Universidad de Alcalá
Escuela Politécnica Superior

GRADO EN INGENIERÍA INFORMÁTICA

Trabajo Fin de Grado

Sistema de control del desabastecimiento de fármacos de alto coste y de riesgo de abuso en las farmacias hospitalarias.

Autor: Cassandra Moreno López

Tutor/es: Julia Clemente Párraga,
CoTutor: Luis Martín Lázaro

2020

UNIVERSIDAD DE ALCALÁ
Escuela Politécnica Superior

GRADO EN INGENIERÍA INFORMÁTICA

Trabajo Fin de Grado

Sistema de control del desabastecimiento de fármacos de alto coste y de riesgo de abuso en las farmacias hospitalarias.

Autor: Cassandra Moreno López

Tutor/es: Julia Clemente Párraga,
CoTutor: Luis Martín Lázaro

TRIBUNAL:

Presidente: Óscar Rodríguez Polo

Vocal 1º: Pablo Parra Espada

2020

Agradecimientos

Después de un intenso período de diez meses, hoy es el día: escribo este apartado de agradecimientos para finalizar mi trabajo de fin de grado. Ha sido un período de aprendizaje intenso, no solo en el campo científico, pero también a nivel personal. Escribir este trabajo ha tenido un gran impacto en mí y es por eso por lo que me gustaría agradecer a todas aquellas personas que me han ayudado y apoyado durante este proceso.

Primero de todo me gustaría agradecérselo a mi hermano, por la ayuda prestada durante la carrera, y, sobre todo, en estos últimos meses. También agradecérselo a mis padres, por el apoyo mostrado durante estos 5 años.

Por último, agradecérselo a esas dos personas que han hecho que siguiese motivada con la carrera, cuando ya no podía más. Y que de una manera u otra han permitido que siguiese creciendo con ellas.

Muchas gracias a todos.

Cassandra López

Resumen

El sector farmacéutico en España es un sector altamente regulado y poco adaptado al mundo tecnológico. Es por ello, que no disponen de soluciones útiles y de aprovechamiento económico, adaptadas a las nuevas tecnologías. Con este proyecto, lo que se quiere es transformar el modelo de negocio de la farmacia hospitalaria creando una red farmacias que facilite el aprovechamiento de unidades sueltas de medicamentos de alto impacto económico, permitiendo la localización y reserva de estos en todo el ámbito nacional. El proyecto está desarrollado con Java y Spring MVC, centrándose en la simplicidad y la funcionalidad.

Abstract

The pharmaceutical sector in Spain is highly regulated and not very well adapted to the world of technology. For this reason, they do not have useful solutions and economic use, adapted to new technologies. With this project, the aim is to transform the business model of the hospital pharmacy by creating a network of pharmacies that facilitates the use of individual units of high economic impact medicines, allowing the location and booking of them throughout the country. The project is developed with Java and Spring MVC, focusing on simplicity and functionality.

Palabras Clave

Stock, encargo, farmacia, *bean*, *controller*

Índice General

Índice de Tablas

Índice de Figuras.

Parte I

Resumen Extendido.

El sector farmacéutico en España está altamente restringido y con normativas muy estrictas. Esta regulación provoca que el sector no pueda evolucionar adaptándose a las nuevas tecnologías y de como resultado problemas como el sobrecoste y el derroche de producto.

Actualmente, existe un gran problema relacionado con el desabastecimiento de medicamentos. Este, aunque ya conocido, en años anteriores, se ha incrementado sustancialmente en el último año. Según el *CIMA* (Centro de Información de Medicamentos) de la *Aemps* (Agencias Española del Medicamento y Productos Sanitarios) del Ministerio de Sanidad, las presentaciones con problemas de suministro aumentan a 1300 en el 2018. Lo que dibuja un escenario de un problema que lejos de reducirse, crece año tras año como revelan los datos de la *Aemps*: en 2015 se superó los 700, en 2016 casi se alcanzan los 800 y en 2017 fue con creces superior a 900. Al desabastecimiento de estos fármacos, hay que añadir una serie de medicamentos conocidos como medicamentos de alto impacto económico y medicamentos de riesgo de abuso, los cuales solo se encuentran disponibles en las farmacias hospitalarias. Los primeros, son medicamentos para un conjunto limitado de enfermedades que registran una baja prevalencia pero que demandan altos costes financieros. Los segundos, son medicamentos altamente regulados y de los cuales solo se pueden tener unas contadas unidades en los botiquines de los centros de salud o en las farmacias hospitalarias.

Cuando un paciente necesita un tratamiento con un medicamento que tiene la etiqueta de alto impacto económico, el proceso habitual para solicitarlo es o bien llamar a los hospitales cercanos, y de tenerlo, cumplimentar todo el papeleo para poder realizar el préstamo, o bien, llamar al laboratorio para que envíen más unidades.
La situación más común es la segunda, debido a la complejidad de la primera y las demoras en los traslados y pagos. Esta situación provoca que si un paciente, no ha terminado sus viales, el medicamento quede sin utilizar y perecer debido a su corta fecha de caducidad, aumentando los costes en los hospitales y en el sector público y provocando una explotación innecesaria de los recursos.

La solución a esto es crear una red de farmacias hospitalarias que puedan comunicarse entre sí, para facilitar el préstamo y distribución de los medicamentos. Se llevará a cabo con una herramienta capaz de mantener el albarán de las farmacias actualizado y permitiendo una búsqueda entre las farmacias de la red del producto deseado. Además, se realizará de manera semi automática la cumplimentación del papeleo necesario para poder realizar el préstamo. Así mismo, se valorará realizar un proceso de trazabilidad para hacer un seguimiento del estado del préstamo.

Esta solución es única, ya que no existe ninguna herramienta con las mismas características. Se podría hablar de la última solución adoptada por el Ministerio de Salud, Valtermed. Es un programa que permite disponer de información óptima a la hora de tomar decisiones en la gestión de fármacos de alto impacto económico y sanitario. Sin embargo, es una herramienta que realiza un estudio, no que ofrece una solución para reducir costes y aprovechamiento de recursos.

Para poder tener un control del stock, como primera aproximación, se realizará la inserción y modificación de este de manera manual por el usuario. Para ello, los sistemas de gestión farmacéutica disponen de una herramienta que permite exportar sus datos a una hoja de cálculo, pudiendo insertarlos en el sistema de control de una manera rápida y eficaz. Además de esa opción, existe la inserción manual e individual, buscando el medicamento en la base de datos del sistema e introduciendo el número de unidades deseadas. El sistema cuenta con una base de datos de los fármacos ya dados de alta por el Ministerio de Salud, para evitar problemas a la hora de registrar cada farmacia sus medicamentos y poder tener una estandarización de los nombres y códigos nacionales.

La búsqueda del fármaco se realizará por código nacional del medicamento. Con esta solución se pretende que los recursos sanitarios ya existentes sean empleados de una manera más eficiente, evitando su desperdicio y por tanto, reduciendo los costes de las farmacias hospitalarias y centro de salud. Además, se garantiza una mayor rapidez en el tratamiento del paciente, ya que el préstamo desde unos hospitales a otros puede realizarse en cuestión de unas horas gracias al transporte interno.

Debido a que es una solución real para un problema emergente, este trabajo se llevará acabo a nivel de proyecto piloto y se realizarán simulaciones ficticias. La subida a producción del sistema ha sido realizada con dos hospitales ficticios.

Como líneas de futuro se podría hablar de una red de farmacias hospitalarias con más de 700 farmacias conectadas. Esto ayudaría a que el aprovechamiento de los recursos aumentase considerablemente. En cuanto a la aplicación, sería recomendable realizar una pasarela entre los programas de gestión farmacéutica hospitalaria más comunes y el propio sistema de control. La idea final de este proyecto es crear una red neuronal de farmacias hospitalarias, capaz de responder en tiempo real, similar a la red de farmacias comunitarias LudaFarma.

Como conclusiones del proyecto, se puede decir que se ha desarrollado una aplicación acorde a los requerimientos del cliente y ampliando estos a nivel de seguridad y servicios auxiliares. A nivel teórico se han aprendido conceptos sobre herramientas que demandan las empresas como

son *Spring MVC*, *AWS*, *Spring Security*, *Jquery*, *JSP* y se ha obtenido una visión general de cómo debe plantearse un proyecto en el ámbito profesional, así como la forma de trabajo con metodologías ágiles.

Parte II

Memoria del Trabajo.

Capítulo 1.

Introducción

El proyecto relacionado con este documento consiste en el desarrollo de una aplicación web que permita gestionar el control de medicamentos de alto impacto económico y de riesgo de abuso entre hospitales utilizando la metodología Scrum.

En este capítulo se proporciona una vista general de los objetivos que se han llevado a cabo durante el desarrollo del proyecto, así como su estructura.

1.1 Contexto

Este proyecto pertenece al área de sistemas de la información y se tiene en cuenta la siguiente definición:

"Un sistema de información es un conjunto de elementos que interactúan entre sí con el fin de apoyar las actividades de una empresa o negocio." [1]

Particularizando podemos decir que el trabajo desarrollado es un sistema de control de stock interno, dentro del área de logística. Sin embargo, es una idea localizable en cualquier otro ámbito: encuentra tu producto de manera inmediata y lo más cerca posible. La informatización de los datos y su posterior uso en la red es, hoy en día, uno de los servicios más usados por otros sectores para el desarrollo de su negocio.

Debido a que es un proyecto donde el cliente debe estar involucrado en todo momento y debe ir aprobando cada uno de los servicios que se van a desarrollar de manera independiente, se ha optado por el empleo de metodologías ágiles durante el proceso.

A pesar de que existen varias metodologías ágiles, todas se rigen bajo cuatro pilares fundamentales:

- Los individuos y su interacción, por encima de los procesos y las herramientas.

- El software que funciona, frente a la documentación exhaustiva.

- La colaboración con el cliente, por encima de la negociación contractual.

- La respuesta al cambio, por encima del seguimiento de un plan.

Con esto se logra una mayor eficiencia del personal con un menor coste. Más adelante se explicará qué metodología se ha empleado y en qué principios se ha basado la elección.

1.2 Motivación

Es un hecho que las empresas requieren para la contratación de sus empleados personas que tengan conocimientos en las herramientas que se están utilizando en el escenario actual o bien que tengan conocimiento en herramientas que sigan el mismo tipo de estructura. Es por ello, que se ha querido aprender la estructura del desarrollo web con una herramienta que dispone de una gran documentación en internet como puede ser *Spring* y se ha combinado con el uso de herramientas punteras, pero con menor recorrido como como pueden ser *Amazon Web Services* o *MongoDatabase,* Así mismo, se ha tenido en cuenta la experiencia profesional dentro de los sistemas de la información del sector farmacéutico.

Por otra parte, el auge del uso de metodologías ágiles en el desarrollo de software gracias a las ventajas que proporcionan frente a metodologías clásicas es la causa de la elección de Scrum como referente en el enfoque de desarrollo del proyecto.

1.3 Estructura del Proyecto

Este Trabajo de Fin de Grado está estructurado en varias partes. La memoria está formada por los siguientes capítulos:

- Capítulo 1: contiene el resumen del proyecto en rasgos generales, así como el contexto y las motivaciones en las que se basa este trabajo.

- Capítulo 2: se detallan los objetivos, requisitos, restricciones y las herramientas de empleadas en el desarrollo del proyecto.

- Capítulo 3: se basa en el análisis y diseño del sistema, haciendo hincapié en la metodología Scrum, se describen las historias de usuario y tareas a realizar, los patrones de diseño empleados como son el patrón MVC así como la arquitectura Cliente-Servidor.

- Capítulo 4: es el desarrollo de la aplicación. En esta parte se detalla las etapas del proyecto, denominadas *Sprints*, y cómo se ha llevado a cabo la implementación de cada una de las historias de usuario.

- Capítulo 5: bibliografía que referencia todas las páginas visitadas para la realización de esta memoria, así como los documentos consultados para el desarrollo de la aplicación.

Capítulo 2.

Planteamiento del Problema y Supuestos de Trabajo.

Conocido el problema existente y con una vista general del proyecto, se procede a explicar en detalle los objetivos, requisitos y supuestos de trabajo necesarios para poder llegar a la solución.

2.1 Objetivos del Proyecto

En el análisis previo realizado para definir los objetivos de este proyecto y ver la amplitud de este, se ha observado que no existe ninguna otra herramienta similar, para la farmacia hospitalaria, a la que se va a desarrollar. Por ello, para poder limitar los objetivos y ver cuáles son los principales, ha sido necesario hablar con los especialistas en el campo, es decir, con el personal de la farmacia hospitalaria. Tras varias reuniones mantenidas, en las que se han expuesto los problemas y dificultades que tienen el día a día, se ha llegado a definir los siguientes requisitos:

- Objetivo 1: el principal objetivo de este proyecto ha sido crear una herramienta capaz de gestionar las unidades sobrantes y no útiles disponibles en los hospitales para que puedan ser utilizados por otras farmacias hospitalarias en pacientes con las mismas necesidades, reduciendo así los costes y la sobreexplotación de recursos.

- Objetivo 2: es necesario un proceso de alta rápido, inmediato y cómodo, sin necesidad de tener que firmar papeles o esperar la respuesta de autorización. Para ello, cualquier farmacia hospitalaria puede darse de alta directamente a través de la web aceptando los términos y condiciones. En esta primera aproximación, solo podrá haber un usuario por farmacia, dejando para líneas de trabajo futuro el poder crear un administrador de la farmacia y usuarios de consulta, similar a los usuarios que tienen los programas de gestión como Farmatic y Farmanager.

- Objetivo 3: también se debe tener una base de datos capaz de almacenar tanto el listado de fármacos oficiales registrados por el CIMA como de los albaranes de cada farmacia y

botiquín. A su vez, debe ser posible insertar y modificar los datos de manera rápida y útil para el usuario. Para ello el farmacéutico debe poder subir su stock de manera masiva si lo desea, o añadiendo puntualmente productos nuevos a su stock.

- Objetivo 4: se ha planteado el objetivo de que las búsquedas deben ser exactas y se deben mostrar según el criterio de cercanía al hospital o centro de salud. Para ello, se mostrará una lista de farmacias primero dentro de su localidad y después dentro de la provincia.

- Objetivo 5: relacionado con los costes, se estima que esta herramienta sea capaz de reducirlos y promueva la colaboración entre hospitales. No solo se habla de costes a nivel económico sino también reducción del impacto medioambiental o de los residuos generados.

- Objetivo 6: objetivos relacionados con el uso y gestión de la herramienta. Debido a que son usuarios no especializados, la aplicación debe ser intuitiva y de fácil aprendizaje ya que no debe suponer un esfuerzo para el farmacéutico.

- Objetivo 7: Utilizar *frameworks* de desarrollo que faciliten la creación de la aplicación.

2.2 Requisitos del Proyecto

Este proyecto, concebido como piloto, no necesita la utilización de software que requiera un ordenador de altas capacidades. Se basa principalmente en el aprendizaje de tecnologías como *MongoDB* o *Spring MVC* y uso de lenguajes ampliamente conocidos como Java.

Esta herramienta debería poder sustituir al sistema de préstamo actual, para ello debe seguir la misma normativa y procesos, pero de una manera más fácil, cómoda y rápida para el usuario.

En cuanto al diseño es flexible, estructurado y fácilmente escalable. Los componentes siguen un patrón de localización para que el usuario pueda aprender de manera rápida.

En el aspecto tecnológico se pretende utilizar herramientas que ofrezcan seguridad ya que se está tratando con datos sensibles y los datos no deben ser fácilmente accesibles por usuarios que no dispongan de los permisos necesarios.

La aplicación debe permitir el acceso a los farmacéuticos de las farmacias hospitalarias. Únicamente podrá acceder un usuario por farmacia.

En un futuro deberá poder de soportar más de 700 usuarios conectados en red, por tanto, debe alojarse en un servidor de la nube.

Por último, el desarrollo debe ser basado en módulos, altamente escalable y flexible.

2.3 Supuestos del Proyecto

Es necesario delimitar cuales van a ser los supuestos del trabajo. Los mismos son enumerados a continuación:

- Se utilizará la estructura de red y el diseño de componentes de la red neuronal de farmacias LudaFarma. Esta aplicación es un traslado de la red neuronal de farmacia comunitaria a la farmacia hospitalaria.

- Es importante conocer también el sector farmacéutico y, en concreto, la farmacia hospitalaria, para poder adaptarse lo máximo posible a sus procesos y poder implementarlos en el ámbito tecnológico.

- Debido a la sensibilidad de los datos, las simulaciones se harán con farmacias ficticias. Se trabajará con productos reales y con escenarios similares a los que se encuentran en la farmacia hospitalaria.

- Así mismo, toda la información relacionada con legislación vigente en el sector farmacéutico es aportada por la empresa LudaPartners S.L.

- La herramienta debe funcionar correctamente, independientemente del navegador que se utilice.

- Se parte con la idea de un sistema ya existente en el ámbito de farmacia comunitaria, por tanto, se mantendrá su forma de diseño y lógica de negocio aplicada.

- Se trabajará con una base de datos oficial de la Aemps. El stock disponible en la plataforma será corroborado con el que publica la Aemps en su página web.

2.4 Restricciones del Proyecto

A continuación, se exponen las restricciones que se van a tener en cuenta durante el desarrollo de este proyecto. Algunas de están pueden deberse a la índole del trabajo y la realidad de este. Otras se barajan como líneas de futuro:

- Es una herramienta que conecta farmacias entre sí, con datos reales. Por tanto, no se realizará la subida a producción con farmacias reales, sino con una simulación de estas.

- Debido a que no existe una pasarela entre el ERP y el sistema de control, no es problema de la herramienta que el stock de la farmacia en la plataforma no esté actualizado a diario.

- Se considerará fuera del ámbito del proyecto cualquier acción para poder transportar el medicamento de un hospital a otro. Quedando limitado el proyecto, a la búsqueda y reserva del producto.

- La aplicación permitirá el acceso a personas no registradas.

- Debido a las condiciones en las que se desarrolla este proyecto, el equipo solo estará formado por una persona, siendo el cliente el cotutor de este proyecto. Esta peculiaridad también afecta al tamaño del Sprint, siendo relativamente pequeño.

2.5 Tecnología Aplicada.

Respecto a la tecnología usada en el desarrollo de este proyecto, y al tratarse de una aplicación web, se puede diferenciar la parte del servidor y la parte del cliente. El *back-end* de la aplicación, correspondiente a la parte del servidor, está implementado en el lenguaje Java. Así mismo, dentro del servidor, podemos diferenciar la parte del controlador y la parte del modelo de datos. La primera parte, está implementada utilizando el *framework Spring*, que mezcla código Java, con etiquetas y código *HTML*. Por otro lado, el modelo de datos está definido en una adaptación de

MongoDB para Java. El servidor, en los primeros *Sprints*, y hasta su posterior subida a producción, estará corriendo en Apache Tomcat versión 8. Tras la subida a producción, el servidor quedará alojado en la nube de *Amazon Web Service (AWS)*, mientras que la base de datos Mongo se encuentra alojada en la nube de Google.

Por último, quedaría la parte del *front-end* que hace referencia a las vistas de la aplicación web. Las vistas se han realizado utilizando de nuevo el *framework Spring* y sus vistas *.jsp* que utilizan código *HTML* que admite CSS y *JQuery*.

2.5.1 Base de Datos NoSQL. MongoDB.[1]

Dentro de las bases de datos, se pueden distinguir dos grandes tipos, las relacionales o SQL y las no relacionales NoSQL. Las primeras, son las más conocidas y utilizadas, pero desde hace unos años las bases de datos no relaciones están cogiendo una gran acogida.

Para el desarrollo del proyecto se emplea una base de datos no relacional orientada a documentos, MongoDB. La razón de elegir una NoSQL es debido a que no es tan necesaria la integridad de los datos como la escalabilidad que se necesitará en líneas de futuro.

Como se ha mencionado anteriormente, Mongo es una base de datos orientada a documentos, es decir, guarda sus datos en documentos no en registros. Una de las diferencias más importantes con respecto a las bases de datos relacionales, es que no es necesario seguir un esquema. Los documentos de una misma colección, concepto similar a una tabla de una base de datos relacional, pueden tener esquemas diferentes.

MongoDB está escrito en C++ aunque las consultas se hacen pasando como parámetro un objeto en JSON. Sin embargo, con la ayuda de *drivers*, se puede ejecutar Mongo en cualquier otro lenguaje como Python o Scala. En este caso, se utilizará el *driver* de MongoDB para java en Eclipse.

Las principales características de MongoDB y por las cuales se ha optado por su implementación han sido:

- Permite procesar gran cantidad de información que se genera hoy en día.

[1] https://www.mongodb.com/

- Permite a las empresas ser más ágiles y crecer más rápidamente, crear nuevos tipos de aplicaciones/productos, mejorar la experiencia del cliente y reducir el tiempo de manufacturado o desarrollo del producto, reduciendo así los costes.

- Está orientada a documentos. Esto permite que en único documento se pueda contener toda la información de un cliente o producto, incluyendo datos de tipo array o subdocumentos sin necesidad de seguir un esquema idéntico para todos los documentos.

- Permite adaptar el esquema de la base de datos a las necesidades del proyecto rápidamente, disminuyendo el tiempo y coste de la puesta en producción de esta. Esto es así porque permite modificar el esquema desde el propio código de la aplicación sin necesidad de tener que realizar labores de administración en la base de datos.

- Da a los desarrolladores todas las funcionalidades que tienen las bases de datos relacionales (como índices sobre campos secundarios, un completo lenguaje para realizar las consultas, etc).

2.5.2 Spring.[2]

Debido al patrón de diseño escogido, MVC es necesario diferenciar claramente el *front-end* del *back-end*. Para ello, se va a utilizar el *framework* Spring.

Spring se lanzó en 2003 con Apache, como una plataforma de Java de código abierto, convirtiéndose en el framework más usado por crear código de alto rendimiento, liviano y reutilizable.

Su elemento clave es el soporte de infraestructura a nivel de aplicación, brindando un modelo completo tanto para la configuración como para el desarrollo de aplicaciones sin discriminar a la hora del despliegue de la plataforma.

La gran ventaja de no hacer discriminación y adaptarse es que solo es necesario enfocarse en la lógica de la aplicación, haciendo el proceso más corto, rápido y eficaz.
Las principales características de Spring son:

[2] https://spring.io/projects/spring-framework

- Tecnologías: como la inyección de dependencias, eventos, recursos o la validación.

- Acceso a datos: soporte DAO, JDBC, ORM, Marshalling XML .

- Gestión de transacciones.

- Integración: comunicación remota, JMS, JCA, JMX, correo electrónico, tareas, programación, caché.

- Pruebas (Testing): simulacro de objetos, el *framework TestContext*, *Spring MVC*, *WebTestClient*.

- Programación orientada a aspectos (AOP): permite la implementación de rutinas transversales.

- MVC (Modelo Vista Controlador).

- Seguridad.

- *Framework web: Spring WebFlux y Spring MVC.*

- Procesamiento de datos por lotes.

- Administración Remota: a través de este módulo se puede configurar la visibilidad y gestión de los objetos Java para la configuración local o remota vía JMX.

- Es un *framework* liviano debido a su implementación POJO (*Plain Old Java Object*) , *Spring Framework* no obliga al programador a heredar ninguna clase ni a implementar ninguna interfaz.

Sin embargo, lo más relevante de este *framework* es la inyección de dependencias. En el momento de escribir una aplicación Java compleja, las clases de la aplicación deben ser lo más independientes posible de otras clases Java, para aumentar la posibilidad de reutilizarlas y probarlas individualmente, mientras se prueban las unidades. Esta inyección permite unir estas clases y al mismo tiempo mantenerlas.

Spring implementa un patrón de diseño llamado *front-controller* desde el cual se canalizan todas las peticiones. Este recurso no deja de ser un *servlet* con una implementación específica del *framework*. Además, a través del *HandlerMapping* reconoce a qué URL hay que llamar para realizar la petición. A continuación, se llama al controlador, que ejecuta la lógica de negocio y en líneas generales devuelve un objeto de tipo *ModelAndView*,. Finalmente, el *front-controller* devuelve el resultado a la vista. (véase Figura 1).

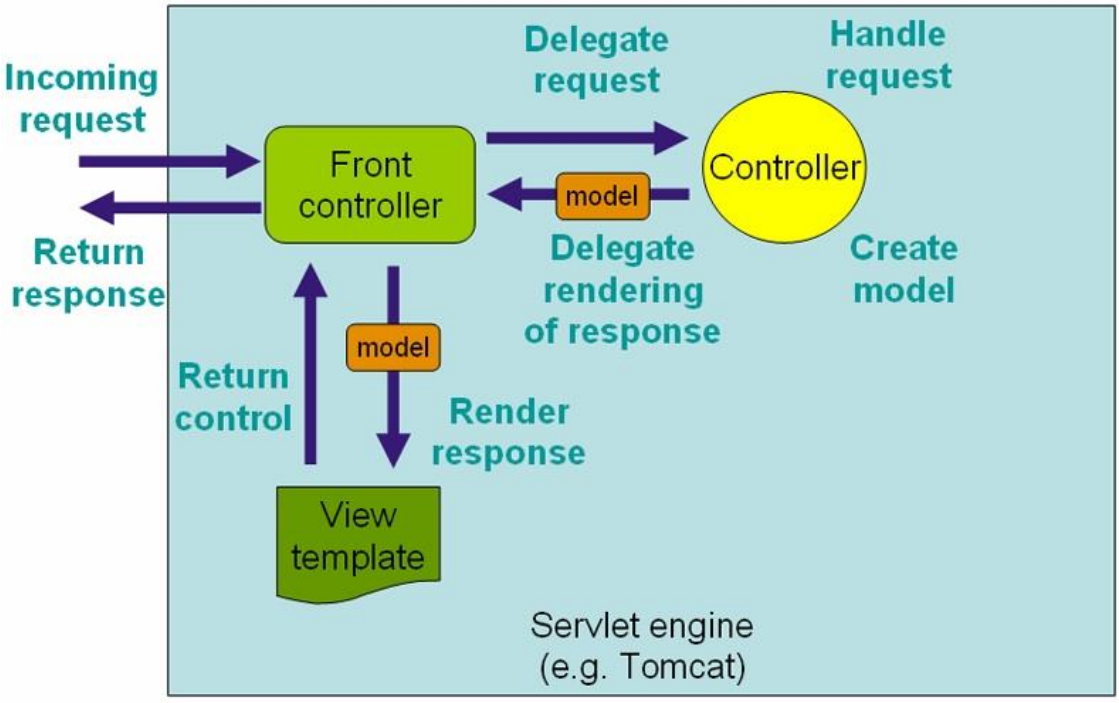

Figura 1. Relación de conexiones entre la vista, el controlador y el controlar frontal. Extraído de [2]

2.5.3 Apache Tomcat 8.[3]

Apache Tomcat es un contenedor de *servlets*, es decir, un programa capaz de recibir peticiones de páginas web y redireccionarlas a un objeto *Servlet*.

El servidor Tomcat ha sido desarrollado por *Apache Software Foundation* y fue uno de los responsables del éxito de Java.

[3] http://tomcat.apache.org/

Para poder instalar Tomcat es necesario tener el *jdk* de Java correspondiente a la versión, ya que en el proceso de instalación buscará la variable del sistema JAVA_HOME.

2.5.4 Amazon Web Services.[4]

Es una plataforma de uno público que ofrece diversidad de servicio en la nube. Dispone de una capa de uso gratuita durante un año que permite utilizar la mayoría de sus servicios con limitaciones en el tamaño o velocidad. Para este proyecto se va a utilizar el servicio *Elastic Beanstalk* que se considera un PaaS, es decir un producto como servicio.

Es de uso a demanda, por tanto. solo se paga por lo que se necesita y su labor es crear aplicaciones y desplegarlas integrando los servicios EC2, para crear el entorno, S3 para registrar el dominio de la web, *Simple Notification Service* para las notificaciones y otros servicios como los balanceadores de carga o el auto escalamiento.

Esta plataforma permite desplegar aplicaciones sin necesidad de tener que comprar un hosting propio y además ofrece herramientas extra para el testeo y el análisis de respuesta de la página web.

2.5.5 Bootstrap.[5]

Es un *framework* de código abierto para el diseño de aplicaciones web. Contiene plantillas de diseño con tipografía, formularios, botones, cuadros, menús de navegación y otros elementos de diseño basado en *HTML* y *CSS*, así como extensiones de *JavaScript* adicionales.

Bootstrap nació de *GitHub* y es empleado por empresas como la NASA o Twitter. Pese a que es un *framework* gratuito, parte de las plantillas creadas con Bootstrap por los diseñadores son de pago. Para usar este *framework* en una página *JSP*, el desarrollador solo debe descargar la hoja de estilo *Bootstrap CSS* y enlazarla en el archivo *HTML*.

[4] https://aws.amazon.com/es/
[5] https://getbootstrap.com/

Pese a que incorpora código *HTML* y *CSS* no tiene un soporte completo, pero si es compatible con la mayoría de los navegadores disponibles hoy en día. Además, también se le pueden incorporar funciones propias de *JavaScript*, añadiendo la librería *Jquery* al archivo JSP.

2.5.6 Maven[6]

Es una herramienta que facilita la compilación de un proyecto que tiene grandes dependencias. Maven es una herramienta capaz de gestionar un proyecto software completo, desde la etapa en la que se comprueba que el código es correcto, hasta que se despliega la aplicación, pasando por la ejecución de pruebas y generación de informes y documentación.

El ciclo por defecto de vida del *build* en Maven contiene las fases de: validación, compilación, test, empaquetas, pruebas de integración, verificar, instalar y desplegar. Permitiendo realizar el ciclo completo o por fases con una sola llamada.

Así mismo, Maven permite gestionar las dependencias entre módulos y versiones distintas de librerías. En este caso, solo hay que indicar los módulos que componen el proyecto, o qué librerías utiliza el software que estamos desarrollando en un fichero de configuración de Maven del proyecto llamado POM.

[6] https://maven.apache.org/

Capítulo 3.

Análisis y Diseño de la Herramienta.

3.1 Arquitectura de la Aplicación Web y Funcionamiento Básico.

Una de las arquitecturas más comunes es la de cliente-servidor. En esta arquitectura hay dos componentes, el servidor que suele ser un ordenador potente con hardware y software específico y que actúa como gestor de base de datos. Y, por otro lado, el cliente o grupos de clientes, estaciones de trabajo que piden varios servicios al servidor.

La principal característica de esta arquitectura es que permite conectar a varios clientes de manera simultánea. Sin embargo, existen distintos tipos dentro de este diseño:

- 2 capas: es la más clásica de todas y fue el inicio de la arquitectura cliente-servidor. El cliente manda peticiones al servidor y este responde directamente utilizando sus propios recursos.

- 3 capas: en este tipo de arquitectura existe un nivel intermedio, llamada software intermedio cuya tarea es proporcionar los recursos solicitados pero que requiere de otro servidor para hacerlo. La última capa es el servidor de datos que proporciona al servidor de aplicaciones los datos necesarios para poder procesar y generar el servicio que solicito el cliente en un principio.

- N capas: la arquitectura en tres niveles es potencialmente una arquitectura en N capas ya que así como está contemplado en tres niveles como el caso anterior puede estar compuesto por N servidores donde cada uno de ellos ofrece un servicio específico.

Este proyecto se basa en la arquitectura cliente – servidor de 3 capas, ya que necesita de otro motor para poder extraer los datos. (Véase Figura 2).

Figura 2 Arquitectura Cliente-Servidor. Extraída de [3]

Una vez conocido la arquitectura cliente – servidor, se procede a mostrar un esquema general de la arquitectura del proyecto y su funcionamiento básico. (Véase Figura 3).

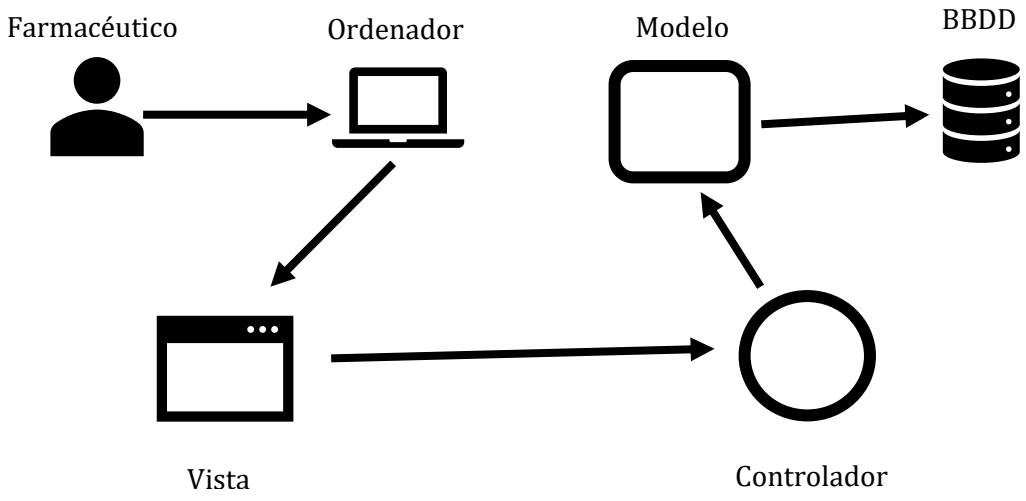

Figura 3 Arquitectura general de la aplicación.

1. El usuario a través de un ordenador y utilizando un navegador web, accede a la aplicación.

2. El usuario debe entrar en la plataforma a través de un proceso de identificación mediante usuario y contraseña.

3. La capa intermedia, que en el patrón de diseño empleado, MVC, correspondería al controlador, obtiene los datos introducidos por el usuario y comprueba si existen en la base de datos a través de la tercera capa, es decir, el modelo.

4. Si los datos son correctos, la segunda capa, el controlador, pide a la capa cliente, la vista, que muestre las operaciones que el usuario puede realizar.

El resto de la aplicación funciona de manera similar, por ejemplo, si un farmacéutico quiere ver los productos que tiene dados de alta:

1. El usuario selecciona en la vista la opción deseada.

2. Cada vez que ocurre un evento, este pasa al controlador y se produce la acción del controlador asociada a la opción elegida por el usuario en la vista.

3. Si es necesario, el controlador llama a la función correspondiente del modelo para que extraiga los datos.

4. El controlador devuelve los datos a la vista para que los muestre.

3.2 Arquitectura Web Basada en el Patrón de Diseño MVC.

Este patrón, formado por el modelo, la vista y el controlador, altamente utilizado en las aplicaciones web, permite un aislamiento de los datos, respetando el principio de la responsabilidad única, es decir, una parte del código no debe saber lo que hace el resto. Esto posibilita que, si se modifica una parte, por el ejemplo el modelo, solo sea necesario este y no el resto de las partes que forman el patrón de diseño.

El modelo, que se encarga de los datos de la *app*, consulta la base de datos y obtiene las farmacias que disponen de ese medicamento, respondiendo al controlador.

Una vez el controlador tiene los datos recibidos desde el modelo, los manda a la vista para que se aplique los estilos y formatos y construya la página web que visualiza el usuario.

Por tanto, existen tres componentes principales: (Véase Figura 4)

- Modelo: contiene una representación de los datos que maneja el sistema, su lógica de negocio, y sus mecanismos de persistencia. Se encarga, por ejemplo, de realizar consultas, inserciones o modificaciones a la base de datos.

- Controlador: se encarga de recibir las peticiones del usuario, enviarlas al modelo para extraer los datos y por último proporcionárselos a la vista.

- Vista: es la representación visual de los datos, la interfaz gráfica.

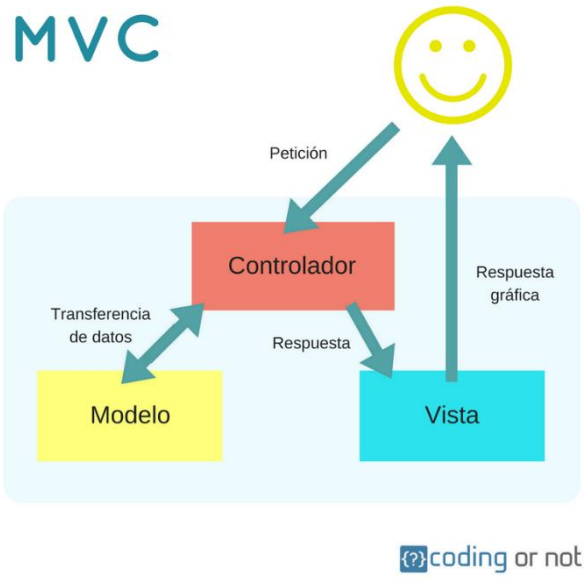

Figura 4 Diagrama de relación entre el modelo, la vista y el controlador. Extraído de [4]

3.3 Diseño mediante Metodología SCRUM.[7]

Scrum es una metodología para proyectos ágiles. Se basa en un proceso continuado de entregas parciales y regulares del producto final, priorizadas por el beneficio que obtiene el receptor final. Es idóneo para proyectos en los que los requisitos no están muy definidos, que, además, necesitan obtener resultados pronto y se caracterizan por ser innovadores y flexibles. Scrum se basa en la productividad del trabajo.

Además de lo anterior, Scrum está diseñado para aquellas situaciones en las que, con otras metodologías, las entregas se alargan demasiado y vence la fecha de riesgo, donde la prioridad pasa a ser la entrega, intentando no disparar los costes aún más y mejorando la calidad. Es decir, cuando se necesita solucionar ineficiencias sistemáticamente y se debe utilizar un proceso especializado de desarrollo del producto.

En Scrum un proyecto se desarrolla en ciclos cortos, de 2 semanas en el caso base. Donde cada iteración debe aportar un valor y un resultado completo al producto final. Se parte desde la lista de requisitos, ordenada por prioridades, las cuales han sido fijadas por el cliente, y se van organizando las iteraciones.

Para poder crear las diferentes iteraciones, lo primero es seleccionar aquellos requisitos que se van a realizar en esa iteración y resolver las dudas posibles. A continuación, se especifican una a una, las tareas necesarias para poder llevarlo a cabo y se reparten entre los miembros del equipo. Una vez creada la iteración, antes de comenzar a desarrollar, es necesario reunir al equipo, reunión de sincronización, y responder a estas tres cuestiones:

- ¿Qué he hecho desde la última reunión de sincronización para ayudar al equipo a cumplir su objetivo?

- ¿Qué voy a hacer a partir de este momento para ayudar al equipo a cumplir su objetivo?

- ¿Qué impedimentos tengo o voy a tener que nos impidan conseguir nuestro objetivo?

[7] https://www.scrum.org/resources/blog/que-es-scrum

Durante la ejecución de la iteración, el cliente, junto al equipo, puede ir modificando los requisitos de la lista en función de los resultados obtenidos, con el objetivo de maximizar la utilidad de lo que se desarrolla y el retorno de inversión.

Una vez que la iteración está completa, se realiza una reunión de inspección y adaptación. Para eso se realiza una demostración del trabajo realizado, el equipo presenta al cliente los requisitos completados en la iteración, en forma de incremento de producto preparado para ser entregado con el mínimo esfuerzo. En función de los resultados mostrados y de los cambios que haya habido en el contexto del proyecto, el cliente realiza las adaptaciones necesarias de manera objetiva, ya desde la primera iteración, re planificando el proyecto. Por último, se produce la retrospectiva, donde se analiza si la forma de trabajar ha sido la correcta y se detectan cuáles son los problemas que pueden impedir que se avance de manera continua. (Véase Figura 5).

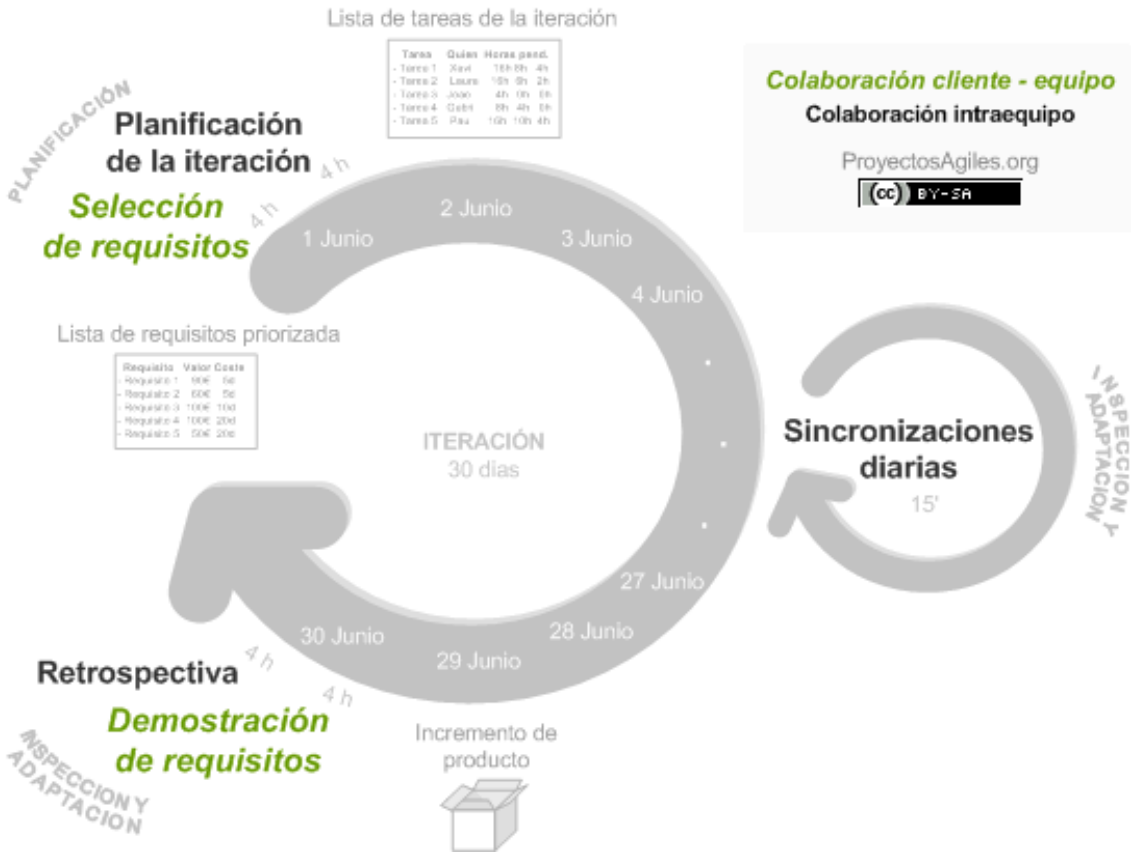

Figura 5 Ciclo de iteración de la metodología SCRUM. Extraída de [5]

3.4 Lista de Requisitos. (*Product Backlog*).

La lista priorizada de objetivos/requisitos representa la visión y expectativas del cliente respecto a los objetivos y entregas del proyecto.

El orden de sus ítems viene determinado por el valor que aporta al cliente final respecto a riesgos y coste estimado de completarlo (ROI). Es una planificación estratégica que evoluciona a lo largo de toda la vida del proyecto, debido a cambios de necesidades del cliente, *feedback* del mercado o aparición de nuevas ideas.

Esta lista de objetivos contiene los objetivos de alto nivel que se expresen en forma de historias de usuario. Para cada objetivo se indica el valor y el coste de realizarlo. Además, incluye también las posibles iteraciones esperadas por el cliente. Y por último aparecen los riesgos que se pueden encontrar a lo largo del proyecto y sus tareas para prevenirlos.

La lista de requisitos en la primera iteración no tiene que estar completa ni detallada, pero si debe contener la idea general y un listado de los principales requisitos para poder definir la meta.

3.5 Historias de Usuario

El proceso de extracción de la información relativa a las necesidades a cubrir se llevará a cabo entre los miembros del equipo de desarrollo y el propio cliente. Este proceso comienza en la fase inicial y se repite de formas reiterada a lo largo de las iteraciones para poder amoldarse lo máximo a los requerimientos del cliente y su producto final.

Las historias de usuario que se extraigan de las reuniones con el cliente describirán una funcionalidad que debe incorporar un sistema de software, y cuya implementación aporta valor al cliente.

La estructura de una historia de usuario está formada por:

- Nombre breve y descriptivo.

- Descripción de la funcionalidad a nivel de usuario.

- Criterio de verificación y aceptación por parte del cliente sobre la terminación de la funcionalidad descrita.

Las historias de usuario se definirán según este modelo:

Historia de Usuario	
ID	XXXX
Nombre	
Prioridad	
Riesgo	
Descripción	
Validación	

Tabla 1 Plantilla de historia de usuario

Donde cada campo tiene el significado de:

- ID: Se trata del identificador único asignado a este elemento del proyecto. Se seguirá el estándar HUXX para las historias de usuario.

- Nombre: es el nombre corto utilizado para describir brevemente la historia.

- Prioridad: es la importancia de desarrollar esta historia frente a las demás. Los posibles valores son baja, media, alta.

- Riesgo: Se trata de la importancia de la historia de usuario en relación con el conjunto del proyecto. De este modo, se cuantifica el daño provocado en caso de fallo. Los posibles valores son: bajo, medio, alto.

- Descripción: breve definición de la historia. Debe dejar clara la idea de la historia.

- Validación: Son las condiciones que deben cumplirse una vez la historia está completamente desarrollada para que se pueda por finalizada.

Historia de Usuario	
ID	HU01
Nombre	Introducir y gestionar automáticamente el stock.
Prioridad	Alta
Riesgo	Alto
Descripción	Como farmacéutico quiero poder meter de manera rápida todo mi stock de productos.
Validación	- Quiero poder introducir grandes cantidades de stock a través de una importación. - Quiero poder ver los detalles del producto, como su código nacional, el laboratorio, las unidades o la presentación. - Los datos almacenados se pueden modificar posteriormente. - Cuando se realice una reserva, las unidades reservadas del producto deben aparecer como reservadas.

Tabla 2 Historia de usuario 1.

Historia de Usuario	
ID	HU02
Nombre	Introducir y gestionar manualmente el stock.
Prioridad	Medio
Riesgo	Baja
Descripción	Como farmacéutico quiero poder meter productos de manera manual completando un formulario que indique el nombre, laboratorio, unidades, fecha de caducidad y CN.
Validación	- Poder elegir de la base de datos del sistema el medicamento que quiero ingresar. - Los datos almacenados se pueden modificar posteriormente. - Quiero poder introducir nuevos productos en el sistema.

Tabla 3 Historia de usuario 2.

Historia de Usuario	
ID	HU03
Nombre	Login/Registro farmacia.
Prioridad	Alta
Riesgo	Alto
Descripción	Quiero poder tener un perfil para cada farmacia desde el que se pueda administrar el stock y realizar las acciones propias del sistema.
Validación	- Quiero poder crear un perfil de farmacia. - El perfil debe estar asociado a un hospital. - Cada farmacia solo puede estar asociada a un hospital. - Quiero poder entrar a la página principal de la web. - Cada farmacia debe tener asociado un usuario.

Tabla 4 Historia de usuario 3.

Historia de Usuario	
ID	HU04
Nombre	Realizar búsquedas de fármacos
Prioridad	Alta
Riesgo	Alto
Descripción	Como farmacéutico quiero poder buscar un medicamento por código nacional y que me indique las farmacias hospitalarias que disponen de él y sus unidades.
Validación	- Quiero buscar por CN. - Quiero que muestre un listado de farmacias que disponen de ese medicamento. - Quiero ver las unidades disponibles.

Tabla 5 Historia de usuario 4.

Historia de Usuario	
ID	HU05
Nombre	Reservar un producto
Prioridad	Alta
Riesgo	Alto
Descripción	Una vez realizada la búsqueda debo poder seleccionar una farmacia y poder reservar el número de unidades que necesito.
Validación	- La farmacia debe poder reservar las unidades de un medicamento a otra farmacia. - La farmacia origen debe poder realizar el papeleo de préstamo a través del sistema. - La farmacia destino debe ser notificada de la reserva.

Tabla 6 Historia de usuario 5.

Historia de Usuario	
ID	HU06
Nombre	Notificación de los cambios de estado.
Prioridad	Media
Riesgo	Media
Descripción	El sistema debe permitir al usuario poder cambiar el estado de la reserva. Posibles estados: pendiente, aceptado, enviado, finalizado y cancelado.
Validación	- El farmacéutico debe ser capaz de saber cómo se encuentra una reserva realizada o emitida. - Se debe poder modificar el estado de manera manual. - El estado pendiente, es el estado inicial y automático al realizar una reserva. A su vez, el estado finalizado solo puede ponerlo quien solicita la reserva una vez que recibe el producto. - El estado cancelado debe poder activarse en cualquier momento.

Tabla 7 Historia de usuario 6.

Historia de Usuario	
ID	HU07
Nombre	Poder visualizar las reservas.
Prioridad	Media
Riesgo	Media
Descripción	El sistema debe permitir al usuario poder ver las reservas que tiene activas, así como las reservas que se han completado.
Validación	- El farmacéutico debe poder diferencia entre reservas emitidas y recibidas y entre activas y completadas.

Tabla 8 Historia de usuario 7.

Historia de Usuario	
ID	HU08
Nombre	Aplicación web
Prioridad	Media
Riesgo	Bajo
Descripción	El sistema debe poder abrirse en cualquier navegador web.
Validación	- El acceso al sistema debe poder hacerse a través de cualquier navegador web. - Para poder entrar deben registrarse con usuario y contraseña. - La aplicación web debe ser visual, predominando las acciones relacionadas con el stock y las búsquedas.

Tabla 9 Historia de usuario 8.

Historia de Usuario	
ID	HU09
Nombre	Trabajar con hojas de cálculo
Prioridad	Media
Riesgo	Bajo
Descripción	Como usuario quiero importar y exportar datos desde hojas de cálculo.
Validación	- Quiero importar datos desde un fichero .xls

Tabla 10 Historia de usuario 9.

Historia de Usuario	
ID	HU10
Nombre	Base de datos
Prioridad	Alta
Riesgo	Alto
Descripción	Como desarrollador quiero que los datos introducidos sean persistentes.
Validación	- Quiero que los datos estén almacenados en una base de datos. - Quiero que se le puedan añadir datos. - Quiero que los datos ya introducidos se puedan modificar. - Quiero poder consultar los documentos.

Tabla 11 Historia de usuario 10.

Historia de Usuario	
ID	HU11
Nombre	Funcionamiento en Windows
Prioridad	Alta
Riesgo	Alto
Descripción	Quiero poder utilizar la aplicación en Windows XP o superior.
Validación	- Diseño de la arquitectura de la aplicación.

Tabla 12 Historia de usuario 11.

Historia de Usuario	
ID	HU12
Nombre	Modificar datos de los usuarios.
Prioridad	Media
Riesgo	Medio
Descripción	Quiero que un usuario puede modificar sus datos y los de su farmacia.
Validación	- Debe poder modificar sus datos personales o los de su farmacia. - Los datos que se pueden modificar son el email del usuario/hospital y el número de teléfono.

Tabla 13 Historia de usuario 12.

Historia de Usuario	
ID	HU13
Nombre	Debe haber un administrador
Prioridad	Alta
Riesgo	Alto
Descripción	Se completará cuando se realice
Validación	- El administrador debe poder ver todos los encargos y modificar el estado de los mismos. - El administrador debe poder ver todos los usuarios y modificar los datos de estos, salvo la contraseña. El administrador debe poder ver todas las farmacias adheridas a la red y modificar los datos de estas.

Tabla 14 Historia de usuario 13.

3.6 Tareas

Las historias de usuario están formadas por un conjunto de tareas que permiten la resolución de la historia.

Tarea	TXX
Historia de usuario	
Estado	
Descripción	

Tabla 15 Esquema general de una tarea.

Donde cada campo tiene los siguientes significados:

- Tarea: es el identificador único para este elemento del proyecto. Su formato es: TXX.

- Historia de usuario: toda tarea pertenece a una historia de usuario que se indica en este campo.

- Estado: se trata de la fase en la que se encuentra esta tarea. Posibles valores: no iniciada, en proceso, finalizado.

- Descripción: una breve explicación de la finalidad de la tarea.

Para evitar sobrecargar la memoria con datos repetitivos, solo se van a mostrar en las siguientes tablas las tareas principales, dentro de cada Sprint, se definirán las tareas propias de cada historia de usuario en detalle.

Tarea	**T01**
Historia de usuario	HU01
Estado	Finalizado
Descripción	Crear interfaz con la BBDD para la colección *items* en la capa de datos.

Tabla 16 Tarea 1

Tarea	**T02**
Historia de usuario	HU01
Estado	Finalizado
Descripción	Crear la clase *Item* en la capa de negocio.

Tabla 17 Tarea 2.

Tarea	**T03**
Historia de usuario	HU01
Estado	Finalizado
Descripción	Crear colección *items* en la base de datos.

Tabla 18 Tarea 3.

Tarea	**T04**
Historia de usuario	HU01
Estado	Finalizado
Descripción	Crear colección *stockItem* en la base de datos.

Tabla 19 Tarea 4.

Tarea	**T05**
Historia de usuario	HU01
Estado	Finalizado
Descripción	Crear procedimiento para inserción, modificación y consulta en la colección *items*.

Tabla 20 Tarea 5.

Tarea	**T06**
Historia de usuario	HU01
Estado	Finalizado
Descripción	Crear procedimiento para inserción, modificación y consulta en la colección *stockItem*.

Tabla 21 Tarea 6.

Tarea	**T08**
Historia de usuario	HU01
Estado	Finalizado
Descripción	Crear la interfaz de usuario para la gestión de la colección *stockItem*.

Tabla 22 Tarea 7.

Tarea	**T08**
Historia de usuario	HU02
Estado	Finalizado
Descripción	Crear interfaz con la BBDD para la colección *stockItem* en la capa de datos.

Tabla 23 Tarea 8.

Tarea	T09
Historia de usuario	HU02
Estado	Finalizado
Descripción	Crear la interfaz de usuario para la inserción manual de productos.

Tabla 24 Tarea 9.

Tarea	T10
Historia de usuario	HU02
Estado	Finalizado
Descripción	Crear interfaz con la BBDD para la colección *pharmacies* en la capa de datos.

Tabla 25 Tarea 10.

Tarea	T11
Historia de usuario	HU03
Estado	Finalizado
Descripción	Crear clase *Pharmacy* en la capa de negocio

Tabla 26 Tarea 11.

Tarea	T12
Historia de usuario	HU03
Estado	Finalizado
Descripción	Crear la colección *pharmacies* en la base de datos.

Tabla 27 Tarea 12.

Tarea	T13
Historia de usuario	HU03
Estado	Finalizado
Descripción	Crear clase *User* en la capa de negocio

Tabla 28 Tarea 13.

Tarea	T14
Historia de usuario	HU03
Estado	Finalizado
Descripción	Crear procedimiento para la inserción, modificación y consulta de la colección *pharmacies*.

Tabla 29 Tarea 14.

Tarea	T15
Historia de usuario	HU03
Estado	Finalizado
Descripción	Crear procedimiento para la inserción, modificación y consulta de la colección *users*.

Tabla 30 Tarea 15.

Tarea	T16
Historia de usuario	HU03
Estado	Finalizado
Descripción	Crear clase *User* en la capa de negocio

Tabla 31 Tarea 16.

Tarea	T17
Historia de usuario	HU03
Estado	Finalizado
Descripción	Crear la interfaz de usuario para la gestión de la colección *pharmacies*.

Tabla 32 Tarea 17

Tarea	T18
Historia de usuario	HU03
Estado	Finalizado
Descripción	Crear la interfaz de usuario para la gestión de la colección *users*.

Tabla 33 Tarea 18.

Tarea	T19
Historia de usuario	HU04
Estado	Finalizado
Descripción	Crear la interfaz de usuario para la gestión del servicio de búsquedas.

Tabla 34 Tarea 19.

Tarea	T20
Historia de usuario	HU04
Estado	Finalizado
Descripción	Crear los procedimientos para consultar, modificar e insertar en la colección *bookings*.

Tabla 35 Tarea 20.

Tarea	T21
Historia de usuario	HU04
Estado	Finalizado
Descripción	Crear la interfaz búsquedas con la base de datos en la capa de datos.

Tabla 36 Tarea 21.

Tarea	T22
Historia de usuario	HU05
Estado	Finalizado
Descripción	Crear la interfaz de usuario para la gestión de la colección *bookings*.

Tabla 37 Tarea 22.

Tarea	T23
Historia de usuario	HU05
Estado	Finalizado
Descripción	Crear la colección *bookings* en la base de datos.

Tabla 38 Tarea 23.

Tarea	T24
Historia de usuario	HU05
Estado	Finalizado
Descripción	Crear la clase *Booking* en la capa de negocio.

Tabla 39 Tarea 24.

Tarea	T25
Historia de usuario	HU05
Estado	Finalizado
Descripción	Crear los procedimientos para consultar, modificar e insertar en la colección *bookings*.

Tabla 40 Tarea 25.

Tarea	T26
Historia de usuario	HU05
Estado	Finalizado
Descripción	Crear la interfaz reservas con la base de datos en la capa de datos.

Tabla 41 Tarea 26.

Tarea	T27
Historia de usuario	HU06
Estado	Finalizado
Descripción	Crear la interfaz de usuario para la gestión del servicio de estados.

Tabla 42 Tarea 27.

Tarea	T28
Historia de usuario	HU06
Estado	Finalizado
Descripción	Crear el atributo *status* en la capa de negocio.

Tabla 43 Tarea 28.

Tarea	T29
Historia de usuario	HU06
Estado	Finalizado
Descripción	Crear la interfaz para gestionar el atributo *status* con la base de datos en la capa de datos.

Tabla 44 Tarea 29.

Tarea	T30
Historia de usuario	HU07
Estado	Finalizado
Descripción	Crear interfaz de usuario para la consulta de la colección *bookings*,

Tabla 45 Tarea 30.

Tarea	T31
Historia de usuario	HU08
Estado	Finalizado
Descripción	Diseño de la arquitectura del sistema.

Tabla 46 Tarea 31.

Tarea	T32
Historia de usuario	HU09
Estado	Finalizado
Descripción	Crear plantilla de Excel para cada tabla

Tabla 47 Tarea 32.

Tarea	T33
Historia de usuario	HU09
Estado	Finalizado
Descripción	Crear interfaz con la base de datos en la capa de datos.

Tabla 48 Tarea 33.

Tarea	T34
Historia de usuario	HU09
Estado	Finalizado
Descripción	Crear la interfaz de usuario para la importación de datos desde Excel.

Tabla 49 Tarea 34.

Tarea	T35
Historia de usuario	HU10
Estado	Finalizado
Descripción	Determinar las colecciones de la base de datos.

Tabla 50 Tarea 35.

Tarea	T36
Historia de usuario	HU11
Estado	Finalizado
Descripción	Determinar las relaciones entre colecciones.

Tabla 51 Tarea 36.

Tarea	T37
Historia de usuario	HU13
Estado	Finalizado
Descripción	Crear interfaz para el perfil administrador

Tabla 52 Tarea 37.

3.7 Estimación

La tarea de estimar consiste en cuantificar el esfuerzo necesario para llevar a cabo las diferentes partes de un proyecto. Esta información ayudará a elegir las tareas que se van a realizar primero en función de su coste y su prioridad.

La estimación de proyectos software es una tarea muy compleja, pero de vital importancia en toda la etapa de desarrollo del software. Nunca podrá ser un valor extremadamente preciso debido a todas las variables que influyen en su cálculo. Sin embargo, cuanto mejor sea la estimación que se haga, mejor se podrá maniobrar para logar cumplir los objetivos y mejorar la rentabilidad del proyecto.

En este caso particular, al tratarse de una metodología ágil se va a realizar la estimación utilizando el método *Planning Poker*.

Además, esta técnica se puede utilizar conjuntamente con otras como el juicio de un experto y las estadísticas de rendimiento del equipo en proyectos similares. En caso de utilizar la combinación de varias técnicas habría que asignar un peso a cada una de ellas y utilizar una fórmula de este estilo, ejemplo:

- Juicio experto: 20%
- *Planning Poker*: 40%
- Estadísticas de velocidad del equipo: 40%

El resultado final para esta historia seria: 20%(20) + 40%(15) + 40%(11) ; es decir 14

Debido a que no se cuenta con el juicio de un experto, que pueda dar su opinión y la falta de experiencia del equipo en proyectos similares, en el desarrollo solo se va a utilizar el *Planning Poker*.

Planning Poker es una de las técnicas más extendidas utilizadas para realizar la estimación en proyectos ágiles. Este proceso tiene como objetivo el poder realizar un cálculo del esfuerzo necesario para llevar a cabo las distintas historias de usuario mediante un consenso entre los miembros encargados de realizar las distintas tareas que componen cada fase.

La técnica consiste en que cada miembro del equipo elige una carta de la baraja, sin mostrarla para evitar que pueda haber influencia. Esta carta representa el valor estimado del esfuerzo que el considera para realizar la tarea. En caso de discrepancias muy altas se puede realizar un pequeño debate para ver los puntos de discordia y llegar a un acuerdo. Los valores de la baraja suelen ser la secuencia de Fibonacci, con alguna pequeña variación, por lo que no es una sucesión lineal, de esta forma se refleja la incertidumbre a la hora de realizar las estimaciones, a mayor grado de complejidad, mayor es esta incertidumbre.

Por tanto, la estimación se lleva a cabo sobre el coste de realizar las historias de usuario completas, es decir, del conjunto de tareas que la componen.

3.8 Estimación en este Proyecto

A continuación, se procede a explicar la estimación que se va a realizar tras el análisis de las historias de usuario y la definición de tareas.

La técnica que se va a utilizar estará basado en el *Planning Poker* anteriormente explicado, pero adaptado a las condiciones especiales de este proyecto. Esta adaptación será lo que se detalla a continuación.

En el inicio de cada Sprint se aplicará la técnica mencionada a las historias de usuario restantes. De este modo se ordenarán por prioridad y se obtendrá como resultado la cantidad de trabajo que se llevará a cabo durante ese Sprint.

El prototipo resultante de una iteración debe ser funcional, en la medida de sus posibilidades. Por tanto, a la hora de asignar las historias de usuario a un Sprint, la suma de tiempos estimados debe ser siempre inferior al tiempo total estimado de la iteración. Con esto se evita que una historia pueda quedarse sin completar y afecte al funcionamiento del prototipo.

Inicialmente, es necesario priorizar las historias de usuario. Y durante el proyecto, en intervalos regulares, se va estimando el esfuerzo de cada una en puntos de usuario. Los valores más frecuentes son:0, 1, 3, 5, 8, 13, 20, 40 y 100. Siendo 0 un esfuerzo prácticamente nulo y 100 un esfuerzo extremo, fuera de las posibilidades del momento. El *Product Owner* expone cada una

de las historias y el equipo intenta llegar a un consenso con el esfuerzo asignado. Se suelen utilizar otras historias para comparar. Estos valores serán los utilizados en este proyecto.

Un ejemplo de funcionamiento y de formato de las tablas de estimación utilizando *Planning Poker* sería el que se muestra a continuación. (Véase Tabla 53). Además, será el utilizado durante el desarrollo del presente proyecto.

Historias de Usuario	**Miembros**			**Estimación media**	**Prioridad**
	A	**B**	**C**		
HU01	8	13	13	11,3	Alta
HU02	5	5	8	7	Baja
HU03	13	20	20	17,6	Alta

Tabla 53 Ejemplo de estimación con Planning Poker

En el ejemplo anterior se supone una capacidad de trabajo de 20h. Como se puede observar, los valores deben estar comprendidos entre la selección arriba mencionada.

El proceso para realizar la estimación es elegir aquellas historias de usuario que pueden ocupar el tiempo de 20h y estén sin completar. Dentro de esta elección deben ordenarse por prioridad, es decir, primero alta, luego media y por último baja.

Por tanto, en la Tabla 53 cada miembro ha elegido una estimación que piensa adecuada y se ha realizado su media. En este caso, la historia de usuario que se debería elegir es la HU03. No se podrían añadir más historias de usuario debido a que no caben en el Sprint.

La diferencia entre el tiempo estimado y tiempo total de la iteración puede utilizarse para solventar los normales errores de estimación. Esto permite recuperar posibles retrasos en el desarrollo debidos a cualquier causa. Lo importante es que al finalizar el Sprint el resultado sea un prototipo funcional que pueda ser presentado al cliente.

Capítulo 4

Desarrollo de la Herramienta.

4.1 Sprint 0

Se conoce como Sprint 0 a la fase inicial del proyecto a la que se dedica aproximadamente una semana, independientemente del formato, duración, etc., del resto de Sprints. En este Sprint es donde se va a preparar el equipo tanto tecnológicamente como metodológicamente para que el desarrollo del proyecto tenga un buen comienzo, en especial en casos en los que se desconoce la metodología o no se tiene mucha práctica con ella.

El Sprint 0 tiene por objetivo preparar el proyecto desde las siguientes perspectivas:

- Organizativa: repartición de tareas y tiempos para tener un buen comienzo.

- Tecnológica: es necesario estipular qué medios se van a necesitar para llevarlo a cabo.

- Metodológica: definir metodología que se va a emplear en función de la necesidad.

Esto se puede expresar en lo siguiente:

- **Definir qué quiere el cliente:** para ello, el jefe de equipo, *product owner*, se encarga de definir con el cliente las características y funcionalidades del proyecto, con el mayor detalle posible. Con esto se crea el documento de las historias de usuario.

- **Construir el *Product Baclog***: Las historias de usuario establecen unidades que pueden presentarse a los clientes como elementos acabados.

- **Reuniones de equipo:** tras la elaboración de la documentación es necesario analizar donde puede faltar información, cuáles son las dependencias entre historias, posibles dificultades técnicas, dividir las historias de usuario debido a su complejidad.

Los motivos para hacerlo de este modo son que nos aporta las siguientes ventajas, siendo estas de gran ayuda en el desarrollo del proyecto, y en la mejora de la productividad del equipo.

- Obtenemos una definición contrastada con el cliente en forma de historias de usuarios.

- El equipo participa en la preparación del desarrollo identificando necesidades, dificultades, ventajas.

- La preparación de cada Sprint será más fácil de realizar porque el equipo conoce con detalle el proyecto.

- El Sprint 0 de este proyecto se podría identificar perfectamente con la fase de análisis descrita en capítulos anteriores de este documento.

Por último, queda definir la capacidad de trabajo del equipo por cada iteración. Para ello es necesario definir el tamaño de la iteración y el rendimiento del equipo. (Véase Tabla 54).

Tamaño del Sprint	2 semanas. (10 días laborables)
Trabajo por día	4 horas.
Horas por Sprint	40 horas.

Tabla 54 Definición de los tiempos de trabajo

Como se ha comentado en las restricciones del trabajo, el trabajo estará desarrollado por una única persona. De esta forma para todo lo relacionado con el desarrollo en sí de la aplicación siempre se tendrá en consideración esta limitación. No siendo así para situaciones que puedan ser simuladas de forma sencilla y que den al proyecto un aspecto más real y natural.

4.2 Sprint 1

Lo primero a realizar es mostrar el *Product Backlog* generado en la fase de análisis. A continuación, se muestra una tabla resumida de las historias de usuario. (Véase Tabla 55).

Historias de Usuario	Miembros		Estimación media	Prioridad
	A	B		
HU01	20	40	30	Alta
HU02	13	20	16,5	Medio
HU03	8	13	10,5	Alta
HU04	8	13	10,5	Alta
HU05	20	8	14	Alta
HU06	5	20	12,5	Media
HU07	8	20	14	Media
HU08	40	40	40	Media
HU09	8	13	10,5	Media
HU10	20	20	20	Alta
HU11	8	3	5,5	Alta
HU12	5	5	5	Media
HU13	20	13	16,5	Alta

Tabla 55 Product backlog Sprint 1.

Como se puede observar en la Tabla 55, varias historias tienen la misma prioridad. En este caso, queda a juicio del equipo aquellas que se realizarán en primer lugar.

Para este primer Sprint se realizan las siguientes tareas. (Véase Tabla 56)

Horas máximas	40
Horas utilizadas	36
Horas Libres	4
Historia	Horas estimadas
HU11	20
HU03	10,5
HU10	5.5

Tabla 56 Resumen de tareas Sprint 1.

En este caso, se observa que las historias de usuario seleccionadas para esta iteración son: Véase Tabla 57, Tabla 58, Tabla 59)

Historia de Usuario	
ID	HU03
Nombre	Login/Registro farmacia.
Prioridad	Alta
Riesgo	Alto
Descripción	Quiero poder tener un perfil para cada farmacia desde el que se pueda administrar el stock y realizar las acciones propias del sistema.
Validación	- Quiero poder crear un perfil de farmacia. - El perfil debe estar asociado a un hospital. - Cada farmacia solo puede estar asociad a un hospital. - Quiero poder entrar a la página principal de la web. - Cada farmacia debe tener asociado al menos un usuario.

Tabla 57 Historia de usuario 3.

Historia de Usuario	
ID	HU10
Nombre	Base de datos
Prioridad	Alta
Riesgo	Alto
Descripción	Como desarrollador quiero que los datos introducidos sean persistentes.
Validación	- Quiero que los datos estén almacenados en una base de datos. - Quiero que se le puedan añadir datos. - Quiero que los datos ya introducidos se puedan modificar. - Quiero poder consultar los documentos.

Tabla 58 Historia de usuario 10.

Historia de Usuario	
ID	HU11
Nombre	Funcionamiento en Windows
Prioridad	Alta
Riesgo	Alto
Descripción	Quiero poder utilizar la aplicación en Windows XP o superior.
Validación	- Diseño de la arquitectura de la aplicación.

Tabla 59 Historia de usuario 11.

Se va a realizar en primer lugar la base de datos de tipo NoSQL, MongoDB. En este caso, vamos a crear una base de datos en MongoDB, concretamente, en la nube de Google. Desde la propia página web de Mongo, permite crear un *cluster* en diferentes nubes, como AWS, *GoogleCloud* o Azure. Se ha elegido la nube de Google con el plan gratuito.

El *cluster* se ha llamado *db01* y tiene un usuario denominado *admin* con todos los permisos. Además, se ha configurado para que se pueda acceder desde cualquier dirección IP.

Debido a que es una base de datos no relacional, no es obligatorio que haya una homogeneidad entre los documentos. Sin embargo, se recomienda que se cree un esquema para poder luego explotarla de manera adecuada.

Procedemos a definir las colecciones necesarias que debe llevar la base de datos y un tipo de esquema de cada colección:

- ***Pharmacies***

La colección pharmacies agrupa a todas las farmacias dadas de alta en la plataforma. Dentro de la colección *pharmacies*, se encuentran los siguientes campos:

- *_id*: corresponde a un *ObjectId* y es el identificador único de este documento. Es un valor que crea automáticamente Mongo y que es obligatorio.
- *Name*: indica el nombre de la farmacia hospitalaria.
- *Address*: es un array que almacena dirección completa de la farmacia.
- *Phone*: indica el número de teléfono por si es necesario contactar.
- Hospital: indica a qué hospital público pertenece esta farmacia. Esto es debido a que los hospitales tienes hospitales privados y centro de salud asociados.
- Email: indica el correo electrónico con el que se ha dado de alta el perfil de farmacia.
- CIF: indica el CIF del hospital y servirá para ver si una farmacia ya está registrada.

- ***Users***

Users es una colección que almacena los usuarios que se encargan de gestionar los perfiles de farmacia. En la colección *users* se encuentran los siguientes campos:

- *_id*: identificador único del documento y que está relacionado con un documento de la colección *pharmacies*.
- Email: indica el email del usuario. Sirve como clave única para no repetir registros de usuarios con el mismo e-mail.
- *Name*: es el nombre del usuario.

- *pharmacyId*: indica como referencia indirecta, el identificador de la farmacia con la que está relacionado este usuario.

- ***Items***

Aquí se almacenan todos los productos declarados en la base de datos. Desde esta tabla se realiza el autocompletado a la hora de insertar los datos. En esta colección se encuentran los siguientes campos:

- *_id*: en este caso trataremos como id al código nacional de los medicamentos. El código nacional es un código único de 5 números.

- *Laboratory*: indica el laboratorio de procedencia del medicamento. Datos oficiales de CIMA.

- *Name*: indica el nombre estandarizado del producto según CIMA.

- ***StockItem***

En esta colección se guarda el stock de cada farmacia. Para poder guardar el stock y relacionarlo con una farmacia se crea un campo dentro del documento de tipo *Map* donde se relacionada cada producto con la farmacia que lo tiene y se va guardando de manera individual. En esta colección se encuentran los siguientes campos:

- *_id*: identificador único.

- *stockItemId* es un campo de tipo Objeto que contiene un documento con el CIF de la farmacia y el código nacional del producto.

- *Units*: almacena las unidades disponibles de ese producto.

- ***Bookings***

Son las reservas, entendidas como procedimiento, que realizan los usuarios. En este colección se encuentran los siguientes campos:

- *_id*: identificador único.
- *bookingId*: hace una referencia indirecta al pedido que lleva asociado una reserva.
- *originPharmacy*: almacena el identificador de la farmacia origen.
- *destinationPharmacy*: similar al anterior, pero es la farmacia que recibe la reserva.
- *Date*: indica la fecha y hora en la que se hizo la reserva.
- *Status*: estado en el que se encuentra la reserva: solicitado, aceptado, enviado, finalizado y cancelado.
- *itemId*: indica el código nacional del producto que se ha solicitado.

Lo siguiente a realizar es la historia de usuario número 12, ya que en parte se encuentra hecha en el análisis. Sin embargo, no se puede confundir análisis con desarrollo. En el análisis se eligió una arquitectura y una configuración en base a esta historia de usuario, pero en este apartado lo que se va a hacer es poner en práctica los puntos establecidos anteriormente.

Lo siguiente es a realizar es crear el proyecto en Eclipse que albergue el sistema completo.
Para ello se crea un proyecto Spring en Eclipse 2018-12, los propios drivers de Eclipse de Spring y Maven hacen que al crear el proyecto se estructure automáticamente el sistema de archivos y el archivo pom.xml. En este último, es necesario añadir las dependencias de Mongo y Spring.

En este primer Sprint se va a trabajar desde el *localhost* con un servidor Tomcat. Para ello es necesario configurarlo en el proyecto cambiando el puerto por defecto a otro distinto como puede ser el 8905.

4.2.1 Iniciar Sesión con un Usuario.

Para la primera acción, iniciar sesión con un usuario, son necesarios los siguientes elementos:

- Clase *User*.java: implementa los servicios relacionados con la colección del modelo de datos *User*.

- Clase *ModelFactory*.java: implementa la conexión con la base de datos y sus colecciones.

- Clase *LoginController*: maneja las llamadas y respuestas al *servlet*. Permite conectar las vistas con el *back-end.*

- Vista *Login*.jsp: es la interfaz con el usuario final.

Para la clase *User.java* se definen los atributos propios del modelo de datos: *name*, email, *pharmacyId*, *password*. Los métodos *getter* y *setter* así como el constructor vacío de la clase.

Además se crea la función *checkUser*(email, *password*, *mongoClient*, *database*). Esta función está comprueba que el usuario y la contraseña coinciden con algún documento de la colección *users* de la base de datos. Para ello genera una lista de *BasicDBObject* donde añade los dos atributos pasados como parámetros, mail y *password*. A continuación, realiza una consulta a través de un *find* y almacena los datos en un *FindIterable*. Si este devuelve un elemento, la función devuelve el valor true. En caso contrario, devuelve false. Con eso, se comprueba que existe un documento en la colección *users* que tiene ambos parámetros.

El elemento *password* está cifrado con md5. Por tanto, se comprueba que las claves generadas tras su cifrado son las mismas. En este primero Sprint se ha hecho un proceso de inicio de sesión manual, sin embargo, una vez que se configure Spring Security, esta función quedará obsoleta.

La clase *MongoFactory.java* implementa la conexión con el modelo de datos. Para ello utiliza objetos de tipo *MongoClient*. A través de la variable *MongoClient* se conecta con la *url* que ofrece Mongo Atlas en la nube de Google para acceder a los datos.

La clase *LoginController* es el controlador, algo propio del *framework* Spring. En esta clase se implementa la relación entre las vistas, el modelo de datos y el DAO. Para ello generamos dos funciones que devuelven cada una un *ModelAndView*, uno para mostrar la vista principal y

mandar la solicitud al servidor y otra para realizar el proceso de *login* y devolver la respuesta del servidor.

La primera función, *showLogin*, cuyo *requestMapping*, esquema de la url para poder acceder, coincide con el nombre de la función. Se encarga de devolver la vista *login.jsp* para que el usuario pueda introducir los datos. Por otro lado, la segunda, *loginProcess*, trabaja sobre un *bean*, es decir, un componente propio de spring que permite guardar los datos de la vista, que le pasa como parámetro para poder coger los datos del formulario de la vista, mapearlos en la clase *User* y proceder a pasar dichos datos como parámetros de *checkUser* para que los valide. De ser correcto, redirige a la página principal. En caso contrario, abre una página de error, indicando que los datos introducidos son incorrectos y que redirecciona a la página de *login.*

Por último, falta definir la vista de la página web a través de un archivo .jsp. Para ello se crear un archivo *login.jsp* que contiene un formulario con dos *inpust* de tipo mail y *password* para el e-mail y contraseña respectivamente. Por último, el botón de tipo *submit* para iniciar sesión. Gracias a la etiqueta *action* propia del formulario HTML y que apunta a *loginProcess*, tras hacer *click* en el botón se enviarán los datos de aquellos *inputs* que contengan la etiqueta *name* al controlador. Además, hay un enlace que te redirige a la página de registro, en caso de no tener una cuenta.

4.2.2 Dar de Alta un Usuario. Dar de Alta la Farmacia Asociada.

La segunda y tercera acción se produce cuando un usuario no tiene una cuenta y quiere darse de alta en el sistema. Para ello se necesitan los siguientes componentes:

- Clase Pharmacy.java: cada usuario viene ligado a una farmacia. Por tanto, es necesario los atributos del modelo de datos de la colección *Pharmacy* y sus respectivos métodos.

- Ampliar la clase *User*.java.

- Ampliar *loginController*.java.

- Vista *signupUser*.

- Vista *signupPharmacy.*

Al igual que con la clase User.java, creamos otra para las farmacias, *Pharmacy*, con los atributos de tipo *String name*, *address*, *CIF*, *mail*, *_id*, que hacen referencia al nombre de la farmacia, la dirección, el CIF, el email y el identificador único respectivamente, De tipo *int* se dan de alta: *phone*, *hospital* y por último de tipo *double*, *longitude y latitude*. Esto dos últimos atributos se dan de alta debido a que se va a utilizar la librería de Google de geolocalización, para en posteriores Sprint poder mostrar las farmacias por cercanía.

Dentro de esta clase se implementa la función de registrar una farmacia, *signupPharmacy*(_id, *address*, *name*, *hospital*, *phone*, e*mail*, cif, *longitude*, *latitude*,, *database*). El objetivo de esta función es comprobar si el CIF está de registrado, a través de una consulta a Mongo y almacenando el resultado en un *FindIterable*. Si este no devuelve datos, es decir su contenido es nulo, se procede a través de una sentencia *insertOne*, función propia de la variable *MongoDatabase*, a introducir un documento en la colección, con los datos pasados como parámetros. Además, liga el identificador de la farmacia con el usuario que tiene relacionado, actualizando así el correspondiente documento de la colección *users*. Para ello, se utiliza la función *findOneAndUpdate* propia también de la variable *MongoDatabase* y se busca bajo la clave común que es el email, antes de introducir el id de la farmacia, En líneas de futuro, para permitir múltiples usuarios a una farmacia, será necesario crear un array de Usuarios dentro del documento de la farmacia.

La clase *User*.java debe ampliarse con una función similar a *signIpPharmacy* pero con los atributos propios de *User* y sin la necesidad de relacionar el usuario con la farmacia.

A continuación, en el controlador, *LoginController* deben añadirse las peticiones al servidor y respuesta propias para el registro de la farmacia y el usuario. Las peticiones que no trabajan con datos para enviar o recibir, es decir, retornan solo una vista, como *signUpUaer* y *signUpPharmacy* se hacen a través del método *GET*, sin embargo, las que llevan un modelo de datos asociado, como *signUpPharmacyProcess* y *signUpUserProcess*, se envían por método POST para así evitar que a través de la URL se puedan mandar datos y corromper la aplicación. Es obligatorio que se completen todos los campos. Si no, salen marcados en rojo y no permite continuar, esto se controlar a través de la vista.

Las vistas que se han creado son *signUpUser*,jsp y *signUpPharmacy*.jsp, ambas tienen la misma estructura de formulario, con los input de cada tipo de dato y el botón de *submit* para enviar los datos al controlador. Estas vistas, trabajan con el *ModelAtributte* que tienen los controladores

pasados como parámetros, por ello, en nombre de los *inputs* debe coincidir con los atributos de la clase. Para poner que un campo es obligatorio está la etiqueta *required*.

En la vista de *signUpPharmacy.jsp* además, se ha implementado un script que utiliza la API de Google Maps para permitir el autocompletado de la dirección y guardar los parámetros de longitud y latitud de la dirección introducida. Este script se base en una función que inicializa los valores de longitud y latitud a -115.141000 y 36.169648, tal y como indica la documentación, y otra variable que llama al *geocder* propio de la librería de *Google Maps*. A continuación, con la función *autocomplete* se llama al *input* del formulario que se quiere autocompletar con las direcciones de *Maps.* Una vez que la dirección ha sido introducida, la librería convierte la dirección en dos coordenadas, que se almacenan en los *inputs* de latitud y longitud declarados en el formulario.

Para poder usar la librería de Google primero hay que darse de alta en su web, solicitar los servicios que se necesitan y el sistema genera una clave que debe ser importada en el proyecto.

4.2.3 Pruebas.

Las pruebas realizadas para comprobar el correcto funcionamiento de estos procesos han sido pruebas individuales por función y luego en conjunto.(Véase Tabla 60**).**

ID	ENTRADA	SALIDA
1.	Mail y contraseña correctos.	Redirige a página principal.
2.	Mail correcto, contraseña incorrecta.	Redirige a página de control.
3.	Mail incorrecto, contraseña correcta.	Redirige a página de control.

Tabla 60 Pruebas unitarias Sprint 1.

Las pruebas realizadas para comprobar el correcto funcionamiento de dar de una alta un usuario y una farmacia han sido las de la Tabla 61.

ID	ENTRADA	SALIDA
1.	Registro de usuario nuevo, sin datos repetidos.	Redirige a la página de registro de farmacia.
2.	Registro de usuario con datos repetidos.	Sale un aviso de error en una nueva ventana y redirige al registro. .
3.	Registro de farmacia con CIF existente.	Sale un aviso de error en una nueva ventana y redirige al registro. .
4.	Registro de farmacia sin completar todos los campos.	Se mantiene en la página y se marca en rojo.
5.	Registro de farmacias con todos los campos y sin datos existentes.	Redirige a la página principal.

Tabla 61 Pruebas unitarias Sprint 1 parte II.

Una vez hecho el desarrollo y las pruebas el Sprint queda completado.

4.3 Sprint 2

Tras la finalización del primer Sprint se evalúa cuáles han sido los resultados y el tiempo de desarrollo final. Se ha podido comprobar que la estimación de tiempos ha sido correcta. Sin embargo, ha sido necesario una preparación y aprendizaje previo sobre Sprint y la estructura de programación web para poder ajustarse a los tiempos marcados.

Llevado a cabo el registro y *login* es posible entregar una parte de la aplicación completa y funcional. A continuación, el siguiente paso es el análisis del *product backlog* para definir las tareas e historias de usuario del Sprint 2.

En este caso se han escogido por orden de relevancia y tiempo las siguientes historias definidas en Tabla 62, Tabla 63, Tabla 64, Tabla 65).

Horas máximas	40
Horas utilizadas	40,5
Horas Libres	0
Historia	Horas estimadas
HU01	30
HU09	10,5

Tabla 62 Resumen de tareas Sprint

Historias de Usuario	Miembros A	B	Estimación media	Prioridad
HU01	20	40	30	Alta
HU02	13	20	16,5	Medio
HU04	8	13	10,5	Alta
HU05	20	8	14	Alta
HU06	5	20	12,5	Media
HU07	8	20	14	Media
HU08	40	40	40	Media
HU09	8	13	10,5	Media
HU12	5	5	5	Media
HU13	20	13	16,5	Alta

Tabla 63 Product backlog Sprint 2.

Historia de Usuario	
ID	HU01
Nombre	Introducir y gestionar automáticamente el stock.
Prioridad	Alta
Riesgo	Alto
Descripción	Como farmacéutico quiero poder meter de manera rápida todo mi stock de productos. Es decir, poder subir varios productos a la vez.
Validación	- Quiero poder introducir grandes cantidades de stock a través de una importación. - Quiero poder ver los detalles del producto, como el laboratorio, el nombre, el código nacional o las unidades. - Los datos almacenados se pueden modificar posteriormente.

Tabla 64 Historia de usuario 1.

Historia de Usuario	
ID	HU09
Nombre	Trabajar con hojas de cálculo
Prioridad	Media
Riesgo	Bajo
Descripción	Como usuario quiero importar y exportar datos desde hojas de cálculo.
Validación	- Quiero importar datos desde un fichero .xls

Tabla 65 Historia de usuario 9.

El desarrollo de este Sprint lleva consigo las siguientes tareas:

- Desarrollo de una página principal donde poner las opciones disponibles; subida de stock y gestión de stock.

- Creación de un servicio de alta de stock a través de la importación de datos mediante un fichero Excel.

- Creación de un servicio que permita visualizar el stock introducido por una farmacia.

- Creación de un servicio que permita modificar las unidades disponibles de un producto.

- Creación de un servicio que permita eliminar productos del stock de una farmacia.

- Interfaz del servicio de visualización de stock.

- Interfaz del servicio de edición del stock.

- Interfaz del servicio de importación de stock.

Para ello es necesario crear o ampliar las siguientes clases y vistas:

- *Pharmacy.java*

- *Item. Java*
- *importStockController. Java*
- *showStock.jsp*
- *editStock.jsp*
- *welcome.jsp*

4.3.1 Desarrollo de una Página Principal

La primera tarea es el desarrollo de una página principal donde estarán las diferentes opciones disponibles de la plataforma. Esta página es el índice del sistema y a la cual se puede acceder desde otras ventanas. A medida que el proyecto avance, esta página irá ampliando sus funciones, llegando a mostrar las reservas, el perfil de usuario o las búsquedas. Esta vista corresponde a la página de bienvenida si el usuario ha hecho el *login* correctamente.

Se crea la página *welcome.jsp* con la ayuda de la librería *BootStrap* para el diseño de la web, de igual modo que en el Sprint anterior. (Véase Figura 6). Esta vista está formada principalmente por componentes que dividen la página en secciones o componentes independientes y un formulario por cada servicio disponible con un botón de tipo *submit* para redirigir a la acción solicitada, En el caso de que la acción solicitada se envíe a través del método *post*, este debe especificarse con la etiqueta *method* dentro del *form.*

Cada botón está formado por un formulario independiente, donde el parámetro *action* apunta a la función del controlador. Esta función le dice qué tiene que hacer una vez presionado el botón. En concreto, para el botón de subir stock, el *action* apunta a la función que se llama *stockManagement* y trabaja sobre un modelo y vista llamado *automaticStockManagement*. Cuando se pulse el botón "Sube tu stock" el sistema accederá a la vista que se llama *automaticStockManagament* y creará los objetos que se definen en *stockManagement*.

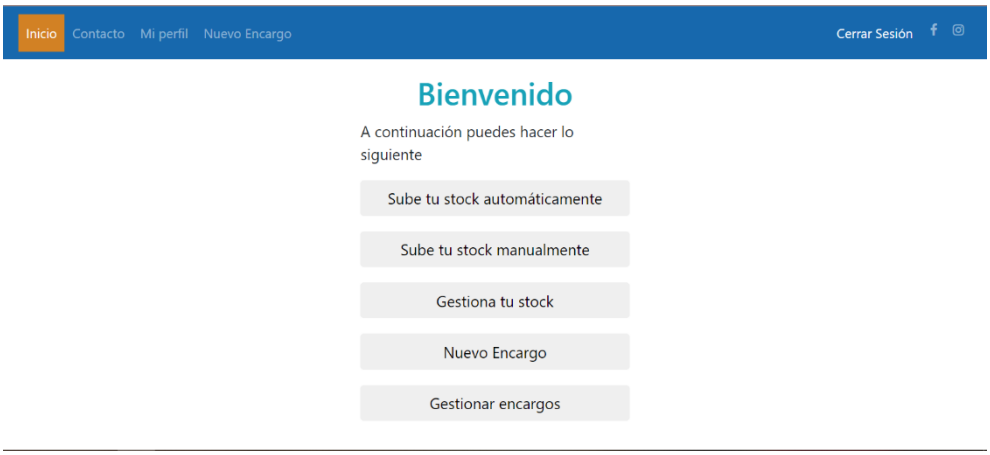

Figura 6 Vista principal del usuario.

El controlador asociado a esta vista se encuentra declarado en la clase *LoginController.java* con el nombre de *main* y con el método *GET y su* función es la de retornar la vista *welcome.jsp* cada vez que se le invoca.

De forma similar, existe otro botón que se encarga de redirigir a la vista de visualizar el stock. Para ello, el *action* está apuntando a la función denominada *showStock* y muestra una vista con el modelo de datos con el mismo nombre.

4.3.2 Servicio de Alta de Stock a través de una Hoja de Cálculo.

Lo siguiente es poder añadir el stock de manera semi automática subiendo a la plataforma un fichero con formato .xls. Para ello se crea una clase *Item.java* con los mismos atributos que están dentro del modelo de datos, es decir, identificador único, nombre del producto, presentación del producto, código nacional, unidades, laboratorio. Asímismo incluye sus respectivos métodos *getter* y *setter*. Salvo las unidades que se han tratado como *int*, el resto de los atributos son del tipo *string*.

Además de esos atributos, se ha creado uno denominado *file* del tipo *CommonsMultipartFile* que almacenará una copia del fichero subido a local.

Una vez creados los atributos de la clase se procede a implementar la primera función de la clase que se encarga de la lectura de un fichero y su posterior serialización en un objeto *Item* (Véase Figura 7). Esta función está denominada como *importItems* y se encarga de copiar el fichero subido por el usuario, a continuación, crea una ruta y extrae en una variable de tipo *XSSFWorkbook* el libro que contiene el archivo con extensión de hoja de cálculo que ha introducido el usuario. A continuación, se obtiene la hoja activa del archivo en una variable *XSSFSheet.* Por último, mediante dos bucles se va leyendo cada fila y columna del archivo y si el índice del *iterator* corresponde con alguna de las condiciones del bucle va añadiendo ese dato a un documento con un nombre asociado. En este caso, el archivo Excel debe cuadrar con la plantilla cuya columna 0 es el nombre de medicamento, la columna 1 es el código nacional, la columna 2 el laboratorio y por último las unidades. Otra forma de importación sería por detección del nombre de columna, es decir, leer la cabecera y en función del dato introducido asociarlo a un campo u otro.

Cada documento creado, uno por fila, hay que introducirlo en la tabla *ítems* para tener todos los artículos dados de alta y además añadirlo con sus respectivas unidades a la tabla *stockItem* que representa el stock de la farmacia. Es necesario añadirlo a la tabla de *ítems* para que se puedan ir añadiendo nuevos productos a la base de datos que la plataforma no haya descargado de CIMA.

```
public void importItems(MongoDatabase database, int count, String cif, File file) throws IOException {
                        // leer archivo excel
                        FileInputStream fs1 = new FileInputStream(file);
                        XSSFWorkbook worbook = new XSSFWorkbook(fs1);

                        //obtener la hoja que se va leer
                        XSSFSheet sheet = worbook.getSheetAt(0);
                        //obtener todas las filas de la hoja excel
                        Iterator<Row> rowIterator = sheet.iterator();
                        Row row;
                        // se recorre cada fila hasta el final
                        while (rowIterator.hasNext()) {
                                row = rowIterator.next();
                                //para saber que ha cambiado de línea y crear un documento nuevo
                                if(row.getRowNum() == count) {

                                        //se obtiene las celdas por fila
                                        Iterator<Cell> cellIterator = row.cellIterator();
                                        Cell cell;
                                        Document d = new Document();
                                        //se recorre cada celda

                                        while (cellIterator.hasNext()) {
                                                // se obtiene la celda en específico
                                                cell = cellIterator.next();
                                                if(cell.getColumnIndex() == 0) {
                                                        d.append("name", cell.getStringCellValue());
                                                }
                                                else if (cell.getColumnIndex() == 1) {
                                                        d.append("_id", cell.getNumericCellValue());
                                                        setItemId((int) cell.getNumericCellValue());

                                                }

                                                else if (cell.getColumnIndex() == 2)  {
                                                        d.append("laboratory", cell.getStringCellValue());
```

```
                                        setLaboratory(cell.getStringCellValue());
                                }
                                else if (cell.getColumnIndex() == 3)  {
                                        setunits( (int) cell.getNumericCellValue());
                                }else {
                                        setprice(cell.getNumericCellValue());
                                }


                        }
                        //si hay un elemento duplicado, lo actualizamos y seguimos.
                        try {
                                database.getCollection("items").insertOne(d);
                                stockItemManagement(database, cif, itemId, units, price);
                                count++;
                        } catch(Exception e) {
                                logger.warn("Elemento duplicado.");

                                database.getCollection("items").findOneAndReplace(d, d);
                                stockItemManagement(database, cif, itemId, units, price);
                                count++;
                                importItems(database, count, cif, file);
                        }
                }
        }
```

Figura 7 Código para insertar el stock de la farmacia.

Como medida de control dentro de esa función, previa a la inserción directa en la base de datos, se ha implementado un servicio de consulta de repetidos. Es decir, una vez que ha terminado de leer una fila completa, que representa un documento de Mongo, accede a la colección *ítems* y en el caso que encuentre un resultado no lo añade. Esto se ha implementado a través de una estructura *try-catch* con el método *insertOne* de *mongodatabase*. Si el código nacional a introducir ya existe, lanza una excepción que es atrapada por la sentencia *catch* y la redirige a una actualización del documento ya insertado anteriormente y su posterior inserción en la clase *stockItem.*

La introducción en la colección *stockItem* se hace a través de la función *stockItemManagement*. Esta función se encarga, en primer lugar, de saber si existe el CIF, a través del método *find*, propio de *MongoDatabase*, como se ha comentado anteriormente. En caso de no existir, emite un aviso de error, en una nueva pantalla, con una redirección a la página anterior. Si encuentra un resultado coincidente, se genera un documento (*Document*) que será el que sustituya, con la sentencia propia de *MongoDatabase*, *findAndReplace* al antiguo, dentro de la colección *stockItem*, con los datos pasados como parámetros, las unidades, el CIF y el código nacional. Una vez creado el documento a insertar se crea un filtro de búsqueda a través de un *BasicDBObject* que almacena la pareja de CIF y código nacional. Por último, a través de la sentencia *findOneAndReplace* se reemplaza el antiguo documento que coincide con la pareja de CIF y código nacional. En caso de no encontrar ninguna coincidencia se crea un nuevo documento.

La vista asociada a esta lógica de negocio viene definida por una pantalla desde la cual se pide el CIF de la farmacia y el archivo que contiene el stock.

La vista denominada *automaticStockManagement* está formada principalmente por un título y una descripción, en modo de párrafo, de lo que se puede realizar en esa pantalla. El campo correspondiente al CIF y al examinador de ficheros están creados como un formulario, del tipo *text* y *file* respectivamente. Por último, está el *submit* que recoge los datos, comprueba con el controlador si los datos son correctos y lanza la siguiente vista acorde a los resultados.

El controlador que lanza esta vista, *stockManagement* está dentro de la clase denominada *StockManagementController.java*. Esta clase alberga todos los controladores relacionados con la subida de stock, visualización y edición de este. *StockManagement* tiene asociado el método POST y se encarga de mostrar el modelo y vista descrito anteriormente.

Por otro lado, el *submit* de la vista *automaticStockManagement* está ejecutando el controlador denominado *stockImportProcess*. Debido a que se van a enviar datos de la vista al controlador, el protocolo de envío de peticiones se hará a través del método POST. Esta función tiene varias partes. La primera comprueba que el campo del CIF no está vacío y que la extensión del archivo introducido es del tipo *.xlsx*. De no ser así, un *PrintWriter* lanza un mensaje por pantalla avisando del error de los datos con un enlace para volver a la pantalla anterior. La segunda parte, se encarga de, si los datos son correctos, acceder a los datos de sesión para subir el CIF de la farmacia. A continuación, se crea la conexión con *Mongo* a través de la función *newConnection* implementada en la clase *MongoFactory.java*. Esta función se encarga de abrir la conexión y de acceder a la base de datos. Si esto no falla, se llama a la función *importItems* mencionada anteriormente. Por último, se lanza la vista *succesImport* que avisa de la correcta subida del stock y permite visualizarlo.

4.3.3 Servicio para Visualizar el Stock Introducido por una Farmacia.

Esta tarea se encuentra como uno de los criterios del cliente para poder dar por finalizada la historia de usuario asociada a este Sprint. El servicio anterior permitía una subida del stock a la plataforma y finalizaba con el aviso por pantalla de que el stock se había subido correctamente, permitiendo al usuario visualizar su stock.

Del mismo modo que en anteriores casos, se procede a explicar en primer lugar la parte del *back-end* o las funciones asociadas al *dao*, para luego seguir con la vista y por último el controlador.

La función que permite ver el stock está implementada dentro de la clase *Item.java* y se denomina *showStock.* Para poder mostrar el stock de una farmacia en concreto es necesario acceder a la colección *stockItem* de Mongo y buscar todos los documentos que contengan el CIF que se pasa como parámetro. Para ello, a través de un *Bson* se crea el filtro de búsqueda y se realiza la búsqueda proyectando solo aquellos valores que son necesarios, es decir, las unidades. Esa búsqueda se almacena en un *MongoCursor* que se va a ir recorriendo, realizando lo siguiente: en primer lugar sacar el código nacional del documento actual para poder cotejarlo en la tabla *ítems.* En segundo lugar, saca la información restante como puede ser el nombre o el laboratorio.
En tercer lugar, esa información se almacena en un documento que se va a mapear a un objeto *ítem*. Finalmente, ese objeto se añade a una lista enlazada denominada *stockList*. Esta lista enlazada es devuelta por la función.

La vista que muestra el stock se denomina *showStock*.jsp y además de usar la librería *Bootstrap*, utiliza una librería especial denominada *JavaServer Pages Standard Tag Library*. Esta biblioteca permite introducir sentencias de control dentro del código HTML. La tabla utiliza el componente *datatable* de *Bootstrap* que permite paginarla, marcar índices, buscar y ordenar por los diferentes campos. Este componente se inicializa creando un script después del código HTML que llame a ese objeto con el nombre de la tabla y personalizar los campos de lenguaje y número de ítems por página. Para que el componente funcione correctamente, el número de columnas declarada en la cabecera de la tabla debe coincidir con el número de campos por fila que se introduzcan. El contenido de la tabla se vuelca a través de un bucle *forEach* que lee el objeto añadido a la vista. Este bucle recorre la variable *stockList* que se retorna en la función anteriormente explicada. La última columna de la tabla corresponde a la acción de borrar, mientras que, para poder editar la cantidad de unidades de un medicamento, existe un botón arriba a la izquierda, al inicio de la tabla. Estas funciones serán explicadas en futuros *Sprints*.

El controlador que lanza esta vista está denominado *showStock* y se encarga de crear la conexión con la base de datos, mediante la función mencionada anteriormente, *createConnection* y además de, llamar a la función *showStock* de la clase *item*. Esta última función devuelve una lista enlazada del stock de la farmacia, añadida como objeto al *ModelAndView* para que pueda ser extraído por la vista.

4.3.4 Servicio para Editar el Stock.

En ocasiones el stock de una farmacia puede variar levemente y no es necesario volver a subir todo el stock a la plataforma. O incluso puede haber casos en los que la farmacia no quiere poner todas las unidades disponibles de un producto en la plataforma. Para ello, es necesario poder editar las cantidades disponibles de un producto. Se ha limitado a que solo se puedan cambiar las unidades ya que el código nacional o descripción del producto no pueden ser modificados por el farmacéutico para mantener la estandarización de los nombres oficiales de CIMA.

Una de las funciones asociadas a este servicio se encuentra implementada dentro de la clase *item.* Trabaja con una pareja de CIF y código nacional que se pasan como parámetros, en una estructura *List<BasicDBObject>* que permite simular los documentos embebidos de Mongo. Con la declaración de esta estructura y a través de una sentencia *findOneAndUpdate,* se accede a la colección *stockItem* actualiza el documento con las unidades que también se le pasan como parámetro.

Además, se implementa una función que simula un buscador por código nacional. Para ello, se ha implementado una función denominada *searchStcokItem*, dentro de la clase *Item.* Esta función, a través del código nacional y del CIF, almacenado en los datos de sesión, busca en la tabla *stockItem* si ese producto está dentro de su stock.

La vista que contiene la tabla de edición se puede dividir en dos partes, la primera, es un buscador, desde el cual se le pide al usuario que introduzca el código nacional del producto que quiere modificar. El buscador, es una etiqueta al lado de un input, de tipo *text,* con un botón *submit,* que permite mandar los datos al controlador y realizar la búsqueda. La segunda parte, y siguiendo la estructura de la vista *showStock,* muestra en una tabla los datos del producto a editar. Sin embargo, en este caso, hay que añadir elementos *input* ocultos en las casillas donde no se van a editar los datos. En las casillas de las unidades, hay que añadir un *input,* de tipo numérico, que permite guardar el valor introducido por el usuario. Además, gracias a la etiqueta *placeholder,* se muestra el valor actual de las unidades que hay disponible en el sistema. Por otro lado, se obliga a que el valor mínimo a introducir sea igual o mayor a 0 con la etiqueta *min*. Por último, a través de un botón *submit* y el método *POST* se envía la información al controlador para que actualice el stock y vuelva a la pantalla de *showStock* con los datos actualizados.

El servicio para editar el stock, tiene dos controladores asociados, uno denominado *editStock* que se encarga de lanzar la vista anterior y el controlador *editStockProcess* que se encarga de recibir

los datos de la vista a través del método *POST*, llamar a la función de *editStock* si la entrada es distinta de cero y a la función de eliminar un producto que se detallará en el siguiente punto, si la entrada introducida es 0. Por último, devuelve la vista *showStock.*

4.3.5 Servicio para Eliminar un Producto del Stock.

Este servicio se podría haber incluido dentro del propio de editar el stock, pero se ha decidido separarlo ya que está formado por funciones distintas.

La función asociada al *dao item* tiene el nombre de *deleteItem* y se encarga de buscar y eliminar a través de la pareja CIF y código nacional pasadas como parámetros. Esta función, hace uso del método propio de *MonogDatabase*, *deleteOne* que permite eliminar el documento que se le pase como parámetro. de la colección elegida por el desarrollador.

El servicio para eliminar un producto del stock, no tiene asociada ninguna vista en concreto. En este caso, desde la propia vista de *showStock* se pasa a través del método *GET* y la URL el código nacional del producto que se quiere eliminar.

El controlador asociado a este servicio, denominado *deleteItem*, se encarga de volver a la página de stock una vez se ha eliminado el producto, llamando a la función correspondiente. Este controlador tiene la etiqueta de *pathVariable* utilizada con el método *GET* cuando es necesario pasar un dato a través de la URL, en este caso el código nacional del medicamento a borrar.

4.3.6 Pruebas

Una desarrollado todos los servicios e implementadas las vistas, es necesario probar el sistema antes de por terminado este *Sprint*. Para ello, se van a realizar pruebas unitarias, con stocks ficticios. Es decir, el nombre de los campos será “códigonacional1”, “nombre1”, “laboratorio1”.

Pruebas para el servicio de alta de stock a través de una hoja de cálculo. Véase Tabla 66.

ID	ENTRADA	SALIDA
1.	Inserción de documento .xlsx correcto.	Muestra aviso de stock subido correctamente. Inserta los documentos en ambas colecciones.
2.	Inserción de un documento duplicado.	Muestra aviso de stock subido correctamente. No inserta en ninguna colección.
3.	Inserción de un documento con campo unidades variado.	Muestra un aviso de stock subido correctamente. Actualiza el documento correspondiente en la colección *stockItem*.
4.	Inserción de un documento correcto sin CIF.	Sale un aviso que es necesario introducir un CIF y redirige a la subida de stock.

Tabla 66 Pruebas realizadas para el servicio de gestión del stock.

Pruebas para el servicio de editar Stock. Véase Tabla 67.

ID	ENTRADA	SALIDA
1.	Editar la cantidad del producto cn1 a 7 unidades.	Muestra el stock con la tabla actualizada. El documento de la colección *stockItem* asociado, se actualiza.
2.	Editar la cantidad del producto cn2 a 0 unidades.	Muestra aviso de stock subido correctamente. No inserta en ninguna colección.
3.	Inserción de un documento con campo unidades variado.	Muestra un aviso de stock subido correctamente. Actualiza el documento correspondiente en la colección *stockItem*.
4.	Inserción de un documento correcto sin CIF.	Sale un aviso que es necesario introducir un CIF y redirige a la subida de stock.

Tabla 67 Pruebas realizadas para el registro de usuarios.

4.4 Sprint 3

En el punto actual, el usuario ya puede registrarse, iniciar sesión y gestionar su stock. Con el análisis del Sprint anterior, se ha concluido el exceso de tiempos destinados a la realización de ese Sprint. El motivo principal ha sido el desconocimiento del uso de las herramientas, principalmente en el uso de la librería *BootStrap* para el diseño de las vistas.

A continuación, se definen las historias de usuario que se van a realizar en este *Sprint.* Del *producto backlog* restante se van a elegir las tareas que tengan prioridad alta. (Véase Tabla 68, Tabla 69, Tabla 70, Tabla 71, Tabla 72).

Horas máximas	40
Horas utilizadas	37
Horas Libres	3
Historia	Horas estimadas
HU04	10.5
HU05	14
HU06	12.5

Tabla 68 Resumen de tareas sprint 3

Historias de Usuario	**Miembros**		**Estimación media**	**Prioridad**
	A	**B**		
HU02	13	20	16,5	Medio
HU04	8	13	10,5	Alta
HU05	20	8	14	Alta
HU06	5	20	12,5	Media
HU07	8	20	14	Media
HU08	40	40	40	Media
HU12	5	5	5	Media
HU13	20	13	16,5	Alta

Tabla 69 Producto Backlog del Sprint 3.

Historia de Usuario	
ID	HU04
Nombre	Realizar búsquedas de fármacos
Prioridad	Alta
Riesgo	Alto
Descripción	Como farmacéutico quiero poder buscar un medicamento por código nacional y que me indiquen las farmacias hospitalarias que disponen de él y sus unidades.
Validación	- Quiero buscar por CN. - Quiero que muestre un listado de farmacias que disponen de ese medicamento. - Quiero ver las unidades disponibles.

Tabla 70 Historia de usuario 4.

Historia de Usuario	
ID	HU05
Nombre	Reservar un producto
Prioridad	Alta
Riesgo	Alto
Descripción	Una vez realizada la búsqueda debo poder seleccionar una farmacia y poder reservar el número de unidades que necesito.
Validación	- La farmacia debe poder reservar las unidades de un medicamento a otra farmacia. - La farmacia origen debe poder realizar el papeleo de préstamo a través del sistema. - La farmacia destino debe ser notificada de la reserva.

Tabla 71 Historia de usuario 5

Historia de Usuario	
ID	HU06
Nombre	Notificación de los cambios de estado.
Prioridad	Media
Riesgo	Media
Descripción	El sistema debe permitir al usuario poder cambiar el estado de la reserva. Posibles estados: solicitado, procesando, enviado, finalizado y cancelado.
Validación	- El farmacéutico debe ser capaz de saber cómo se encuentra una reserva realizada o emitida. - Se debe poder modificar el estado de manera manual. - El estado pendiente, es el estado inicial y automático al realizar una reserva. A su vez, el estado finalizado solo puede ponerlo quien solicita la reserva una vez que recibe el producto. - El estado cancelado debe poder activarse en cualquier momento.

Tabla 72 Historia de usuario 6

Las tareas que llevan asociadas estas historias de usuario son:

- Subir a la plataforma el listado de medicamentos de uso hospitalario sacado de CIMA.
- Añadir un nuevo botón de encargos a la página principal *main*.
- Servicio que permite buscar en el stock de las farmacias por código nacional.
- Servicio que liste las farmacias disponibles con el stock disponible.
- Interfaz que permita visualizar las farmacias.
- Interfaz de buscador.
- Servicio que inserte las nuevas reservas en la base de datos.
- Servicio que permite extraer los datos para completar el formulario de préstamo.
- Interfaz para visualizar los datos de la reserva.

- Servicio de notificación de los nuevos encargos.

- Servicio para permitir los cambios de estado de los encargos.

Para poder realizar estos procesos se deben crear o actualizar las siguientes clases o vistas:

- *Booking.java*

- *Item.java*

- *Mail.java*

- *BookingsController.java*

- *MailController.java*

- *MongoDataConverter.java*

- *MailServie.java*

- *Welcome.jsp*

- *newBooking.jsp*

- *newBookingProcess.jsp*

- *bookingConfirmation.jsp*

- *bookingStatus.jsp*

- *showBookings.jsp*

- *showBookingsOrigin.jsp*

- *showBookingsTarget.jsp*

Antes de comenzar con este Sprint es necesario sacar un listado de medicamentos de uso hospitalario que debe ser introducido en la plataforma. Este listado se extrae de la base de datos de Ludafarma. Para líneas de futuro, se podría implementar un proceso que conectase con el API de CIMA y que cada determinado periodo de tiempo extraiga los nuevos medicamentos dados de alta y los inserte en la base de datos. Esta tarea no se explica ya que no implica un desarrollo propio.

4.4.1 Servicio para Realizar Búsquedas.

El primer paso para realizar una reserva es poder seleccionar qué producto necesitas y en qué farmacia se va a realizar la reserva. En este caso, las farmacias que se muestren van a estar ordenadas por cercanía.

El buscador se va a implementar con el método definido en la clase *item.java searchItemPharmacies.*. Este método recibe como parámetros: *un MongoDatabase* para poder conectar con la base de datos, un *int* que hará referencia al código nacional del producto y dos datos de tipo *double* que harán referencia a la longitud y latitud de la farmacia origen. Dentro de la función se crea la variable *ObjectMapper* para convertir el documento que se extraiga de Mongo en un objeto de la clase *item.* Además, como en Sprint anteriores, a través de *MongoDatabase* y *MongoCollection* se accede a la colección *stockItem.* En este caso, el filtro de búsqueda que se va a crear es sencillo. Por tanto, no se necesita crear ninguna estructura auxiliar, directamente se llama a la sentencia *find,* a través de *MongoDatabase,* pasándole como criterio de búsqueda el código nacional del producto. El resultado de esta consulta se almacena en un *MongoCursor.* Lo siguientes es recorrer la variable que almacena la consulta con un bucle *while.* En cada iteración, se convierte el documento en un objeto *item* gracias a *ObjectMapper* y se almacenan los atributos de unidades y CIF en variables, a través de los métodos *getter* y *setter* de la clase. Una vez conocido el CIF y utilizando este como filtro de búsqueda, se accede a la colección *pharmacies* y se obtiene el nombre y las coordenadas de longitud y latitud correspondientes a ese CIF. A continuación, se invoca al método *distanceBetweenPharmacies*. Esta función, permite calcular la distancia en línea recta entre dos farmacias y se almacena su valor en otra variable llamada *distance.* Con todos los datos calculados y almacenados en sendas variables, se añaden a un *TreeMap* denominado *availablePharmacies* cuya estructura es <*String, Double[]*> donde la clave es el nombre de la farmacia destino, y el valor es un array que guarda en su primera posición la distancia y en su segunda posición las unidades. Al ser una estructura

TreeMap, es decir, una estructura de tipo árbol, con parejas de clave y valor, cada inserción que se hace en el árbol, en cada iteración, se inserta ordenadamente en función de la distancia, permitiendo así almacenar las farmacias ordenadas por cercanía. Este árbol de farmacias es devuelto por la función.

El método auxiliar *distanceBetweenPharmacies*, recibe como parámetros dos pares de coordenadas y devuelve mediante la fórmula de cálculo de distancia entre segmentos, la distancia entre las dos farmacias. (Véase Figura 8).

$$\sqrt{(a2 - a1)^2 + \ (b2 - b1)^2}$$

Figura 8 Cálculo de distancia en línea recta.

La vista del buscador se denomina *newBooking.jsp* y al igual que en *Sprint* anteriores, utiliza la librería *Bootstrap*, *Jquery* y jsp. Esta vista tiene tres divisiones, el título, el buscador y la tabla de resultados. El buscador se implementa con un *form* + *input* + *label* cuya *action* apunta a controlador *searchItem.* La tabla utiliza el componente *datatabke* para dar formato a la tabla y paginarla. Las columnas son el nombre de la farmacia, las unidades disponibles y un botón de acción que permite reservar el producto en esa farmacia. El contenido de la tabla, del mismo modo que en otras vistas, se imprime a través de un bucle *forEach* que recorre el *TreeMap* que devolvía la función de la lógica de negocio. Debajo de la etiqueta de cierre de la tabla, se coloca un *input* de tipo *hidden* para almacenar el nombre de la farmacia que selecciona el usuario. Como particularidad, el botón de acción tiene una etiqueta llamada *onClick* que simula un botón de tipo *submit* pero programable por el desarrollador. En este caso, a la etiqueta se le pasa el nombre de la función *submitForm* más el parámetro que va a necesitar, que será el nombre de la farmacia.

La función *submitForm* (Véase Figura 9*)* se declara en un *script* de *Jquery* debajo del código HTML. Esta función se encarga de almacenar el valor que se la pasa como parámetro y enviarlo al controlador. El valor que se le pasa como parámetro es el valor que tenga el *input* de tipo *hidden* introducido en el código *HTML*. En este caso, es necesario almacenar el producto que ha solicitado el usuario, así como la farmacia que ha seleccionado en la vista. El identificador de los inputs que se declaren para extraer los datos se denominará *itemId*, y se asocia a una variable externa, pues el dato ha sido introducido por el usuario, y, *target*, pasado como parámetro de la función.

```
<script>

var itemId = ${itemId};
function submitForm(value) {
$("#target")[0].value = value; //cuando los datos vienen del controlador
$("#idItemSend")[0].value = itemId;
$("#formId").submit();

}
</script>
```

Figura 9 Extracción de datos personalizada.

Los controladores asociados a esta vista son dos y se crean dentro de la clase *BookingController.java*. El primero de ellos, *newBooking*, se encarga de enviar la vista al usuario. El segundo controlador, *newBookingProcess* recibe los datos introducidos por el usuario y se encarga de llamar al método para listar las farmacias. Este controlador, además, saca el email de los datos de sesión del *SecurityContextHolder* (se explicará en un futuro *Sprint)*. Con estos datos, se relaciona el usuario con la farmacia que tiene asociado y se sacan los parámetros de longitud y latitud que necesita la función arriba mencionada. El *TreeMap* devuelto por la farmacia debe ser añadido como un objecto al *ModelAndView* que retorna el controlador.

4.4.2 Servicio para Hacer Reservas.

En primer lugar, se crea una nueva clase denominada *Booking.java* que hará referencia a la colección en Mongo con el mismo nombre. Dentro de esta clase, se definen atributos con sus métodos *getter y setter*. Los atributos de tipo *string son los siguientes, origin, target, bookingId, status, address* y *description.* Estos campos hacen referencia a la farmacia origen, el destino, el número de reserva, el estado de la reserva, la dirección de entrega de la reserva y el nombre del producto respectivamente. Los atributos de tipo *int* son: *itemId, units* y *phone* correspondientes al código nacional del producto, las unidades que se quieren reservar del mismo y el teléfono de la farmacia. Por último, de tipo *date,* se crea *createdDate* hace referencia a la fecha de creación del encargo. Este último atributo se ha marcado con la etiqueta *JsonDeserializer(using: MongoDataConverter.java)* que permite deserializar el tipo *Date* en un formato de ser entendido por Mongo.

La clase *MongoDataConverter.java* es una clase auxiliar que hereda de la clase *JsonDeserializer*. Contiene una variable estática y final del tipo *SimpleDateFormat* con la que se indica el tipo de formato en el que debe estar la fecha (dd-mm-aaaa). Además, se sobrescribe el método *deserialize* para que tome el campo *date* de la colección de Mongo como un tipo *long* y convertirlo al tipo *Date* de java.

Antes de registrar un nuevo documento en Mongo, en la colección de *bookings,* es necesario extraer los datos que se van a necesitar en la reserva y mostrarlos para la confirmación del usuario. Para ello, se implementa la función *searchBookingInfo* que recibe, un *booking.java,* dos *strings* con el nombre de la farmacia y el email, un *int* con las unidades y un *MongoDatabase*. Se accede a la colección *pharmacies* con filtro de búsqueda el nombre de la farmacia destino, se extraen los datos de dirección y teléfono y al objeto *booking* pasado como parámetro se le fijan los atributos mencionados. Se repite el proceso de extracción de datos, para la farmacia origen. Por tanto, el filtro de búsqueda en la colección será, en este caso, el email sacado de los datos de sesión. Se fijan en el *booking* los atributos de farmacia origen, el estado se pone a solicitado y por último la fecha de creación se ancla con la hora del sistema.

Una vez comentados los atributos y funciones auxiliares de la clase, se procede a explicar la función principal asociada al servicio de reserva de medicamentos (Véase Figura 10). Este método se ha denominado *makeBooking,* recibe *MongoDatabase,* tres *String* de farmacia origen, destino y el estado de la reserva, dos *int*, uno del código nacional y otro de las unidades. Dentro de esta función se invoca a otra que genera el identificador del encargo, llamada *setId* que devuelve un identificador de 5 cifras. A continuación, se comprueba que ese identificador devuelto sea único y no exista en la colección *bookings* a través de un bucle *while* y su posterior consulta con la sentencia *find*. También es necesario controlar que el número de unidades introducidas por el usuario no supera la cantidad máxima disponible por esa farmacia. Eso se puede hacer en la vista con una etiqueta, pero en este caso se realiza a través de la lógica. Por tanto, en primer lugar, se saca a través del nombre de la farmacia destino y la colección *pharmacies* el CIF de la farmacia. Con el par de CIF y código nacional se crea un documento embebido con *DBBasicObject* y se filtra en la colección *stockItem*. De ser mayor el dato introducido por el usuario la función devuelve false y el controlador emite un mensaje de error. De ser correcto y teniendo ya los datos que van a ir en el encargo, a través de la sentencia *insertOne,* función propia de *MongoCollection,* se registra el documento en la base de datos dentro de la colección *bookings.*

```
public boolean makeBooking(MongoDatabase database, String origin, String target, int idItem,
                    int units, Date createdDate,
                    String status) {
        MongoCollection<Document> bookings = database.getCollection("bookings");
        setBookingId(setId());
        //comprobamos que el bookingId no esté ya registrado.
        FindIterable bookingIds = bookings.find(Filters.eq("bookingId", bookingId));
        //comprobamos si el resultado devuelto por find es un documento o está nulo.
        while ( bookingIds.iterator().hasNext()){
                setBookingId(setId());
                bookingIds = bookings.find(Filters.eq("bookingId", bookingId));
        }

        //sacamos los datos correspondientes a la farmacia destino
```

```
                MongoCollection<Document> pharmacies = database.getCollection("pharmacies");

                FindIterable<Document> pharmacyInfo = pharmacies.find(Filters.eq("name", target));

                Document doc = (Document) pharmacyInfo.first();
                String cif = (String) doc.get("cif");

                //se controla que las unidades introducidas no sean superiores al máximo disponible por la farmacia
destino.
                MongoCollection<Document> stockitems = database.getCollection("stockItem");
                Document idStockItem = new Document();
                idStockItem.append("cif", cif).append("itemId", idItem);
                FindIterable<Document> stockItem = stockitems.find(Filters.eq("_idStockItem", idStockItem));

                Document doc2 = (Document) stockItem.first();
                int unitsStock = (int) doc2.get("units");

                if (units > unitsStock) {
                        return false;
                }else {

                        bookings.insertOne(new Document().append("bookingId", bookingId).append("origin",
origin).append("target", target).append("createdDate", new Date())
                                                .append("itemId", idItem).append("units", units)
                                                .append("status", status));
                        return true;

                }
        }
```

Figura 10 Función para realizar una reserva.

El método *setId*, genera 4 *strings* que hacen referencia a los dígitos del código de referencia con la ayuda de la librería *UUID.randomUUID*. A continuación de cada *string* se quita el guion con la función *replace* y se añaden *StringBuffer*. Esta variable se genera con un *SecureRandom* que permite crear números aleatorios seguros y con poca probabilidad de que salgan repetidos.

Además de la función de generar el encargo, el sistema manda un email de confirmación a la farmacia origen y un email de nuevo encargo a la farmacia destino. Para ello se utiliza el servicio propio de Spring *JavaMailSender*. Se crea una clase *Mail.java* que guardará los datos relacionados con los emails, el asunto y la descripción que se le quiera añadir y sus métodos *getter y setter*, un controlador llamado *emailController.java* y una interfaz que implementará las funciones de la clase *Mail.java*. Debido a que es un email con contenido HTML, se crea una plantilla en un archivo dentro de la carpeta *resources*, para cada tipo.

Previamente a crear las clases que implementarán el cliente de correo, es necesario añadir la dependencia Maven al archivo pom.xml de Spring que será utilizada para enviar los correos: *javax.mail-api* . Además, dentro del archivo de configuración de *Spring, servlet-context.xml* se configura un *Bean* con la configuración del servidor de correo y otro *Bean* con las plantillas de los emails que se van a utilizar.

La configuración del cliente de correo se hace con el servidor de *ludapartners.com* en el puerto 587 y con un protocolo de envío SMTP, para evitar problemas con el protocolo SSL, se desactiva

este protocolo. A su vez, se fija un tamaño máximo del email a enviar, así como de los archivos que se pueden adjuntar. Por último, se especifica dónde se encuentran las plantillas de correo que el cliente va a utilizar. Esta configuración queda reflejado en el código siguiente. (Véase Figura 11).

```
<!-- Spring Email Sender Bean Configuration -->
	<beans:bean id="mailSender"
		class="org.springframework.mail.javamail.JavaMailSenderImpl">
		<beans:property name="host" value="ludapartners.com" />
		<beans:property name="port" value="587" />
		<beans:property name="username"
			value="cassandra.moreno@ludapartners.com" />
		<beans:property name="password" value="******** /> //se oculta por motivos de seguridad.
		<beans:property name="javaMailProperties">
			<beans:props>
				<beans:prop key="mail.smtp.auth">true</beans:prop>
				<beans:prop key="mail.debug">true</beans:prop>
				<beans:prop key="mail.transport.protocol">smtp</beans:prop>
				<beans:prop
key="mail.smtp.socketFactory.class">javax.net.ssl.SSLSocketFactory
				</beans:prop>
				<beans:prop key="mail.smtp.socketFactory.port">465</beans:prop>
				<!-- <beans:prop key="mail.smtp.starttls.enable">true</beans:prop> -->
			</beans:props>
		</beans:property>
	</beans:bean>

	<!-- Spring Email Attachment Configuration -->
	<beans:bean id="multipartResolver"
		class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
		<!-- Maximum Upload Size In Bytes -->
		<beans:property name="maxUploadSize" value="20971520" />
		<!-- Maximum Size Of File In Memory (In Bytes) -->
		<beans:property name="maxInMemorySize" value="1048576" />
	</beans:bean>

	<beans:bean id="fmConfiguration"
		class="org.springframework.ui.freemarker.FreeMarkerConfigurationFactoryBean">
		<beans:property name="templateLoaderPath"
			value="classpath:/resources/templates" />
		<beans:property name="preferFileSystemAccess"
			value="false" />
	</beans:bean>
```

Figura 11 Configuración del cliente de correo.

Tras configurar el servicio de correo, se crea la clase *MailService.java*. A continuación, se explica el código de la siguiente figura. (Véase Figura 12). Esta clase hace uso de las etiquetas *autowired* de *Spring*, para importar las clases *JavaMailSender* y *Configuration.* La primera hace referencia al cliente de mensajería y la segunda a la plantilla que se va a utilizar con código *HTML* y que debe ser seleccionada para cada email. Se declara una función *sendEmail* que recibe como parámetros un *Mail.java* y captura excepciones de entrada y salida (*IOException*) y relacionadas con la configuración de la plantilla *TemplateException*. Esta función comprueba de qué tipo de email es el que debe mandar y se asocia a *Configuration* la plantilla correspondiente. Es necesario declarar un *MimeMessageHelper* inicializándolo con un *MimeMessage* que se crea a través de la clase *JavaMailSender*. Esta variable se encargará de crear el mensaje con contenido *HTML* y a través del parámetro pasado, extraer los datos de emisor, receptor y tipo de email. *MimeMessage*

es una clase que implementa la clase abstracta *Message* y la interfaz *MimePart* para crear mensajes de correo de tipo *MIME* (extensiones multipropósito de correo de internet).

```
@Service("MailService")
public class MailService {

	@Autowired
	JavaMailSender mailSender;

	@Autowired
	Configuration fmConfiguration;


	public void sendEmail(Mail mail) throws MessagingException, IOException, TemplateException {
		MimeMessage message = mailSender.createMimeMessage();

		MimeMessageHelper helper = new MimeMessageHelper(message);

		// Using a subfolder such as /templates here
		fmConfiguration.setClassForTemplateLoading(this.getClass(), "/templates");

		Template t;
		if (mail.getMailType() == 0 ) {
			t =fmConfiguration.getTemplate("email_template.txt");
		}else if (mail.getMailType() == 1) {
			t = fmConfiguration.getTemplate("email_newBooking.txt");
		}else {
			t = fmConfiguration.getTemplate("forgot-password.html");
		}
		String text = FreeMarkerTemplateUtils.processTemplateIntoString(t, mail.getModel());
		helper.setFrom("cassandra.moreno@ludapartners.com");
		helper.setTo(mail.getMailTo());
		helper.setText(text, true);
		helper.setSubject(mail.getMailSubject());
		mailSender.send(message);
	}
}
```

Figura 12 Función para enviar un email con contenido HTML.

La vista asociada, *bookingConfirmation.jsp* tiene una etiqueta de formulario (*form)* principal que engloba a todos los *inputs* de los datos a mostrar, la farmacia origen, la farmacia destino, la dirección de la farmacia destino, el teléfono de la farmacia destino, el código nacional del producto y las unidades de este. Todos los inputs salvo el de las unidades están deshabilitados con la etiqueta *disabled* para evitar que el usuario pueda alterar los datos. Además, el *input* de unidades tiene la etiqueta *min* con valor 1, que impone al usuario a pedir al menos una unidad del producto. Cada *input* está dentro de una etiqueta *div* distinta para así crear un panel como diseño de la página. Por último, está el botón de tipo *submit* que permite enviar los datos al controlador y ejecutar así la reserva.

Los controladores asociados a este servicio son los siguientes: 1) *bookingConfirmation* permite mostrar la vista de los datos de la reserva y completarlos llamando a la función *searhBookingInfo* mencionada arriba. 2) *BookingConfirmationProcess* es el encargado de recoger los datos que se han mostrado en la vista y el dato de las unidades que ha insertado el usuario, para llamar a la función *makeBooking* y ejecutar el registro del encargo en la base de datos. Además, si la reserva

se ha podido realizar correctamente, crea dos objetos de tipo *Mail.java* y le asocia a cada uno el email de la farmacia origen y destino, el tipo de mensaje que va a ser (1 si es confirmación de encargo y 2 si es solicitud de nuevo encargo), el asunto del mensaje personalizado para cada uno. A continuación, con una estructura de tipo mapa, que almacena la clave y el valor, rellena los datos de la plantilla que están personalizados, como son la referencia del *booking* o el email de la farmacia a la que va dirigido. Una vez creados los dos mensajes llama a *MailService* para enviarlos. Por último, dentro de este controlador, con un *printWriter* sale un aviso por pantalla de que el encargo se ha realizado correctamente y avisando que se ha enviado un email de confirmación. En el caso de que haya habido un problema al ejecutar la reserva, porque las unidades son superiores a las disponibles, a través de un *printWriter* se envía un aviso por pantalla comentando el error y con un enlace para volver a hacer la reserva.

4.4.3 Servicio de Cambios de Estado.

Tras realizar el encargo, las farmacias deben poder saber en qué estado se encuentran sus encargos y poder modificarlos cuando así se requiera. Los estados disponibles son: a) solicitado, es un estado automático cuando se genera el encargo, b) procesando, cuando la farmacia lo recibe y lo acepta, c) enviado, cuando la farmacia destino lo ha sacado del hospital, d) finalizado, cuando la farmacia origen lo recibe y e) cancelado cuando finalmente el encargo no pueda ser ejecutado.

A continuación, se crea un método que permite extraer los datos del encargo en función del usuario que esté visualizando la reserva, la farmacia origen o la farmacia destino. (Véase Figura 13). Este método se llama *searchBooking* y recibe un objeto de tipo *Booking* que será el encargo a mostrar, un parámetro de tipo *MongoDatabase* para poder acceder a la base de datos y un atributo de tipo *int* que define si es el origen o el destino. Como es habitual, se accede a la colección *bookings*, con filtro de búsqueda la referencia del encargo que se pasa como parámetro dentro del objeto *Booking*, y se extrae el único documento que concuerda con la referencia. A través de una estructura *if-else* condicionada por el *int* del parámetro, se extraen los datos del encargo como son el origen y el destino del documento y se actualizan en el objeto *Booking* a través de los métodos *set* de cada atributo. Si la farmacia que visualiza el encargo es la farmacia destino, es decir, quien recibe el encargo, se accede a la colección *pharmacies* y se extraen los datos de la farmacia origen como son la dirección o el número de teléfono. Estos datos se actualizan también en el objeto *Booking*. En este caso, el filtro de búsqueda sería el campo *origin* del documento extraído al principio del método. Si la farmacia que visualiza el encargo es la farmacia origen, el proceso es el mismo, salvo que el filtro de búsqueda correspondería con el

campo *target* del documento extraído. Tras obtener los datos de la farmacia, falta actualizar en el objeto *Booking* los campos que corresponden al código nacional del producto (*itemId)*, el nombre (*productName*), las unidades (*units*), el estado (*status*) y la fecha de creación del encargo. Todos estos datos se obtienen directamente del documento extraído de la colección *bookings*, salvo el nombre del producto. Este hay que sacarlo haciendo una nueva búsqueda en la colección *ítems* con el código nacional del producto como filtro de búsqueda.

```
public Booking searchBooking(Booking booking, MongoDatabase database, int type) {

		MongoCollection<Document> bookings = database.getCollection("bookings");

		//sacamos los datos correspondientes a la farmacia destino
		FindIterable<Document> bookingInfo = bookings.find(Filters.eq("bookingId",
booking.getBookingId()));
		Document doc = (Document) bookingInfo.first();

		//información que ve la farmacia que emite el encargo.
		if(type == 0 ) {
			String origin = (String) doc.get("origin");
			String target = (String) doc.get("target");

			booking.setOrigin(origin);
			booking.setTarget(target);

			//sacamos los datos correspondientes a la farmacia que recibe el encargo
			MongoCollection<Document> pharmacies = database.getCollection("pharmacies");

			FindIterable<Document> pharmacyInfo = pharmacies.find(Filters.eq("name", target));

			Document doc2 = (Document) pharmacyInfo.first();
			String address = (String) doc2.get("address");
			int phone = (int) doc2.get("phone");

			booking.setAddress(address);
			booking.setPhone(phone);

		}else { //información que ve la farmacia que recibe el encargo.
			String origin = (String) doc.get("target");
			String target = (String) doc.get("origin");

			booking.setOrigin(origin);
			booking.setTarget(target);

			//sacamos los datos correspondientes a la farmacia que emitió la reserva
			MongoCollection<Document> pharmacies = database.getCollection("pharmacies");

			FindIterable<Document> pharmacyInfo = pharmacies.find(Filters.eq("name", target));

			Document doc2 = (Document) pharmacyInfo.first();
			String address = (String) doc2.get("address");
			int phone = (int) doc2.get("phone");

			booking.setAddress(address);
			booking.setPhone(phone);
		}

		int idItem = (int) doc.get("itemId");
		Date createdDate = (Date) doc.get("createdDate");
		String status = (String) doc.get("status");

		booking.setItemId(idItem);
		booking.setCreatedDate(createdDate);
		booking.setUnits((int) doc.get("units"));
		booking.setStatus(status);

		//sacamos el nombre del producto a buscar.
		MongoCollection<Document> items = database.getCollection("items");
		FindIterable<Document> itemInfo = items.find(Filters.eq("_id", idItem));
```

```
Document doc3 = (Document) itemInfo.first();
String productName = (String) doc3.get("name");

booking.setProductName(productName);
return booking;

}
```

Figura 13 Función para buscar los datos del encargo.

Para controlar los posibles estados del encargo, se crea *editStatus* que recibe dos *strings* con el estado y la referencia del encargo y un *MongoDatabase.* Se crea dos *BasicDBObject, uno* que guardará el campo a modificar, *status*, con el nuevo valor pasado como parámetro y al que se le asigna la etiqueta *set* Esta etiqueta permite que a través de una sentencia *findOneAndUpdate* propio de *MongoCollection*, se actualice solo el campo pasado y no se borren el resto de los campos del documento.

Las vistas encargadas de modificar el estado del encargo se denominan *bookingStatusOrigin.jsp y bookingStatusTarget.jsp. (*Véase Figura 14). Ambas vistas, mantienen la estructura de la vista *bookingConfirmationProcess* que muestra los datos de la reserva antes de confirmarla. La principal diferencia con respecto a esa vista es que se añaden dos campos nuevos y un botón. Los dos nuevos campos hacen referencia al nombre del medicamento reservado y a una etiqueta *select/option* de *HTML*. Esta etiqueta permite mostrar una lista desplegable por pantalla con las diferentes opciones declaradas. Si es la farmacia origen, solo puede cambiar a estado finalizado o cancelado y si es la farmacia destino a estado procesando, enviado o cancelado. El botón de tipo *submit*, tiene un *action,* etiqueta que indica qué controlador está asociado a la vista, apuntando al controlador que actualizará el estado de la reserva.

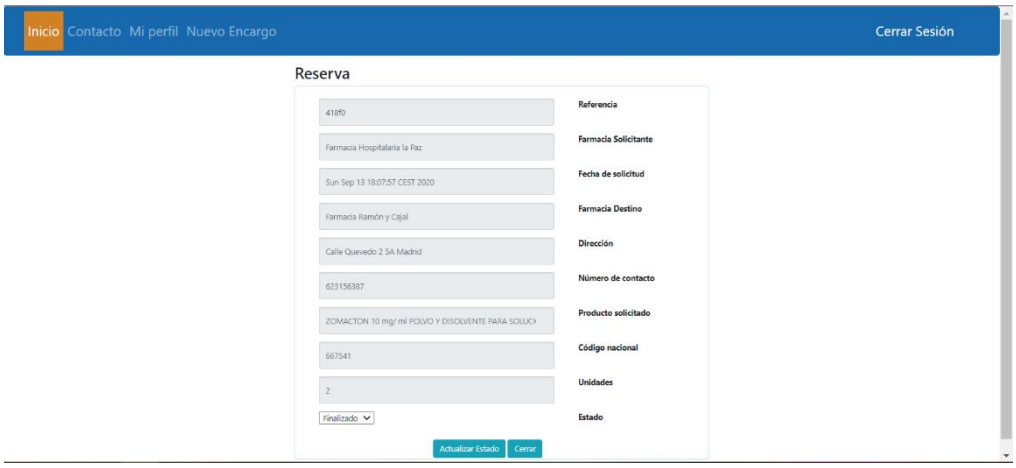

Figura 14 Vista de la gestión de los estados de los encargos.

Los controladores son: *bookingStatusOrigin*, *bookingStatusTarget* y *editStatus*. Los dos primeros se encargan de lanzar las vistas que tienen el mismo nombre, pasándoles los datos que reciben de la función *searchBookings*, y añade la lista devuelta por el método al modelo del controlador. Por otro lado, *editStatus*, se encarga de recoger los datos de la vista, es decir, recoger los datos de la referencia y el estado del encargo, llamar a la función de *editStatus* mencionada previamente y si es correcto, devolver la vista de los encargos con el estado actualizado.

En este caso, las pruebas realizadas para este servicio han sido triviales, por tanto, no se especifican.

Con el desarrollo de este último servicio se da por finalizado el *Sprint* 3 y se procederá en la próxima reunión al análisis de los tiempos y evolución del proyecto.

4.4.4 Pruebas.

A continuación, se resumen las pruebas realizadas para los servicios de búsqueda de productos y realización de un encargo. En este caso, las pruebas ya se pueden realizar con productos reales ya que se ha subido el stock de medicamentos de uso hospitalario a la plataforma.

- Búsqueda de un producto. (Véase Tabla 73 Pruebas para buscar un productoTabla 73).

ID	ENTRADA	SALIDA
1.	Buscar un código nacional disponible en los stocks de las farmacias piloto.	Muestra una tabla con las farmacias ordenadas por cercanía que disponen del producto.
2.	Buscar un código nacional existente en el sistema, pero sin stock.	La tabla no devuelve resultados.
3.	Introducir letras o caracteres en el cuadro de búsqueda.	No permite ningún carácter que no sea un número.

Tabla 73 Pruebas para buscar un producto

- Reserva de un producto. (Véase Tabla 74).

ID	ENTRADA	SALIDA
1.	Introducir un número inferior de unidades disponibles en el cuadro de reserva.	Muestra un mensaje de encargo realizado correctamente.
2.	Introducir un número inferior de unidades disponibles en el cuadro de reserva.	Muestra un mensaje de error donde dice que el número de unidades introducidas no es correcto.
3.	No introducir ninguna unidad.	No deja hacer la reserva, sale un mensaje que el campo es obligatorio.
4.	Introducir una cantidad menor que 1.	No permite introducir signos negativos ni 0.
5.	Introducir un número decimal en la casilla de unidades.	No dejar hacer la reserva, sale un mensaje que el formato no es correcto.

Tabla 74 Pruebas para reservar un producto.

Para el servicio de cambios de estado de los encargos no se muestran las pruebas ya que han sido triviales.

4.5 Sprint 4

Tras el análisis del estado del proyecto, se observa que se ha llegado al ecuador del proyecto, donde las funciones principales y de más alta demanda por el cliente ya están entregadas. Sin embargo, el último *Sprint* trajo una demora considerable en los tiempos de entrega, llegando a extenderse un 50% del tiempo inicial requerido. Eso es debido a que a la hora de realizar la planificación de los tiempos por tarea no se tuvo en cuenta el coste de crear el servicio de mensajería ni se tuvo en cuenta los problemas de relación que podrían darse entre la librería *DataTable* y los *scripts JQuery* para poder introducir tablas dinámicas.

También se han decidido qué tareas de usuario se van a realizar en este *Sprint*: (Véase Tabla 75, Tabla 76, Tabla 77, Tabla 78, Tabla 79).

Horas máximas	40
Horas utilizadas	35,5
Horas Libres	4,5
Historia	Horas estimadas
HU07	14
HU12	5
HU13	16,5

Tabla 75Resumen de tareas Sprint 4

Historias de Usuario	**Miembros** **A**	**B**	**Estimación media**	**Prioridad**
HU02	13	20	16,5	Medio
HU07	8	20	14	Media
HU08	40	40	40	Media
HU12	5	5	5	Media
HU13	20	13	16,5	Alta

Tabla 76 Product backlog para el Sprint 4.

Historia de Usuario	
ID	HU07
Nombre	Poder visualizar las reservas.
Prioridad	Media
Riesgo	Media
Descripción	El sistema debe permitir al usuario poder ver las reservas que tiene activas, así como las reservas que se han completado.
Validación	- El farmacéutico debe poder diferencia entre reservas emitidas y recibidas así como activas y completadas.

Tabla 77 Historia de usuario 7.

Historia de Usuario	
ID	HU12
Nombre	Modificar datos de los usuarios.
Prioridad	Media
Riesgo	Medio
Descripción	Quiero que un usuario puede modificar sus datos y los de su farmacia.
Validación	- Debe poder modificar sus datos personales o los de su farmacia. - Los datos que se pueden modificar son el email del usuario/hospital y el número de teléfono.

Tabla 78 Historia de usuario 12.

Historia de Usuario	
ID	HU13
Nombre	Debe haber un administrador.
Prioridad	Alta
Riesgo	Alto
Descripción	Se completará cuando se realice
Validación	- El administrador debe poder ver todos los encargos y modificar el estado de estos. - El administrador debe poder ver todos los usuarios y modificar los datos de estos salvo la contraseña. - El administrador debe poder ver todas las farmacias adheridas a la red y modificar los datos de estas.

Tabla 79 Historia de usuario 13

Para este Sprint hay que realizar las siguientes tareas:

- Servicio para listar los encargos en función de si son encargos solicitados o recibidos.
- Servicio para mostrar los datos del usuario.
- Servicio para actualizar los datos del usuario como son el nombre o el email.
- Servicio para mostrar todos los encargos de todas las farmacias.
- Servicio para mostrar todas las farmacias adheridas.
- Servicio para mostrar todos los usuarios.
- Servicio para editar los datos de la farmacia como son el email, el teléfono o la dirección.
- Servicio para cambiar la contraseña desde el perfil usuario.
- Interfaz para mostrar los encargos recibidos y solicitados.
- Interfaz para mostrar todas las farmacias.

- Interfaz para mostrar todos los usuarios.
- Interfaz para mostrar todos los encargos.
- Interfaz para mostrar los datos del usuario.
- Interfaz para mostrar los datos de la farmacia.

Las clases que se deben de crear o modificar y las vistas que se van a generar son las siguientes:

- *BookingController.java*
- *AdminController.java*
- *User.java*
- *Pharmacy.java*
- *welcomeAdmin.jsp*
- *showBookings.jasp*
- *showBookingsOrigin.jsp*
- *showBookingsTarget.jsp*
- *showAllBookings.jsp*
- *showAllPharmacies.jsp*
- *showAllUsers.jsp*
- *newPasword.jsp*
- *editProfile.jsp*

4.5.1 Servicio para Mostrar los Encargos.

La función *showBookings* (Véase Figura 15) declarada dentro de la clase *Bookings.java* recibe un *string* que será el nombre de la farmacia, un *int* que diferencia si se muestran los encargos de origen o de destino y un *MongoDatabase* para poder conectar con la base de datos. Accede a la colección *bookigns* y con filtro de búsqueda, el nombre de la farmacia en el campo *origin o target* en función del *int* que se le pasa como parámetro, extrae los documentos en un *MongoCursor*, variable que simula una lista y que puede ser recorrida fácilmente. Cada colección de documentos almacenada en *MongoCursor* se recorre para deserializar cada documento en un objeto *Booking.java* a través de un *ObjectMapper* y se añade a una lista enlazada. Esta función tiene excepciones controladas relacionadas con la extracción del *json* o la deserialización.

```
public  LinkedList<Booking> showBookings(String name, MongoDatabase database, String type) {

		Booking booking;
		ObjectMapper mapper = new ObjectMapper();

		LinkedList<Booking> bookingsList = new LinkedList<Booking>();

		//el filtro aplica para origenes y destinos, se especifica en type.
		Bson filter = Filters.eq(type, name);
		Bson projections = Projections.exclude( "_id");

		MongoCollection<Document> collection = database.getCollection("bookings");

		MongoCursor<Document> pharmacyBookings = collection.find(filter).projection(projections).iterator();
		try {
			//Por cada encargo de la farmacia, buscamos en la tabla items la información.
			while(pharmacyBookings.hasNext()) {
				Document doc = (Document) pharmacyBookings.next();
				String pharmacyName;
				if (type == "origin") {
					//sacamos los datos de la farmacia que lo recibe
					pharmacyName = (String) doc.get("target");
				}else {
					//sacamos los datos de la farmacia que lo emite
					pharmacyName = (String) doc.get("origin");
				}

				MongoCollection<Document> collection2 = database.getCollection("pharmacies");
				Bson filter2 = Filters.eq("name", pharmacyName);

				Document bookingInfo = (Document) collection2.find(filter2).first();
				doc.append("phone", bookingInfo.get("phone")).append("address", bookingInfo.get("address"));

				try {
					String s = doc.toJson();
					booking = mapper.readValue(doc.toJson(), Booking.class);
					bookingsList.add(booking);

				} catch (JsonParseException e) {
					// TODO Auto-generated catch block
					e.printStackTrace();
				} catch (JsonMappingException e) {
					// TODO Auto-generated catch block
					e.printStackTrace();
				} catch (IOException e) {
					// TODO Auto-generated catch block
					e.printStackTrace();
				}
```

```
                    }
                }
                finally {
                    pharmacyBookings.close();
                }
                return bookingsList;
    }
```

Figura 15 Función para mostrar los encargos.

Este servicio tiene varias vistas asociadas, las vistas que se encargan de mostrar los encargos en función de origen y destino y una vista que te permite decidir qué tipo de encargos quieren. Además, hay que añadir un nuevo botón a la página principal (*welcome.jsp*) para gestionar los encargos.

La vista *showBookings.jsp* redirigirá el menú principal, cuenta con dos botones, uno para los encargos recibidos y otro para los encargos emitidos. Además, dispone de una pequeña descripción de lo que debes hacer en esa página.

ShowBookingsOrigin.jsp (Véase Figura 16) muestra en un *DataTable* propio de la librería *BootStrap* todos los encargos emitidos por la farmacia. La estructura de esta vista es similar a la vista *newBooking.jsp* que lista las farmacias que disponen de un producto concreto. En este caso, los datos mostrados son la referencia del encargo, la farmacia destino, las unidades solicitadas, el estado del encargo y un botón en cada fila de detalles que redirigirá al detalle del encargo. *ShowBookingsTarget.jsp* mantiene la misma estructura, pero muestra los encargos que la farmacia ha recibido.

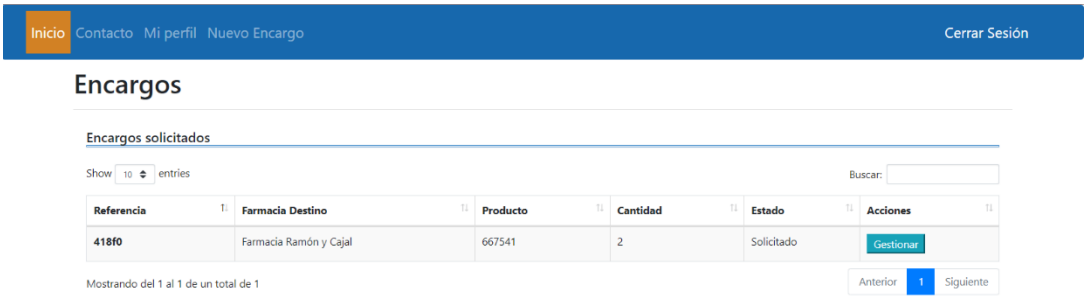

Figura 16 Vista que muestra los encargos solicitados por la farmacia.

Los controladores de este servicio están declarados dentro de la clase *bookingController.java* y son los siguientes: 1) *showBookings* que muestra la vista con el mismo nombre, en la que se elige qué encargos se quieren gestionar, 2) *showBookingsOrigin* y 3) *showBookingsTarget* que llaman a la función *showBookings* del DAO mencionada previamente con el tipo 0 o 1 respectivamente. En ambos controladores se añade el objeto devuelto por la función *showBookings* al modelo para que la vista pueda extraer los datos. Además, de la sesión creada se obtienen los datos del email, para poder buscar en la colección *pharmacies* y sacar los atributos que necesita la función *showBookings* mencionada previamente.

4.5.2 Servicio para Modificar los Datos de los Usuarios.

Los datos que el usuario puede modificar desde su perfil son el nombre del titular de la farmacia hospitalaria, el email que utiliza para recibir las notificaciones e iniciar sesión y la contraseña.

Para los datos no sensibles, es decir, el email y el nombre se crea la función *editUser* dentro de la clase *User.java*. Esta función recibe como parámetros, tres atributos de tipo *strings*, el primero corresponde al antiguo email, el segundo es el email nuevo, en caso de que haya cambiado y el tercero es el nombre de la persona titular. También recibe un *MongoDatabase* para conectarse con la base de datos.

Como en otros métodos implementados anteriormente, la actualización de registros de un documento se hace a través de la función *findOneAndUpdate* a la que se le pasan dos parámetros, uno que es el filtro de búsqueda, en este caso el email antiguo del usuario y el otro parámetro es una variable *DBObject* que almacena el nombre y el nuevo email junto con la etiqueta $*set* que indica a Mongo los registros del documento que debe actualizar sin eliminar los demás.

Para el dato sensible de la contraseña se crea el método *updatePassword* dentro de la misma clase que el anterior, que recibe como parámetros dos de tipo *string* correspondientes a la nueva contraseña y al email del usuario, así como un *MongoDatabase.* En este caso, el dato correspondiente a la contraseña viene cifrado. Por tanto, se repite el proceso que en el método anterior.

Las dos vistas que editan los datos de usuario, llamadas *editProfile.jsp* y *newPassword*.jsp siguen la misma estructura de divisiones y diseño que las vistas relacionadas con los datos de la reserva como es *bookingConfirmationProcess*.jsp. (Véase Figura 17, Figura 18). En este caso los *inputs*

se han adaptado a los datos del perfil de usuario. La vista relacionada con actualizar la contraseña solicita introducir la contraseña antigua para poder cambiarla, El *action* del *form* que hay en cada una de las vistas apunta a los controladores *editProfile* y *newPasswordProcess* que se definirán a continuación. Además, los *inputs* están tipados, siendo obligatorio meter una estructura tipo email en el *input* que hace referencia al email. El *input* de la contraseña está tipado para que no pueda verse lo que se está escribiendo.

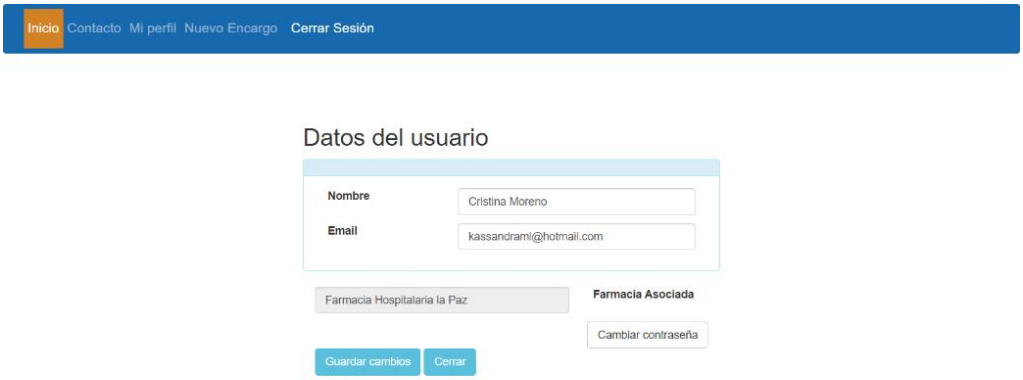

Figura 17 Vista para modificar los datos del usuario.

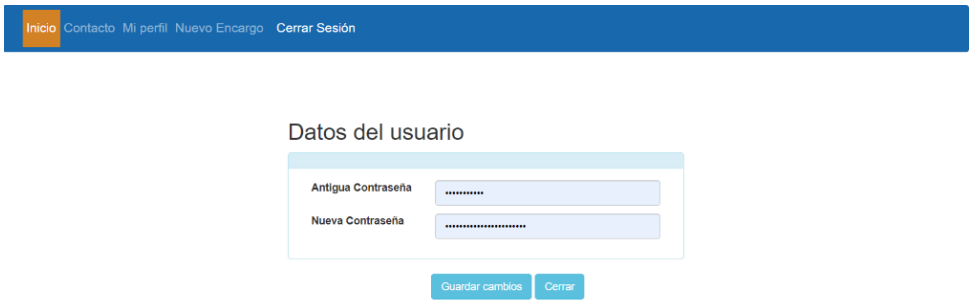

Figura 18 Vista para cambiar la contraseña del usuario.

EditProfile es un controlador que se ha definido dentro de la clase *LoginController.java*. (Véase Figura 19). Este controlador extrae los datos de sesión, a través de una variable *Authentication* y del contexto de seguridad (estos conceptos son de *Spring Security* que se explicarán en futuros *Sprints, Véase* 4.7 Sprint 6.*)*, el email con el que el usuario se ha registrado en el sistema. A continuación, llama a la función previamente mencionada, *editProfile*. Los parámetros que tiene la función se extraen del *bean* que tiene asociado este controlador. Si el resultado es favorable, retorna la vista de la página principal, es decir, *welcome.jsp*, en caso contrario sale una página de error.

```
@RequestMapping(value = "/editProfile", method = RequestMethod.POST)
	public ModelAndView editProfile (HttpServletRequest request, HttpServletResponse response,
			@ModelAttribute("userBean") MyUser userBean) {
		ModelAndView mav = null;
		MongoClient mongoClient = mongo.crearConexion();
		MongoDatabase database = mongoClient.getDatabase("db01");

		mav = new ModelAndView("welcome");

		Authentication auth = SecurityContextHolder
					.getContext()
					.getAuthentication();

		//Sacamos el mail de la sesion para relacionarlo con el stock.
		String email = (String) auth.getPrincipal();

		if(mongo != null){
			userBean.editUser(userBean.getName(), email, userBean.getemail(),
						database);
			logger.info("El usuario se ha modificado correctamente");

			return mav;
		}else {
			logger.error("Conexión con MongoDB no establecida.");
			return mav;
		}

	}
```

Figura 19 Controlador asociado a la función editProfile

NewPasswordProcess tiene un funcionamiento similar pero antes de llamar a la función de *updatePassword* es necesario cifrar la contraseña utilizando el algoritmo MD5. Además, como medida de seguridad, se requiere comprobar la antigua contraseña, por tanto, del *bean* que tiene asociado el controlador, se extrae la contraseña antigua introducida por el usuario, se cifra y se comprueba con la que hay almacenada en el sistema. De coincidir, entonces se llama a la función de actualizar la contraseña. De ser errónea el controlador te redirige a una pantalla de error con un mensaje de advertencia de que la contraseña introducida no coincide.

4.5.3 Servicio de Administrador.

El sistema debe tener un usuario que tenga permisos para poder modificar los datos de los encargos, de los usuarios y de las farmacias. Este usuario tendrá una vista de la aplicación distinta ya que debe poder ver la posición global de la aplicación. Para ello se definen una serie de vistas, funciones y clases exclusivas.

En primer lugar y de manera manual se va a crear un usuario en la base de datos denominado *admin*, que será con el que se acceda desde el *login* principal de la página.

A continuación, se explican las funciones que se van a implementar en función del servicio que cubren (Véase Figura 20). El administrador del sistema tiene que poder ver todos los usuarios, farmacias y encargos que haya en la aplicación, por tanto, se van a crear tres funciones, 1) *showAllUsers* en *User.java*, 2) *showAllPharmacies* en *Pharmacy.java* y 3) *showAllBooking*s en *Booking.java* que cubrirán esas acciones respectivamente. Las tres funciones mantienen la misma estructura lógica, todas acceden a su respectiva colección de Mongo (*users*, *pharmacies y bookings).* Además a través de una sentencia *find*, almacenan en un *MongoCursor* todos los documentos que contenga cada colección. En este caso, no hay filtro de búsqueda que pasarle a la sentencia *find* ya que se quieren extraer todos los documentos. A continuación, se recorre cada *MongoCursor* y con la ayuda de un *ObjectMapper* se deserializa cada documento en su clase correspondiente, es decir, se convierten cada uno de los registros del documento en un atributo de la clase que se le pase como parámetro al *ObjectMapper*, a través de los métodos *setter* de la clase. En este caso, es importante que los atributos de la clase tengan el mismo nombre que los registros del documento, si no saltaría una excepción. Además, para que el *ObjectMapper* funcione correctamente, el documento debe ser convertido a *Json* antes de pasarlo como parámetro. Una vez que el documento se ha convertido en objeto, este se añade a una lista que será devuelta por la función.

```
public  LinkedList<MyUser> showAllUsers(MongoDatabase database) {

		MyUser user = new MyUser();
		ObjectMapper mapper = new ObjectMapper();

		LinkedList<MyUser> usersList = new LinkedList();

		Bson projections = Projections.exclude( "_id");


		MongoCollection<Document> collection = database.getCollection("users");

		MongoCursor<Document> users = collection.find().projection(projections).iterator();
		try {
			//Por cada  usuario, buscamos la información
			while(users.hasNext()) {
				Document doc = (Document) users.next();

				try {
					String s = doc.toJson();
					user = mapper.readValue(doc.toJson(), MyUser.class);
					usersList.add(user);


				} catch (JsonParseException e) {
					// TODO Auto-generated catch block
					e.printStackTrace();
				} catch (JsonMappingException e) {
					// TODO Auto-generated catch block
					e.printStackTrace();
				} catch (IOException e) {
					// TODO Auto-generated catch block
					e.printStackTrace();
				}
			}
		}
		finally {
			users.close();
```

```
        }
        return usersList;
    }
```

Figura 20 Función para listar todos los usuarios.

Las funciones para editar los usuarios y los encargos son las mismas que se crearon para el perfil usuario, *editProfile* y *updateStatus* respectivamente, ya que desde el administrador no se permite cambiar ningún dato del encargo salvo el estado. En cambio, desde el perfil usuario no existía una función para editar los datos de la farmacia. Por tanto, eso es un privilegio exclusivo del administrador. Para poder editar los datos, hay que implementar dos funciones nuevas, una que extraiga los datos almacenados en la base de datos y otra que permita actualizarlos. Para la extracción de datos, dentro de la clase *Pharmacy.java* se crea la función *getPharmacyInfo* que recibe como parámetro el teléfono de la farmacia. Se ha escogido el teléfono porque es un dato que se puede recuperar fácilmente de los datos que muestra la tabla en la vista *showAllPharmacies.jsp.* Ese parámetro se utiliza como filtro de búsqueda en la función *find* que accede a la colección *pharmacies*. A continuación, el documento que se almacena en un *MongCursor*, se *deserializa* a un objeto de tipo *Pharmacy* objeto devuelto por la función.

Para la acción de actualizar los datos, se desarrolla dentro de la misma clase, el método *editPharmacy*. (Véase Figura 21). Esta función recibe como parámetros los datos de la farmacia, el CIF, el email, el nombre y el teléfono. En este caso se descarta poder modificar la dirección ya que es una farmacia hospitalaria y no puede cambiar su ubicación al menos que el hospital se traslade. Del mismo modo que en la función de actualización de datos similares mencionadas anteriormente, se crea un objeto *BasicDBObject*, en el cual se añaden los campos que se quieren actualizar, pasándole en primer lugar, el nombre del campo en la base de datos y como segundo parámetro el nuevo valor. Este objecto se pasa como segundo parámetro de otro objeto del mismo tipo y como primer parámetro la etiqueta *$set* que permite actualizar los campos. Por último, se llama a la función *findOneAndUpdate* pasándole como filtro de búsqueda el CIF y como valores para actualizar el segundo *BasicDBObject* creado.

```
public boolean editPharmacy(String name, int phone, String mail, String cif,
                            MongoDatabase database){

        //Accedemos a la colección usuarios de la base de datos-
        MongoCollection<Document> pharmacies =  database.getCollection("pharmacies");

        BasicDBObject oquery = new BasicDBObject();
        oquery.put("cif", cif);

        BasicDBObject doc = new BasicDBObject();
        BasicDBObject op = new BasicDBObject();
        op.put("name", name);
        op.put("phone", phone);
        op.put("email", mail);
        doc.put("$set", op);

        pharmacies.findOneAndUpdate(oquery, doc);
```

```
        return true;
}
```

Figura 21 Función para editar los datos de una farmacia.

Las vistas que se diseñan para estas funciones también comparten el mismo patrón de diseño entre sí y es el mismo que tienen vistas con funciones similares como *newBookig.jsp* que se encargaba de listar las farmacias que disponían de un producto en stock. La repetición de estructuras en las diferentes vistas se ha hecho para mantener una homogeneidad de diseño y apariencia, evitando así que cada página tenga un estilo distinto.

Por tanto, se diseñan tres vistas llamadas *showAllUsers.jsp*, *showAllPharmacies.jsp*, *showAllBookings.jsp*. (Véase Figura 22). Estas vistas cuentan con el uso de la función *DataTable* de *BootStrap* y con *JQuery* para implementar los scripts que activan los botones de acción de cada tabla. Las vistas están compuestas por tres divisiones principales, una para el título, otra para una pequeña descripción y la última que desplegará la tabla. La tabla inicialmente y al estar diseñada en formato *HTML* utiliza las etiquetas *<tr>*, *<th> y <td>* para definir los elementos de la cabecera y el cuerpo, las columnas y las filas respectivamente. Además, el contenido de la tabla se muestra a través de un bucle *forEach* donde la etiqueta *ítems* representa el objeto a recorrer, es decir, la lista que devuelve la función de la lógica correspondiente y la etiqueta *var* corresponde al nombre de la variable que accederá a cada una de las posiciones de la lista. Esta variable muestra en cada fila un atributo del objeto de la lista. Para acceder a los atributos de los objetos de la lista, se sigue el esquema *nombreVariable.nombreAtributo*. Los datos, para los usuarios, que se muestran en las tablas son: el nombre, el correo electrónico y el identificador de la farmacia a la que pertenecen. Para las farmacias, el nombre de esta, la dirección y el teléfono. Por últimopara los encargos, la referencia, la farmacia origen, el destino y el estado del encargo. Además, como cada tabla tiene un botón de acción en cada fila, para poder gestionar o editar los datos del usuario, de la farmacia o del encargo, es necesario incluir fuera de la etiqueta de la tabla, un input oculto que guarde los parámetros que necesitará la función de edición correspondiente. Tras el cierre de la etiqueta *HTML*, se declaran dos scripts, uno que llamará a la función *DataTable* y se le pasará como atributo el nombre de la tabla creada en el *HTML*. Además, como en vistas anteriores, se personalizan datos de la tabla como por ejemplo el lenguaje o el número de objetos que se muestran por página. El segundo script viene relacionado con el botón de gestionar disponible en cada fila. Dentro de este script se recogen los valores de los *inputs* declarados en cada tabla, como puede ser el correo para la tabla de usuarios, el nombre de la farmacia para la tabla de farmacias o la referencia del encargo para la tabla de reservas. Con este script se implementa la función de *submit* de los *inputs* de forma personalizada y enviando al controlador un dato concreto de la tabla y no todas las entradas.

```
<div class="container">
	<br> <br> <br> <br>
	<h2>Listado de usuarios</h2>

	<div class="panel panel-primary filterable">
		<div class="panel-heading">
			<h3 class="panel-title"></h3>

		</div>

	</div>
	<div>
		<table id="showAllUsers"
			class="table table-striped table-bordered table-sm">
			<thead>
				<tr>
					<th><span>Nombre</span></th>
					<th><span>Email</span></th>
					<th><span>Farmacia Asociada</span></th>
					<th><span>Acciones</span></th>
				</tr>
			</thead>
			<tbody>

				<c:forEach items="${usersList}" var="showUsers">
					<tr>
						<td>${showUsers.name}</td>
						<td>${showUsers.email}</td>
						<td>${showUsers.idPharmacy}</td>
						<td><input type="button" class="btn-info btn-md"
							value="Gestionar"
onClick="submitForm1('${showUsers.email}')">
						</td>
					</tr>
				</c:forEach>


			</tbody>
		</table>

		<form name="formId" id="formId" action="userInfo" method="post">
			<input name="email" id="email" type="hidden">
		</form>
	</div>
</div>


<script>
	$(document)
			.ready(
					function() {
						$('#showAllUsers')
								.DataTable(
{

	"language" : {


	"zeroRecords" : "No se encontró nada",

	"info" : "Mostrando del _START_ al _END_ de un total de _TOTAL_",

	"infoEmpty" : "No hay registros",

	"emptyTable" : "No hay datos para mostrar",

	"loadingRecords" : "Cargando...",

	"processing" : "Procesando...",

	"search" : "Buscar:",


	"paginate" : {
```

```
                    "first" : "Primera",

                    "last" : "Última",

                    "next" : "Siguiente",

                    "previous" : "Anterior"

            }

            }
                                                                                                    });
                                                                $('.dataTables_length').addClass('bs-select');

                                                        });
</script>
<script>
            function submitForm1(value) {
                    $("#email")[0].value = value; //cuando los datos vienen del controlador

                    $("#formId").submit();

            }
</script>
```

Figura 22 Código de la vista showAllUsers.jsp

Además de esta vista, el administrador del sistema tiene una pantalla principal distinta a la del usuario. Por tanto, es necesario crear una nueva página de bienvenida que se denomina *welcomeAdmin.jsp*. Esta vista mantiene la misma estructura que la página principal del usuario, salvo que los botones hacen referencia a los métodos arriba mencionados, propios del administrador.

Los controladores asociados a este servicio se definen en una nueva clase denominada *AdminController.java.* A continuación, se comentan los controladores creados, agrupados en función de la acción que desempeñan. Existen tres controladores denominados *showAllUsers*, *showAllPharmacies* y *showAllBookings*, respectivamente, que se encargan de recoger los datos de la función lógica que tiene el mismo nombre, subirlos al modelo y mostrar la vista asociada con los datos incrustados. Estos controladores no reciben *beans* de datos, ya que el envío de información es únicamente del controlador a la vista.

Por otro lado, están los controladores, *userInfo, pharmacyInfo y bookingInfo* que se encargan de recoger el dato seleccionado por el administrador, ya sea el nombre de un usuario, el teléfono de una farmacia o la referencia de un encargo que quiere gestionar y con ese dato llamar a la función correspondiente que extrae los datos de Mongo (*getUserInfo*. *getPharmacyInfo, searchBooking*). En este caso, como el flujo de datos es bidireccional, es decir emiten datos de la vista al controlador y viceversa, es necesario añadir un *bean* al controlador que pueda almacenar el dato que selecciona el usuario. En cada caso será un *bean* del objeto al que hacen referencia, es decir,

para *userInfo* será un *bean* de tipo *User*, para *pharmacyInfo* de tipo *Pharmacy* y para *searchBooking* de tipo *Booking*.

Finalmente, están los controladores que se encargan de recoger los datos de la vista y por medio de los métodos previamente mencionados actualizar los datos en Mongo. Estos controladores se llaman *editUserInfo*, *editPharmacyInfo* y *editBoookingInfo*. Debido a que reciben datos de la vista, es necesario añadirles un *bean* de cada tipo como en el caso de los controladores anteriores. La estructura de estos es acceder a la base de datos, llamar a la función correspondiente a la actualización de datos (*editUser*, *editPharmacy*, *updateStatus*) y por último devolver la vista de los datos actualizada.

4.5.4 Pruebas

Las pruebas ejecutadas para el servicio relacionado con la visualización de los encargos, no se explican detenidamente ya que se consideran triviales, en este caso, no existe modificación en base de datos que requiera una atención especial

Para el servicio de modificar los datos de usuario se realizan las siguientes pruebas: (Véase
Tabla 80).

ID	ENTRADA	SALIDA
1.	Actualizar el nombre de usuario y el correcto electrónico.	El sistema redirige a una página con un mensaje de datos actualizados correctamente.
2.	Borrar el contenido de los campos y dejarlo en blanco. Pulsar el botón de actualizar.	La página emite un mensaje de que es un campo requerido. No permite actualizar.
3.	Introducir la antigua contraseña correctamente y una nueva contraseña.	El sistema redirige a una página con un mensaje de datos actualizados correctamente.
4.	Introducir la antigua contraseña incorrectamente y una nueva contraseña.	El sistema redirige a una página con un mensaje, avisando que la antigua contraseña no coincide.
5.	Introducir la antigua contraseña correctamente y dejar el campo de la nueva contraseña vacío.	La página emite un mensaje de que es un campo requerido. No permite actualizar.

Tabla 80 Pruebas para el servicio de modificación de datos del usuario.

Para el caso de los servicios de administrador, la mayoría de los que se implementan son servicios de extraer datos de la base de datos y mostrarlos por pantalla en base a los campos indicados en las vistas, por tanto, no se consideran pruebas con contenido importante.

4.6 Sprint 5

Con el inicio de este Sprint se ha completado el 80% del proyecto. Sin embargo, quedan requisitos por completar necesarios para el cliente. Como análisis del anterior Sprint se ha observado una mayor agilidad en la definición de los métodos y en el diseño de las vistas. Así como una mayor rapidez a la hora de controlar las excepciones y errores correspondientes. Las tareas que se an a realizar en este Sprint son: (Véase Tabla 81. Tabla 82, Tabla 83).

Horas máximas	40
Horas utilizadas	16,5
Horas Libres	23,5
Historia	Horas estimadas
HU02	16,5

Tabla 81 Resumen de tareas Sprint 5.

Historias de Usuario	**Miembros** **A**	**B**	**Estimación media**	**Prioridad**
HU02	13	20	16,5	Medio
HU08	40	40	40	Media

Tabla 82 Product backlog Sprint 5.

Historia de Usuario	
ID	HU02
Nombre	Introducir y gestionar manualmente el stock.
Prioridad	Medio
Riesgo	Baja
Descripción	Como farmacéutico quiero poder meter productos de manera manual completando un formulario que indique el nombre, laboratorio, unidades, fecha de caducidad y CN.
Validación	- Poder elegir de la base de datos el medicamento que se quiere ingresar. - Los datos almacenados se pueden modificar posteriormente. - Quiero poder introducir nuevos productos en el sistema.

Tabla 83 Historia de usuario 2.

Este Sprint no implica muchas tareas que desempeñar ya que solo hay que desarrollar un nuevo servicio que sería poder introducir el stock de manera manual. Esta historia de usuario es importante ya que muchas veces en la farmacia hospitalaria no es necesario o no se quiere subir el stock de la farmacia al completo, solo se quieren prestar determinados medicamentos que con tiene una probabilidad baja de ser utilizados antes de que caduquen. La implantación de este servicio lleva consigo las siguientes tareas:

- Servicio que permita buscar un medicamento por código nacional.
- Servicio que permita mostrar el medicamento.
- Servicio para añadir las unidades introducidas por el usuario al stock de la farmacia.
- Interfaz para buscar el medicamento.
- Interfaz para mostrar el medicamento.
- Interfaz para añadir el medicamento.

Las clases y vistas que deberán ser modificadas o creadas son las siguientes:

- *Item.java*

- *StockManagementController.java*

- *AddItem.jsp*

- *Welcome.jsp*

4.6.1 Servicio para Añadir Manualmente el Stock.

Dentro de la clase *Item.java* hay que definir la función *searchItem* que recibe el código nacional del producto y elemento *MongoDatabase.* Con estos parámetros accede a la colección *items* de Mongo y con el filtro de búsqueda que equivaldría al *_id* del documento, se saca el documento que coincide con el código nacional. A continuación, este documento se convierte a *json* para poder deserializarlo mediante un *ObjectMapper* en un objeto de tipo *Item.* Este objeto será devuelto por la función.

La vista de este servicio se llama *addItem.jsp* y mantiene la misma estructura que la vista de *showStock.jsp.* (Véase Figura 23). Ambas utilizan la librería *BootStrap y Jquery.* La estructura de la página se divide en tres divisiones, y cuenta con un título en la parte superior, debajo un buscador que permitirá introducir el código nacional del producto al usuario y en la parte principal de la página, una tabla que mostrará los datos del medicamento y la posibilidad de añadirlo al stock. La tabla está definida como en otras ocasiones con etiquetas *tr, th y td* y los datos a mostrar son, el código nacional del producto, el nombre, el laboratorio y las unidades que el usuario quiere introducir en su stock, así como un botón de añadir. Como elementos diferenciadores de esta página es que se incluyen dos *forms* con acciones independientes. El primero de eso se encarga de buscar el código introducido y mostrar los resultados. El segundo, se encarga de recoger el código nacional y las unidades introducidas por el usuario y enviarlas al controlador correspondiente para que lo inserte en el stock de la farmacia. En este caso, no se utiliza ningún script personal ya que solo hay una entrada en la tabla y se recogen los datos de los dos *inputs* declarados.

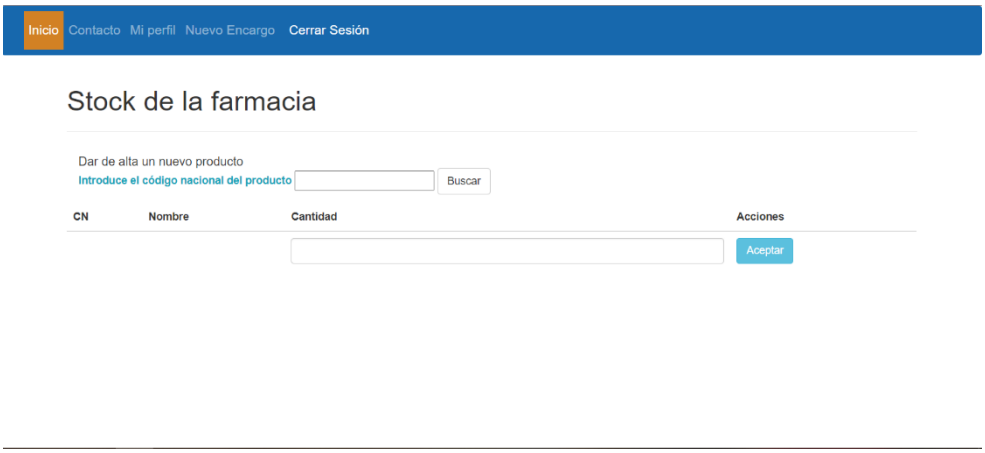

Figura 23 Vista para añadir un nuevo producto.

Además, hay que modificar la página principal, *welcome.jsp* y añadir un nuevo botón que permita mostrar la página de añadir ítems.

Los controladores asociados a este servicio están declarados en la clase *stockManagementController.java* y son los siguientes: 1) *manualStockManagement* que permite lanzar la vista sin datos. 2) *SearchItem* que se encarga de recoger el dato introducido por el usuario gracias al *bean* que tiene pasado como parámetro y llamar a la función *searchItem* para devolver el medicamento correspondiente. Este controlador, a su vez, añade el *item* al modelo y vuelve a llamar a la vista para que extraiga el dato y lo muestre en la tabla. Por último, estaría *importStockProcesManual* que se encarga de extraer los datos de la vista, las unidades y el código nacional, los almacena en variables y extrae del contexto de sesión el email del usuario. Con esos datos, se accede a la colección *users*, se extrae el id de la farmacia y este se utiliza para sacar el CIF de esta. A continuación se llama a la función de añadir el ítem (*addItem)* y se retorna la vista de *sucessImport.jsp* que saca un mensaje por pantalla confirmando que el stock se ha añadido correctamente.

El perfil del administrador quedaría como se muestra en la Figura 24 . Debido a que se ha hecho la captura después de finalizar todo el proyecto, sale una cabecera que será explicada en el siguiente *Sprint* (Véase 4.7 Sprint 6.).

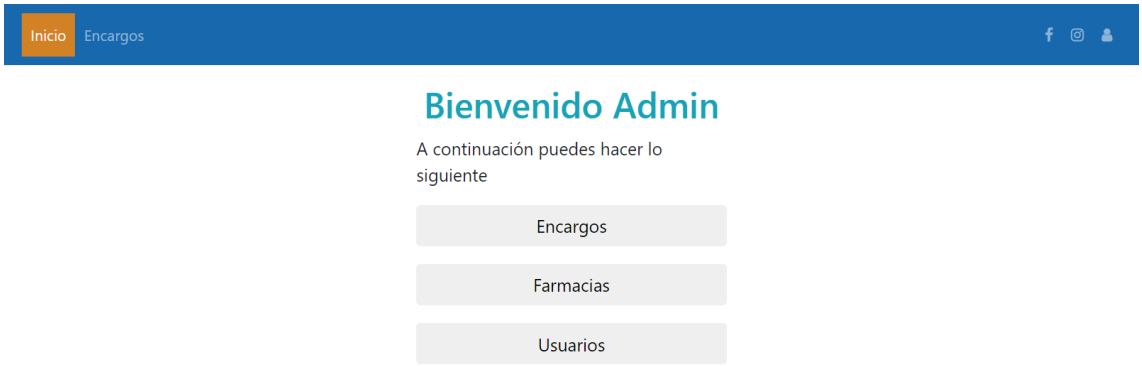

Figura 24 Vista principal desde el perfil administrador.

4.6.2 Pruebas

Las pruebas realizadas para este Sprint han sido las siguientes: (Véase Tabla 84).

ID	ENTRADA	SALIDA
1.	Buscar un producto existente en la base de datos de medicamentos y no dado de alta en el stock. Añadir 1 unidad.	El sistema redirige a una página con un mensaje de stock subido correctamente.
2.	Buscar un producto existente en la base de datos de medicamentos y no dado de alta en el stock. Añadir -1 unidad.	La página emite un mensaje que el valor mínimo a introducir es 1 unidad.
3.	Buscar un producto existente en la base de datos de medicamentos y no dado de alta en el stock. Dejar el campo de unidades vacío.	La página emite un mensaje que el campo de unidades es obligatorio.
4.	Buscar un producto existente en la base de datos de medicamentos y dado de alta en el stock. Añadir 3 unidades.	El sistema redirige a una página con un mensaje de stock subido correctamente. En este caso, actualiza el valor del stock al último introducido.
5.	Buscar un producto no existente en la base de datos de medicamentos. .	La página no muestra resultados.

Tabla 84 Pruebas del servicio de edición de stock.

4.7 Sprint 6.

El anterior *Sprint*, debido al avanzado estado del proyecto, solo contenía una historia de usuario a realizar y, además, esta era de bajo coste. Esto implicaba que las horas del *Sprint* no fueran utilizadas en su mayoría.

En este último Sprint *se procede a completar la historia de usuario que falta y con esto se dará por terminada lista de requisitos que solicitaba el cliente. (Véase Tabla 85,*

Tabla 86, Tabla 87).

Horas máximas	40
Horas utilizadas	40
Horas Libres	0
Historia	Horas estimadas
HU08	40

Tabla 85 Resumen de tareas Sprint 7.

Historias de Usuario	**Miembros**		**Estimación media**	**Prioridad**
	A	**B**		
HU08	40	40	40	Media

Tabla 86 Product backlog Sprint 7.

Historia de Usuario	
ID	HU08
Nombre	Aplicación web
Prioridad	Media
Riesgo	Bajo
Descripción	El sistema debe poder abrirse en cualquier navegador web.
Validación	- El acceso al sistema debe poder hacerse a través de cualquier navegador web. - Para poder entrar deben registrarse con usuario y contraseña. - La aplicación web debe ser visual, predominando las acciones relacionadas con el stock y las búsquedas.

Tabla 87 Historia de usuario 8.

Esta historia de usuario, declarada por el cliente, viene definiéndose a lo largo de todo el proyecto ya que, desde el inicio, se ha desarrollado una aplicación enfocada al entorno web y que tras su aprobación será subida a un servidor, para ser publicada y accesible por el resto de los usuarios.

En primer lugar, se tomó Spring MVC como *framework* de desarrollo. Este entorno está enfocado en desarrollo web ya que permite separar las vistas del modelo de datos y relacionar todo mediante controladores. Para ello, antes del comienzo del proyecto es necesario configurar un archivo que permitirá a Spring saber dónde se encuentran las clases que ejercerán de controladores y cómo será la estructura de la página. Este archivo se denomina web.xml y contiene la configuración del *servlet*, que en este caso se llama *AppServlet*. (Véase Figura 25). Este es una instancia de la clase propia de Spring denominada *org.springframework.web.servlet.DispatcherServlet.* El *DispatcherServlet* es quien realiza las tareas de controlador frontal de la aplicación y captura todas las peticiones. Además, este queda inicializado con un archivo denominado *servlet.-context.xml.*

```
<servlet>
		<servlet-name>appServlet</servlet-name>
		<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
		<init-param>
			<param-name>contextConfigLocation</param-name>
			<param-value>/WEB-INF/spring/appServlet/servlet-context.xml</param-value>
		</init-param>
</servlet>
```

Figura 25 Configuración de Spring MVC.

El archivo *servlet-context.xml* permite definir una serie de *beans* que marcarán la forma de leer las peticiones http y definirá dónde se encuentra los recursos tales como las vistas o los

controladores. En primer lugar, se configura *annotation-driven* para que admita las etiquetas de tipo *@Controller, @Autowired o @Service* que se utilizan en las clases Java. Esto permite que *Spring* automáticamente al reconocer esas etiquetas configure cada clase y no sea necesario definir a través de *beans*, en el archivo *xml*, cada controlador que se implementa. Es necesario especificar también el paquete dónde se tienen que buscar las etiquetas asociadas a los controladores, a través de *component-scan*. Por último, se define la ubicación de las vistas que los controladores deben utilizar así como los prefijos y sufijos que deben tener las vistas para que puedan ser entendidas por los controladores. En este caso, las vistas se encuentran en el lugar predefinido por *Spring*, dentro de la carpeta *WEB-INF/views* y el sufijo de las vistas será el de las páginas *.jsp*. En este archivo se declaran todos los *beans* de las herramientas utilizadas como puede ser el servicio de Email de Spring.

Una vez configurado la forma en la que la aplicación recibirá las peticiones, hay que añadirle un *servlet-mapping* que define qué peticiones podrá atender la aplicación. (Véase Figura 26). En este caso se utiliza un patrón general y podrá admitir todas las URL que comiencen por \.

```
<servlet-mapping>
                    <servlet-name>appServlet</servlet-name>
                    <url-pattern>/</url-pattern>
</servlet-mapping>
```

Figura 26 Configuración de las URL.

Con estos componentes los archivos de configuración ya están preparados y se puede lanzar la aplicación con un servidor local. Esta configuración también podría haberse hecho sin el archivo *.xml*, declarando la configuración en clases Java directamente. Sin embargo, se ha decidido hacer un híbrido entre las dos configuraciones para así poder aprender ambos sistemas.

Además de la configuración básica de *Spring* y aunque los requerimientos del cliente están cubiertos, se decide añadir seguridad a la web para dependiendo del role que tenga el usuario pueda acceder a unas páginas u otras. Añadir esta funcionalidad implica realizar varios cambios en el proyecto. En primer lugar, se va a utilizar un módulo de Spring, denominado Spring Security que permite realizar estos ajustes sin necesidad de programar todo desde cero. Para ello, se añaden las dependencias correspondientes de Spring Security en Maven al archivo pom.xml. A continuación, es necesario configurar Spring Security para que pueda extraer los datos de Mongo y hacer el proceso de autentificación y verificación. Debido a que Mongo no es una base de datos SQL, no se puede utilizar el esquema de conexión con bases de datos SQL que viene definido de forma predeterminada en Spring Security.

Dentro del archivo web.xml se crea un filtro de seguridad. (Véase Figura 27). Este filtro delegará en los filtros que se definan en el archivo de configuración *spring-securiry.xml*. El filtro principal declarado en el web.xml, afectará a todas las URL de la aplicación. También se delimita a 15 minutos el tiempo que puede estar iniciada la sesión sin actividad.

```
<session-config>
        <session-timeout>15</session-timeout>
</session-config>

<filter>
        <filter-name>springSecurityFilterChain</filter-name>
        <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>
<filter-mapping>
        <filter-name>springSecurityFilterChain</filter-name>
        <url-pattern>/*</url-pattern>
</filter-mapping>
```

Figura 27 Configuración del archivo .xml.

El archivo de configuración de seguridad de *Spring* está formado principalmente por dos filtros. El filtro de autenticación (*authenticationProcessingFilter*) que se encarga de generar el token y la sesión entre otras tareas en el contexto de seguridad creado. Dentro de este filtro hay que definir una propiedad denominada *AuthenticationManager* que se encarga de recuperar los datos de sesión que el usuario introduce al iniciar sesión. Esto lo consigue definiendo un *provider* que hay que, en este caso, al ser Mongo habrá que hacer una clase personalizada. Este *provider* es el que proporciona al *AuthenticationManager* los detalles del usuario. Así, el *provider* construye un objeto *UsernamePasswordAuthenticationToken* si la contraseña es la correcta y se lo pasa al *AuthenticationManager* encargado de rellenar el *SecurityContextHolder*. Por otro lado, está el *bean, springSecurityFilterChain*, encargado de gestionar los diferentes filtros de la cadena de seguridad, es decir, de definir las páginas a las que el usuario puede entrar según su rol. Esto se hace a través de la etiqueta *http y intercept-url.* Además, el http se puede configurar con la etiqueta de *auto-configure* a *true* para que él mismo defina los filtros básicos necesarios para arrancar el sistema que el desarrollador no ha personalizado.

Antes de definir la clase que accederá a los datos es necesario hacer unos cambios en el proyecto. En primer lugar, se debe renombrar la clase *User.java* definida en anteriores Sprints a *MyUser.java* ya que coincide con la clase *User* de *Spring Security* que se necesita para configurar el *bean* personalizado. Asimismo, es necesario añadir el atributo role a esta misma clase que definirá si el usuario tiene rol de administrador o de usuario. Este cambio implica modificar la función de registro del usuario, para añadir este campo también en el modelo de datos.

Por otro lado, para evitar tener que modificar el proyecto se va a crear una clase que extienda de *org.springframework.security.core.userdetails.User* denominada *CurrentUser.java*. En esta clase se definen todos los atributos que sean necesarios subir a los datos de sesión del contexto de seguridad. Para este proyecto solo se necesita subir el dato del email del usuario. Sin embargo, para dejarlo adaptado para las líneas futuras de trabajo. También se va a almacenar un objeto de tipo *MyUser.java*. Por tanto, esta clase contará con un constructor que utilizará los atributos propios de su súper clase, *username, password, enabled, accountNonExpired, credentialsNonExpired, accountNonLocked, ¿Collection<? extends GrantedAuthority>* que se encargan de definir el usuario, la contraseña, si está conectado, la sesión y los credenciales no han expirado, si la cuenta no ha sido bloqueada y por último los permisos que tiene ese usuario. Y, además, se añade el objeto MyUser.java.

A continuación, se va a definir el proveedor de servicio personalizado para el proyecto (Véase Figura 28).. Para ello, se crea una clase con la etiqueta *@Component* y llamada *CustomUserDetailsService* que implementa la clase *UserDetailsService*. Esta clase va a extraer los datos almacenados en Mongo y si coinciden con los introducidos, devolverá un token correcto. Para ello, se sobrescribe la función *loadUserbyUsername,* pasándole como parámetro el email del usuario. Se accede a la colección *users* de Mongo y se extrae el documento que coincida con el email facilitado a través de una sentencia *find*. Después, se crea un objeto de tipo *CurrentUser* a través del constructor, y que se devuelve en la función. En el caso que haya un error con el documento, captura una excepción a través de la estructura *try-catch.* Para poder traducir el atributo rol que se ha definido en la clase MyUser.java hay que sobrescribir la clase *getAuthorities* propia de *UserDetailsService*. En este caso la función devuelve un *List<GrantedAuthority>* de tipo ROLE_USER o ROLE_ADMIN en función de si el parámetro pasado es un 0 o 1 respectivamente.

```
@Service("customerUserDetailsService")
public class CustomUserDetailsService implements UserDetailsService {

	private MongoFactory mongo = new MongoFactory();
	private static final Logger logger = LoggerFactory.getLogger(CustomUserDetailsService.class);


	@Override
	public UserDetails loadUserByUsername(String email) throws UsernameNotFoundException {
		MongoClient client = mongo.crearConexion();
		ObjectMapper objectMapper = new ObjectMapper();


		MongoDatabase database = client.getDatabase("db01");
		MongoCollection<Document> collection = database.getCollection("users");
		Bson projections = Projections.exclude( "_id");
		Document document = collection.find(Filters.eq("email", email)).projection(projections).first();
		MyUser myUser = new MyUser();
		try {
			myUser = (MyUser) objectMapper.readValue(document.toJson(), MyUser.class);
			logger.info("Se ha encotrado coincidencia de email y contraseña en mongo.");

		} catch (IOException e) {
```

```
                    logger.error("Error al serialziar el documento bson. No existe el usuario");

                    // TODO Auto-generated catch block
                    e.printStackTrace();
            }
            boolean enabled = true;
            boolean accountNonExpired = true;
            boolean credentialsNonExpired = true;
            boolean accountNonLocked = true;


            String password = document.getString("password");
            int role = (int) document.get("role");

            CurrentUser userDetails = new CurrentUser(myUser, email,
                                password,
                                enabled,
                                accountNonExpired,
                                credentialsNonExpired,
                                accountNonLocked,
                                getAuthorities(role));
            return userDetails;

    }
```

Figura 28 Definición de la clase CustomUserDetailsService

Ya se ha definido la forma de extraer los datos de inicio de sesión, ahora se va a definir el manejador de estos datos. Se define una clase llamada, *CustomAuthentication* también con la etiqueta *@Component*, que hereda de la clase *AuthenticationProvider*. Esta clase se encarga de definir la forma en la que se va a comprobar que los datos introducidos por el usuario coinciden con los que ha devuelto *CustomUserDetailsService.java*. Para ello, se sobrescribe la función *authenticate* que devuelve un objeto *Authentication* formado por *UsernamePasswordAuthenticationToken.* Esta función lo que hace es a través de la función *checkUser* de la clase *MyUser.java* y definida en el *Sprint* 1, comprueba que el usuario y contraseña coinciden. De ser correcto, comprueba que el email pasado como parámetro de la función *checkUser* sea distinto de *admin* y le asocia la autoridad de ROLE_USER, en caso de coincidir le otorga la autoridad de ROLE_ADMIN. Por último, se crea un objeto de tipo *UsernamePasswordAuthenticationToken* que es devuelto por la función. En el caso de que checkUser no sea *true*, salta la excepción *BadCredentialsException.*

Siguiendo con el archivo de configuración se definen los filtros de seguridad que estarán dentro del *bean http.* (Véase Figura 29). Las *urls* que tendrán acceso sin restricción serán, las asociadas con el formulario de inicio de sesión y con el registro de nuevos usuarios y farmacias. Para ello se sigue la estructura:

```
<intercept-url pattern="/nombrePagina" access="permitAll" />
```

Figura 29 Selección de las páginas accesibles.

En cambio, para el resto de las páginas, se establece que tienes que tener el rol de usuario o administrador para poder entrar. (Véase Figura 30).

```
<intercept-url pattern="/**" access="hasRole('ROLE_USER') or hasRole('ROLE_ADMIN') " />
```

Figura 30 Selección de las páginas con restricciones.

Spring Security tiene su propio formulario de inicio de sesión, por tanto, para personalizarlo hay que indicarlo dentro del *bean http*. (Véase Figura 31). Además, hay que indicar que controlador maneja el proceso de *login* y a que páginas debe redirigir si el *login* es correcto o por el contrario falla. También se especifica cuáles son los nombres de los *inputs* que debe recoger los datos que equivalen al usuario y la contraseña.

```
<form-login login-page="/login" username-parameter="email"
                    login-processing-url="/loginProcess" password-parameter="password"
                    authentication-failure-url="/errorEmail" default-target-url="/main" />
```

Figura 31 Formulario de inicio de sesión personalizado.

Por último, se define el proceso de cierre de sesión. (Véase Figura 32). Para eso, además, de indicarlo en *security-context.xml*, hay que definir el controlador dentro de *loginController.java*. Este controlador se encarga de extraer el objeto *authentication* del contexto de seguridad y si este no es nulo, llama a *logout* del objeto *SecurityContextLogoutHandler* creado para borrar los datos.

```
<logout logout-success-url="/login" delete-cookies="JSESSIONID" />
```

Figura 32 Formulario de cierre de sesión personalizado.

Con esto, queda configurado *Spring Security* con un entorno de seguridad básico.

También se decide implantar un servicio de contacto con la empresa vía email haciendo del servicio ya implementado (Véase 4.4.2 Servicio para Hacer Reservas.). Para ello, se crea la vista, *emailForm.jsp* que contiene un formulario que llama al controlador *sendEmail*. Los *inputs* de esta vista son, uno para el mail a quien va dirigido la solicitud de soporte, (en este caso el usuario no puede modificarlo), otro hace referencia al email desde el cual se manda el correo, es necesario, para saber a quién debe dirigirse el equipo de soporte y, por último, un *textarea* para que el usuario pueda escribir el mensaje.

A continuación, siguiendo con el propósito de este *Sprint*, todas las aplicaciones web mantienen el mismo estilo en sus diferentes vistas, esto se ha logrado manteniendo la misma estructura de divisiones en todas ellas y utilizando los mismos colores a lo largo del diseño. Sin embargo, para estandarizar todas las páginas se va a crear una cabecera que se incorporará a todas las vistas. Para ello se define la cabecera *header.jsp* que utiliza un compone especial de *BootStrap* denominado *navBar*. Este componente es una barra de navegación y permite introducir acciones dentro de la cabecera. La cabecera se define dentro de las etiquetas *header* de *HTML* y cuenta con dos divisiones marcadas por la etiqueta ul. La primera división posee los *links* a acciones como la página de inicio, el perfil del usuario, una opción de contacto y la posibilidad de hacer un nuevo encargo. La segunda división estará en la parte derecha de la cabecera y tendrá el botón de cierre de sesión y los links a las páginas de *Instagram* y *Facebook* de *LudaFarma*. Para definir los colores y el tamaño de la cabecera se utiliza código *CSS*. Esta cabecera debe incluirse en las vistas *.jsp* a través del comando *include*. Se muestra la página principal del usuario. (Véase Figura 33).

Figura 33 Vista principal desde el perfil usuario.

Por último, antes de finalizar el proyecto es necesario subirlo a producción. En este caso, en vez de utilizar el hosting de LudaFarma, se va a realizar la subida al entorno de AWS utilizando la herramienta: *ElasticBeanstalk*. Este servicio de AWS permite crear un servidor sin necesidad de configuración por parte del usuario, solo es necesario crearse una cuenta, seleccionar el servidor bajo el que está funcionando la aplicación, en este caso *Tomcat* y subir el archivo *.war* del proyecto. El servicio se encargará de crear el servidor y mantenerlo en ejecución.

Como detalle, debido a que *Mongo* no está habilitado dentro del entorno AWS y dado que la base de datos de este proyecto se encuentra creada en la nube de Atlas, es necesario, mientras se

configura el entorno, añadirlo a una red privada (VPC) y permitir la conexión por SSH de todas las direcciones.

Con la subida a producción se daría por finalizado el proyecto, habiendo cumplido todos los requisitos del cliente.

4.7.1 Pruebas

Para este último Sprint, las pruebas que se han realizado no han sido unitarias. Se han realizado pruebas de integración tras la subida a producción, comprobando el buen funcionamiento del sistema tras cambiar el servidor a un servidor en la nube.

Por otro lado, las pruebas realizadas para el sistema de correo, han sido las relacionadas con el envío correcto del mensaje y comprobación de este.

Por último, las pruebas realizadas para comprobar el correcto funcionamiento de Spring Security han sido las que se muestran en la Tabla 88.

ID	ENTRADA	SALIDA
1.	Acceso con usuario y contraseña correcto. Tipo usuario.	El sistema redirige a la página principal del usuario. .
2.	Acceso con usuario y contraseña incorrectos.	El sistema recarga la página de inicio de sesión.
3.	Cierre de sesión y volver a cargar la página anterior.	El sistema redirige a la página de inicio de sesión.
4.	Acceso sin iniciar sesión, a través de la URL a /mainAdmin.	El sistema redirige a la página de inicio de sesión.
5.	Acceso a las opciones de registro.	La página redirige al sistema de registro.

Tabla 88 Pruebas para Spring Security

Capítulo 6.

Conclusiones

Desde hace unas décadas, los sistemas de información se han convertido en una herramienta imprescindible para las empresas, sin embargo, el sector farmacéutico no ha podido hacer uso de ellas en su gran medida debido a las restricciones del sector. Esto provoca que pocas personas se atrevan a adentrarse y crear lo que sería un nuevo modelo de negocio de la farmacia hospitalaria.

Hasta la fecha se desconoce que exista una red farmacias a nivel nacional salvo la creada por la empresa LudaFarma. La creación de una red de farmacias permite en primer lugar frenar los problemas de desabastecimiento existentes en España, ya que se permite localizar un producto a nivel nacional y en tiempo real. En segundo lugar abre la puerta al uso de la telefarmacia y al bloqueo de los grandes gigantes como Amazon y su *Marketplace*. Esta misma idea se ha querido trasladar a la farmacia hospitalaria, que a pesar de ser uno de los sectores que más dinero mueven en España, es el menos avanzado en aspectos tecnológicos.

Teniendo cuenta esta situación, se ha desarrollado y validado una plataforma que aporta lo siguiente:

- Conexión entre las más de 700 farmacias hospitalarias públicas existentes en España.

- El software es multiplataforma, se puede acceder desde cualquier sistema operativo y está disponible para cualquier programa de gestión de farmacia hospitalaria. Solo es necesario seguir la plantilla de subida de stock ofrecida en el manual de usuario.

- Se ha desarrollado una aplicación portable para diferentes navegadores web. Con esta aplicación se puede comprobar fácilmente el estado de los encargos, realizar búsquedas de medicamentos y encargarlas en otras farmacias.

- La herramienta es suficientemente flexible como para poder ampliar su funcionalidad fácilmente en un futuro.

Además, se han cumplido todos los requerimientos del cliente marcados inicialmente. Así mismo, como resultado, se ha obtenido una herramienta que cumple las expectativas previas sin pérdida de funcionalidad o eficiencia. En un futuro, es posible que la herramienta pueda mejorar añadiéndole nuevas funcionalidades y completando las ya existentes.

Capítulo 7.

Líneas de Trabajo Futuro.

Se pueden destacar las siguientes futuras líneas de investigación que permitirían mejorar la herramienta:

- Se podrían añadir nuevas funcionalidades que permitirían mejorar la herramienta y completarla. Estas nuevas funcionalidades podrían ser: la integración de la herramienta con los principales programas de gestión de la farmacia hospitalaria como pueden ser *Athos Pharma*, u *Oasis.* Esto posibilitaría conocer en tiempo real el stock de las farmacias y convertiría a la herramienta en un servicio más productivo e independiente. Para implantar este servicio sería necesario, además crear una aplicación de escritorio que permita acceder a la base de datos del programa de gestión.

  Otra de las funcionalidades que estaría pendiente de añadir sería la integración con algún servicio de transporte para poder completar el proceso de desde la reserva hasta la dispensación, sin necesidad de que sea la propia farmacia sea quien tenga que realizar las acciones de manera manual.

  Por último, sería muy recomendable, debido a que vivimos en un mundo de nuevas tecnologías, el desarrollo de una aplicación web que permita realizar las acciones de nueva reserva, gestionar los encargos o el stock de manera manual.

- Mejoras del sistema ya existente. En este sentido, se podría realizar: una subida de stock automática más flexible. Para ello el programa podría ser capaz de leer una cabecera de una hoja de cálculo y asociarlo a un atributo determinado, sin necesidad de un orden concreto.

Respecto a seguridad se podrían introducir medidas como el sistema de recuperar contraseña basado en un email y un token o el recordatorio de contraseña.

Por otro lado, el diseño de la aplicación es sencillo y funcional pero poco atractivo y la accesibilidad se podría mejorar. En esta línea posibles mejoras serían permitir al usuario cambiar entre colores claros y oscuros, permitir leer la página web o crear una distribución específica de la Vista para la página web.

Parte III

Apéndices

Apéndice A.

Creación de un Proyecto con Maven y Spring.

A continuación. se procede a explicar el funcionamiento de Maven, sus principales características y cómo debe ser instalado en el entorno de trabajo. Maven es una herramienta de software para la gestión y construcción de proyectos Java, principalmente se basa en la inyección de dependencias. Además, no es necesario tener descargadas todas las librerías que se van a usar en el proyecto, ya que Maven tiene capacidad de ser usado en red. Por tanto, el propio motor puede acceder al repositorio y descargar los *plugins* que necesite.

En este ejemplo se va a realizar la integración de Maven con el IDE Eclipse Jee 2018-12. El primer paso es descargar el plugin de Maven para Eclipse, para ello se abre el entorno y se accede dentro del menú *Help-> Install New Software*. En la pantalla que aparece se va a *Work with* la opción *All Available Sites.* De entre las opciones que nos apareceren, seleccionamos *Collaboration -> m2e - Maven Integration for Eclipse*. Una vez seleccionada la opción, pulsamos *Next*. (Véase Figura 34).

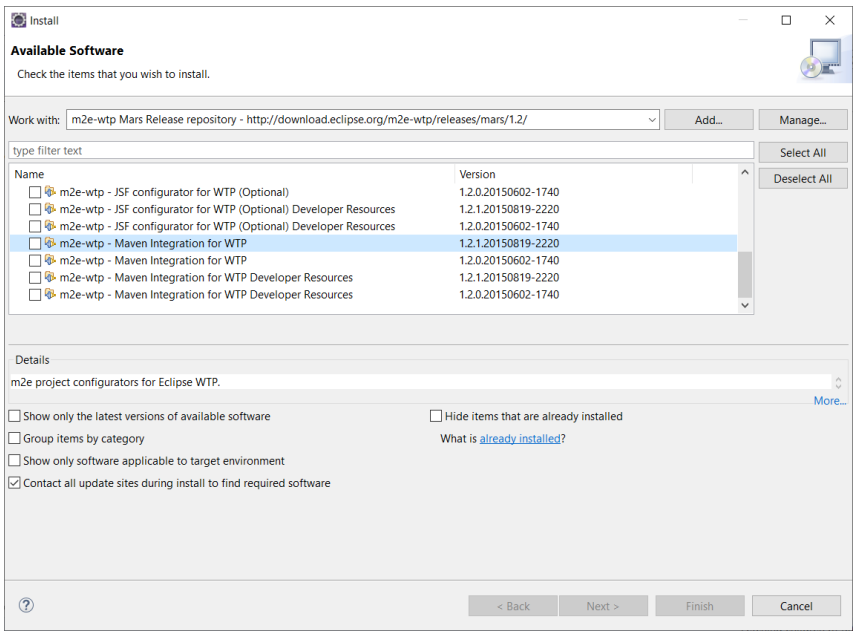

Figura 34 Instalación de Maven en Eclipse

Tras la descarga del *plugin* se procede a la instalación de este y posterior reinicio del entorno de Eclipse. Una vez que el proceso termine, el entorno ya tendrá disponible la opción de crear o convertir proyecto con Maven.

Tras la instalación de Maven se procede a explicar la instalación y creación de un proyecto con el *framework* Spring. En este caso, la instalación del plugin en Eclipse se hará desde un repositorio externo. Se accede a Eclipse, *help-> Eclipse Marketplace* y en el buscador *escribimos Spring Tools.* A continuación, se da a *install* y el programa comenzará un proceso de descarga e instalación. (Véase Figura 35).

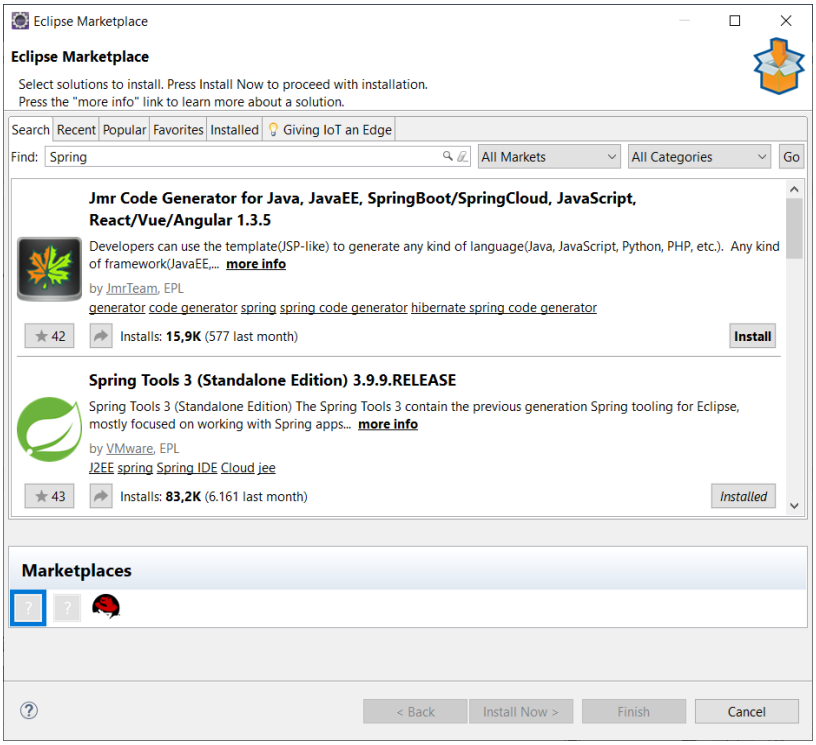

Figura 35 Instalación de Spring en Eclipse

Tras la instalación del plugin de Maven y el de Spring para Eclipse ya se puede crear un proyecto Spring MVC. Para ello vamos *a New->Project->Spring-> Spring Legacy Project.* En la siguiente ventana elegimos que el proyecto sea de tipo Spring MVC. (Véase Figura 36). Por último, se escribe el nombre del paquete base, por ejemplo: com.mycompany.myapp.

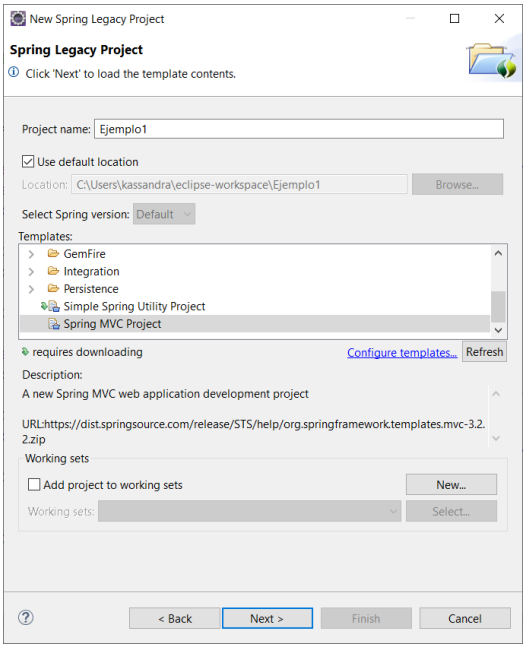

Figura 36 Elección del proyecto Spring MVC

El proyecto ya está creado y tiene la siguiente estructura de archivos. (Véase Figura 37).

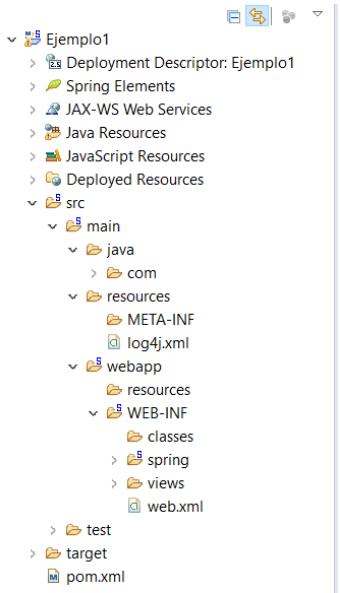

Figura 37 Estructura de archivos del proyecto.

Gracias a Maven y Spring no es necesario implementar una jerarquía, ya que al crear el proyecto se forma la jerarquía y se va actualizando en función de los archivos que se le inserten. Al crear el proyecto, viene con la información básica en el archivo *web.xml* y en el archivo *pom.xml* para que se puede lanzar la aplicación sin tener que modificar ninguna configuración.

Dentro del archivo *pom.xml* es donde tendrán que insertarse todos los enlaces a los repositorios de los servicios que se vayan a utilizar en el proyecto. Por otro lado, es muy recomendable añadir la perspectiva de Spring que ofrece el plugin al entorno de trabajo para así poder ver la dependencia entre los *beans*. Para ello, en la opción *window* de Eclipse, se selecciona *showView->Other* y se selecciona la vista Spring. A continuación, se despliega el explorador y aparecerá la relación de *beans*, muy útil cuando se quiere ver la estructura del proyecto. (Véase Figura 38).

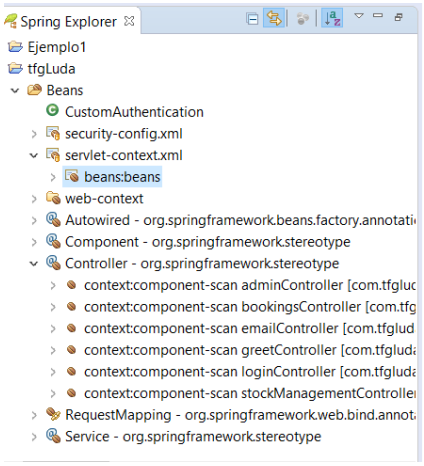

Figura 38 Relación de beans en el proyecto.

Apéndice B

Pliego de Condiciones.

En todo proyecto se definen una serie de condiciones que permiten que la aplicación funcione de manera correcta y ofrezca una buena experiencia de usuario al cliente. A continuación, se exponen los requisitos técnicos mínimos para el funcionamiento de la aplicación, así como una normativa que el usuario final debe cumplir.

- Pliego de condiciones generales:

    - El proyecto está desarrollado en dos partes, el cliente que será ejecutado en un navegador web en las diferentes máquinas de los usuarios. Y un servidor que se encuentra alojado en la nube de Amazon.

    - El sistema de gestión del stock y reservas no puede ser ejecutado directamente en el lado del cliente, ya que está alojado en un servidor externo. Sin embargo, si se puede acceder a él desde cualquier cliente.

    - El cliente es el único responsable de compartir cualquier dato de la aplicación con terceros.

    - LudaFarma dispone de libertad de actuación en el caso que se detecten acciones maliciosas o anómalas.

    - Se recomienda que, si un usuario detecta una incidencia, lo notifique a la empresa a través del formulario de contacto facilitado en la web.

    - El desconocimiento de la normativa no exime de su cumplimiento.

- LudaFarma no se hace responsable de los errores de stock que la farmacia pueda tener.

- Pliego de condiciones específicas:
  - Especificaciones de material y equipos:
    - Conexión a internet.
    - Navegador web. (Preferiblemente Chrome v40 o superior, también disponible con Safari v11 o superior, Firefox v44 o superior, o Edge v17 o superior).
    - Memoria RAM de 500MB o superior.
    - Procesador de 2GHz o superior.
    - Disco Duro de 5GB o superior.
    - Java v8 o superior.
    - La aplicación deberá funcionar correctamente independientemente del sistema operativo
  - Especificaciones de ejecución
    - Se recomienda tener las últimas actualizaciones del sistema operativo, Java y del navegador web para garantizar el funcionamiento de la aplicación.
    - Junto con la herramienta se entregará un manual para facilitar la compresión.

- Pliego de cláusulas administrativas:

    - La aplicación tendrá una velocidad de respuesta variable dependiendo del número de usuarios conectados a la red y del plan escogido para el servidor.

    - La disponibilidad está determinada por el servidor y por la calidad de su conexión.

    - La aplicación tendrá una capacidad variable dependiendo del servidor.

    - El cliente podrá accederse desde cualquier ordenador que permita conectarse la red del servidor.

Apéndice C.

Presupuesto.

En este apartado se hace una estimación de los costes que podría haber tenido este proyecto si se hubiese realizado por personal experto. En primer lugar, se define el grupo de trabajo:

- Jefe de proyecto. En este caso ocuparía el puesto de *Scrum Master*.
    - Realizar tareas de liderazgo del equipo.
    - Responsable de que se cumpla la metodología Scrum.
    - Responsable de hacer el análisis previo junto con el analista-programador.
    - Gestión de los tiempos del proyecto.
    - Se estima un salario bruto de 32€/h
- Analista – Programador. Pertenece al equipo Dev-Team de Scrum.
    - Su función principal es hacer un análisis previo al inicio del proyecto para conocer los requisitos de este.
    - Analizar las historias de usuario.
    - Realizar un diseño de la aplicación.
    - Realizar la implementación de la aplicación.
    - Realizar el manual de usuario.
    - Se estima un salario bruto de 22€/h.

- Experto en pruebas. Debe confirmar que el sistema es estable.
    - Realiza las pruebas de todo el sistema.
    - Se estima un salario bruto de 18€/h.
- *Product Owner*. Es la persona encargada del producto.
    - Toma las decisiones de lo que se debe construir y cuándo.
    - Valida cuando una funcionalidad sale al mercado.
    - Mantiene comunicación directa con el cliente final y con el jede del proyecto.
    - Se estima un salario bruto de 30€/h.

El presupuesto se va a realizar en base a una jornada parcial, es decir de lunes a viernes de 4h diarias, sin posibilidad de horas extra ni trabajo en días festivos. Las tarifas por hora estimadas son para trabajadores *freelance*. Además, las herramientas utilizadas eran gratuitas por tanto no se asume ningún gasto extra.

El presupuesto se basa en el *Product Backlo*g (Véase Tabla 55Tabla 55) que se hizo al inicio del proyecto donde se declaran las diferentes fases del proyecto y el coste de estas realizadas o bien por un experto o bien por un programador junior. En este caso, se va a incrementar la columna del desarrollador junior (B) un 20% ya que debido a la falta de experiencia la estimación de tiempo puede ser errónea. Debido a que se usa Scrum como metodología, en cada *Sprint* se incorporan las fases de análisis, diseño y pruebas, por tanto, todo el equipo está implicado. Para facilitar los cálculos, se va a deducir que todo el equipo usa las mismas horas para desarrollar su parte, salvo el *tester* que se le reduce el tiempo un 30%. (Véase Tabla 89)

	Experto	Junior
Sprint 0	40	80
Sprint 1	36	43,2
Sprint 2	28	63,6
Sprint 3	33	49,2
Sprint 4	33	45,6
Sprint 5	13	24
Sprint 6	40	48
Total de horas	223	353,6
En días	55,75	88,4

Tabla 89 Total de horas del proyecto comparando Senior vs Junior

Por tanto, para el supuesto de trabajo se cuenta con un *Scrum Master*, dos personas del equipo de desarrollo, un *tester* y un *Product Owner*. El presupuesto final quedaría como se muestra en la Tabla 90.

	Scrum Master 32€/h	**Dev Team 1 22€/h**	**Dev Team 2 22€/h**	**Tester 18€/h**	**Product Owner 30€/h**
Sprint 0	1280	880	880	0	1200
Sprint 1	1152	792	792	453,6	1080
Sprint 2	896	616	616	352,8	840
Sprint 3	1056	726	726	415,8	990
Sprint 4	1056	726	726	415,8	990
Sprint 5	416	286	286	163,8	390
Sprint 6	1280	880	880	504	1200
Coste por empleado	7136	4906	4906	2305,8	6690
Coste del proyecto	25943,8€				

Tabla 90 Presupuesto final del proyecto

Apéndice D

Manual de Usuario.

En este apartado se mostrará un manual de usuario pensado para permitir al usuario final manejar correctamente la herramienta. Se asume que el usuario tiene fundamentos básicos en el uso de hojas de cálculo.

El manual se encuentra dividido en varias partes:

- **Qué es LudaFarma Hospitalaria:** esta sección trata el propósito de la aplicación y sus funciones.

- **Cómo utilizar la aplicación web**: este apartado explica en detalle la interfaz web y cómo realizar acciones dentro de ella.

- **Cómo crear los archivos de hojas de cálculo:** se explican las normas que debe seguir el archivo de stock para que pueda ser entendido correctamente por la aplicación.

- **Preguntas frecuentes:** este apartado contiene preguntas y respuesta que pueden surgir durante el uso de la herramienta.

D.1 ¿Qué es LudaFarma Hospitalaria?

LudaFarma Hospitalaria es una aplicación web que permite conectar las farmacias hospitalarias de España a través de su stock.

El objetivo principal de este servicio es poder reducir los costes económicos y medio ambientales que se producen al no aprovechar las unidades restantes de determinados tipos de medicamentos.

Para ello, se ha puesto a disposición de la farmacia hospitalaria, un sistema que permite conocer si una farmacia dispone de unidades de un medicamento de alta impacto económico, permite reservarlo e iniciar el procese de préstamos entre los hospitales.

D.2 ¿Cómo utilizar la aplicación web?

El acceso al servicio web puede realizarse desde la siguiente web: http://tfgluda.us-east-1.elasticbeanstalk.com/login. A continuación, tiene dos opciones, o bien iniciar sesión con su correo electrónico y contraseña, o bien si no está registrado, pulsar en no estoy registrado y seguir el proceso de registro. (Véase Figura 39).

Figura 39 Inicio de sesión de LudaFarma Hospitalaria.

Tras iniciar sesión, aparece la pantalla principal desde la cual se pueden realizar las siguientes acciones: (Véase Figura 40).

- Subir el stock de la farmacia a través de una hoja de cálculo.
- Subir un producto al stock de la farmacia de manera manual.
- Editar los productos que tiene la farmacia en su stock.
- Realizar un nuevo encargo.

- Ver el estado de los encargos solicitados y recibidos.

Figura 40 Página principal del usuario.

Además de estas funciones, desde la cabecera, presente en todas las páginas, se puede acceder al soporte técnico de la web vía email, se pueden editar los datos de usuario o cerrar sesión entre otras acciones.

Para poder subir el stock de manera automática, es necesario introducir el CIF de la farmacia y un archivo con extensión hoja de cálculo que contenga el stock de la farmacia que se quiere publicar en la web. (Véase Figura 41).

Figura 41 Subida de stock automática.

En caso de que no se quiera subir una gran cantidad de productos y solo se quiera dar de alta determinados productos, es posible gracias a la opción de subida de stock manual. Para ello, solo es necesario introducir el código nacional del producto que se quiera añadir, darle a buscar y una vez que el sistema muestre el producto, indicar las unidades disponibles en la farmacia. (Véase Figura 42).

Figura 42 Añadir un producto al stock de la farmacia.

Si, por el contrario, se quiere borrar o modificar las unidades disponibles de un producto, desde el botón gestiona tu stock es posible.

Para borrar un producto, hay que hacer clic sobre el botón borrar del producto que se quiera eliminar del stock, esta acción es definitiva, no se puede volver atrás. En cambio, si se desea editar el número de unidades disponibles, hay que hacer clic sobre el botón editar (Véase Figura 43) y a continuación, se busca por código nacional el producto que se quiere modificar. Por último, se introduce el nuevo valor de las unidades, antes de pulsar sobre aceptar. (éase Figura 44).

Figura 43 Stock de la farmacia.

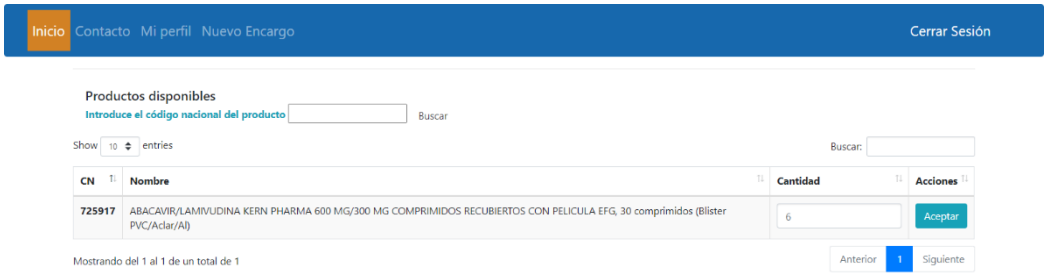

Figura 44 Editar las unidades de un producto.

Otra de las funciones disponibles es la de realizar un encargo, para ello, de manera similar, se busca el producto deseado por código nacional y aparecerán una serie de farmacias ordenadas por cercanía. A continuación, se elige la farmacia en la que se quiere reservar (Véase Figura 46) y se completan los datos requeridos. Con la confirmación del encargo, llega un aviso por email a las dos farmacias involucradas. (Véase Figura 45).

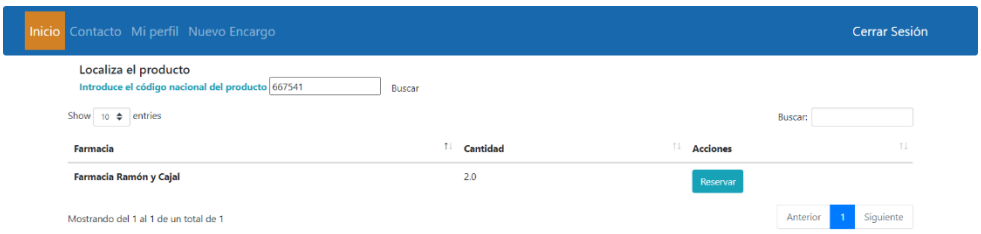

Figura 45 Datos del encargo.

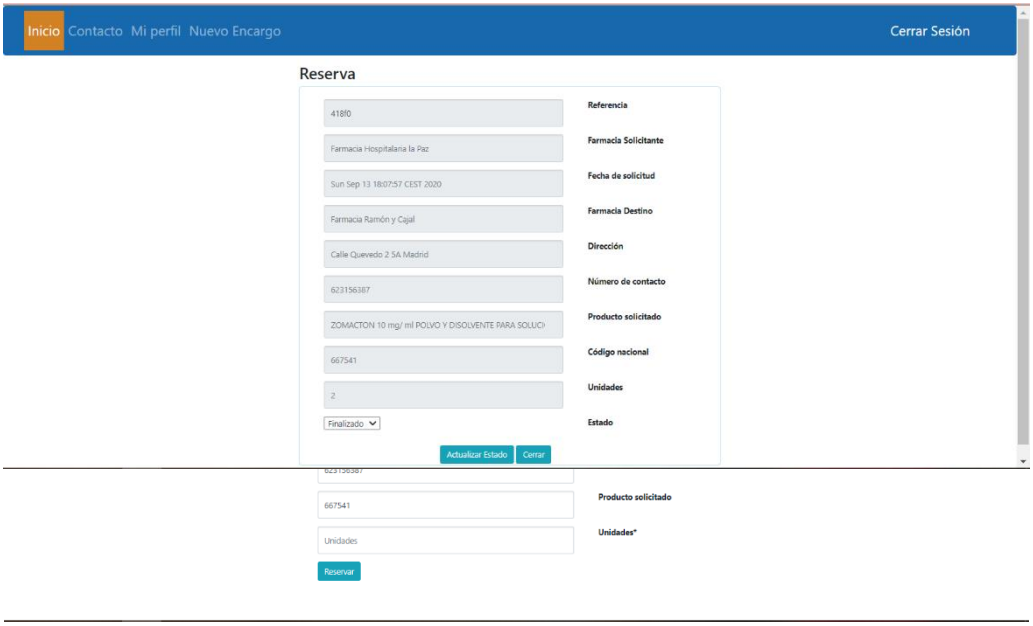

Figura 47 Estado del encargo.

Por último, los encargos solicitados y recibidos se pueden gestionar desde el apartado, gestionar encargos. Están clasificados en función de si los ha solicitado la farmacia o lo has recibido. Además, desde esta pantalla se puede ver el estado en el que se encuentran los encargos, y modificarlo. (Véase Figura 47).

Además de los servicios principales, la aplicación ofrece contacto directo con el servicio de soporte técnico y permite cambiar los datos del usuario desde la opción habilitada en la cabecera.

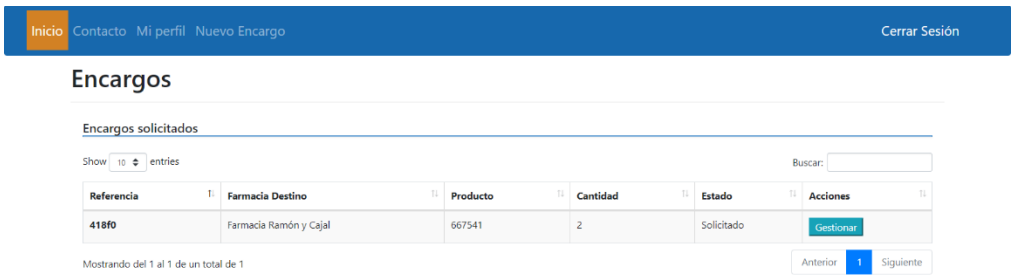

Figura 48 Listado de encargos solicitados.

D.3 ¿Cómo crear los archivos de hojas de cálculo?

El sistema necesita una plantilla específica del archivo de stock para poder subirlo a la plataforma. El archivo debe ser una hoja de cálculo y debe estar formado por las siguientes columnas:

- Columna 1: Nombre del medicamento.
- Columna 2: Código nacional del medicamento.
- Columna 3: Laboratorio.

El archivo no debe contener cabecera ni títulos.

D.4 Preguntas frecuentes.

- ¿Quién puede utilizar el sistema?

El sistema está enfocado exclusivamente para la farmacia hospitalaria. La farmacia comunitaria dispone de su propio servicio.

- ¿Es necesario completar la documentación de préstamo?

LudaFarma Hospitalaria es una red de farmacias que permite conocer el stock de una farmacia y reservar un producto en ella, sin embargo, el farmacéutico debe cumplir con la normativa vigente y realizar la documentación requerida por el Ministerio para realizar dicho intercambio. En líneas de futuro se podrá añadir un modelo de préstamo estandarizado en la web.

- ¿Para qué se utilizan los datos?

LudaFarma Hospitalaria no usa los datos con ningún otro fin más que el de localizar un producto en una farmacia y en un momento determinado. Los datos son de la farmacia hospitalaria, no se comunican a terceros.

- He encontrado un fallo en la aplicación, ¿qué debo hacer?

Cualquier anomalía que detecte debe ser comunicada por email en el formulario de contacto que se encuentra en su perfil. Si lo prefiere, siempre puede mandar un email a info@ludapartners.com.

- ¿Cuánto tarda el envío de medicamentos de un hospital a otro?

El transporte corre a cargo de la farmacia hospitalaria, por tanto, no podemos fijar un tiempo máximo de entrega.

Parte IV

Bibliografía

Bibliografía

[1] M. Peralta, «Monografías,» [En línea]. Available: https://www.monografias.com/trabajos7/sisinf/sisinf.shtml.

[2] U. Hernández, «Código Facilito,» 22 Febrero 2015. [En línea]. Available: https://codigofacilito.com/articulos/mvc-model-view-controller-explicado. [Último acceso: Febrero 2020].

[3] A. Schiaffarino, «Infra Networking,» 12 Marzo 2019. [En línea]. Available: https://blog.infranetworking.com/modelo-cliente-servidor/. [Último acceso: Marzo 2020].

[4] «Edu4java,» Octubre 2018. [En línea]. Available: http://www.edu4java.com/es/servlet/servlet1.html. [Último acceso: Febrero 2020].

[5] «Metodologías Ágiles,» [En línea]. Available: https://proyectosagiles.org/que-es-scrum/.

[6] Y. Muradas, «OpenWebinars,» 5 Junio 2018. [En línea]. Available: https://openwebinars.net/blog/conoce-que-es-spring-framework-y-por-que-usarlo/.

[7] P. Pérez, «La Razón,» 20 Marzo 2003. [En línea]. Available: https://www.larazon.es/atusalud/salud/por-que-hay-problemas-de-suministro-de-farmacos-en-espana-OF22503940/.

[8] C. Arganda, «DiarioFarma,» 18 Noviembre 2016. [En línea]. Available: https://www.diariofarma.com/2016/11/18/medicamentos-alto-coste-alta-prevalencia-nuevo-reto-las-ccaa.

[9] E. Fariña, «El médico interactivo,» 22 Octubre 2019. [En línea]. Available: https://elmedicointeractivo.com/ya-funciona-valtermed-para-medicamentos-de-alto-impacto-sanitario-y-economico/.

[10] U. d. Cauca, «Universidad del Cauca,» [En línea]. Available: http://fccea.unicauca.edu.co/old/siconceptosbasicos.htm.

[11] R. Fa, «GenBeta,» 3 Febrero 2014. [En línea]. Available: https://www.genbeta.com/desarrollo/mongodb-que-es-como-funciona-y-cuando-podemos-usarlo-o-no.

[12] J. Garzas, «Javier Garzas,» 6 Junio 2014. [En línea]. Available: https://www.javiergarzas.com/2014/06/maven-en-10-min.html.

[13] M. Poletti, «Rsearch Gate,» Octubre 2015. [En línea]. Available: https://www.researchgate.net/figure/Figura-3-Arquitectura-Cliente-Servidor_fig3_283302810. [Último acceso: Marzo 2020].

[14] A. Urteaga, «Universidad Carlos III,» Septiembre 2015. [En línea]. Available: https://e-archivo.uc3m.es/bitstream/handle/10016/23750/TFG_Aitor_Urteaga_Pecharroman.pdf?sequence=1&isAllowed=y. [Último acceso: Marzo 2020].

[15] B. Author, «Baeldung,» Junio 2020. [En línea]. Available: https://www.baeldung.com/java-config-spring-security. [Último acceso: Agosto 2020].

[16] C. A. Caules, «Arquitectura Java,» 30 Agosto 2017. [En línea]. Available: https://www.arquitecturajava.com/spring-security-annotation-y-su-configuracion/. [Último acceso: Julio 2020].

[17] Farmatic, *Farmatic software de gestión farmacéutica.,* Barcelona, 1998.

[18] Farmanager, *Farmanager, software de gestión de Farmacias,* Sevilla, 1998.

[19] A. S. CABEZUELO, *INTRODUCCIÓN A LAS BASES DE DATOS NOSQL USANDO MONGODB,*

S.L. EDITORIAL UOC, 2014.

[20] Apache , «Spring,» [En línea]. Available: https://spring.io/projects/spring-framework. [Último acceso: Julio 2020].

[21] Github, «Bootstrap,» 19 Agosto 2011. [En línea]. Available: https://getbootstrap.com/. [Último acceso: Marzo 2020].

[22] J. D. Davidson, «Tomcat Apache,» 1999. [En línea]. Available: http://tomcat.apache.org/. [Último acceso: Febrero 2020].

[23] Amazon, «Amazon Web Services,» 2010. [En línea]. Available: https://aws.amazon.com/es/. [Último acceso: Septiembre 2020].

[24] Apache, «Maven,» [En línea]. Available: https://maven.apache.org/. [Último acceso: Marzo 2020].

[25] «Spring MVC Documentation,» 2002. [En línea]. Available: https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html#spring-web.

[26] Amazon Web Services, «Elastic Beanstalk Documentation,» 2018. [En línea]. Available: https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/Welcome.html. [Último acceso: Septiembre 2020].

[27] MongoDB, «MongoDB Documentation,» 2008. [En línea]. Available: https://docs.mongodb.com/. [Último acceso: Agosto 2020].

[28] J. Francia, «SCRUM,» 25 Septiembre 2017. [En línea]. Available: https://www.scrum.org/resources/blog/que-es-scrum. [Último acceso: Marzo 2020].