

FirstMarket

Misrraim Suárez Pérez

20 de julio de 2020

Índice

1. Introducción	2
1.1. Motivación y Objetivos	3
1.2. Internet, Web y Aplicaciones Web	3
1.2.1. ¿Qué es Internet?	4
1.2.2. ¿Qué es la Web?	5
1.2.3. ¿Qué es una Aplicación Web?	6
1.3. Estructura de la Memoria	6
2. Análisis	7
2.1. Requisitos	7
2.2. Casos de Uso	9
3. Diseño	16
3.1. Arquitectura del Sistema	16
3.1.1. Inversión de Control	18
3.1.2. Capas funcionales	19
3.2. Modelo de Datos	23
4. Implementación	23
4.1. Stack Tecnológico	24
4.1.1. Git	24
4.1.2. Java	28
4.1.3. Apache Maven	33
4.1.4. Spring	35
4.1.5. Thymeleaf	41
4.1.6. Bootstrap	42
4.2. Pagos con Stripe	43
4.3. Control de Concurrency	48
4.4. Datos Jerárquicos	52

4.5. Experiencia de Usuario Mejorada	55
4.6. Libros Referenciados	57
5. Despliegue	59
5.1. Heroku	60
5.2. Heroku Add-ons	61
5.3. firstmarket.tech	62
6. Seguridad	63
6.1. HTTPS	63
6.2. Spring Security	65
6.2.1. FilterChain	65
6.2.2. Configuración	67
6.2.3. Autenticación	69
6.2.4. Autorización	70
6.2.5. Cross-Site Request Forgery	71
6.2.6. Encabezados HTTP	73
6.3. Validación de Entradas	76
6.4. Stripe Radar	77
6.5. Análisis de Vulnerabilidades	78

1. Introducción

El presente proyecto se enmarca dentro del Proyecto Final de Grado correspondiente al Grado en Ingeniería Informática de la Universidad Nacional de Educación a Distancia.

Se basa en el proyecto específico ofertado por el Departamento de Sistemas de Comunicación y Control de la UNED titulado *Desarrollo de un portal de comercio electrónico*. Citando la propia descripción de dicha oferta de proyecto específico:

El proyecto consiste en el desarrollo de un portal web orientado al comercio electrónico. Dicha aplicación permitirá al comprador seleccionar artículos, realizar pedidos y pagos a través de una pasarela. El sistema también ofrecerá al comerciante gestionar los artículos expuestos en el portal.

En este contexto, el comercio electrónico desarrollado a sido una librería, denominada **FirstMarket**.

1.1. Motivación y Objetivos

Desde un primer momento la intención fue desarrollar un proyecto que tuviese la mayor relación posible con el mercado actual, lo que, junto con un fuerte interés personal hacia el ecosistema de Internet, llevó a la decisión de escoger esta propuesta.

La idea era desarrollar algo moderno, útil, con reflejo real en la sociedad y que cumpliera el papel de facilitar la inserción laboral. En este sentido, las aplicaciones web en general, y las específicas de comercio electrónico en particular, cumplen a la perfección los requisitos comentados. A nadie se le escapa hoy en día la implantación casi ubicua que estas tecnologías tienen en la sociedad.

Por otro lado, y en clara conexión con lo comentado, otro objetivo de este último trabajo del plan de estudios fue el de servir como *compensador* final de lagunas formativas. En este sentido, el control de versiones con Git era una tarea pendiente inaplazable, así como profundizar en tecnologías web básicas como HTTP, HTML, CSS ó JavaScript. Además, ampliar el dominio de las tecnologías Java siempre fue algo muy deseable.

Conviene no dejar de lado, por obvio, que un objetivo fundamental del presente trabajo es el de dar fin al plan de estudios. Remarcar esto es relevante porque en múltiples situaciones puede entrar en conflicto con la intención de desarrollar una aplicación web lo más moderna posible. Como se explicará más adelante en esta memoria, el mundo de las tecnologías web es muy cambiante, y pretender desarrollar una aplicación web alineada con el estado del arte en la materia partiendo, como es el caso, desde prácticamente cero conocimientos, sale fuera del marco de un trabajo de estas características. Por tanto, siempre se ha tenido en cuenta un compromiso entre recursos disponibles (principalmente tiempo y conocimientos) y grado de modernidad en las tecnologías usadas.

En definitiva, el principio fundamental que ha guiado la toma de decisión ha sido el reforzar, y alinear en la medida de lo posible con el estado del arte, las habilidades técnicas adquiridas fruto del plan de estudios, de forma a facilitar una futura integración en la industria web, a la par de satisfacer una histórica inquietud personal en la materia.

1.2. Internet, Web y Aplicaciones Web

En esta sección se ofrece una introducción, meramente descriptiva, del contexto en el que situar la aplicación web desarrollada. Se tratará de diferenciar los conceptos de Internet, World Wide Web y aplicaciones web.

1.2.1. ¿Qué es Internet?

Tal como se explica en [bib ref], esta pregunta puede ser enfocada desde dos puntos de vista complementarios.

Por un lado, desde una perspectiva **hardware**, Internet es una red que conecta miles de millones de dispositivos en todo el mundo, denominados *hosts* o *sistemas finales*, por medio de *enlaces de comunicación* y *conmutadores de paquetes*.

Los enlaces de comunicación son diferentes tipos de medios físicos a través de los cuales se transfieren ondas electromagnéticas que portan la información. Puede tratarse de medios guiados, como cables de cobre, cables coaxiales o fibra óptica, o no guiados, como el espectro de radio. Cuando un sistema final decide enviar información a otro sistema final, el emisor fragmenta los datos en *paquetes* de información y los envía al destinatario a través de la red. Una vez en destino, el sistema final receptor ensambla los paquetes para reconstruir los datos originales. Un conmutador de paquetes toma un paquete que llega a uno de sus enlaces de comunicación entrantes y decide hacia cuál de sus enlaces de comunicación salientes lo reenvía.

Estas *redes de conmutación de paquetes* se pueden entender de forma análoga a una empresa que necesita mover una gran cantidad de carga entre dos almacenes separados gran distancia. En el almacén de origen la carga se divide y organiza en diferentes contenedores. Cada uno de los contenedores viaja de forma independiente a través de la red de transporte disponible siguiendo posiblemente rutas no del todo iguales. Por ejemplo, unos contenedores pueden ir por ciertas carreteras en ciertos camiones mientras que otros pueden viajar en tren o incluso en barco, o cualquier combinación de los anteriores. Lo importante es la independencia entre la ruta de un contenedor concreto con los demás. Una vez llegados al almacén de destino, la carga se extrae de los contenedores y se agrupa con el resto que llega del mismo envío. Así, los paquetes son análogos a los contenedores, los enlaces de comunicación son análogos a las vías de transporte, los conmutadores de paquetes son análogos a las intersecciones o discontinuidades en las vías de transporte (piénsese en una simple rotonda), y los sistemas finales son análogos a los almacenes. Pues bien, de igual forma que un contenedor toma un camino a través de la red de transporte, un paquete de información toma un camino a través de la infraestructura de Internet.

Por otro lado, es posible describir Internet desde un punto de vista **software** como un servicio prestado a las aplicaciones distribuidas, es decir, como una interfaz de programación que las aplicaciones distribuidas consumen.

Se dice que las aplicaciones son aplicaciones distribuidas, o **aplicaciones de Internet**, cuando se ejecutan en diferentes máquinas que intercambian datos

entre sí. Es importante destacar que las aplicaciones de Internet se ejecutan estrictamente en los sistemas finales, no en la infraestructura de la red, que es agnóstica (o debiera serlo) de la semántica de la información que está transportando. Así, puede pensarse en Internet como un servicio postal, que garantiza el envío de información entre partes, las cuales pueden estar desarrollando cualquier tipo de actividad basada, entre otras cosas, en la propia comunicación que mantienen. Al igual que el servicio postal impone una serie de reglas para ser usado, como dónde depositar la carta que se pretende enviar o dónde y de qué manera especificar la dirección de envío, Internet ofrece unas reglas en forma de interfaz de programación que las aplicaciones de Internet deben adoptar.

1.2.2. ¿Qué es la Web?

Siguiendo el hilo de la discusión del epígrafe anterior, la World Wide Web es una de las muchas aplicaciones de Internet existentes. Otras serían el correo electrónico, las aplicaciones para acceder remotamente a otra máquina, la transferencia de archivos, el streaming de video, la telefonía por Internet, y más.

A veces, dado que la Web es la aplicación más conocida, se confunde la parte con el todo al identificarla con la propia Internet. Sin embargo, esta aplicación surgió bastante tiempo después de que otras ya estuvieran ampliamente implantadas y maduras, como el correo electrónico o la transferencia de ficheros, si bien su uso era principalmente en ámbitos académicos.

Lo que da tanta importancia a la Web es que fue la aplicación de Internet que, a principios de los años 90, abrió al gran público a la hasta entonces desconocida Internet. De hecho, esta aplicación llevó a Internet de ser tan sólo una más de muchas redes existentes a ser la dominante por excelencia.

Entonces, como se ha dicho, la Web es una aplicación de Internet, y como tal se trata de software que se ejecuta en diferentes máquinas que intercambian mensajes entre sí. El formato de estos mensajes, su timing y demás *reglas de conversación* se recogen en un protocolo, el HyperText Transfer Protocol (HTTP). Este protocolo es el corazón de la Web, encontrándose definido en [RFC 1945] y en [RFC 2616], y su implementación se materializa en dos programas ejecutados en diferentes sistemas finales, un programa cliente y un programa servidor.

La mecánica básica gira en torno a peticiones y respuestas. El programa cliente envía un mensaje HTTP conteniendo una petición al programa servidor. Por su parte, siguiendo las reglas definidas en el protocolo, el programa servidor contesta enviando otro mensaje HTTP al cliente. En el caso más común, el cliente le solicita al servidor el envío de una *página web*.

Una página web es un documento de texto escrito con unas reglas de sintaxis específicas. Estas reglas definen el Hypertext Markup Language (HTML). En

el sistema final cliente, la página web es presentada en pantalla de una manera amigable al usuario por medio de programas que procesan el contenido en HTML.

En definitiva, y sin entrar en matices que desvíen de la esencia, la Web es la aplicación de Internet que permite el consumo de páginas web bajo demanda por parte de los usuarios (en contraste con el modelo broadcast, en el que se emite y el usuario sólo puede consumir lo emitido, como en la radio).

1.2.3. ¿Qué es una Aplicación Web?

Las páginas web que son enviadas desde el servidor al cliente se dice que pueden ser estáticas o dinámicas. Esta característica, más que hablar de la página web en sí misma, habla del proceso mediante el cual su contenido ha sido creado y modificado a lo largo del tiempo.

Las páginas estáticas lo son en el sentido de que su contenido, creado comúnmente por un humano, no varía con el tiempo de manera programática. La información que presentan es la misma, a menos que se modifique *a mano*. Así eran todas las páginas web en los primeros años. En contraste, en las páginas web dinámicas, el contenido no está dado de antemano, sino que se genera en cada ciclo de petición-respuesta. Esto es útil porque entre otras cosas permite páginas web personalizadas para cada usuario, e incluso que sea él mismo quien aporte contenido a la página web.

Una **aplicación web** es el sistema software encargado de la generación dinámica de páginas webs. Normalmente, y a diferencia del modelo estático, entre sus componentes se encuentra una base de datos que permita la persistencia de la información, cambiante por definición de la propia arquitectura.

Desde este punto de vista, en el presente proyecto se ha desarrollado una aplicación web encargada de generar contenido web que los usuarios pueden visitar con el objetivo de comprar libros a través de Internet.

1.3. Estructura de la Memoria

El resto de la presente memoria está dedicado a presentar los aspectos más relevantes del trabajo desarrollado.

En la sección 2 se detallan los aspectos relacionados con la fase de análisis en el desarrollo de la aplicación web. Para ello, se comentan los requisitos de partida y se ofrecen varios casos de uso.

La sección 3 está enfocada a las decisiones de diseño. En ella se esbozan las principales coordenadas de la aplicación web que se ha desarrollado, discutiendo las decisiones tomadas.

El apartado 4 se ha utilizado para describir y justificar las decisiones de implementación. Se tratan en esta sección aspectos tan importantes como la tecnología que se ha usado para el desarrollo, la manera de llevar a cabo la gestión de los pagos, o aspectos internos de código que por su especial relevancia se ha decidido explicar.

Después, en la sección 5, se comentan los pasos seguidos para conseguir que la aplicación web sea accesible al público. Aunque esto no fuese un requisito de partida, se ha considerado muy deseable, ya que sirve para aprender y ganar experiencia en el mundo de las tecnologías web.

El siguiente apartado, el 6, está focalizado en los aspectos de seguridad que rodean a FirstMarket. Es un apartado clave, que, aunque tampoco estaba contenido en los requisitos básicos, se ha querido profundizar en él. Así, se describen las principales características de seguridad implementadas en firstmarket.tech, además de un análisis de las vulnerabilidades detectadas.

2. Análisis

En esta sección se aborda la fase de análisis del desarrollo software. Partiendo de la especificación de los requisitos básicos que debe implementar la aplicación web, se ofrece un conjunto de casos de uso que dé una visión de cuál debe ser el flujo de uso.

2.1. Requisitos

La aplicación, fundamentalmente, debe admitir los roles y capacidades siguientes:

1. Usuario anónimo (UA).
 - Por defecto, al acceder al sitio web se hace como UA, sin ninguna validación ni credencial. Basta con acceder a la URL de inicio de la aplicación.
 - Un UA debe poder realizar búsquedas de libros, es decir, debe tener pleno acceso a la exploración del catálogo.
 - La plena exploración del catálogo debe permitir realizar búsquedas filtradas según 0, 1 o más criterios, tales como: categoría, título o autor.
 - Un UA debe poder visualizar información detallada de un libro, por ejemplo de entre los obtenidos tras una búsqueda.
 - Un UA debe poder consultar las promociones disponibles.

- Un UA debe poder registrarse en el sistema completando un formulario (nombre, contraseña, dirección de correo electrónico, etc.).
 - Finalizado el proceso de registro, el nuevo UR debe recibir confirmación por correo electrónico.
 - En su caso, un UA debe poder iniciar sesión en el sistema.
 - En su caso, un UA debe poder realizar el procedimiento de recuperación de contraseña.
2. Usuario registrado (UR). Este perfil representa a un usuario que ha pasado de anónimo a registrado. Un UR posee todas las capacidades del UA, más otras específicas suyas, a saber:
- Poder editar la información de su perfil de usuario.
 - Realizar pedidos y efectuar los correspondientes pagos a través de una pasarela segura.
 - Disponer de una cesta virtual para la gestión de la compra.
 - En la cesta se debe poder introducir, modificar la cantidad o eliminar libros (esto último de uno en uno o todos a la vez).
 - En cualquier momento del proceso de realizar un pedido, el UR debe poder cancelarlo.
 - Tras una compra, el UR debe recibir confirmación en su correo electrónico.
 - Un UR debe poder consultar el estado de sus pedidos.
 - Puntuar (de alguna manera, p.e. estrellas del 1 al 5) un determinado libro que haya adquirido. Debe poder hacerlo en cualquier momento tras la compra.
 - Consultar un histórico de sus transacciones, detallando los libros comprados, la fecha de la compra y el precio de cada uno.
 - Darse de baja como UR.
 - Cerrar sesión.
 - Un UR debe poder ponerse en contacto con el administrador de la aplicación web a través de un formulario de contacto, recibiendo confirmación por correo electrónico tras el envío del mismo.
3. Usuario Administrador
- Ver y editar (añadir, modificar, eliminar) la jerarquía de categorías (CRUD categorías).

- Ver y editar (añadir, modificar, eliminar) la información relativa a los libros (título, autor/es, editorial, precio, disponibilidad, ...) (CRUD libros).
- Crear, modificar o eliminar promociones de libros (CRUD promociones).
- Tener acceso a la información de los UR, salvo sus contraseñas.
- Bloquear-desbloquear a un UR.
- Visualizar la información de los pedidos, tanto los que estén en curso como los finalizados.
- Poder alterar el estado de un pedido.
- Poder generar informes (p.e. ventas durante un determinado periodo con su importe y la facturación total).

Además de lo anterior, la aplicación debe:

- Garantizar la persistencia de los datos referentes a UR, pedidos, pagos, productos y sus categorías.
- Mostrar un mensaje de error cuando un usuario introduzca incorrectamente sus credenciales de autenticación.

2.2. Casos de Uso

En este apartado se presentan las interacciones más comunes que los usuarios pueden realizar con la aplicación web. No se pretende proporcionar una enumeración exhaustiva de todos los casos de uso, sino un subconjunto relevante de los mismos, a modo de introducción a las capacidades básicas que se espera de la aplicación.

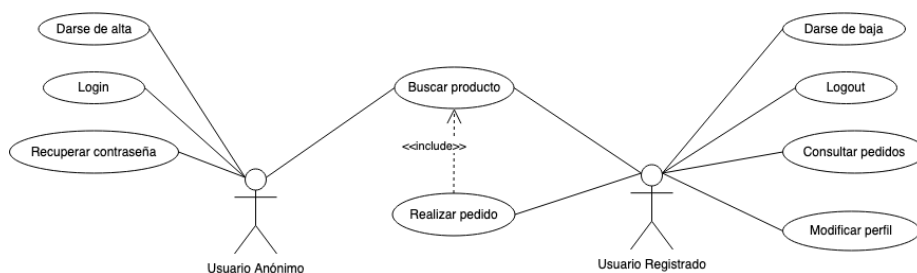


Figura 1: Diagrama de casos de uso

CU01.Búsqueda Proceso por el cual un usuario explora el catálogo y acaba por visualizar la página de un libro.

- + Actores implicados: Usuario Anónimo, Usuario Registrado.
- + Precondiciones: Usuario situado en cualquiera de las páginas que dan acceso al catálogo.
- + Flujo principal:
 1. El usuario hace click en alguna de las categorías principales mostradas a modo de catálogo flotante en la barra de navegación o hace click en *All Books*.
 2. El sistema muestra la página de resultados de búsqueda.
 3. El usuario hace click en alguno de los productos mostrados.
 4. El sistema muestra la página del libro.
- + Flujo alternativo: *refinamiento_búsqueda*. El usuario altera en la página de resultados algún criterio de filtro.
 - 3.b. El usuario selecciona/deselecciona algún criterio de filtro de entre los mostrados en la página de resultados de búsqueda.
 - 4.b. El sistema vuelve al punto 2 del flujo principal.
- + Flujo alternativo: *búsqueda_por_texto*. El usuario introduce una cadena de texto en el cuadro de búsqueda.
 - 1.b. El usuario introduce una cadena de texto en el cuadro de búsqueda y hace click en buscar.

CU02.Alta Proceso por el cual se crea un nuevo usuario registrado.

- + Actores implicados: Usuario Anónimo.
- + Flujo principal:
 1. El usuario hace click en el link de nuevo registro, disponible en la página de *login*.
 2. El sistema presenta un formulario donde introducir la dirección de email y la contraseña.
 3. El usuario introduce y envía la información pedida.
 4. El sistema comprueba la información proporcionada.
 5. El sistema crea una nueva cuenta de usuario, pero la mantiene inactiva a la espera de confirmar la dirección de email.

6. El sistema envía un email con un link de confirmación a la dirección proporcionada, e informa al usuario por pantalla.
 7. El usuario accede a su email y hace click en el link enviado, confirmando que la dirección de email es suya.
 8. El sistema activa la cuenta de usuario.
 9. El sistema envía al usuario un email de bienvenida.
 10. El sistema redirige al usuario a la página de *login*.
- + Flujo alternativo: *usuario_ya_registrado*. La dirección proporcionada se encuentra registrada en el sistema.
- 5.b. El sistema no crea una nueva cuenta de usuario. Por razones de seguridad, la manera en que el sistema informa al usuario tras esta situación es indistinguible del flujo principal, de forma que no se pueda deducir que ese email ya tiene cuenta asociada en el sistema.
- + Flujo alternativo: *link_ya_enviado*. El sistema está a la espera de la confirmación de un link válido en esta dirección de email.
- 5.b. El sistema no crea una nueva cuenta de usuario. Se informa al usuario acerca de una condición de error genérica, por seguridad en relación con la dirección de email proporcionada, pidiéndose que compruebe su dirección de email.
- + Flujo excepcional: *time_out*. El usuario no confirma su dirección de email dentro de un plazo determinado.
- 7.b. El sistema detecta el timeout e invalida el link de confirmación enviado. Si el usuario hace click en el link caducado se le informa de dicha condición.

CU03 Login Proceso por el cual un usuario se autentica en el sistema.

- + Actores implicados: Usuario Anónimo.
- + Flujo principal:
1. El usuario hace click en el link de *login*, o intenta realizar alguna operación que requiera autenticación (por ejemplo, añadir un libro a la cesta).
 2. El sistema presenta el formulario de acceso (dirección de email y la contraseña).
 3. El usuario introduce y envía la información pedida.

4. El sistema comprueba las credenciales.
 5. El sistema redirige al usuario a la página principal.
- + Flujo alternativo: *fallo_autenticación*. El email no se encuentra registrado en el sistema, o la contraseña proporcionada no es correcta.
- 5.b. El sistema informa al usuario acerca de un fallo genérico de autenticación, de forma que, por razones de seguridad, no se pueda deducir si el email proporcionado se encuentra registrado en el sistema.
- + Flujo alternativo: *bloqueo_cuenta*. El usuario anónimo realiza, dentro de un marco de tiempo (configurable), un número de intentos (configurable) de login con contraseña errónea.
- 5.b. El sistema bloquea la cuenta asociada al email para prevenir ataques por fuerza bruta.
 6. El sistema informa al usuario por pantalla y mediante el envío de un email.
 7. Pasado el tiempo de seguridad (configurable), el sistema desbloquea la cuenta del usuario.
- + Post-Condiciones: El usuario pasa a tener rol de usuario registrado en el sistema.

CU04.RecuperarContraseña Un usuario previamente registrado en el sistema intenta acceder al mismo, pero no recuerda su contraseña. El sistema intentará crear una nueva contraseña y enviarla al e-mail del usuario.

- + Actores implicados: Usuario Anónimo.
- + Flujo principal:
1. El usuario hace click en la opción *¿olvidó su contraseña?*.
 2. El sistema presenta un formulario donde introducir la dirección de email.
 3. El usuario introduce y envía su dirección de email.
 4. El sistema comprueba la dirección de email.
 5. Comprobación correcta. El sistema envía un email con un link de confirmación a la dirección proporcionada, e informa de ello al usuario por pantalla.
 6. Link no caducado. El usuario accede a su email y hace click en el link, confirmando que efectivamente la dirección proporcionada es la suya.

7. El sistema genera una nueva contraseña y se la asigna al usuario.
 8. El sistema envía la nueva contraseña a la dirección de email del usuario, y le avisa de ello por pantalla.
- + Flujo alternativo: *usuario_no_registrado*. La dirección proporcionada no se encuentra registrada en el sistema.
 - 5.b. El sistema no envía correo alguno pero, por razones de seguridad, desde el punto de vista del usuario este flujo alternativo es indistinguible del principal.
 - + Flujo excepcional: *time_out*. El usuario no confirma su dirección de email dentro de un plazo determinado.
 - 6.b. El sistema detecta el timeout e invalida el link de confirmación enviado. Si el usuario hace click en el link caducado se le informara de dicha condición.

CU05.EditarPerfil Proceso por el cual un usuario modifica alguno de los datos de su perfil personal.

- + Actores implicados: Usuario Registrado.
- + Flujo principal:
 1. El usuario hace click en *área personal*.
 2. El sistema muestra la página de área personal, en donde se muestra el formulario de datos personales, rellenado con la información actual del usuario.
 3. El usuario modifica los datos del formulario y lo envía al sistema.
 4. El sistema comprueba los datos.
 5. Comprobación correcta.El sistema actualiza la información del usuario.
 6. El sistema muestra la página de inicio.
- + Flujo alternativo: *error_datos_formulario*. Los datos proporcionados en el formulario de información personal no son válidos.
- 5.b. El sistema regresa al punto 3 del flujo principal, e informa al usuario acerca del fallo producido.
- + Flujo alternativo: *cancelar*. El usuario cancela el proceso de edición de su información personal.

3.b. El usuario hace click en el botón *cancelar*. El sistema muestra la página de inicio.

+ Post-Condiciones: El usuario ha alterado su información personal almacenada en la aplicación.

CU06.RealizarPedido Proceso por el cual un usuario realiza una compra a través de la aplicación web.

+ Actores implicados: Usuario Registrado.

+ Precondiciones: Desde cualquier página en la que se muestren libros (página de inicio, página de resultados de búsqueda o página de un libro) el usuario hace click en *add to cart*, para uno o mas libros.

+ Flujo principal:

1. El usuario hace click en el icono de la cesta.
2. El sistema muestra la página de la cesta del usuario, con todos los libros contenidos en el mismo.
3. El usuario puede aumentar o disminuir el número de unidades de un libro, o incluso eliminarlo por completo, y hace click en *checkout*.
4. El sistema comprueba la disponibilidad de los libros contenidos en la cesta.
5. Stock suficiente. El sistema muestra la página de checkout, con el formulario de datos de envío y pago, y el resumen de la compra.
6. El usuario completa los datos de envío y de pago y hace click en *Pay*.
7. El sistema tramita el pago.
8. Pago ok. El sistema registra el nuevo pedido.
9. El sistema confirma al usuario por pantalla y por email que el pedido se ha realizado con éxito.

+ Flujo alternativo: *stock_insuficiente*. No hay stock suficiente para satisfacer el contenido de la cesta.

5.b. El sistema vuelve al punto 2 del flujo principal, informando al usuario de los libros para los cuales no hay stock suficiente.

3.b. El usuario reduce la cantidad demandada de los libros correspondientes y hace click en *checkout*.

+ Flujo alternativo: *error_pago*. Error al efectuar el pago.

- 8.b. El sistema vuelve al punto 5 del flujo principal, informando al usuario del problema respecto al pago.
- + Flujo excepcional: *libro_eliminado*. Algún libro de la cesta ya no esta disponible en el sistema.
- 5.b. El sistema elimina de la cesta automáticamente los libros que el administrador haya deshabilitado.
- 6.b. El sistema vuelve al punto 2 del flujo principal, informando al usuario de los libros que han sido eliminados.
- + Post-Condiciones: El usuario ha efectuado un nuevo pedido.

CU07.ConsultarPedido Proceso por el cual un usuario consulta el historial de pedidos que ha realizado.

- + Actores implicados: Usuario Registrado.
- + Flujo principal:
 1. El usuario hace click en *My Purchases* en la barra de navegación.
 2. El sistema muestra la página de los pedidos del usuario.
 3. El usuario puede expandir-contrair la información mostrada relativa a un pedido.

CU08.Baja Proceso por el cual un usuario elimina su cuenta de la tienda online.

- + Actores implicados: Usuario Registrado.
- + Flujo principal:
 1. El usuario hace click en *área personal*.
 2. El sistema muestra la página de área personal.
 3. El usuario selecciona *eliminar cuenta*.
 4. El sistema muestra el formulario de eliminación de cuenta.
 5. El usuario completa y envía el formulario.
 6. El sistema comprueba los datos.
 7. Comprobación correcta. El sistema actualiza el estado del usuario y envía un email de confirmación.
 8. El sistema muestra la página de inicio.

- + Flujo alternativo: *error_datos_formulario*. La contraseña proporcionada en el formulario de baja no es correcta.
 - 7.b. El sistema regresa al punto 4 del flujo principal, e informa al usuario acerca del fallo producido.
- + Flujo alternativo: *cancelar*. El usuario cancela el proceso de baja.
 - 5.b. El usuario cancela el proceso de baja. El sistema regresa al punto 2 del flujo principal.
- + Post-Condiciones: El usuario ya no figura como dado de alta en la aplicación.

CU09 Logout Proceso por el cual un usuario cierra su sesión en el sistema.

- + Actores implicados: Usuario Registrado.
- + Precondiciones: el usuario está con su sesión abierta en el sistema.
- + Flujo principal:
 1. El usuario hace click en el link de *logout*.
 2. El sistema cierra la sesión del usuario.
 3. El sistema muestra la página de inicio.
- + Post-Condiciones: El usuario ha efectuado el logout con éxito.

3. Diseño

En este apartado se ofrece una descripción de las decisiones tomadas a nivel de diseño a lo largo del desarrollo de la aplicación web.

3.1. Arquitectura del Sistema

Desde un punto de vista lo suficientemente alejado como para no percibir los detalles internos, la aplicación web está organizada siguiendo la arquitectura clásica cliente-servidor-datos. Hoy en día, quizás, esta no sea la organización más vanguardista. Los términos *serverless*, *microservicios* o *single-page-application* son los que con mayor eco resuenan en la industria web actual.

Como muchas tendencias dentro del mundo de la computación, estos términos no están del todo delimitados, ofreciendo en ocasiones diferentes interpretaciones y/o solapamientos entre sí. Serverless hace mención a aquellas aplicaciones web que en su mayor parte, o por completo, incorporan servicios cloud

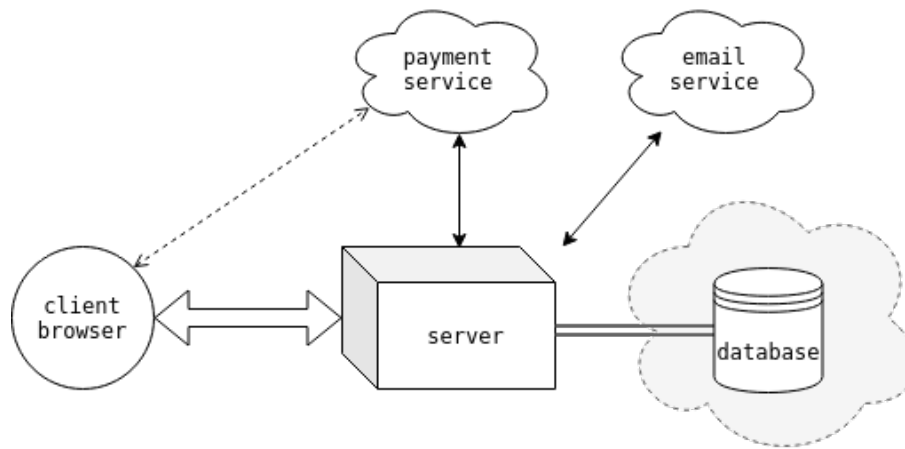


Figura 2: Modelo del sistema

de terceras partes para gestionar su propia lógica de negocio. La arquitectura de microservicios está muy relacionada con lo anterior, ya que esta concibe las aplicaciones como un agregado de servicios muy especializados ejecutados de forma independiente, puede que en máquinas remotas, que se comunican entre sí de manera ligera y eficiente. Por su parte, las SPA usan la elevada capacidad de cómputo de los clientes actuales para trasladar allí parte de la lógica que tradicionalmente implementa el servidor. En una SPA o bien todo el contenido HTML, JavaScript y CSS es cargado una sola vez, o bien se carga dinámicamente bajo demanda, normalmente como respuesta a las acciones del usuario, dando así una mayor sensación de fluidez.

No obstante, a pesar de las muchas ventajas que presentan estas formas de enfocar las aplicaciones web, se ha optado por la arquitectura clásica (con pinceladas, como se verá, de aspectos serverless y SPA), y el motivo es doble:

- Antes del 2 va el 1. El punto de partida en el desarrollo de la presente aplicación web fue el de prácticamente nulos conocimientos sobre esta rama de la informática. En este escenario, antes de estudiar directamente las arquitecturas surgidas en los últimos años, se ha preferido estudiar la arquitectura clásica, en el convencimiento de que es el camino formativo correcto.
- Tecnología viva. En contra de lo que hace unos años algunas voces avanzaban, la arquitectura browser-server-database no está muerta y previsiblemente no lo va a estar. Esta es una tecnología madura, robusta y muy extendida. Aunque es muy cierto que su cuota de mercado ha descendido en pro de otras arquitecturas más recientes, muestra tener un suelo esta-

ble y un lugar propio dentro de las tecnologías web. En parte esto es así porque, como casi siempre, no es oro todo lo que reluce. Las arquitecturas modernas no son la perfección, mostrando tener algunos puntos débiles [bib ref].

En la figura 2 se presenta un esquema de la arquitectura de la aplicación web. Como se aprecia, responde a la mencionada arquitectura clásica cliente-servidor-datos. No obstante, existen ciertos elementos que descansan en servicios cloud externos, dotando a la aplicación de cierto carácter serverless. Así, el servicio de pagos es ofrecido por Stripe (ver 4.2). Como se aprecia en la figura 2, existe una vía de comunicación directa entre el cliente y Stripe. Esto es así para garantizar que la aplicación web no tiene nunca acceso a los datos confidenciales de pago del cliente (p.e., número de la tarjeta de crédito), sino que esta información es enviada directamente desde el cliente a Stripe. Por su parte, el servicio de correo es delegado en Gmail, a través de la capa de abstracción que Spring Framework ofrece de la API JavaMail. Y, por último, la base de datos. Este caso es especial porque dependiendo del entorno de ejecución se puede asimilar a un servicio cloud o no. En este sentido, cuando el entorno de ejecución es producción, la base de datos es accedida como un servicio que AWS presta a Heroku.

Por otro lado, centrando ahora la atención en la organización interior de la aplicación, el principio de diseño protagonista ha sido la **separación de responsabilidades** (*separation of concerns*, SoC). Este principio es un viejo conocido en el mundo de la ingeniería, haciendo manejables problemas que de otra manera serían muy costosos, si no directamente intratables.

3.1.1. Inversión de Control - Inyección de Dependencias

Un primer paso en la división de responsabilidades es la separación de la construcción y el uso. Piénsese por ejemplo en un aeropuerto. Los trabajos que se deben llevar a cabo, las herramientas necesarias, el personal técnico implicado, etc. son distintos en la fase de construcción que durante su posterior explotación. En este sentido, los sistemas software no son excepción. Las tareas a realizar cuando la aplicación arranca son diferentes de las que se llevan a cabo durante el normal funcionamiento posterior. Por tanto, la lógica de arranque, encargada de crear los objetos y de resolver y conectar sus dependencias, debe ser separada de la lógica que comienza tras el arranque.

Un mecanismo efectivo para separar la construcción del uso es la **inyección de dependencias** (*dependency injection*, DI), que, tal como explica Martin Fowler en su [artículo seminal](#) en la materia [bib ref], es un tipo de **inversión de control** (*inversion of control*, IoC) para la gestión de dependencias. Por inversión de control se entiende la técnica de traspasar las responsabilidades

secundarias de un objeto a otros objetos especializados para el propósito. Así, en el contexto de gestión de dependencias, un objeto no debe asumir la responsabilidad de instanciar él mismo sus dependencias. En su lugar, debe ceder esta responsabilidad a otra entidad especializada, invirtiendo así el control. Debido a que la configuración inicial es una cuestión global, esta entidad especializada será la rutina *main* o un contenedor especializado en este propósito. En el caso de la presente aplicación web se ha usado el contenedor de inyección de dependencias del mundo Java líder en la actualidad, Spring Framework (ver 4.1.4). Como se muestra en el listado 1, el contenedor es invocado en la primera y única instrucción de *main*.

```
1  @SpringBootApplication
2  public class FirstmarketApplication {
3
4      public static void main(String[] args) {
5          //el contenedor de IoC se encarga de crear los objetos y
6          //de resolver sus interdependencias
7          SpringApplication.run(FirstmarketApplication.class, args);
8      }
9
10 }
```

Listing 1: Inversión de control de dependencias

A modo de ejemplo, las figuras 3, 4 y 5 muestran, respectivamente, los diagramas de inyección de dependencias para las tres capas de abstracción relacionadas con los libros: *bookRepository*, *bookServer* y *bookController*. La flecha verde, que parte siempre de *firstmarketApplication*, indica que el objeto apuntado ha sido creado por el contenedor de IoC. Las flechas azules significan que el objeto apuntado (la dependencia) ha sido insertado por el contenedor de IoC en el objeto apuntador.

3.1.2. Capas funcionales

Otro paso importante dado en la dirección de separar las responsabilidades es la división funcional del código en tres capas: web, negocio y persistencia. Estas capas están contenidas dentro del paquete *core*, y están inspiradas en el patrón de diseño *Fron Controller*, que es el que ordena, a su vez, la infraestructura web que proporciona Spring Framework (ver sección 4.1.4).

La capa web se encarga de gestionar las peticiones HTTP encauzadas a través del *DispatcherServlet* y de resolver y generar el contenido HTML. A parte de toda la infraestructura que ofrece Spring Framework para esta tarea, se ha desarrollado una serie de *controladores* (anotados con *@Controller*), contenidos en el paquete *core.controller* (ver figura 7), que sirven un conjunto de *vistas*,

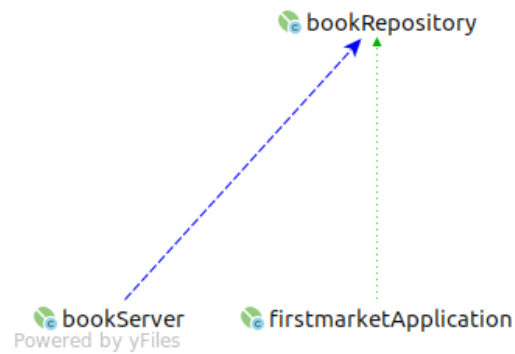


Figura 3: Creación e inyección de dependencias en BookRepository

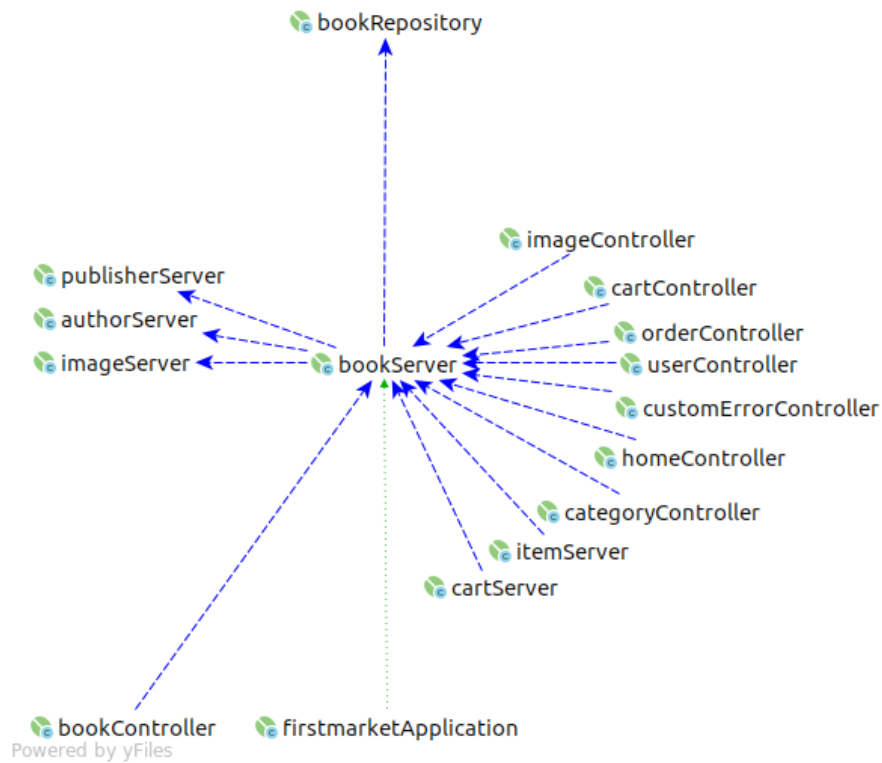


Figura 4: Creación e inyección de dependencias en BookServer

contenidas en *classpath: resources/templates*. Esta capa recibe el servicio de la capa de negocio.

En la capa de negocio reside la lógica (objetos anotados con *@Service*) que modela el funcionamiento de una tienda de libros a través del código contenido

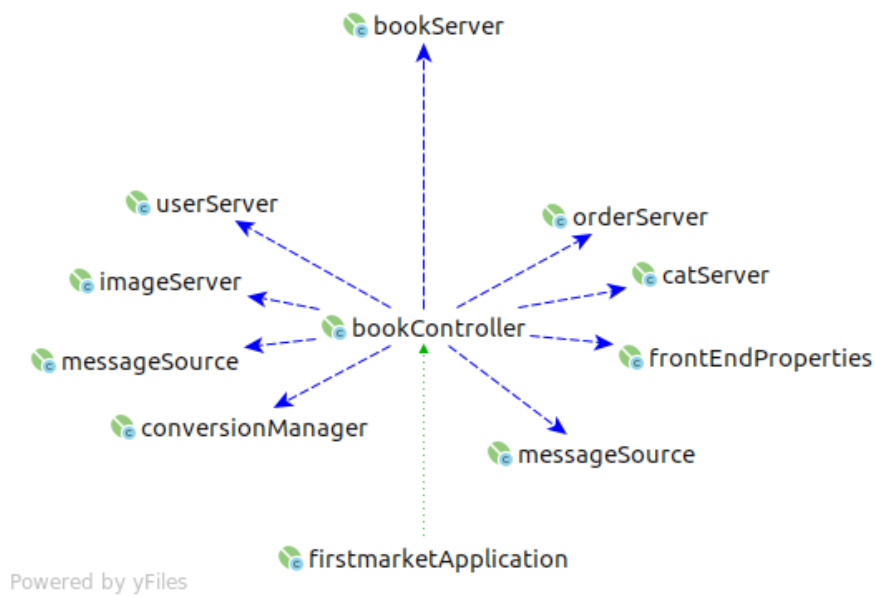


Figura 5: Creación e inyección de dependencias en BookController

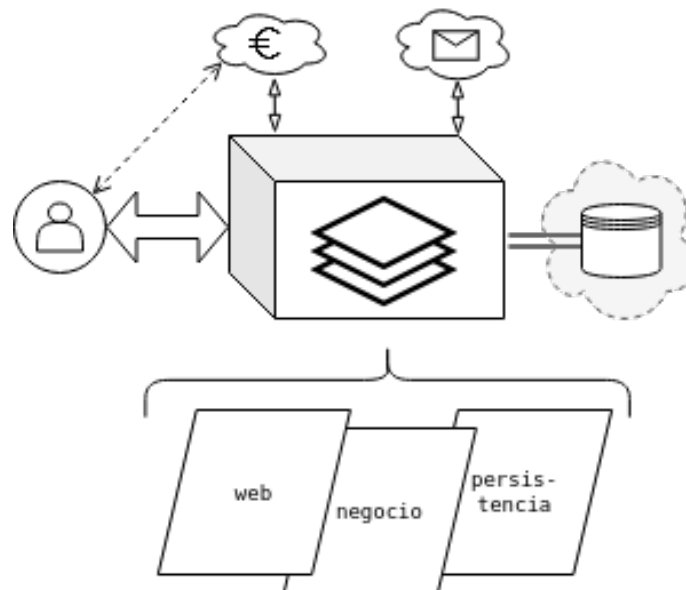


Figura 6: Modelo del sistema en capas funcionales

en el paquete *core.service* (ver figura 9). Esta capa da servicio a la capa web y a sí misma, mientras que se apoya en el servicio que le ofrece la capa de persistencia.

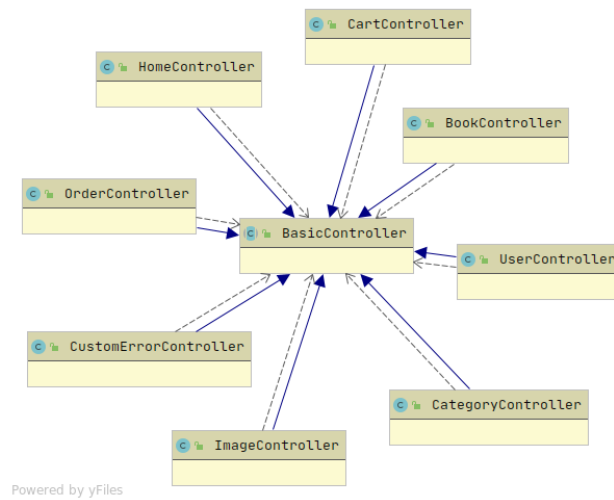


Figura 7: Diagrama del paquete core.controller

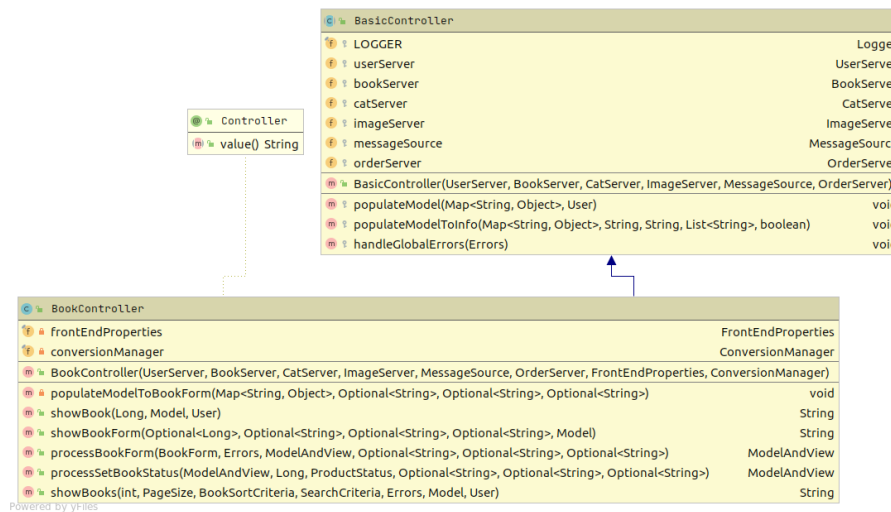


Figura 8: Detalle de la clase BookController

La capa de persistencia sigue el patrón de diseño *Repositorio*. Así, toma el control de todos los aspectos relacionados con la solución concreta de almacenamiento y manipulación de los datos, desacoplando por completo esta funcionalidad de la lógica de negocio, a la cual sirve mediante un conjunto de interfaces (anotadas con *@Repository*) definidas en *core.data* (ver figura 11).

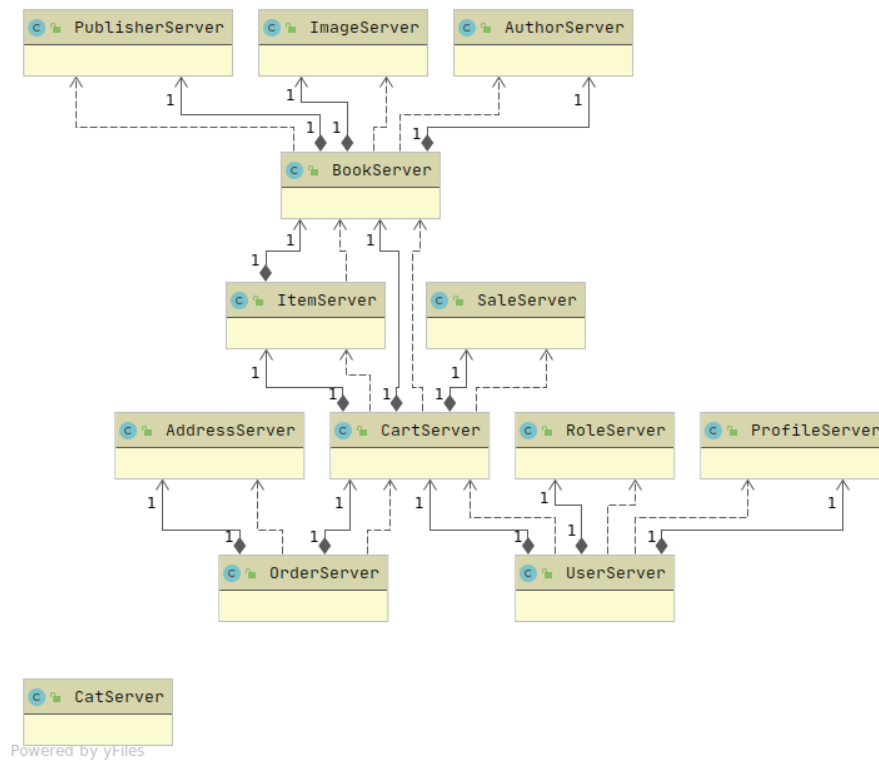


Figura 9: Diagrama del paquete core.service

3.2. Modelo de Datos

Para dar solución al problema presentado (la venta de libros online) se ha realizado un modelado de entidades del mundo real. Estas entidades se encuentran definidas en el paquete *core.model*. En la figura 13 se muestra el diagrama de clases de este paquete. Este modelo encuentra una traducción directa en las entidades gestionadas en la base datos. En las figuras 14 y 15 se ofrecen, respectivamente, los diagramas entidad relación sin y con las propiedades de los objetos visibles.

4. Implementación

La materialización del análisis y diseño detallado en las secciones anteriores lo conforma el conjunto software que se incluye en el CD distribuido con la presente memoria. No obstante, en este apartado se ha querido comentar ciertos aspectos relevantes, que por la dificultad presentada y/o por el papel clave dentro de la aplicación merecen mención especial.

BookServer		
LOGGER	Logger	
bookRepository	BookRepository	
imageServer	ImageServer	
publisherServer	PublisherServer	
authorServer	AuthorServer	
maxNumOfTrendingBooks	int	
maxNumOfNewBooks	int	
persist(Book)	Book	
edit(Book)	Book	
isbnExists(String)	boolean	
findById(Long)	Book	
findTopTrendingBooks()	List<Book>	
findTopNewBooks()	Set<Book>	
getRandomBooks(int)	List<Book>	
updateCategoryIdByCategoryId(Long, Long)	void	
updateImageByImageId(Long, Image)	void	
findTopAuthorViewsByCategoryId(Long, int)	List<AuthorView>	
findTopPublisherViewsByCategoryId(Long, int)	List<PublisherView>	
findTopLanguagesByCategoryId(Long, int)	List<LanguageView>	
findSearchResults(SearchCriteria, Pageable)	Page<Book>	
getIdsByStatus(ProductStatus)	Set<Long>	
getIdsByQueryText(String)	Set<Long>	
getIdsByQueryTextFromIsbn(String)	Set<Long>	
extractIsbnsFromQ(String)	Set<String>	
getIdsByPriceIntervals(Set<PriceInterval>)	Set<Long>	
intersect(Set<Long>...)	Set<Long>	
existsBook(Long)	boolean	
checkAvailabilityFor(Set<Item>)	void	
removeFromStock(Set<Item>)	void	
restoreStock(Set<Item>)	void	
setStatus(Long, ProductStatus)	void	
incrementCartBookRegistry(Long)	void	
decrementCartBookRegistry(Long)	void	
incrementCartBookRegistry(List<Long>)	void	
cartBookRegistry	Map<Long, Integer>	

Service
value() String

Figura 10: Detalle de la clase BookServer

4.1. Stack Tecnológico

En este apartado se ofrece un comentario sobre las principales decisiones tomadas acerca de las tecnologías empleadas en el desarrollo de la aplicación web.

4.1.1. Git

Como se comentó en la exposición de motivos, Git es una tecnología clave para el control de versiones, y desde el principio se puso el foco en ella. Aunque existen otras herramientas muy válidas, como Mercurial, la decisión fue seguir la tendencia global.

Formalmente, Git es un sistema de control de versiones distribuido, permitiendo gestionar los cambios introducidos en un conjunto de archivos, conocido como *repositorio*, por un número arbitrario de personas. Sus principales características son:

- **Basado en instantáneas.** Otros sistemas de control de versiones tratan



Figura 11: Diagrama del paquete core.data

la información que almacenan como un conjunto de archivos más sus cambios a lo largo del tiempo, los incrementos. En cambio, Git piensa en sus datos más como una serie de instantáneas de un sistema de archivos en miniatura. Así, cada vez que se efectúa un *commit* o se guarda el estado de un proyecto, Git toma una fotografía de cómo se están todos los archivos en ese momento y almacena una referencia a esa instantánea. Además, para ser eficiente, si los archivos no han cambiado no almacena el archivo nuevamente, sino un enlace al archivo idéntico anterior almacenado. En resumen, Git trata los datos que gestiona como una secuencia de capturas.

- **Trabajo en local.** Git es un sistema distribuido que permite que todos los implicados en un proyecto tengan en su máquina una copia completamente funcional del repositorio. Así, la mayoría de las operaciones necesitan sólo archivos y recursos locales para funcionar, siendo muy poco lo que no se

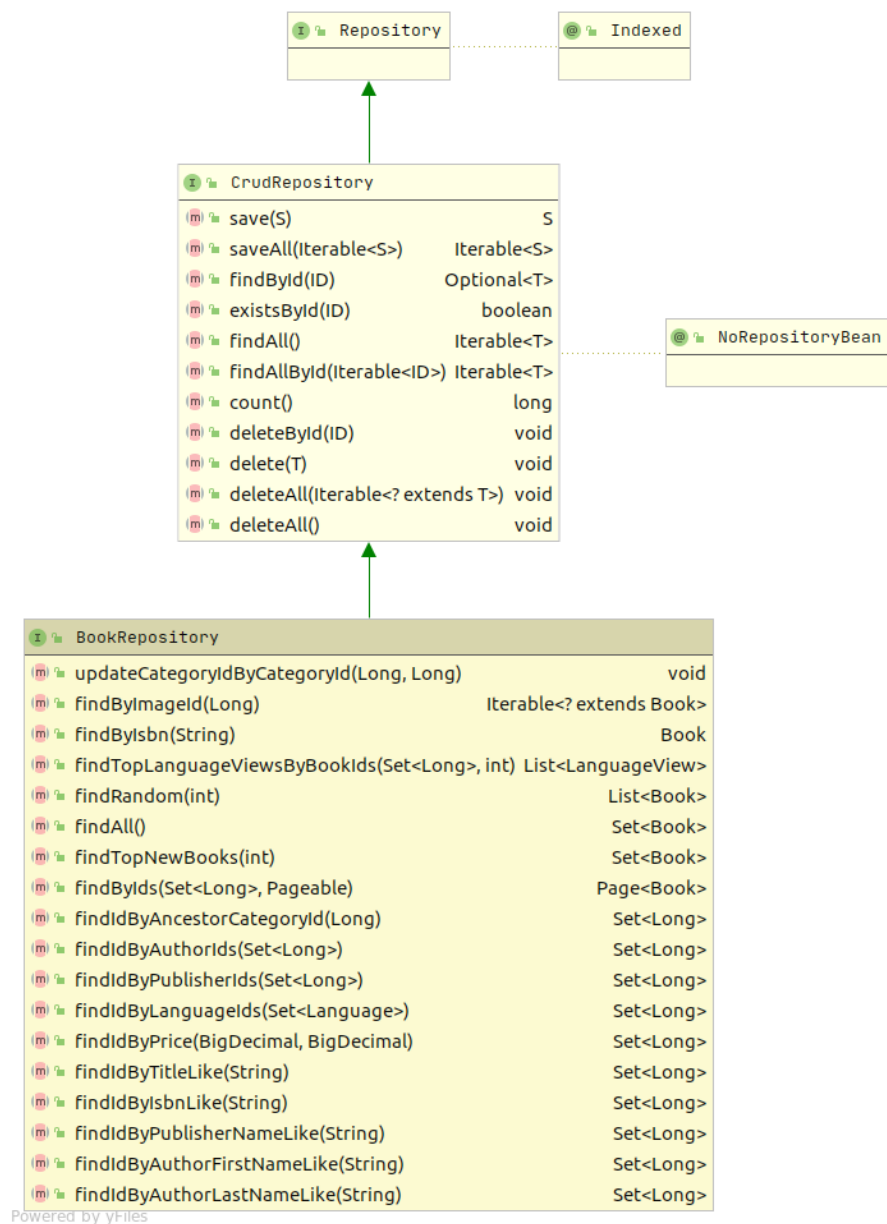


Figura 12: Detalle de la interfaz BookRepository

puede hacer sin conexión a Internet.

- **Integridad de la información.** Es casi imposible que algo varíe en los archivos gestionados por Git sin que el sistema lo detecte, dado que a todos estos archivos se les aplica la función *hash* SHA-1, obteniendo como resultado una cadena de 40 caracteres hexadecimales. Esta cadena es luego

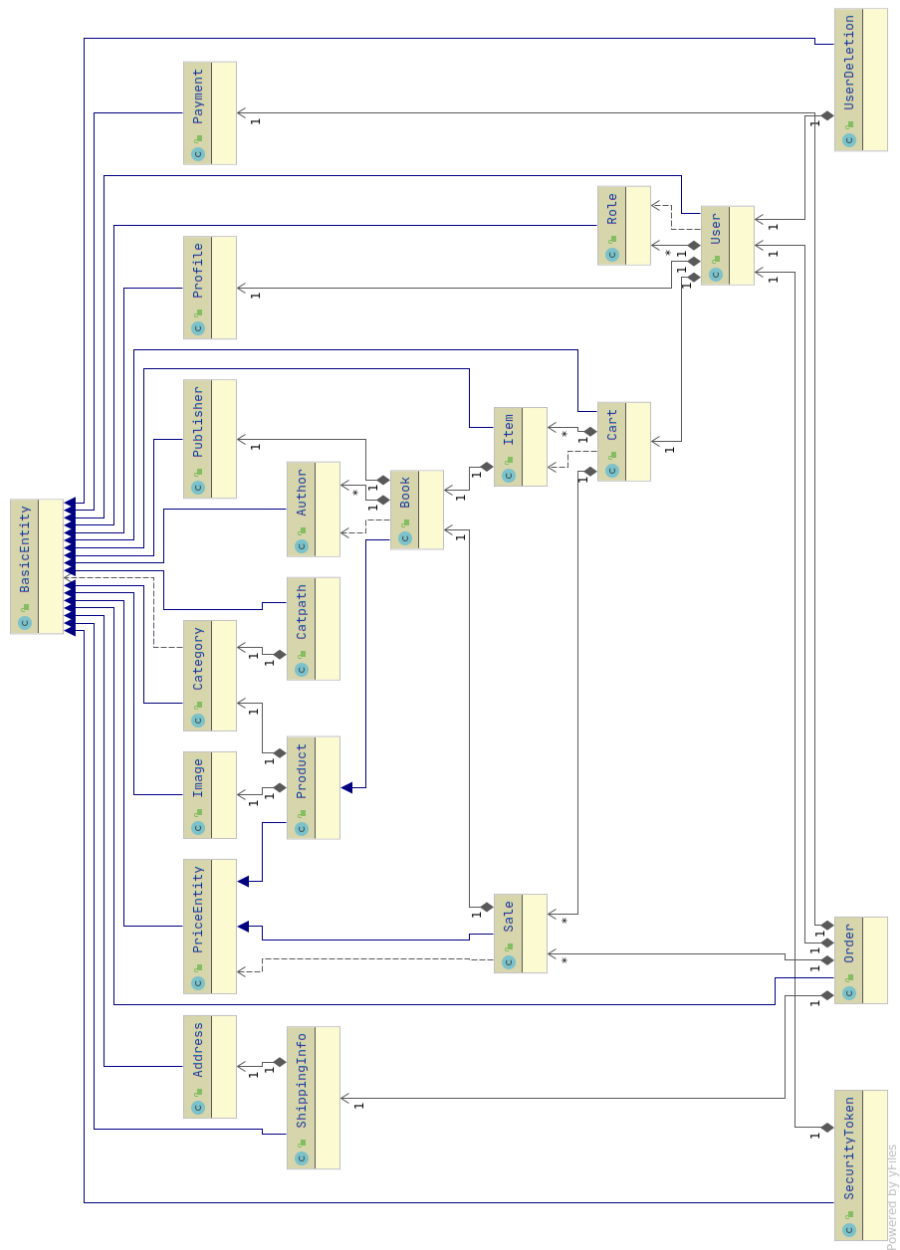


Figura 13: Diagrama de clases del paquete core.model

usada en todo momento para referenciar el contenido usado para generarla.

- **Seguridad ante accidentes.** Con Git es muy difícil que, una vez realizado un *commit*, se pierda el trabajo almacenado.

En el presente proyecto el uso de Git ha sido intensivo y fundamental. El

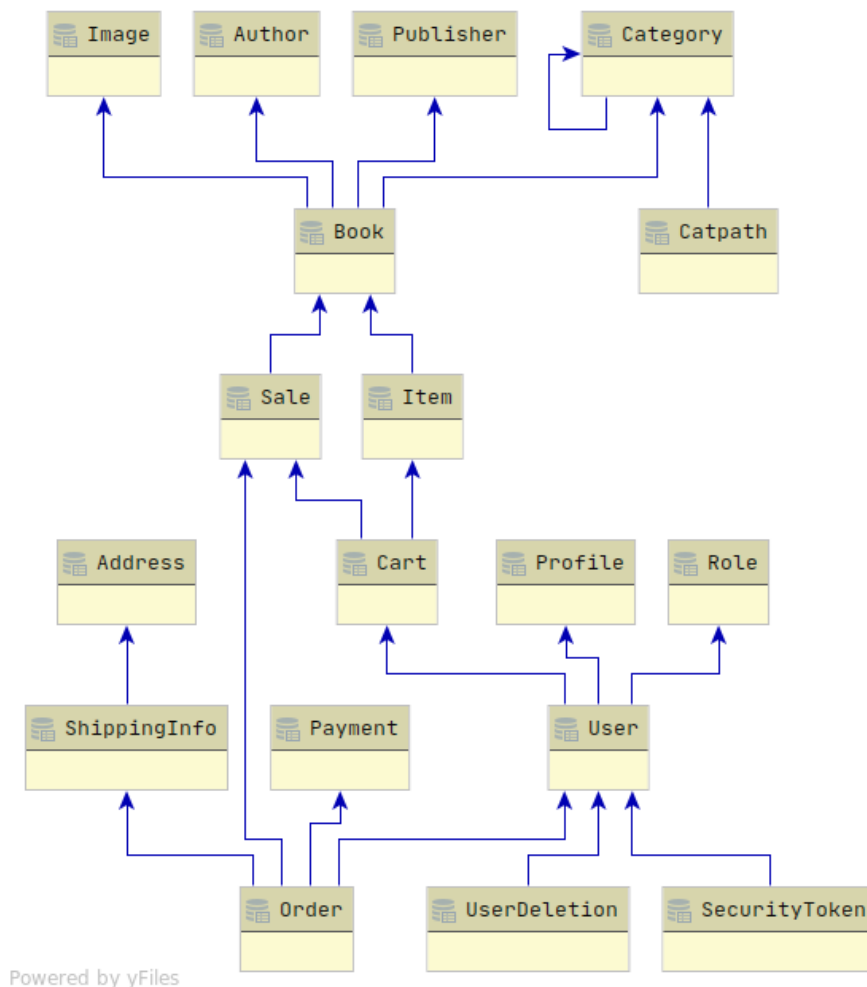


Figura 14: Diagrama entidad-relación

repositorio del proyecto, alojado en GitHub, puede ser accedido [aquí](#).

4.1.2. Java

Una de las primeras decisiones que se tuvo que tomar fue qué lenguaje de programación emplear para construir la aplicación web, ya que aunque en la especificación de la oferta del proyecto se establecía que este debiera ser PHP, a posteriori la libertad fue total. Esta no es una decisión trivial, ya que determina en gran medida el resto del stack tecnológico utilizado. Tras una exploración de las diferentes alternativas, y dado el background adquirido a lo largo del plan de estudios, la opción tomada fue Java. Es cierto que existen alternativas potentes en el mundo de la programación web, pero también lo es que Java ocupa una

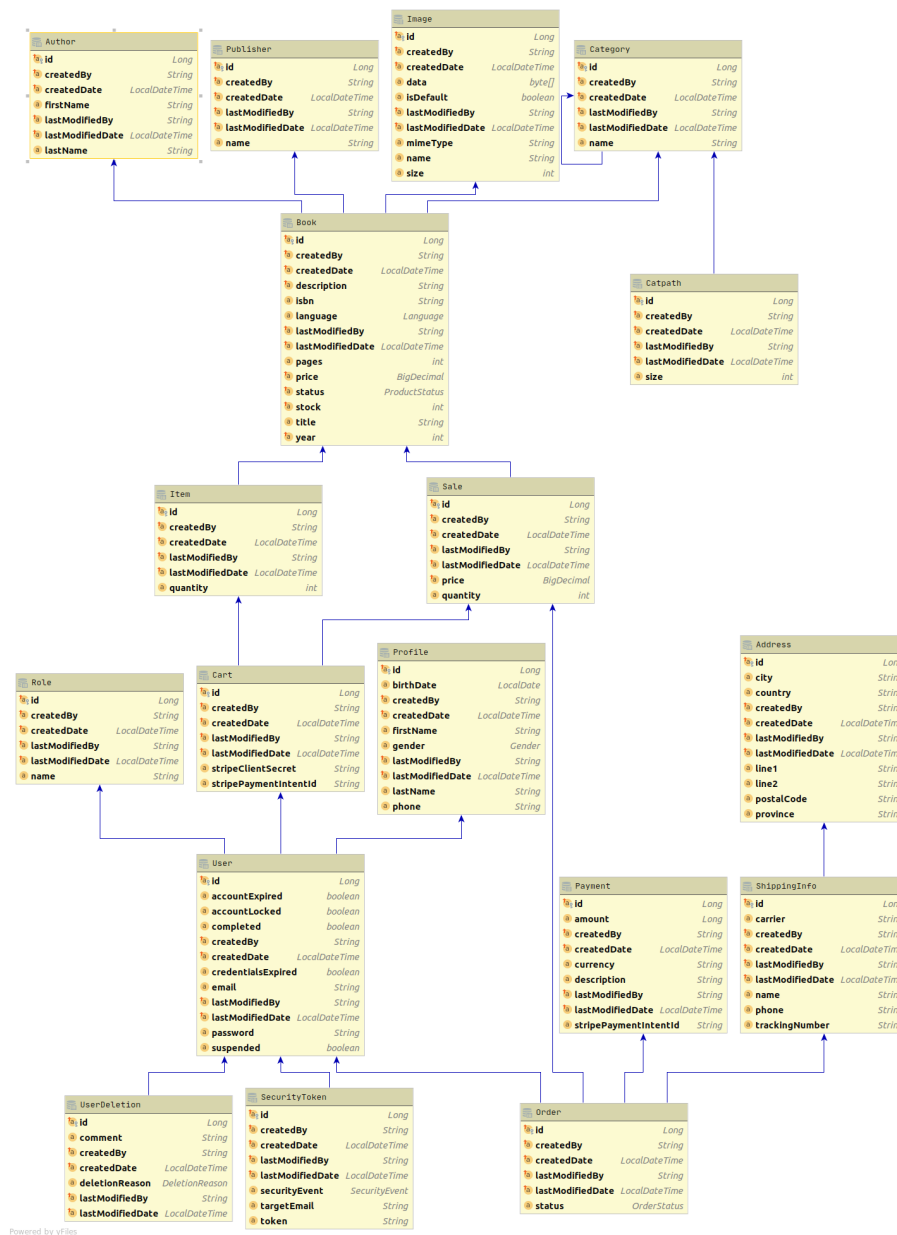


Figura 15: Diagrama entidad-relación expandido

posición muy madura y asentada en la industria, con un entorno de recursos disponibles para el estudiante difícilmente superable, y con una perspectiva de futuro, como mínimo, estable. La decisión tomada primó profundidad antes que anchura, en el sentido de preferir aumentar y depurar las habilidades en Java antes de expandir a otros lenguajes. Por supuesto, esto es muy discutible, pero

se creyó que era la opción correcta.

Resaltar, además, que la versión utilizada ha sido Java 11, que en la actualidad es la más reciente que ofrece soporte a largo plazo (LTS, *Long Term Support*).

Java es un lenguaje que, como muchos otros, ha ido evolucionando con el tiempo, dando como resultado un conjunto de capacidades muy variado. En 2014, con la publicación de Java 8, el lenguaje experimentó una de las mayores expansiones, si no la mayor, de su historia. En ese momento se le introdujeron potentes capacidades de programación declarativa, como las *Lambda Expressions* y la *Stream API*, que, en cierto modo, han revolucionado la programación Java (véase, por ejemplo, el framework de desarrollo de aplicaciones web [Spark](#), que sólo usa capacidades Java 8+)

```
1 private Set<Long> getIdsByExcludedStatus(ProductStatus
   excludedStatus) {
2 return bookRepository
3 .findAll()
4 .stream()
5 .filter(book -> !book.getStatus().equals(excludedStatus))
6 .map(Book::getId)
7 .collect(Collectors.toSet())
8 ;
9 }
```

Listing 2: Programación declarativa con Java

Como se ha comentado, una de las intenciones de usar la tecnología Java era precisamente profundizar en su conocimiento. Así, se ha aprovechado el desarrollo de la presente aplicación para estudiar estas capacidades declarativas, aplicándolas progresivamente. Como ejemplo, en el listado 2 se muestra la implementación del método *getIdsByExcludedStatus*, perteneciente a la clase *BookServer*. Se puede apreciar la facilidad de lectura que el estilo declarativo proporciona, además de la potencia expresiva.

También es necesario comentar un par de librerías que merecen especial mención, por la relevancia que han tenido en el proyecto:

Project Lombok Aparte del nombre de una isla indonesia (como Java), Lombok es una librería Java destinada a aumentar la productividad. Se encarga de suprimir la necesidad de especificar cierto código repetitivo, como los constructores, los *getters* y los *setters*, o los *equals* y los *hashCode*. Así, como se muestra en el listado 3, a través de anotaciones se puede ahorrar una grandísima canti-

dad de código, que es generado por esta herramienta automáticamente antes de la compilación.

```
1 package misrraimsp.uned.pfg.firstmarket.core.model;
2
3 import lombok.Data;
4 import lombok.EqualsAndHashCode;
5
6 import javax.persistence.Entity;
7
8 @Data
9 @EqualsAndHashCode(callSuper = true)
10 @Entity
11 public class Role extends BasicEntity {
12
13     private String name;
14
15 }
```

Listing 3: Clase *Role* usando las anotaciones Lombok

No obstante, en cualquier momento se puede efectuar un *Delombok* y ver el código generado, como se muestra en el listado 4. Como se aprecia, la cantidad de código ahorrado es más que significativa.

```
1 package misrraimsp.uned.pfg.firstmarket.core.model;
2
3 import javax.persistence.Entity;
4
5 @Entity
6 public class Role extends BasicEntity {
7
8     private String name;
9
10     public Role() {
11     }
12
13     public String getName() {
14         return this.name;
15     }
16
17     public void setName(String name) {
18         this.name = name;
19     }
20
21     public String toString() {
22         return "Role(name=" + this.getName() + ")";
23     }
24 }
```

```

25     public boolean equals(final Object o) {
26         if (o == this) return true;
27         if (!(o instanceof Role)) return false;
28         final Role other = (Role) o;
29         if (!other.canEqual((Object) this)) return false;
30         if (!super.equals(o)) return false;
31         final Object this$name = this.getName();
32         final Object other$name = other.getName();
33         if (this$name == null ? other$name != null : !this$name.
            equals(other$name)) return false;
34         return true;
35     }
36
37     protected boolean canEqual(final Object other) {
38         return other instanceof Role;
39     }
40
41     public int hashCode() {
42         final int PRIME = 59;
43         int result = super.hashCode();
44         final Object $name = this.getName();
45         result = result * PRIME + ($name == null ? 43 : $name.
            hashCode());
46         return result;
47     }
48 }

```

Listing 4: Clase *Role* tras efectuar un *Delombok*

SLF4J - Logback SLF4J (Simple Logging Facade for Java) es una API Java para tareas de registro que implementa el patrón de diseño *facade*, el cual permite desacoplar completamente el código de la aplicación de la tecnología específica de registro que subyaga bajo la fachada.

Logback es la tecnología elegida que en el presente proyecto se conecta como backend de SLF4J. Se trata de una librería moderna, diseñada por el fundador de la famosa Log4j (Ceki Gülcü), y que pretende ser su sucesora.

La configuración se encuentra en el fichero *classpath: logback-spring.xml*, donde se detallan todos los aspectos relevantes, como el formato utilizado en los registros o la política de escritura y rotación de los ficheros de registro producidos.

Esta es una de las funcionalidades añadidas al proyecto que, aún siendo muy humilde, ha tenido enorme impacto en la facilidad de desarrollo y mantenimiento. Es clave que desde fases tempranas se tenga un sistema de registro, el cual poder utilizar, por ejemplo, en las múltiples tareas de depuración que se deben llevar a cabo casi constantemente.

Por último, un apunte acerca del entorno de desarrollo utilizado. Durante las primeras semanas se usó el mismo que en toda la carrera: Eclipse. Sin embargo, en el transcurso de una formación en Spring tomada en fases iniciales del proyecto, se entró en contacto con **IntelliJ IDEA**. Desde entonces ese fue el IDE empleado, destacándose su capacidad de aumentar la productividad y su facilidad de uso.

4.1.3. Apache Maven

En el mundo del desarrollo software una fase de especial importancia es aquella encargada de transformar todos los archivos de código fuente del proyecto en un paquete susceptible de ser distribuido, instalado y ejecutado en cualquier máquina. A esto se le conoce como el proceso de construcción del proyecto, y representa una tarea no trivial desde el momento en que el proyecto en sí adquiere cierta complejidad.

Antiguamente, este proceso tenía que ser diseñado e implementado *ad hoc*, de una manera específica para cada proyecto. Sin embargo, en la actualidad este proceso se encuentra altamente optimizado. En el caso del mundo Java las principales herramientas empleadas al efecto son Apache Maven y Gradle.

En el presente proyecto se ha optado por utilizar Apache Maven. La justificación reside en que el punto de partida fue el total desconocimiento de este tipo de tecnologías, y al representar Gradle, aparentemente, una suerte de superconjunto o mejora sobre Apache Maven, se juzgó conveniente tomar a este último como puerta de entrada o primer escalón, teniendo en cuenta, además, que Apache Maven aún posee una profunda implantación en la industria y la documentación y recursos de todo tipo para su estudio es ingente.

Apache Maven supuso una revolución en la forma de construir software en el mundo Java. Pero esta herramienta va más allá, siendo concebida como una utilidad de gestión y de comprensión de los proyectos software. Tal como se explica en su [web oficial](#), sus principales objetivos son:

- Facilitar la construcción software.
- Homogeneizar el proceso de construcción software.
- Ofrecer un lenguaje común a la hora de obtener información sobre un proyecto, es decir, establecer una manera homogénea de *definir* a un proyecto software.
- Incentivar buenas prácticas de desarrollo software.

Las características que han permitido a Apache Maven alcanzar estos objetivos descansan en gran medida en:

- **Convención sobre configuración.** Como también se comentará en la sección 4.1.4, este principio de diseño persigue que el desarrollador sólo deba especificar los aspectos no convencionales, confiando para el resto en la configuración por defecto ofrecida por Apache Maven. Como dichos valores por defecto normalmente son reflejo de buenas prácticas, más o menos consensuados por agentes expertos en la materia, automáticamente se está consiguiendo alinear el proyecto a dichas buenas prácticas, al tiempo de estarse facilitando el propio trabajo de construcción software. La adopción de Apache Maven de este paradigma se traduce en, entre otras muchas cosas, establecer una distribución de carpetas y ficheros estándar para el proyecto, o seguir determinadas convenciones a la hora de compilar el código fuente o ensamblar los paquetes.
- **Interfaz común.** Muy ligado con el aspecto anterior está el hecho de que Apache Maven ofrece un marco de trabajo o interfaz común para la construcción de proyectos software. Como ya se comentó, antes de la existencia de este tipo de herramientas el software se construía de una manera singular en cada proyecto. Con Apache Maven, sin embargo, se establece una manera estándar o canónica de hacerlo. Así, cuando un desarrollador aprecia que un determinado proyecto está gestionado con Apache Maven puede asumir mucha información sobre él.
- **Reutilización a través de *plugins*.** Apache Maven está diseñado de forma que se maximiza la flexibilidad y la reutilización de la lógica de construcción. El núcleo de la herramienta se encarga de muy pocas tareas, actuando más como un orquestador que lee la información centralizada acerca del proyecto en un fichero XML y delega el grueso del trabajo en plugins. Es decir, son los plugins los que realizan las diferentes labores de construcción, como compilar, instalar o empaquetar. Apache Maven ofrece una serie de plugins por defecto, pero también la posibilidad de construir nuevos e implementar la lógica de construcción que se desee. De hecho, en el presente proyecto se ha añadido el plugin que ofrece Spring Boot.
- **Modelo conceptual de proyecto.** Dicho lo anterior, la gran característica que aporta Apache Maven es la abstracción de toda la información necesaria referente a un proyecto en un fichero, el **Project Object Model** (en el presente proyecto ver el fichero *pom.xml*). En él se especifica, entre otras cosas, las dependencias con otros proyectos, de forma que es Apache Maven quien se encarga de gestionarlas, esto es, resolverlas, descargarlas

de un repositorio central, colocarlas en las carpetas apropiadas, enlazarlas durante la construcción, etc. En el listado 5 se muestra, como ejemplo, la declaración de una dependencia con la librería [Passay](#), utilizada en la aplicación web para generar nuevas contraseñas aleatorias.

```
1 <dependency>
2 <groupId>org.passay</groupId>
3 <artifactId>passay</artifactId>
4 <version>1.5.0</version>
5 </dependency>
```

Listing 5: Declaración de dependencia en Apache Maven

El desarrollo del presente proyecto ha servido para tomar un primer contacto serio con Apache Maven, resultando de vital importancia para la construcción y para la gestión de dependencias.

4.1.4. Spring

Sin duda, Spring ha sido una de las tecnologías que más impacto ha tenido en el presente proyecto. En esta sección se pretende dar una idea lo más general y amplia posible de sus características.

Lo primero a aclarar es qué se entiende por *Spring*, ya que su significado puede variar dependiendo del contexto. Puede que se refiera en concreto al proyecto *Spring Framework*, donde empezó todo allá por el año 2003, o al ecosistema completo formado por todos los [proyectos Spring](#), esto es, por Spring Framework más todos los otros proyectos Spring que se desarrollaron posteriormente utilizando como núcleo al primero.

Los proyectos Spring usados en el desarrollo de la aplicación web han sido *Spring Framework*, *Spring Boot*, *Spring Data JPA* y *Spring Security*. En la presente sección se dará una descripción de las principales características de cada uno de ellos que han tenido relación con el desarrollo del presente proyecto, salvo aquellas referentes a Spring Security, tratadas por separado en la sección [6.2](#) en el marco de la seguridad de la aplicación web.

Spring Framework En este apartado se esbozará las principales características de esta parte fundamental dentro del ecosistema Spring, tomando para ello como principal fuente su propia [documentación](#).

Tal como establecen sus autores, los principios de diseño que han guiado su desarrollo a lo largo del tiempo han sido:

- Estar orientado a permitir que se tomen las decisiones de diseño lo más tarde posible. Por ejemplo, permite cambiar la tecnología de persistencia

sin alterar el código de la aplicación (esto fue constatado en el desarrollo de la aplicación web, al pasar de forma transparente, en una etapa bastante avanzada del proyecto, de MariaDB a PostgreSQL).

- Ser capaz de acomodar diferentes perspectivas de desarrollo. Spring es extensamente configurable, no imponiendo ningún estilo o manera de resolver los problemas.
- Ser altamente retrocompatible.
- Tomar muchas precauciones a la hora de diseñar las APIs, de forma que sean intuitivas y perduren a través de las diferentes versiones.
- Seguir los más altos estándares de calidad del código.

De todas las funcionalidades integradas dentro del proyecto Spring Framework, la más importante es el contenedor de Inversión de Control (inversion of control, IoC). Este concepto, tratado en la sección 3.1.1, hace referencia a la capacidad de gestionar el ciclo de vida de los *beans*, que es como en el ecosistema Spring se conoce a los objetos gestionados por el contenedor.

La interfaz *org.springframework.context.ApplicationContext*, que representa el contenedor de IoC de Spring, es la responsable de instanciar, configurar y ensamblar los *beans*.

Por su parte, el contenedor conoce qué objetos instanciar y cómo configurarlos y ensamblarlos entre sí a través de la lectura de metadatos de configuración especificados por el desarrollador.

La figura 16, tomada de la documentación oficial, muestra un diagrama de alto nivel de cómo funciona lo explicado. Esto es, el código desarrollado en forma de POJOs (Plain Old Java Object) es gestionado en el contenedor de IoC según se haya especificado en los metadatos de configuración, produciendo como resultado una aplicación completamente funcional.

Existen diversas formas de especificar los metadatos de configuración. La manera original fue a través de un fichero XML, pero en la actualidad también se puede realizar con anotaciones insertadas en el código de la aplicación (añadido en la versión 2.5), o con código Java dedicado de configuración (añadido en la versión 3.0). El contenedor de IoC está completamente desacoplado de estos mecanismos de especificación de los metadatos de configuración.

Existe la cuestión de qué tipo de configuración es mejor. La respuesta no es clara, ya que cada opción posee su dosis de ventajas e inconvenientes. Las anotaciones son la manera más concisa y quizás mas simple para un principiante, pero mezclan código de negocio con configuración del framework, algo que mina el principio de separación de responsabilidades. Además poseen el inconveniente

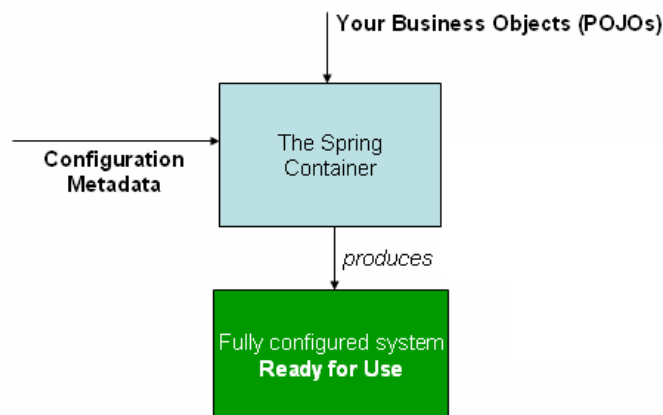


Figura 16: Flujo de información en el contenedor de IoC

de descentralizar la configuración, impactando negativamente en su mantenibilidad. La configuración con código Java permite un control centralizado, pero se pierde en facilidad y brevedad. Por su parte, los ficheros XML son inmejorables a la hora de separar responsabilidades, pudiéndose alterar la configuración sin tocar una línea de código y, por tanto, sin tener que recompilar. El inconveniente de este sistema es el elevado tamaño y complejidad que muchas veces estos ficheros alcanzan, siendo además su curva de aprendizaje más pronunciada.

En el desarrollo de la presente aplicación web se ha utilizado la configuración a través de anotaciones y de código Java dedicado. Además, se ha utilizado Spring Boot, lo que, como se verá, implica que una gran parte (la mayoría) de la configuración necesaria ha sido establecida por defecto.

Además del contenedor de IoC, la aplicación web desarrollada hace uso de otras capacidades ofrecidas por Spring Framework:

- Publicación y escucha de eventos.
- Gestión de transacciones.
- Planificación de tareas.
- Validación.
- El framework de desarrollo web *Spring Web MVC* (comúnmente conocido como simplemente Spring MVC).

Spring MVC, como muchos otros frameworks de desarrollo web, está implementado alineado con el patrón de diseño *Front Controller*, donde un Servlet central, el llamado *DispatcherServlet*, recibe las peticiones HTTP y delega su

procesamiento a los componentes apropiados, en el caso de la presente aplicación los objetos anotados con *@Controller*. Cada uno de estos objetos es responsable de llevar a cabo las acciones necesarias para cada *endpoint* expuesto, apoyándose para ello en la capa de negocio, es decir, en los objetos anotados con *@Service*. Como ya se explicó en la sección 3.1.2, estos objetos *@Service* se apoyan, a su vez, en otros objetos de su clase y en objetos anotados con *@Repository*, que actúan como puerta de entrada a los datos. Cumplida la lógica de negocio, el objeto *@Controller* implicado devuelve el nombre de la vista a proporcionar como respuesta, así como los datos con los que poblarla. El motor de plantillas genera en este punto el contenido HTML, que es devuelto por el *DispatcherServlet* al usuario.

En resumen, Spring MVC ofrece la infraestructura *Front Controller*, esto es, los objetos necesarios para resolver el mapeo de las peticiones HTTP a los métodos apropiados de los objetos *@Controller*, los objetos encargados de resolver las vistas, los encargados del enlace de datos, etc, permitiendo al desarrollador centrarse en especificar la lógica de negocio.

Spring Boot Este proyecto ha sido el gran avance de los últimos años dentro del universo Spring. De hecho, ha cambiado por completo el paradigma, permitiendo el desarrollo de aplicaciones totalmente funcionales en un espacio de tiempo dramáticamente inferior al necesario con anterioridad a su aparición. Tal como viene recogido en la [documentación](#) oficial, sus principales objetivos son:

- Ofrecer una vía de entrada al uso de las tecnologías Spring radicalmente más rápida y sencilla.
- Proporcionar una configuración por defecto sensata, y al mismo tiempo de fácil modificación si los requisitos así lo necesitan.
- Ofrecer todo un abanico de funcionalidades transversales a la mayoría de aplicaciones, como los servidores embebidos.
- Supresión de la necesidad de configuración a través de ficheros XML.

Así, Spring Boot no es una alternativa a otros proyectos Spring. Su objetivo no es proporcionar nuevas soluciones para problemas ya resueltos, sino ofrecer una manera de mejorar el aprovechamiento del ecosistema entero, fomentando una experiencia de desarrollo que simplifique el uso de los módulos ya disponibles. Esto hace que Spring Boot sea una opción ideal para toda clase de desarrolladores, tanto los que ya están familiarizados con el ecosistema Spring como aquellos recién llegados, al permitirles adoptar las tecnologías de Spring de una manera simplificada. En este sentido, Spring Boot ha supuesto un vector

de expansión de las tecnologías Spring, disminuyendo su barrera de entrada y maximizando el aprovechamiento de sus variadas funcionalidades.

En gran medida, lo expuesto se consigue gracias a que Spring Boot ejercita el paradigma de diseño conocido como Convención sobre Configuración (*Convention over Configuration, CoC*), o también como Código por Convención (*Coding by Convention*). Este principio de diseño trata de disminuir el número de decisiones que el desarrollador debe tomar y aliviar así la complejidad de tener que configurar todas y cada una de las áreas que conforman una aplicación, todo ello sin necesariamente merma alguna en la flexibilidad. Así, el desarrollador sólo está llamado a especificar los aspectos no convencionales de la configuración de la aplicación, obteniendo como resultado inmediato un aumento en la productividad considerable.

Conviene destacar también que Spring Boot permite ejecutar aplicaciones web sin necesidad de hacerlo en un contenedor de servlets externo o un servidor de aplicaciones, ya que el propio Spring Boot incluye un Tomcat embebido, el cual se despliega automáticamente en tiempo de ejecución.

Spring Data JPA - Hibernate ORM El proyecto Spring Data JPA, parte de la amplia familia Spring Data, es una capa de abstracción extra sobre la implementación de la especificación JPA que se utilice, que en el caso del presente proyecto ha sido Hibernate ORM.

Por persistencia se hace referencia a la funcionalidad que permite que determinados objetos creados por la aplicación puedan sobrevivir más allá de la frontera de la misma. En términos Java, se trata de que el estado de ciertos objetos pueda perdurar fuera del entorno de la JVM, de modo que dicho estado esté disponible en cualquier momento posterior.

JPA es una especificación que define una API de mapeo objeto-relacional y gestión de objetos persistentes. Además, hace innecesario el tratamiento explícito de las conexiones/recursos usuales al trabajar directamente con JDBC. Actualmente se encuentra en su versión 2.2, siendo su implementación de referencia EclipseLink. No obstante, Spring usa por defecto Hibernate ORM como proveedor de JPA, y en el presente proyecto no se ha modificado dicha configuración preestablecida.

El mapeo objeto-relacional es la funcionalidad que permite la traducción entre los objetos de la aplicación (llamados entidades en este contexto) y las tablas de la base datos. Esto implica gestionar, entre otras cosas, las relaciones que existen entre las entidades. Esta es una tarea no trivial que los proyectos con necesidades de persistencia relacional (en contraposición con los modelos no relacionales, no tratados en el presente proyecto) deben afrontar. En este sentido, Hibernate ORM (Object/Relational Mapping) es un framework que permite la

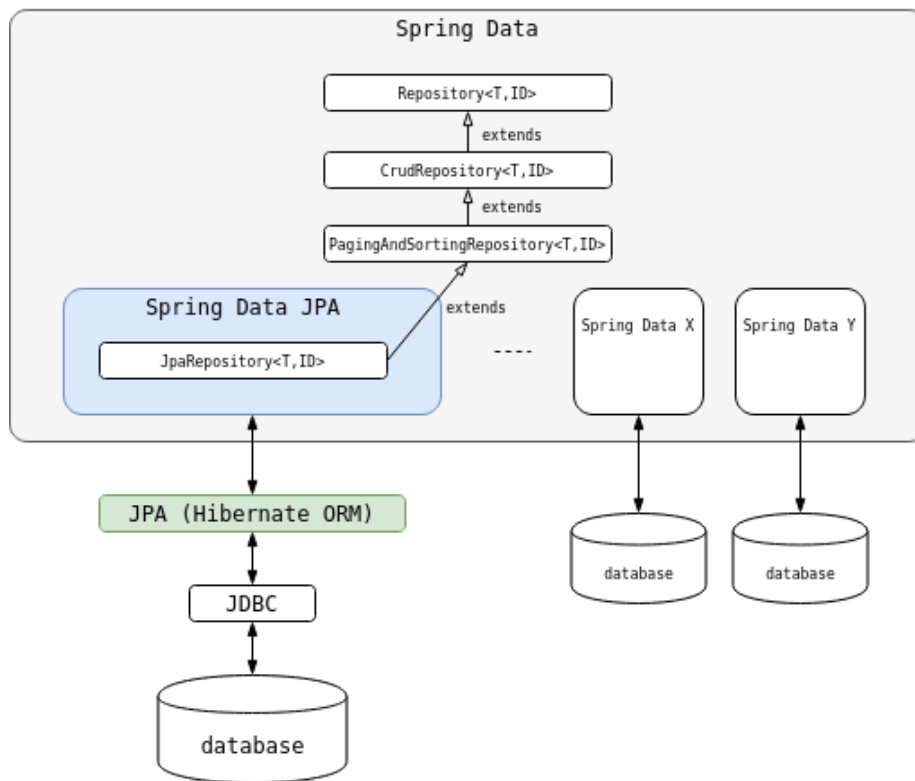


Figura 17: Esquema de la infraestructura de acceso a datos

simplificación y en parte automatización de esta tarea. Como ejemplo, y sin entrar en detalles, las relaciones entre las entidades son especificadas a través de las anotaciones *@OneToOne*, *@OneToMany*, *@ManyToOne* y *@ManyToMany*.

Resaltar que además del mapeo objeto-relacional, la abstracción JPA ofrece (de la mano de Hibernate ORM, en este caso) la gestión del ciclo de vida de las entidades y el lenguaje de consultas relacionales orientado a objeto JPQL. En el listado 6 se muestra un ejemplo de consulta con este lenguaje extraído del presente proyecto.

```

1  @Query("SELECT b.id FROM Book b WHERE b.publisher.id IN :
    publisherIds")
2  Set<Long> findByIdByPublisherIds(@Param("publisherIds") Set<Long>
    publisherIds);

```

Listing 6: Ejemplo de consulta especificada en JPQL

En este punto, vistas las capacidades de abstracción que ofrece JPA a través de sus implementaciones, cabe preguntarse porqué añadir otra capa de abstracción encima con Spring Data JPA. La respuesta es que esta tecnología añade aún más facilidades de desarrollo y mantenimiento, en concreto:

- Permite la integración de JPA con el ecosistema Spring de una forma natural.
- Permite la adopción del patrón de diseño *Repositorio* (ver sección 3.1.2) sin necesidad de añadir código extra, ya que Spring Data JPA, como subconjunto de Spring Data, proporciona una serie de interfaces al efecto, tal como se esquematiza en la figura 17. Así, construir una interfaz repositorio para cualquier entidad de la aplicación es tan fácil como extender alguna de las proporcionadas. En el presente proyecto todos los repositorios extienden a *CrudRepository*<*T*,*ID*>.
- Además de ofrecer las interfaces, también lo hace con sus implementaciones. Es decir, el desarrollador puede olvidarse de crear código que implemente las interfaces de acceso a datos, Spring Data JPA lo realiza automáticamente.
- Posibilidad de inferir las consultas de acceso a datos a partir del nombre del método en la interfaz. Esta funcionalidad ha sido usada extensamente en el desarrollo del presente proyecto. En el listado 7 se muestra un método de la interfaz *SecurityTokenRepository*, a partir del cual Spring Data JPA construye la consulta necesaria para obtener los *SecurityToken* que cumplan estar asociados a un usuario determinado, con un *SecurityEvent* dado y creados después de un punto temporal dado.

```

1  Set<SecurityToken>
    findByTargetUserAndSecurityEventAndCreatedDateAfter (User
        targetUser, SecurityEvent securityEvent, LocalDateTime
        dateTime);

```

Listing 7: Ejemplo de método a partir del cual Spring Data JPA infiere la consulta apropiada

En resumen, Spring Data JPA añade una capa de abstracción sobre JPA, lo que significa que hace uso de todas las funcionalidades que esta especificación proporciona, y además suministra sus propias facilidades, como la implementación sin código del patrón *Repositorio* o la creación de consultas a la base datos a partir del nombre de los métodos.

4.1.5. Thymeleaf

Thymeleaf es un motor de plantillas para aplicaciones Java, tanto web como de escritorio, que permite crear dinámicamente documentos HTML5 (y otros, como XML o XHTML). Como principales características de esta tecnología cabe resaltar:

- Thymeleaf ejerce el concepto llamado *natural templating*, esto es, las propias plantillas son documentos HTML5 válidos, de forma que prácticamente cualquier navegador las puede renderizar, mostrando su contenido por defecto. Esto es muy útil en la fase de desarrollo de la aplicación, ya que no hay necesidad de contar con un contenedor web para visualizar el resultado de la plantilla (como ocurre con la tecnología JSP), facilitando el trabajo paralelo sobre una misma vista en las labores frontend y backend.
- Es altamente extensible. Cualquiera puede crear su propio conjunto de atributos o etiquetas, con los nombres que se desee, definir sintaxis para expresiones y su lógica de evaluación, etc. Esto es, es posible crear lo que se denominan *dialectos* de Thymeleaf, actuando así esta tecnología como un framework de plantillas. Por defecto se usa el conocido como dialecto *Standard*.
- Thymeleaf se integra con muchísima facilidad con la tecnología Spring. De hecho, desde Spring se recomienda preferentemente su uso. Así, se dispone de un dialecto propio: el *SpringStandard*, el cual permite el uso de Spring Expression Language. La integración con Spring Security es muy sencilla, permitiendo acoplar lógica de autenticación y/o autorización en las plantillas de una manera muy poco intrusiva.

En el presente proyecto, los documentos incorporados en *classpath: resources/templates* son las plantillas HTML que el motor toma, junto con los datos de negocio que Spring MVC le suministra, para producir el contenido HTML5 que se sirve al cliente.

4.1.6. Bootstrap

En el ámbito del diseño frontend de la presente aplicación, la herramienta que más ha destacado ha sido Bootstrap, versión 4. Se trata de un framework de HTML, CSS y JavaScript, que prima el desarrollo *responsive*. Esto quiere decir que la aplicación web adapta su presentación al tamaño de la pantalla disponible, esto es, que *responde* automáticamente a los cambios de tamaño de pantalla, de forma que en cada caso optimiza la presentación.

Gracias a Bootstrap, con relativamente poco esfuerzo se puede ofrecer una experiencia de usuario muy correcta. Un gran número de elementos HTML como formularios, botones, encabezados, etc. son proporcionados por este framework, con estilo CSS ya preconfigurado, además de otros elementos como *spinners*, *modals*, *cards* etc., lo que aumenta enormemente la productividad. Además, aparte de los objetos, Bootstrap permite controlar el layout de las vistas de una manera muy sencilla y potente, basada en los Flexbox de CSS.

Cuando el framework no es capaz de ofrecer una funcionalidad siempre se pueden usar directamente hojas de estilo CSS3, como ha sido el caso en el presente proyecto en varias ocasiones (*classpath: resources/static/css/fmstyle.css*). Además, se han utilizado otros componentes que merecen especial mención:

- [FontAwesome](#). Todos los iconos mostrados en la aplicación web han sido obtenidos de FontAwesome.
- [Pretty Checkbox](#). Con el objetivo de dar un mejor aspecto a los *checkboxes* utilizados, respecto del ofrecido por Bootstrap, se decidió usar esta librería CSS.
- [Google Fonts](#). A efectos de dar un cariz más personalizado a la aplicación web se usó este servicio de Google. La fuente usada en la aplicación web ha sido [Noto Serif JP](#).

4.2. Gestión de Pagos con Stripe

Los comercios, ya sean presenciales u *online*, son en esencia lugares de intercambio. Normalmente, lo que una de las partes ofrece a la otra es dinero, a través de un pago. Por tanto, cabe esperar que, en cualquiera de estos establecimientos, el procedimiento por el cual se materializa el pago tome un lugar especial dentro de la lógica de negocio.

Los comercios en Internet tienen, además, la responsabilidad de ofrecer altos estándares de seguridad a sus clientes, especialmente en lo concerniente con el pago. De hecho, una de las principales barreras que aparta a potenciales usuarios del comercio *online* es la percepción de inseguridad que el proceso de pago telemático lleva asociado. En este sentido, el comercio a través de Internet depende de una manera radical en la confianza de los usuarios, siendo así este uno de los principales capitales a cultivar y fortalecer.

La percepción de inseguridad no es en absoluto infundada. El notable desarrollo de la sociedad de la información a lo largo de las últimas décadas lleva inevitablemente asociado el incremento de las modalidades y de la capacidad de fraude digital. No obstante, en una pugna continua, la tecnología anti-fraude trata de mantenerse en todo momento actualizada para ejercer de contrapeso.

Así, como se ha comentado, el pago es una tarea de vital importancia, transversal a todas las plataformas de comercio, con unos estrictos requisitos de seguridad a fin de generar la confianza que demanda el mercado. Es en este contexto donde juegan un papel esencial los proveedores de servicios de pago *online*.

Actualmente en el mercado existen múltiples alternativas de servicios de este tipo, como [PayPal](#) o [SecurionPay](#), pero el elegido para el desarrollo del presente

proyecto ha sido [Stripe](#). Amén de su consolidada posición en el mercado como uno de los servicios líderes en el sector, la principal característica diferenciadora ha sido la [cuidada colección](#) de documentación, guías, ejemplos, etc. que mantienen para facilitar la integración con su infraestructura. En el resto de esta sección se comentará los detalles de dicha integración con Stripe, mientras que en la sección [6.4](#) se hará lo propio con la funcionalidad anti-fraude que esta plataforma de pagos *online* ofrece.

Fundamentalmente, la aplicación web desarrollada se integra con la infraestructura de Stripe haciendo uso de dos puntos de enlace, uno en backend y otro en frontend:

- Una extensa interfaz de programación backend, [Stripe API](#), para varios lenguajes (entre ellos Java), plena de funcionalidades que cubren multitud de casos de uso relacionados con la gestión de transacciones monetarias y sus aspectos asociados.

La integración de la presente aplicación web se ha basado en el objeto [PaymentIntent](#). Un `PaymentIntent` mantiene la información relacionada con un pago puntual. Entre dicha información, una pieza clave es el *client_secret*, que es el código identificativo de la transacción que permite al cliente, usándolo como parámetro en la invocación de funciones en *Stripe.js*, finalizar el pago. Además, el `PaymentIntent` mantiene la información sobre la situación del pago que gestiona al recorrer una serie de [estados](#) desde su creación (su ciclo de vida). Como se verá, FirstMarket conoce de estos estados al escuchar las notificaciones que Stripe le envía al *webhook* configurado.

- Una librería frontend, [Stripe.js](#), que permite, entre otras cosas, gestionar la comunicación entre el cliente y Stripe, y la inserción de elementos gráficos configurables en las vistas que la aplicación web sirve al cliente. Estos [elementos](#) son empleados para recabar la información sensible de los formularios de pago (por ejemplo, el número de la tarjeta de débito).

La presente aplicación web coordina todo el proceso de pago en el lado del cliente mediante el script desarrollado *checkout.js*. En él, como se muestra en el listado [8](#), se ha usado el elemento *card* ofrecido en *Stripe.js*. En la figura [18](#) se muestra el resultado obtenido, siendo el citado elemento *card* la entrada del formulario donde se solicita la información de la tarjeta bancaria.

```
1 // ...
2
```

```

3  let stripe = Stripe(stripePublicKey);
4  let elements = stripe.elements();
5  let cardElement = elements.create('card');
6  cardElement.mount('#card-element');
7
8  // ...

```

Listing 8: Extracto de checkout.js donde se crea y enlaza el elemento *card*

Checkout

Recipient details

Name * Phone

Shipping address

Address *

City * Postal Code *

Province * Country *

Credit/Debit Card *

Card number MM / YY CVC

Figura 18: Formulario de checkout

Una vez se ha comentado todo lo anterior, se está en disposición de esbozar el flujo de información que tiene lugar durante el proceso de pago, tomando como apoyo el esquema presentado en la figura 19:

1. El cliente, estando en la vista de su cesta, decide proceder a finalizar la compra y hace click en el botón de ckeckout. Esto se traduce en una petición a FirstMarket al *endpoint* GET *~/user/checkout*.
2. Tal como se explica en 4.3, el sistema comprueba la disponibilidad de los libros pedidos y, si es el caso, compromete la cesta. En el proceso de comprometer la cesta contacta con Stripe para crear el *PaymentIntent*, especificando en este punto, entre otras cosas, la cantidad exacta a cobrar.
3. Stripe devuelve al sistema el *PaymentIntent* recién creado, el cual contiene el *client_secret*.
4. El sistema sirve la vista *checkoutForm.html* al cliente, la cual lleva inserta el *client_secret*. Resaltar en este punto la importancia del encriptado que

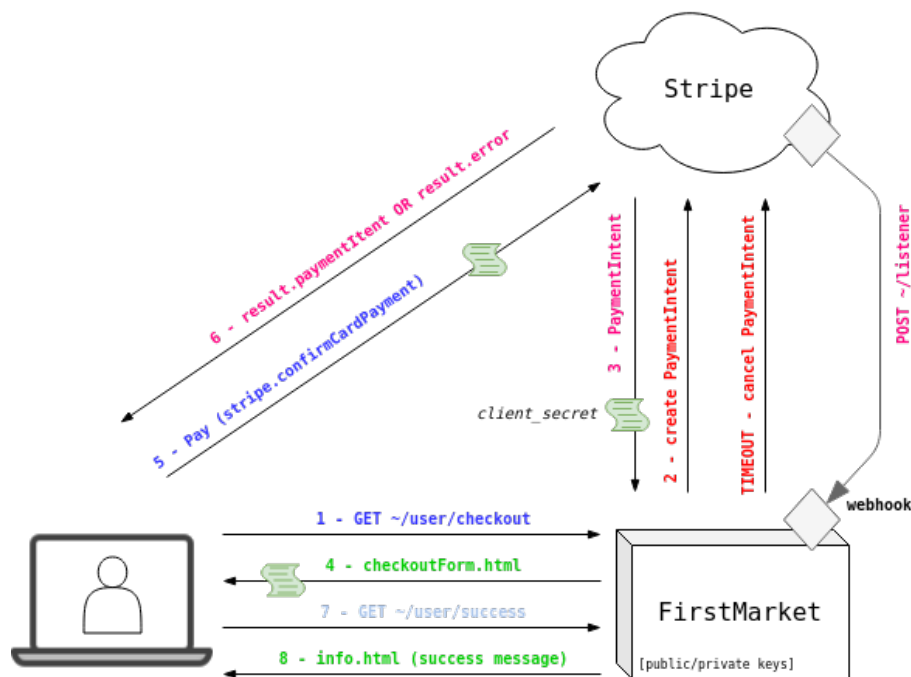


Figura 19: Flujo de información durante un pago

protocolos de transporte seguro proporcionan, ya que este código sólo debe ser conocido, fuera de Stripe y de Firstmarket, por el cliente. Así, como se explica en la sección 6.1, la aplicación web desarrollada emplea HTTPS.

- Tras completar correctamente el formulario de checkout (presentado en la figura 18), el cliente presiona el botón de pagar. Esto desencadena que desde *checkout.js* se invoque la función *confirmCardPayment* (perteneciente a la librería *Stripe.js*), tomando como parámetro, entre otros valores, el *client_secret*.
- Stripe procesa el pago y responde al cliente enviándole el objeto *result* (más adelante se explicará que en este punto, y en otros, Stripe notifica a FirstMarket a través del *webhook*). Dependiendo del éxito o no del pago, este objeto contendrá en su interior, respectivamente, el *PaymentIntent* o un objeto *error*. En ambos casos, *checkout.js* toma las medidas oportunas. En caso de éxito, se realiza lo explicado en el siguiente punto. En caso contrario se notifica por pantalla al cliente para que tome las medidas oportunas.
- checkout.js* realiza en background una petición a FirstMarket al *endpoint* GET *~/user/success*.

8. El sistema sirve la vista *info.html* al cliente, la cual le comunica que el pedido se ha realizado con éxito y que pronto recibirá confirmación por email.

Notar en la figura 19 que, tal como se explicó en la sección 4.3, cuando se produce el *timeout* y el pago no se ha efectuado se activa la vía de descomprometer la cesta y, por tanto, cancelar el `PaymentIntent`.

Por otro lado, podría pensarse que FirstMarket conoce de la finalización con éxito del pago a través de la solicitud que en el punto 7 anterior el cliente le realiza. Sin embargo, esa solicitud sólo es usada en backend como disparador para servir el mensaje estándar de finalización de una compra. Es decir, el sistema no realiza ninguna acción de lógica de negocio (no crea ningún pedido, de modifica ninguna cesta, etc.) al recibir la petición del punto 7. Sólo se limita a servir una vista con un mensaje. De hecho, cualquier cliente con sesión abierta en FirstMarket puede escribir en su navegador la dirección `~/user/success` y el sistema le devolverá el mensaje de compra finalizada (por supuesto, sin haber realizado ninguna compra en realidad).

Esto es así porque para conocer del estado de un pago FirstMarket no debe depender del cliente, sino de Stripe directamente. Y esta comunicación directa se realiza por medio del *webhook* configurado en Stripe. Esto es, Stripe permite configurar el envío de notificaciones POST ante los eventos que se desee, existiendo multitud de [eventos disponibles](#). FirstMarket escucha notificaciones POST provenientes de Stripe (se comprueba que es Stripe quien notifica al comparar la dirección IP de origen con las [oficiales](#) de Stripe, además de por existir un código de verificación en las notificaciones) en el *endpoint* `~/listener`. En concreto, se ha configurado la notificación de los eventos relacionados con el ciclo de vida de los `PaymentIntent`:

- *payment_intent.succeeded*. El pago ha concluido con éxito. Es al recibir esta notificación cuando el sistema realiza las acciones necesarias de lógica de negocio para, por ejemplo, crear el nuevo pedido.
- *payment_intent.canceled*. El `PaymentIntent` ha sido cancelado. Se procede, si fuese necesario, a descomprometer la cesta.
- *payment_intent.payment_failed*. Ocurre cuando el pago ha fallado. No se realiza ninguna acción aparte de su registro.
- *payment_intent.created*. El `PaymentIntent` ha sido creado. No se realiza ninguna acción aparte de su registro.
- *payment_intent.processing*. Ocurre cuando el `PaymentIntent` ha comenzado su procesamiento. No se realiza ninguna acción aparte de su registro.

- *payment_intent.amount_capturable_updated*. Se da cuando el PaymentIntent posee fondos aún por capturar. No se realiza ninguna acción aparte de su registro.

La principal consecuencia de la arquitectura explicada es que **FirstMarket nunca tiene contacto con la información sensible del cliente**. Esta se envía directamente a Stripe desde el cliente. Las implicaciones en términos de seguridad de este aspecto son profundas, mejorándose la exposición al riesgo de los clientes al tiempo que se alivian los requisitos y el coste de la aplicación web. De hecho, todo agente involucrado en el procesamiento, transmisión o almacenamiento de datos bancarios sensibles debe cumplir con los estándares del *Payment Card Industry Security Standards Council*. En este sentido, Stripe ha sido certificado como *PCI Level 1 Service Provider*, el nivel de certificación más estricto disponible en la industria de pagos. Por su parte, Firstmarket se alinea con los estándares PCI al no tener relación, como se ha dicho, con la información sensible de los clientes, y al garantizar la utilización de TLS en sus comunicaciones.

Como último apunte, Stripe no es gratis. Su modelo de negocio se basa en cobrar una comisión por cada transferencia que procesa. Para identificar a cada comercio ofrece un par de llaves, una pública y otra privada. Además, para facilitar las labores de desarrollo software, proporciona dos juegos de estas llaves: unas para *test* y otras *live*. Como su nombre indica, las primeras pueden ser usadas durante las labores de testeo de la integración, pudiendo simularse transacciones usando los datos de *tarjetas bancarias ficticias* ofrecidas por Stripe. El juego de llaves *live*, por su parte, permite realizar pagos reales. Sin embargo, dada la política de Stripe para el control de blanqueo de capitales, para activar este juego de llaves se debe presentar la documentación oficial de la constitución de la persona jurídica que recibe los pagos. Es por ello que en el presente proyecto sólo se ha empleado el juego de llaves *test*, aceptando únicamente transacciones simuladas. En la figura 20 se muestra el *dashboard* de Stripe, desde donde es posible controlar todo lo referente a la integración con esta plataforma. Se puede apreciar el requerimiento para proveer la información legal necesaria para desbloquear las llaves *live*.

4.3. Control de Concurrencia

Uno de los aspectos más críticos para el correcto funcionamiento de una tienda online es el control del stock de productos y del proceso de realización de una compra. Muchos usuarios, quizás simultáneamente, pueden añadir productos a sus cestas y proceder a su compra. La aplicación debe garantizar que

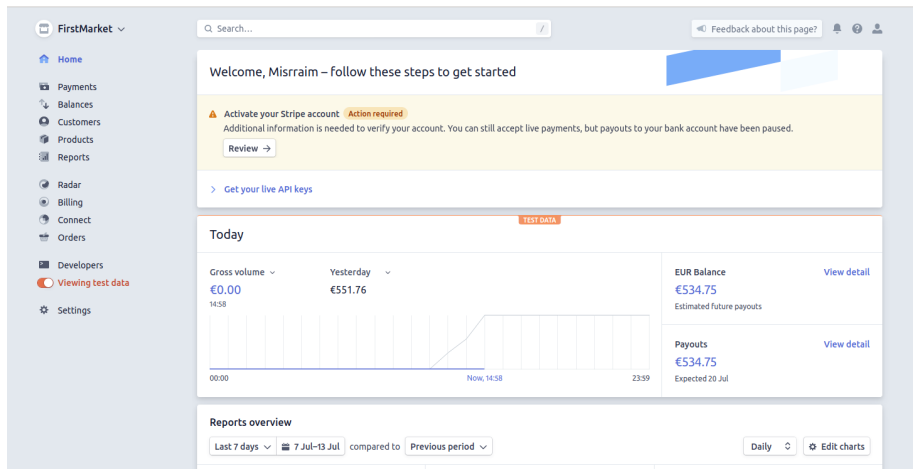


Figura 20: Stripe Dashboard

estas acciones se realicen de forma a conservar en todo momento la consistencia de los datos.

Como se puede apreciar en la figura 21, cuando un usuario decide proceder con el pedido de los libros contenidos en su cesta, lo primero que hace el sistema es comprobar que esos libros están disponibles. Pueden no estarlo por dos motivos, a saber, que estén agotados en ese momento o que el administrador de la aplicación web los haya deshabilitado (además, en este último caso el sistema automáticamente retira de la cesta dichos libros al detectar dicha situación). Si alguno de estos escenarios ocurriese la aplicación informaría al usuario en la propia vista de la cesta, sin avanzar a la vista de checkout, tal como se muestra en la figura 22.

Tras comprobar que los libros están disponibles el sistema **compromete** la cesta. Este es el paso fundamental donde se garantiza la consistencia. El sistema se compromete con el usuario a que si este finaliza la compra (dentro de un tiempo establecido) se le entregarán los libros seleccionados y al precio actual. Internamente, lo más destacado que el sistema realiza se recoge en:

- Se actualiza el stock, sustrayendo los libros en las cantidades correspondientes.
- Se crea en la base de datos una entidad *sale* por cada item de la cesta, para así capturar el estado (principalmente el precio) de los libros en ese momento.
- Se contacta con Stripe para crear un nuevo *Payment Intent*.
- Se activa el contador de tiempo disponible para finalizar la compra.

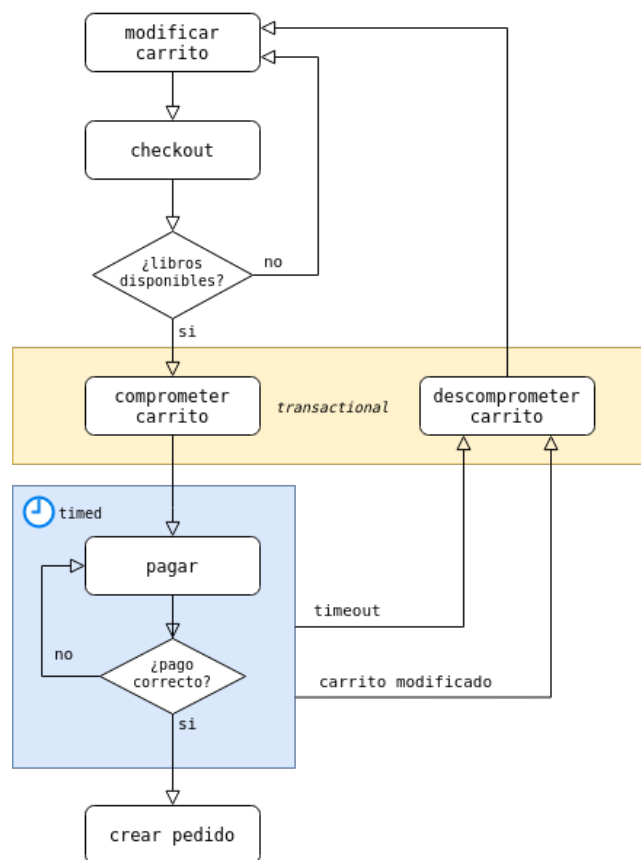


Figura 21: Diagrama del proceso de finalización de una compra

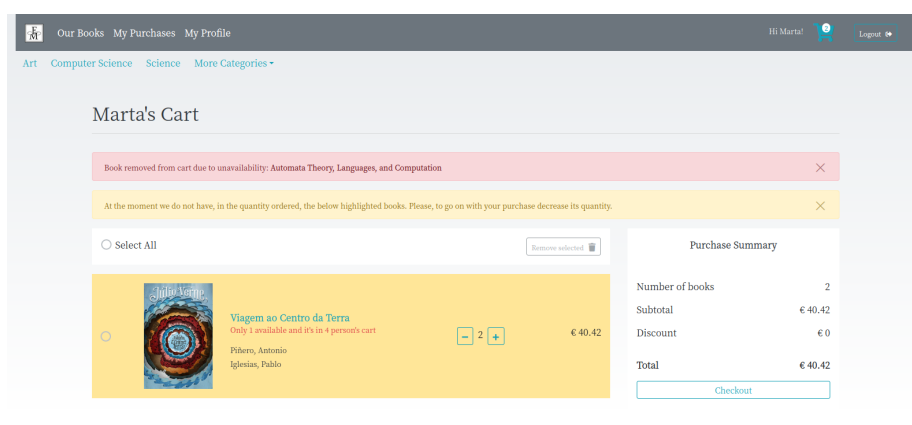


Figura 22: Aviso por pantalla de no disponibilidad de libros

Todas estas acciones que realiza el sistema para comprometer la cesta las lleva a cabo de forma transaccional, apoyándose en la API de Spring Data JPA

(ver 4.1.4). Las transacciones software se describen en términos de las características ACID, del acrónimo en inglés de Atomicity, Consistency, Isolation, y Durability:

- Atomicity. Cada paso en la secuencia de acciones realizadas dentro de los límites de una transacción deben completarse con éxito o todo el trabajo debe retroceder. La finalización parcial no es posible, o sucede todo o no sucede nada.
- Consistency. Los recursos de un sistema deben estar en un estado coherente, no corrupto, tanto en el inicio como en la finalización de una transacción.
- Isolation. El resultado de una transacción no debe ser visible para otras transacciones hasta que la primera se confirme con éxito.
- Durability. El resultado de una transacción comprometida debe hacerse permanente, independientemente a cualquier fallo del sistema.

Una vez comprometida la cesta, que otro usuario modifique el stock (ya sea un cliente al comprar un libro o el administrador al variar el stock), o que el administrador varíe el precio de algún libro de los recién comprometidos, o que incluso deshabilite alguno de estos libros, sería transparente para el usuario que acaba de comprometer su cesta. Mientras dure el tiempo disponible para finalizar la compra, la cesta comprometida es un contrato inmodificable. En este punto pueden suceder dos cosas:

1. El usuario completa el formulario de checkout y paga correctamente dentro del plazo, en cuyo caso se tramitaría el nuevo pedido.
2. El tiempo disponible para finalizar el pago se agota sin haber sido realizado, o el usuario, en la vista de la cesta, modifica las cantidades de los libros. Ante ambos eventos el sistema procede a **descomprometer** la cesta, abortándose de facto la compra.

El procedimiento de descomprometer la cesta es el inverso al de comprometerla, y también se realiza transaccionalmente. Así, lo más destacado que el sistema lleva a cabo internamente en este proceso se resume en:

- Se actualiza el stock, aumentando los libros en las cantidades correspondientes.
- Se eliminan las entidades *sale* correspondientes de la base de datos.
- Se contacta con Stripe para cancelar el *Payment Intent*.

4.4. Persistencia y Visualización de Datos Jerárquicos

Desde muy pronto se reparó en la necesidad de gestionar en la base de datos información con relaciones de jerarquía, ya que las categorías de los libros están organizadas de esta manera. Además, si se implementase la funcionalidad de que los usuarios pudiesen añadir comentarios a los libros, con capacidad de respuestas anidadas, también se estaría en el escenario de estructuras jerárquicas.

Tal como se explica con detalle en [bib ref], existen diversas formas de dar solución a esta necesidad, cada una de ellas con sus puntos fuertes y sus debilidades. La idoneidad de una solución frente a otra viene en gran medida determinada por la cantidad de información jerárquica a gestionar y por la frecuencia o importancia relativa de las operaciones de lectura, creación, actualización y eliminación.

Si los datos jerárquicos son siempre de pequeño tamaño y las operaciones son principalmente de lectura, una solución sería cargarlos en memoria principal y gestionarlos desde allí con alguna estructura de datos apropiada. Este podría argumentarse que es el caso de la información de las categorías de los libros, ya que en principio estas no superarían a lo sumo algunas decenas de centenas, y la actividad principal realizada sería la lectura. La frecuencia con que el administrador de la tienda crea, modifica o elimina una categoría es despreciable respecto de la frecuencia con que las categorías son leídas por los usuarios en general.

Por otro lado, si se implementase la funcionalidad de los comentarios anidados, el escenario es claramente diferente. La cantidad de información es potencialmente mucho mayor, con lo que trabajar directamente en memoria principal no es una opción. Además, las operaciones de creación, modificación y eliminación cobran mayor protagonismo.

La solución más común en este caso se conoce como listas de adyacencia, y no es otra cosa que añadir a cada entidad una referencia (clave extranjera) al id de su predecesor jerárquico. El problema de esta solución es que escala muy mal a medida que aumenta la profundidad del árbol. Imagínese que se tiene un hilo de comentarios arbitrariamente profundo, el cual precisaría de consultas recurrentes por cada nivel si se pretendiera extraer todo el hilo (algo habitual en estos sistemas de comentarios), ya que a priori se desconoce la profundidad. Sin embargo, existen métodos para extraer todo el hilo de comentarios con una sola consulta (en general, extraer cualquier subárbol), como se verá a continuación.

La funcionalidad de que los libros estén clasificados por categorías es imprescindible para la aplicación, y en consecuencia ha sido implementada. No es el caso de los comentarios. No obstante, en un intento de hacer la aplicación fácilmente ampliable en este sentido, y dado que se trata de un proyecto académico,

se optó por una solución que fuese eficiente y versátil: la denominada **Closure Table**.

La idea principal es mantener la información de las relaciones entre las entidades en una tabla diferente. Es decir, por una parte se encuentra la tabla *category* y por otra la tabla *catpath*. En la primera se almacena la información relativa a las categorías (su id, nombre, etc), mientras que en la segunda se almacena la información de los caminos en el árbol de categorías. De todos los caminos, incluso de una categoría consigo misma. Así, por cada fila en la tabla *catpath* se tiene el identificador del ancestro (clave extranjera de la tabla *category*), el identificador del descendiente (también clave extranjera de la tabla *category*) y el tamaño del camino. Para una relación de una categoría consigo misma el tamaño del camino es 0, para una relación directa el tamaño es 1, abuelo-nieto es 2 y así sucesivamente.

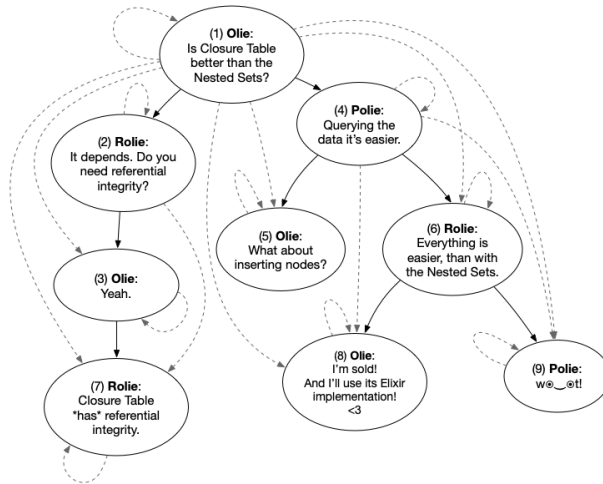


Figura 23: Diagrama de conexiones registradas en una closure table

Esta solución es la más versátil y rápida de todas las mostradas en [bib ref], permitiendo incluso a un nodo pertenecer a varios árboles. Sin embargo, estos beneficios los consigue a costa de espacio. Este consumo puede ser importante si la estructura almacena jerarquías muy profundas.

No obstante todo lo discutido con anterioridad, la solución real implementada para el trato específico de las categorías es triple (ver listado 9): Closure table, lista de adyacencia y trabajo en memoria principal con una estructura de datos en árbol. Esto es así por una cuestión de eficiencia y por desarrollar experiencia en el uso de estas soluciones.

```

1      create table category (
2          id bigint not null,
3          created_by varchar(255),
4          created_date timestamp,
5          last_modified_by varchar(255),
6          last_modified_date timestamp,
7          name varchar(255),
8          parent_id bigint,
9          primary key (id)
10     );
11
12     create table catpath (
13         id bigint not null,
14         created_by varchar(255),
15         created_date timestamp,
16         last_modified_by varchar(255),
17         last_modified_date timestamp,
18         size integer not null,
19         ancestor_id bigint,
20         descendant_id bigint,
21         primary key (id)
22     );
23
24     alter table category
25         add constraint fk_parentIdOnCategory foreign key (parent_id)
26             references category(id)
27     ;
28
29     alter table catpath
30         add constraint fk_ancestorIdOnCatpath foreign key (
31             ancestor_id) references category(id)
32     ;
33
34     alter table catpath
35         add constraint fk_descendantIdOnCatpath foreign key (
36             descendant_id) references category(id)
37     ;

```

Listing 9: Tablas que gestionan las categorías

Aparejado con el problema descrito en los párrafos anteriores se encuentra el de presentar al usuario dicha información jerárquica. Muy al principio del desarrollo de este proyecto se utilizó la tecnología de plantillas Java Server Pages para generar el contenido HTML, la cual permite insertar en las vistas código Java de servidor. Así, se desarrolló una vista que contenía una función recursiva que permitía generar código HTML que mostrase la estructura anidada de las categorías. Pero esta aproximación de mezclar HTML y código Java de servidor es considerada una mala práctica y está en desuso.

Posteriormente se adoptó el uso de Thymeleaf como motor de plantillas (ver

4.1.5), que no contempla esta posibilidad de insertar código de servidor en las vistas (al menos de la manera tan natural como lo permite JSP). Thymeleaf permite generar contenido lineal (una lista, por ejemplo) haciendo uso de la directiva *th:each*, que hace las veces de bucle *for*, pero para generar contenido anidado no dispone de ninguna funcionalidad nativa.

Por todo lo anterior, para resolver este problema se optó por desarrollar código JavaScript (*categoriesBuilder.js*) que se encargase, en el cliente, de construir y conectar, entre sí y al lugar apropiado en *categories.html* (ver línea 15 del listado 10), los componentes DOM necesarios para mostrar la estructura anidada de las categorías.

```
1  <div class="container-fluid fm-content">
2    ...
3    <!-- main content -->
4    <div class="row">
5      <div class="col-sm-1"></div>
6      <div class="col-sm-10" id="root-hook">
7        <!-- new category -->
8        <div class="my-2 d-flex">
9          <a class="btn btn-outline-secondary align-center px-4
10             ml-auto" th:href="@{/admin/categoryForm}">
11            <i class="fas fa-folder-plus fa-lg mr-4"></i>New
12              Category
13          </a>
14        </div>
15        <hr/>
16        <!--
17          js dynamic generated content here
18        -->
19      </div>
20    </div>
21  </div>
```

Listing 10: Contenido principal de la vista categories.html

Como resultado, el administrador de la aplicación web puede visualizar la estructura jerárquica de las categorías a través de despleables anidados, como se muestra en la figura 24.

4.5. Experiencia de Usuario Mejorada

Uno de los requisitos básicos de la aplicación web desarrollada es el de permitir a los usuarios gestionar su compra mediante el uso de una cesta virtual, que sirva de almacenamiento para la compra en curso. Esta funcionalidad permite

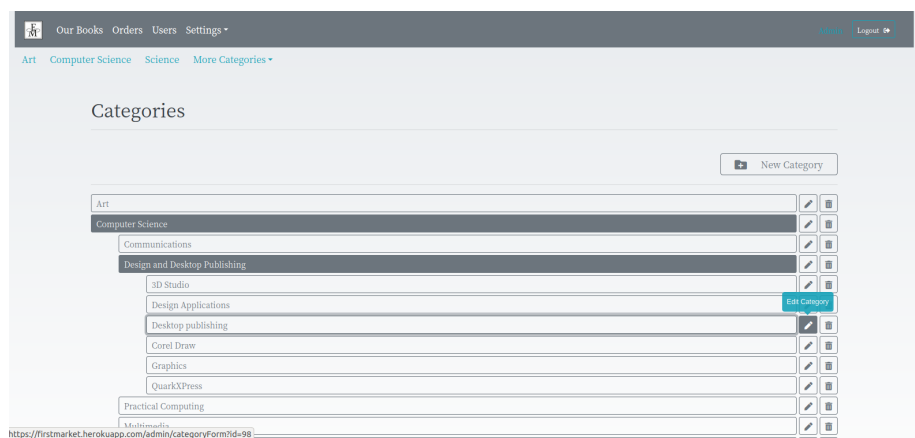


Figura 24: Visualización de la estructura anidada de las categorías

a los usuarios:

- Añadir libros a la cesta. Esto se puede realizar desde cualquier vista que muestre información de un libro, a saber, la página de inicio, la página de resultados de búsqueda y la página de detalles de un libro.
- Modificar la cantidad de un libro en la cesta, o eliminar un libro de la cesta. Estas acciones se pueden realizar desde la vista de la cesta de un usuario.

Para la gestión de estas acciones, la arquitectura clásica de petición de un recurso por parte del cliente y respuesta con contenido HTML por parte del servidor (con su consecuente refresco de la página) se juzgó inapropiada. Un usuario que esté visualizando la página de su cesta y aumente en una unidad un libro tendría que esperar, para ver el resultado de dicho aumento, a que el servidor responda con el nuevo contenido HTML y que el navegador lo renderice, todo ello para únicamente cambiar unos pocos números respecto de la página previa. Situaciones similares ocurren en el caso de disminuir la cantidad de un libro, o eliminarlo de la cesta por completo. Peor aún es el caso del usuario que decide añadir un libro a su cesta, ya que el contenido HTML por el que debiera esperar sería igual (salvo el icono del número de elementos en la cesta) al de la página desde donde se solicita tal acción.

En este contexto, y como pretexto perfecto para su estudio, se decidió desarrollar estas acciones con tecnología Ajax, de forma que no tenga lugar el refresco de la página en la que el usuario se encuentre. Así, el script *ajaxCart.js* se encarga, entre otras cosas, de actualizar el DOM de la vista de la cesta del usuario, *cart.html*, de manera consistente con las acciones que este realice, mientras que

el script *ajaxAddBook.js* gestiona el DOM de las citadas vistas desde las cuales es posible añadir libros a la cesta. Ambos scripts establecen en background la comunicación con el servidor, de forma transparente al usuario, haciendo uso del objeto *XMLHttpRequest*, tal como muestra el extracto de *ajaxCart.js* mostrado en el listado 11.

```
1  xmlhttprequest = new XMLHttpRequest();
2  xmlhttprequest.open("GET", url, true);
3  xmlhttprequest.setRequestHeader('isAjaxCartRequested', '1');
4  xmlhttprequest.send();
5  xmlhttprequest.onreadystatechange = function() {
6      if (this.readyState === 4 && this.status === 200) {
7          onHttpOk(action, id, this.responseText);
8      }
9      if (this.readyState === 4 && this.status === 401) {
10         onHttpUnauthorized();
11     }
12 };
```

Listing 11: Comunicación Ajax con el servidor

4.6. Registro de Libros Referenciados en Cestas

En este apartado se describe una funcionalidad añadida que, si bien no estaba presente en los requisitos de la aplicación, se ha mostrado muy útil y de relativa facilidad de implementación. La inspiración provino, como se aprecia en la figura 25, del portal de comercio electrónico [Etsy](#), al querer imitar su capacidad de informar de la situación en la que, para un producto determinado, queden igual o menos unidades en stock de las que están siendo referenciadas en cestas de los usuarios. Esto es, que un libro esté en la cesta de x usuarios diferentes, que de ese mismo libro queden en stock y unidades, y que x sea igual o mayor que y .

Con este objetivo en mente, se pensó en tres alternativas para implementarlo. Por un lado, la solución trivial sería realizar una consulta a la base de datos para conocer de esta situación cada vez que se necesite mostrar al usuario información de un libro. En este sentido, sería necesario explorar las cestas de todos los usuarios, para cada libro del cual se requiera información. No es desacertado estimar que la frecuencia con se requiere información de libros en una tienda de libros sea alta. Además, el número de usuarios puede ser todo lo grande que se pueda. Por ello, esta solución sería bastante pobre en tiempo de respuesta.

La segunda alternativa, con objeto a disminuir el tiempo de respuesta, sería aumentar la información de cada libro que se almacena en la base de datos, creando un nuevo campo en el que se contabilizase el número de referencias que a dicho libro le son realizadas en las cestas de los usuarios. Esta solución sería muy

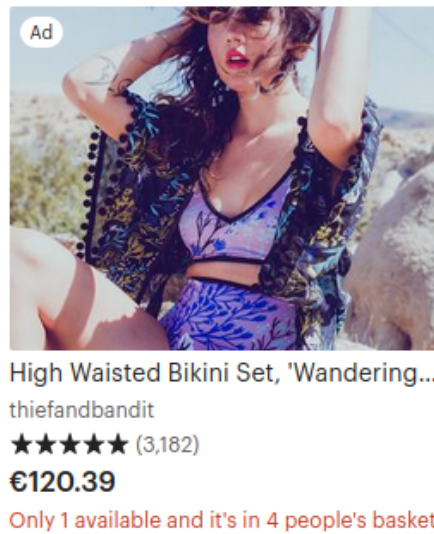


Figura 25: Aviso de producto escaso y demandado en Etsy.com

rápida, pero a costa de un uso redundante de los recursos de almacenamiento, puesto que la información necesaria *ya* estaba en la base de datos.

Finalmente, la tercera vía, que es la implementada, trata de aunar rapidez y no redundancia en la base de datos. Esto se consigue manteniendo la información necesaria en una estructura de datos llave-valor en memoria principal, sólo para los libros referenciados. La llave sería el *id* de un libro, y el valor el número de referencias en cestas que posea. Los tiempos necesarios para consultar esta estructura de datos, así como para actualizarla, son muy bajos. Además, la base de datos se mantiene no redundante. Es cierto que existe un grado de redundancia, pero esta es ajena a la base de datos, y se limita a los libros que estén referenciados (en contraste con la segunda solución, en la que *todos* los libros en la base de datos ampliaban su información).

Esta estructura de datos es mantenida eficientemente por la clase *BookServer*, como se aprecia en el listado 12, de manera que cada vez que algún usuario añade o elimina un libro de su cesta queda reflejado en el registro.

```
1  @Service
2  public class BookServer {
3
4      private final Map<Long,Integer> cartBookRegistry = new
        HashMap<>();
5
6      ...
7  }
```

```

8      public void incrementCartBookRegistry(Long cartBookId) {
9          cartBookRegistry.merge(cartBookId, 1, (oldValue,
10             defaultValue) -> ++oldValue);
11      }
12
13      public void decrementCartBookRegistry(Long cartBookId) {
14          cartBookRegistry.computeIfPresent(cartBookId, (key, value)
15             -> (value > 1L) ? --value : null);
16      }
17
18      public void incrementCartBookRegistry(List<Long> cartBookIds)
19      {
20          cartBookIds.forEach(this::incrementCartBookRegistry);
21      }
22
23      public Map<Long,Integer> getCartBookRegistry() {
24          return cartBookRegistry;
25      }

```

Listing 12: Gestión del registro de libros referenciados en cestas

Así, este registro es utilizado para avisar a los usuarios cuando un libro presenta escasez y alta demanda, como se muestra en la figura 26. Pero también es usado como **medida de la popularidad** de los libros, empleándose este uso en la página de inicio de la aplicación. Es cierto que el parámetro *popularidad* se puede definir de muchas maneras, y que sería conveniente que en él se reflejase el volumen de ventas en una ventana temporal local, pero como una primera aproximación de bajo coste al concepto funciona perfectamente.

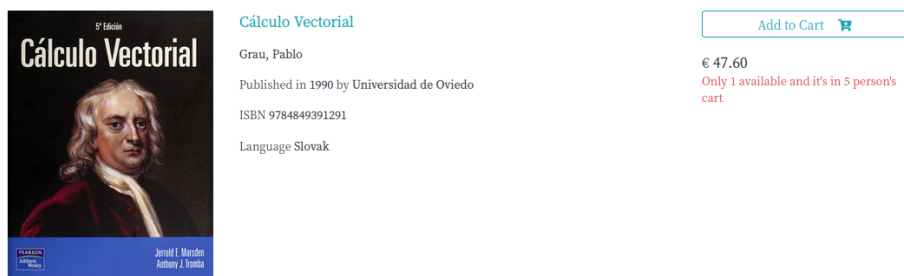


Figura 26: Aviso de producto escaso y demandado en FirstMarket

5. Despliegue

En esta sección se abordan aspectos relevantes en relación con el despliegue de la aplicación web desarrollada.

5.1. Heroku

Desde el surgimiento de las tecnologías web hasta nuestros días, las herramientas a disposición de los desarrolladores para abrir al público una aplicación web se han incrementado considerablemente. Desde tener cada uno que adquirir, configurar y mantener el hardware necesario, a los modernos servicios de *cloud-computing*.

La presente aplicación web ha sido desplegada en Heroku, una infraestructura de *cloud-computing* que sigue el modelo *platform as a service*.

Aparte de las ventajas que de por sí ofrece el modelo de computación en la nube, Heroku destaca por proporcionar una experiencia de desarrollo excepcional. Así, los desarrolladores sólo tienen que enfocarse en las tareas propias de programación, reduciéndose las labores de despliegue a unas pocas configuraciones y a efectuar un *git push*. Literalmente, tras una mínima configuración (en el común de los casos sólo necesaria la primera vez), desplegar la aplicación en Heroku se realiza enviando el repositorio Git a la plataforma (existen otras formas, como integrar GitHub o IntelliJ IDEA con Heroku). Es decir, a Heroku se le envía el código fuente del proyecto, tras lo cual automáticamente lo construye, configura el hardware necesario, y lo despliega.

Las aplicaciones desplegadas en Heroku son ejecutadas en contenedores Linux, denominados en la jerga Heroku como *dynos*. Cada uno de estos contenedores puede tener una configuración *Web* (los únicos que admiten tráfico HTTP), *Worker* o *One-off*. Además, Heroku ofrece varios tipos de *dynos*, cada uno adaptado a diferentes necesidades, desde pequeños proyectos hasta servicios en producción de gran tráfico. Así, como se muestra en la figura 27, tomada de la documentación oficial, cada tipo de dyno posee diferentes características, como la memoria disponible o la exclusividad de los recursos que utiliza.

Dyno Type	Memory (RAM)	CPU Share	Compute	Dedicated	Sleeps
free	512 MB	1x	1x-4x	no	yes
hobby	512 MB	1x	1x-4x	no	no
standard-1x	512 MB	1x	1x-4x	no	no
standard-2x	1024 MB	2x	4x-8x	no	no
performance-m	2.5 GB	100%	11x	yes	no
performance-l	14 GB	100%	46x	yes	no

Figura 27: Diferentes tipos de *dynos* en Heroku

FirstMarket ha sido desplegada en Heroku usando un *dyno* del tipo *hobby* con configuración *web*. En la figura 28 se muestra el panel central desde donde se gestiona el despliegue.

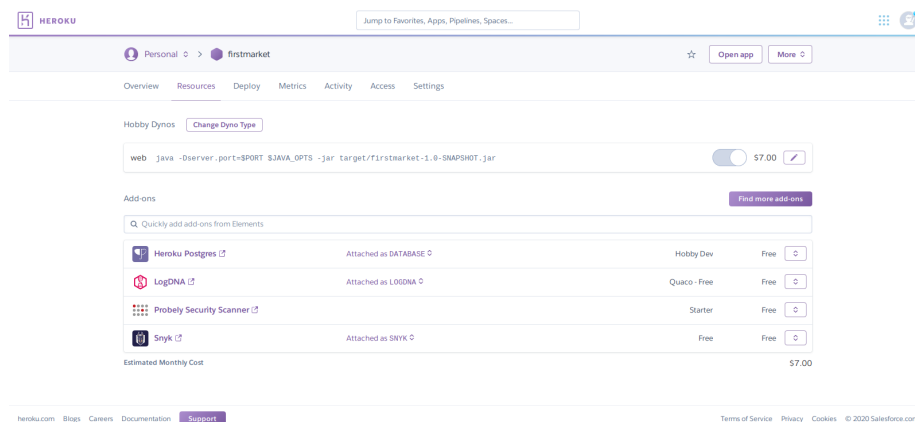


Figura 28: Panel de control del despliegue de FirstMarket en Heroku

5.2. Heroku Add-ons

Un aspecto interesante de Heroku es su oferta de *Add-ons*, accesible en su [marketplace](#). Se trata de módulos que se pueden añadir a la aplicación web, algunos gratuitos y otros de pago. La aplicación web desarrollada hace uso de los siguientes:

- [Heroku Postgres](#). Este es el servicio de base de datos utilizado para la capa de persistencia de la aplicación web cuando está desplegada en Heroku. El plan contratado es el gratuito, estando restringido a que el total de filas en la base de datos no supere las 10000, y con un máximo de 20 conexiones disponibles. En la figura 29 se muestra el panel central de este servicio. Destacar la posibilidad de crear *Dataclips*, que son consultas almacenadas sobre la base de datos. Como se aprecia, el número de filas se encuentra por debajo del límite.
- [LogDNA](#). Este servicio permite acceder a los registros emitidos, tanto por la Heroku como por la aplicación web, de una manera sencilla a través del navegador. Tiene multitud de opciones de configuración, siendo la interfaz gráfica la mostrada en la figura 30.
- [Probely](#). Este es un servicio de detección de vulnerabilidades de seguridad. En su versión pro, a la cual se ha tenido acceso gratuito durante dos semanas, permite realizar multitud de pruebas a la aplicación web, generando un informe de forma automática con los problemas encontrados y sus posibles soluciones. Se evaluó la aplicación web desarrollada con dos test diferentes: un análisis en profundidad estándar de Probely y un análisis OWASP top 10. Los resultados son comentados en la sección 6.5.

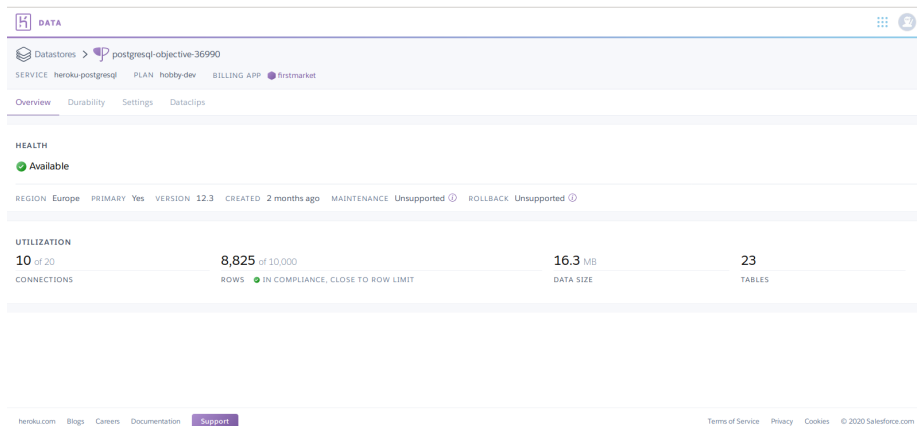


Figura 29: Panel de control de Heroku Postgres

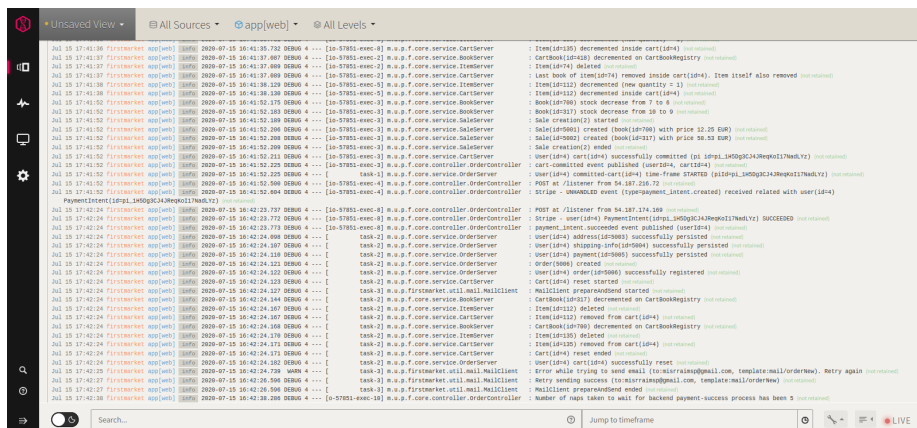


Figura 30: Registros con LogDNA

- **Snyk**. Este add-on revisa las dependencias del proyecto en busca de vulnerabilidades de seguridad conocidas. En su versión gratuita permite realizar hasta 200 test al mes, pero no emite informes es formato pdf. Los resultados son comentados en la sección 6.5.

5.3. firstmarket.tech

Por defecto, Heroku muestra sus aplicaciones web en el dominio *nombre_aplicacion.herokuapp.com*, aunque permite de forma sencilla y sin coste adicional **añadir otros dominios**. Así, para la apertura al público de FirstMarket se ha comprado el dominio **firstmarket.tech**, a través del registrador de dominios **namecheap**.

Una vez adquirido el dominio, se debe configurar sus registros DNS para que apunten a Heroku, que es donde se está ejecutando la aplicación. La figura

31 muestra dichos registros DNS configurados en el panel de control que ofrece namecheap.





<input type="checkbox"/>	Type	Host	Value	TTL	
<input type="checkbox"/>	CNAME Record	www.firstmarket.te...	radiant-indominus-t4h65gtpx8iwrswwa1f3lherokudns.com.	30 min	
<input type="checkbox"/>	ALIAS Record	@	human-lion-r70gc7fg4sys3hyim7r1ja3y.herokudns.com.	5 min	
<input type="checkbox"/>	CNAME Record	www	firstmarket.tech.	30 min	
 ADD NEW RECORD					

Figura 31: Configuración de los registros DNS de firstmarket.tech en namecheap

6. Seguridad

La palabra seguridad es empleada en multitud de situaciones dentro del mundo de las tecnologías de la información, normalmente con significados dispares. No obstante, generalizando lo suficiente, viene a referirse a la protección que se debe proporcionar a determinados recursos, por su valor y/o sensibilidad, frente a posibles usos malintencionados.

Esta sección está dedicada a las cuestiones de seguridad, en sentido amplio, relacionadas con [FirstMarket](#).

6.1. HTTPS

Como se ha comentado, una de los principales retos que afronta el comercio electrónico es la generación de confianza. Los usuarios demandan seguridad a la hora de realizar sus compras por la red. En este sentido, el desarrollo de protocolos seguros de comunicación ha jugado un papel protagonista en el rápido incremento que el comercio *online* ha vivido en las últimas décadas.

HTTPS, la extensión segura de HTTP, es el protocolo de comunicación segura por excelencia en la actualidad. Los datos enviados con HTTPS son protegidos mediante el protocolo Transport Layer Security (TLS), el cual brinda tres tipos de protección:

- Privacidad. Los datos son encriptados antes de moverse por la red, de forma que nadie ajeno a la conversación pueda comprender su semántica.
- Integridad. De una manera similar a como Git detecta los cambios en los documentos que gestiona, TLS calcula determinados parámetros criptográficos que hacen extremadamente difícil que los datos sean alterados

en su camino por la red, ya sea accidental o intencionadamente, sin que el protocolo lo detecte.

- Autenticidad. Aún teniendo garantizado que nadie podrá entender los datos enviados, y que nadie los podrá alterar sin notarse, persiste la duda de si se está comunicando con el interlocutor pretendido. Esto es, cuando alguien accede a `www.firstmarket.tech` debe tener la certeza de que efectivamente está intercambiando información con FirstMarket (hipotética persona jurídica). Para esta labor TLS hace uso de los certificados digitales, que un tercer agente *de confianza* emite garantizando que la web de FirstMarket es `www.firstmarket.tech`. Este agente es el conocido en la industria como *Certificate Authority* (CA). Pero, ¿quién establece la autoridad de este CA, es decir, quién asegura que es *de confianza*? Otro CA, de mayor autoridad, y así sucesivamente. Esto crea una cadena de confianza, finita, englobada dentro de la *Public Key Infrastructure* (PKI), la tecnología fundamental de Internet para construir la confianza acerca de la autenticidad de los agentes en la red.

La aplicación web desarrollada puede usar HTTPS, ya que por defecto este es el protocolo que Heroku proporciona para la comunicación de sus aplicaciones ejecutadas con *dynos* de pago (como es el caso). En concreto, por defecto y sin coste añadido, Heroku gestiona los certificados digitales de las aplicaciones automáticamente con su servicio *Automated Certificate Management* (ACM). Por su parte, ACM utiliza *Let's Encrypt*, un CA abierto, gratuito y automatizado que ofrece para el bien común el *Internet Security Research Group*.

Para finalizar, resaltar que `www.firstmarket.tech` impone el uso de HTTPS, es decir, los intentos de comunicarse con la aplicación web por medio de HTTP son redirigidos al uso de la versión segura. Esto se ha implementado con Spring Security, como se muestra en el listado 13, donde se detalla el fragmento del método *configure* que especifica este comportamiento.

```
1  @Override
2  protected void configure(HttpSecurity httpSecurity) throws
    Exception {
3      httpSecurity
4          //...
5          .and()
6          .requiresChannel()
7          .antMatchers("/**")
8          .requiresSecure()
9          //...
10 ;
```


Listing 13: Configuración de Spring Security para imponer el uso de HTTPS

6.2. Spring Security

Un enfoque comúnmente empleado, en el ámbito de la seguridad de la información, se basa en considerar los sistemas complejos como un conjunto de capas de abstracción, aplicándose los mecanismos de seguridad apropiados para cada una de ellas.

Si se piensa en la aplicación de comercio electrónico que se ha desarrollado, esta puede modelarse en esencia como un proceso que se ejecuta en el contexto de un sistema operativo, en el seno a su vez de una máquina, la cual se comunica con otras haciendo uso de la infraestructura de Internet. Este modelo simple ya es capaz de hacer contraste entre la aplicación, el sistema operativo, la máquina y la red. Cada una de estas capas implementa sus propias medidas destinadas a proteger la información que maneja.

A nivel de aplicación, los conceptos de autenticación y autorización (o acceso) son nucleares en su ámbito de seguridad. Autenticación hace referencia a la capacidad de la aplicación para identificar al usuario. Es decir, desde el punto de vista de la aplicación, responde a la pregunta ¿quién me está usando? Autorización, por su parte, refleja la capacidad de la aplicación de permitir, o denegar, el acceso a un recurso o funcionalidad de la aplicación a un determinado usuario. Como puede notarse, para que la aplicación pueda *autorizar* a un usuario, debe primero *autenticar* a dicho usuario.

Spring Security es la piedra angular de la seguridad a nivel de aplicación del portal de comercio electrónico desarrollado, integrando las funcionalidades de autenticación y autorización encima de las características propias de Spring Framework. Además, como se verá, también ofrece mecanismos de protección frente a multitud de vulnerabilidades comunes a nivel de aplicación.

6.2.1. FilterChain

Como se comenta en la sección 4.1.4, las aplicaciones web con Spring basadas en el stack servlet, como es el caso, siguen el patrón *front controller*, en el que el servlet *DispatcherServlet* ejerce de punto de entrada y distribuidor de las peticiones HTTP hacia los *@Controller*/*@RestController* adecuados.

El *DispatcherServlet* no integra lógica de autenticación/autorización, por lo que, en ausencia de alternativas, esta debería ser implementada en las clases *@Controller*/*@RestController* en las que delega las peticiones HTTP. Esto, por supuesto, destrozaría el principio de separación de responsabilidades.

Afortunadamente, sí que existe alternativa, los Java [Filters](#). Estos componentes interceptan el tráfico entrante y/o saliente de los servlets, permitiendo inyectar la lógica que se desee. Así, la autenticación/autorización puede ser implementada en estos *filters*, de forma que las labores de seguridad sean llevadas a cabo antes de que las peticiones HTTP alcancen el *DispatcherServlet*.

Además, los *filters* poseen la capacidad de concatenarse entre sí, creando un *filter chain* o cadena de filtros, como se muestra en la figura 32, tomada de la documentación oficial de Spring Security. Esto permite, otra vez, separar responsabilidades, e insertar en cada filtro la lógica de una funcionalidad concreta.

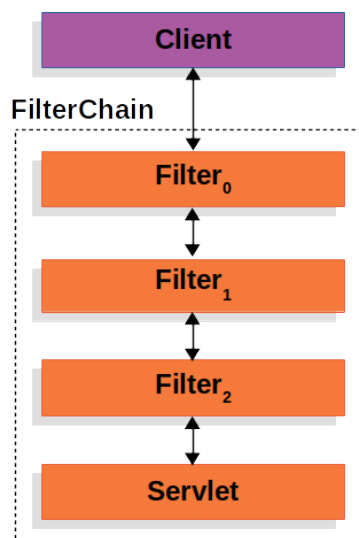


Figura 32: Encadenamiento de filtros

Por ejemplo, una petición HTTP podría primero alcanzar un filtro destinado a recabar las credenciales de autenticación del usuario (por ejemplo un formulario de *login*). Después, con dichas credenciales, atravesar el filtro que las comprueba, y autenticar así o no al usuario. Seguidamente podría estar el filtro que decide si el usuario recién autenticado posee el permiso necesario para acceder al recurso o funcionalidad pedida. Por último, en caso necesario, la petición HTTP llegaría al *DispatcherServlet*.

En el listado 14 se aprecia el mensaje de registro emitido durante el arranque de la aplicación en el que se detalla la cadena de filtros que Spring Security proporciona. Pues bien, en esencia, **Spring Security es esta concatenación de filtros** antes del *DispatcherServlet*.

```
1 2020-07-18 17:20:19.718 DefaultSecurityFilterChain:
```

```

2   Creating filter chain: any request ,
3   [
4   ChannelProcessingFilter ,
5   WebAsyncManagerIntegrationFilter ,
6   SecurityContextPersistenceFilter ,
7   HeaderWriterFilter ,
8   CsrfFilter ,
9   LogoutFilter ,
10  UsernamePasswordAuthenticationFilter ,
11  RequestCacheAwareFilter ,
12  SecurityContextHolderAwareRequestFilter ,
13  AnonymousAuthenticationFilter ,
14  SessionManagementFilter ,
15  ExceptionTranslationFilter ,
16  FilterSecurityInterceptor
17  ]

```

Listing 14: Cadena de filtros creada por Spring Security

6.2.2. Configuración

En estos filtros Spring Security implementa la autenticación/autorización y la protección frente a determinados tipos de ataques. Lo que resta es configurarlos para satisfacer las necesidades propias de cada aplicación. Esta configuración se lleva a cabo extendiendo la clase de Spring Security *WebSecurityConfigurerAdapter* y sobrescribiendo su método *configure(HttpSecurity)* (expuesto en el listado 15), tarea que en la aplicación web desarrollada realiza la clase *SecurityConfig*.

```

1   @Override
2   protected void configure(HttpSecurity httpSecurity) throws
      Exception {
3       httpSecurity
4           // authenticate through login form
5           .formLogin()
6           .loginPage("/login")
7           .failureHandler(customAuthenticationFailureHandler())
8
9           // set logout page
10          .and()
11          .logout()
12          .logoutSuccessUrl("/home")
13
14          // authorization
15          .and()
16          .authorizeRequests()
17          // protect role_admin resources

```

```

18     .antMatchers("/admin/**")
19     .access("hasRole('ROLE_ADMIN')")
20     // protect role_user resources
21     .antMatchers("/user/**")
22     .access("hasRole('ROLE_USER')")
23     // publicly open
24     .antMatchers("/", "/home", "/login", "/newUser")
25     .access("permitAll")
26
27     // force HTTPS always
28     .and()
29     .requiresChannel()
30     .antMatchers("/**")
31     .requiresSecure()
32
33     // enable csrf protection
34     .and()
35     .csrf()
36     .ignoringAntMatchers("/listener") // open for stripe
37
38     // always create new session
39     .and()
40     .sessionManagement()
41     .sessionCreationPolicy(SessionCreationPolicy.ALWAYS)
42
43     .and()
44     .headers(headers -> headers
45         // HTTP Strict Transport Security
46         .httpStrictTransportSecurity(hsts -> hsts
47             .includeSubDomains(true)
48             .preload(true)
49             .maxAgeInSeconds(31536000)
50         )
51         // Content Security Policy
52         .contentSecurityPolicy(csp -> csp
53             .policyDirectives("default-src 'self' https://*.stripe.
54                 com" +
55                 "; " +
56                 "script-src 'self' 'unsafe-inline' " +
57                 "https://*.stripe.com " +
58                 "https://*.fontawesome.com " +
59                 "https://maxcdn.bootstrapcdn.com " +
60                 "https://cdnjs.cloudflare.com " +
61                 "https://ajax.googleapis.com" +
62                 "; " +
63                 "style-src 'self' 'unsafe-inline' " +
64                 "https://cdn.jsdelivr.net " +
65                 "https://maxcdn.bootstrapcdn.com " +
66                 "https://fonts.googleapis.com " +
67                 "https://*.fontawesome.com" +

```

```

67         "; " +
68         "font-src 'self' " +
69         "https://fonts.gstatic.com " +
70         "https://*.fontawesome.com" +
71         "; " +
72         "img-src 'self' data:"
73     )
74 )
75 // Referrer Policy
76 .referrerPolicy(referrer -> referrer
77     .policy(ReferrerPolicyHeaderWriter.ReferrerPolicy.
78         STRICT_ORIGIN)
79 )
80 ;
81 }

```

Listing 15: Configuración de Spring Security en FirstMarket

6.2.3. Autenticación

Spring Security ofrece toda una gama de métodos de autenticación. El usado en FirstMarket, configurado entre las líneas 5 y 7 del listado 15, ha sido a través de un formulario, contenido en la vista *login.html*, en el que el usuario introduce su dirección de correo electrónico y su contraseña. En caso de un fallo de autenticación, la clase *custom.AuthenticationFailureHandler* toma el control, determina la causa del fallo y presenta el mensaje adecuado al usuario.

Es importante resaltar que, con el fin de garantizar la seguridad de la información de los usuarios, la aplicación web desarrollada almacena las contraseñas en la base de datos encriptadas. En concreto, se usa el algoritmo **Bcrypt**. De esta forma, para saber si la contraseña es correcta, la aplicación web encripta la contraseña enviada en el formulario y la compara con la que tiene almacenada.

Además, Spring Security permite especificar la lógica tras un *logout*, que en el caso de la presente aplicación es simplemente una redirección a la página de inicio, como se muestra en las líneas 12 y 13 del listado 15.

Autenticación por Fuerza Bruta Como ya se ha comentado, el ecosistema Spring destaca por su extensibilidad. Allá donde se requieran funcionalidades especiales, el framework permite alcanzarlas. En este sentido, aunque Spring Security no implementa ningún mecanismo para prevenir la autenticación por fuerza bruta, sí ofrece facilidades para incorporarlo.

Cada vez que un usuario trata de autenticarse, en función del éxito o del motivo del fracaso, Spring Security emite el correspondiente evento. Si la autenticación tuvo éxito se emite el evento *AuthenticationSuccessEvent*, mientras que

si falla por ser la información proporcionada incorrecta se emite *AuthenticationFailureBadCredentialsEvent*. Otros motivos de fracaso (por ejemplo estar la cuenta de usuario bloqueada por el administrador) emiten diferentes eventos.

Escuchando estos eventos, la aplicación web desarrollada implementa, con las clases contenidas en el paquete *firstmarket.security*, una protección frente a intentos de autenticación por fuerza bruta. Esto es, se impide la posibilidad de probar sistemáticamente un número muy grande de contraseñas para descubrir cuál es la correcta.

Para ello, la clase *LockManager* mantiene un registro caché (usando la clase *LoadingCache* de la librería [Guava](#)) con el que llevar el conteo de los intentos fallidos de autenticación, debido a credenciales incorrectas, dentro de una ventana temporal.

Así, dentro de un tiempo determinado se restringe el número de intentos con credenciales fallidas, de forma que, si se supera el límite, la cuenta de usuario queda bloqueada temporalmente (por un tiempo igual a la ventana temporal). Por contra, si dentro de dicho intervalo, y sin haber superado el límite de intentos, se proporcionara las credenciales correctas, el registro se limpia. Además, cada entrada en la caché se limpia automáticamente tras superarse un tiempo igual a la ventana disponible sin que se haya producido un bloqueo.

Tanto el número de intentos permitidos (*num-of-attempts*) como la ventana temporal (*locking-minutes*) son externalizados en el fichero de propiedades de la aplicación, *application.yml*. En el listado 16 se muestra su configuración en producción.

```
1  fm:
2    security:
3      lock:
4        num-of-attempts: 5
5        locking-minutes: 60
```

Listing 16: Parámetros para configurar la prevención de autenticación por fuerza bruta

6.2.4. Autorización

Como se puede apreciar en el listado 15, líneas de la 15 a la 26, FirstMarket implementa tres tipos de recursos:

- Los que sólo pueden ser accedidos por los usuarios con rol *ROLE_ADMIN*, ubicados en *~/admin/***.
- Los que sólo pueden ser accedidos por los usuarios con rol *ROLE_USER*, ubicados en *~/user/***.

- El resto, que puede ser accedido por cualquiera.

Notar que los dos primeros tipos requieren necesariamente autenticación previa.

6.2.5. Cross-Site Request Forgery

Cross-Site Request Forgery representa un tipo de ataque que toma ventaja de la confianza que un determinado sitio web tiene en un cliente, por ejemplo porque previamente se ha autenticado. Así, el cliente envía (sin saberlo) una petición (maliciosa) al sitio web y este la acepta dada la confianza que deposita en el primero. Es el caso inverso a los ataques *Cross-Site Scripting*, donde se saca partido de la confianza que el cliente tiene en el sitio web, ejecutando el código (malicioso) que este le envía (sin saberlo).

Imagínese que un usuario se encuentra, autenticado, en la aplicación web de su banco, en la vista que le permite transferir fondos de su cuenta a otra. Esta vista, por ejemplo, mostraría un formulario como el mostrado en el listado 17. Al completarlo y presionar *Transfer* se enviaría al servidor del banco la petición HTTP mostrada en el listado 18.

```
1 <form method="post" action="/transfer">
2   <input type="text" name="amount"/>
3   <input type="text" name="account"/>
4   <input type="submit" value="Transfer"/>
5 </form>
```

Listing 17: Formulario inseguro para la transferencia de fondos

```
1 POST /transfer HTTP/1.1
2 Host: bank.example.com
3 Cookie: JSESSIONID=randomid
4 Content-Type: application/x-www-form-urlencoded
5
6 amount=100.00&account=9876
```

Listing 18: Petición HTTP insegura para la transferencia de fondos

Hasta aquí todo normal, pero imagínese que, antes de realizar *logout* en la aplicación del banco, se visita otro sitio web, el cual sirve al cliente el formulario mostrado en el listado 19. Si en esta situación el usuario presiona *Win Money!* se enviará al servidor una petición HTTP exactamente igual que la mostrada en el listado 18, salvo el número de cuenta a donde se envía el dinero. Es decir, el usuario habrá sido estafado.

```

1 <form method="post" action="https://bank.example.com/transfer">
2   <input type="hidden" name="amount" value="100.00"/>
3   <input type="hidden" name="account" value="evilAccountNum"/>
4   <input type="submit" value="Win Money!"/>
5 </form>

```

Listing 19: Formulario para la transferencia ilegítima de fondos

El banco no tiene posibilidad de distinguir si alguna de las dos peticiones es ilegítima, ya que en ambas cree que las está haciendo el usuario autorizado para ello. ¿Por qué? porque el medio que utiliza para identificar al usuario, la *cookie* de la línea 3 del listado 18, es exactamente igual en ambas peticiones, ya que el navegador por defecto adjunta las cookies en todas las peticiones.

La manera más extendida de solucionar este problema, que Spring Security implementa por defecto, es adjuntar una pieza de información extra (*token*) que sólo conozca el usuario legítimo. Obviamente, este *token* no puede ser colocado en una *cookie*, porque se estaría en la misma situación vulnerable anteriormente descrita. En este sentido, existen diversas alternativas de lugares donde colocar el *token*, siendo en un campo oculto (llamado *_csrf*) la utilizada por defecto en Spring Security. Así, el formulario que el banco enviaría ahora al usuario sería como el mostrado en el listado 20, mientras que la petición HTTP sería la mostrada en el listado 21.

```

1 <form method="post" action="/transfer">
2   <input type="hidden" name="_csrf" value="4bfd1575-3ad1-4d21-96
   c7-4ef2d9f86721"/>
3   <input type="text" name="amount"/>
4   <input type="hidden" name="account"/>
5   <input type="submit" value="Transfer"/>
6 </form>

```

Listing 20: Formulario seguro para la transferencia de fondos

```

1 POST /transfer HTTP/1.1
2 Host: bank.example.com
3 Cookie: JSESSIONID=randomid
4 Content-Type: application/x-www-form-urlencoded
5
6 amount=100.00&account=9876&_csrf=4bfd1575-3ad1-4d21-96c7-4
  ef2d9f86721

```

Listing 21: Petición HTTP segura para la transferencia de fondos

De esta forma, el agente malintencionado ya no podrá tener acceso a la información necesaria para suplantar la identidad del usuario legítimo, teniendo ahora el banco un mecanismo para distinguirlos.

Como se ha dicho, esta funcionalidad se implementa por defecto con Spring Security. Sin embargo, como se aprecia en las líneas 34 a 37 del listado 15, se ha modificado ligeramente el comportamiento por defecto para permitir que desde Stripe se realicen peticiones POST sin necesidad de contar con el token `_csrf`.

6.2.6. Encabezados de Seguridad en las Respuestas HTTP

Los principales navegadores web implementan diversos controles de seguridad que entran en acción cuando desde el servidor se les especifica la configuración deseada. Los [encabezados de seguridad](#) insertados en las respuestas HTTP cumplen este papel. Esto es, son directrices que los servidores proporcionan a los clientes web, con el objetivo de establecer la configuración de la política de seguridad que se desea que el navegador implemente.

Cache Control Por defecto, Spring Security instruye al navegador, por medio de las cabeceras de respuesta mostradas en el listado 22, para que no guarde en su caché ninguna información, debido al riesgo que esto representa para el usuario. Piénsese, por ejemplo, en una situación en la que un usuario, tras autenticarse, consulta información confidencial, que el navegador almacena en caché. Después, tras cerrar su sesión, otro usuario podría tener acceso a dicha información privada sin más que consultar las páginas guardadas en el historial de navegación.

```
1 Cache-Control: no-cache, no-store, max-age=0, must-revalidate
2 Pragma: no-cache
3 Expires: 0
```

Listing 22: Cabeceras para impedir el almacenamiento en caché

No obstante, para el contenido estático (principalmente hojas de estilo y código JavaScript), sí se ha permitido al navegador almacenarlo en su caché, con el fin de mejorar el tiempo de carga de las páginas y ganar en experiencia de usuario, sin merma de seguridad.

Content Type Options Normalmente, cuando el campo *content type* (tipo de contenido) no es especificado, los navegadores tratan de deducirlo para mejorar la experiencia de usuario. Por ejemplo, si el navegador encuentra un archivo JavaScript sin el tipo de contenido especificado, intentará deducir su naturaleza a partir del propio contenido y, en caso de que finalmente lo juzgue como código JS, lo ejecutará. A este procedimiento de inferencia del tipo de contenido se le conoce como [content sniffing](#)

Este sistema de inferencia del tipo de contenido presenta el problema de ser un vector para ataques XSS (*cross-site scripting*). En efecto, existen archivos contruidos de tal manera que pueden ser interpretados de varias formas al mismo tiempo. Por ejemplo, una aplicación web puede permitir a los usuarios enviar un documento PostScript válido para posteriormente visualizarlo. Un usuario malintencionado podría crear un documento PostScript que también sea un archivo JavaScript válido y ejecutar un ataque XSS con él.

Por estas razones, la configuración que por defecto ofrece Spring Security es instruir al navegador para que no realice *content sniffing*, a través de la cabecera mostrada en el listado 23. Notar que, como contraparte, ahora en todo momento se debe indicar el tipo de contenido en lo que se envíe al navegador para que funcione correctamente.

```
1 X-Content-Type-Options: nosniff
```

Listing 23: Cabecera para impedir el *content sniffing*

HTTP Strict Transport Security Cuando un usuario teclea la dirección web del sitio que pretende visitar, normalmente no especifica que la comunicación se haga a través de HTTPS. Es decir, se suele escribir sólo *firstmarket.tech* en lugar de *https://firstmarket.tech*. Esto representa un riesgo, ya que las comunicaciones HTTP son vulnerables a ataques *man in the middle*. Es más, aunque la comunicación HTTP sea redirigida a usar la versión segura (como es el caso en la aplicación web desarrollada), la vulnerabilidad comentada persistiría entre la comunicación inicial con HTTP y la redirección.

Para gestionar esta situación se creó el **HTTP Strict Transport Security (HSTS)**, un mecanismo para que los sitios web puedan declararse como *HTST host*, esto es, como accesibles únicamente mediante conexiones seguras.

Una manera de alcanzar este estatus es ingresar en la lista de dominios web *preloaded* gestionada en hstspreload.org. En la figura 33 se muestra el mensaje tras el envío del formulario de inscripción.

Otra forma es incluyendo el encabezado *Strict-Transport-Security* en las respuestas HTTP. En las líneas de la 46 a la 51 del listado 15 se muestra la configuración realizada en Spring Security para incluir la cabecera mostrada en el listado 24 en las respuestas HTTP.

```
1 Strict-Transport-Security: max-age=31536000 ; includeSubDomains  
; preload
```

Listing 24: Cabecera de declaración *HSTS host*

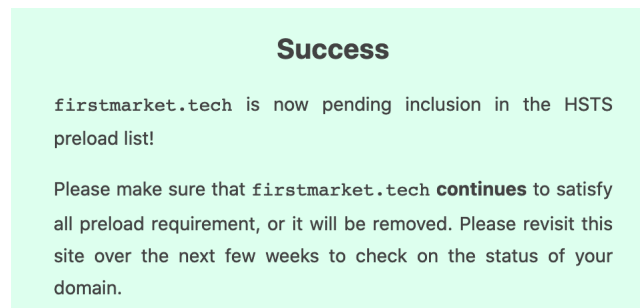


Figura 33: Candidatura para la inclusión en la lista de dominios web *preloaded*

X-Frame-Options Con el objetivo de prevenir los ataques conocidos como *clickjacking*, Spring Security inhabilita el renderizado de vistas que contengan etiquetas HTML5 *iframe*. Esto se consigue enviando a los navegadores el encabezado que se muestra en el listado 25.

```
1 X-Frame-Options: DENY
```

Listing 25: Cabecera que inhibe el uso de las etiquetas *iframe*

X-XSS-Protection Muchos navegadores implementan filtros anti-XSS. Aún no representando una solución infalible al problema, ayudan mucho a su mitigación. Normalmente estos filtros están habilitados por defecto, siendo posible configurar qué hacer en caso de un positivo (por ejemplo, intentar enmendar el contenido detectado).

Spring Security, por defecto, a través del encabezado que se muestra en el listado 26, activa explícitamente el filtro del navegador y lo configura para que ante un positivo bloquee el contenido.

```
1 X-XSS-Protection: 1; mode=block
```

Listing 26: Cabecera que activa y configura el filtro anti-XSS del navegador

Content Security Policy La *Content Security Policy* (CSP) es un mecanismo de protección frente a vulnerabilidades de inyección de contenido, como por ejemplo el comentado *cross-site scripting*. A través de esta funcionalidad se puede declarar y, en última instancia, informar al navegador acerca de cuáles son las fuentes autorizadas desde las cuales descargar los recursos que las vistas de la aplicación web necesita.

Esta política de permisos, no trivial, depende de las necesidades de cada sitio web, y es por ello que Spring Security no la configura por defecto. Así, en

el listado 15, entre las líneas 52 y 75, se declara la CSP de la aplicación web desarrollada. Como se aprecia, se establecen los dominios desde los cuales obtener 4 tipos de recursos: los *scripts*, las hojas de estilo, las fuentes y las imágenes. También se establece que todo el contenido fuera de lo explicitado sólo pueda ser obtenido desde *firstmarket.tech* y desde **.stripe.com*.

Referrer Policy Uno de los muchos encabezados que se incluyen en una petición HTTP es el *Referer*, que contiene la dirección de la página web anterior desde la cual se realizó la navegación a la página actual. Por otro lado, el encabezado *Referrer-Policy* controla qué restricciones aplicar a la información suministrada en el encabezado *Referer*.

Es importante establecer un criterio apropiado ya que, por ejemplo, si no se especifica política alguna que restrinja la información contenida en el encabezado *Referer*, este podría contener el token de seguridad que FirstMarket adjunta en el link que envía a los usuarios para verificar la dirección de correo electrónico. En general, la información sensible que puedan contener las URL quedaría expuesta en el encabezado *Referer*.

Spring Security permite de una manera sencilla configurar la política deseada, que en el caso de FirstMarket ha sido ***strict-origin***, como se aprecia en las líneas 76 y 79 del listado 15. Esta política establece que sólo se envía información en el encabezado *Referer* si el destinatario usa HTTPS. Esta es la parte *strict* de *strict-origin*. La parte *origin* significa que sólo se envía el origen de la dirección. Por ejemplo, de *www.firstmarket.tech/login?error=true* sólo sería enviado *www.firstmarket.tech*.

6.3. Validación de Entradas

Por validación se entiende las comprobaciones que la aplicación web hace sobre los datos que le son proporcionados externamente. El caso más común es cuando un usuario envía un formulario cumplimentado a la aplicación web, y esta, antes de procesar la información, comprueba que se cumplen ciertas reglas. Por ejemplo, puede requerirse que el campo de email sea efectivamente una dirección de correo válida, o que la fecha de nacimiento sea una fecha pasada válida (nadie puede nacer el 30 de febrero, por ejemplo). Esto se hace para garantizar la consistencia de los datos en la base de datos y, quizás más importante, para evitar ataques de inyección de contenido.

Esta comprobación se puede hacer tanto en cliente como en servidor. Desde el punto de vista de la seguridad, realizar la comprobación en servidor es **obligatorio**, ya que las comprobaciones en cliente son muy fácilmente circunvaladas. Por otro lado, desde el punto de vista de la experiencia de usuario es

muy recomendable implementarla también en cliente, ya que permite al usuario tener un *feedback* mucho más rápido que si debiese esperar a que el servidor responda. En definitiva, la mejor práctica es implementar la validación tanto en *backend* como en *frontend*, y así se ha implementado en la aplicación web desarrollada. Resaltar, además, que los componentes de validación en *backend* están en el paquete *util.validation*.

Mucha es la información que FirstMarket debe validar, pero a grandes rasgos se puede dividir en numérica y textual. El primer grupo presenta poca dificultad, se trata de comprobar rangos de valores principalmente. Para el segundo grupo las expresiones regulares son el arma perfecta. En el archivo de configuración *application.yml* se detallan las expresiones regulares y los rangos numéricos utilizados. A modo de ejemplo, la propiedad definida en este archivo de configuración que especifica la expresión regular contra la que validar las contraseñas se muestra en el listado 27.

```
1  fm:
2    validation:
3      regex:
4        password: ^(?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{8,16}$
```

Listing 27: Expresión regular para las contraseñas

Así, esta expresión regular es utilizada por la aplicación web para comprobar, tanto en *frontend* como en *backend*, que cuando un usuario proporciona una nueva contraseña esta tenga una longitud de entre 8 y 16 caracteres e incluya al menos una minúscula, una mayúscula y un dígito.

Existe un campo que destaca por su manera de ser validado. Se trata del ISBN (International Standard Book Number) de los libros. Este campo necesita validación en dos vertientes. La primera se realiza utilizando las ya mencionadas expresiones regulares, pero, para validar si se trata de un ISBN válido, aún hay que calcular el dígito de control y comprobar que coincida con el último dígito del código. Los ISBN tuvieron 10 dígitos hasta diciembre de 2006, pero desde entonces tienen siempre 13 (ambas versiones con diferentes algoritmos de cálculo del dígito de control). Así pues, la validación implementada en FirstMarket da soporte a ambos formatos de ISBN.

6.4. Stripe Radar

TODO

6.5. Análisis de Vulnerabilidades

Como se comentó en la sección 5.2, la seguridad de la aplicación web ha sido testada con la ayuda de [Probely](#) y [Snyk](#). La primera herramienta detectó 5 vulnerabilidades, mientras que la segunda 13. Del total, 14 responden a vulnerabilidades conocidas en librerías utilizadas por la aplicación web, por lo que su resolución se basa en actualizar las versiones de dichas dependencias. A continuación se pasa a comentar los problemas encontrados, haciendo especial énfasis en las 4 vulnerabilidades propias de la aplicación web desarrollada:

- **Reflected cross-site scripting.** Esta vulnerabilidad, considerada de **alto riesgo**, fue encontrada durante el test OWASP top 10 realizado con Probely.

Este riesgo se encuentra bastante extendido por Internet y ocurre cuando el servidor de la aplicación toma un input del cliente y lo devuelve nuevamente sin validar o codificar adecuadamente para ser mostrado por pantalla. El adjetivo *reflected* proviene del código malicioso que se envía al servidor y se refleja en el navegador de la víctima en el código fuente de la página.

En el caso concreto de la presente aplicación, esta vulnerabilidad se encontró en la vista de las categorías, *categories.html*. En esta vista, como se explica en la sección 4.4, el código HTML necesario para mostrar la estructura de categorías anidadas se genera dinámicamente con *categoriesBuilder.js*. Para que este *script* pueda realizar su función, el servidor incrusta dentro de *categories.html* la información de las categorías en formato JSON.

```
1  ...
2  <!-- categories dynamically generated -->
3  <i id="catHolder" th:attr="jsonStringCategories=${
4      jsonStringCategories}"></i>
5  <!-- insecure: reflected XSS
6  <script th:utext="'let jsonStringCats = \'' + ${
7      jsonStringCategories} + '\';'"></script>
8  -->
9  <script th:src="@{/js/categoriesBuilder.js}"></script>
10 ...
```

Listing 28: Extracto de categories.html con la vulnerabilidad encontrada

Como se aprecia en la línea 5 del listado 28, la manera vulnerable de hacer disponible la información de categorías a *categoriesBuilder.js* es haciendo uso de la directiva de Thymeleaf **th:utext** en la declaración de la variable

JavaScript *jsonStringCats*, donde la *u* significa *unscaped*. Esta manera de renderizar la información del servidor es el origen del problema, ya que los caracteres especiales que puedan contener los nombres de las categorías no son apropiadamente codificados.

Una manera segura, que resuelve el problema, es la implementada en la línea 3 del mismo listado 28, donde se utiliza la directiva de Thymeleaf *th:attr* para incrustar la estructura JSON como un atributo HTML. Esta directiva sí codifica adecuadamente los caracteres especiales, mitigando el riesgo de insertar código malicioso en el nombre de las categorías y que sea servido y expuesto a los clientes.

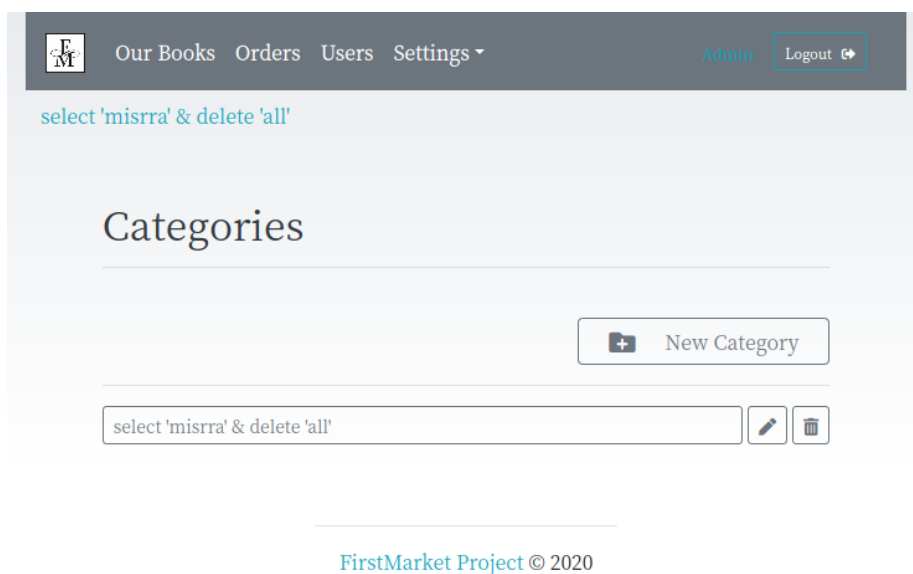


Figura 34: Categoría con caracteres especiales en su nombre

En la figura 34 se muestra un nombre de categoría con caracteres especiales, exponiéndose en el listado 29 un extracto de su código fuente, generado con ambos métodos: el seguro en la línea 3 y el inseguro, sin codificar los caracteres, en la línea 5.

```

1  ...
2  <!-- categories dynamically generated -->
3  <i id="catHolder" jsonStringCategories="{ &quot;name&quot;
   ;:&quot;firstmarket&quot;;, &quot;id&quot;;:6, &quot;
   children&quot;;:[{ &quot;name&quot;;:&quot;select &#39;
   misrra&#39; &amp; delete &#39;all&#39;&quot;;, &quot;id
   &quot;;:68, &quot;children&quot;;:[ ]}]}"></i>
4  <!-- insecure: reflected XSS

```

```

5   <script>let jsonStringCats = '{ "name":"firstmarket", "id
      ":6, "children":[{" "name":"select 'misrra' & delete '
      all'", "id":68, "children":[]}]}'</script>
6   -->
7   <script src="/js/categoriesBuilder.js"></script>
8   ...

```

Listing 29: Extracto del HTML generado a partir del template categories.html

- **Potential DoS on TLS Client Renegotiation.** Esta vulnerabilidad (CVE-2011-1473) está basada en la asimetría de trabajo que tiene lugar cuando el cliente inicia una renegociación de los parámetros criptográficos de una conexión TLS con el servidor, soportando este la mayor carga de trabajo. Así, un cliente malicioso podría iniciar una conexión TLS y seguidamente requerir el procedimiento de renegociación repetidas veces, en busca de colmar los recursos del servidor.

La renegociación para acordar nuevos parámetros de cifrado se desarrolla dentro de una única conexión TCP. Esto es clave, ya que otros ataques DoS relacionados con TLS se basan en inundar al servidor con nuevas conexiones TCP, contra lo cual muchos cortafuegos implementan algún tipo de control del ritmo de peticiones de nuevas conexiones TCP. Pero esta vulnerabilidad explota que la renegociación tiene lugar dentro de una única conexión TCP, evitando así este tipo de controles.

Para controlar esta amenaza en la aplicación web desarrollada sería necesario implementar una restricción sobre el ratio de peticiones de renegociación iniciadas por el cliente, o simplemente deshabilitar esta capacidad. Se ha optado por la segunda opción, dada la sencillez de su implementación y el bajo o nulo impacto en la experiencia de usuario. Así, para deshabilitar esta opción, Java 8 introdujo una nueva variable de la JVM, de forma que únicamente es necesario invocar la aplicación con la opción detallada en el listado 30.

```

1   -Djdk.tls.rejectClientInitiatedRenegotiation=true

```

Listing 30: Deshabilitado del inicio por parte del cliente de la renegociación TLS

- **Outdated TLS protocol version 1.0 supported.** Como se explica en la sección 6.1, Heroku ofrece conexiones seguras con TLS, por defecto, a través de su funcionalidad *Automated Certificate Management*. La vulnerabilidad descrita en este apartado se produce por el hecho de que ACM no fuerza el uso de TLS v1.2+, es decir, no deshabilita el uso de versiones obsoletas de TLS.

Tal como se explica en su [documentación](#), para deshabilitar el soporte a las versiones inseguras, TLS v1.0 y TLS v1.1, sería necesario sustituir ACM por [SSL Endpoint](#), el cual es un servicio con un coste de 20 dólares mensuales.

Teniendo en cuenta el coste indicado, y la reciente (8 Jul 2020) notificación recibida desde Heroku, mostrada a seguir, se ha optado por continuar usando ACM:

Dear Heroku Customer,

At Salesforce, our top priority is providing you with a trusted Heroku platform, and today we begin our migration off of older, less secure TLS versions with a plan to completely block TLS v1.0/v1.1 next year after July 31, 2021. [...]

Heroku currently supports TLS v1.0/v1.1, as well as the latest, more secure TLS v1.2+ protocol on all apps. [...]

Today, Heroku begins implementing these recommendations to transition all apps to TLS v1.2+, so that we can End of Life TLS v1.0/v1.1 next year. [...]

Beginning on June 1, 2021, we will begin migration all apps to the new cipher suites and block TLS v1.0/v1.1 completing this migration by July 31, 2021.

After July 31, 2021, clients that access Heroku apps using TLS v1.0/v1.0 will be blocked. [...]

Sincerely, Heroku.

- ***Referrer policy not defined.*** La vulnerabilidad comentada en este punto se debe a que no se había especificado qué política de *Referrer* emplear, con el consecuente riesgo de enviar información sensible en dicho encabezado. Como se explica en la sección [6.2](#), el riesgo queda mitigado al especificar la política *strict-origin* con Spring Security.
- ***Vulnerabilidades en librerías de terceros.*** Las restantes 14 vulnerabilidades se han solucionado por medio de actualizaciones. Por la propia naturaleza de la herramienta, todas los problemas detectados por Snyk

entran dentro de este epígrafe, mostrándose en la figura 35 dos de las vulnerabilidades más graves que ha detectado. Las actualizaciones realizadas han sido las siguientes:

1. org.springframework.boot:spring-boot → 2.3.1.RELEASE
2. org.springframework:spring-web → 5.2.4.RELEASE
3. org.springframework:spring-webmvc → 5.2.4.RELEASE
4. org.hibernate:hibernate-core → 5.4.18.Final
5. org.springframework.security:spring-security-core → 5.3.2.RELEASE
6. org.apache.tomcat.embed:tomcat-embed-core → 9.0.37
7. org.webjars:jquery → 3.5.1

Un aspecto que debe resaltarse es que las actualizaciones de la 2 a la 6, incluidas ellas, han tenido que establecerse sobrescribiendo la configuración que por defecto ofrece Spring Boot a través del fichero *spring-boot-starter-parent*, del cual hereda el *pom.xml* de la aplicación web desarrollada. Esto es importante porque se está saliendo del terreno seguro que esta configuración por defecto suministra, en el sentido de que desde Spring se ha testado que las versiones establecidas por defecto funcionan sin problemas de compatibilidades entre sí.

HIGH SEVERITY

🛡️ Reflected File Download (RFD)

Vulnerable module: org.springframework:spring-web

Introduced through: org.springframework.boot:spring-boot-starter-web@2.3.1.RELEASE and org.springframework.boot:spring-boot-starter-security@2.3.1.RELEASE

Detailed paths

- **Introduced through:** misraimsp/firstmarket@misraimsp/firstmarket#c6ba79c854e58d6c1db6556236ca885042a41c56 › org.springframework.boot:spring-boot-starter-web@2.3.1.RELEASE › org.springframework:spring-web@5.2.2.RELEASE
- **Introduced through:** misraimsp/firstmarket@misraimsp/firstmarket#c6ba79c854e58d6c1db6556236ca885042a41c56 › org.springframework.boot:spring-boot-starter-security@2.3.1.RELEASE › org.springframework:spring-security-web@5.2.1.RELEASE › org.springframework:spring-web@5.2.2.RELEASE
- **Introduced through:** misraimsp/firstmarket@misraimsp/firstmarket#c6ba79c854e58d6c1db6556236ca885042a41c56 › org.springframework.boot:spring-boot-starter-web@2.3.1.RELEASE › org.springframework.boot:spring-boot-starter-json@2.3.1.RELEASE › org.springframework:spring-web@5.2.2.RELEASE

[...and 1 more](#)

Overview

org.springframework:spring-web is a package that provides a comprehensive programming and configuration model for modern Java-based enterprise applications - on any kind of deployment platform.

Affected versions of this package are vulnerable to Reflected File Download (RFD). A reflected file download attack is possible when the filename attribute of the Content-Disposition header is derived from user-supplied input.

[Reflected File Download \(RFD\) vulnerability report](#)

HIGH SEVERITY

NEW

🛡️ SQL Injection

Vulnerable module: org.hibernate:hibernate-core

Introduced through: org.springframework.boot:spring-boot-starter-data-jpa@2.3.1.RELEASE

Detailed paths

- **Introduced through:** misraimsp/firstmarket@misraimsp/firstmarket#c6ba79c854e58d6c1db6556236ca885042a41c56 › org.springframework.boot:spring-boot-starter-data-jpa@2.3.1.RELEASE › org.hibernate:hibernate-core@5.4.17.Final

Overview

org.hibernate:hibernate-core is a library providing Object/Relational Mapping (ORM) support to applications, libraries, and frameworks.

Affected versions of this package are vulnerable to SQL Injection. A SQL injection in the implementation of the JPA Criteria API can permit unsanitized literals when a literal is used in the SELECT or GROUP BY parts of the query. This flaw could allow an attacker to access unauthorized information or possibly conduct further attacks.

[SQL Injection vulnerability report](#)

Figura 35: Ejemplo de vulnerabilidades encontradas por Snyk