*", pero se ha querido mantener una separación entre ambos trabajos.

Cabe recalcar que se deberían crear índices para cada columna usada para buscar una tupla concreta en nuestra tabla, como por ejemplo un identificador.

Los índices permiten que las consultas se realicen más rápidamente al no buscar en toda la tabla entera.

En este proyecto solo se crean índices a las columnas espaciales (de tipo "*geometry*"). Estos índices (espaciales) agilizan las operaciones espaciales que hagamos.

El sistema de referencia empleado para almacenar los tipos de datos "geometry" en *PostgreSQL* es el *ETRS89 (EPSG 4258)*.

También se usan *triggers* para realizar comprobaciones antes de una consulta, por ejemmplo:

CREATE TRIGGER puntos_check_otra_tabla

BEFORE INSERT

ON denuncias_puntos

FOR EACH ROW

EXECUTE PROCEDURE trg_check_denuncias_puntos_otra_tabla();

El *trigger* anterior se disparará antes de cada inserción en la tabla *denuncias_puntos* y ejecutará la función *trg_check_denuncias_puntos_otra_tabla.*

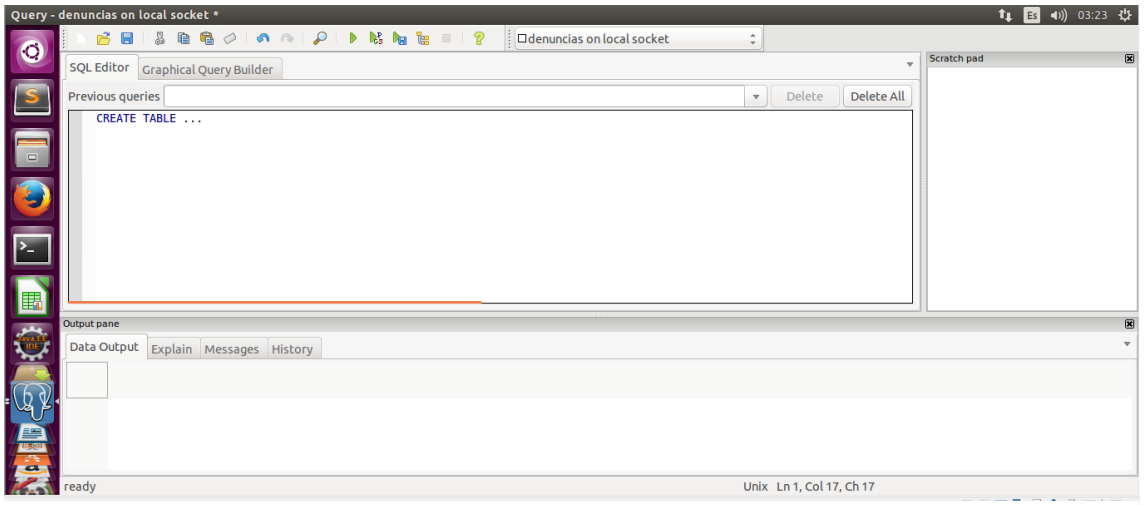


Imagen 79 - Interfaz gráfica en pgAdmin III ejecutar consultas sobre una base de datos.

La estructura de tablas que se va a crear dentro de la base de datos se puede resumir mediante este diagrama *UML*:

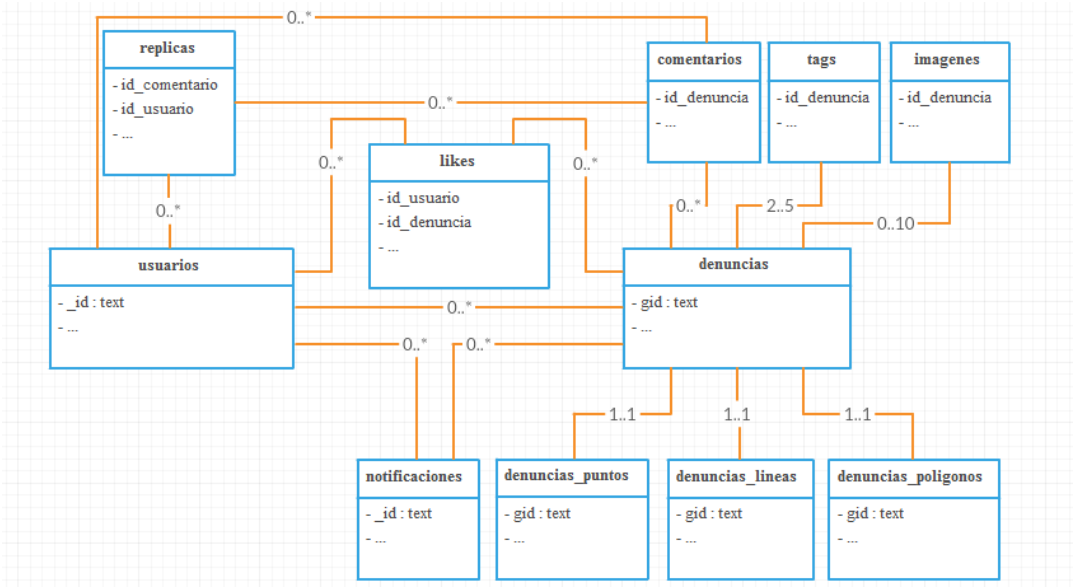


Imagen 80 - Diagrama UML de la estructura de la base de datos.

- **Creación de la tabla "usuarios"**

La tabla "*usuarios*" se encargará de almacenar la información de los usuarios que se registren. La estructura de la tabla "usuarios" se detalla a continuación:

Atributo	Tipo	Descripción
_id	Text	Identificador de un usuario. Será la **clave primaria** de esta tabla.
Profile	Json	Perfil del usuario. (Nombre de usuario, avatar,…)
Local	Json	Datos de la cuenta local. (Correo electrónico, válido)
Facebook	Json	Datos de la cuenta de Facebook sincronizada.
Twitter	Json	Datos de la cuenta de Twitter sincronizada.
created_at	timestamp	Fecha de creación del usuario.
location_pref	geometry	Localización establecida por el usuario para recibir notificaciones de denuncias cercanas.
distancia_aviso	Int	Constituye el radio del buffer para la notificación de denuncias cercanas.
resetpasswordtoken	Text	Identificador de solicitud para cambiar contraseña olvidada.
resetpasswordexpires	timestamp	Fecha en la que expira la solicitud para cambiar contraseña olvidada.
password	Text	Contraseña del usuario. (Se guardará encriptada)

Tabla 5 - Estructura de la tabla "usuarios".

La clave primaria, atributo mediante el cual una fila de la tabla queda unívocamente identificada, está formada por el campo "*_id*". Otro constreñimiento aplicado a esta tabla es forzar el sistema de referencia del atributo "*location_pref*" a ser *ETRS89 (EPSG 4258).* Se crea además un

- **Creación de la tabla "denuncias"**

Se encargará de almacenar toda la información de una denuncia exceptuando etiquetas, imágenes, comentarios y geometría que se almacenarán en tablas separadas. La estructura de la tabla se muestra a continuación:

Atributo	Tipo	Descripción
Gid	Text	Identificador de una denuncia. **Clave primaria** de la tabla.
id_usuario	Text	Identificador del usuario que ha realizado la denuncia. **Clave ajena** que apunta al atributo _id de la tabla usuarios.
Titulo	Text	Título de la denuncia.
descripcion	Text	Descripción de la denuncia.
Fecha	timestamp	Fecha de creación.
veces_vista	Int	Número de veces que ha sido vista.

Tabla 6 - Estructura de la tabla "denuncias".

Tiene como clave primaria "*gid*", el identificador de la denuncia.

Hace referencia a la tabla usuarios mediante el campo "*id_usuario*" que apunta al campo "*_id*" de la tabla usuarios.

Se aplica sobre esta tabla un *trigger* antes de cada inserción que comprueba que el usuario que añade una denuncia al menos dos minutos antes. Esto se hace para evitar que alguien o algún programa informático colapse la aplicación propagando denuncias iterativamente.

- **Creación de la tabla "denuncias_puntos"**

Se encargará de almacenar la geometría de las denuncias de tipo puntual.

Atributo	Tipo	Descripción
Gid	Text	Identificador de una denuncia. **Clave primaria** de la tabla y **clave ajena** que apunta al atributo gid de la tabla denuncias.
the_geom	Geometry	Geometría de la denuncia. (Puntual)

Tabla 7 - Estructura de la tabla "denuncias_puntos".

Se fuerza la geometría a ser de tipo "*POINT*" y el sistema de referencia "*EPSG:4258*". Se aplica sobre esta tabla un *trigger* antes de cada inserción para comprobar que el *gid* que se está introduciendo no está presente en cualquiera de las otras dos tablas de geometría de denuncias.

- **Creación de la tabla "denuncias_lineas"**

Se encargará de almacenar la geometría de las denuncias de tipo lineal.

Atributo	Tipo	Descripción
Gid	Text	Identificador de una denuncia. **Clave primaria** de la tabla y **clave ajena** que apunta al atributo gid de la tabla denuncias.
the_geom	Geometry	Geometría de la denuncia. (Lineal)

Tabla 8 - Estructura de la tabla "denuncias_lineas".

Se fuerza la geometría a ser de tipo "*LINESTRING*" y el sistema de referencia "*EPSG:4258*".

Se aplica sobre esta tabla un *trigger* antes de cada inserción para comprobar que el gid que se está introduciendo no está presente en cualquiera de las otras dos tablas de geometría de denuncias.

- **Creación de la tabla "denuncias_poligonos"**

Se encargará de almacenar la geometría de las denuncias de tipo poligonal.

Atributo	Tipo	Descripción
Gid	Text	Identificador de una denuncia. **Clave primaria** de la tabla y **clave ajena** que apunta al atributo gid de la tabla denuncias.
the_geom	Geometry	Geometría de la denuncia. (Poligonal)

Tabla 9 - Estructura de la tabla "denuncias_poligonos".

Se fuerza la geometría a ser de tipo "*POLYGON*" y el sistema de referencia "*EPSG:4258*".

Se aplica sobre esta tabla un *trigger* antes de cada inserción para comprobar que el *gid* que se está introduciendo no está presente en cualquiera de las otras dos tablas de geometría de denuncias.

- **Creación de la tabla "comentarios"**

Se encargará de almacenar los comentarios que reciba una denuncia. Como un comentario pertenece tanto a la denuncia como al usuario que comenta ambas tablas están implicadas.

Atributo	Tipo	Descripción
Id	Text	Identificador de un comentario. Será la **clave primaria** de esta tabla.
id_denuncia	Text	Identificador de la denuncia, a la cual pertenece el comentario. **Clave ajena** que apunta al atributo gid de la tabla denuncias.
id_usuario	Text	Identificador del usuario que ha comentado. **Clave ajena** que apunta al atributo _id de la tabla usuarios.
Fecha	timestamp	Fecha en la que se ha realizado el comentario.
contenido	Text	Contenido del comentario.

Tabla 10 - Estructura de la tabla "comentarios".

Se aplica sobre esta tabla un *trigger* antes de cada inserción para evitar que un usuario comente en la misma denuncia en un período de 30 segundos.

- **Creación de la tabla "replicas"**

Se encargará de almacenar las contestaciones que reciba un comentario que un usuario haya publicado en una denuncia. Se hace obvia por tanto la implicación de la tabla comentarios y la tabla usuarios.

Atributo	Tipo	Descripción
Id	Text	Identificador de una réplica. Será la **clave primaria** de esta tabla.
id_comentario	Text	Identificador del comentario, al cual pertenece la réplica. **Clave ajena** que apunta al atributo id de la tabla comentarios.
id_usuario	Text	Identificador del usuario que ha replicado. **Clave ajena** que apunta al atributo _id de la tabla usuarios.
Fecha	timestamp	Fecha en la que se ha realizado la réplica.
Contenido	Text	Contenido de la réplica.

Tabla 11 - Estructura de la tabla "replicas".

- **Creación de la tabla "tags"**

Se encargará de almacenar las etiquetas que un usuario le ponga a la denuncia.

Atributo	Tipo	Descripción
id_denuncia	Text	Identificador de la denuncia que contiene este tag. **Clave ajena** que apunta al atributo gid de la tabla denuncias.
Tag	Text	Contenido del tag.

Tabla 12 - Estructura de la tabla "tags".

Se aplica sobre esta tabla un *trigger* antes de cada inserción para comprobar que la denuncia tenga menos de 5 etiquetas. En caso de tener ya 5 rechazamos la consulta lanzando un error.

- **Creación de la tabla "imagenes"**

Se encargará de almacenar ruta donde se almacenan las imágenes de las denuncias en el servidor.

Atributo	Tipo	Descripción
id_denuncia	Text	Identificador de la denuncia que contiene esta imagen. **Clave ajena** que apunta al atributo gid de la tabla denuncias.
Path	Text	Ruta en la que la imagen está alojada (En nuestra máquina).
Fecha	timestamp	Fecha en la que se ha añadido la imagen.

Tabla 13 - Estructura de la tabla "imagenes".

Se aplica sobre esta tabla un *trigger* antes de cada inserción para comprobar que la denuncia tenga menos de 10 imágenes. En caso de tener ya 10 imágenes rechazamos la consulta lanzando un error.

- **Creación de la tabla "likes"**

Se encargará de almacenar los apoyos que una denuncia reciba de los usuarios. De esta manera si una fila contiene la id de la *denuncia X* y la id del *usuario A* quiere decir que el *usuario A* apoya la *denuncia X*.

Atributo	Tipo	Descripción
id_denuncia	Text	Identificador de la denuncia. **Clave ajena** que apunta al atributo gid de la tabla denuncias.
id_usuario	Text	Identificador del usuario al que le gusta la denuncia. **Clave ajena** que apunta al atributo _id de la tabla usuarios.

Tabla 14 - Estructura de la tabla "likes".

- **Creación de la tabla "notificaciones"**

Se encargará de almacenar las notificaciones que reciba un usuario. Puede haber 4 tipos de notificaciones:

1. Notificación por denuncia cerca → Un usuario ha publicado una denuncia dentro del área de influencia según tus datos del perfil en la base de datos.
2. Notificación por denuncia comentada → Un usuario ha comentado una denuncia publicada por ti.
3. Notificación por réplica en un comentario → Un usuario ha contestado a un comentario iniciando una conversación. Es avisado tanto los usuarios que hayan replicado el comentario, como el autor del comentario y el autor de la denuncia.
4. Notificación por apoyo → Un usuario indica que apoya o deja de apoyar tu denuncia.

Atributo	Tipo	Descripción
id_noti	Text	Identificador de una notificación. **Clave primaria** de la tabla.
id_denuncia	Text	Identificador de la denuncia, a la que hace referencia la notificación. **Clave ajena** que apunta al atributo gid de la tabla denuncias.
id_usuario_from	Text	Identificador del usuario que emite la notificación.
id_usuario_to	Text	Identificador del usuario que recibe la notificación.
Tipo	Text	Tipo de notificación.
Fecha	Timestamp	Fecha de emisión de la notificación.
Vista	Boolean	Valor que indica si la notificación ha sido vista por el usuario que la recibe.
Datos	Json	Datos que puede almacenar la notificación. Según el tipo serán datos distintos.

Tabla 15 - Estructura de la tabla "notificaciones".

7.2. CREACIÓN Y ESTRUCTURA DEL PROYECTO NODEJS

7.2.1. ARCHIVO "server.js"

Archivo de arranque de nuestro servidor. En este archivo se configurará nuestro servidor *HTTP* y *Express* entre otros.

7.2.2. DIRECTORIO "app"

- **Directorio "models"**

Este directorio almacenará los archivos encargados de realizar las peticiones a la base de datos. En concreto, habrá dos archivos "*usuarios.js*" y "*denuncias.js*" para mantener una separación lógica entre dos objetos bien diferenciados pero que guardan cierta relación.

- **Directorio "middlewares"**

Contendrá un archivo por cada función *middleware*. Una función *middleware* se define como una porción de código que se ejecuta antes o después del manejador de rutas. Estas funciones son útiles para prevenir que un usuario no autenticado realice una petición o para manejar errores entre otras muchas aplicaciones.

- **Directorio "controllers"**

Contendrá tres archivos encargados de gestionar las traducciones que se envían a un usuario (según el idioma que esté usando), las consultas que haremos a la base de datos y los eventos en tiempo real por medio de *web sockets*.

- **Directorio "jobs"**

Contendrá un único archivo y será el encargado de la búsqueda y eliminación de archivos obsoletos dentro del directorio temporal para la subida de imágenes.

- **Directorio "queries"**

Contendrá cuatro directorios para cada tipo de consultas (*SELECT, INSERT, UPDATE, DELETE*) y dentro de estos un archivo para cada consulta. De esta forma conseguimos tener cierto orden y evitamos tener que escribir la consulta entera cada vez que la queramos usar. También contendrá un archivo "*helper.js*" que se encargará de leer los archivos y guardar las consultas en variables.

- **Directorio "routes"**

Contendrá cuatro archivos encargados de la lógica del enrutamiento de nuestra aplicación *("home.js", "usuarios.js", "denuncias.js")* y Geoportal (*"geoportal.js"*).

7.2.3. DIRECTORIO "config"

Contendrá varios archivos encargados de la configuración de ciertos parámetros y funciones como:

- **auth.js** → Archivo *JSON* de configuración que contiene las "*API Keys*" o claves que nos proporcionan *Twitter* y *Facebook* y que son requeridas por *PassportJS* para la autenticación con estas redes sociales en nuestra aplicación. Contiene también las direcciones de "*login*" y "*callback*" de las *APIs* de *Twitter* y *Facebook*.
- **config_passport_pg** → Se encarga de definir las estrategias de autenticación y creación de usuarios con *PassportJS*, haciendo consultas sobre nuestra base de datos. Emplea también el módulo *nodemailer* para enviar un mensaje de autenticación al correo cuando un usuario se registra.
- **database.js** → Se encargará de definir las conexiones a nuestras bases de datos "*carto_torrent*" y "*denuncias*".
- **i18n** → Encargado de "montar" y configurar el *framework i18n* que maneja las traducciones.
- **multer.js** → Configura y exporta las funciones necesarias para la gestión de subida de imágenes.
- **upload.js** → Archivo *JSON* que almacena las rutas hacia los directorios de subidas de imágenes.

7.2.4. DIRECTORIO "geoportal"

Almacena los documentos HTML estáticos del Geoportal en cada idioma disponible (Castellano, Valenciano e Inglés).

7.2.5. DIRECTORIO "locales"

Contiene los archivos *Json* de las traducciones requeridas por el *framework i18n.*

7.2.6. DIRECTORIO "node_modules"

Consiste en un directorio auto generado al instalar un módulo de *NPM* y contendrá las dependencias de los módulos *NPM* instalados.

7.2.7. DIRECTORIO "public"

Contendrá los archivos que se sirven de forma estática desde nuestro servidor como *plugins* (OpenLayers 3, Bootstrap, etc), archivos *CSS* y *JavaScript*, etc.

7.2.8. DIRECTORIO “views”

Contendrá las vistas (*Templates*) renderizadas por nuestro motor de plantillas *JADE*.

7.3. IMPLEMENTACIÓN DE LAS CONSULTAS

Gracias a la potente librería que se está utilizando para la conexión con *PostgreSQL* podemos escribir la consulta en archivos de texto y posteriormente leer esos archivos, formar las consultas con los parámetros y ejecutarlas. Esta separación de las consultas en archivos permite una mejor estructuración de las mismas y una menor carga de líneas de código en nuestros archivos y produce un código limpio y fácilmente legible e interpretable.

A continuación se describen cada una de las consultas implementadas según el tipo de consulta (*SELECT, INSERT, UPDATE o DELETE*).

7.3.1. CONSULTAS DEL TIPO "SELECT"

7.3.1.1. DATOS DE LA APLICACIÓN

Consulta que obtiene varios datos referentes al estado actual de la aplicación. Estos datos son el número de denuncias, número de usuarios y el número de denuncias realizadas en el día actual.

```
select t1.cnt as num_denun_total,
        t2.cnt as num_denun_hoy,
        t3.cnt as num_usuarios_total
from (select count (*) as cnt from denuncias) as t1
cross join (select count(*) as cnt from denuncias where fecha >= to_char(current_timestamp, 'YYYY-MM-DD')::date) as t2
cross join (select count(*) as cnt from usuarios) as t3
```

Imagen 81 - Consulta para obtener datos básicos de la aplicación.

7.3.1.2. USUARIOS

Para los usuarios, interesa tener varias consultas para buscar un usuario por distintos atributos.

- **por_id.sql, por_email.sql, por_username.sql, por_email_o_username.sql, por_id_facebook.sql, por_id_twitter.sql, por_reset_token.sql.**

Interesará buscar un usuario de acuerdo a su e-mail, nombre de usuario, identificador, etc.

- **accion.sql, acciones.sql, cerca_denuncia.sql, denuncias.sql, denuncias_favoritas.sql, localizacion_favorita.sql, notificacion.sql, notificaciones.sql, perfil_otro.sql.**

También se implementan consultas para obtener ciertos datos acerca de un usuario en concreto. Estos datos son obtener una notificación o una acción concreta, obtener todas sus notificaciones y acciones, datos del perfil a mostrar, su localización y distancia de aviso, las denuncias realizadas, sus denuncias favoritas y saber si una geometría entra en su área de aviso.

7.3.1.3. DENUNCIAS

De la misma forma que los usuarios, debemos realizar consultas para que sean filtradas por algunos de sus atributos.

- **por_id.sql, por_path_imagen.sql**

Buscaremos denuncias atendiendo a su identificador y por la ruta de una de sus imágenes (Útil para comprobar que el usuario que intenta borrar una imagen es el dueño de la denuncia).

- **comprobar_geometria_puntual.sql, comprobar_geometria_lineal.sql, comprobar_geometria_poligonal.sql**

Tres consultas útiles para comprobar la geometría de una denuncia que se va a añadir a nuestra base de datos. Independientemente del tipo de geometría se obtendrá de la consulta si la geometría está dentro de los límites del municipio de Torrent. En caso de geometría lineal o poligonal además se obtendrá además su longitud o su área, respectivamente.

- **denuncias_cerca.sql**

Consulta para obtener las denuncias "cercanas" a un punto dado, considerando denuncia cercana cualquier denuncia en un buffer de 100 metros de radio del punto que recibe como parámetro.

- **is_equal.sql**

Devuelve un valor booleano indicando si la geometría de la denuncia es exactamente igual a una geometría dada.

- **me_gusta.sql**

Devuelve un valor booleano indicando si un usuario apoya una denuncia en concreto.

- **por_pagina.sql**

Devuelve las denuncias de una página en concreto. (Suponiendo las denuncias ordenadas por fecha de mayor a menor y un número máximo de 10 denuncias por página).

- **visor.sql**

Devuelve las denuncias que se visualizarán en el visor. (Las creadas en las últimas 24 horas).

- **sin_where.sql, sin_where_gml.sql**

Varias consultas que no tienen claúsula *WHERE* (Útiles para crear consultas dinámicas más adelante). Una de ellas devolverás las geometrías en formato *GeoJSON* y la otra en *GML*.

7.3.2. CONSULTAS DEL TIPO "INSERT"

7.3.2.1. USUARIOS

- **crear.sql**

Sirve para almacenar un usuario en nuestra base de datos.

- **denuncia_cerca.sql, denuncia_comentada.sql, otras.sql**

Diversas consultas para insertar las notificaciones de un usuario.

7.3.2.2. DENUNCIAS

- **denuncia.sql**

Se crearán distintas consultas para insertar en las tablas correspondientes los datos de las mismas. La consulta "principal" insertará únicamente el título, la descripción y el identificador del usuario en la tabla *denuncias*. Los campos "*identificador de la denuncia"* y *fecha* se generarán automáticamente en *PostgreSQL*.

- **punto.sql, línea.sql, polígono.sql**

Se implementan también tres consultas para insertar cada tipo de geometría en su tabla correspondiente.

- **comentario.sql, tag.sql, imagen.sql, replica.sql, like.sql**

Varias consultas para insertar un comentario, un *tag*, una imagen, una réplica a un comentario y un *like* o apoyo a la denuncia.

7.3.3. CONSULTAS DEL TIPO "UPDATE"

7.3.3.1. USUARIOS

- **deslincar_facebook.sql, delincar_twitter.sql**

Consulta para poner a *NULL* la información sobre las cuentas conectadas de *Facebook* o *Twitter*.

- **contraseña.sql**

Consulta para actualizar la contraseña de un usuario.

- **perfil.sql**

Consulta para actualizar el perfil de un usuario.

- **distancia_aviso.sql**

Consulta para actualizar la distancia de aviso de un usuario.

- **local.sql**

Consulta para actualizar el atributo "local" de un usuario.

- **localizacion_preferida.sql**

Consulta para actualizar la localización preferida de un usuario.

- **notificacion_vista.sql**

Consulta para actualizar el estado de una notificación a vista cuando el usuario ha interactuado con la notificación.

- **perfil.sql**

Consulta para actualizar el perfil de un usuario.

- **reset_token.sql**

Consulta para actualizar contraseña y poner a *NULL* los campos "*resetPasswordToken*" y "*resetPasswordExpires*".

- **token_hora.sql**

Consulta para actualizar los campos "*resetPasswordToken*" y "*resetPasswordExpires*".

- **facebook.sql, twitter.sql**

Consulta para actualizar la información sobre las cuentas conectadas de Facebook o Twitter.

7.3.3.2. DENUNCIAS

- **denuncia.sql, set_titulo.sql, set_contenido.sql**

Actualiza título y descripción de la tabla "*denuncias*" para un determinado identificador de denuncia.

- **punto.sql, línea.sql, polígono.sql**

Actualizan la geometría de su tabla correspondiente.

- **veces_vista.sql**

Actualiza el número de veces que la denuncia ha sido vista por un usuario.

7.3.4. CONSULTAS DEL TIPO "DELETE"

7.3.4.1. DENUNCIAS

- **denuncia.sql**

Consulta para eliminar una denuncia por su identificador. Al haber definido las claves ajenas y un borrado en cascada en la definición de nuestras tablas, no nos tenemos que preocupar por eliminar registros de otras tablas.

- **imagen.sql**

Borra una imagen de la denuncia cuyo identificador se pasa como parámetro, de acuerdo a la ruta de la imagen también pasada como parámetro.

- **like.sql**

Elimina un registro de la tabla "*likes*" en el que coincidan los identificadores de la denuncia y el usuario pasados como parámetro.

- **tag.sql**

Elimina un *tag* concreto de la denuncia.

- **punto.sql, línea.sql, polígono.sql**

Elimina una geometría de la tabla correspondiente.

7.4. IMPLEMENTACIÓN DE LOS MODELOS

Los modelos definen las funciones de nuestros objetos y son los encargados de realizar las operaciones contra la base de datos. Por lo tanto el flujo de trabajo de las funciones que definamos en los modelos pasará por ejecutar las consultas implementadas anteriormente, realizar ciertos procesos con la información y devolver un valor de retorno.

Estas operaciones sobre la base de datos (seleccionar, editar, actualizar y eliminar) tienen un carácter asíncrono, afortunadamente con JavaScript podemos hacer que nuestras funciones reciban como parámetro una función "*callback*" que se ejecute cuando nuestra operación asíncrona se haya llevado a cabo. Aunque por el contrario esto lleva a implementar unos patrones de código que son etiquetados como malas prácticas. Desde la aparición en 2015 de la especificación *ECMAScript 6*, existe una clase nativa en JavaScript llamada "*Promise*". La clase *Promise* a grandes rasgos nos permite ejecutar código asíncrono dentro de una función que recibe como parámetros dos "*callbacks*", "*resolve*" y "*reject*". Nos permite devolver el valor de retorno de esa operación asíncrona en un método "*then*" que se ejecuta cuando llamamos al método "*resolve*" dentro de la función. Llamar al método "*reject*" sería sinónimo de lanzar un error y pararía por completo la ejecución de la función. La clase *Promise* también dispone de varios métodos estáticos como *Promise.all()* que nos ayuda a ejecutar varias "*Promise*" en paralelo.

En este caso para evitar mayores confusiones y debido a que no vamos a tener en nuestro código muchas llamdas a *callbacks* dentro de otros *callbacks*, problema conocido como "*Nested Callback Hell*" y que se soluciona, entre otras formas, usando *Promises*, se ha optado por utilizar el patrón del callback, aunque hace de nuestro código algo improductivo ya que usamos *Promise* dentro de estos métodos y sería conveniente devolver estos objetos *Promise* en vez de llamar al *callback* que se pasa, en otras palabras, se debería eliminar el *callback* y devolver el objeto Promise para evaluar el éxito o error de estos métodos llamando a los métodos "*then*" y "*catch*" de *Promise* respectivamente.

La siguiente imagen trata de representar con código la problemática mencionada.

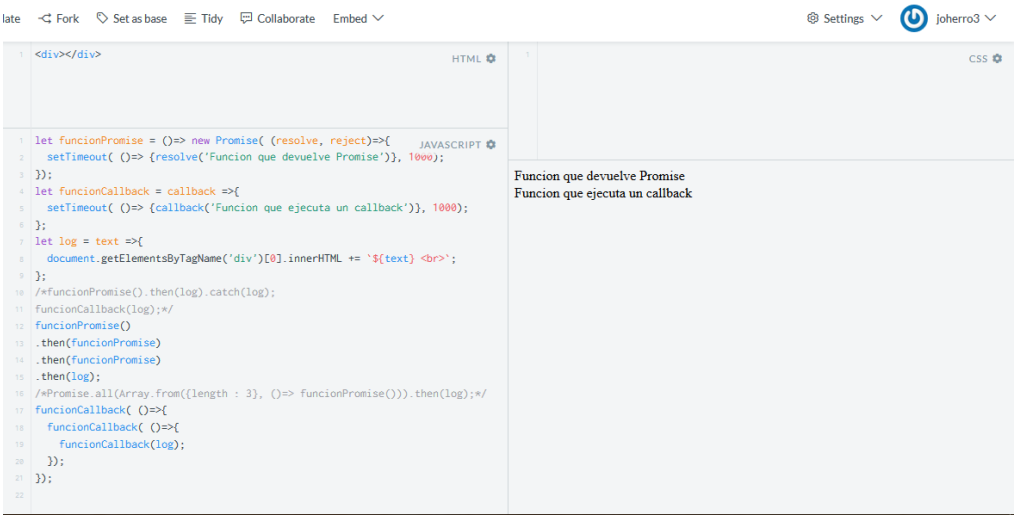


Imagen 82 - Promises vs Callbacks - Ejecutados secuencialmente.

También cabe destacar que los modelos se implementan a partir de clases que se pueden instanciar. En JavaScript no existe una noción de Clase realmente funcional como otros lenguajes orientados a objetos incluso con la aparición en *ECMASCRIPT 6* de las palabras clave "*class*" y "*super*". Estas *pseudo-clases* son funciones/objetos prototipados las cuales pueden heredar sus propiedades.

Veamos más en detalle qué es lo que nuestros modelos se encargan de manejar.

7.4.1. MODELO "USUARIO"

En primer lugar importamos los módulos que vamos a usar para implementar los métodos del modelo Usuario.

```
var consultas = require('../controllers/queries.js');
var crypto = require('crypto');
var nodemailer = require('nodemailer');
var validator = require('validator');
var config = require('../../config/mailer.js');
var formatsAllowed = 'png|jpg|jpeg|gif'; // Podríamos poner más
var bcrypt   = require('bcrypt-nodejs');
var this_;
var db = require('../../config/database.js').db;
var path = require('path');
/*
 * Constructor
 */
function Usuario(){
        this_ = this;
}
```

Imagen 83 - Módulos importados, variables en el nivel más alto y constructor de la clase Usuario.

Estos módulos nos facilitan la implementación de los métodos, los cuales realizaremos con la intención de que sean reusables en otras partes de la aplicación. Un ejemplo de los métodos implementados se puede ver en la siguiente imagen:

```
/*
=====================================================================
Obtener localización preferida del usuario
@id_usuario
=====================================================================
*/
Usuario.prototype.get_localizacion_preferida = function(id_usuario, callback){
        db.one(consultas.obtener_loc_preferida, id_usuario)
        .then(function(location){
                callback(null, location.loc_pref);;
        })
        .catch(function(error){
                callback({type : 'error', msg : error.toString()});
        });
};
```

Imagen 84 - Ejemplo de un método de la clase Usuario.

A continuación se detalla la lista de métodos de la clase Usuario.

- **find_by_id**
  - Recibe => id (*String*), callback (*Function*)
  - Conexión con la base de datos, ejecuta la consulta de tipo *SELECT* llamada *find_by_id*, ejecuta el *callback* pasando como parámetros el error (en caso de error) o el usuario que concuerda con la id pasada como argumento.
  - Sirve para obtener los datos de un usuario a partir de su *id*.
- **find_by_email**
  - Recibe => email (*String*), callback (*Function*)
  - Sirve para obtener los datos de un usuario a partir de su email.
- **find_by_pass_token**
  - Recibe => token (*String*), callback (*Function*)
  - Sirve para obtener los datos de un usuario a partir de su "*resetpasswordtoken*". Campo que se rellena con un código hexadecimal que se genera automáticamente cuando un usuario solicita un cambio de contraseña debido a la pérdida de las credenciales para poder *loggearse*.
- **find_by_email_o_username**
  - Recibe => email/username (*String*), callback (*Function*)
  - Sirve para obtener los datos de un usuario a partir de su email o nombre de usuario. Útil para el formulario de *login*, de esta forma el usuario puede utilizar cualquiera de los dos campos para identificarse.
- **find_by_username**
  - Recibe => username (*String*), callback (*Function*)
  - Sirve para obtener los datos de un usuario a partir de su nombre de usuario.
- **find_by_facebook_id / find_by_twitter_id**
  - Recibe => id_twitter / id_facebook (*String*), callback (*Function*)
  - Sirve para obtener los datos de un usuario a partir de su id de *Twitter* o *Facebook*, en el caso de que tenga conectada su cuenta.
- **set_facebook / set_twitter**
  - Recibe =>id_usuario, Facebook_profile / twitter_profile (*Object*), callback (*Function*)
  - Sirve para actualizar los datos de las cuentas de Twitter o Facebook sincronizadas.
- **crear**
  - Recibe => password (*String*), local (*Object*), profile (*Object*), callback (*Function*)
  - Inserta un usuario en la tabla "*usuarios*".

- **create_ranom_token**
  - Recibe => email (*String*), callback (*Function*)
  - Genera un texto aleatorio de 20 dígitos y lo usa para actualizar el campo "*resetpasswortoken*" del usuario que coincide con el email que se pasa como parámetros. También actualiza el campo "*resetpasswordexpires*" con el valor de la fecha actual más una hora.
- **cambiar_pass**
  - Recibe => password (*String*), id_usuario (*String*), callback (*Function*)
  - Genera una versión encriptada de la contraseña que se pasa como parámetro y actualiza la base de datos con esta nueva información.
- **cambiar_pass_token**
  - Recibe => password (*String*), id_usuario (*String*), callback (*Function*)
  - Genera una versión encriptada de la contraseña y actualiza la base de datos con esta nueva información, además de poner a *NULL* los campos "*resetpasswordtoken*" y "*resetpasswordexpires*".
- **unkink_twitter**
  - Recibe =>id_usuario (*String*), callback (*Function*)
  - Actualiza el campo twitter del usuario poniéndolo a *NULL*.
- **confirmar**
  - Recibe =>user (*Object*), callback (*Function*)
  - Actualiza el campo "local" del usuario (de tipo *JSON*). Dentro de este *JSON* actualiza el valor de la clave "*valid*" a *TRUE*.
- **perfil_visible**
  - Recibe =>id_usuario (*String*), callback (*Function*)
  - Obtiene ciertos datos del usuario que coincide con el identificador pasado como parámetro.
- **get_denuncias**
  - Recibe =>id_usuario (*String*), callback (*Function*)
  - Obtiene las denuncias del usuario cuyo identificador se pasa como parámetro.
- **get_denuncias_fav**
  - Recibe =>id_usuario (*String*), callback (*Function*)
  - Obtiene las denuncias apoyadas por el usuario cuyo identificador se pasa como parámetro.
- **get_notificaciones**
  - Recibe =>id_usuario (*String*), callback (*Function*)
  - Obtiene las notificaciones del usuario cuyo identificador se pasa como parámetro.
- **get_acciones**
  - Recibe =>id_usuario (*String*), callback (*Function*)
  - Obtiene las acciones del usuario cuyo identificador se pasa como parámetro.
- **update**
  - Recibe => user (*Object*), callback (*Function*)
  - Actualiza el perfil (campo "*perfil*" en la base de datos) del usuario cuyo identificador se pasa como parámetro.

- **cambiar_imagen_perfil**
  - Recibe => user (*Object*), callback (*Function*)
  - Actualiza la imagen de perfil del usuario con la imagen que recibe del objeto que se pasa como parámetro.
- **get_localizacion_preferida**
  - Recibe =>id_usuario (*String*), callback (*Function*)
  - Obtiene las coordenadas del centro y el radio del buffer de aviso del usuario cuyo identificador se pasa como parámetro.
- **update_localizacion_preferida**
  - Recibe => opciones (*Object*), callback (*Function*)
  - Actualiza las coordenadas del centro y el radio del buffer de aviso de un usuario.
- **noti_vista**
  - Recibe => id_notificacion (*String*), callback (*Function*)
  - Actualiza el estado de la notificación cuyo identificador se pasa como parámetro a vista, modificando el atributo "*vista*" en la base de datos.
- **me_gusta_la_denuncia**
  - Recibe => id_usuario (*String*), id_denuncia (*String*), callback (*Function*)
  - Obtiene un valor booleano indicando si un usuario apoya o no la denuncia cuyo identificador se pasa como parámetro.
- **likear_denuncia**
  - Recibe => id_usuario (*String*), denuncia (*Object*), like (*Boolean*), callback (*Function*)
  - Añade una fila en la tabla "*likes*" con el identificador de la denuncia y del usuario si el valor de la variable *like* que se pasa como parámetro es false, en caso contrario ya habrá dicha fila en la base de datos y por tanto la eliminamos. También se encarga de almacenar la notificación que se genera en la base de datos.
- **enviar_email**
  - Recibe => opciones (*Object*), callback (*Function*)
  - Envía un email a un usuario cuyo email se pasa dentro de las opciones.
- **get_noti_by_id**
  - Recibe => id_noti (*String*), callback (*Function*)
  - Obtiene la información de la notificación cuyo identificador se pasa como parámetro.
- **get_accion_by_id**
  - Recibe => id_noti (*String*), callback (*Function*)
  - Obtiene la información de la acción cuyo identificador de la notificación correspondiente se pasa como parámetro.
- **generateHash**
  - Recibe => password (*String*)
  - Devuelve una versión encriptada de la contraseña que se pasa como parámetro.
- **validPassword**
  - Recibe => password_encriptada (*String*), password(*String*)
  - Devuelve un valor booleano indicando si el valor encriptado de una contraseña coincide con su versión sin encriptar.

- **gravatar**
  - Recibe => email (*String*)
  - Devuelve una *URL* a un avatar que se genera en función del email introducido.

7.4.2. MODELO "DENUNCIA"

En primer lugar importamos los módulos que vamos a usar para implementar los métodos del modelo Denuncia.

```
'use strict';
var fs = require('fs'), // file System
        path = require('path'), // util para paths
        exec = require('child_process').exec, // Ejecutar comandos
        denunciasPorPagina = 10,
        maxPaginas = 0,
        config = require('../../config/upload.js'),
        validator = require('validator'), // validator
        database = require('../../config/database.js'),
        pgp = database.pgp,
        db = database.db,
        dbCarto = database.dbCarto,
        consultas = require('../controllers/queries.js'),
        crypto = require('crypto'),
        mkdirp = require('mkdirp'),
        formatsAllowed = 'png|jpg|jpeg|gif', // Podríamos poner más
        this_;
        /*
         * Constructor
         */
        function Denuncia(){
                this_ = this;
        }
```

Imagen 85 - Módulos requeridos para implementar el modelo "Denuncia" y constructor de la clase Denuncia.

Como se puede observar en la imagen, la primera línea de código "*use strict*"; está diciendo a *NodeJS* que debe ejecutarse en modo estricto, dándonos acceso a algunos módulos que implementan muchas de las funcionalidades que nos brinda *ECMASCRIPT 6* y que están ocultos por defecto.

En la implementación de los métodos de la clase de este modelo se usan muchas de estas funcionalidades como son los *generators, promises*, la palabra reservada *yield*, etc. y es por eso que debemos ejecutar estas líneas de código en modo estricto.

A continuación se detalla la lista de métodos que contiene la clase Denuncia:

- **set_titulo**
  - Recibe => titulo (*String*), id_denuncia (*String*)
  - Actualiza el título de la denuncia cuyo identificador se pasa como parámetro. Devuelve un objeto *Promise*.
- **set_contenido**
  - Recibe => contenido (*String*), id_denuncia (*String*)
  - Actualiza el contenido de la denuncia cuyo identificador se pasa como parámetro. Devuelve un objeto *Promise*.

- **is_equal**
  - Recibe => wkt (*String*), geojson (*Object*)
  - Obtiene un valor booleano indicando si las geometrías generadas por el *WKT* y el *GeoJSON* pasados como parámetro coinciden. Devuelve un objeto *Promise*.
- **eliminar_tag**
  - Recibe => tag (*String*), id_denuncia (*String*)
  - Elimina el *tag* de la denuncia cuyo identificador se pasa como parámetro. Devuelve un objeto *Promise*.
- **find**
  - Recibe => filtro (*Object*), geojson_format (*Boolean*), callback (*Function*)
  - Obtiene la denuncia o las denuncias que coinciden con el filtro de búsqueda pasado. Recibe un valor booleano indicando si la geometría de salida se devuelve en formato *GeoJSON*.
- **find_by_id**
  - Recibe => id_denuncia (*String*), callback (*Function*)
  - Obtiene la información de la denuncia cuyo identificador se pasa como parámetro.
- **find_by_path_image**
  - Recibe => path (*String*), callback (*Function*)
  - Obtiene la información de la denuncia que contiene una imagen almacenada en disco con la ruta que se pasa como parámetro.
- **find_by_id**
  - Recibe => id_denuncia (*String*), callback (*Function*)
  - Obtiene la información de la denuncia cuyo identificador se pasa como parámetro.
- **comprobar_geometria**
  - Recibe => wkt (*String*), callback (*Function*)
  - Comprueba que una geometría que se pasa en formato *WKT* cumple con los requisitos para ser aceptada en la aplicación.
- **get_usuarios_cerca**
  - Recibe => wkt (*String*), id_usuario (*String*), callback (*Function*)
  - Obtiene los usuarios cuya área de influencia intersecta con el *WKT* que se pasa como parámetro, exceptuando el usuario cuyo identificador se pasa como parámetro.
- **crear_temp_dir**
  - Recibe => callback (*Function*)
  - Crea una carpeta temporal con un texto hexadecimal aleatorio de 25 dígitos como nombre dentro de la carpeta temporal "*temp*" donde se almacenan las subidas temporales de imágenes de las denuncias.
- **eliminar_imagen_temporal**
  - Recibe => tempdir (*String*), filename (*String*), callback (*Function*)
  - Recibe el nombre de la carpeta temporal y el nombre de la imagen y elimina dicha imagen alojada en ese directorio temporal.

- **subir_imagen_temporal**
  - Recibe => file (*Object*), callback (*Function*)
  - Comprueba que la imagen subida coincide con algunos de los formatos aceptados por nuestro servidor para el almacenamiento de imágenes.
- **sumar_visita**
  - Recibe => id_denuncia (*String*), callback (*Function*)
  - Aumenta en uno el número de visitas de la denuncia cuyo identificador se pasa como parámetro.
- **añadir_comentario**
  - Recibe => opciones (*Object*), callback (*Function*)
  - Añade un comentario en la base de datos con las opciones que se pasan como parámetro. Además se encarga de añadir y emitir las notificaciones generadas por esta acción.
- **añadir_replica**
  - Recibe => opciones (*Object*), callback (*Function*)
  - Añade una réplica a un comentario en la base de datos con las opciones que se pasan como parámetro. Además se encarga de añadir y emitir las notificaciones generadas por esta acción.
- **guardar**
  - Recibe => opciones (*Object*), callback (*Function*)
  - Añade una denuncia en la base de datos con las opciones que se pasan como parámetro. Además se encarga de añadir y emitir las notificaciones generadas por esta acción.
- **find_by_pagina**
  - Recibe => page (*Number*), callback (*Function*)
  - Obtiene las denuncias según la página que se pasa como parámetro. Para ello se supone un número máximo de 10 denuncias por página y las denuncias ordenadas por fecha descendente.
- **eliminar**
  - Recibe => id_denuncia (*String*), callback (*Function*)
  - Elimina todos los datos que pueda haber en la aplicación de la denuncia cuyo identificador se pasa como parámetro.
- **eliminar_imagen**
  - Recibe => path (*String*), callback (*Function*)
  - Elimina la imagen cuya ruta se pasa como parámetro.
- **editar**
  - Recibe => opciones (*Object*), callback (*Function*)
  - Actualiza una denuncia con las opciones que se pasan como parámetro.
- **denuncias_visor**
  - Recibe => callback (*Function*)
  - Obtiene las denuncias que se muestran en el visor de la aplicación por defecto. Estas serán aquellas que se crearon en el mismo día que se ejecuta la petición.

- **denuncias_cerca**
    - Recibe => posicion (*String*), callback (*Function*)
    - Obtiene las denuncias que se encuentran a menos de 100 metros de la posición que se pasa como parámetro.
- **getAllTags**
    - Recibe => callback (*Function*)
    - Obtiene todos los *tags* almacenados en la base de datos.
- **filtro_denuncias**
    - Recibe => filtro (*Object*)
    - Formatea la cláusula *WHERE* para generar consultas dinámicas atendiendo a los valores pasados en el filtro.

7.5. ENRUTAMIENTO Y FUNCIONES MIDDLEWARE

El enrutamiento consiste en definir de forma lógica y ordenada las peticiones que implementará nuestro servidor. Se encargarán de procesar una petición *HTTP*, hará uso de nuestros modelos para almacenar y obtener información de la base de datos y enviarán la respuesta de nuevo al cliente. Estas rutas se implementan en archivos separados y se llaman en el archivo de arranque de la aplicación *Node* (*server.js*).

- **Principal**

Ruta	*Método*	*Descripción*
/	GET	Renderiza la página de bienvenida de la aplicación.

Tabla 16 - Ruta principal. Relativa a "/app".

- **Usuarios**

Ruta	*Método*	*Descripción*
/perfil	GET	Renderiza el perfil del usuario que está conectado.
/perfil	PUT	Actualiza el perfil del usuario conectado.
/perfil/actualizar	GET	Renderiza la página en la que se puede actualizar datos del perfil mediante un formulario.
/perfil/avatar/facebook	PUT	Actualiza la foto de perfil con la foto de perfil de Facebook en caso de tener cuenta asociada.
/perfil/avatar/twitter	PUT	Actualiza la foto de perfil con la foto de perfil de Twitter en caso de tener cuenta asociada.
/perfil/avatar/gravatar	PUT	Actualiza la foto de perfil con un avatar generado aleatoriamente. Se emplea el servicio de gravatar.
/perfil/avatar	PUT	Actualiza la foto de perfil con la imagen que se le pase en el cuerpo de la petición (Subida de imagen).
/perfil/localizacion	GET	Renderiza la página en la que se puede actualizar los parámetros de aviso (centro y radio) mediante un mapa y un formulario.
/perfil/localizacion	PUT	Actualiza los parámetros de aviso (centro y radio) mediante un mapa y un formulario.
/perfil/password	GET	Renderiza la página en la que se puede actualizar la contraseña (si se está conectado) mediante un formulario.
/perfil/password	PUT	Actualiza la contraseña del usuario conectado.
/olvidaste	GET	Renderiza la página en la que se puede rellenar un formulario para recuperar las credenciales.
/olvidaste	POST	Recibe el email del usuario, genera un código aleatorio y lo inserta en la base de datos. Este código sirve generar una ruta y enviar un correo informativo al usuario con esa ruta

		donde podrá cambiar la contraseña; Este código solo será válido durante una hora.
/unlink/facebook	GET	Deja de asociar la cuenta de Facebook del usuario con su cuenta local, para ello pone a NULL el atributo facebook en la base de datos.
/unlink/twitter	GET	Deja de asociar la cuenta de Twitter del usuario con su cuenta local, para ello pone a NULL el atributo twitter en la base de datos.
/conectar/facebook	GET	Redirecciona a una página de inicio de sesión de Facebook. Usada cuando se quiere asociar una cuenta local con una de Facebook.
/conectar/twitter	GET	Redirecciona a una página de inicio de sesión de Twitter. Usada cuando se quiere asociar una cuenta local con una de Twitter.
/conectar/facebook/callback	GET	Ruta a la que se redirecciona desde "/conectar/facebook" una vez nos autenticamos con Facebook. Ejecuta el manejador implementado con "FacebookStrategy" de Passport en el archivo de configuración de Passport.
/conectar/twitter/callback	GET	Ruta a la que se redirecciona desde "/conectar/twitter" una vez nos autenticamos con Twitter. Ejecuta el manejador implementado con "TwitterStrategy" de Passport en el archivo de configuración de Passport.
/auth/facebook	GET	Redirecciona a una página de inicio de sesión de Facebook. Usada para el inicio de sesión con Facebook si se tiene cuenta asociada.
/auth/twitter	GET	Redirecciona a una página de inicio de sesión de Twitter. Usada para el inicio de sesión con Twitter si se tiene cuenta asociada.
/auth/facebook/callback	GET	Ruta a la que se redirecciona desde "/auth/facebook" una vez nos autenticamos con Facebook. Ejecuta el manejador implementado con "FacebookStrategy" de Passport en el archivo de configuración de Passport.
/auth/twitter/callback	GET	Ruta a la que se redirecciona desde "/auth/twitter" una vez nos autenticamos con Twitter. Ejecuta el manejador implementado con "TwitterStrategy" de Passport en el archivo de configuración de Passport.
/iniciar	GET	Renderiza una página en la que se puede iniciar sesión mediante el relleno de un formulario.
/iniciar	POST	Inicia sesión con las credenciales de la cuenta local del usuario. Ejecuta el manejador implementado con "LocalStrategy" para el inicio de sesión local en el archivo de configuración de Passport.

/registrarse	GET	Renderiza una página en la que se puede crear una cuenta de usuario local mediante el relleno de un formulario.
/registrarse	POST	Registra un usuario con los datos que se pasan en el cuerpo de la petición. El usuario en todo caso aún no es válido y no podrá acceder a la página. Con el fin de validar el usuario se le envía un correo a la cuenta de email que pasa en la petición con una ruta a la que debe acceder para validar su cuenta. "/:id_usuario/confirmar".
/logout	GET	Cierra la sesión del usuario.
/:id_usuario	GET	Perfil del usuario que coincide con la id pasada como parámetro en la ruta. Este perfil será visible por todos los usuarios.
/:id_usuario/confirmar	GET	Valida al usuario, dejándole acceder al contenido de la página.
/resetear/:token	GET	Comprueba que la ruta es válida, es decir existe un usuario al que se le ha asignado un "token" o código aleatorio con el fin de recuperar sus crendenciales y hace menos de una hora de ello. En caso afirmativo renderiza una página en la que el usuario puede cambiar su contraseña.
/resetear/:token	PUT	Comprueba que la ruta es válida, es decir existe un usuario al que se le ha asignado un "token" o código aleatorio con el fin de recuperar sus crendenciales y hace menos de una hora de ello. En caso afirmativo actualiza la contraseña del usuario con la que se ha pasado en el cuerpo de la petición.

Tabla 17 - Rutas referentes a los usuarios. Relativas a "/app/usuarios".

- **Denuncias**

Ruta	*Método*	*Descripción*
/	GET	Renderiza una página en la que se muestra las últimas denuncias añadidas en modo lista y distribuidas en páginas. Es por ello que la página "page" es un parámetro obligatorio en el QueryString (Ej.: "/app/denuncias?page=2").
/api	GET	Consiste en una API JSON que devuelve las denuncias que satisfacen los parámetros de búsqueda. Estos parámetros son añadidos como parte del QueryString (Ej.: "/app/denuncias/api?titulo=suciedad&tags=suciedad,olores").
/visor	GET	Renderiza la página en la que se muestra inicialmente un mapa con las denuncias creadas en el día actual.

/nueva	GET	Renderiza la página en la que se muestra un mapa con las funcionalidades para crear una nueva denuncia.
/nueva	POST	Crea la denuncia conforme los datos que se le pasen en el cuerpo de la petición.
/imagen/temporal	POST	Se encarga de gestionar la subida de una imagen en una carpeta temporal.
/imagen/temporal	DELETE	Se encarga de eliminar una imagen ubicada en un directorio temporal.
/imagen	DELETE	Elimina la imagen de una denuncia ya creada.
/:id_denuncia	GET	Redirecciona a "/app/denuncias/:id_denuncia/:titulo".
/:id_denuncia	PUT	Actualiza la denuncia cuyo identificador se pasa como parámetro en la URL con los datos que se le pasan en el cuerpo de la petición.
/:id_denuncia	DELETE	Elimina la denuncia cuyo identificador se pasa como parámetro en la URL.
/:id_denuncia/actualizar	GET	Renderiza una página en la que se muestra un mapa con las funcionalidades para actualizar la denuncia cuyo identificador se pasa como parámetro en la URL.
/:id/comentario/:id/replicar	POST	Añade una réplica al comentario cuyo identificador se pasa como parámetros en la URL de la denuncia cuyo identificador también se pasa como parámetro en la URL.
/:id_denuncia/comentar	POST	Añade un comentario en la denuncia cuyo identificador se pasa como parámetro.
/:id_denuncia/:titulo	GET	Renderiza la página en la que se muestra un mapa con toda la información de la denuncia cuyo identificador se pasa como parámetro en la URL.

Tabla 18 - Rutas referentes a las denuncias. Relativas a "/app/denuncias".

Las funciones middleware nos permiten hacer ciertas comprobaciones y pasar ciertos datos entre funciones. Todas las funciones middleware utilizadas se ejecutan antes del manejador de rutas, excepto algunas encargadas de manejar errores.

Algunas funciones middleware se han escrito en archivos dentro de una carpeta en el directorio "*app*" del proyecto. De esta forma se evita tener que repetir el mismo código en distintas partes de la aplicación. Este proceso de exportación se lleva a cabo gracias a los objetos globales "*module*" y "*module.exports*" que nos provee *Node*.

A continuación se detalla una lista con las funciones middlewares más importantes del proyecto, que aún no se han comentado, son aquellas escritas en archivos separados en la carpeta */app/middlewares*:

ARCHIVO	DESCRIPCIÓN
datos.js	Pasa como objeto a la respuesta de una petición los datos básicos de la aplicación como variables. Estos datos son las traducciones, mensajes de error, el propio objeto del usuario si está conectado, número de denuncias en las últimas 24 horas, …
i18n.js	Cambia el idioma por defecto almacenado en las cookies del navegador, al que se le pase como parámetro usando la clave "*lang*" en la URL de la petición.
logged.js	Comprueba que el cliente que realiza la petición es un usuario conectado y válido y continua con la ruta que se solicita, en caso contrario redirecciona a la página para *loggearse* en la aplicación.
usuario.js	Pasa como variables locales las notificaciones y acciones del usuario si está conectado.

Tabla 19 - Funciones middlewares escritas en archivos separados.

Una función middleware es una función normal en JavaScript, que recibe tres argumentos:

```
let middleware = (req, res, next)=>{/* Esto es una función middleware */};
```

Las variables ***req*** y ***res*** son objetos que corresponden con la petición y la respuesta a esa petición, respectivamente. La variable ***next*** es una función que una vez llamada dispara la siguiente función middleware que sigue en el orden.

Estas funciones son usadas por *ExpressJS* para realizar distintas funciones en el transcurso de una petición.

El manejador de ruta también es una función middleware, pero al establecer la cabecera de la respuesta y enviarla con las funciones *res.status, res.send o res.json*, omitimos la variable *next*, ya que es redundante.

Un ejemplo de este tipo de función es:

```
let routeHandler = (req, res) =>{/* Manejador de rutas */}
```

7.6. SISTEMA DE AUTENTICACIÓN

Cualquier aplicación web que gestiona usuarios hace uso de sesiones para mantener conectado al usuario durante el uso de esta o conectar un usuario al acceder al navegador si tiene almacenadas las credenciales.

Passport nos brinda un sistema de sesiones, que no es más que una función *middleware*. Es por ello que Passport está pensado para ser usado a través de *Express*. Así pues, utilizar este sistema de sesiones para las peticiones que recibamos en el servidor de un usuario en concreto es tan sencillo como usar este *middleware*. Además hay que configurar correctamente Passport para el inicio y cerrado de sesión, así como el registro de los usuarios, para lo cual deberá comunicarse con la base de datos.

```
app.use(passport.initialize()); // Usa passport
app.use(passport.session()); // Sesiones Login Persistentes - Passport
```

Imagen 86 - Uso de sesiones de Passport. Archivo server.js.

También se le debe decir a *Passport* como debe responder cuando un usuario intente acceder desde su cuenta local o sus cuentas en redes sociales asociadas a la cuenta local y cómo responder cuando alguien ajeno a la aplicación quiera registrarse. Para ello se ha exportado la lógica a un archivo en la carpeta "*config*" del proyecto, el cual, requerimos en el archivo "*server.js*" principal de la aplicación.

```
// Configuración de passport -- Estrategias para autentificación y creación de usuarios
require('./config/config_passport_pg')(passport);
```

Imagen 87 - Pasamos el objeto passport al archivo de configuración.

Este archivo, lógicamente exporta una función que recibe el objeto *Passport* y utiliza sus métodos.

```
var LocalStrategy = require('passport-local').Strategy;
var FacebookStrategy = require('passport-facebook').Strategy;
var TwitterStrategy = require('passport-twitter').Strategy;
var usuarioModel = require('../app/models/usuario.js');
    usuarioModel = new usuarioModel();
var configAuth = require('./auth');
var validator = require('validator');
var consultas = require('../app/controllers/queries.js');

module.exports = function(passport) {

    // Serializa al usuario para la sesión
    passport.serializeUser(function(user, done) {
        done(null, user._id);
    });

    // Deserializa el usuario
    passport.deserializeUser(function(id, done) {
        usuarioModel.find_by_id(id, function(error, user){
            if(error)
                return done(error);
            done(null, user);
        });
    });

    // LOGIN LOCAL
    passport.use('local-login', new LocalStrategy({
```

Imagen 88 - Parte del archivo "config_passport_pg.js"

Se puede observar que hacemos uso de otros módulos como "*passport-local*", el cual nos facilita y ahorra escribir parte del código para iniciar sesión y registrarse de forma local y "*passport-twitter*" y "*passport-facebook*" que nos proporcionan métodos para autenticar las credenciales en cada red social respectivamente.

Al haber configurado Passport para que se comunique con nuestra base de datos y haga las comprobaciones pertinentes podremos utilizar las funciones middleware que nos provee *Passport* para la autenticación y registro (*authenticate* y *authorize*).

```
//Facebook Auth -- Callback de Facebook una vez autentificados
router.get('/auth/facebook/callback', passport.authenticate('facebook', {
    successRedirect : '/app/usuarios/perfil',
    failureRedirect : '/app#iniciar'
}));

//Facebook Auth -- Facebook Renderiza la página de Inicio de Sesión
router.get('/auth/facebook', passport.authenticate('facebook', { scope : 'email' }));
//Conectar una cuenta de Twitter con otra existente -- Callback
router.get('/conectar/twitter/callback', passport.authorize('twitter', {
    successRedirect : '/app/usuarios/perfil',
    failureRedirect : '/app/usuarios/perfil'
}));

//Conectar una cuenta de Twitter con otra existente
router.get('/conectar/twitter', passport.authorize('twitter', { scope : 'email' }));
// Rutas iniciar sesión
router.route('/iniciar')
// Página iniciar sesión -- OK
.get(function(req, res) {
        res.redirect('/app#iniciar');
})
// Método POST iniciar sesión -- OK
.post(passport.authenticate('local-login', {
    successRedirect : '/app/usuarios/perfil',
    failureRedirect : '/app#iniciar',
    failureFlash : true
}));
```

Imagen 89 - Funciones middleware de passport usadas en el enrutamiento.

También nos provee métodos que son accesibles desde el objeto *request* de una petición y que nos dan información acerca de la sesión, como por ejemplo, si la petición corresponde a un usuario conectado. Esto nos facilita responder de una determinada manera cuando un usuario no identificado trata de acceder a rutas que solo un usuario identificado pueda acceder.

```
module.exports = function (req, res, next) {

    if (req.isAuthenticated()){
        if(req.user.local.valid)
                next();
        else{
                req.flash('error', 'Revisa tu correo:"' + req.user.local.email + '" y activa tu cuenta.');
                res.redirect('/app');
        }
    }
    else{
        console.log('no');

        req.flash('error', 'Debes estar loggeado');
        res.redirect('/app#iniciar');
    }
};
```

Imagen 90 - Función middleware para saber si el cliente que realiza la operación es un usuario conectado.

En la imagen se puede ver como en el objeto *request* "*req*" está accesible un método llamado "*isAuthenticated*" que nos indica si la petición contiene una sesión activa de un usuario.

7.7. VISTAS

Jade es un motor de renderizado de vistas que viene con *ExpressJS*. Tiene una sintaxis sencilla que suprime las marcas HTML y se basa en la indentación del código. En realidad estas vistas o plantillas se compilan a HTML y se sirven como tales.

Para usar Jade como el motor de renderizado de plantillas de la aplicación se debe configurar nuestra aplicación Express de tal forma.

```
app.set('view engine', 'jade'); // Motor de renderizado de vista - Jade
```

Imagen 91 - Configurar Express para usar Jade como motor de renderizado plantillas.

Lo realmente potente de estos sistemas de plantillas es que nos permiten pasarles variables del lado del servidor e interactuar con ellas creando contenidos *HTML* dinámicos.

También cabe destacar que Jade nos permite hacer código reusable, gracias a la definición de bloques y evitar tener que escribir el mismo código para dos plantillas diferentes. En este sentido también podemos hacer plantillas "*base*" y hacer que otras "*sub-plantillas*" hereden la plantilla base y añadan cierta información propia de cada plantilla.

En el caso que concierne se ha decidido por la siguiente estructura de plantillas.

7.7.1. PLANTILLAS BASE

Las plantillas base sirven para definir la estructura básica que otras plantillas deben de seguir. Jade permite heredar una plantilla padre y sobrescribir los bloques de esta con más etiquetas. Esto hace que el código común se pueda escribir en el mismo archivo, se evita redundancia en el código y facilita seguir una misma línea en el diseño.

Las plantillas "*base*" creadas se describen a continuación:

- **layout.jade** → Plantilla base empleada para la página de bienvenida de la aplicación debido a que contiene una portada con imágenes.
- **layout_sin_cabecera.jade** → Plantilla base empleada para ciertas páginas de la aplicación que requieren tener la portada de la aplicación.
- **map_layout.jade** → Plantilla que contiene la inicialización de un mapa de *OpenLayers* y algunos componentes del mapa comunes para todas las sub-plantillas que requieran un mapa.

7.7.2. PANTILLAS REFERENTES A LAS RUTAS DE DENUNCIAS

A continuación se describen las plantillas referentes a algunas rutas de las rutas referentes a las denuncias:

- **denuncia.jade**

Plantilla que hereda los componentes de "*map_layout.jade*" y añade los componentes necesarios para ver la información una denuncia, renderizar la geometría de la denuncia en el mapa, etc. Como vemos en la imagen este mapa dispone de controles específicos para interactuar con la denuncia. Esta vista se renderiza al acceder a la ruta:

"/app/denuncias/:id_denuncia"

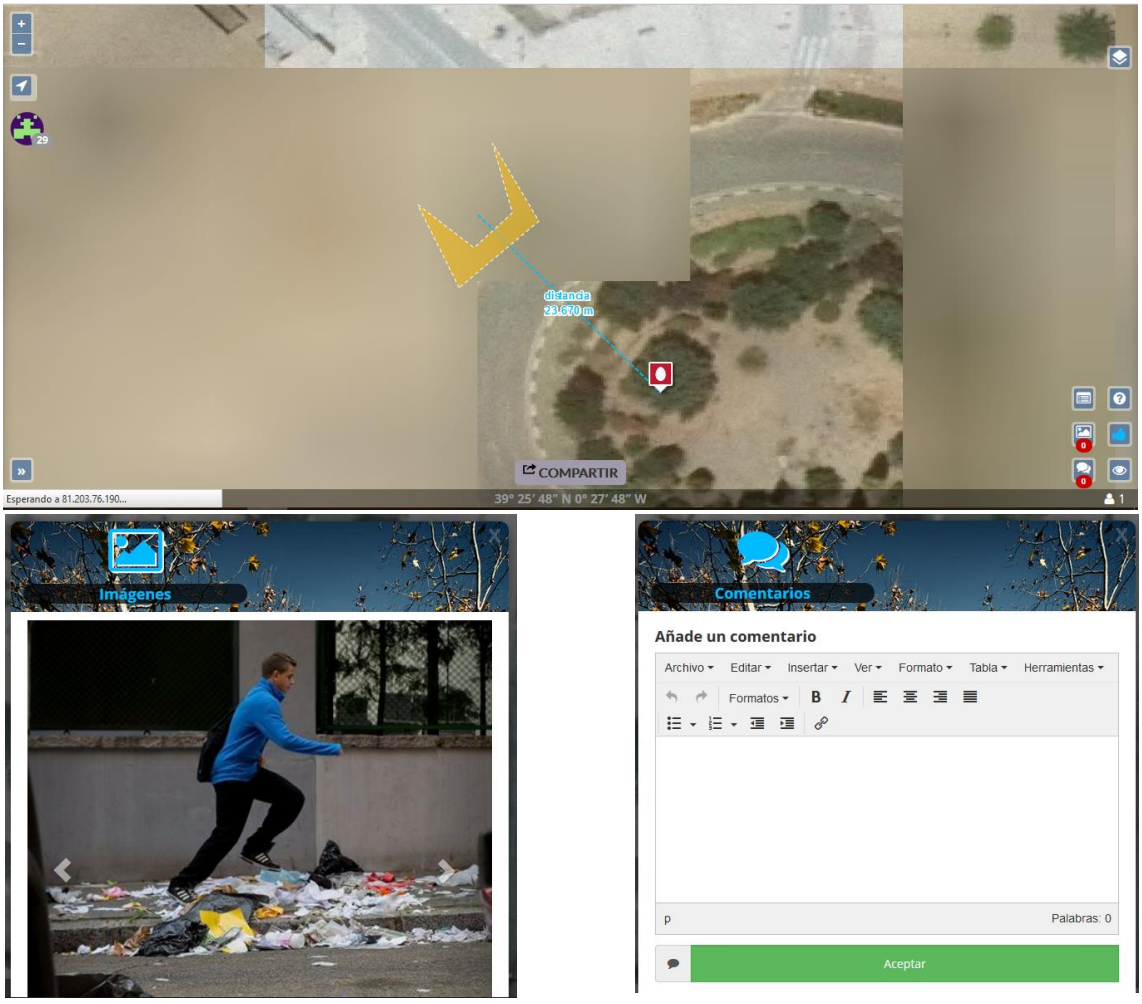


Imagen 92 - Página de una denuncia.

- **editar.jade**

Plantilla que hereda los componentes de "*map_layout.jade*" y añade los componentes necesarios para ver y editar la información de una denuncia. Esta vista se renderiza al acceder a la ruta:

"/app/denuncias/:id_denuncia/actualizar"

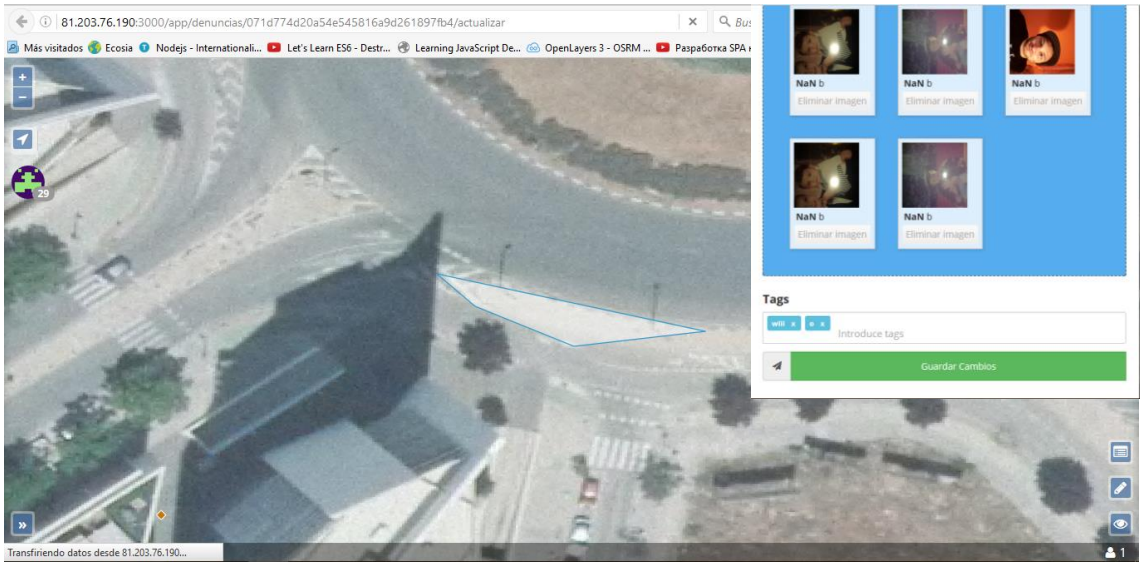


Imagen 93 - Página para actualizar una denuncia.

- **lista.jade**

Plantilla que hereda los componentes de "*layout_sin_cabecera.jade*" y añade los componentes necesarios para ver una página de denuncias presentadas en forma de lista, suponiendo las denuncias ordenadas por fecha descendente y un número máximo de 10 denuncias por página. Esta vista se renderiza al acceder a la ruta:

"/app/denuncias o /app/denuncias?page=[página válida]"

Imagen 94 - Página que muestra el listado de denuncias.

- **nueva.jade**

Plantilla que hereda los componentes de "*map_layout.jade*" y añade los componentes necesarios para crear una denuncia. Esta vista se renderiza al acceder a la ruta:

"/app/denuncias/nueva"

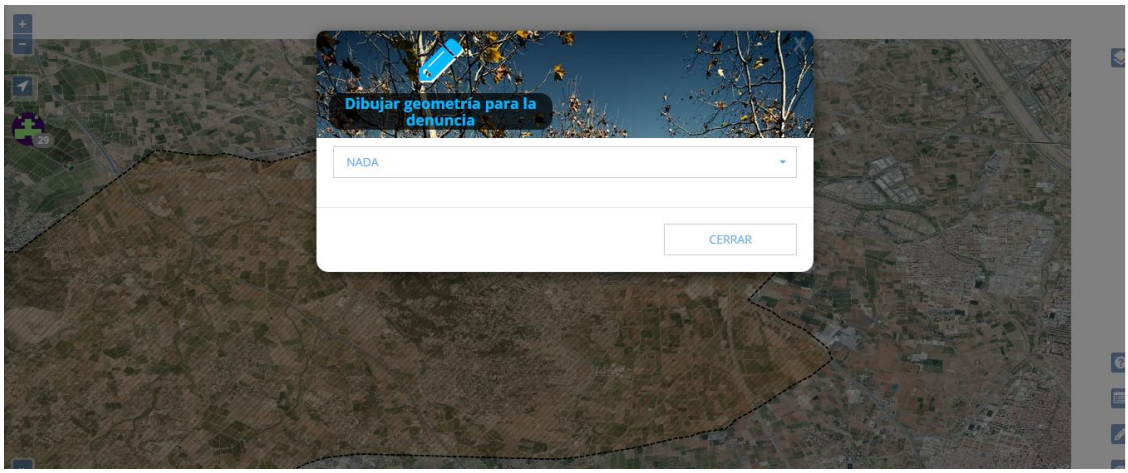


Imagen 95 - Página para añadir una nueva denuncia.

- **visor.jade**

Plantilla que hereda los componentes de "*map_layout.jade*" y añade los componentes necesarios para visualizar denuncias y realizar diversas tareas. Esta vista se renderiza al acceder a la ruta:

"/app/denuncias/visor?..."

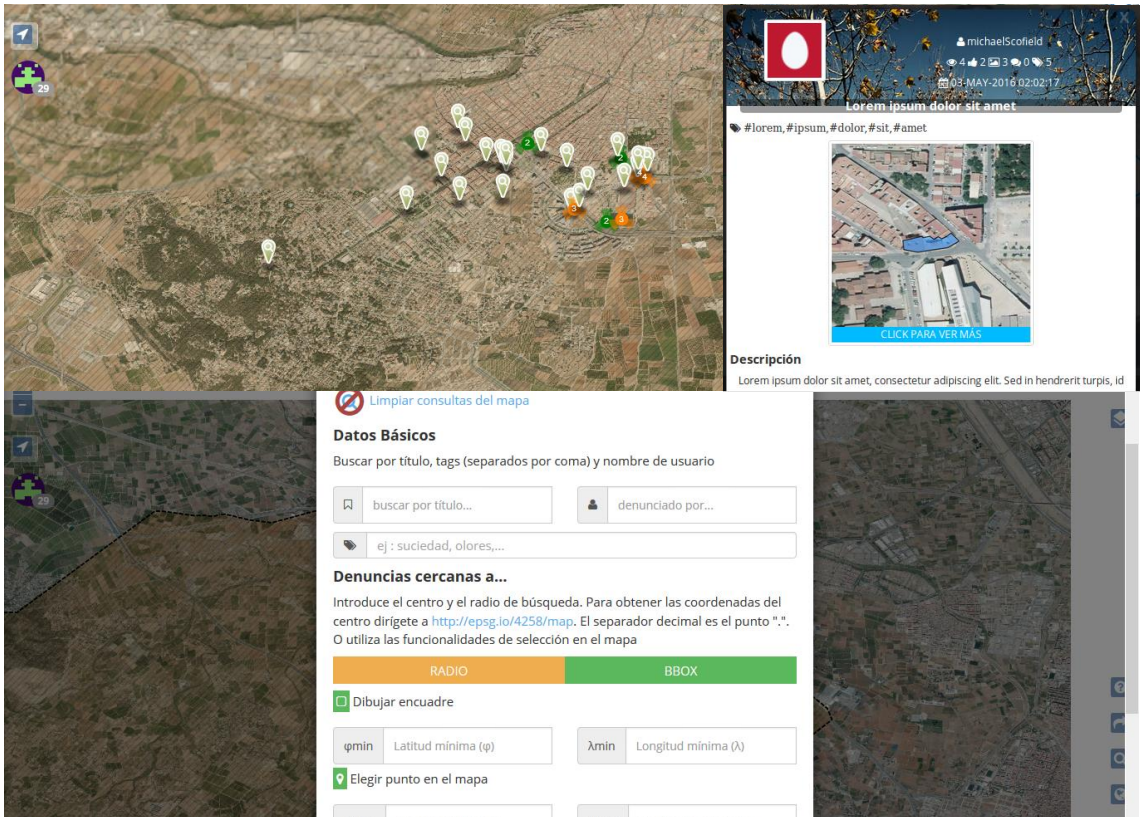


Imagen 96 - Visor de denuncias.

7.7.3. PLANTILLAS REFERENTES A LAS RUTAS DE USUARIOS

- **perfil.jade**

Plantilla que hereda los componentes de "*layout_sin_cabecera.jade*" y renderiza el perfil del usuario conectado con toda su información. Esta vista se renderiza al acceder a la ruta:

"/app/usuarios/perfil"

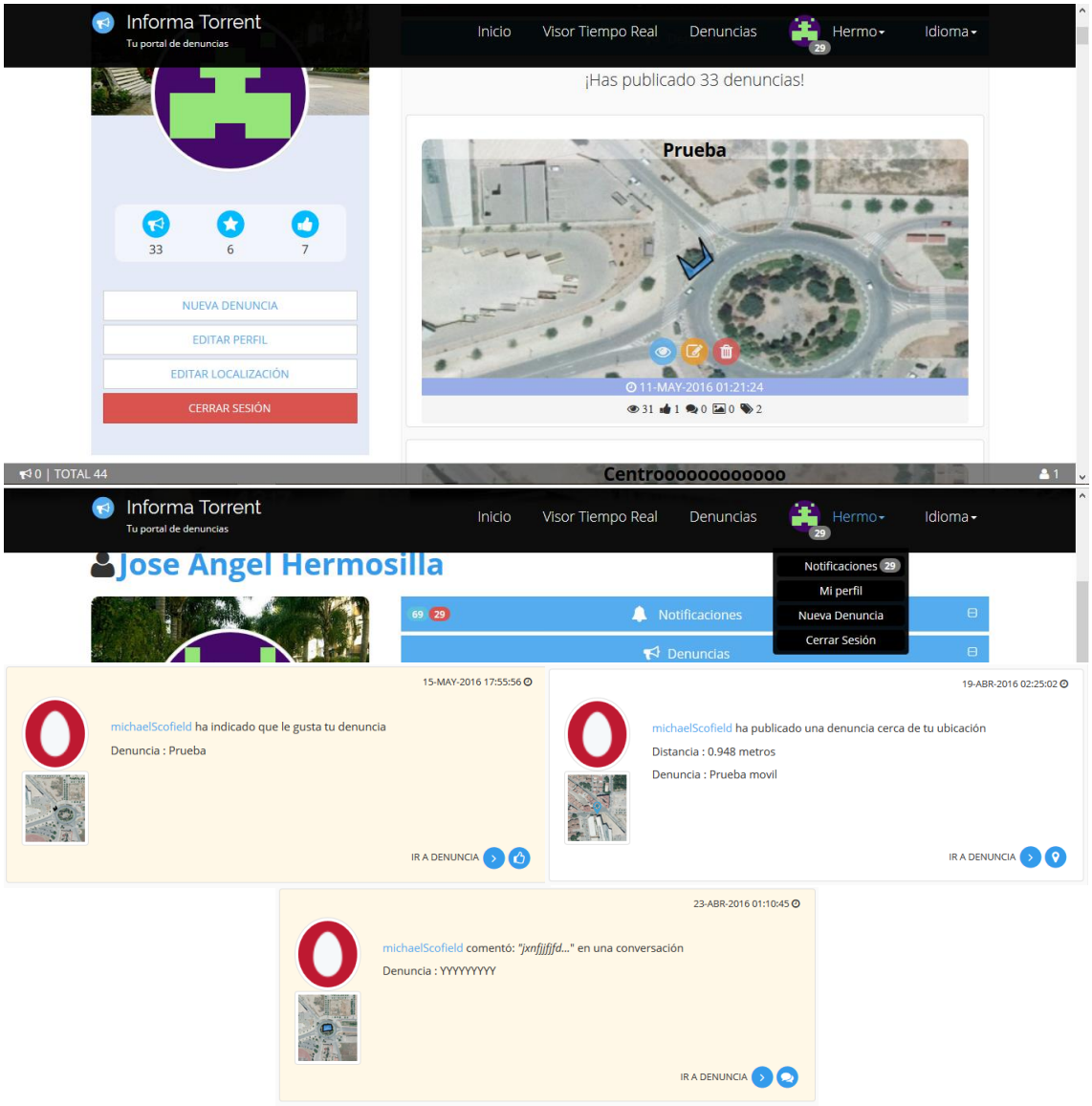


Imagen 97 - Perfil de usuario.

- **perfil_visible.jade**

Plantilla que hereda los componentes de "*layout_sin_cabecera.jade*" y renderiza el perfil un usuario con cierta información. Esta vista se renderiza al acceder a la ruta:

"/app/usuarios/:id_usuario"

- **editar_perfil.jade**

Plantilla que hereda los componentes de "*layout_sin_cabecera.jade*" y añade formularios para editar la información del usuario. Esta vista se renderiza al acceder a la ruta:

"/app/usuarios/perfil/actualizar"

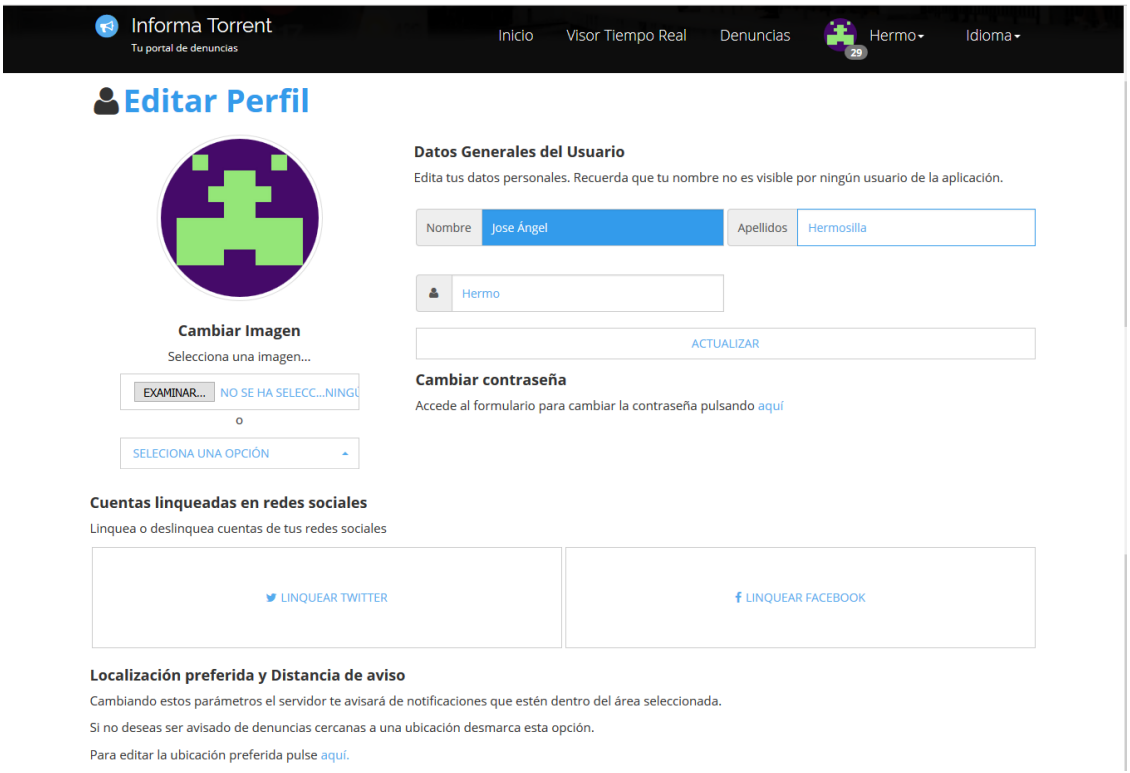


Imagen 98 - Página para editar el perfil de usuario.

- **editar_pass.jade**

Plantilla que hereda los componentes de "*layout_sin_cabecera.jade*" y añade formularios para poder actualizar la contraseña. Esta vista se renderiza al acceder a la ruta:

"/app/usuarios/perfil/password"

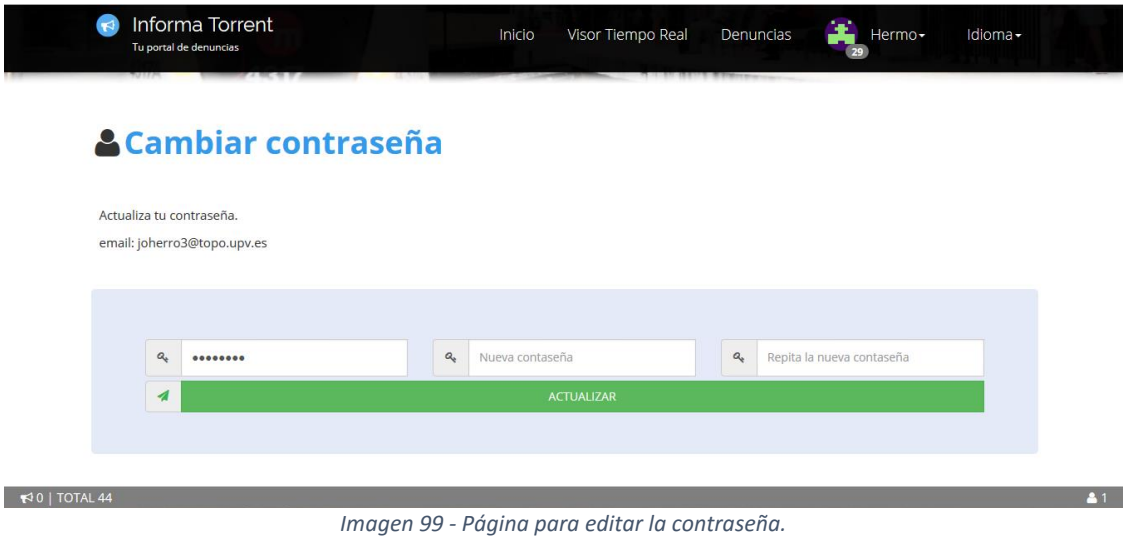


Imagen 99 - Página para editar la contraseña.

- **editar_loc.jade**

Plantilla que hereda los componentes de "*map_layout.jade*" y añade funcionalidades para ver y editar la localización preferida y el radio a usar como parámetros para calcular el área de influencia y avisar al usuario de nuevas denuncias. Esta vista se renderiza al acceder a la ruta:

"/app/usuarios/perfil/localizacion"

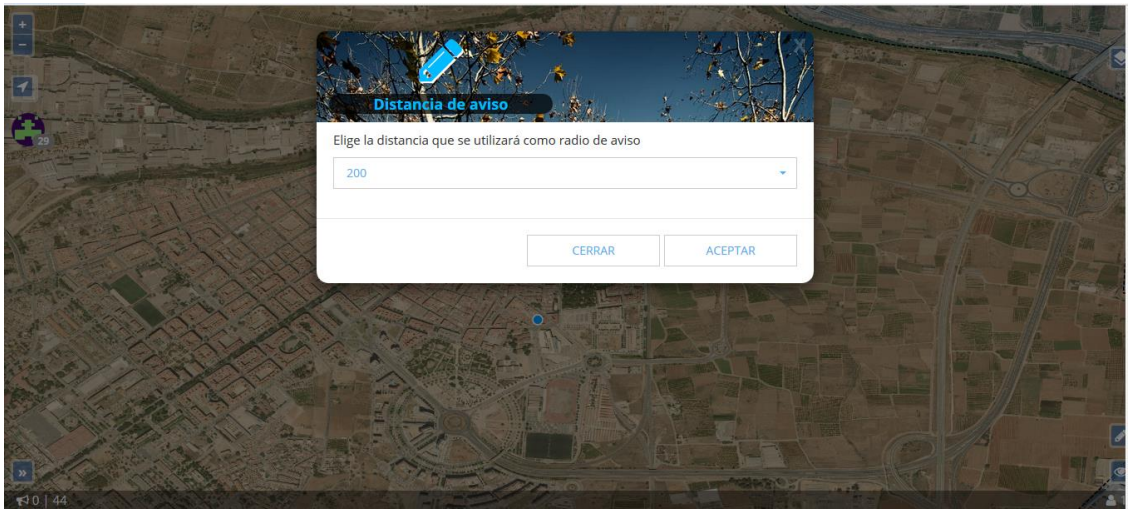


Imagen 100 - Página para editar los parámetros "localización preferida" y "distancia de aviso".

- **cambiar_pass_token.jade**

Plantilla que hereda los componentes de "*layout_sin_cabecera.jade*" y añade un formulario para actualizar la contraseña. Esta vista se renderiza al acceder a la ruta:

"/app/usuarios/resetear/:token"

Imagen 101 - Muestra el email enviado tras haber rellenado el formulario para recuperar las credenciales.

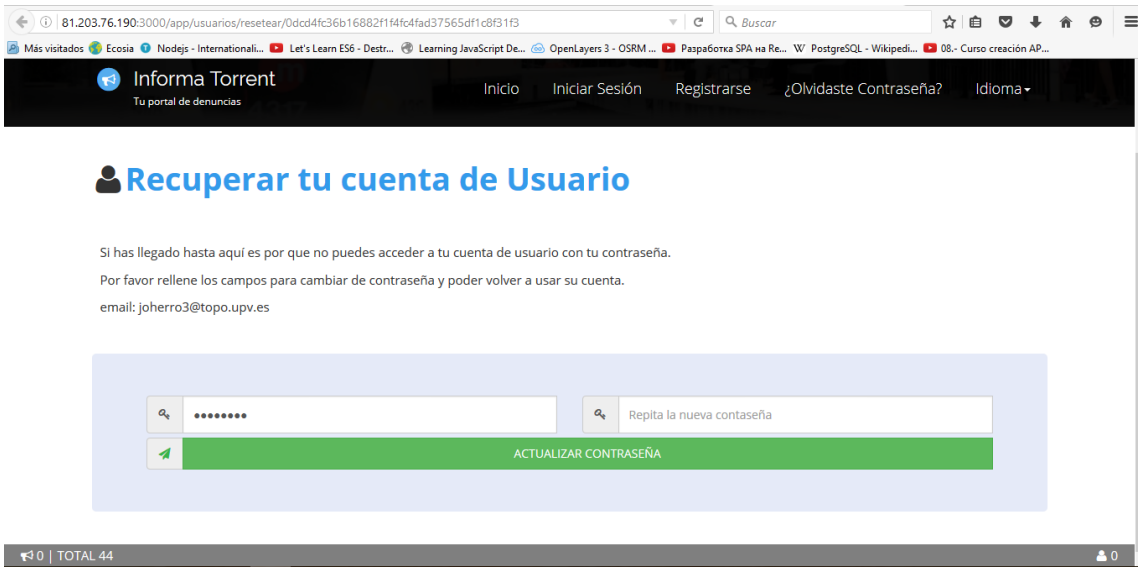


Imagen 102 - Formulario para cambiar la contraseña, al cual se accede desde la dirección que se indica en el correo.

7.7.4. PLANTILLA DE ERROR

- **error.jade**

Plantilla que hereda los componentes de "*layout_sin_cabecera.jade*" y añade un texto informativo referente al error que se ha producido, así como el estado del error (*500, 404*, etc.). Esta vista se renderiza cuando hay algún error renderizando alguna otra vista. Un error común suele ser que tratemos de acceder a un recurso que no existe, como puede ser la página de una denuncia que no existe.

Imagen 103 - Página de error. Errores 500 y 404.

7.7.5. PLANTILLA PRINCIPAL

- **index.jade**

Plantilla que hereda los componentes de "*layout.jade*" y añade los formularios para el inicio de sesión y registro de un usuario y otro para recuperar las credenciales. Esta vista se renderiza al acceder a la ruta:

"/app"

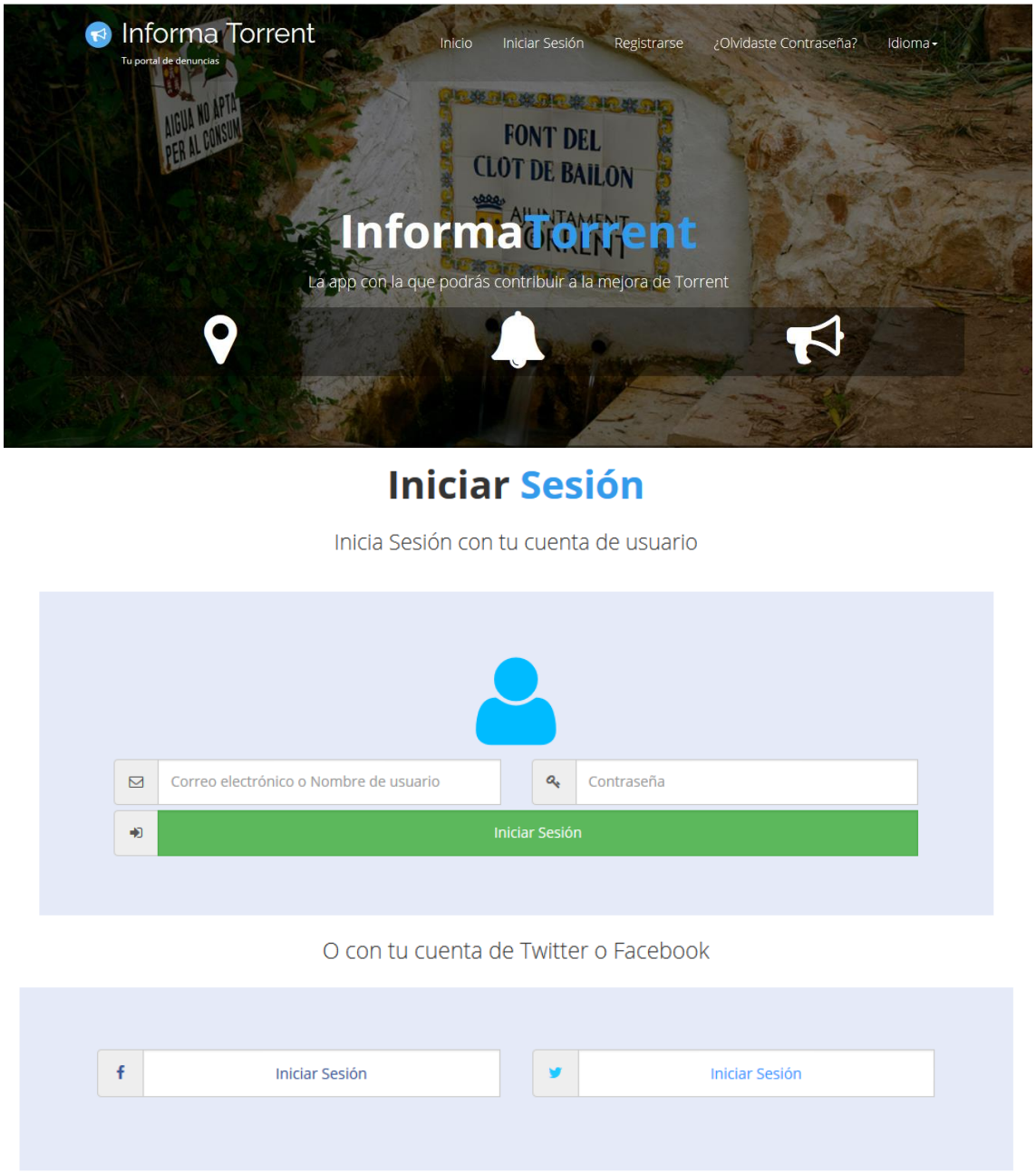


Imagen 104 - Página principal. Portada y menú de Inicio de sesión.

También contiene los siguientes formularios para el registro o la pérdida de credenciales:

Imagen 105 - Formularios para el registro de un nuevo usuario y la recuperación de las credenciales, respectivamente.

7.7.6. ARCHIVOS JAVASCRIPT DEL LADO DEL CLIENTE

Las vistas Jade pueden almacenar código JavaScript en el mismo documento gracias a sus elementos "*script*" que se transforman en marcas "*script*" una vez *transpilado* el código *JADE* en *HTML*. Es cierto que almacenar todo el código *JavaScript* en las plantillas puede hacer que estas vean incrementado su tamaño en numerosas líneas de código y pierda legibilidad haciéndolo difícil de depurar.

Es por ello que se decide crear archivos *JavaScript* separados y requerirlos en las plantillas en vez de implementar la lógica en ellas. De esta forma, además, se conseguirá que el código sea reusable.

Estos archivos se encargarán de implementar la lógica del cliente web, como por ejemplo la configuración de un mapa, la creación de nuevos controles para el mapa, el procesado del envío de un formulario, etc.

A continuación se detalla una lista con los archivos *JavaScript* del lado del cliente que se han realizado.

7.7.6.1. PUESTA EN MARCHA DE OPENLAYERS

- **mapa.js** → Se encarga de inicializar el mapa de *Openlayers* con la configuración y componentes comunes que tienen los mapas creados.
- **capas.js** → Se encarga de crear las variables que contendrán cada una de las capas que se usan en los mapas.
- **proj.js** → Define la proyección *ETRS89 (EPSG:4258)*, mediante la librería *projJS*, que se usará en los mapas.
- **style.js** → Inicializa las variables que contienen estilos de *Openlayers* usados para simbolizar las capas de denuncias.
- **olvisorapp.js** → Se encarga de inicializar los componentes para el mapa del visor en tiempo real de la aplicación (Cargar las denuncias en el mapa, escuchar cuando se añaden denuncias, simbolizar cada denuncia y asignarle su tipo de marcador, agrupar las denuncias según el nivel de zoom (*Clúster*), mostrar la información al hacer *click* sobre una denuncia, etc).

7.7.6.2. CONTROLES PROPIOS PARA OPENLAYERS

- **ayuda.js** → Control creado para mostrar diálogos de ayuda para cada mapa, con videos y texto de cómo usar cada componente o sección. Estos diálogos funcionan con una cookie que se inserta en el navegador en el caso de que marquemos la opción "no volver a mostrar", de modo que si no tenemos marcada esta opción se mostrará al iniciar la página. Pueden ser reabiertos en cualquier momento al pulsar sobre un botón en el mapa.

- **bitly.js** → Control creado para obtener el estado del mapa actual basado en la propiedad "*hash*" de la URL, que almacena la posición y el nivel de zoom actuales y los identificadores de las denuncias que hay en el mapa, y utilizar el servicio *Bitly* para crear una URL más corta y útil para compartir el mapa. La *URL* generada se muestra en un diálogo.
- **comentarios_denuncia.js** → Control creado para mostrar los comentarios y réplicas que contiene una denuncia y los formularios para poder enviar un comentario o una réplica a un determinado comentario. Además se encarga de manejar el envío de dichos formularios.
- **cql.js** → Control creado para filtrar las capas por alguno de sus atributos, usando una consulta *CQL* a *Geoserver*. Este control solo se emplea en los mapas del Geoportal.
- **download_bbox.js** → Control creado para la descarga de cartografía desde el mapa filtrando los elementos por el encuadre que dibujemos en el mapa. Para ello enviamos una petición *GetFeature* al servicio *WFS* utilizando entre otros el parámetro *BBOX*. Este control solo se emplea en los mapas del Geoportal.
- **draw.js** → Control creado para manejar los eventos de creación de elementos (*puntuales, lineales, poligonales*) sobre el mapa.
- **external_wms.js** → Control creado para cargar capas de otros servicios *WMS* externos. Se encargará de enviar al servidor una petición con la *URL* del servicio *WMS*, el servidor devolverá su documento de capacidades que será leído por el "*parser*" de *Openlayers* para este tipo de documentos y se mostrará un formulario con las capas que contiene. En este formulario el usuario podrá elegir cuales desea cargar y al aceptar se crearán las distintas capas y se añadirán al mapa. Este control solo se emplea en los mapas del Geoportal.
- **get_feature_info.js** → Control creado para realizar peticiones *GetFeatureInfo* al servidor *WMS* al hacer *click* sobre el mapa. Este control solo se emplea en los mapas del Geoportal.
- **imágenes_denuncia.js** → Control creado para mostrar en un diálogo las imágenes de una denuncia.
- **info_denuncia.js** → Control creado para mostrar en un diálogo la información de una denuncia.
- **lateral.js** → Control creado para mostrar en un diálogo los formularios para el relleno o edición de la información de una denuncia.
- **layers.js** → Se encarga de mostrar un menú lateral con el componente encargado de mostrar y manejar las capas cargadas en el mapa.
- **like_denuncia.js** → Se encarga de renderizar un *checkbox* que muestra el apoyo o no del usuario por una denuncia. Al clicar sobre el modificamos su estado.
- **query_denuncias.js** → Muestra un formulario para poder buscar denuncias según ciertos criterios y mostrarlas sobre el mapa.
- **tracking.js** → Control que muestra la posición del usuario en ese momento y actualiza su posición cuando hay cambios.
- **tracking_denuncia_cerca.js** → Realiza la misma función que el componente "*tracking*" y además muestra las denuncias cercanas a la posición del usuario, ordenadas por distancia al usuario y fecha, para ello envía peticiones al servidor *NodeJS*.

7.7.6.3. OTRO ARCHIVOS

- **client_sockets.js** → Se encarga de manejar algunos de los eventos en tiempo real (*websockets*) del lado del cliente con el servidor *NodeJS*.
- **template.js** → Se encarga de responder de determinada forma a ciertos eventos del navegador como el *scroll* y el *resize* para la correcta visualización en todos los dispositivos.
- **utils.js** → Archivo que contiene funciones útiles para renderizar ciertas partes de la aplicación como las notificaciones y denuncias de un usuario en el perfil.

7.8. WEBSOCKETS, EVENTOS EN TIEMPO REAL

Uno de los rasgos que hace más interesante esta aplicación son los *websockets* ya que permiten un flujo de intercambio bidireccional de información entre el cliente y el servidor continuo y tan rápido que se puede considerar "*en tiempo real*", en contraposición a las peticiones *HTTP* que no permiten tener este intercambio continuo de información una vez cargada la página. En la aplicación se está usando el módulo socket.io que implementa una librería de *websockets* para *NodeJS* y es uno de los más usados y mejor valorados.

Con la ayuda de esta tecnología se han implementado eventos para hacer más dinámico ciertos contenidos de la aplicación para que se actualicen cuando haya cambios en el servidor sin necesidad de refrescar la página.

En la aplicación se han usado para informar a otros usuarios de la una acción de un usuario como publicar una denuncia, por ejemplo, cuando un usuario realiza una denuncia se busca entre los usuarios de la base de datos a quienes afecta la denuncia, se guarda la notificación en la base de datos de todos los usuarios afectados y de entre estos comprueba cuáles están conectados y se emite esa misma notificación para que se muestre en todas sus ventanas abiertas sin necesidad de refrescar la página en ninguna de ellas. Por lo tanto algunos los eventos se van a emitir desde el manejador de rutas de la petición *HTTP* correspondiente.

Para esto es necesario tanto código del lado del servidor como del cliente para modificar el contenido cuando se reciba un evento.

También se han usado en ciertas ocasiones como sustituto de peticiones *HTTP*, sobre todo en algún sistema de búsqueda implementado.

```
// Alguien ha emitido un evento para saber qué denuncias tiene cerca
// Emite su posición
socket.on('tengo_denuncias_cerca_?', function(data){
        // si no data salimos
        if (!data) return;
        // Ejecutamos la consulta
        denunciaModel.denuncias_cerca(data, function(error, denuncias){
                if(error)
                        socket.emit('si_que_tengo_denuncias_cerca', []);
                else
                        socket.emit('si_que_tengo_denuncias_cerca', denuncias);
        });
});
```

Imagen 106 - Ejemplo de comunicación entre cliente y servidor mediante sockets.

En la imagen anterior se escucha a un evento determinado "*tengo_denuncias_cerca_?*" que se emite desde el cliente que envía una posición, se buscan denuncias cercanas a esa posición en la base de datos gracias al modelo de denuncias y se emiten las denuncias encontradas, si las hay. Algunos de los eventos están definidos en el archivo "*sockets.js*" de la carpeta *app/controllers*, otros se emiten desde el manejador de rutas de una petición *HTTP*, por ejemplo la petición *POST* para añadir una denuncia.

Un ejemplo de cómo se realizan las modificaciones de los elementos *DOM* de la página cuando se recibe un evento en el lado del cliente se puede observar en la siguiente imagen:

```
// Un usuario ha publicado una denuncia cerca de
// nuestra ubicación
num_denuncias_io.on('denuncia_cerca', function(data){

        audio.play();
        //console.log(JSON.stringify(data));
        //data.noti = data.noti[1];
        data.noti.profile_from = data.from.profile;
        data.noti.denuncia = data.denuncia;
        // Aumentamos en uno las notificaciones nuevas
        var nuevas = parseInt($('.noti_up:eq(1)').text()) + 1;
        $('.noti_up').empty();
        $('.noti_up').append(nuevas);
        // Aumentamos en uno las notificaciones totales
        var not_tot = parseInt($('.noti_tot').text()) + 1;
        $('.noti_tot').empty();
        $('.noti_tot').append(not_tot);

        // Insertamos la notificación en la lista de notificaciones
        var html = getNotificacionRow(data.noti, traducciones);
        $(html).prependTo($('#notificaciones > .panel-body'));
        $('#notificaciones > .panel-body').find('p.lead').prependTo($('#notificaciones > .panel-body'));

        // Insertamos la notificación en el array de notificaciones
        if(notificaciones) notificaciones.unshift(data.noti);

        notificar('Replica', data.from.profile.username + ' ha publicado una denuncia cerca de tu ubicación : ');
```

Imagen 107 - Ejemplo de evento recibido en el cliente mediante websockets.

8. CONCLUSIONES

Tras la consecución de todos los puntos anteriores, se pueden valorar los objetivos conseguidos.

Se ha conseguido desarrollar una IDE con una gran diversidad de datos cartográficos, aplicando la normativa Europea y las distintas normas y estándares abiertos existentes para la implementación de los servicios relacionados con la IG.

El Geoportal como cara visible de nuestra IDE muestra un diseño sencillo, amigable y bastante elaborado. A parte de los servicios mínimos de visualización y localización, el Geoportal ofrece otro tipo de servicios complementarios que mejoran la experiencia del usuario.

Cabe destacar el esfuerzo realizado en la traducción de los dos entornos desarrollados ya que tanto la aplicación como el Geoportal son multilenguaje y soportan el castellano, valenciano e inglés.

Respecto a la aplicación, cabe destacar el proceso de investigación de las nuevas tecnologías para la consecución de la misma, el proceso de aprendizaje, el seguimiento de las distintas versiones, búsqueda y solución de errores, etc. Todas estas situaciones no se ven reflejadas en este documento y pienso que aunque se den por hecho, deben ser nombradas de alguna forma.

La aplicación cumple con el objetivo de ser una herramienta social para los ciudadanos de Torrent, la cual utiliza la IG como medio para ubicar un cierto tipo de denuncia. Ha sido elaborada utilizando una tecnología novedosa y eficiente como *NodeJS* y se alimenta de IG empleando consultas sobre una base de datos espacial.

Todo el desarrollo se ha llevado a cabo utilizando software completamente libre y en ocasiones de código abierto lo que supone un ahorro de trabajo y costes importante. El software de código abierto ha sido especialmente útil debido a las modificaciones que se pueden hacer sobre el código original.

Se concluye con lo anteriormente expuesto que se han empleado de forma correcta las tecnologías necesarias para llevar a cabo los objetivos planteados al principio de este documento y por lo tanto se han logrado dicho objetivos.

9. BIBLIOGRAFÍA

Algunas notas sobre EcmaScript 6. (s.f.). Obtenido de https://babeljs.io/docs/learn-es2015/

AskUbuntu. (s.f.). Obtenido de http://askubuntu.com/

Documentación de Express.JS. (s.f.). Obtenido de http://expressjs.com/es/api.html

Documentación de Geoserver. (s.f.). Obtenido de http://docs.geoserver.org/latest/en/user/

Documentación de Node.JS. (s.f.). Obtenido de https://nodejs.org/en/docs/

Documentación de OpenLayers 3. (s.f.). Obtenido de http://openlayers.org/en/latest/apidoc/

Documentación de PassportJS. (s.f.). Obtenido de http://passportjs.org/docs

Documentación de Socket.io. (s.f.). Obtenido de http://socket.io/docs/#

GIS StackExchange. (s.f.). Obtenido de http://gis.stackexchange.com/

IDE de la Comunidad Valenciana (Terrasit). (s.f.). Obtenido de http://terrasit.gva.es/

Inspire. (s.f.). Obtenido de http://inspire.ec.europa.eu

Instituto Geográfico Nacional (IGN). (s.f.). Obtenido de http://www.ign.es

Llario, J. C. (s.f.). *Apuntes de la asignatura de IDE*.

Mozzilla Developer Network. (s.f.). Obtenido de https://developer.mozilla.org/en-US/docs/Web/JavaScript

NPM (Node Package Manager). (s.f.). Obtenido de https://www.npmjs.com/

Repositorios en GitHub. (s.f.). Obtenido de https://github.com/

StackOverflow. (s.f.). Obtenido de http://stackoverflow.com/

StackOverflow en Español. (s.f.). Obtenido de http://es.stackoverflow.com/

Wikipedia. (s.f.). Obtenido de http://es.wikipedia.org

10. AGRADECIMIENTOS

Quiero dar las gracias a quienes han creído en este proyecto y han apoyado y colaborado de alguna forma y en especial a José Carlos Martínez Llario, tutor de este proyecto, por su sabiduría y profesionalidad a la hora de resolver dudas y por orientarme cuando no sabía cómo avanzar.

También quiero dar las gracias a mi familia y amigos, los cuales han sido testigos de la evolución de este documento y sus críticas han servido para mejorar el producto final.