Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

Grado en Ingeniería Informática

Trabajo Fin de Grado

Actualización y Mejora del Sistema de Entrega de Prácticas Deliverit

Autor: Fernando Prados Vicente
Tutor(a): Ángel Herranz Nieva

Madrid, Abril 2025

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado
Grado en Ingeniería Informática

Título: Actualización y Mejora del Sistema de Entrega de Prácticas Deliverit

Abril 2025

Autor: Fernando Prados Vicente
Tutor: Ángel Herranz Nieva
Departamento de Lenguajes y Sistemas Informáticos e Ingeniería de Software
Escuela Técnica Superior de Ingenieros Informáticos
Universidad Politécnica de Madrid

Dedico este trabajo a mi familia,
por confiar siempre en mí.

Y a Ángel Herranz, porque se
necesitan más profesores como él.

Resumen

La *ETSIINF* oferta desde hace años una amplia variedad de grados. La facultad, debido al interés en cursarlos, atrae a una gran cantidad de alumnos, con la gestión tanto administrativa como educativa correspondiente. En los currículos de la escuela se disponen de varias materias con distintos enfoques en la programación. Sin embargo, la mayoría cuenta con un componente práctico que requiere de una compleja gestión para su elaboración, entrega y corrección. Por ello se presentan y utilizan distintas alternativas de software para atacar a este problema.

Desde 2020 se ha venido desarrollado la plataforma en la cual se delimita este Trabajo de Fin de Grado, *DeliverIt*. Elaborada a partir de la contribución conjunta de profesores, estudiantes y aportaciones externas a la facultad, este proyecto busca otorgar un entorno en el cual se facilita el acceso y entrega de prácticas, así como una corrección automatizada. En la búsqueda de su soporte y mejora he podido intervenir bajo la tutela del profesor Ángel Herranz.

El proyecto se desarrolla bajo el paradigma de la programación funcional y se apoya en nuevas metodologías y procesos, junto con un robusto despliegue de infraestructura. Todo ello busca ofrecer una solución accesible a los problemas tradicionales en la enseñanza de la ingeniería informática.

Abstract

The ETSIINF has been offering a wide variety of degrees for years. The faculty, due to the interest in these programs, attracts a large number of students, necessitating the corresponding administrative and educational management. The curricula of the school include several subjects with different focuses on programming. However, most of them have a practical component that requires complex management for its preparation, submission, and grading. Therefore, various software alternatives are presented and utilized to address this issue.

Since 2020, the platform that delineates this Bachelor's Thesis, DeliverIt, has been developed. Created from the joint contributions of professors, students, and external inputs to the faculty, this project aims to provide an environment that facilitates access to and submission of practical assignments, as well as automated grading. In the pursuit of its support and improvement, I have been able to intervene under the guidance of Professor Ángel Herranz.

The project is developed under the paradigm of functional programming and is supported by new methodologies and processes, along with a robust infrastructure deployment. All of this seeks to offer an accessible solution to the traditional problems in the teaching of computer engineering.

Tabla de contenidos

Capítulo 1

Introducción

En este capítulo se incluye una descripción acerca de Deliverit, las motivaciones, sus objetivos y una guía sobre las secciones del documento y su lectura.

1.1. Deliverit

Deliverit es una plataforma de gestión de prácticas de programación. Encabezada por los profesores Ángel Herranz Nieva y Lars-Åke Fredlund, su desarrollo ha progresado desde sus inicios en 2020 con la ayuda de varios estudiantes.

A nivel tecnológico se presenta con una interfaz en navegador web (la opción más conocida), pero también cuenta con una API REST mínima. El proyecto Deliverit trabaja con tecnologías modernas como Elixir, un lenguaje de programación funcional creado en el año 2012, o Docker, un software de aislamiento de aplicaciones en contenedores. Por otro lado, las funcionalidades giran en torno a las prácticas. El sistema permite la creación, gestión de entrega y corrección automatizada de las mismas; así también de la gestión de los alumnos y el control de acceso para su realización.

El aporte más significativo a la comunidad educativa es la sistematización estandarizada. La utilización de la herramienta en los estudios académicos hace más viable la formación con componentes prácticos para grandes cantidades de estudiantes. La robustez y la delegación (o facilitación) de las tareas del profesorado da acceso a una formación más diversa, útil e innovadora.

1.2. Motivación

La Escuela Técnica Superior de Ingeniería Informática ofrece una cartera de estudios cuyo centro es la informática. Esta rama de la ingeniería se fundamenta en la programación siendo, por tanto, fundamental en sus itinerarios. Sus asignaturas se basan en el aprendizaje mediante la aplicación de los conceptos teóricos provistos en las lecciones. Algunas de las formas de aplicaciones prácticas en la enseñanzan son: ejercicios en clase, que son pruebas pequeñas sobre

las lecciones recientes; prácticas de laboratorio, pruebas destinadas a probar problemas de forma más extensa bajo la dirección o ayuda de un profesor; o las prácticas virtuales, en las cuales no se necesita la presencialidad y pueden suponer un trabajo más intenso.

El uso de prácticas desarrolladas en casa como método de evaluación de conocimientos supone varios problemas de base:

- Primero, la disponibilidad del enunciado de la práctica. La entrega física del enunciado a todos los estudiantes puede suponer un problema, por lo que se suele optar por plataformas digitales de acceso común como *Moodle*.
- Segundo, la entrega de las prácticas. Cada una puede suponer unos plazos distintos, y llevar a cabo la validación de la forma y de los plazos puede resultar tedioso.
- Tercero, la corrección de las prácticas. Aún pudiendo no ser evaluables, suele ser un proceso muy demandado. La complejidad de la tarea a realizar aumenta también la dificultad de correción de una práctica individual.
- Cuarto, la cantidad de prácticas. La cantidad de personas y prácticas posibles limitan su uso debido a que el tiempo del profesorado es el cuello de botella más común en su gestión.

El presente proyecto busca reducir estos problemas mediante una plataforma digital que permita a los profesores gestionar sus prácticas en un entorno especializado en ello. Se debe plantear la automatización tanto de las pruebas como de la valoración de las prácticas además de facilitar la descarga de enunciados y la validación de los ficheros de prueba.

El sistema de prácticas también debería de ser capaz de gestionar los alumnos de forma individual, grupal y a nivel de convocatorias académicas, con interfaces sencillas y accesibles. Ya existen algunas soluciones propuestas de forma parcial, como correctores automatizados de exámenes tipo test utilizados en la facultad, pero se necesitaba una solución completa y especializada para abarcar las necesidades específicas de su potencial capacidad de utilización.

Los potenciales beneficios que se pueden derivar de implementar un elemento con estas características afectarían a todas las partes. Los profesores reducirían su tiempo de gestión de prácticas y podrían enfocarse, mejorar e innovar otras partes del currículo. Por otro lado, para los alumnos, la automatización y una plataforma digital permitiría mayor rapidez y sencillez a la hora de trabajar y aprender con las prácticas. Finalmente la facultad ETSIINF y el equipo gestor del proyecto podrían mejorar la calidad académica y ofrecerlo como solución para entidades externas a la propia entidad.

1.3. Lectura del documento

A continuación, y para ayuda del lector, se redacta un listado con las secciones de interés del Trabajo de Fin de Grado **recomendaciones**. Las secciones se

clasifican en *capítulos*, con el contenido principal, y *anexos*, con aportaciones auxiliares para la compensión del trabajo realizado.

- **Capítulo 2 - Preliminares**, donde se revisan más en profundidad el estado actual de Deliverit y los problemas a resolver.
- **Capítulo 3 - Funcionalidades**, donde se detalla el comportamiento y las funcionalidades a desarrollar durante el proyecto.
- **Capítulo 5 - Desarrollo**, donde se incluyen las tecnologías empleadas, el diseño para resolver las funcionalidades e implementación de las mismas.
- **Capítulo 6 - Conclusiones y trabajo futuro**, donde se muestran las dificultades técnicas encontradas y el resultado del trabajo. Además se explica el estado final y futuro de los objetivos.

Se recomienda realizar la lectura en el orden de presentación de las secciones. Sin embargo, no es mala idea saltar puntualmente a alguno de los anexos para entrar más en detalle del contenido o la resolución de dudas.

Capítulo 2

Preliminares: Deliverit

En este capítulo se muestra el estado actual de Deliverit, tanto a nivel funcional como a nivel interno, y las tareas a realizar.

2.1. El Sistema de Entrega Deliverit

Deliverit cuenta con dos accesos distintos a las funcionalidades integradas en la aplicación. Por un lado, el profesorado y los administradores del servicio tienen acceso mediante la **aplicación de Admin** a las herramientas de gestión de prácticas, asignaturas y alumnos. Aquí también se incluye todo lo necesario para la actividad de los administradores. Por otro lado, la **aplicación de Alumno** está destinada al alumnado de los profesores y asignaturas registrados. Ambos accesos son aplicaciones webs ligeras y de diseño sencillo.

2.1.1. Aplicación de Admin

El acceso a la aplicación de Admin da lugar a un menú principal ante el cual se muestran las funcionalidades disponibles para sus usuarios. Aquí se muestran: la gestión de alumnos, de asignaturas, de prácticas, de entornos y la cola de entregas. Las funcionalidades se muestran encerradas en cajas en el centro del diseño, a las cuales se accede a través de un botón para cada una.

La gestión de alumnos y de prácticas permiten acceder a sus menús correspondientes para gestionar todos los alumnos y todas las prácticas actuales del sistema. En el primero se puede tanto añadir estudiantes, editar sus datos y gestionar su acceso a asignaturas como su eliminación del sistema. Para la gestión de prácticas ocurre de forma similar. Si se selecciona alguna práctica en específico, se pueden ver datos como entregas, grupos o correciones. También desde el menú permite la creación y personalización de prácticas nuevas.

El menú de asignaturas (figura 2.1) ofrece un acceso más controlado a los alumnos y prácticas por parte del profesorado. Está disponible la gestión completa del listado de asignaturas, pero lo interesante sucede al ver los detalles de una de ellas. Cada asignaturaspermitenver y controlar los alumnos añadidos a la misma, pero también permite gestionar las prácticas definidas para ella. Este menú suele ser la vista más consultada por el profesorado con asignaturas con prácticas reales en los planes de estudio de la ETSIINF.

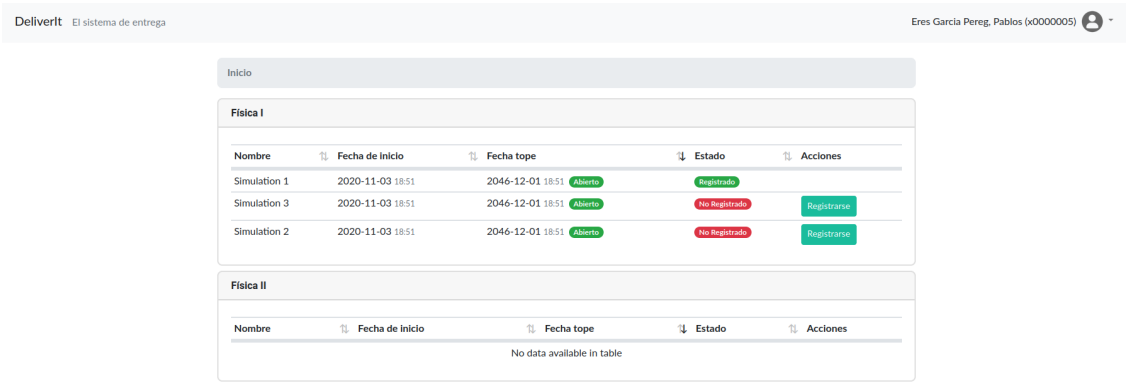
Figura 2.1: Vista de la lista de alumnos de una asignatura.

Dos últimos menús son accesibles desde el menú principal, quedando más habitualmente al alcance diario de los administrados. El menú de gestión de entor-

nos permite crear entornos de programación configurados para poder corregir las entregas de los alumnos en las prácticas según la construcción previa de la práctica. Por debajo, los entornos son contenedores aislados con sistemas independientes y portables que utilizan Docker. Finalmente, el menú con la cola de entregas permite ver el orden y los detalles de las entregas a nivel global que procesa Deliverit históricamente, en el momento actual y en espera de ser corregidas.

2.1.2. Aplicación de Alumno

La aplicación de Alumno da lugar a un menú principal con una lista de bloques, cada uno de ellos representando una asignatura en la que el alumno está registrado. Dentro de cada bloque el alumno puede ver las prácticas disponibles, e incluso registrarse en ellas. Dentro de la vista de la práctica, el alumno tiene información acerca de los detalles de la práctica y puede gestionar sus entregas y el estado de las mismas.



El diseño de esta aplicación es muy directo: el alumno sólo puede encargarse de gestionar su tarea de las prácticas. Pero no es cierto, pues puede acceder a los datos de su perfil e incluso cambiar su contraseña de la cuenta accediendo al menú desde el icono arriba a la derecha de la ventana.

2.2. Tecnologías

Las diferentes funcionalidades y procedimientos del sistema exigen un conjunto variado de tecnologías para su funcionamiento y mantenimiento. A continuación, se detallan las principales herramientas y lenguajes utilizados, junto con una explicación de su propósito y uso dentro del proyecto:

La más importante de ellas es Elixir. **Elixir** [1] es un lenguaje de programación funcional, concurrente y orientado a la construcción de sistemas distribuidos y tolerantes a fallos. Basado en la máquina virtual de Erlang (BEAM), Elixir es una gran opción para proyectos que buscan robustez, alta disponibilidad y poco uso de memoria. Se utiliza para la mayoría del código de las subaplicaciones de Deliverit, abarcando desde el apartado visual hasta las consultas a base de datos.

Las bibliotecas de Elixir utilizadas más importantes son tres. **Phoenix** [2] es un framework web que permite diseñar los componentes HTML y es ampliamente utilizada en el ecosistema de Elixir. **Ecto** [3] es la biblioteca oficial para la manipulación de bases de datos (BBDD). Permite definir las entidades, facilita migraciones e integra un sistema de mapeo objeto-relacional para simplificar el uso de BBDD. Por último, **Dockerex** [4] es una biblioteca cliente para interactuar con Docker desde Elixir. Ha sido empleada en este proyecto para gestionar contenedores para las prácticas internamente en la aplicación.

En el apartado de interfaces web también se utiliza **JavaScript** [5]. Es un lenguaje interpretado que permite acceder a un gran ecosistema de librerías relacionadas con el apartado visual y funcional web. Parte de los componentes visibles están escritos en este lenguaje, pero se está migrando de forma gradual a Phoenix para que el stack tecnológico de Deliverit sea más compacto y cercano entre sus partes.

La BD con la que se interactúa a través de Ecto es **PostgreSQL** [6]. Es uno de los sistemas de gestión de bases de datos relacionales más demandados para la gestión de guardado, lectura y escritura de datos estructurados. Además de ser de código abierto, la personalización y consultas complejas que dispone han hecho a PostgreSQL esencial para el manejo de datos en el proyecto.

Por otro lado la librería de Dockerex requiere de Docker. **Docker** [7] es un software de gestión de contenedores. Esto es, permite el aislamiento de entornos de ejecución en unidades llamadas contenedores sin necesidad de la dependencia de la infraestructura e incluso permitiendo hacer estos entornos portables para mayor conveniencia del despliegue. Aquí es dónde se despliegan los entornos de ejecución que permiten ejecutar los códigos de las prácticas entregados por los alumnos.

Finalmente está Git y GitLab como herramientas para la gestión de desarrollo de código. **Git** [8] es el sistema de control de versiones más utilizado en los proyectos modernos y **GitLab** [9] se encarga de mantener y gestionar el código y documentación de Deliverit como repositorio central para el acceso por el equipo.

2.3. Diseño y Arquitectura

Deliverit es un proyecto Umbrella [10] compuesto por un conjunto de aplicaciones internas y archivos de configuración. A continuación podemos ver la parte esencial de la estructura:

```
.
|-- apps
|   |-- admin
|   |-- components
|   |-- database
|   |-- logic
|   |- student
|-- config
```

```
|   |-- config.exs
|   |-- dev.exs
|   |-- prod.exs
|   |-- test.exs
|-- mix.exs
|-- mix.lock
```

Los ficheros `mix.exs` y `mix.lock` configuran las dependencias globales versionadas comunes a todas las aplicaciones internas. También existe una configuración global (en el directorio `config`) que contiene configuraciones específicas para el entorno de desarrollo, producción y testing mediante `dev.exs`, `prod.exs` y `test.exs` respectivamente. `config.exs` actúa como configuración global para cada uno de esos entornos.

Por parte del directorio apps tenemos las 5 aplicaciones internas que componen la aplicación: *admin*, *components*, *database*, *logic* y *student*. Cada una de ellas contiene su `mix.exs` y directorio `config` para dependencias y configuraciones específicas, un directorio `test` con módulos para testing y un directorio `lib` con el código fuente de esa misma aplicación interna.

```
logic
|-- config
|   |-- config.exs
|   |-- dev.exs
|   |-- prod.exs
|   |-- test.exs
|-- lib
|-- mix.exs
|-- test
```

2.3.1. Aplicaciones visuales

Las aplicaciones visuales (*admin*, *student* y *components*) crean componentes e interactúan con las aplicaciones lógicas para guardar y recibir datos y también saber cuando cambiar entre grupos de componentes visuales. *Admin* es la aplicación de administradores y profesores, *student* es la aplicación para estudiantes y *components* da soporte a ambas con algunos componentes adicionales.

```
lib
|-- admin
|   |-- application.ex
|-- admin_web
|   |-- controllers
|   |-- templates
|   |-- views
|-- admin_web.ex
```

Podemos ver el directorio `lib` de la app *admin* con sus contenidos más relevantes, muy similar también en la app *student* pero con sus contenidos y nomenclaturas adaptados. El subdirectorio `admin` contiene `application.ex`, que arranca

la aplicación y sus derivados. Por otro lado el subdirectorio `admin` (recordemos que para la app student se llamaría `student`) contiene el directorio `views` con las vistas de la aplicación, el directorio `controllers` con los módulos controlladores y el directorio `templates` con los diseños html que se renderizarán dentro de las vistas. El nombre de estos directorios sugieren la estructura de estas applicaciones de Phoenix, las cuales siguen la arquitectura modelo-vista-controlador.

2.3.2. Aplicaciones lógicas

Tanto *logic* como *database* son las aplicaciones que gestionan la lógica interna de las aplicaciones visuales y el acceso a la base de datos respectivamente. *database* es la app que se encarga del acceso mediante la librería Ecto a PostgreSQL y *logic* hace de intermediario entre los módulos controladores de las aplicaciones *admin* y *student*, y *database*.

```
lib
|-- logic
|   |-- application.ex
|   |-- corrections
|   |-- email
|   |-- environments
|   |-- exports.ex
|   |-- file
|   |-- file.ex
|   |-- groups
|   |-- projects
|   |-- students
|   |-- subjects
|   |-- submissions
|   |-- time.ex
|   |-- tools.ex
|-- logic.ex
```

```
lib
|-- database
|   |-- application.ex
|   |-- corrections
|   |-- corrections.ex
|   |-- custom
|   |-- envs
|   |-- envs.ex
|   |-- repo.ex
|   |-- students
|   |-- students.ex
|   |-- subjects
|   |-- subjects.ex
|-- database.ex
```

Los árboles anteriores muestran el interior de los directorios `lib` de *logic* y de *da-*

tabase respectivamente. Como se pueden ver, son similares. Ambos directorios cuentan con `logic.ex`/`database.ex`, el módulo de arranque de las apps.

Dentro del subdirectorio con el nombre de dicha aplicación existen una serie de directorios que a su vez contienen los módulos de la gestión de la lógica o de las consultas a la BD en el ámbito del nombre de cada directorio. Por ejemplo, el directorio `lib/database/corrections` contiene los módulos que hace consultas de lectura o escritura relacionadas con las correciones.

Por otro lado, los módulos sueltos del subdirectorio o bien contienen funciones de utilidad (en el caso de *logic*), o bien son módulos que representan entidades de la BD (en el caso de *database*).

2.3.3. Arquitectura

La arquitectura fue inicialmente diseñada por Andrés Mareca Mínguez [11], quedando dividida en tres capas distintas diseñadas de forma paralela a la estructura. Existen las capas *web*, *lógica* y *procesamiento*. La arquitectura queda representada en la figura 2.2.

La **capa web** consiste de las aplicaciones de administración y profesores, y de alumnos. Ambas webs comparten componentes creados en común debido a que se comparten ciertas vistas o se reciclan visualmente algunos componentes. Mencionar que a futuro la aplicación de administración y profesores está previsto o bien que se divida en dos webapps distintas o bien que se les de a administradores y profesores permisos y accesos distintos.

Esta primera capa se subdivide en dos subcapas: la subcapa visual, con los componentes, y la subcapa de controladores, la cual comunica la primera subcapa con la capa lógica.

La **capa lógica** se encarga de atender a las solicitudes de los controladores como si se tratase de los modelos de la arquitectura MVC. Esta capa se divide en una serie de ámbitos que abarcan un concepto como entidad dentro de Deliverit. Algunos ejemplos son emails, alumnos, grupos, correciones, etc. Algunas funciones dentro de los ámbitos requerirán comunicación con la capa de procesamiento de datos.

Finalmente la **capa de procesamiento** es la que se encarga de la comunicación con la base de datos y contiene los entornos de ejecución para las pruebas y correcciones de prácticas. Cada una de estas funciones la realizará una subcapa: la subcapa de base de datos y la subcapa de entornos de ejecución.

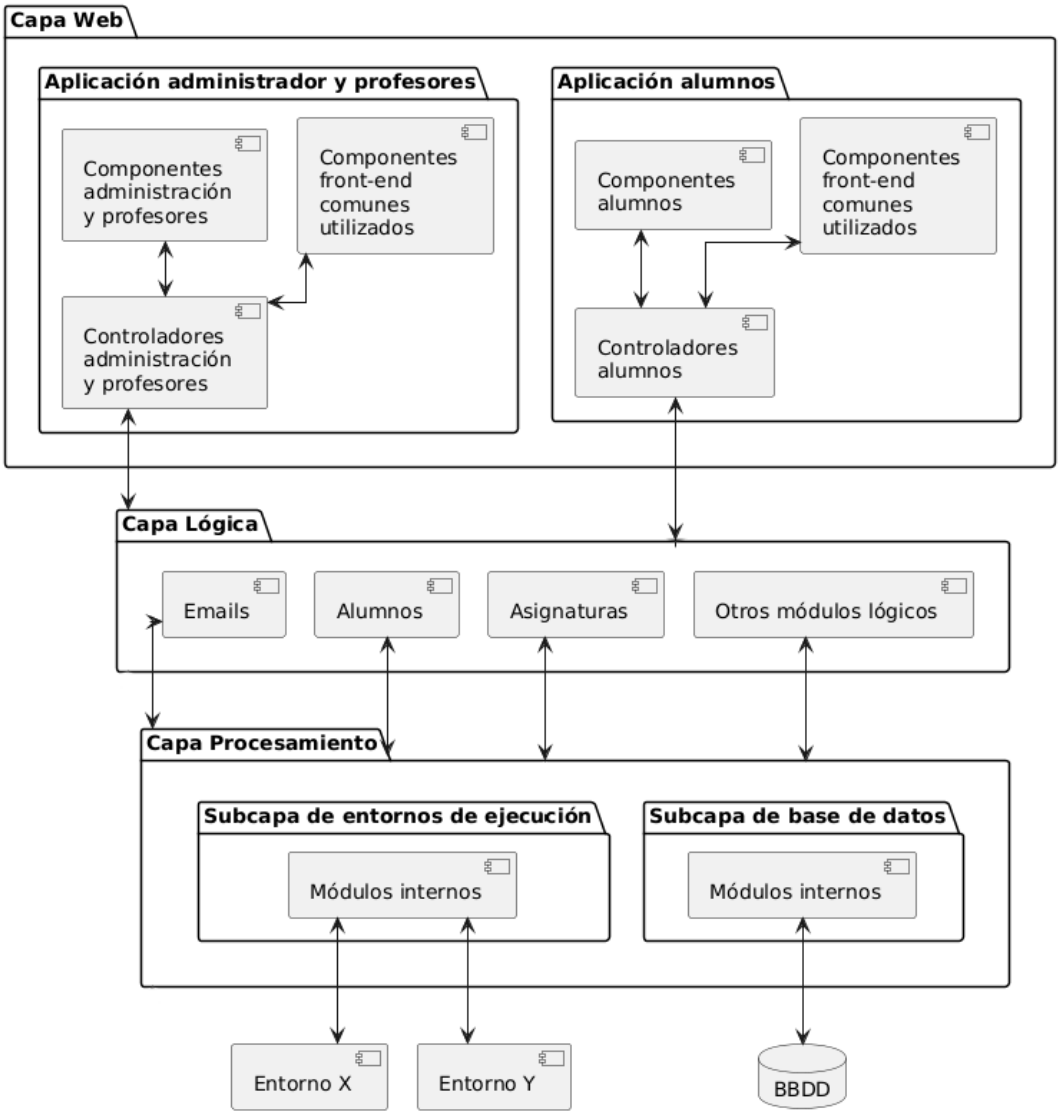


Figura 2.2: Arquitectura de Deliverit.

2.4. Evolución

En estos cinco años desde su nacimiento, Deliverit ya ha conseguido marcarse un recorrido existencial: es utilizado en varias asignaturas de programación de la ETSIINF, además de contar con un desarrollo y soporte continuados con miras de crecimiento en su utilización.

El desarrollo ha sido gracias al trabajo realizado y posteriormente recogido en Trabajos de Fin de Grado y Trabajos de Fin de Máster desde el 2020. Algunas de las aportaciones importantes han sido la primera implementación de Andrés Mareca [12]; y la monitorización y corrección de entregas tanto de Elías Herrero

[13] como de Nicola Carambia [14].

2.5. Problemas a Resolver

Las principales funcionalidades de Deliverit en torno a las prácticas y su gestión de alumnos están implementadas, pero durante el soporte en el ciclo de vida se pueden encontrar ciertas vulnerabilidades o comportamientos inesperados. Por ello, existe todavía un margen de mejora en el estado actual del software sobre el cual trabajar.

Además, se desea poder implementar pequeñas funcionalidades auxiliares que mejoren la experiencia de usuario para estudiantes y del profesorado. Un ejemplo de esto mismo es el acceso adicional al sistema mediante la API REST. Por ello, sobre el contenido del proyecto ya existente, se desea llegar a alcanzar los siguientes objetivos:

1. **Revisión y organización de las tareas del Backlog.** Revisión de todas las tareas del backlog, dispuestas en el Kanban de los repositorios de GitLab. Eliminar tareas duplicadas y fusionar tareas similares. Se tendrá que realizar un etiquetado apropiado de cada una, además de mejorar la redacción escrita de los cambios propuestos y su traducción al inglés.
2. **Admisión de ficheros de alumnos en latin1.** Codificar los cambios necesarios para admitir que un profesor pueda cargar de forma masiva alumnos mediante ficheros codificados en latin1. Actualmente sólo se puede cargar con ficheros en UTF-8, por lo que se tendrá que proceder a desarrollar una solución de conversión en Elixir entre formatos para una lectura correcta de los datos.
3. **Modificación del sistema de envío de emails.** Se buscará analizar, mejorar y expandir los módulos del proyecto relacionados con el envío de emails. Actualmente existen problemas visuales y se han planteado mejoras a futuro, por lo que se procederá a compartimentar y realizar mejoras funcionales y visuales al envío de correos.
4. **Mejoras Avanzadas de Emails.** Hay que investigar sobre algún mecanismo para limitar la cantidad de envío de emails del sistema. Entre las posibles opciones primero se buscará la posibilidad de implementar *rate limiting*, o en caso de imposibilidad habría que plantear otras alternativas.
5. **Documentación de la API con OpenAPI.** Documentación de la API de Deliverit mediante la especificación OpenAPI. Para ello, primero se recogerán todos endpoints de Deliverit y a continuación se redactarán siguiendo OpenAPI para su lectura por otros desarrolladores.

Capítulo 3

Funcionalidades

En este capítulo se describen los requisitos que se deben cumplir como tarea del Trabajo de Fin de Grado.

Se van a clasificar los requisitos en 4 categorías diferentes, de la siguiente manera:

- **Requisitos de Gestión del Backlog**: relacionados con la gestión de requisitos y arreglo de bugs en la lista de tareas pendientes dentro de GitLab, la plataforma web que gestiona el ciclo de vida del software.
- **Requisitos de Emails**: relacionados con cambios en el módulo de la aplicación encargado del envío de distintos tipos de emails a los estudiantes del sistema.
- **Requisitos de Fichero de Alumnos**: relacionados con la funcionalidad encargada de permitir añadir una cantidad de alumnos a través de un fichero de formato específico en codificación latin1.
- **Requisitos de API**: relacionados con la necesidad de la documentación de la API de Deliverit.

3.1. Requisitos de Gestión de Backlog

A continuación, se van a describir los distintos requisitos que se han de cumplir y que afectarán a la lista de tareas pendientes por realizar en Deliverit:

REQ-B1	Revisión de tareas definidas	**Importancia**	Baja
Descripción			
Hay que revisar una por una todas las tareas definidas en el Backlog de GitLab. La revisión de una tarea conlleva acciones como el cambio del título y la descripción por textos más clarificadores y precisos en caso necesario. También incluye la fusión de tareas, relación entre sí mediantes enlaces o creación de subtareas. En caso de que la tarea represente un bug o una propuesta, añadir su etiqueta correspondiente. Se deberá de comunicar a Ángel Herranz los tipos de tareas que se ajusten a una o varias de las siguientes categorías: ▪ Tareas duplicadas. ▪ Tareas obsoletas. Aquí entran tareas ya completadas, que no se ajustan a los objetivos de Deliverit o deprecadas. ▪ Tareas incorrectamente definidas. Se define por incorrecta una tarea cuya definición de objetivos no sea suficientemente precisa para su realización. El objetivo del requisito es acabar con un conjunto de tareas más definidas para facilitar su implementación por los desarrolladores.			

REQ-B2	Traducción al inglés de las tareas	**Importancia**	Media
Descripción			
Las tareas del backlog deben de ser traducidas al inglés. Ésto afecta a todos los componentes de una tarea: ▪ Título. ▪ Descripción. ▪ Comentarios adicionales de la tarea. ▪ Especificaciones adicionales del profesorado. ▪ Archivos y recursos adjuntos, en la medida que sea posible.			

3.2. Requisitos de Emails

A continuación, se van a describir los distintos requisitos que se han de cumplir y que afectarán al módulo interno de envío de emails:

REQ-E1	Revisar el UX de los tipos de emails ya implementados	**Importancia**	Baja
Descripción			
El objetivo de este requisito será revisar el UX de todos los tipos de emails actualmente implementados. Para ello, se comprobarán si el título y el cuerpo del email son lo suficientemente precisos y claros de cara al usuario. En caso contrario, modificar sus textos y datos pertinentes para poder cumplir esas metas.			

REQ-E2	Enviar un email cuando la contraseña de un alumno es cambiada	**Importancia**	Alta
Descripción			
Actualmente cuando la contraseña de un alumno es cambiada por una contraseña específica, ya sea por un alumno, profesor o administrador, el alumno recibe un email indicando que su contraseña fue reseteada (con una contraseña aleatoria). Se tendrá que crear un tipo de email específico cuyo contenido indique al alumno que su contraseña fue explícitamente cambiada incluyendo la nueva contraseña introducida y evitar que se envíe el email de contraseña cambiada. Adicionalmente se tendrá que asegurar que en ambos casos no se pueda enviar una contraseña vacía o inexistente.			

REQ-E3	Enviar un email cuando el perfil de un alumno es modificado	**Importancia**	Baja
Descripción			
Cuando un profesor o administrador cambia algún dato del perfil de un alumno, ese alumno recibirá un email informándole que los datos de su perfil han sido modificados.			

REQ-E4	Enviar un email cuando un alumno es dado de alta en una asignatura	**Importancia**	Media
Descripción			
Cuando se da de alta a un alumno en una asignatura, el alumno recibirá un email informándole de en qué asignatura se le ha dado de alta. El alta de alumnos de una asignatura se puede dar de dos formas distintas: el alta específica del alumno en la asignatura por un profesor o administrador, o si el alumno se ha añadido al sistema mediante la funcionalidad de carga masiva de alumnos (siempre que se haya especificado una asignatura en la misma).			

REQ-E5	Enviar un email cuando un alumno es dado de baja en una asignatura	**Importancia**	Baja
Descripción			
Cuando se da de baja a un alumno en una asignatura, el alumno recibirá un email informándole de en qué asignatura se le ha dado de baja. La baja de alumnos de una asignatura se puede dar de tres formas distintas: la baja específica del alumno por un profesor o administrador, la baja de todos los alumnos de la asignatura o la eliminación de la asignatura del sistema. Se deberán añadir tests para probar la funcionalidad del envío del email e implementarlo de tal forma que permita la traducción del contenido al idioma de la aplicación en ese momento.			

REQ-E6	Añadir limitación temporal a la cantidad simultánea de envío emails	**Importancia**	Alta
Descripción			
Actualmente el sistema de envío de emails de Deliverit no limita la cantidad de emails simultáneos que se pueden enviar. Tendrá que implementar una solución que permita introducir rate limiting, ralentizando la cantidad de emails que se pueden enviar por unidad de tiempo. La implementación será compacta y centralizada para permitir así que aplique a todos los tipos de emails.			

3.3. Requisitos de Fichero de Alumnos

A continuación, se van a describir los distintos requisitos que se han de cumplir y que permitirán aceptar un fichero de alumnos en latin1:

REQ-F1	Lectura del fichero en codificación latin1 [ISO-8859-1]	**Importancia**	Alta
Descripción			

El sistema deberá permitir aceptar y procesar ficheros con un formato específico para la carga masiva de alumnos en el sistema. El formato aceptado se define con la descripción que hay a continuación:

> El fichero deberá ser un CSV con tabuladores, con una línea de cabecera que deberá empezar por Expediente. Para cada alumno se proporcionará la siguiente información:
>
> Expediente: número de matrícula; DNI: DNI del alumno; Alumno: nombre completo como apellidos, nombre; Email Facultad: email del alumno.
>
> Se admiten más campos pero no se tendrán en cuenta.
>
> Si el alumno ya existe entonces no se añadirá pero se considera como una inserción correcta. Se admiten líneas antes de las cabeceras.
>
> Si una de estas líneas tiene la forma Asignatura: {código_de_asignatura}-{nombre_de_asignatura}, entonces se creará la asignatura (si no existe) y los alumnos se darán de alta en la misma.
>
> > Ejemplo:
> > Asignatura: 1234568-Programación I Expediente Alumno DNI Email Facultad x0000001 García García, Miguel 12345678A correo@alumnos.upm.com x0000002 Perez Perez, Pablo 12345675A correo1@alumnos.upm.com

El requisito precisa que no se sustituya la ya aceptada codificación UTF-8 por latin1. Ambas deben de ser aceptadas para considerar el requisito como completo.

REQ-F2	Reestructuración del código de lectura del fichero	**Importancia**	Baja
Descripción			
La reestructuración del codigo de lectura del fichero de alumnos busca documentar y estructurar el código para hacerlo más legible para otros desarrolladores. Hacerlo más modular permitirá poder añadir a futuro nuevos tipos de codificación de texto plano. Se deberá separar la acción de lectura del fichero del guardado de los alumnos en base de datos.			

3.4. Requisitos de API

A continuación, se van a describir los distintos requisitos que se han de cumplir y que permitirán documentar la API pública:

REQ-A1	Implementar la configuración de Open API Spex	**Importancia**	Alta
Descripción			
El objetivo del requisito se divide en dos partes. Primero, instalar la dependencia Open API Spex e implementar la configuración en Elixir de la misma. También incluye la creación de dos endpoints en la API, uno de ellos para la documentación de la API en texto plano y otro para la herramienta Swagger. Ambas formas de documentación generada seguirán el estándar OpenAPI. Segundo, estudiar la viabilidad final de la dependencia en el proyecto debido a su estructura como proyecto Umbrella. El resultado del estudio de la viabilidad será comentado con Ángel Herranz.			

REQ-A2	Escribir la documentación en el estándar OpenAPI	**Importancia**	Media
Descripción			
Se deberán documentar todos los endpoints públicos de la API del proyecto siguiendo el estándar OpenAPI. Para ello, documéntense los endpoints, entidades y formatos de respuesta requeridos de forma nativa en el código de Elixir.			

REQ-A3	Hacer accesible la documentación de forma pública	**Importancia**	Media
Descripción			
Un profesor o administrador el sistema deberá de ser capaz de acceder y visualizar la documentación de OpenAPI tanto en texto plano como en el formato de Swagger. El acceso se hará a través de dos endpoints distintos de la aplicación de administrador. La documentación mostrada quedará generada previamente antes del arranque de la aplicación.			

Todos los requisitos descritos en este capítulo deben ser implementados de for-

ma independiente entre sí para facilitar la revisión de cada uno de ellos antes de su posible integración en producción. No se establece un orden en el cual completarlos, si bien es cierto que se establece unos niveles de importancia de para el sistema.

Cabe destacar los requisitos REQ-E6 y REQ-F1, los cuales son los que aportan sustancialmente más mejoras en el sistema en términos de rendimiento y experiencia de usuario. Su impacto en la llegada de emails a los usuarios y la carga masiva de alumnos al sistema pueden suponer una reducción del trabajo total si mejoran esas funcionalidades.

Capítulo 4

Desarrollo

En este capítulo se desarrollarán los requisitos definidos en el Capítulo 3, desde su diseño hasta su implementación, y las tecnologías implicadas en los cambios.

4.1. Tecnologías utilizadas

- **Elixir:**
  Elixir [1] es un lenguaje de programación funcional rápido y robusto con un fuerte desarrollo en el apartado de concurrencia. Como principal lenguaje de programación de Deliverit en la lógica interna, será utilizado para implementar los cambios propuestos al sistema.

- **Bamboo:**
  Bamboo [15] permite la construcción, gestión y envío de correos electrónicos mediante SMTP a los usuarios del sistema. Esta biblioteca de Elixir ya está siendo utilizada en Deliverit, por lo que la utilizaremos para expandir y mejorar la funcionalidad de emails.

- **Open API Spex:**
  La librería Open API Spex [16] de Elixir permite a los proyectos configurar y documentar sus APIs para generar documentación y acceder a ésta. Permite generar documentación siguiendo el estándar OpenAPI.

- **Ecto y PostgreSQL:**
  Ecto [3] es una biblioteca de Elixir para la manipulación de datos en bases de datos con funcionalidades como la creación de consultas personalizadas o mapeo objeto-relacional (ORM), que se utilizará con el gestor de bases de datos relacionales (DBMS) PostgreSQL.

4.2. Cambios en los tipos de emails

4.2.1. Contexto

La funcionalidad de emails está gestionada por la app logic mediante el uso de la librería Bamboo. Se compone de 5 partes: los módulos *Logic.Email.View*, *Logic.Email.Mailer*, *Logic.Email.Emails* y *Logic*, y los emails templates del directorio `apps/logic/lib/logic/email/templates/email`.

```
lib
|-- logic
|   |-- email
|   |   |-- emails.ex
|   |   |-- mailer.ex
|   |   |-- templates
|   |   |   |-- email
|   |   |       |-- group_email.html.eex
|   |   |       |-- group_email.text.eex
|   |   |       |-- individual_email.html.eex
|   |   |       |-- individual_email.text.eex
|   |   |       |-- password_reset_email.html.eex
|   |   |       |-- password_reset_email.text.eex
|   |   |       |-- registration_email.html.eex
|   |   |       |-- registration_email.text.eex
|   |   |       |-- subject_unenroll_email.html.eex
|   |   |       |-- subject_unenroll_email.text.eex
|   |   |       |-- submission_email.html.eex
|   |   |       |-- submission_email.text.eex
|   |   |-- view.ex
|-- logic.ex
```

Cuando un módulo de la lógica quiere enviar un email, llama a la función de *Logic.Email.Emails* cuyo nombre coincide con el tipo deseado a enviar y se le pasan los datos necesarios para crear el email. Este email se rellenará con los datos otorgados y con la plantilla del mismo nombre (la del fichero acabado en `.html.eex` es para emails enriquecidos y `.text.eex` son emails de texto plano). Después el email se enviará al invocar la función *deliver_later*/1 del módulo *Logic.Email.Mailer*. El objetivo de los dos módulos restantes será el renderizado visual del email gracias a Phoenix.

Se tienen que añadir 4 tipos más de emails (figura 4.1): al dar de alta y de baja un alumno en una asignatura, al cambiar datos del perfil del alumno y cuando la contraseña de acceso del alumno sea cambiada manualmente por otra contraseña distinta. El desarrollo de cada uno de estos tipos se realizará de forma independiente en ramas distintas de Git para su revisión por separado antes de pasar a producción.

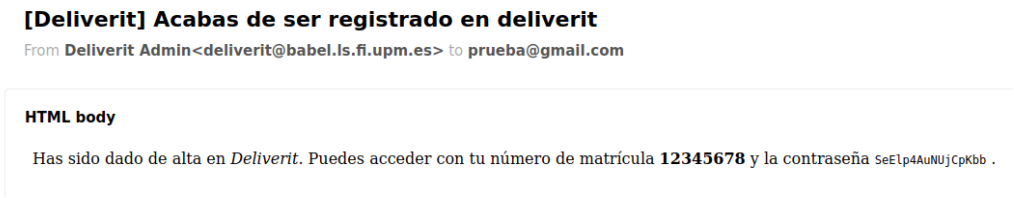


Figura 4.1: Ejemplo de email de alumno registrado en Deliverit.

4.2.2. Diseño

Para añadir el **email de alta de alumno en asignatura** se crea un nuevo tipo de email, sus dos templates y se lanzará tras la acción del módulo lógico del alta del alumno. Los ficheros nuevos/editados son:

Dentro del subdirectorio la aplicación logic (`lib/logic`):

- *`Logic.Email.Emails`*
- Nuevo template enriquecido, `subject_enroll_email.html.eex`
- Nuevo template plano, `subject_enroll_email.text.eex`
- *`Logic.Students.Enroll`*
  (ubicado en: `students/enroll.ex`)

En el módulo *`Emails`* se añade la función *`subject_enroll_email/2`* para crear el email si no ocurren errores durante el alta del alumno.

Para añadir el **email de baja de alumno en asignatura** se crea un nuevo tipo de email y sus dos templates. También debido a que las acciones para eliminar uno o varios alumnos de una asignatura a la app de database se están realizando desde los controladores *`AdminWeb.EnrolledController`* y *`AdminWeb.SubjectController`*, habrá que modificar éstos y crear dos nuevos módulos en la lógica. Estos dos nuevos módulos serán *`Logic.Students.Unenroll`* y *`Logic.Students.UnenrollBulk`* para desacoplar los controladores de la base de la lógica de la subcapa de base de datos. Los ficheros nuevos/editados son:

Dentro del subdirectorio la aplicación logic (`lib/logic`):

- *`Logic.Email.Emails`*
- Nuevo template enriquecido, `subject_unenroll_email.html.eex`
- Nuevo template plano, `subject_unenroll_email.text.eex`
- *`Logic.Students.Unenroll`*
  (ubicado en: `students/unenroll.ex`)
- *`Logic.Students.UnenrollBulk`*
  (ubicado en: `students/unenroll_bulk.ex`)

Unenroll contará con una función *exec/2* con el id del estudiante y de la asignatura para realizar las transacciones de base de datos a través de la app database para comprobar que ambos existen y así poder dar de baja al alumno. *UnenrollBulk* recibirá únicamente el id de la asignatura en *exec/1* para realizar las transacciones para dar de baja a todos los alumnos de la asignatura. En ambos casos, y si no ha habido errores en el proceso, se creará y enviará el email por cada alumno dado de baja. Para poder crear el email hay que añadir la función *subject_enroll_email/2* en el módulo *Emails* y construir el email.

Para añadir el **email de cambio de datos del alumno** se crea un nuevo tipo de email, sus dos templates y se lanzará tras la acción del módulo lógico de actualizar el alumno. Los ficheros nuevos/editados son:

Dentro del subdirectorio la aplicación logic (`lib/logic`):

- *Logic.Email.Emails*
- Nuevo template enriquecido, `profile_changed_email.html.eex`
- Nuevo template plano, `profile_changed_email.text.eex`
- *Logic.Students.Update*
  (ubicado en: `students/update.ex`)

En el módulo *Emails* se añade la función *profile_changed_email/1* para crear el email si no ocurren errores durante la actualización de los datos del alumno.

Para añadir el **email de cambio de contraseña** se crea un nuevo tipo de email, sus dos templates y se lanzará tras la acción del módulo lógico de actualizar el alumno también, pero cualquier cambio no afectará al tipo de email anterior pues se desarrolla en un entorno independiente para poder desarrollar en paralelo. Los ficheros nuevos/editados son:

Dentro del subdirectorio la aplicación logic (`lib/logic`):

- *Logic.Email.Emails*
- Nuevo template enriquecido, `password_change_email.html.eex`
- Nuevo template plano, `password_change_email.text.eex`
- *Logic.Students.Update*
  (ubicado en: `students/update.ex`)

En el módulo *Emails* se añade la función *password_change_email/1* para crear el email si no ocurren errores durante la actualización de la contraseña del alumno. También se envía el email cuando se actualizan los datos del alumnos pero el email de la contraseña no se enviara en alguna actualización previa (revisando que el atributo booleano del alumno llamado *password_sent* estuviera asignado como falso).

4.2.3. Implementación

En la **implementación del email de alta de un alumno** se añadió la llamada a `subject_enroll_email/2` tras valorar el resultado `result` de las múltiples transacciones realizadas a través de `Database.multi/3`.

Este es el cambio realizado dentro de la función `exec/2` de `Logic.Students.Enroll` para registrar al alumno en una asignatura en BBDD.

```
case result do
  {
    :ok,
    %{create_enroll: %{student: student, subject: subject}}
  } ->
      try do
        %Bamboo.Email{} =
          Emails.subject_enroll_email(student, subject)
          |> Mailer.deliver_later()

        {:ok, student, subject}
      rescue
        error ->
          Logger.error
            (
              "Error sending an email: #{inspect(error)}"
            )
          {:ok, student, subject}
      end

  {:error, _, error} ->
      {:error, error}
```

Se valora que si ninguna de las transacciones ha ido mal, cogerá el estudiante que ha sido enrolado y su asignatura, crea el tipo deseado de email con los datos del estudiante y la propia asignatura y lo envía en cuanto `Mailer` esté disponible con la función `deliver_later/1`. Si algo va mal durante el proceso de email se considerará que el proceso de alta del alumno seguirá siendo exitoso pero el `Logger` escribirá el error. Si alguna de las transacciones fue mal, directamente no se envía el email y la función `exec/2` devolverá el error con la razón por la que falló.

Por otro lado, se introdujo una cláusula `try ... rescue` en la transacción `Students.create_enrolled/1` para capturar los errores relacionados con la creación del enrolamiento en BBDD. Ésto permite controlarlos mejor y devolverlos como una tupla con un mensaje reconocible y evitar que se propage a otras partes de la ejecución. En caso de no haber ningún error, en ésta transacción se devuelve el resultado esperado.

```
try do
    {:ok, _enrolled} = Students.create_enrolled(attrs)
```

```
    {:ok, %{student: student, subject: subject}}
rescue
    e in Ecto.ConstraintError ->
        if e.constraint == "enrolled_pkey" do
          {:error, :student_already_enrolled}
        else
          Logger.error
            (
              "Unhandled constraint error: #{inspect(e)}"
            )
          {:error, :constraint_error}
        end
    end
```

Nuestro email se crea en la invocación de `subject_enroll_email/2` del módulo `Emails`. Este tipo de email se dirigirá al email del estudiante con la traducción del título al idioma del usuario gracias a `gettext/1`.

```
def subject_enroll_email(student, subject) do
  base_email()
  |> to(student.email)
  |> subject
       (gettext
          (
            "[Deliverit] You have been enrolled in a subject"
          )
       )
  |> render(:subject_enroll_email, subject: subject)
end
```

El cuerpo del email lo renderizaremos con la plantilla indicada que comienza por `subject_enroll_email`:

```
<% import ComponentsWeb.Gettext %>
<p>
  <%= Phoenix.HTML.raw(
        gettext
          (
            "You have been enrolled in the subject
                <b>%{subject}</b> <b>(%{code})</b>.",
            subject: @subject.name,
            code: @subject.code
          )
        ) %>
</p>
```

o bien:

```
<% import ComponentsWeb.Gettext %>
```

```
<%= gettext
      (
        "You have been enrolled in the the subject %{subject}
            (%{code}).",
        subject: @subject.name,
        code: @subject.code
      ) %>
```

El cuerpo se renderizará en formato enriquecido o plano en el email con la traducción del texto informando que se dió de alta al usuario en una asignatura, especificando con los datos pasados al cuerpo del email (nombre y código de la asignatura).

En la **implementación del email de baja de un alumno o todos los alumnos de la asignatura** se implementaron tanto `Logic.Students.Unenroll` como `Logic.Students.UnenrollBulk`.

En primer lugar tenemos `Logic.Students.Unenroll`. La función `exec/2`, a la cual se le pasa el id del estudiante a dar de baja y el id de la asignatura objetivo, es la que se invocará cuando se quiera dar a un estudiante en específico. Se compone principalmente de una ejecución de múltiples transacciones mediante `Database.multi/4` cuyo resultado se recoge en el parámetro `result`. Si en alguna transacción se produce un error el resto de transacciones se revertirán y se guardará en el parámetro una tupla con `:error` por primer elemento.

Las transacciones se ejecutan en un orden:

1. Se revisa que el estudiante exista.
2. Se revisa que la asignatura exista.
3. Se revisa que el alumno esté previamente dado de alta.
4. El alumno se intenta dar de baja en la asignatura.

```
defmodule Logic.Students.Unenroll do
  ...

  def exec(student_id, subject_id) do
    result =
      Database.multi(
        check_student: fn _ ->
          case Students.get_student(student_id) do
            nil -> {:error, :student_not_found}
            student -> {:ok, student}
          end
        end,
        check_subject: fn _ ->
          case Subjects.get_subject(subject_id) do
            nil -> {:error, :subject_not_found}
            subject -> {:ok, subject}
```

```
        end
      end,
      check_enrollment: fn _ ->
        case Students.get_enrolled(student_id, subject_id) do
          nil -> {:error, :enrollment_not_found}
          enrolled -> {:ok, enrolled}
        end
      end,
      delete_enroll: fn %{check_enrollment: enrolled} ->
        Students.delete_enrolled(enrolled)
      end
    )
  ...
```

Después de la multi transacción se evalúa su resultado. En caso de error se devuelve junto al mensaje propio. Pero en caso de que haya sido exitosa pasa a ejecutar el código del email.

```
case result do
  {:ok, %{check_student: student, check_subject: subject}} ->
    try do
      %Bamboo.Email{} =
        Emails.subject_unenroll_email(student, subject)
        |> Mailer.deliver_later()

      {:ok, student, subject}
    rescue
      error ->
        Logger.error
          (
            "Error sending an email: #{inspect(error)}"
          )
        {:ok, student, subject}
    end

  {:error, _, error} ->
    {:error, error}
end
```

Nuestro email se crea en la invocación de `subject_unenroll_email/2` del módulo `Emails`. Este tipo de email se dirigirá al email del estudiante con la traducción del título al idioma del usuario gracias a `gettext/1`.

```
def subject_unenroll_email(student, subject) do
  base_email()
  |> to(student.email)
  |> subject
      (
        gettext
```

```
            (
              "[Deliverit] You have been unenrolled from a subject"
            )
        )
  |> render(:subject_unenroll_email, subject: subject)
end
```

El cuerpo del email lo renderizaremos con la plantilla indicada que comienza por `subject_unenroll_email`:

```
<% import ComponentsWeb.Gettext %>
<p>
  <%= Phoenix.HTML.raw(
        (
          gettext
            (
              "You have been unenrolled from the subject
                <b>%{subject}</b> <b>(%{code})</b>.",
              subject: @subject.name,
              code: @subject.code
            )
        ) %>
</p>
```

o bien:

```
<% import ComponentsWeb.Gettext %>
<%= gettext
      (
        "You have been unenrolled from the the subject
          %{subject} (%{code}).",
        subject: @subject.name,
        code: @subject.code
      ) %>
```

El cuerpo se renderizará en formato enriquecido o plano en el email con la traducción del texto informando que se dió de baja al usuario en una asignatura, especificando con los datos pasados al cuerpo del email (nombre y código de la asignatura).

En segundo lugar tenemos `Logic.Students.UnenrollBulk`. La función `exec/1`, a la cual se le pasa el id de la asignatura de la que se quieran dar de baja todos los alumnos, es la que se invocará cuando se quiera realizar esta acción. Se compone principalmente de una ejecución de múltiples transacciones mediante `Database.multi/3` cuyo resultado se recoge en el parámetro `result`. Si en alguna transacción se produce un error el resto de transacciones se revertirán y se guardará en el parámetro una tupla con `:error` por primer elemento.

Las transacciones se ejecutan en un orden:

1. Se revisa que la asignatura exista.
2. Se revisa que los alumnos estén previamente dados de alta.
3. Para los alumnos dados de alta, se les intenta dar de baja en la asignatura.

```
defmodule Logic.Students.UnenrollBulk do
  ...

  def exec(subject_id) do
    result =
      Database.multi(
        check_subject: fn _ ->
          case Subjects.get_subject(subject_id) do
            nil -> {:error, :subject_not_found}
            subject -> {:ok, subject}
          end
        end,
        check_enrollments: fn _ ->
          subject_enrollments = Students.list_enrolled
                                  (
                                    subject_id
                                  )

          if Enum.empty?(subject_enrollments) do
            {:error, :no_students_enrolled}
          else
            {:ok, subject_enrollments}
          end
        end,
        unenroll_students:
          fn %{check_enrollments: subject_enrollments} ->
            unenroll_results =
              Enum.map(subject_enrollments, fn enrollment ->
                case Students.get_enrolled
                       (
                         enrollment.student_id,
                         subject_id
                       ) do
                  nil -> {:error, :student_not_enrolled}
                  enrollment ->
                    Students.delete_enrolled(enrollment)
                end
              end)

            if Enum.any?
                 (
                   unenroll_results,
                   &match?({:error, _}, &1)
```

```
                ) do
              {:error, :failed_to_unenroll_some_students}
            else
              {:ok, subject_enrollments}
            end
          end
        )
    ...
```

Después de la multi transacción se evalúa su resultado. En caso de error se devuelve junto al mensaje propio. Pero en caso de que haya sido exitosa pasa a ejecutar el código del email (se crea el mismo tipo de email que el de una baja de alumno única con *subject_unenroll_email/2*) para los alumnos que se consiguiron dar de baja.

```
case result do
    {
      :ok,
      %{
         check_subject: subject,
         unenroll_students: subject_enrollments
       }
    } ->
        try do
          Enum.each(subject_enrollments, fn enrollment ->
            student = Students.get_student(enrollment.student_id)

            if student do
              %Bamboo.Email{} =
                Emails.subject_unenroll_email(student, subject)
                |> Mailer.deliver_later()
            end
          end)

          {:ok, :all_students_unenrolled}
        rescue
          error ->
            Logger.error
              (
                "Error sending an email: #{inspect(error)}"
              )
            {:ok, :all_students_unenrolled}
        end

    {:error, _, error} ->
        {:error, error}
end
```

La integración de estos nuevos módulos lógicos en los controladores ocurre en 3 instancias de dos controladores distintos.

Para la eliminación de un único estudiante, realizada con la función *AdminWeb.EnrolledController.delete/2*, sustituimos la comprobación de estudiante más su eliminación por la ejecución de la lógica de *Unenroll*. El resultado podrá tratarse con un mensaje de éxito o mostrando el error que ha ocurrido durante el proceso de baja del alumno.

```
case Unenroll.exec(student_id, subject_id) do
  {:ok, _, subject} ->
    conn
    |> put_flash
         (
           :info,
           gettext("Student successfully withdrawn")
         )
    |> redirect(to: Routes.subject_path(conn, :show, subject.id))

   {:error, type} when is_atom(type) ->
        error_msg = Atom.to_string(type)
                    |> String.replace("_", " ")
                    |> String.capitalize()

    conn
    |> put_flash(:error, error_msg)
    |> put_view(AdminWeb.ErrorView)
    |> render("404.html", msg: error_msg)
end
```

Se puede ir más allá y dar de baja a todos los estudiantes con la función *AdminWeb.EnrolledController.delete_all/1*. Una ejecución similar, sustituyendo esta vez la comprobación y eliminación de estudiantes por la lógica de *UnenrollBulk*, reduce el acoplamiento en el diseño. También se maneja el resultado de la lógica de forma parecida.

```
case UnenrollBulk.exec(subject_id) do
  {:ok, _} ->
    conn
    |> put_flash
         (
           :info,
           gettext("All students successfully withdrawn")
         )
    |> redirect
         (to:
           Routes.subject_path
             (
               conn,
```

```
                :show,
                subject_id
              ) <> "#students"
          )

    {:error, type} when is_atom(type) ->
         error_msg = Atom.to_string(type)
                     |> String.replace("_", " ")
                     |> String.capitalize()

     conn
     |> put_flash(:error, error_msg)
     |> put_view(AdminWeb.ErrorView)
     |> render("404.html", msg: error_msg)
end
```

Por último la eliminación de una asignatura también obliga a dar de baja todos los alumnos. El cambio está al final de la función *AdminWeb.SubjectController.delete/1*, la mejorada gestión del resultado y la sustitución de la eliminación anterior cambiada por la realizada en *UnenrollBulk*.

```
subject ->
case UnenrollBulk.exec(subject.id) do
  {:ok, _} ->
    {:ok, _subject} = Subjects.delete_subject(subject)

    conn
    |> put_flash(:info, gettext("Subject successfully deleted."))
    |> redirect(to: Routes.subject_path(conn, :index))

    {:error, type} when is_atom(type) ->
      error_msg = Atom.to_string(type)
      |> String.replace("_", " ")
      |> String.capitalize()

    conn
    |> put_flash(:error, error_msg)
    |> put_view(AdminWeb.ErrorView)
    |> render("404.html", msg: error_msg)
  end
end
```

En la **implementación del email de cambio de datos de alumno** se añadió la llamada a *profile_changed_email/1* tras valorar el resultado *result* de las múltiples transacciones realizadas a través de *Database.multi/3*. Este es el cambio realizado dentro de la función *exec/2* de *Logic.Students.Update*, que actualiza al alumno en BBDD con sus nuevos valores.

```
case result do
```

```
  {:ok, %{update_student: updated_student}} ->
    try do
      %Bamboo.Email{} =
        Emails.profile_changed_email(updated_student)
        |> Mailer.deliver_later()

      {:ok, updated_student}
    rescue
      error ->
        Logger.error("Error sending an email: #{inspect(error)}")
        {:ok, updated_student}
    end

  {:error, _, error} ->
    {:error, error}
```

Se valora que si ninguna de las transacciones ha ido mal, cogerá el estudiante que ha sido actualizado, crea el tipo deseado de email con los datos del estudiante y lo envía en cuanto `Mailer` esté disponible con la función `deliver_later/1`. Si algo va mal durante el proceso de email se considerará que el proceso de cambio de datos seguirá siendo exitoso pero el `Logger` escribirá el error. Si alguna de las transacciones fue mal, directamente no se envía el email y la función `exec/2` devolverá el error con la razón por la que falló.

```
def profile_changed_email(student) do
  base_email()
  |> to(student.email)
  |> subject
       (gettext
          (
            "[Deliverit] Your profile data has been changed"
          )
        )
  |> render(:profile_changed_email)
end
```

Nuestro email se crea en la invocación de `profile_changed_email/1` del módulo `Emails`. Este tipo de email se dirigirá al email del estudiante con la traducción del título al idioma del usuario gracias a `gettext/1`. El cuerpo del email lo renderizaremos con la plantilla indicada que comienza por `profile_changed_email`:

```
<% import ComponentsWeb.Gettext %>
<p>
  <%= Phoenix.HTML.raw
        (
          gettext(
            "Your profile data in Deliverit has been changed."
          )
```

```
        ) %>
</p>
```

o bien:

```
<% import ComponentsWeb.Gettext %>
<%= gettext
      (
        "Your profile data in Deliverit has been changed."
      ) %>
```

El cuerpo se renderizará en formato enriquecido o plano en el email con la traducción del texto informando que se cambió los datos del usuario.

En la **implementación de cambio de contraseña de un alumno** se añadió la llamada a `password_change_email/2` tras valorar el resultado `result` de las múltiples transacciones realizadas a través de `Database.multi/3`.

Este es el cambio realizado dentro de la función `exec/2` de `Logic.Students.Update`, que actualiza al alumno en BBDD pero también tiene la capacidad de actualizar simplemente la contraseña si se le pasa una nueva.

```
case result do
  {
    :ok,
    %{
      update_student:
        %{
          updated_student: updated_student,
          password: password
         }
     }
  } ->
      if updated_student.password_sent do
        {:ok, updated_student}
      else
        try do
          %Bamboo.Email{} =
            Emails.password_change_email
              (
                updated_student,
                password
              )
            |> Mailer.deliver_later()

          Students.update_student
            (
              updated_student,
              %{password_sent: true}
            )
```

```
          {:ok, updated_student}
        rescue
          error ->
            Logger.error
              (
                "Error sending an email: #{inspect(error)}"
              )
            {:ok, updated_student}
        end
      end

  {:error, _, error} ->
      {:error, error}
```

Se valora que si ninguna de las transacciones ha ido mal, cogerá el estudiante al que se le ha cambiado la contraseña y la contraseña, crea el tipo deseado de email con ambos datos y lo envía en cuanto `Mailer` esté disponible con la función `deliver_later/1`. Si algo va mal durante el proceso de email se considerará que el proceso de cambio de datos seguirá siendo exitoso pero el `Logger` escribirá el error. Si alguna de las transacciones fue mal, directamente no se envía el email y la función `exec/2` devolverá el error con la razón por la que falló.

Por otro lado, se ha actualizado la transacción que actualiza los datos del usuario para revisar si la contraseña nueva no es vacía o nula. En caso de ser vacía, la función `Tools.clean_map/1` eliminará el campo de la contraseña vacía de los datos a actualizar. Si hubiera una nueva contraseña, la hasheamos por temas de seguridad con `BCrypt.hash_pwd_salt/1`.

La contraseña hasheada se guardará en la base de datos tras la ejecución de `Students.update_student/2`. Si el alumno se ha actualizado sin errores, devolveremos al estudiante actualizado y su contraseña en texto plano (para poder enviarla en el email). En caso de haber un error devolvemos una tupla para notificarlo a la función que haya llamado a `exec/2` para cambiar la contraseña.

```
update_student: fn %{check_student: student} ->
  attrs =
    case Map.get(params, :password) do
      nil ->
        params

      password when password == "" ->
        params
        |> Map.put(:password, nil)

      password ->
        params
        |> Map.put(:password, Bcrypt.hash_pwd_salt(password))
        |> Map.put(:password_sent, false)
```

```
    end
    |> Tools.clean_map()

    case Students.update_student(student, attrs) do
      {:ok, updated_student} ->
        {
          :ok,
          %{
             updated_student: updated_student,
             password: Map.get(params, :password)
           }
        }

      {:error, reason} ->
        {:error, :update_student, reason}
    end
```

Nuestro email se crea en la invocación de `password_change_email/2` del módulo `Emails`. Este tipo de email se dirigirá al email del estudiante con la traducción del título al idioma del usuario gracias a `gettext/1`.

```
def password_change_email(student, password) do
  base_email()
  |> to(student.email)
  |> subject
      (
        gettext
          (
            "[Deliverit] Your password has been changed"
          )
      )
  |> render
       (
         :password_change_email,
         student: student,
         password: password
       )
end
```

El cuerpo del email lo renderizaremos con la plantilla indicada que comienza por `password_change_email`:

```
<% import ComponentsWeb.Gettext %>
<p>
  <%= Phoenix.HTML.raw
        (
          gettext
            (
              "Your password in <em>Deliverit</em> has been
```

```
                changed. You can now access using your
                registration number <b>%{student_number}</b>
                and the password <code>%{password}</code> .",
              student_number: @student.registration_number,
              password: @password
            )
        ) %>
</p>
```

o bien:

```
<% import ComponentsWeb.Gettext %>
<%= gettext
      (
        "Your password in Deliverit has been changed.
           You can now access using your registration number
           %{student_number} and the password
           %{password} .",
        student_number: @student.registration_number,
        password: @password
      ) %>
```

El cuerpo se renderizará en formato enriquecido o plano en el email con la traducción del texto informando que se cambió la contraseña del usuario, especificando con los datos pasados al cuerpo del email (número de registro del usuario y su nueva contraseña).

4.3. Limitación de cantidad de emails

4.3.1. Contexto

El sistema de envío de emails, anteriormente explicado en el apartado 4.2.1, necesita de comunicación externa para entregar los emails a los usuarios. Deliverit está configurado para utilizar Relay UPM [17], un servicio de relay de la universidad para los proyectos que ocurren en la misma.

Relay es un servidor SMTP que actúa como intermediario entre el sistema de envío y los usuarios. El servidor utiliza software como MySQL, Postfix o Mailman sobre Debian 9. Dentro de la aplicación se ha configurado el fichero de configuración `apps/logic/config/prod.exs` para que la librería Bamboo utilice la dirección IP, puerto y credenciales de un fichero de configuración externo para enviar esos correos durante producción.

```
import Config

delivery_data_dir = System.get_env("SUBMISSIONS_PATH")
  || "/srv/deliverit"

File.mkdir_p(delivery_data_dir)
```

```
config :logic,
  path: delivery_data_dir

config :logic, Logic.Email.Mailer,
  adapter: Bamboo.SMTPAdapter,
  server: System.get_env("SMTP_SERVER"),
  port: (System.get_env("SMTP_PORT") || "587")
    |> String.to_integer(),
  username: System.get_env("SMTP_USER"),
  password: System.get_env("SMTP_PASSWORD"),
  tls: :always,
  ssl: false,
  retries: 1,
  auth: :if_available
```

El inconveniente encontrado es que la funcionalidad de emails puede llegar a generar una gran cantidad de emails en un período corto de tiempo (por ejemplo, si se registran en el sistema 5.000 alumnos de forma simultánea) y provocar una saturación en Relay. Estos casos eventuales imponen restricciones desde Relay como la limitación a una cantidad de correos electrónicos por unidad de tiempo.

Como esta medida supone un riesgo potencial de la pérdida de algunos emails enviados, se ha decidido implementar de forma nativa un sistema de limitación de envío guardando temporalmente los emails en espera.

4.3.2. Diseño

La limitación de emails, o *rate limiting*, se ha realizado mediante la implementación de un GenServer con una cola de emails. Un GenServer o Generic Server es una interfaz nativa de Elixir que permite de forma sencilla gestionar un proceso de forma paralela al proceso principal. Define una serie de funciones a implementar o utilizar el estado o las acciones internas del GenServer [18]. En términos simples se pueden considerar como un mini servidor interno.

El módulo que se encargaba previamente de enviar los emails es `Logic.Email.Mailer`. Bien, se buscará transformarlo en un GenServer con la cola de mensajes pendientes de enviar. Contará con la función `init/1`, utilizada para inicializar el GenServer pasando una lista vacía como estado inicial de la cola.

La otra función públicamente accesible será `send_email/1`, pasando una tupla compuesta por el tipo del email y un mapa con los argumentos necesarios para crear el email. Estos datos del email se guardarán de forma asíncrona en la cola de emails gracias a una función asíncrona llamada `handle_cast/2` a la cual se invocará desde `send_email/1`.

Otras dos funciones, en este caso ambas privadas, gestionarán el proceso de envío: `schedule_send/0` y `send_email_now/1`. `schedule_send/0` programará la invocación a una función asíncrona de nombre `handle_info/2` de forma regular. La función asíncrona a su vez llamará a `send_email_now/1` tantas veces

como sea necesario en función de la cantidad máxima definida de emails máximo a enviar en ese intervalo, con los datos de los emails al inicio de la cola y borrándolos. `send_email_now/1` tratará de construir el email según del tipo que sea cada uno e intentará enviarlo.

Este diseño nos permite reducir la responsabilidad de aquellos módulos que antes creaban los emails y ordenaban enviarlos a simplemente dejar en espera los emails para ser construidos y enviados. Por tanto modificaremos ligeramente estos módulos para adaptarlos al nuevo sistema.

4.3.3. Implementación

La implementación del GenServer cambia totalmente el código previo de `Logic.Email.Mailer`. Tenemos dos valores especiales. `@rate_limit` es la cantidad de emails que se pueden enviar como máximo en el intervalo `@interval` de microsegundos.

Al inicializar el servidor empezamos a programar el primer envío y dejamos constancia que tenemos una lista vacía de datos de emails (pasada por argumento a `init/1`) y su longitud es 0.

Si algún módulo encarga al `Mailer` el envío de unos datos de email, usará `send_email/1`. El tipo y los datos para construir el email se pasan con un átomo `:send_email` como etiqueta a `GenServer.cast/2`. Es una función interna del GenServer que llama a nuestro `handle_cast/2` para recibir los nuevos datos y el estado actual del mailer.

La función asíncrona no devuelve una respuesta pero añade al final de la cola los datos para el email. El valor numérico de emails pendientes por enviar no se actualiza aquí, pero sí más adelante.

```
@rate_limit 2      # Maximum number of emails per interval
@interval 5000     # Time interval in milliseconds

def init(state) do
  schedule_send()
  {:ok, {state, 0}}
end

def send_email(email) do
  GenServer.cast(__MODULE__, {:send_email, email})
end

def handle_cast({:send_email, email}, {queue, sent}) do
  {:noreply, {queue ++ [email], sent}}
end
```

La llamada a intervalos se debe a `schedule_send/0`, que le dice a este proceso (el mailer) que invoque `handle_info/2` con el átomo `:send_emails` pasado el intervalo de tiempo.

Nuestra otra función asíncrona se encarga de extraer los datos de los emails que puede enviar en ese intervalo de tiempo y manda la construcción y el envío de cada uno de ellos gracias a `send_email_now/1`. Después, borrará los mensajes extraídos de la cola y actualiza la cuenta de emails pendientes.

La última fase del email ocurre en `send_email_now/1`. De los datos del email, se revisa si el tipo del email es uno de los que tiene listados e intenta construirlo. En caso de no construirse el email o ser un tipo desconocido se ignorará el envío, pero en caso de tener ya formalmente un email lo envía en el momento.

```
defp schedule_send() do
  Process.send_after(self(), :send_emails, @interval)
end

def handle_info(:send_emails, {queue, sent}) do
  emails_to_send = Enum.take(queue, @rate_limit - sent)
  Enum.each(emails_to_send, &send_email_now/1)

  remaining_queue = Enum.drop(queue, length(emails_to_send))
  schedule_send()
  {:noreply, {remaining_queue, length(emails_to_send)}}
end

defp send_email_now({email_type, args}) do
  case email_type do
    :registration -> Emails.registration_email
      (
        args.student, args.password
      )
    :password_reset -> Emails.password_reset_email
      (
        args.student, args.password
      )
    :project_enroll -> Emails.group_email
      (
        args.students, args.subject, args.project
      )
    :submission -> Emails.submission_email
      (
        args.students, args.project
      )
    _ -> nil
  end
  |> case do
    nil -> :unknown_email_type
    email -> deliver_now(email)
  end
end
```

Del módulo `Mailer` queda mencionar la función de arranque `start_link/1`, que es llamada desde el arranque de la app logic para arrancar el Genserver e inicializar el estado para hacer operativo el sistema de emails.

```
def start_link(_) do
  GenServer.start_link(__MODULE__, [], name: __MODULE__)
end
```

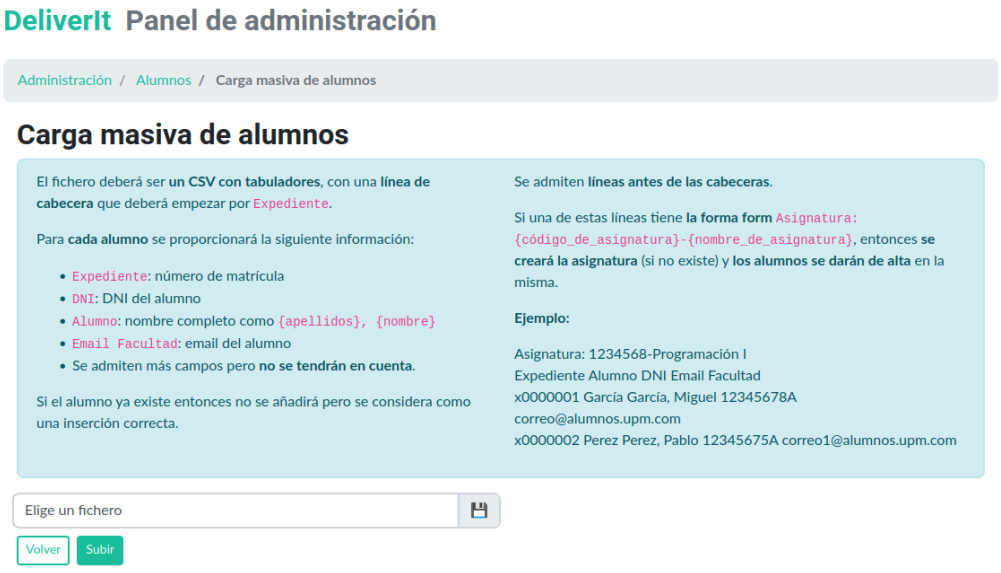
Acabamos el desarrollo de los cambios adaptando el envío de los emails preexistentes de Deliverit al nuevo formato del mailer. En los 4 módulos de la lógica que se envía un tipo de email el proceso es idéntico. Si antes actuaban en dos pasos, crear el tipo de email deseado más enviar después, ahora simplemente pasarán los datos al `Logic.Emails.Mailer`. A continuación, se muestra la nueva forma de querer mandar un email de registro:

```
Mailer.send_email
  (
    {
      :registration,
      %{student: student, password: params.password}
    }
  )
```

4.4. Cambios funcionalidad de fichero de alumnos

4.4.1. Contexto

La carga masiva de alumnos a través del fichero de alumnos es una funcionalidad de la aplicación Admin. Permite añadir cualquier cantidad de alumnos al sistema (y opcionalmente a una asignatura). La vista en la cual se puede subir el fichero a Deliverit para la carga de alumnos se puede ver a continuación:



En el estado inicial, se permite la subida de un fichero CSV al sistema siguiendo el formato expuesto en el cuadro azul. Sin embargo dado que ni se solicita al usuario una codificación específica del CSV ni el propio sistema la restringe, pueden ocurrir errores. En esta funcionalidad sólo se pueden llegar a procesar exitosamente ficheros con codificación UTF-8 debido a que esa es la codificación base que utiliza Elixir para la lectura de ficheros del sistema.

4.4.2. Diseño

Para poder realizar los objetivos relacionados con el fichero de alumno, se necesitará implementar la codificación latin1 [ISO-8859-1] en la parte correspondiente de la lógica responsable de la lectura del mismo fichero. También se tiene que reestructurar el código para desacoplarlo. Actualmente todo el código se encuentra dentro del módulo *AdminWeb.StudentBulkController* del fichero `student_bulk_controller.ex` de la aplicación admin. Este módulo es un controlador y contiene secciones que podrían destinarse a un módulo nuevo en la capa lógica.

A continuación se lista los ficheros manipulados para implementar la solución:

Dentro del subdirectorio la aplicación admin (`lib/admin_web`):

- *AdminWeb.StudentBulkController*
  (ubicado en: `controllers/student_bulk_controller.ex`)

Dentro del subdirectorio la aplicación logic (`lib/logic`):

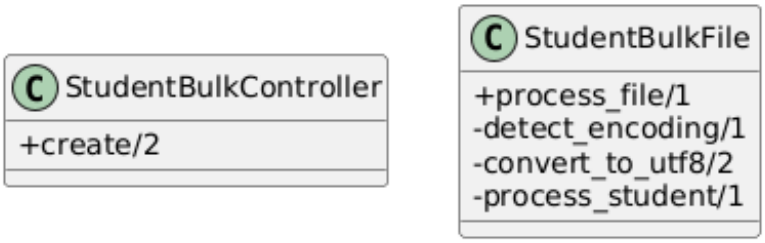
- *Logic.File.StudentBulkFile*
  (ubicado en: `file/student_bulk_file.ex`)

El módulo *AdminWeb.StudentBulkController* contiene dos funciones utilizadas para el procesamiento del fichero. *create/2* se divide internamente en varias secciones lógicas para el procesamiento:

1. Validación del nombre del fichero
2. Lectura del fichero
3. Identificación de asignatura
4. Identificación y creación de alumnos de forma iterativa
5. Recuperación o creación de la asignatura (si se identificó)
6. Registro de los alumnos en la asignatura en *bulk* (en forma de lista de alumnos)
7. Retorno con la actualización de la página del navegador

La otra función, *process_student/1* recibe un mapa clave-valor de los atributos leídos del alumno en el fichero y retorna un mapa clave-valor con los atributos procesados y necesarios para crear un alumno.

Ahora bien, se plantea el nuevo diseño:



El módulo `Logic.File.StudentBulkFile` contendría `process_file/1`, que realizaría la lectura e indentificación de alumnos y asignatura. La lectura detectará la codificación del fichero y la transformaría a codificación UTF-8 con la ayuda de `detect_encoding/1` y `convert_to_utf8/2` en el caso de que se tratara de latin1. Después, la detección de patrones para indentificar las entidades contaría con el traslado de la función `process_student/1` del controlador al módulo lógico.

Así el trabajo de `create/2` sería validar y llamar al procesamiento del fichero, y tras invocar a las funciones correspondientes de la aplicación database para guardar los alumnos y la asignatura (en caso de ser nueva), devolver el resultado de forma visual al usuario.

4.4.3. Implementación

Para empezar a implementar toda la lógica de lectura del fichero se empieza recibiendo la ruta del fichero para acceder al contenido.

```
def process_file(file_path) do
  file_path = Path.expand(file_path, __DIR__)

  case File.read(file_path) do
    {:ok, file_binary} ->
      utf8_binary = convert_to_utf8
                    (
                      file_binary,
                      detect_encoding(file_binary)
                    )
      stream = utf8_binary
               |> String.split("\n")
               |> Stream.map(&String.trim/1)
```

Utilizando `File.read/1` en contraposición al previamente usado `File.stream/1` nos permite leer el fichero como binario en lugar de UTF-8, donde se producían errores para leer caracteres si la codificación no era ésta. Después, se detecta la codificación del binario y se transforma a una secuencia en UTF-8 desde la codificación detectada. Ahora ya se puede formatear el nuevo stream de caracteres UTF-8 para poder identificar los alumnos y la asignatura.

```
defp detect_encoding(binary) do
  if String.valid?(binary) do
```

```
    "UTF-8"
  else
    "ISO-8859-1"
  end
end

defp convert_to_utf8(binary, encoding) do
  case encoding do
    "UTF-8" -> binary
    "ISO-8859-1" -> :unicode.characters_to_binary
                      (
                        binary,
                        :latin1
                      )
    _ -> {:error, "Unsupported encoding"}
  end
end
```

La codificación se detecta en `detect_encoding/1` comprobando que el binario leído cuenta con la codificación base que utiliza Elixir, UTF-8. Podemos asumir para la implementación que si no es válida será latin1, puesto que en `convert_to_utf8/2` controla la transformación de los caracteres en función de la codificación correcta. Si era previamente un fichero UTF-8, no hará nada; si la codificación detectada y la real son latin1, podrá convertir de latin1 a UTF-8; pero si finalmente estaba anotada la notificación detectada como latin1 y no lo era, el posible error llega al controlador con mensaje sobre que la codificación no es soportada.

El reconocimiento de los datos del fichero para los alumnos y asignatura se realiza mediante los patrones definidos como normas a la hora de subir el fichero en la aplicación admin.

```
metainfo = Enum.take_while
             (
               stream,
               &(!String.starts_with?(&1, "Expediente"))
             )

subject =
    Enum.find_value(metainfo, fn line ->
        re = ~r/Asignatura: (?<code>\w+)-(?<name>.*)$/

    case Regex.named_captures(re, line) do
      %{"code" => code, "name" => name} ->
         %{code: code, name: name}
      _ -> false
    end
  end)
```

```
students =
  stream
  |> Stream.drop(length(metainfo))
  |> CSV.decode
       (
         headers: true,
         separator: ?\t,
         validate_row_length: false
       )
  |> Enum.map(fn {:ok, entry} -> entry end)
  |> Enum.filter(
    &match?(
      %{
        "Expediente" => _,
        "Alumno" => _,
        "DNI" => _,
        "Email Facultad" => _
      },
      &1
    )
  )
  |> Enum.map(&process_student/1)
```

Este código ha sido trasladado de *AdminWeb.StudentBulkController* con el resto del código ya explicado en *StudentBulkFile* Primero trata de buscar el patrón correspondiente a la asignatura. Si lo encuentra con los datos construirá la información necesaria para crear una asignatura y descarta esos caracteres del stream, pero si no encuentra nada ignora este proceso. Con la secuencia de caracteres restantes buscará todas las posibles estructuras que representen alumnos, extraerá los datos para que sean procesados por la función *process_student/1* y guardará en una lista el resultado.

```
{:ok, students, subject}

{:error, reason} ->
    {:error, reason}
```

Si todo ha ido bien, el controlador recibirá un ok junto a los datos encontrados de la asignatura y los alumnos, o un error en el caso contrario.

Tras todo ésto, el *StudentBulkController* pierde toda la responsabilidad de lectura e identificación. La llamada a la función *process_file/1* del módulo lógico se realiza de la siguiente forma:

```
def create(conn, %{"request_params" => request_params}) do
    changeset = RequestParams.from(request_params)

    if changeset.valid? do
      params = Params.data(changeset)
```

```
      case Logic.File.StudentBulkFile.process_file
             (params.students.path) do
        {:ok, students, subject} ->
```

create/2 sigue validando la ruta del fichero dado, pero después pasa directamente su ruta por argumento a la función para externalizar esa lectura. Tras el proceso de lectura continuará con el trabajo habitual: o bien la lectura fue exitosa y procede a crear estudiantes y asignatura (si existiesen sus datos), o bien procesa el error recibido. Al final de cualquiera de estos dos procesos la página se actualizará con los mensajes de error o se abrirá la vista de la lista de alumnos del sistema.

4.5. Documentación de la API

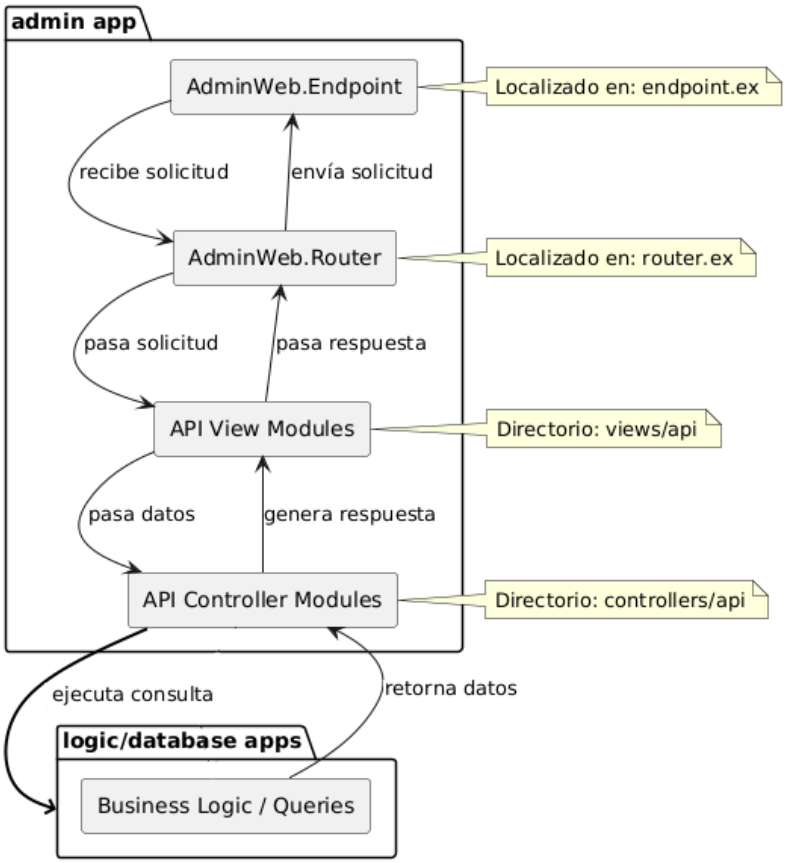
4.5.1. Contexto

Deliverit adicionalmente de su acceso mediante navegador web a la aplicación de Admin tiene un acceso mediante API. Esta API fue propuesta e implementada por Lars-Åke Fredlund, la cual permite la ejecución de peticiones HTTP al sistema sin necesidad de utilizar la GUI.

```
lib
|-- admin
    |-- admin_web
        |-- controllers
        |   |-- api
        |       |-- enrolled_controller.ex
        |       |-- group_controller.ex
        |       |-- handle_error.ex
        |       |-- project_controller.ex
        |       |-- student_controller.ex
        |       |-- subject_controller.ex
        |       |-- submission_controller.ex
        |-- endpoint.ex
        |-- router.ex
        |-- views
            |-- api
                |-- enrolled_view.ex
                |-- error_view.ex
                |-- group_view.ex
                |-- project_view.ex
                |-- student_view.ex
                |-- subject_view.ex
                |-- submission_view.ex
                |-- view_helpers.ex
```

Todos esos módulos se encuentran en la app admin. El módulo *AdminWeb*.

Endpoint localizado en el fichero `endpoint.ex` recibe las solicitudes que pasan a las vistas del directorio `views/api`. Posteriormente las vistas pasan las solicitudes a los controladores del directorio `controllers/api` a través de `router.ex` (cuyo módulo se llama *AdminWeb.Router*). Los controladores ejecutan internamente la consulta y se recorre el camino inverso para enviar la respuesta al usuario.



El siguiente fragmento del módulo *AdminWeb.Router* muestra todas las posibles consultas que se pueden hacer, incluyendo por supuesto en la URI la solicitud como segmento `/api`.

```
scope "/api", AdminWeb.Api do
  pipe_through(:api)

  resources
    (
      "/students",
      StudentController,
      except: [:delete, :edit]
    )
  get
    (
      "/students/registration_id/:id",
      StudentController,
```

```
    :get_by_registration_id
  )
get
  (
    "/students/:id/password_reset",
    StudentController,
    :password_reset
  )

resources
  (
    "/students/:student_id/enrolled",
    EnrolledController,
    only: [:index, :create]
  )

resources
  (
    "/projects",
    ProjectController,
    only: [:index, :show]
  )

resources("/groups", GroupController, only: [:show])

resources
  (
    "/subjects",
    SubjectController,
    except: [:delete, :edit]
  )
get
  (
    "/subjects/code_id/:id",
    SubjectController,
    :get_by_code_id
  )

resources
  (
    "/submissions",
    SubmissionController,
    only: [:show]
  )
get
  (
    "/submissions/:id/code",
```

```
      SubmissionController,
      :get_code
    )
  get("/submissions/:id/run", SubmissionController, :run)
end
```

El problema es que no hay ningún acesso posible a algún tipo de documentación de forma pública para permitir explorar los posibles endpoints, parámetros y otros datos de la API.

4.5.2. Diseño

Se eligió la librería de Elixir Open API Spex debido a que permite la creación de documentación tanto en texto plano como la visualización a través de la herramienta visual de documentación de APIs Swagger con documentación integrada en el código. Para poder implementar la documentación siguiendo el estándar OpenAPI, se deben de seguir 3 pasos:

1. Añadir la dependencia en el/los ficheros de documentación correspondientes.
2. Configurar la aplicación para producir la documentación y hacerla accesible de forma externa.
3. Escribir la documentación de los endpoints a documentar.

Entonces se tendrá que añadir la dependencia de Open API Spex al fichero de configuración de la app admin, puesto que allí es donde se encuentran los controladores de las consultas de la API.

La configuración necesaria concierne dos aspectos. Primero se tendrá que crear un módulo que implemente el comportamiento de OpenAPI para que en la generación de la documentación la librería pueda navegar dinámicamente los endpoints y leer la documentación escrita. Para ello el módulo a crear será *AdminWeb.ApiSpec* (creado en la ruta `apps/admin/lib/admin_web/api_spec.ex`). Segundo, hay que crear dos endpoints nuevos para acceder a ambos tipos de documentación.

El `/api/openapi` se creará para la generación de la documentación en texto plano. El endpoint `/swaggerui` se creará para leer la documentación en texto plano y generar la documentación en Swagger.

Finalmente, por cada endpoint a documentar se analizarán los verbos, parámetros de entrada y resultado y se escribirá la documentación siguiendo los ejemplos de la documentación oficial de Open API Spex [19].

4.5.3. Implementación

Para añadir la dependencia, hay que añadir la declaración de la dependencia en el módulo *Admin.MixProject*, que corresponde al fichero de configuración de dependencias `mix.exs` de la app admin. La declaración *:open_api_spex* añade

la versión 3.21.2 de la librería para el proyecto dentro de la función `deps/1`. Esta función es una función privada especial del módulo de configuración para leer y descargar la librería externa en tiempo de compilación del proyecto.

La implementación `AdminWeb.ApiSpec` fue la siguiente:

```
defmodule AdminWeb.ApiSpec do
  alias OpenApiSpex.{Info, OpenApi, Paths, Server}
  alias AdminWeb.{Endpoint, Router}
  @behaviour OpenApi

  @impl OpenApi
  def spec do
    %OpenApi{
      servers: [
        Server.from_endpoint(Endpoint)
      ],
      info: %Info{
        title: "Deliverit",
        version: Application.spec(:admin, :vsn)
      },
      paths: Paths.from_router(Router)
    }
    |> OpenApiSpex.resolve_schema_modules()
  end
end
```

La función `spec/1` es anotada por `@impl OpenAPI` para indicar a la biblioteca que éste es el método a ejecutar para generar la documentación. Se configura como punto de entrada HTTP el módulo `AdminWeb.Endpoint` y las rutas derivadas las lee del módulo `AdminWeb.Router`. Tras la configuración realizada procede a descubrir dinámicamente sobre esos endpoints todos los esquemas de entidades y esquemas de endpoints (siendo un esquema la documentación para la API).

La creación de los endpoints se hizo en sus scopes, que actúan como grupos de endpoints que eligen el endpoint invocado.

```
  pipeline :api do
  ...
  plug OpenApiSpex.Plug.PutApiSpec, module: AdminWeb.ApiSpec
end

scope "/", AdminWeb do
  pipe_through(:browser)
  ...
  get
    "/swaggerui",
    OpenApiSpex.Plug.SwaggerUI,
    path: "/api/openapi"
end
```

```
scope "/api", AdminWeb.Api do
  pipe_through(:api)
  ...
  get("/openapi", OpenApiController, :spec)
end
```

El endpoint de documentación de texto plano se crea en el scope `/api` invocando a un controlador a implementar a futuro. No es casualidad que se utilice este scope, pues obliga a pasar por un pipeline que añade la configuración necesaria con un plug para enlazar el endpoint con el módulo *AdminWeb.ApiSpec*. Ese endpoint ahora puede buscar y generar la documentación a mostrar. En el caso del endpoint de Swagger, se añade en el scope base `/` y se configura para indicar a Open API Spex con un plug la generación de la documentación Swagger con la otra documentación generada por el primer endpoint implementado.

Se ha probado también la generación práctica de la documentación con esquemas de ejemplo y no se ha conseguido poder visualizar en el navegador. Resulta que Open API Spex no está familiarizada con proyectos Umbrella como Deliverit que están compuestos por aplicaciones internas.

Mis conclusiones son que la librería no tiene forma de reconocer desde la raíz del proyecto la configuración necesaria para poder generar y mostrar la documentación que esté descrita. Tras comprobar la viabilidad actual de la documentación, se informó al tutor para así poder investigar a futuro una posible forma de hacer compatible la generación de documentación en un proyecto Umbrella.

Capítulo 5

Conclusiones y trabajo futuro

En este capítulo veremos el progreso realizado en Deliverit y el impacto del desarrollo acontecido.

5.1. Conclusiones

Internamente el proyecto cuenta con una estructura grande debido a un enfoque de dividir todo su funcionamiento en distintas aplicaciones. Para añadir nuevos cambios se requiere posiblemente de una integración a todos los niveles de su arquitectura. El uso de Elixir incrementa también la dificultad de procesamiento de todo el contexto al no ser un lenguaje tan popular como otros que existen en el mercado. Sin embargo, el respaldo de la profunda documentación del desarrollo de Deliverit y la comunicación con los miembros del equipo ayudan a aprender a navegar por los directorios y enterder la comunicación entre los distintos componentes.

Respecto a los objetivos iniciales y también las funcionalidades detalladas en el Capítulo 3 se ha conseguido abordar su mayoría. Se ha conseguido realizar una revisión exhaustiva del Backlog, actualizando meses previos de nuevos requisitos e ideas propuestas. Ésto, junto a la traducción del mismo, abre la puerta a la cooperación de desarrollo con personas de habla no española en la descripción, implementación y mejora de Deliverit. El resto de funcionalidades a implementar están pendientes de revisión y consideración de ser añadidas al código en producción del proyecto por los gestores del mismo.

Por un lado se ha conseguido la adición de los nuevos tipos de emails definidos de cara a los usuarios. Cada uno de ellos ha sido testeado a nivel funcional con test unitarios en su rama de Git independiente en preproducción. Si tras su revisión pasan a producción, en el futuro su texto interno deberá de ser revisado para añadir las traducciones al resto de idiomas deseados. Pero ésto no supone un gran esfuerzo de trabajo pues debido al desarrollo independiente se podrá ir realizando de manera modular y progresiva.

El sistema de limitación de mensajería de emails también ha quedado desarrollado. Los tipos de emails actualmente incluidos en producción son capaces de

esperar temporalmente para no saturar el envío de emails de Deliverit a través del servidor SMTP de la facultad, permitiendo ajustar antes de cualquier despliegue los parámetros para definir cuántos emails se envían por período de tiempo. Ayudará a hacer más fiable la llegada de emails a los alumnos, pero teniendo en cuenta que los tipos de emails nuevos tendrán que ser adaptados a la nueva lógica de envío retardado en el futuro basándose en la nueva forma que utilizan los tipos de emails que existían previamente.

El problema de la admisión de ficheros de alumnos en latin1 ha sido arreglado. En pruebas con ficheros grandes y pequeños tanto en latin1 como en UTF-8 se ha comprobado que el funcionamiento era el correcto y esperado. Si el arreglo del bug es aceptado, el profesorado podrá volver a utilizar ambos formatos de fichero.

Por último no se ha podido completar la documentación de la API de Deliverit con OpenAPI. Se ha debido a problemas de ejecución de la documentación al no estar preparada la biblioteca utilizada con el software de Umbrella que se utiliza. Sin embargo, la configuración necesaria sí fue implementada por lo que posibles avances futuros no necesitarán realizar todo el proceso desde el inicio.

5.2. Objetivos futuros

Cabe mencionar que existen otras ambiciones del proyecto al estar en constante evolución, que no me han sido asignadas pero están pendientes de continuar en el desarrollo interno. Por dar constancia de unos cuantos cambios a realizar próximamente tenemos el arreglo continuo de bugs derivado del soporte en producción, la novedad de reevaluación de entregas en las prácticas abiertas o una aportación personal de implementar un posible sistema de ajuste de preferencia de emails por cada alumno.

5.3. Análisis de impacto

Los nuevos cambios realizados y futuros objetivos planteados además de tener un fuerte impacto en una educación de calidad e innovación del sector educativo y en el resto de sectores que precisan de él para mejoras en los procesos productivos, permitiendo un acercamiento a los Objetivos de Desarrollo Sostenible (ODS) [20]. Se debe a que se brinda una mayor resiliencia, experiencia de alumnado y profesorado, y una mejora en escalabilidad que hacen más viable al proyecto.

Consecuentemente permite que Deliverit actúe como una herramienta innovadora y simplificadora en áreas de la educación como el aprendizaje práctico en el desarrollo académico. Y por ejemplo brinda a proyectos similares de campos STEM nuevos procesos de referencia con los cuales conseguir destinar más tiempo a una formación más enfocada en permitir desarrollar el proceso creativo de los futuros trabajadores.

5.4. Valoración personal

Deliverit como sistema de gestión de prácticas es una gran iniciativa. He abarcado el proyecto desde fuera, como usuario del sistema en mi 2º y 3º año de carrera, y desde dentro, desarrollando mejoras para su aplicación en el campo educativo. La sencillez de su interfaz, su diseño funcional cercano a lo esperado en la gestión de prácticas y la automatización del proceso de creación, gestión y correción de prácticas lo hacen un magnífico candidato a ser una herramienta ampliamente utilizada en el sector académico. Para eso está la tecnología: mejorar procesos para mejorar la calidad de vida de sus usuarios.

Bibliografía

[1] Elixir Core Team, *Elixir Programming Language*, Sitio web oficial, 2011. dirección: https://elixir-lang.org/.

[2] Phoenix Core Team, *Phoenix Framework*, Sitio web oficial, 2014. dirección: https://www.phoenixframework.org/.

[3] Ecto Maintainers, *Ecto*, Repositorio en GitHub, 2013. dirección: https://github.com/elixir-ecto/ecto.

[4] Hexedpackets, *Dockerex*, Repositorio en GitHub, 2015. dirección: https://github.com/hexedpackets/dockerex.

[5] «MDN Web Docs | JavaScript». (2024), dirección: https://developer.mozilla.org/es/docs/Web/JavaScript.

[6] PostgreSQL Global Development Group, *PostgreSQL*, Sitio web oficial, 1989. dirección: https://www.postgresql.org/.

[7] Docker Inc., *Docker*, Sitio web oficial, 2013. dirección: https://www.docker.com/.

[8] Git Project Contributors, *Git*, Sitio web oficial, 2005. dirección: https://git-scm.com/.

[9] GitLab Inc., *GitLab*, Sitio web oficial, 2011. dirección: https://about.gitlab.com/.

[10] Elixir Core Team, *Umbrella Projects*. dirección: https://hexdocs.pm/elixir/dependencies-and-umbrella-projects.html#don-t-drink-the-kool-aid.

[11] A. Mareca Mínguez. «Sistema de entrega Deliverit: prototipo». (2020), dirección: https://oa.upm.es/63096/.

[12] A. Mareca Mínguez. «Plataforma para el desarrollo, publicación y seguimiento de retos de programación.» (2022), dirección: https://oa.upm.es/70705/.

[13] E. Herrero Lázaro. «Monitorización de entregas en Deliverit». (2023), dirección: https://oa.upm.es/75054/.

[14] N. Carambia Corbella. «DeliverIt: extensión para la corrección de prácticas». (2024), dirección: https://oa.upm.es/82497/.

[15] B. Community, *Bamboo*, Repositorio en GitHub, 2019. dirección: https://github.com/beam-community/bamboo.

[16] Equipo desarrollador y colaboradores del proyecto Open API Spex, *Open API Spex documentation writing examples*, Repositorio en GitHub, 2019. dirección: https://github.com/open-api-spex/open_api_spex.

[17] ETS de Ingenieros Informáticos - Universidad Politénica de Madrid, *Relay UPM*, Especificaciones técnicas de Relay, 2021. dirección: https://www.fi.upm.es/?pagina=809.

[18] Elixir School, *Concurrencia en OTP*, 2019. dirección: https://goo.su/r6UowCJ.

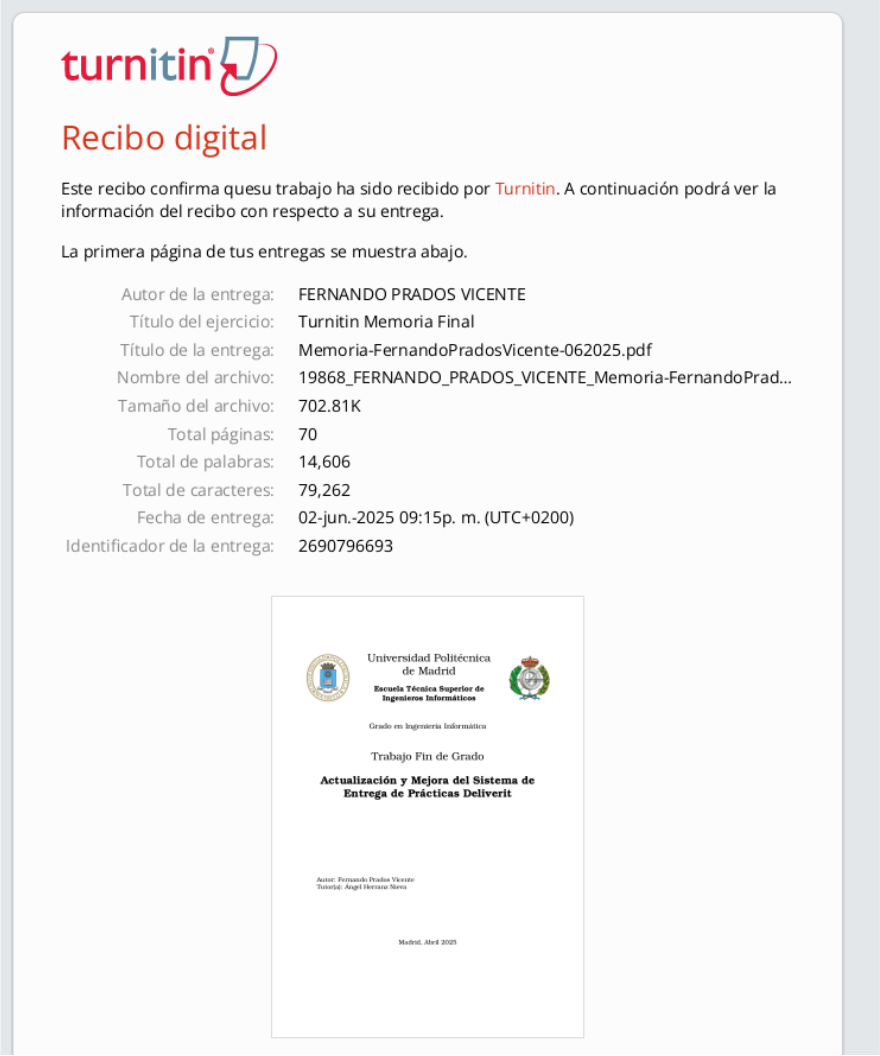
[19] Equipo desarrollador y colaboradores del proyecto Open API Spex, *Open API Spex documentation writing examples*. dirección: https://github.com/open-api-spex/open_api_spex/blob/master/lib/open_api_spex.ex.

[20] United Nations, *Sustainable Development Goals*. dirección: https://www.un.org/sustainabledevelopment/sustainable-development-goals/.

Anexos

Apéndice A

Turnitin



turnitin

Recibo digital

Este recibo confirma quesu trabajo ha sido recibido por Turnitin. A continuación podrá ver la información del recibo con respecto a su entrega.

La primera página de tus entregas se muestra abajo.

Autor de la entrega:	FERNANDO PRADOS VICENTE
Título del ejercicio:	Turnitin Memoria Final
Título de la entrega:	Memoria-FernandoPradosVicente-062025.pdf
Nombre del archivo:	19868_FERNANDO_PRADOS_VICENTE_Memoria-FernandoPrad...
Tamaño del archivo:	702.81K
Total páginas:	70
Total de palabras:	14,606
Total de caracteres:	79,262
Fecha de entrega:	02-jun.-2025 09:15p. m. (UTC+0200)
Identificador de la entrega:	2690796693

Este documento esta firmado por

	**Firmante**	CN=tfgm.fi.upm.es, OU=CCFI, O=ETS Ingenieros Informaticos - UPM, C=ES
	Fecha/Hora	Tue Jun 03 16:32:44 CEST 2025
	Emisor del Certificado	EMAILADDRESS=camanager@etsiinf.upm.es, CN=CA ETS Ingenieros Informaticos, O=ETS Ingenieros Informaticos - UPM, C=ES
	Numero de Serie	561
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)