SmartCampus

Laboratorio de Ingeniería de Software 2015-2016

DANIEL FORCÉN (558471)
EDUARDO IBÁÑEZ (528074)
CRISTINA LAHOZ (544393)
PATRICIA LÁZARO (554309)
JORGE MARTÍNEZ (571735)

TABLA DE CONTENIDO

1. INTRODUCCIÓN

1.1 Nuestra Aplicación

Smartcampus es una aplicación para dispositivos móviles, tanto iOS como Android, la cual podrá interactuar con el edificio de un campus universitario con el fin de ayudar principalmente al personal trabajador contratado en la entidad, ya sea conserjes, profesores, guardias de seguridad, empleados de limpieza… También puede ser usada en menor medida por los estudiantes, para obtener información sobre el estado de los espacios del campus.

La aplicación será capaz de mostrar en un mapa la distribución de todas las habitaciones de los edificios (en este caso Ada Byron, Torres Quevedo y Betancourt) sobre un mapa estándar como Open Street Map o Google Maps.

El usuario podrá mostrar sobre el mapa la información que necesite, superponiendo capas de información a través de un menú. Dispondrá de opciones para mostrar Aulas, Despachos, Laboratorios y Servicios respectivamente, todo ello categorizado con distintos colores en base al tipo de espacio que se muestre.

Estas funcionalidades serán posibles gracias a un servidor de mapas WMS, GeoServer, el cual trabajará con una base de datos PostGreSQL.

Además, será posible obtener la información de un espacio en concreto, para obtener: su temperatura, temperatura del climatizador, disponibilidad, y el estado de puertas y ventanas de dicho espacio. A través de la aplicación se podrá interactuar con estos elementos de manera sencilla (abrir/cerrar puertas o ventanas, modificar la temperatura del climatizador…)

Por último, estarán disponibles unas tareas de gestión general del campus para evitar tener que interactuar por cada espacio por separado. Estas tareas serán tales como: cerrar todos los espacios, cambiar la temperatura global…

La aplicación trabajará en un primer lanzamiento sobre el Campus Río Ebro de la universidad de Zaragoza, pero no se cierra la posibilidad de añadir nuevos campus en un futuro.

1.2 Equipo

El equipo de desarrollo estará formado por 5 personas, estudiantes de la propia Universidad de Zaragoza:

- **Daniel Forcén**: Desarrollador back-end.
- **Eduardo Ibáñez**: Scrum master y desarrollador back-end.
- **Cristina Lahoz**: Desarrolladora de base de datos.
- **Patricia Lázaro**: Desarrolladora back-end.
- **Jorge Martínez**: dueño de producto y responsable de la parte front-end del cliente.

1.3 Estructura del documento

El siguiente documento se estructura de la siguiente manera: un primer apartado, donde se da una introducción al producto, así como la presentación de los distintos miembros del equipo. Un apartado de producto, el cual explica cosas relacionadas sobre cómo se ha realizado, qué funcionalidades realiza, cómo comprobar su funcionamiento y cuál es su interfaz. Este apartado contiene los siguientes puntos:

- **Planificación de lanzamientos (separado entre primera y segunda iteración):** Se describen los distintos lanzamientos planteados para el producto, así como las funcionalidades que deberían estar implementadas y los procedimientos a cumplir en los distintos lanzamientos.
- **Análisis de riesgos:** En él se describen los distintos riesgos que podrían surgir durante el desarrollo del producto y cómo mitigarlos.
- **Requisitos:** Los requisitos que debe cumplir el producto final, consensuados con el cliente.
- **Interfaz de Usuario:** Las distintas pantallas que tendrá la aplicación cliente, así como su funcionalidad. Esta parte puede ser útil para el usuario, de forma que puede saber cómo navegar por ella.
- **Arquitectura (separado entre primera y segunda iteración):** Se especifica cómo está diseñada la aplicación por dentro. Esto incluye un diagrama de CyC, un diagrama de despliegue y otro de módulos, así como la distribución por capas y las decisiones arquitecturales más importantes que han sido tomadas. También para la segunda iteración se han añadido dos diagramas de secuencia.
- **Estado actual de la aplicación (separado entre primera y segunda iteración)::** Se enumeran las distintas funcionalidades que tiene en estos momentos el producto.
- **Test y Calidad del Producto (separado entre primera y segunda iteración)::** Se describen los procesos de pruebas realizados al producto, así como las

herramientas utilizadas para ello. También se proporcionan métricas de calidad y los distintos pasos a seguir para comprobar su funcionamiento.

El documento consta de un tercer punto donde se trata el proceso de creación del producto (herramientas utilizadas, sistema de versiones, distribución del tiempo y esfuerzos). Consta de los siguientes puntos:

- **Estrategia de control de versiones:** Se describe el sistema de versiones y los repositorios utilizados.
- **Esfuerzos por persona y por actividades:** Cómo se han almacenado los esfuerzos de cada persona y cada cuánto tiempo.
- **Reparto del trabajo en el tiempo:** Cómo se ha repartido el tiempo entre los distintos integrantes del grupo.
- **Estrategia de mejora de procesos:** Describe cómo se podría haber mejorado el proceso de creación del producto para utilizarlo en distintos lanzamientos o proyectos.
- **Herramientas utilizadas:** Las herramientas utilizadas durante la creación del producto, así como la justificación de su uso.

Finalmente, en el último apartado se realiza un resumen del documento, así como el grado de cumplimiento de los distintos requisitos.

2. PRODUCTO

2.1 Planificación de lanzamientos

Se han planificado dos lanzamientos, los cuales representan dos iteraciones. La primera iteración acaba el 15 de abril y la segunda iteración el 27 de mayo. Los objetivos a cumplir por cada iteración son los siguientes:

Primera Iteración:

- El código y la documentación del proyecto se alojan en GitHub. Se trabaja de forma habitual contra Git.
- Compilación y gestión de dependencias están basadas en scripts.
- Se llevará un control de esfuerzos con las horas dedicadas por persona. Se entregará un resumen cada dos semanas.
- La aplicación cumple adecuadamente con sus requisitos.
- La documentación arquitectural es la adecuada al momento del proyecto, refleja fielmente el sistema e incluye al menos tres vistas: módulos, componentes-y-conectores, y despliegue del sistema.
- La arquitectura del sistema es por capas.
- Se usan adecuadamente estos conceptos de Diseño Dirigido por el Dominio: entidades, objetos valor, agregados, factorías y repositorios.
- Se ha puesto en marcha y se usa un servicio de mapas tipo WMS con los edificios disponibles del campus Río Ebro. Los mapas de este servicio se superponen en el cliente sobre otro servicio externo (p.ej. Open Street Map) que proporcione un mapa de la zona.
- Cobertura de tests automáticos de al menos el 25% del código (unitarios y/o de integración).
- La documentación arquitectural incluye una discusión adecuada sobre razones arquitecturales.

Segunda Iteración:

- Se cumplen todos los objetivos de la primera iteración.
- El modelo de dominio utiliza adecuadamente estos conceptos de diseño (dirigido por el dominio): servicios, paquetes, interfaces reveladoras, aserciones, funciones libres de efectos secundarios.
- El estilo cartográfico de los edificios en el servicio de tipo WMS refleja el tipo de uso de cada espacio (por ejemplo, los laboratorios de un color, los despachos de otro, etcétera).
- El modelo de dominio incluye alguna restricción o especificación correctamente implementada, y ésta se utiliza en alguna funcionalidad de la aplicación.

- La arquitectura del sistema es hexagonal.
- El servicio de mapas WMS se ha teselado, y se usa así desde el cliente.

2.2 Análisis de riesgos

Los riesgos del proyecto, principalmente tecnológicos, son los siguientes:

- Poco conocimiento sobre geografía, sistemas WMS etc.
- Implementación del cliente con Ionic y AngularJS, con poco conocimiento sobre la misma.

Como estrategias de mitigación, se propone reducir la funcionalidad implementada en la primera iteración para poder investigar estas tecnologías; en la segunda iteración se añadirá la funcionalidad dejada de lado en la primera iteración.

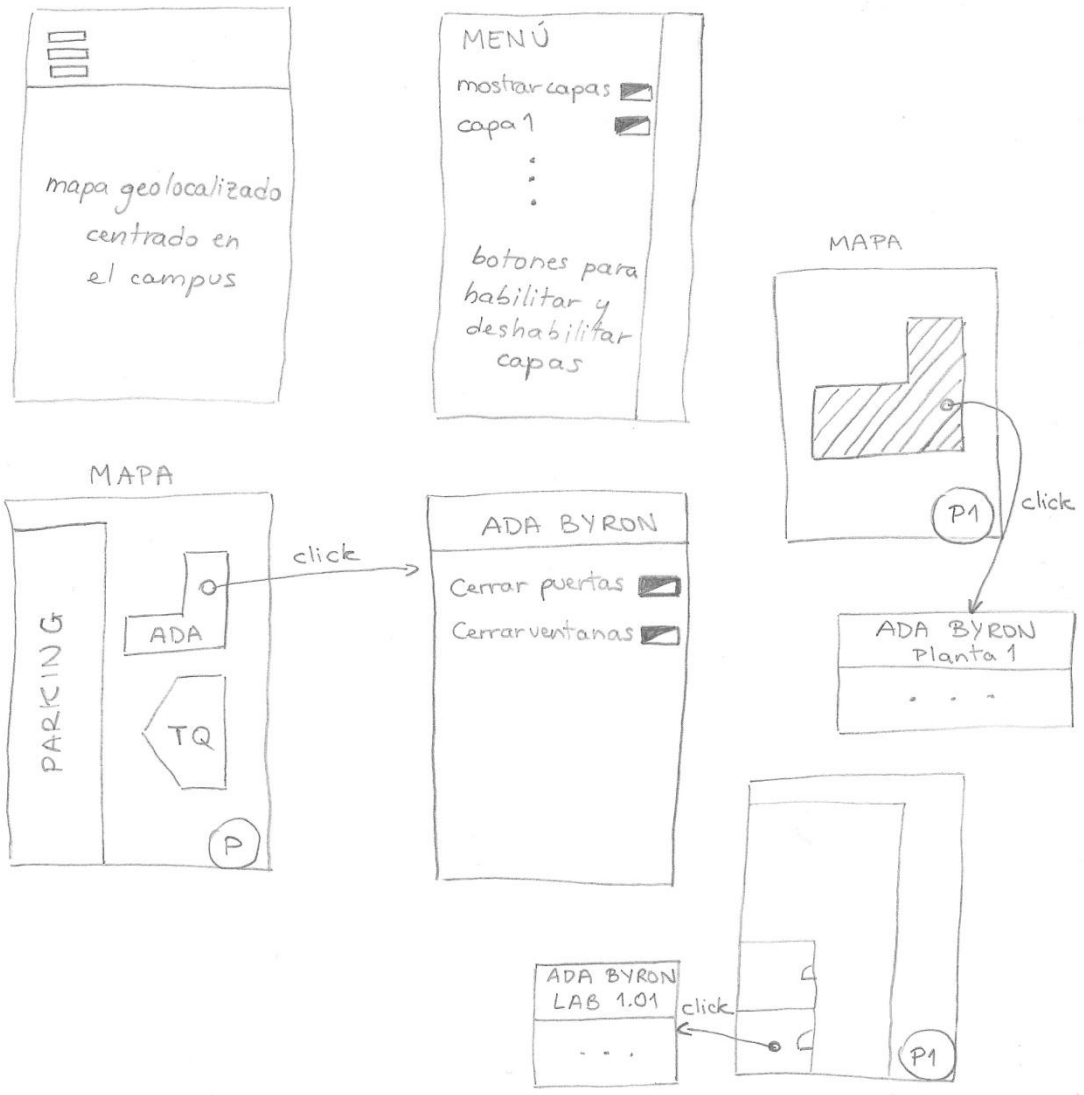
2.3 Requisitos

Los requisitos establecidos fueron:

- Mostrar un mapa global.
- Crear un mapa de los edificios.
- Interactuar con el mapa de edificios.
- Localizar puntos y espacios de interés en el mapa de edificios.
- Simular sensores (luz, temperatura, presencia…).
- Mostrar las capas de información.
- Interactuar con dichas capas.
- Funcionalidades de los espacios y edificios (cerrar puertas, ventanas…).

2.4 Interfaz de Usuario

En un primer intento, un boceto de la interfaz de usuario fue la siguiente:



A pesar de no estar completa, sirvió como base para el desarrollo. En ella se puede ver el menú principal, los distintos menús laterales y la navegación entre la interfaz del mapa y los distintos elementos que hay disponibles.

Actualmente la interfaz puede representarse a través de este mapa de navegación:

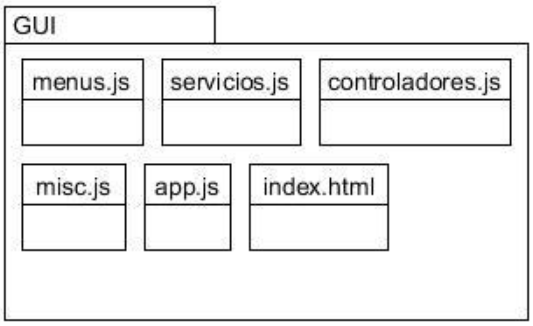
Hay 5 pantallas disponibles, un menú principal donde elegir el campus, un botón flotante para elegir el piso, y dos botones para acceder a los menús en los que realizar acciones o mostrar diferentes capas sobre el mapa.

2.5 Arquitectura

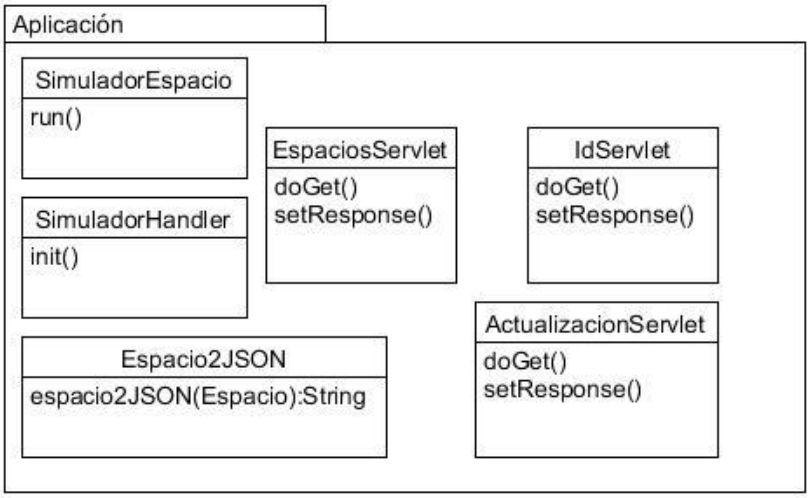
Primera Iteración:

La arquitectura del sistema para la primera iteración ha sido de cuatro capas: Interfaz, Aplicación, Dominio e Infraestructura. A continuación, se adjuntan los diagramas con los módulos de cada capa.

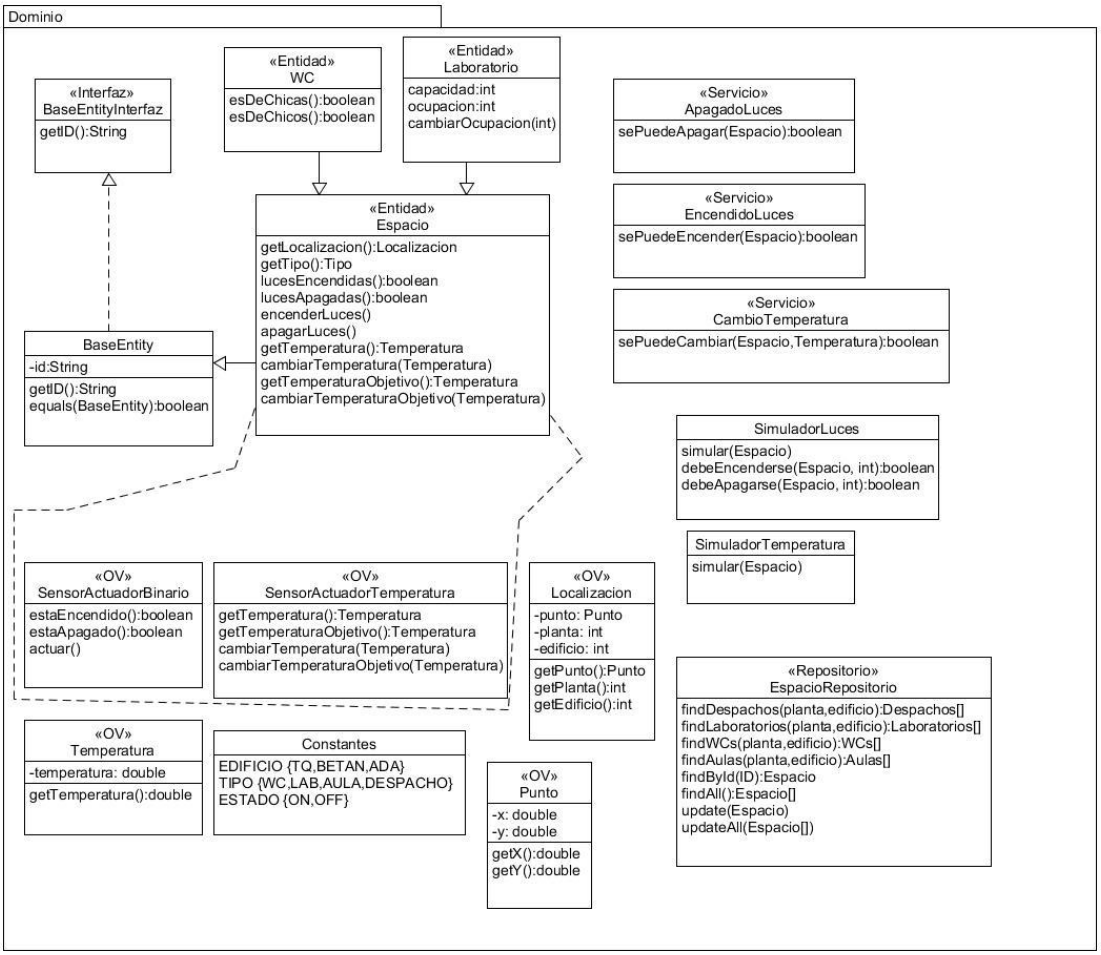
Interfaz de Usuario:



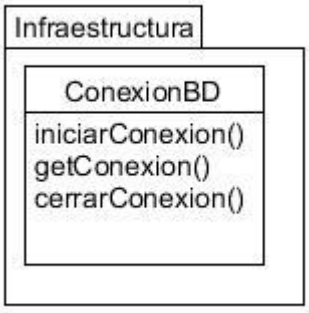
Aplicación:



Dominio:



Infraestructura:

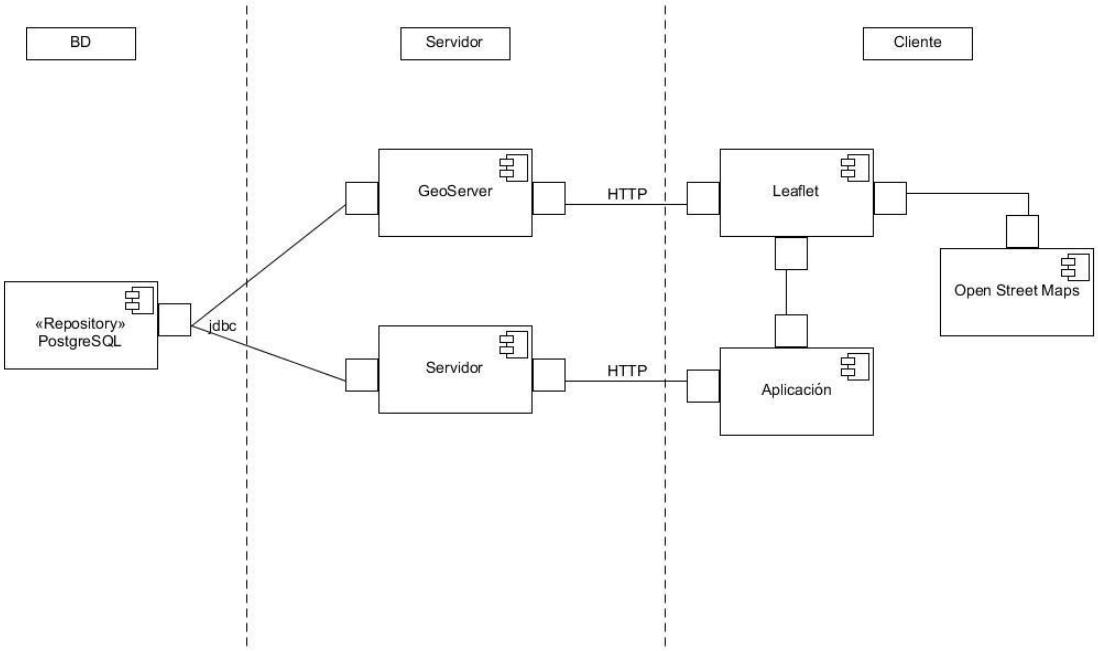


Se ha decidido no usar ninguna factoría ya que no había ningún objeto complicado sobre el cual usarla. Casi todas las operaciones están relacionadas con Espacio, por lo que sería el único objeto al que se le podría asignar una factoría. La implementación del repositorio se ha realizado en la capa de infraestructura, ya que depende de dicha tecnología, en este caso sobre una BD PostgreSQL y así facilita la transformación a una arquitectura hexagonal para la siguiente iteración.

Se ha creado un agregado de Espacio, con los sensores que debe tener cada espacio, y un repositorio de espacios para poder obtener los distintos espacios disponibles y sus propiedades (si está ocupado, con las luces encendidas, su temperatura…).

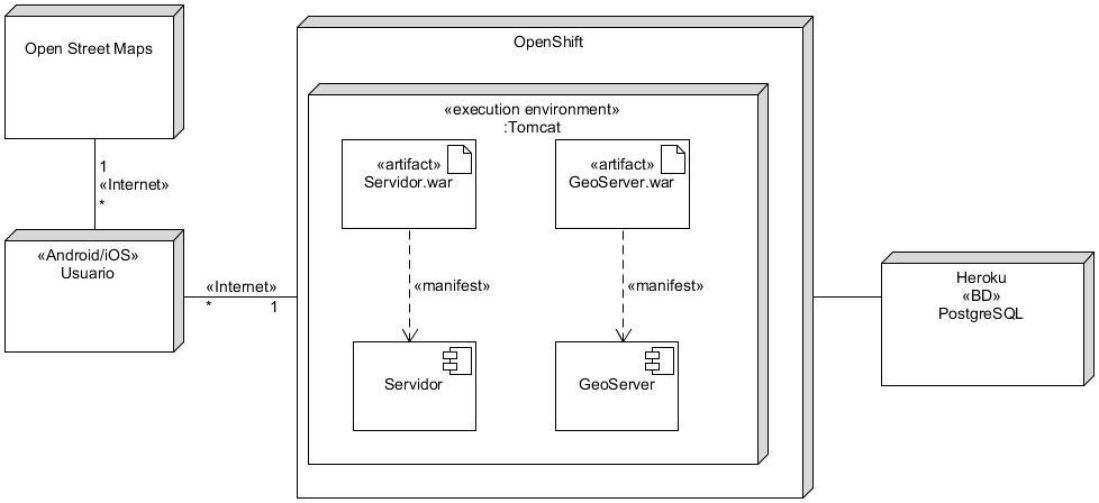
Para el paso de mensajes, finalmente se ha optado por realizarlo mediante Servlets con un método GET o POST. Se intentó dicha implementación con Spring, pero hubo problemas al usar Maven en vez de Gradle, así como con el servicio WMS Geoserver y su despliegue en la nube, ya que el conjunto superaba el espacio de 1GB proporcionado por Openshift, por lo que hubo que desecharlo.

CyC:



Los componentes realizados por nosotros son: El componente de Aplicación, el cual hará de Interfaz de Usuario y el Servidor, que se comunicará con la aplicación. La aplicación mostrará la información recibida por GeoServer y la pintará de distintos colores en el caso que fuera necesario gracias al componente Leaflet. Cuando se necesiten saber características específicas de un espacio, se obtendrá dicha información mediante el servidor, el cual almacena la información en una base de datos PostgreSQL al igual queGeoserver.

Despliegue:



Como se puede apreciar en el diagrama de despliegue, la aplicación cliente estará incluida en el sistema Android o IOS del usuario. Ésta se puede conectar a un sistema externo de mapas, en este caso Open Street Maps para pintar el mapa o conectarse al servidor o Geoserver, los cuales están desplegados en Openshift. La BD está alojada en Heroku debido a problemas con la propia BD de Openshift.

API del servidor:

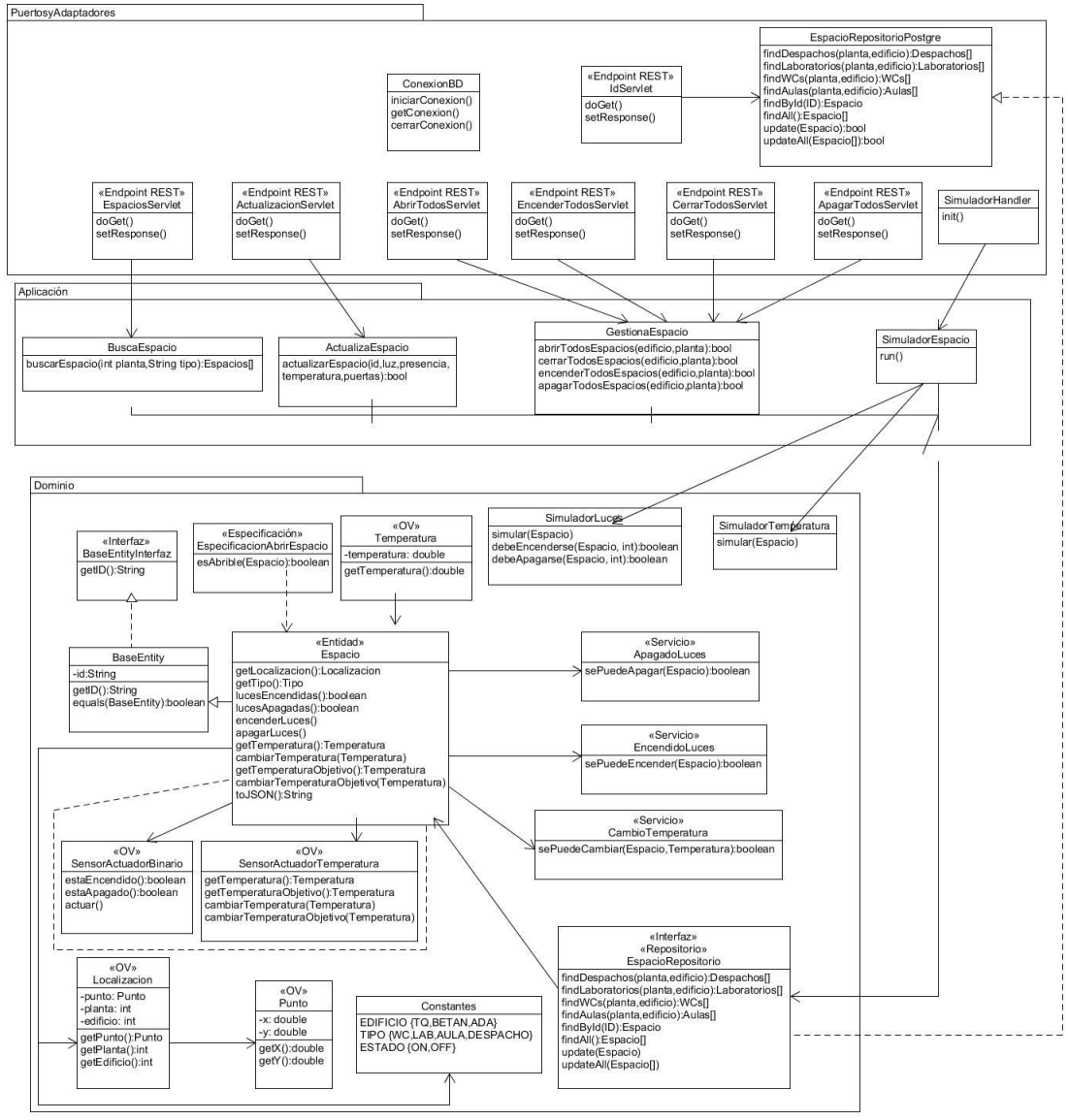
Los métodos que proporciona el servidor se suministran mediante el uso de la tecnología de los Servlets, como se ha mencionado anteriormente. Los métodos implementados en el proyecto son los siguientes:

- **/api/espacios.** Esta petición devuelve todos los espacios del tipo y de la planta especificada en los parámetros de la URI de la petición. En caso de no concretar el tipo de espacio que se desea que sea devuelto en los parámetros de la petición, se devolverán todos.
- **/api/actualizacion.** Esta petición permite actualizar o modificar ciertos datos que componen un espacio, como son la luz o la temperatura. Además, devuelve los datos ya actualizados.
- **/api/identificacion.** Esta petición devuelve un espacio específico, identificado por el id del espacio que se desea obtener.

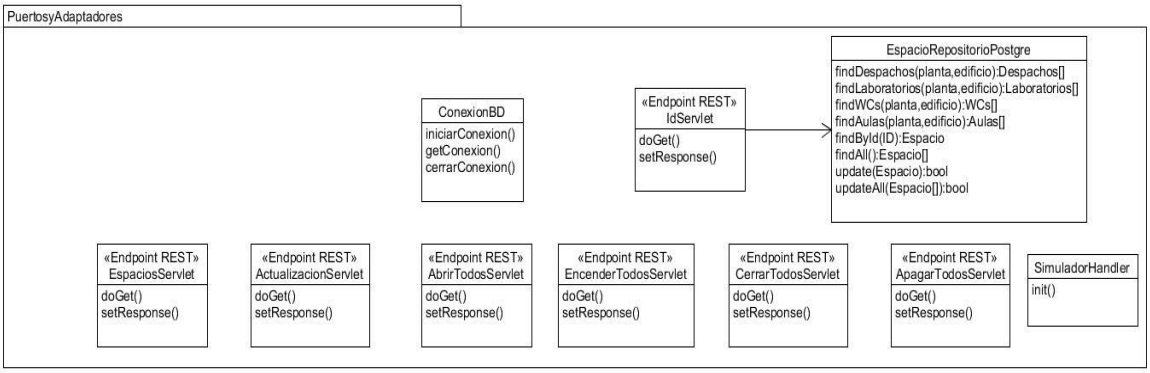
Segunda Iteración:

La arquitectura del sistema para la segunda iteración ha sido hexagonal: Puertos y adaptadores, Aplicación, y Dominio. A continuación, se adjuntan los diagramas con los módulos de cada capa.

Módulos:

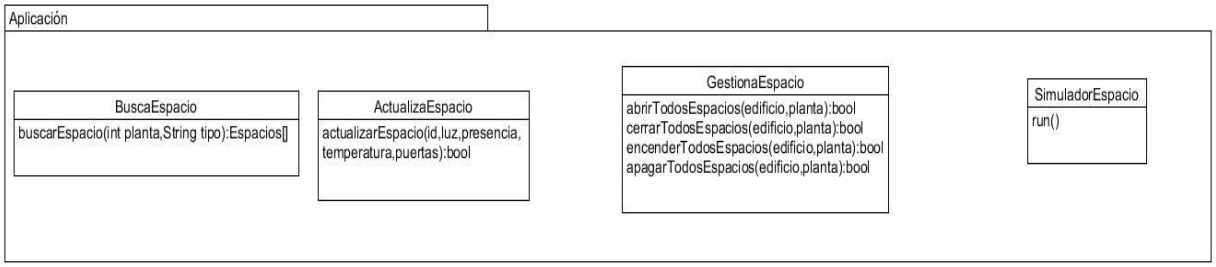


Puertos y adaptadores:



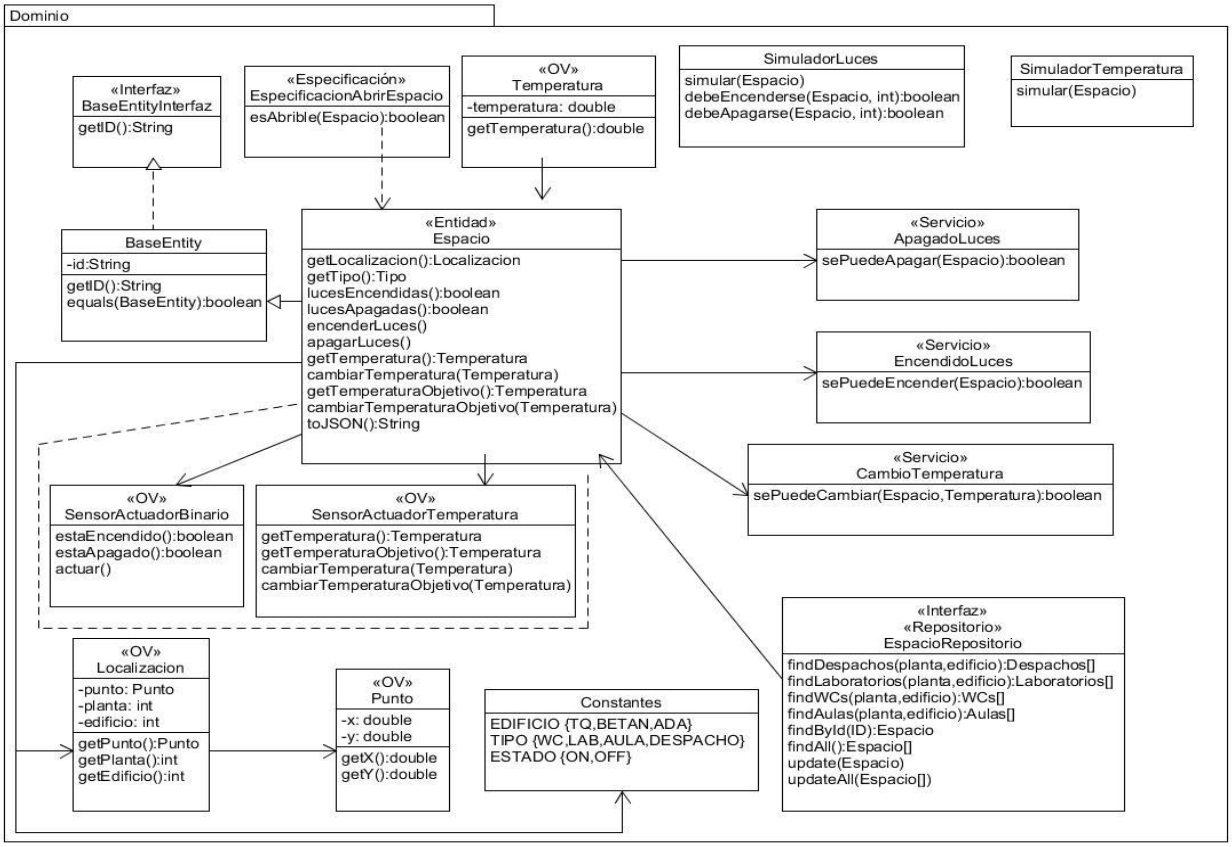
Esta capa consta de los distintos puntos de acceso a la aplicación, así como de comunicación con otros componentes. La mayoría de las clases son endpoints REST para comunicarse con el cliente, así como una clase para conectarse a la BD PostgreSQL. También se ha procedido a la inversión de dependencias para abstraer al repositorio del tipo de BD usada así como permitir a la clase IdServlet ejecutar directamente la operación findById() de forma sencilla.

Aplicación:



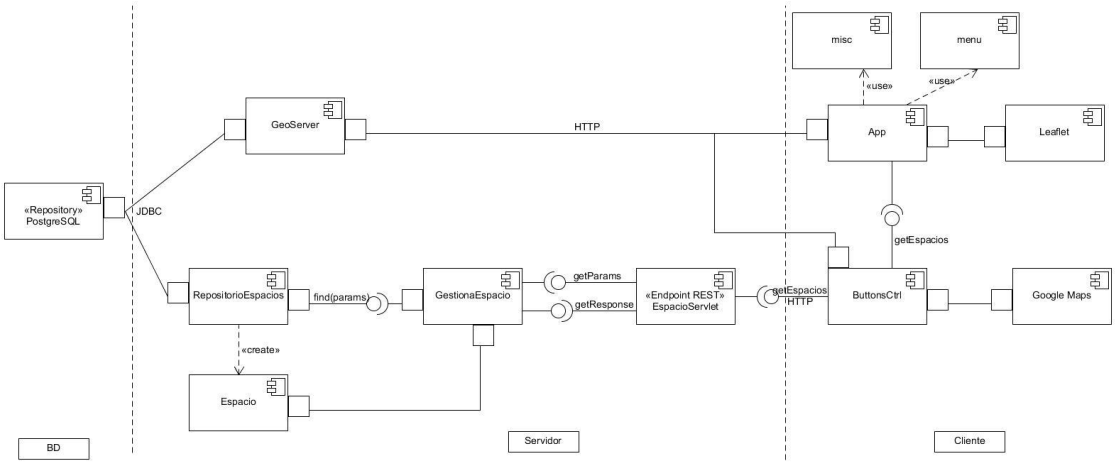
La capa de aplicación tiene las distintas clases para comunicar la capa de puertos y adaptadores con la capa de dominio. Es la encargada de ejecutar las operaciones solicitadas a la capa de puertos y adaptadores.

Dominio:



La capa de dominio se mantiene similar a dicha capa en la primera iteración. Se ha añadido una especificación la cual se encarga de permitir abrir un espacio o no, tomando variables como la presencia de dicho espacio, temperatura, hora actual etc.

CyC:



Los componentes PostgreSQL, GeoServer, Leaflet y Google Maps son ajenos a nosotros por lo que no tenemos información sobre ellos.

En la parte del servidor:

- El componente EspacioServlet es un Endpoint REST que se encarga de recibir la petición del cliente sobre un espacio y devolverle un JSON con la información correspondiente. Hay más componentes con la misma funcionalidad que son Endpoints REST pero se han obviado para simplificar el diagrama. Éstos Endpoints en su interfaz proporcionan una operación GET accesible mediante HTTP cuyos parámetros son pasados en la URL si son necesarios.

- El componente GestionaEspacio, de la capa de aplicación, se encarga de relacionar el dominio con los puertos y adaptadores. Ejecuta la operación pedida por el Endpoint. Llama al repositorio de espacios para obtener el espacio requerido y se lo devuelve al Servlet para que lo transforme el JSON y lo pase al cliente.

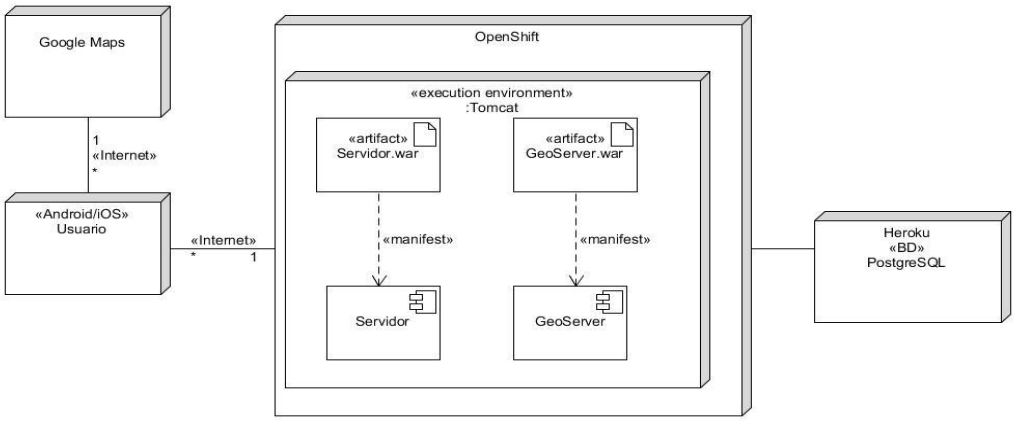
- El componente RepositorioEspacios se encarga de hacer las consultas correspondientes a la BD y con los datos obtenidos transformarlos en un Espacio manejable por la aplicación.

En la parte del cliente:

- El componente App se encarga de mostrar la interfaz de usuario usando las plantillas proporcionadas por “misc” y “menú” así como comunicarse con Leaflet y GeoServer para pintar los mapas. También de reaccionar al usuario, de forma que cuando pide información sobre un espacio se comunica con el componente ButtonCtrl.

- El componente ButtonCtrl se encarga de acceder a los datos del servidor mediante peticiones REST y una vez obtenidos estos datos pasárselos al componente App para mostrárselos al usuario.

Despliegue:



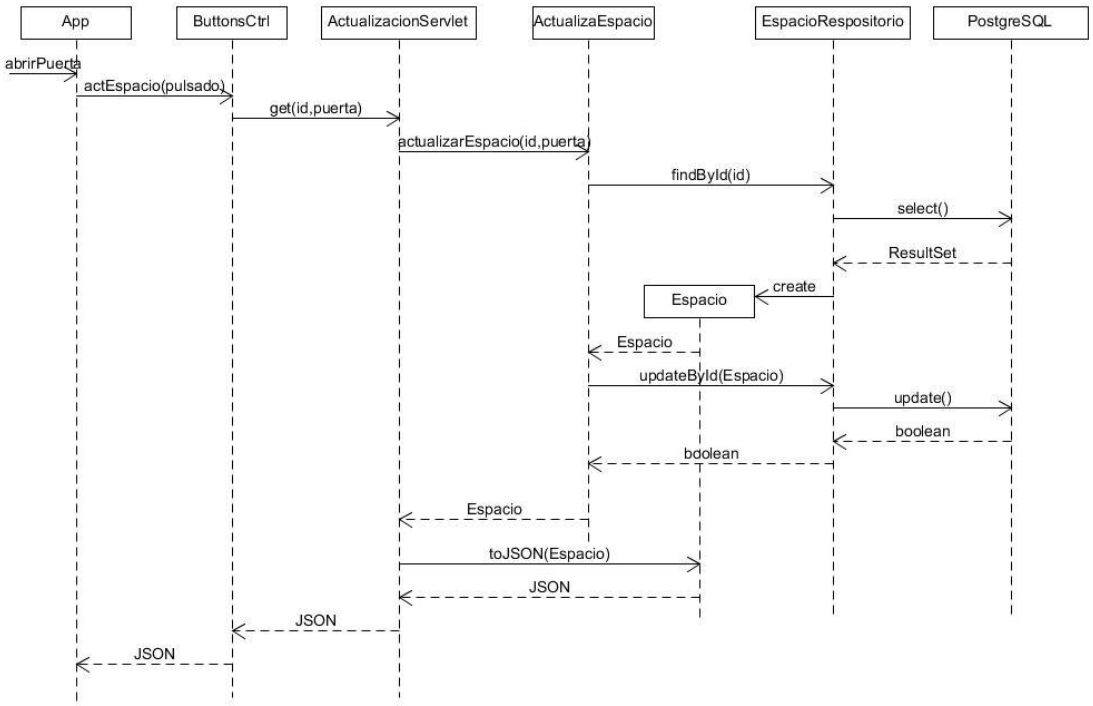
Se ha mantenido el mismo despliegue que en la primera iteración sustituyendo Open Street Layers por Google Maps debido a problemas con el sistema de coordenadas.

API del servidor:

Las funcionalidades que presenta el servidor, por medio de la utilización de la tecnología Servlet, son las siguientes:

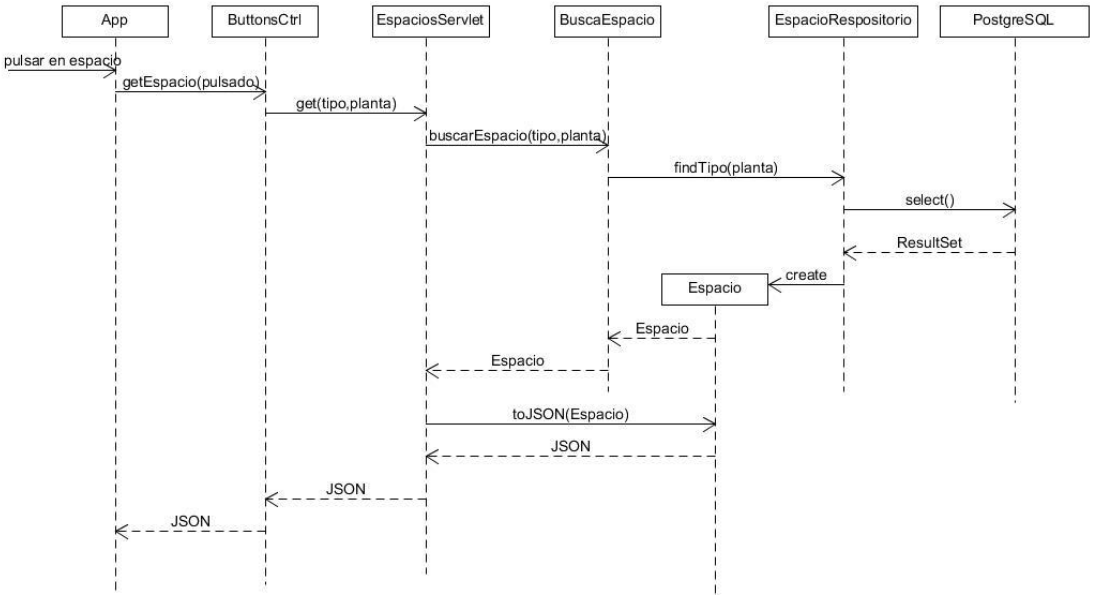
- **/api/espacios.** Esta petición permite obtener los espacios que concuerden con el tipo y la planta especificada en la URI de la petición enviada. En caso de no señalar el tipo de espacio, se devolverán todos los encontrados que se correspondan a la planta determinada. Además, si no se solicita ninguna planta en concreto, se devolverán directamente todos los espacios que se hallen en la base de datos.
- **/api/actualizacion.** Esta petición permite actualizar o modificar los datos que componen un espacio, como son la luz, las puertas, la presencia, la temperatura o la temperatura objetivo.
- **/api/identificacion.** Esta petición devuelve un espacio especifico, identificado por el id determinado en los parámetros de la petición.
- **/api/encender.** Esta petición permite encender las luces de todos los espacios que se encuentran en la base de datos.
- **/api/apagar.** Esta petición permite apagar las luces de todos los espacios que hallan en la base de datos.
- **/api/abrir.** Esta petición permite abrir las puertas de todos los espacios que se encuentran en la base de datos.
- **/api/cerrar.** Esta petición permite cerrar las puertas de todos los espacios se hallen en la base de datos.

Diagrama de secuencia Abrir Espacio:



Se representa la interacción desde que el usuario pulsa el botón de abrir puerta en un espacio en el cliente hasta que le llega a dicho cliente el JSON adecuado modificarlo.

Diagrama de secuencia obtener espacio:



Se representa la interacción desde que el usuario pulsa en un espacio en el cliente hasta que le llega a dicho cliente el JSON adecuado para mostrar sus atributos.

Rationale Arquitectural:

Se ha decidido eliminar las clases WC y Laboratorios respecto a la primera iteración debido a que la poca información que contenían no se veía reflejada en ninguna funcionalidad ni proporcionaba ninguna ventaja.

Se ha decidido comprobar que ciertos atributos no son nulos, mediante condicionales y aserciones, teniendo una precondición que ya lo indica, para evitar cambios incorrectos en los espacios.

Se ha decidido implementar el método que transforma un espacio a formato JSON en el propio espacio en vez de en la capa de aplicación ya que forma un JSON estándar con todos los atributos del espacio y así se evita llamar a dicho método de la capa de aplicación cada vez que se devuelve un espacio a una aplicación externa.

Se ha decidido mantener el Simulador dentro del mismo hexágono ya que usaba espacios y objetos valor como la temperatura que son propios del dominio. A pesar de esto, también se ha dividido, de forma que la capa de puertos y adaptadores tiene una clase para acceder a él, aunque actualmente se lance al iniciar el servidor, y la capa de aplicación otra para acceder al dominio.

Se ha decidido que el tipo de espacio se pase como parámetro en los endpoints REST en vez de tener distintos endpoints, como por ejemplo /laboratorios, /aulas etc ya que sus funcionalidades son muy similares y se ahorran clases y repetición de código innecesario.

2.6 Estado actual de la aplicación

Primera Iteración:

Actualmente la aplicación:

- Usa un sistema WMS.
- Pinta el mapa del Campus Río Ebro así como su entorno.
- Se puede ver el contenido de cada edificio y planta.
- Están disponibles todos los espacios del edificio Ada Byron.
- Se puede filtrar mediante capas de información.
- Se pinta de distinto color los espacios ocupados basándose en un estilo disponible en Geoserver.

Segunda Iteración:

Actualmente la aplicación:

- Usa un sistema WMS.
- Pinta el mapa del Campus Río Ebro así como su entorno.
- Se puede ver el contenido de cada edificio y planta.
- Están disponibles todos los espacios de los edificios Ada Byron, Torres Quevedo y Betancourt.
- Se puede filtrar mediante capas de información.
- Se pinta de distinto color los espacios ocupados basándose en un estilo disponible en Geoserver.
- Se puede obtener información de los espacios en los tres edificios del campus Río Ebro: temperatura de la habitación, temperatura del climatizador de la habitación, estado de las ventanas de la habitación, estado de las puertas de la habitación, ocupación de la habitación.
- Se puede modificar el estado de un espacio: cambiar la temperatura del climatizador, abrir o cerrar las ventanas, y abrir o cerrar las puertas.
- Se simula periódicamente la vida en el campus, modificando la información de los espacios según los criterios de simulación.
- El mapa está teselado para un acceso más rápido.

2.7 Test y Calidad de Producto

Primera Iteración:

En la parte front-end, se han creado test end-2-end, es decir, test que trabajen sobre la interfaz de la aplicación, ya que la única manera de interactuar con el sistema será a través de la interfaz de usuario que ofrece la aplicación.

Para la realización de los test y análisis de la cobertura se ha usado una combinación de Jasmine, Protractor, Istanbul, Gulp y Npm.

Jasmine y Protactor (http://angular.github.io/protractor/#/) son los encargados de realizar los test automáticos disponibles en la carpeta “tests” del proyecto. Éstos crean una instancia de la aplicación en un navegador (en este caso Google Chrome) e interactúan a través de los elementos disponibles en el documento HTML. Protractor trabaja con el framework de Jasmine.

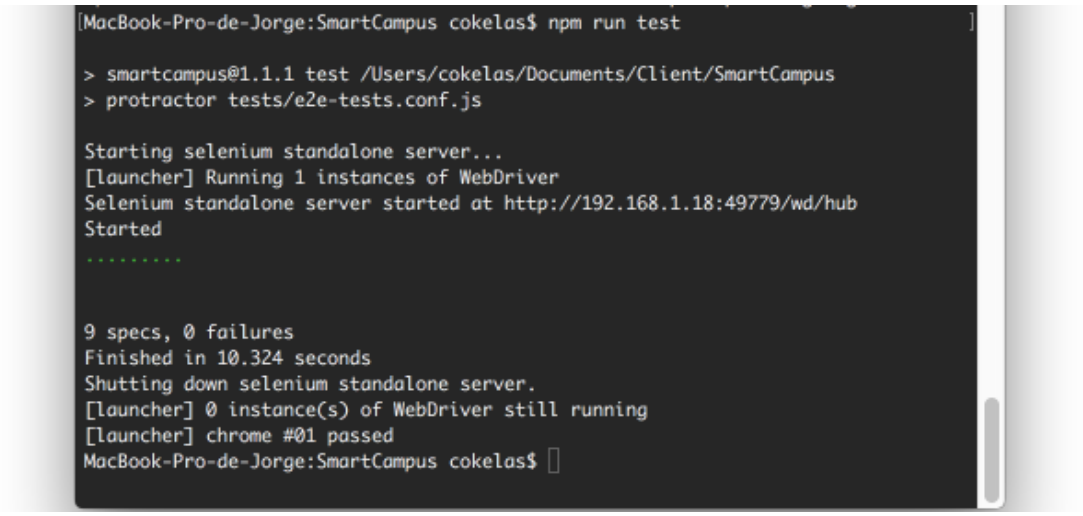
Para la obtención de la cobertura se ha usado Istanbul (para la instrumentación y análisis de código ejecutado).

Para automatizar todas las tareas necesarias para el lanzamiento de test y análisis de cobertura se ha usado Gulp, junto a Npm, de esta forma se consigue resumir en dos simples comandos:

```
$ npm run test
```

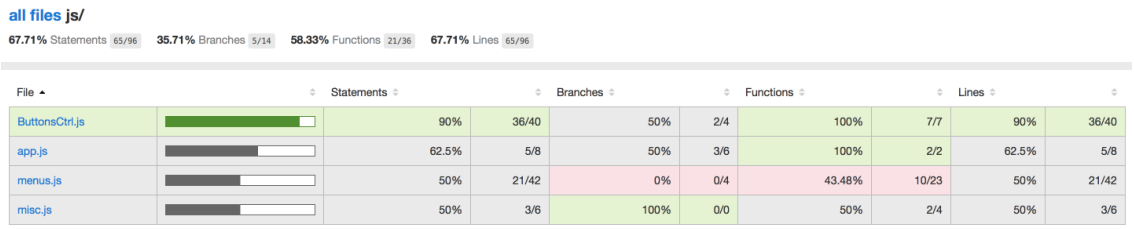
Esto ejecutará el comando con dicho nombre que hay dentro del archivo package.json. En este caso ejecutará "protractor tests/e2e-tests.conf.js" , esto lanzará la aplicación en Google Chrome, y ejecutará todos los tests que hay dentro de la carpeta tests/e2e-tests y nos indicará si hay fallos o no.

La salida nos mostrará qué test han pasado y cuáles dan error:



```
$ npm run testcover
```

Este script generará el report con la cobertura de código en el archivo coverage/integration/index.html. El report tendrá el siguiente formato:



all files js/

67.71% Statements 65/96 35.71% Branches 5/14 58.33% Functions 21/36 67.71% Lines 65/96

File	Statements		Branches		Functions		Lines	
ButtonsCtrl.js	90%	36/40	50%	2/4	100%	7/7	90%	36/40
app.js	62.5%	5/8	50%	3/6	100%	2/2	62.5%	5/8
menus.js	50%	21/42	0%	0/4	43.48%	10/23	50%	21/42
misc.js	50%	3/6	100%	0/0	50%	2/4	50%	3/6

En él, se pueden ver tanto las líneas ejecutadas de cada fichero JavaScript, como los métodos, de manera global o de cada fichero individualmente. Como vemos, la cobertura total por parte del cliente es de 67,71%.

Se ha intentado automatizar esta tarea lo máximo posible. Esta es la causa de que sean necesarios todos estos componentes diferentes para la ejecución de los test.

Los test unitarios en el servidor se han realizado con JUnit apoyándose en Mockito para simular las llamadas al sistema en algunos tests. La cobertura de código se ha realizado mediante el plugin para Eclipse EclEmma, dando como resultado la siguiente tabla:

Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
smartcampus	47,2 %	1.159	1.297	2.456
src/main/java	44,2 %	1.012	1.277	2.289
dominio	33,5 %	273	542	815
aplicacion	36,7 %	248	427	675
infraestructura	61,5 %	491	308	799

La compilación y gestión de dependencias se ha gestionado mediante Maven, de forma que no hace falta importar librerías manualmente y mediante Openshift, cada vez que se subía un cambio a GitHub, el servidor se compilaba y desplegaba en Openshift automáticamente.

La cobertura total por parte del servidor es de 47,2%.

Segunda Iteración:

Se ha mantenido el proceso de la primera iteración. La cobertura total del servidor ha aumentado al 55,3%, como se ve en la imagen siguiente.

Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
src/main/java	55,3 %	1.610	1.301	2.911
dominio	39,6 %	392	598	990
puertosyadaptadores	70,7 %	1.100	455	1.555
aplicacion	32,2 %	118	248	366

Los test unitarios comprueban la correcta ejecución de las principales funcionalidades de la API del servidor, así como los métodos que acceden a la BD.

3. PROCESO

Se ha seguido el mismo proceso para ambas iteraciones, cambiando el uso de Open Street Maps por Google Maps.

3.1 Estrategia de control de versiones

Para la gestión de control de versiones se ha utilizado GitHub tal y como se especificaba en el documento. Ha sido necesario crear tres repositorios con sus respectivas wikis y documentos: uno para la implementación del cliente (repositorio Client), otro para la implementación del servidor (repositorio Server) y un último repositorio para la documentación (Documentacion-LabIS). Se ha usado un workflow centralizado en todos ellos, de forma que los cambios de cualquier usuario se suben a *master* y no es necesario el uso de ramas, lo cual ha creado algún problema de sincronización.

3.2 Esfuerzos por persona y por actividades

Los esfuerzos han sido seguidos por persona y cada dos semanas, de forma que se almacenan en una hoja de Excel detallando la actividad realizada, las horas empleadas en la actividad y la fecha en que se realizó dicha actividad. La hoja Excel se adjunta junto con el documento.

3.3 Reparto del trabajo en el tiempo

El trabajo se ha repartido según las funcionalidades que quedaban por desarrollar, y viendo las dependencias entre ellas. El equipo se ha dividido en desarrolladores de front-end, desarrolladores de back-end y desarrolladores de base de datos. A su vez también ha habido encargados de configurar y desplegar la aplicación, así como crear sus conexiones.

3.4 Estrategia de mejora de procesos

Para futuros procesos, se mejorará la división del trabajo entre los integrantes ya que ciertos cambios se solapaban o había que esperar a que terminara una parte uno de los integrantes para empezar otra. De igual manera se intentará mejorar la comunicación entre los integrantes del grupo, con el fin de saber en todo momento el estado del proyecto, dado que ayuda a mejorar el desarrollo del proyecto.

3.5 Herramientas utilizadas

Las herramientas utilizadas para el desarrollo del producto han sido:

- Ionic para el desarrollo de la aplicación cliente.
- Eclipse como IDE de desarrollo para el servidor.
- GeoServer como servicio WMS.
- Leaflet como servicio para pintado de mapas.
- Google Maps para la obtención de mapas.
- Github para el control de versiones.
- Openshift para el despliegue online de la aplicación.
- Heroku para el despliegue de la Base de Datos PostgreSQL.
- Maven para las dependencias y compilación del servidor.
- JUnit para las pruebas del servidor.
- EclEmma para la cobertura del servidor.
- Protractor e Istanbul para las pruebas y cobertura del cliente.
- Gulp y Npm para la automatización de tareas en el cliente.

4. CONCLUSIONES

4.1 Resumen del documento

Primera Iteración:

El producto se va a realizar en dos iteraciones y este documento corresponde a los resultados de la primera. En esta primera iteración la aplicación muestra el mapa con los edificios del Campus Río Ebro y permite interactuar con ellos, de forma que se puede filtrar por capas (ver laboratorios, aulas, despachos…) así como pintarlos de diferente color dependiendo de su tipo, para una fácil diferenciación de estancias.

Para el desarrollo se ha seguido una arquitectura de 4 capas, divididas en: Interfaz, Aplicación, Dominio e Infraestructura. La capa en la que se ha puesto más énfasis, así como la más complicada, ha sido el dominio, usando entidades, objetos valor, agregados, repositorios y servicios.

Segunda Iteración:

El producto se va a realizar en dos iteraciones y este documento corresponde a los resultados de las dos iteraciones. Al final de la segunda iteración, la aplicación muestra el mapa con los edificios del Campus Río Ebro y permite interactuar con ellos, de forma que se puede filtrar por capas (ver laboratorios, aulas, despachos…) así como pintarlos de diferente color dependiendo de su tipo, para una fácil diferenciación de estancias. A su vez, permite la interacción con los espacios de los edificios, modificación de sus propiedades (temperatura, luz, puertas) así como poder ver su presencia. También se puede cerrar y abrir todos los espacios de un edificio y apagar o encender todos los espacios de un edificio.

Para el desarrollo se ha seguido una arquitectura hexagonal, divididas en: Aplicación, Dominio y Puertos-y-Adaptadores. La capa en la que se ha puesto más énfasis, así como la más complicada, ha sido el dominio, usando entidades, objetos valor, agregados, repositorios, servicios, interfaces reveladoras, especificaciones etc.

El siguiente proceso se ha realizado igual para ambas iteraciones.

Para su desarrollo se ha desplegado en OpenShift y se ha seguido una gestión de versiones basada en Git. Su funcionamiento se ha comprobado mediante la automatización de tests, code coverage y scripts para su compilación. La base de datos se encuentra alojada en un servidor de Heroku.

El cliente ha sido implementado en Ionic de forma que es compatible con Android e iOS y permitirá la comunicación con el servidor para los usuarios de dicha aplicación

(conserjes de campus y personal del mismo, principalmente). La comunicación será de tipo REST.

Así mismo se ha seguido el esfuerzo de cada persona del equipo y se ha verificado el grado de cumplimiento de los objetivos para la primera iteración.

4.2 Grado de cumplimiento de los objetivos

Primera Iteración:

En esta primera iteración se han superado todos los objetivos para sacar un notable:

- El código se aloja completamente en GitHub, se trabaja contra él de forma continuada y contiene su respectiva carpeta con la documentación. https://github.com/UNIZAR-30249-2016-UniDev

- La compilación y gestión de dependencias se hace de forma automática por parte del cliente a través de la interfaz de comandos que ofrece Ionic (, junto a Npm para la gestión de dependencias. El servidor se gestiona mediante Maven y se compila y despliega automáticamente en Openshift.

- Se lleva un control de horas dedicadas por persona a través del documento Excel y se entrega cada fecha indicada.

- La aplicación cumple con todos los requisitos hasta el momento de la entrega.

- Se dispone de tres vistas de la arquitectura: módulos, componentes-y-conectores, y despliegue del sistema. Disponibles en el apartado 2.5.

- La arquitectura del sistema está por capas. Detallado en el apartado 2.5.

- Se han trabajado componentes del dominio del problema: entidades, objetos valor, agregados, factorías y repositorios. Detallado en el apartado 2.5.

- Se ha puesto en marcha un servicio de WMS (de momento de manera local), en el que se superponen mapas sobre un servicio externo de Openstreetmap.

- La cobertura de test automáticos por parte del cliente es de un 67,71%, y de un 47,2% por parte del servidor, tal y como se muestra en el apartado 2.7.

- La documentación arquitectural dispone de un rationale. Apartado 2.5.

Segunda Iteración:

En esta primera iteración se han superado todos los objetivos para sacar un sobresaliente exceptuando la operación de análisis SIG:

- El código se aloja completamente en GitHub, se trabaja contra él de forma continuada y contiene su respectiva carpeta con la documentación. https://github.com/UNIZAR-30249-2016-UniDev

- La compilación y gestión de dependencias se hace de forma automática por parte del cliente a través de la interfaz de comandos que ofrece Ionic (, junto a Npm para la gestión de dependencias. El servidor se gestiona mediante Maven y se compila y despliega automáticamente en Openshift.

- Se lleva un control de horas dedicadas por persona a través del documento Excel y se entrega cada fecha indicada.

- La aplicación cumple con todos los requisitos hasta el momento de la entrega.

- Se dispone de tres vistas de la arquitectura: módulos, componentes-y-conectores, y despliegue del sistema. Disponibles en el apartado 2.5.

- La arquitectura del sistema es hexagonal. Detallado en el apartado 2.5.

- Se han trabajado componentes del dominio del problema: entidades, objetos valor, agregados, factorías y repositorios. Detallado en el apartado 2.5.

- Se ha puesto en marcha un servicio de WMS (de momento de manera local), en el que se superponen mapas sobre un servicio externo de Google Maps.

- La cobertura de test automáticos por parte del cliente es de un 67,71%, y de un 55,3% por parte del servidor, tal y como se muestra en el apartado 2.7.

- La documentación arquitectural dispone de un rationale. Apartado 2.5.

- El modelo de dominio utiliza adecuadamente estos conceptos de diseño (dirigido por el dominio): servicios, paquetes, interfaces reveladoras, aserciones, funciones libres de efectos secundarios. Apartado 2.5.

- El estilo cartográfico de los edificios en el servicio de tipo WMS refleja el tipo de uso de cada espacio (por ejemplo, los laboratorios de un color, los despachos de otro etc.).

- El modelo de dominio incluye alguna restricción o especificación correctamente implementada, y ésta se utiliza en alguna funcionalidad de la aplicación. Apartado 2.5.

- El servicio de mapas WMS se ha teselado, y se usa así desde el cliente.