Facultad de Ciencias

PULSESPAM: APLICACIÓN PARA GESTIÓN DE ENCUESTAS

(PulseSpam: A survey management app)

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Irene Zamanillo Zubizarreta

Director: Diego García Saiz

Co-Director: Guillermo Ménguez Álvarez

Julio - 2023

Resumen

Este Trabajo de Fin de Grado aborda el proceso de desarrollo de una aplicación de escritorio y una aplicación web, que permiten valorar la satisfacción de los miembros de una organización a través de la realización de una pregunta diaria. Las respuestas a dichas preguntas se computan y se muestran a los administradores de la empresa en una aplicación web tipo *dashboard*, permitiendo controlar los resultados a través de distintas gráficas y filtros, así como realizar la gestión de usuarios y planificación de las preguntas.

Este proyecto ha sido desarrollado en base a lo propuesto por el Observatorio Tecnológico de HP SCDS, presentando una versión simplificada de las aplicaciones actualmente disponibles en el mercado pero que contiene las funcionalidades básicas para su utilización.

La aplicación web ha sido desarrollada utilizando React y una de sus librerías, Material UI. La API web del servicio RESTful ha sido implementado en ASP.NET Core, permitiendo la conexión con la base de datos no relacional realizada en Mongo DB. La aplicación de escritorio ha sido desarrollada en C# a través de WPF (*Windows Presentation Foundation*), un estándar para desarrollo de aplicaciones para Windows.

Palabras clave: aplicación web, aplicación de escritorio, REST, React, MongoDB

Abstract

This Bachelor's Thesis describes the development of a desktop application and a web application, meant to allow an organization to check its members' satisfaction by asking a daily question. Answers to these daily questions are computed and shown to the administrators in a dashboard style web application, showing and filtering the data in different graphics and managing both user management and the questions' own scheduling.

This project has been developed based on HP SCDS Technology Observatory's proposal for a simplified alternative to already existent applications in the market but retaining all the core features for its basic use.

The web application has been developed using React and one of its libraries, Material UI. The web API for the RESTful service has been fully implemented in ASP.NET Core, allowing the connection with a NoSQL database made in MongoDB. The desktop app has been developed in C# using WPF (Windows Presentation Foundation), a standard for Windows based applications.

Key words: web application, desktop application, REST, React, MongoDB

Índice de contenido

Índice de imágenes

Índice de tablas

1. Introducción

A fin de controlar la salud y el estado de su empresa, son muchas las organizaciones que realizan distintos sondeos o encuestas a sus miembros de forma diaria, semanal o mensual [1]. Puesto que no resultaría viable realizarlo de forma manual, se recurre a aplicaciones que ofrecen un sistema integrado para la realización de preguntas y la posterior recogida de datos. Prueba de esto son las distintas aplicaciones disponibles en el mercado, como HappyForce[1] o Leapsome[2], que se ofrecen como opciones más automatizadas para poder gestionar el *feedback* de los miembros.

Estas aplicaciones permiten que los administradores diseñen un conjunto de preguntas personalizadas en base a los intereses de su organización, para comprobar la opinión de sus empleados con respecto a cuestiones como su satisfacción en la empresa, sueldo o incluso su relación con otros compañeros de trabajo.

Las preguntas se muestran de forma automática a los usuarios, y los datos de respuesta se computan posteriormente, permitiendo a la empresa filtrar y ordenar la opinión de sus empleados según distintas métricas de interés como puede ser el género o el rango de edad de quienes han respondido, lo que sirve de evaluación del funcionamiento de la propia organización, pero manteniendo el anonimato de quienes han respondido.

Este trabajo desarrolla una aplicación como la descrita, encargada de sondear a los miembros de una organización a través de una pregunta diaria elegida por la empresa y mostrar los datos de respuesta de forma sencilla en una aplicación web, a través de un gráfico resumen.

Puesto que para los empleados puede resultar molesto tener que acceder a una página web para introducir sus opiniones o usar una determinada aplicación, se propone una automatización del proceso de recogida de las preguntas. Esto se realiza a través de una aplicación de escritorio instalada en el equipo de cada empleado, diseñada para encenderse a una determinada hora todos los días y permitir la respuesta del usuario, aplazarla a un horario posterior (por ejemplo, en caso de que el empleado se encuentre en una reunión) o rechazarla.

El objetivo principal de este Trabajo de Fin de Grado es aplicar los conocimientos adquiridos durante el Grado para realizar un desarrollo completo de una aplicación que satisfaga los requisitos de la proposición original.

Para ello, se ha dividido el proyecto en tres secciones principales, cada una con un conjunto de subobjetivos o funcionalidades que deben ser cubiertas:

a) *Backend* (Base de datos NoSQL, servicio web):
    a. Permitir la gestión a través de operaciones CRUD de los principales elementos en el sistema (preguntas, respuestas, categorías, etc.)
    b. Controlar la programación de las preguntas, asegurando la existencia de un máximo de una pregunta por cada día laboral.
    c. Anonimizar las respuestas de los usuarios.

[1] https://myhappyforce.com/es/encuestas/
[2] https://www.leapsome.com/product/engagement-surveys

    d. Asegurar que cada usuario puede responder la pregunta diaria una única vez.
b) *Frontend* (Aplicación web para gestión y administración):
    a. Permitir la visualización sencilla de los datos asociados a las respuestas de un día determinado, a través de un gráfico resumen.
    b. Gestión de usuarios: dar de alta y baja usuarios con distintos permisos (administrador o usuario).
    c. Permitir el acceso únicamente a los administradores.
    d. Programación de tareas de preguntas en un rango de fechas dado.
    e. Creación y gestión de preguntas y categorías.
c) *Frontend* (Aplicación de escritorio):
    a. Lanzarse de forma automática todos los días en el mismo horario.
    b. Permitir al usuario responder la pregunta diaria, aplazarla a un horario posterior dentro del mismo día o declinar responderla.
    c. Controlar el acceso a la aplicación a través de un sistema de login, permitiendo el acceso únicamente a los usuarios básicos.

De igual forma, también se considera objetivo de este proyecto el aprendizaje de nuevas tecnologías no vistas durante el Grado y que sin embargo están presentes en el mundo laboral (React, desarrollo de servicios en .NET, etc.), así como afianzar todos los conocimientos obtenidos.

El código desarrollado para el proyecto está disponible en un repositorio público en GitHub[3].

2. Herramientas utilizadas. Metodología

En esta sección se describen las distintas tecnologías y herramientas utilizadas para el desarrollo de cada sección del proyecto, así como una breve justificación de su uso. Por último, se detalla la metodología llevada a cabo durante el proyecto.

2.1 Tecnologías

En primer lugar, se muestran las tecnologías de desarrollo usadas para el proyecto, descritas desde el nivel más bajo (base de datos) al más alto (*frontend* e interfaces gráficas).

2.1.1 MongoDB

MongoDB es una de las principales bases de datos no relacionales disponibles en el mercado. A pesar de que no garantiza las propiedades ACID *(Atomicity, Consistency, Isolation and Durability)* [2] con las que sí que cuentan sus competidoras relacionales, permite consultas rápidas y más sencillas a través de su sistema de almacenamiento mediante documentos.

Es además una base de datos más escalable y con un mejor rendimiento, capaz de gestionar una gran cantidad de datos.

[3] https://github.com/ireneeZ/TFGPulseSpam

2.1.2 ASP.NET Core

ASP.NET Core es un marco multiplataforma de código abierto para la creación y compilación de aplicaciones, destacando sus servicios y aplicaciones web.

Cuenta con un *driver* para la conexión con MongoDB (MongoDB Driver) que facilita el manejo de la propia base de datos desde el servicio. Se ha programado en C#, utilizando .NET 6.0 para garantizar la compatibilidad con la mayor cantidad de sistemas.

2.1.3 Node.js

Node.js es un entorno de ejecución para Javascript. En lo que a este proyecto respecta, se trata de una de las dependencias necesarias para poder usar React, gestionando entre otras el despliegue de la propia aplicación web y el uso del gestor de paquetes npm. Npm está formado por cliente de línea de comandos que accede a un registro de paquetes en línea, permitiendo descargar e instalar las dependencias necesarias para la aplicación web (uso de MaterialUI y sus componentes).

2.1.4 React

React es una librería Javascript que permite diseñar interfaces de usuario para el desarrollo de aplicaciones, basada en el uso de componentes. Estos componentes son secciones de código independientes que pueden reutilizarse en distintas secciones del proyecto, facilitando el desarrollo y la modularización de código.

A diferencia de otras alternativas para interfaces gráficas como Angular, React no es un *framework* completo y ha de combinarse con otras librerías y *frameworks* como React Router, encargado del enrutamiento de las distintas páginas.

React ha permitido desarrollar la vista de la aplicación web, así como la conexión via HTTP con el servicio web. Para la sección de seguridad, se ha utilizado en conjunto con JWT (JSON Web Tokens), uno de los estándares actuales para seguridad en web basado en el uso de ficheros JSON y la existencia de un token que identifica al usuario y gestiona sus permisos.

2.1.5 MaterialUI

MaterialUI es una librería de componentes gratuita disponible para React, que pone a disposición de los desarrolladores una gran cantidad de elementos de uso habitual en páginas web (botones, paneles, tablas, etc.), que ya cuentan con un formato y estilo propio, pero totalmente customizable, para simplificar el proceso de desarrollo y disminuir la cantidad de HTML y CSS necesarios para crear la interfaz gráfica de la aplicación web.

2.1.4 Windows Presentation Foundation (WPF)

API para el desarrollo de aplicaciones de escritorio para Windows. Facilita el proceso de creación de aplicaciones a través de una paleta de elementos ya diseñados por Microsoft y el uso de ficheros XAML (*eXtensible Application Markup Language*), un lenguaje declarativo similar a XML, que son a su vez controlados por clases #C.

2.2 Herramientas

A continuación, se presentan las herramientas y aplicaciones utilizadas para el desarrollo del proyecto, incluyendo tanto los IDEs *(Integrated Development Environment)* utilizados como otras herramientas destacadas.

2.2.1 Visual Studio 2022

Microsoft Visual Studio es un IDE disponible para Windows y macOS. Permite usar distintos lenguajes de programación y cuenta con distintas funcionalidades para agilizar el desarrollo. Se considera uno de los IDEs de referencia en lo que a la programación en .NET respecta.

2.2.2 Visual Studio Code

Perteneciente a la misma familia que Visual Studio, Visual Studio Code es un IDE más ligero, habitualmente usado para el desarrollo web dada la gran cantidad de *plugins* existentes y su fácil conexión con navegadores web.

2.2.3 MagicDraw UML

MagicDraw UML es una herramienta CASE *(Computer Aided Software Engineering)* que facilita el modelado de distintos diagramas y diseños de código según el estándar UML 2.3. Todos los diagramas de este documento han sido desarrollados mediante esta herramienta.

2.2.4 GitLab

GitLab es un servicio web basado en Git que permite realizar control de versiones sobre distintos repositorios, así como gestionar información relacionada con el seguimiento del proyecto como historias de usuarios y tareas, gestionadas desde distintos *sprints*.

GitLab ha permitido guardar el desarrollo del proyecto en la nube, así como servir de backlog para la colección de historias de usuario del proyecto y su división en *sprints*. Al pertenecer el repositorio usado en GitLab a la empresa HP y ser privado, se ha utilizado también GitHub para presentar una versión pública del código del proyecto.

2.2.5 MongoDB Compass y MongoDB Shell

MongoDB Compass permite el acceso a las bases de datos MongoDB existentes en un equipo a través de una interfaz gráfica, de forma rápida y sencilla. Permite insertar, editar y eliminar documentos a través de botones y generando de forma automática el cuerpo de los elementos JSON, por lo que facilita la gestión de la base de datos.

Como complemento a MongoDB Compass se ha utilizado MongoDB Shell, una consola que permite el acceso a las bases de datos MongoDB a través de terminal, de forma mucho más rápida que su alternativa gráfica. Puesto que no necesita confirmaciones como Compass, facilita el proceso de creado y destrucción de bases de datos y categorías.

2.2.6 Postman

Postman es una API (*Access Point Interface*) que permite probar otras APIs. Cuenta con alternativas de pago, pero su versión gratuita permite, entre otras, la posibilidad de realizar pruebas sobre servicios web a través de peticiones HTTP y organizar estas pruebas en una batería de tests automatizables.

2.2.7 SeleniumIDE

Se trata de un IDE diseñado para poder grabar y ejecutar tests sobre aplicaciones web, comprobando el funcionamiento de los distintos elementos de cada página y su contenido. Una de sus características destacadas es la posibilidad de poder depurar los propios tests a través de *breakpoints* y excepciones en tiempo real.

Está disponible como extensión para Google Chrome y Mozilla Firefox. Para este trabajo se ha utilizado su versión para Chrome.

2.2.8 Google Chrome

Navegador web desarrollado por Google y uno de los más utilizados a nivel mundial. A nivel de desarrollo web, cuenta con un potente conjunto de herramientas (Chrome DevTools) que facilitan la prueba de páginas web, pudiendo visualizar las peticiones HTML desde el navegador y recibiendo información acerca de los errores y su localización en tiempo de ejecución.

Todas las pruebas y despliegues relacionados con la aplicación web se han realizado en este navegador.

2.3 Metodología

Con respecto a la metodología a utilizar durante el proyecto, inicialmente se consideraron dos posibles opciones: en cascada o una metodología ágil, mostradas en la Ilustración 1.

La metodología en cascada es la versión más clásica en lo que al desarrollo software respecta, con etapas muy diferenciadas que se suceden en un orden fijo, abarcando todo el sistema desde la obtención de requisitos hasta sus pruebas finales y mantenimiento.

Sin embargo, esta misma estructura rígida planteaba varios problemas para este proyecto, incluyendo la necesidad de un análisis de requisitos exhaustivo y completo, para poder obtener una especificación que incluya todas las posibles necesidades del cliente con respecto al producto. Como inicialmente se contaba con una propuesta del proyecto, con solo algunas funcionalidades básicas definidas, no resultaba la metodología idónea.

Por ello, para el desarrollo de este proyecto se ha decidido llevar a cabo una metodología iterativa e incremental a través de *sprints*, etapas de duración definida en las que se implementan una serie de funcionalidades del producto final en base a las decisiones tomadas por el cliente.

A diferencia de las metodologías tradicionales en cascada, las metodologías ágiles permiten un desarrollo más rápido y cercano de cara al cliente, quien se encuentra involucrado en todas las grandes decisiones relacionadas con el proyecto. Las entregas de código son mucho más frecuentes y permiten que el producto se acerque más a lo deseado por el cliente, que puede observar la evolución de la aplicación y aportar *feedback* sobre el estado de esta.

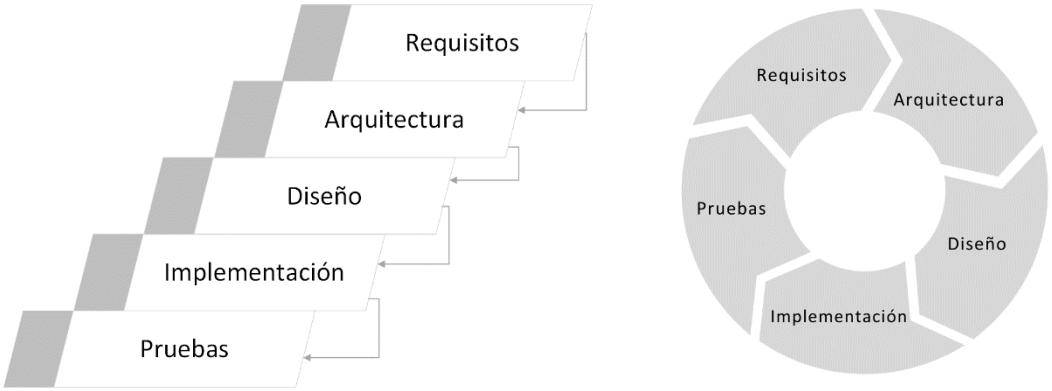


Ilustración 1: Metodología en cascada (izquierda) y metodología ágil (derecha)

Esto permite realizar cambios durante el transcurso del proyecto, descartando ideas que inicialmente iban a formar parte del ámbito del sistema, pero no se han considerado lo suficientemente relevantes en comparación a otras. Definidas las funcionalidades básicas del sistema, se prescinde de una documentación tan rigurosa como en las metodologías en cascada para poder centrar el esfuerzo en el desarrollo.

En lo que a la estructura de los *sprints* respecta, al comienzo de este Trabajo de Fin de Grado se realizó un primer sprint en forma de "rampa" para permitir familiarizarse con los distintos entornos de trabajo y todas las tecnologías que debían utilizarse, al tener únicamente algo de experiencia en .NET. A través de la realización de pequeños programas de estilo *"Hello World"*, se adquirieron unas bases sencillas para poder realizar el posterior desarrollo del sistema.

A continuación, en base a lo definido en el documento inicial de la propuesta del proyecto y una entrevista con los codirectores para comentar los aspectos principales del sistema, se definieron las historias de usuario del proyecto como épicas. Estas tareas épicas se descompusieron en historias de usuario al uso, de forma que pudiesen incluirse dentro de algún sprint.

Posteriormente, dentro del sprint correspondiente a cada historia de usuario, se añadieron subtareas en su interior, a fin de contar con tareas que pudiesen realizarse en la duración de un sprint (por ejemplo, dividiendo la implementación en distintas capas) y se completaron las historias de usuario seleccionadas con sus descripciones y toda la información relevante.

Para el desarrollo de este proyecto se han llevado a cabo un total de 10 *sprints*, con 2 semanas de duración cada uno, así como semana adicional al final del proyecto para poder revisar el sistema. Cada sprint ha quedado definido en una reunión de 1 hora en la que se comprobaba el desarrollo realizado durante el anterior sprint y los posibles aspectos de mejora en lo desarrollado. Después, se comprobaba el backlog del proyecto y se elegían las funcionalidades a implementar para la siguiente iteración.

En el Anexo 1 se muestra un diagrama de Gantt que indica la división en sprints del proyecto y las principales tareas llevadas a cabo, dividido en dos imágenes.

3. Análisis y especificación de requisitos

En esta sección se especifican los requisitos definidos para la aplicación. Puesto que se ha utilizado una metodología ágil, se ha omitido la realización de casos de uso, plantillas y otros tipos de documentaciones adicionales, centrando la especificación de requisitos en el desarrollo de historias de usuario con sus correspondientes criterios de aceptación y en la definición de los requisitos funcionales y no funcionales de la aplicación.

3.1 Requisitos funcionales

Se han agrupado los requisitos funcionales en base a los distintos roles disponibles en el sistema, usuario (U) y administrador (A), descritos respectivamente en las Tablas 1 y 2.

Cada requisito cuenta con un identificador, un nombre y una descripción sobre la funcionalidad esperada del sistema. No todos los requisitos obtenidos durante el proceso de captura de requisitos del sistema han sido implementados en la aplicación final, pero se muestran como parte de la captura de requisitos realizada.

Tal y como se explicaba en el apartado 2.3, la metodología utilizada permite seleccionar durante el transcurso del proyecto aquellas funcionalidades consideradas más importantes por el cliente y priorizarlas en su implementación, dejando de lado otras que, si bien ayudarían a mejorar el sistema, no se consideran esenciales y por ello pueden no realizarse. Por ello, durante las reuniones al final de cada sprint, se han ido escogiendo las funcionalidades más relevantes en base a las necesidades de la empresa.

3.1.1 Rol Usuario

RF-U001	**Iniciar sesión para usuario**
	El usuario deberá poder iniciar sesión en el sistema a través de su email y contraseña.
RF-U002	**Limitar respuesta diaria**
	El usuario podrá responder un máximo de una pregunta cada día.
RF-U003	**Responder pregunta**
	El usuario podrá responder la pregunta diaria introduciendo la información relevante según el tipo de esta (texto abierto, puntuación entre 1 y 5, respuesta sí/no).
RF-U004	**Omitir pregunta**
	El usuario podrá no responder la pregunta diaria.
RF-U004	**Posponer pregunta**
	El usuario podrá posponer su respuesta de la pregunta diaria, retrasándola en intervalos de media hora, 1 hora o 3 horas.

Tabla 1: Requisitos funcionales del rol Usuario

3.1.2 Rol Administrador

RF-A001	**Iniciar sesión para administrador**
	El administrador deberá poder iniciar sesión en el sistema a través de su email y contraseña, verificando que cuente con los permisos adecuados.
RF-A002	**CRUD de preguntas**
	El sistema deberá permitir la creación, borrado, modificación y actualización de preguntas, así como su visualización.

RF-A003	**CRUD de categorías**
	El sistema deberá permitir la creación, borrado, modificación y actualización de categorías, así como su visualización.
RF-A004	**Representación de la pregunta**
	El sistema deberá representar cada pregunta con un tipo de entre tres disponibles (respuesta abierta, respuesta con puntuación y respuesta sí/no).
RF-A005	**Organizar preguntas**
	El sistema deberá organizar cada pregunta con al menos una categoría que facilite el filtrado de datos.
RF-A006	**Visualización de usuario**
	El sistema deberá mostrar todos los usuarios existentes, con su nombre de usuario y email.
RF-A007	**Creación y borrado de usuarios**
	El administrador podrá crear nuevos usuarios para el sistema, tanto de tipo usuario como de tipo administrador.
RF-A008	**Visualización de resultados**
	El sistema deberá mostrar todos los resultados obtenidos para una determinada pregunta en un mismo día.
RF-A009	**Filtrado de resultados**
	El sistema permitirá organizar las respuestas según los siguientes filtros: género, departamento y edad.
RF-A010	**Creación de tareas de programación**
	El administrador podrá crear tareas de programación que incluyan un conjunto cerrado de preguntas, una fecha de inicio y fin y una hora.
RF-A011	**Ejecución de tareas de programación**
	El sistema deberá ejecutar las tareas según lo definido por el administrador, mostrando diariamente una pregunta de la lista a la hora indicada.
RF-A012	**Borrado de tareas de programación**
	El sistema deberá permitir el borrado de tareas de programación.

Tabla 2: Requisitos funcionales del rol Administrador

3.2 Requisitos no funcionales

En base a lo definido en las historias de usuario del proyecto, se han extraído los siguientes requisitos no funcionales mostrados en la Tabla 3.

Id	Descripción	Tipo
RNF-001	Se debe garantizar la anonimidad de los usuarios, impidiendo rastrear sus respuestas	Seguridad
RNF-002	El acceso a la aplicación web debe estar limitado a usuarios de tipo administrador	Seguridad
RNF-003	La autenticación de los usuarios en el sistema y la gestión de su rol debe realizarse a través de JWT	Seguridad
RNF-004	La aplicación de escritorio deberá ser compatible con Windows (versiones 8 en adelante)	Compatibilidad
RNF-005	La base de datos de la que hace uso el sistema debe ser no relacional y usar MongoDB	Compatibilidad.

RNF-006	Se deberán mostrar confirmaciones visuales (alertas) tras el borrado de los siguientes elementos del sistema: preguntas, categorías y usuarios	Accesibilidad
RNF-007	Se deberán mostrar una confirmación visual (alerta) tras la creación de una nueva tarea de programación en el sistema	Accesibilidad

Tabla 3: Requisitos no funcionales

3.3 Historias de usuario

Al inicio del proyecto se creó un *backlog* en el que almacenar las historias de usuario definidas para el proyecto. A la hora de definir dichas historias de usuario se ha seguido la estrategia habitual de las metodologías ágiles, indicando únicamente el nombre de cada historia de usuario en el momento de su creación y añadiendo los criterios de aceptación y su descripción únicamente cuando la historia se iba a tratar durante el sprint actual.

Las historias de usuario se han agrupado en épicas como se muestra en la Tabla 4.

Id	Nombre	Historias contenidas
E-000	Crear preguntas conforme a las necesidades de la organización	HU-001 Crear pregunta HU-002 Eliminar pregunta HU-003 Modificar pregunta HU-004 Ver preguntas
E-001	Agrupar preguntas en categorías identificativas	HU-005 Crear categoría HU-006 Eliminar categoría HU-007 Modificar categoría HU-008 Filtrar preguntas por categoría HU-009 Filtrar preguntas por tipo
E-002	Programar conjuntos de preguntas para un rango de fechas	HU-010 Crear tarea HU-011 Eliminar tarea HU-012 Modificar tarea HU-013 Ver tareas
E-003	Responder a la pregunta diaria según su contenido	HU-014 Responder pregunta diaria HU-015 Posponer respuesta HU-016 Cancelar respuesta HU-017 Anonimizar resultados
E-004	Visualizar los resultados de las respuestas	HU-018 Ver respuestas HU-019 Filtrar respuestas
E-005	Definir un sistema de usuarios con distintos roles que controlen su acceso	HU-020 Ver sistema de usuarios HU-021 Añadir usuario HU-022 Eliminar usuario

Tabla 4: Historias épicas

Las historias de usuario definidas para el proyecto se muestran en la tabla 5. Como se mencionaba en la sección de requisitos, no todas las historias han sido implementadas en la versión final de proyecto, debido tanto a limitaciones de tiempo en el desarrollo como a decisiones de la empresa cliente a lo largo de los distintos sprints. Estas historias de usuario que permanecen en el *backlog* y, por tanto, no se han realizado, están marcadas con una anotación (X) bajo su identificador y han sido valoradas como *nice to have* dentro del sistema. Es decir, se consideran elementos adicionales

Id	Nombre	Descripción
HU-001	Crear pregunta	Yo, como administrador, quiero crear una pregunta de manera que pueda comprobar la opinión de los empleados con respecto a un cierto tema que resulta relevante para mi empresa.
HU-002	Eliminar pregunta	Yo, como administrador, quiero eliminar una pregunta de manera que los empleados no opinen sobre temas que ya no sean relevantes para mi empresa
HU-003	Modificar pregunta	Yo, como administrador, quiero poder modificar una pregunta de manera que pueda corregir errores en su texto o clasificarla en una categoría que resulte más relevante.
HU-004	Ver preguntas	Yo, como administrador, quiero ver el listado completo de preguntas del sistema de manera que pueda comprobar cuáles son las preguntas que van a realizarse y valorar su utilidad para mi empresa, eligiendo cuales se realizarán a los empleados.
HU-005	Crear categoría	Yo, como administrador, quiero crear una categoría para poder agrupar preguntas de temática similar y que resulte más fácil encontrarlas.
HU-006	Eliminar categoría	Yo, como administrador, quiero borrar una categoría de manera que pueda eliminar una categoría que he introducido por error.
HU-007	Modificar categoría	Yo, como administrador, quiero modificar una categoría de manera que pueda corregir errores que haya cometido al crearla.
HU-008	Filtrar preguntas por categoría	Yo, como administrador, quiero filtrar las preguntas por categorías de manera que me resulte más sencillo navegar entre ellas o encontrar una pregunta concreta según su temática.
HU-009	Filtrar preguntas por tipo	Yo, como administrador, quiero filtrar las preguntas por categorías de manera que me resulte más sencillo navegar entre ellas o encontrar una pregunta concreta según su tipo.
HU-010	Crear tarea	Yo, como administrador, quiero crear una tarea de manera que pueda ahorrar tiempo, programando un determinado conjunto de preguntas para un periodo de tiempo de forma conjunta, sin necesidad de hacerlo individualmente
HU-011	Eliminar tarea	Yo, como administrador, quiero eliminar una tarea de manera que cancele la programación de las preguntas para las fechas elegidas.
HU-012 (X)	Modificar tarea	Yo, como administrador, quiero modificar una tarea de manera que pueda cambiar el rango de fechas escogido o las preguntas y adaptar la tarea a mis necesidades.
HU-013	Ver tareas	Yo, como administrador, quiero ver las tareas programadas en el sistema de manera que sepa en

		todo momento cuales son las preguntas que van a realizarse.
HU-014	Responder pregunta diaria	Yo, como usuario, quiero responder la pregunta diaria de manera que pueda dar mi opinión con respecto a un determinado tema de la empresa.
HU-015 (X)	Posponer respuesta	Yo, como usuario, quiero posponer una respuesta diaria de manera que no interrumpa mi trabajo y pueda responderla después, cuando no esté tan ocupado.
HU-016	Cancelar respuesta diaria	Yo, como usuario, quiero cancelar una respuesta diaria de manera que no esté obligado a opinar con respecto a un tema que me incomoda o que no quiero responder.
HU-017	Anonimizar resultados	Yo, como usuario, quiero que mis resultados sean anónimos de manera que pueda responder a las preguntas con sinceridad, sabiendo que mis opiniones no pueden impactar de forma negativa mi trabajo.
HU-018	Ver respuestas	Yo, como administrador, quiero ver las respuestas de una determinada pregunta para comprobar la opinión de los empleados y que mi empresa pueda actuar en consecuencia.
HU-019 (X)	Filtrar respuestas	Yo, como administrador, quiero filtrar las preguntas en base a distintas métricas definidas por mi empresa como el género, edad y departamento, para poder localizar los grupos de empleados que se encuentran más descontentos.
HU-020	Ver sistema de usuarios	Yo, como administrador, quiero ver el sistema de usuarios de manera que sepa en cualquier momento cuales son los usuarios registrados en el sistema.
HU-021	Añadir usuario	Yo, como administrador, quiero añadir usuario de manera que un miembro de mi empresa pueda acceder al sistema.
HU-022	Eliminar usuario	Yo, como administrador, quiero eliminar un usuario de manera que alguien que ya no pertenece a mi empresa o cuyos datos son incorrectos no forme parte del sistema.

Tabla 5: Historias de usuario

4. Diseño de software

En este apartado se detalla el diseño realizado en cada una de las secciones de la aplicación, así como las decisiones que han llevado a utilizar una determinada arquitectura frente a otra.

Al tratarse de un trabajo compuesto por múltiples aplicaciones y secciones interconectadas, tanto el diseño como la implementación han sido llevados a cabo desde las secciones inferiores (dominio, base de datos, servicio) hasta las aplicaciones de cliente.

4.1 Dominio

En la Ilustración 4 se muestra el diseño de dominio de la aplicación, con las entidades usadas a lo largo del sistema para representar los distintos elementos de los que hace uso el sistema y las relaciones que existen entre ellos.

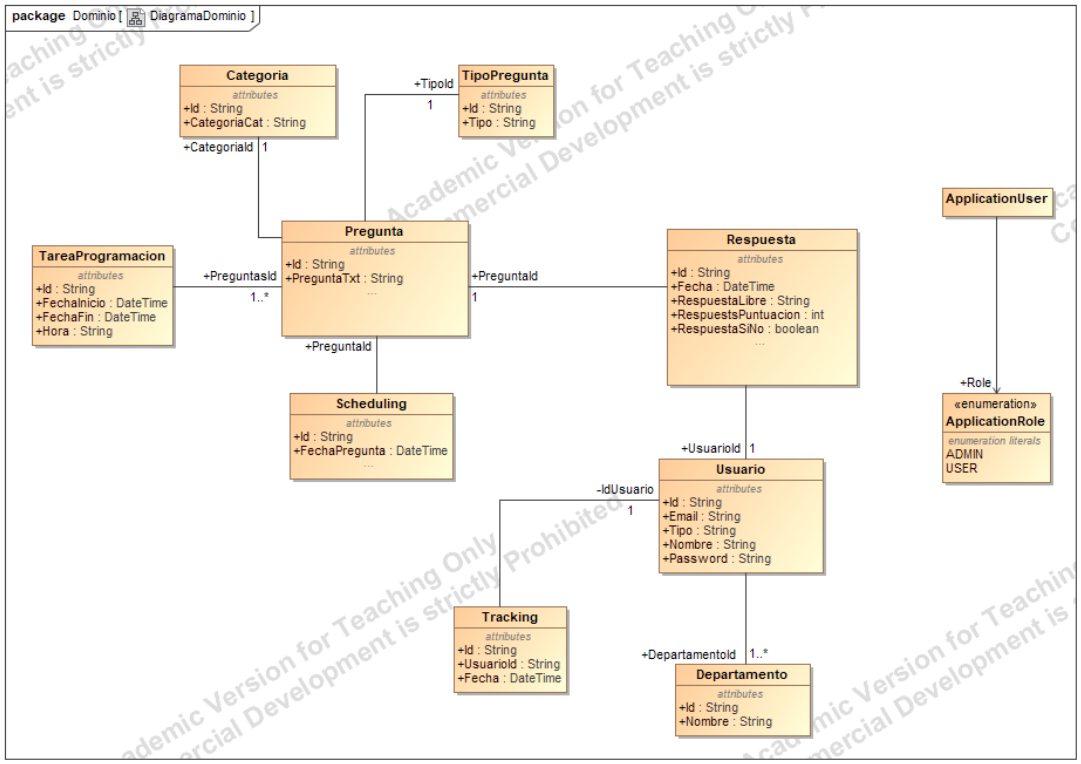


Ilustración 2: Dominio de la aplicación

Todos los nombres de las clases son representativos, pero cabe explicar el funcionamiento de algunos de ellos:

- *Tracking*. Comprueba si un cierto usuario ha respondido la respuesta diaria de la fecha indicada, lo que impide que un mismo usuario responda varias veces la misma pregunta.
- *Scheduling*. Horario referente a la próxima aparición de una cierta pregunta en la aplicación web. Se genera al incluir la pregunta en una TareaProgramación.
- TareaProgramación: Representa un conjunto de preguntas elegido para aparecer en la aplicación de escritorio en un rango de fechas concreto, según la hora elegida.

Además de estas clases de dominio, también se han definido un conjunto de DTOs, que toman información de distintas clases de dominio para poder mostrarla desde la aplicación web. Por ejemplo, en el caso de las preguntas, se agrupan sus datos propios (texto de la pregunta, ids) con los datos cargados de su categoría y tipo. Esto permite reducir el número de llamadas necesarias al servicio y reducir la complejidad de las aplicaciones que hacen uso de ellas.

A partir de este modelo de dominio se ha desarrollado el resto de la arquitectura de la aplicación.

4.2 Diseño de arquitectura

La arquitectura general del proyecto se ha diseñado conforme al patrón MVC (Modelo Vista Controlador). En este patrón se disponen tres capas principales, cada una con una función definida según se muestra a continuación:

- **Modelo**. Capa inferior encargada de la representación de los datos. Gestionada por la base de datos y las clases que modelan los documentos.
- **Controlador**: Capa intermedia en la que se encuentra la lógica de negocio del sistema y que controla la interacción entre el usuario y los datos. Gestionada por el servicio web.
- **Vista**: Capa superior encargada de la interfaz gráfica, visualización y obtención de datos. Gestionada por la aplicación de escritorio (para las respuestas) y la aplicación web.

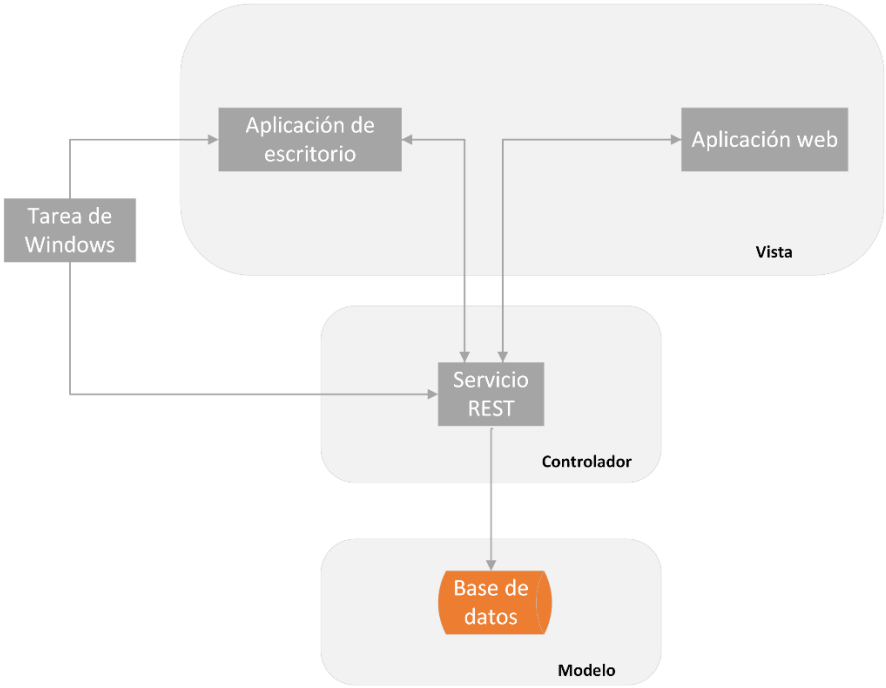


Ilustración 3: Diagrama MVC del sistema

Puesto que existen dos tipos de vistas en el sistema, se reconocen además dos tipos de uso principales en la aplicación en base al tipo de usuario.

1. Administrador. Utiliza la aplicación web para gestionar los usuarios del sistema, crear preguntas, categorías y ver las distintas estadísticas de las respuestas.
2. Usuario. Utiliza la aplicación de escritorio para contestar la pregunta diaria.

4.1.1 Diseño de la base de datos

Decidido el dominio de la aplicación y la arquitectura general del sistema, la siguiente decisión que se tuvo que tomar con respecto a este trabajo fue la forma en la que se iba a realizar la gestión de datos y su escalabilidad, a fin de decidir un tipo de base de datos que se adaptase a las necesidades del proyecto.

Actualmente existen dos grandes paradigmas en el diseño de bases de datos: relacionales y no relacionales.

Los modelos relaciones o SQL disponen los datos en tablas, relacionadas entre sí a través de claves. Dado que cada tabla únicamente recoge un tipo de dato, para extraer una pregunta sería necesario comprobar tanto la tabla en la que esté alojada dicha pregunta como todas aquellas con las que esté relacionado (por ejemplo, su categoría, tipo, etc.), lo que podría dar lugar a un gran número de consultas.

Con volúmenes de datos pequeños es una pérdida de rendimiento no demasiado apreciable, pero teniendo en cuenta el público objetivo de esta aplicación es algo que se debe tener en cuenta.

Asumiendo que una empresa de un tamaño medio (unos 200 empleados) desea utilizar la aplicación, esto generaría un más de 43000 respuestas que han de gestionarse a lo largo de un año, sin contar la propia gestión interna de preguntas, usuarios o información referente a la programación interna.

Si esto se escala a una empresa más grande, el volumen de datos a gestionar de forma diaria aumenta de forma importante, principalmente debido a la existencia de una respuesta por cada empleado de la empresa. Por lo tanto, en un sistema como el descrito en este documento, prima la velocidad en la gestión de los datos frente a cualquier otra característica.

Las bases de datos no relacionales o NoSQL alojan los datos de forma distinta a las relacionales: en lugar de basar su gestión en tablas utilizan documentos en los que se recogen los datos que guardan relación entre sí. Esto disminuye el número de consultas necesarias para obtener el mismo dato desde su alternativa SQL, haciéndolas más rápidas y efectivas cuando se trata de consultas sencillas. Además, uno de los requisitos impuestos por la empresa fue el de utilizar una tecnología no relacional.

Decidido entonces el estilo de la base de datos, se realizó un diseño inicial de los documentos del sistema y la relación entre ellos, para tener en cuenta si existían documentos incrustados o relaciones entre ellos.

Otro de los requisitos cruciales para el diseño de la base de datos era garantizar la anonimidad de los usuarios que hiciesen uso de la aplicación, para evitar tanto que las respuestas de un usuario en concreto se pudiesen rastrear como la posible interpolación de la identidad de alguien mediante sus respuestas.

El diseño original de la base de datos contaba principalmente con tres documentos, preguntas, respuestas y usuarios, donde los usuarios contaban con todos los datos identificativos necesarios para los posibles métodos de filtrado definidos en los requisitos.

Dado que esto resultaba una contradicción directa del requisito de anonimidad, se decidió mover estos posibles datos identificativos al cuerpo del documento respuestas, desnormalizando los datos, pero impidiendo que se puedan asociar datos representativos a un usuario en concreto.

4.1.2 Diseño del servicio web

Para permitir la conexión entre la base de datos y las distintas aplicaciones, se ha desarrollado un servicio RESTful. A través de los métodos propios de HTTP (GET, POST, PUT y DELETE), permite realizar consultas y modificar los datos disponibles en la base de datos a través de la aplicación web. Su conexión con la aplicación de escritorio permite la creación de las respuestas.

El despliegue del servicio se realiza de forma local en el puerto 5001. Por defecto, Visual Studio despliega también el servicio en el puerto 5000 para el protocolo http, pero se ha omitido al ser menos seguro.

Para cada recurso del sistema se permite realizar operaciones CRUD sobre él *(Create, Read, Update, Delete)*. Para algunos recursos existe la posibilidad de filtrar los datos en base a distintas *queries*, representadas con un guion (-) bajo el correspondiente método GET.

Todas las URIs mostradas a continuación parten de la URI base del servicio (https://localhost:5001/api), pero solo se muestra el *path* de acceso al recurso por simplicidad.

Recurso	**URI**	**Métodos**	**Códigos de respuesta**
Pregunta	/preguntas/{idPregunta}	GET POST PUT DELETE	200, 404 200,404,409 204,404 200,404
Lista Preguntas	/preguntas	GET -categoria -tipo	200,404
Respuesta	/respuestas/{idRespuesta}	GET POST DELETE	200, 404 200,404,409 200,404
Lista Respuestas	/respuestas	GET	200,404
Usuario	/usuarios/{email} /usuarios/id/{idUsuario}	GET POST PUT DELETE	200, 404 200,404,409 204,404 200,404
Lista Usuarios	/usuarios	GET -tipo	200,404
Categoria	/categorias/{idCategoria}	GET POST PUT DELETE	200,404

Lista Categorias	/categorías	GET	200,404
TiposPregunta	/tipos/{idTipo}	GET POST PUT DELETE	200, 404 200,404,409 204,404 200,404
Lista Tipos	/tipos	GET	200,404
Departamento	/departamento/{idDepartamento}	GET POST PUT DELETE	200, 404 200,404,409 204,404 200,404
Lista Departamentos	/departamentos	GET	200,404
Tracking	/trackings/{idUsuario}/{fecha} /trackings/usuarios/{email}/{fecha}	GET POST PUT DELETE	200, 404 200,404,409 204,404 200,404
Schedule	/schedules/{idSchedule} /schedules/{fecha}	GET -fecha	200,404
Lista Schedules	/schedules	GET	200,404
TareaProgramacion	/tareas/{idTarea}	GET POST DELETE	200, 404 200,404,409 200,404
Lista Tarea Programacion	/tareas	GET	200,404

Tabla 6: Diseño del servicio RESTful

Al ser MongoDB una tecnología basada en el uso de documentos, el diseño de cada recurso ha sido realizado en JSON. Se muestra en la tabla 7 unos documentos de ejemplo de algunos recursos representativos del sistema, según se encuentran actualmente en la base de datos.

Recurso	Documento
Pregunta	_id: ObjectId('6475b7bc766f4cfa636ad72b') pregunta: "¿Se encuentra satisfecho con la empresa?" categoria_id: ObjectId('64634b787608976f76583495') tipo_id: ObjectId('64750c381d13403e1789d563')
Respuesta	_id: ObjectId('649df4a4517222aa95702f97') tipo: "Libre" respuesta_libre: "Sí, estoy muy satisfecho. Hay un excelente ambiente de trabajo y buena…" respuesta_sino: null respuesta_puntuacion: null fecha_pregunta: 2023-06-29T21:15:56.801+00:00 usuario_id: "lola@example.com" pregunta_id: ObjectId('6475b7bc766f4cfa636ad72b')

Tabla 7: Diseño de documentos

Para el diseño de arquitectura del servicio, se ha seguido de nuevo el modelo MVC. Puesto que se trata de un servicio, no existe una capa de vista como tal, sino que es la capa de controlador la encargada de realizar la conexión entre las peticiones HTTP y el resto del sistema.

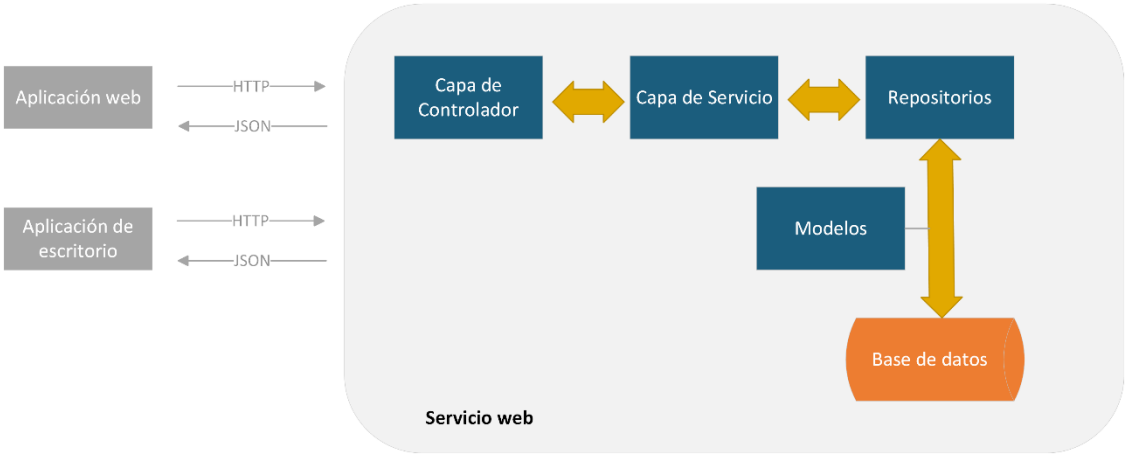


Ilustración 4: Diagrama MVC del servicio

El servicio cuenta con las siguientes secciones, vistas de nivel inferior al superior:

- **Modelos**. Incluyen la representación de los distintos elementos que se van a almacenar y recoger en la base de datos, permitiendo el mapeado entre objetos de C# y los documentos existentes en la base de datos.
  Además de las representaciones habituales, también se ha creado un conjunto de DTOs *(Data Transfer Objects)* que permiten cargar datos entre distintos modelos de forma transparente para el usuario.
- **Capa de servicio**. Gestiona el acceso a la base de datos y la modificación de sus distintas colecciones. Cada servicio definido gestiona una única colección o repositorio de datos.
- **Capa de controlador**. Capa externa de la API, accesible desde la aplicación web y de escritorio. Realiza la gestión de los métodos HTTP así como de los distintos DTOs. Existen controladores con acceso a múltiples servicios, en base a sus necesidades.

De esta forma, cada tipo de recurso del sistema está gestionado por un servicio, accedido por uno o más controladores para efectuar cambios sobre los datos. Para facilitar el proceso de implementación y las pruebas del sistema, se han diseñado una serie de interfaces que relacionan estos servicios (Ilustración 5). La representación de las relaciones entre cada controlador y sus servicios se muestran en la Ilustración 4, a continuación.

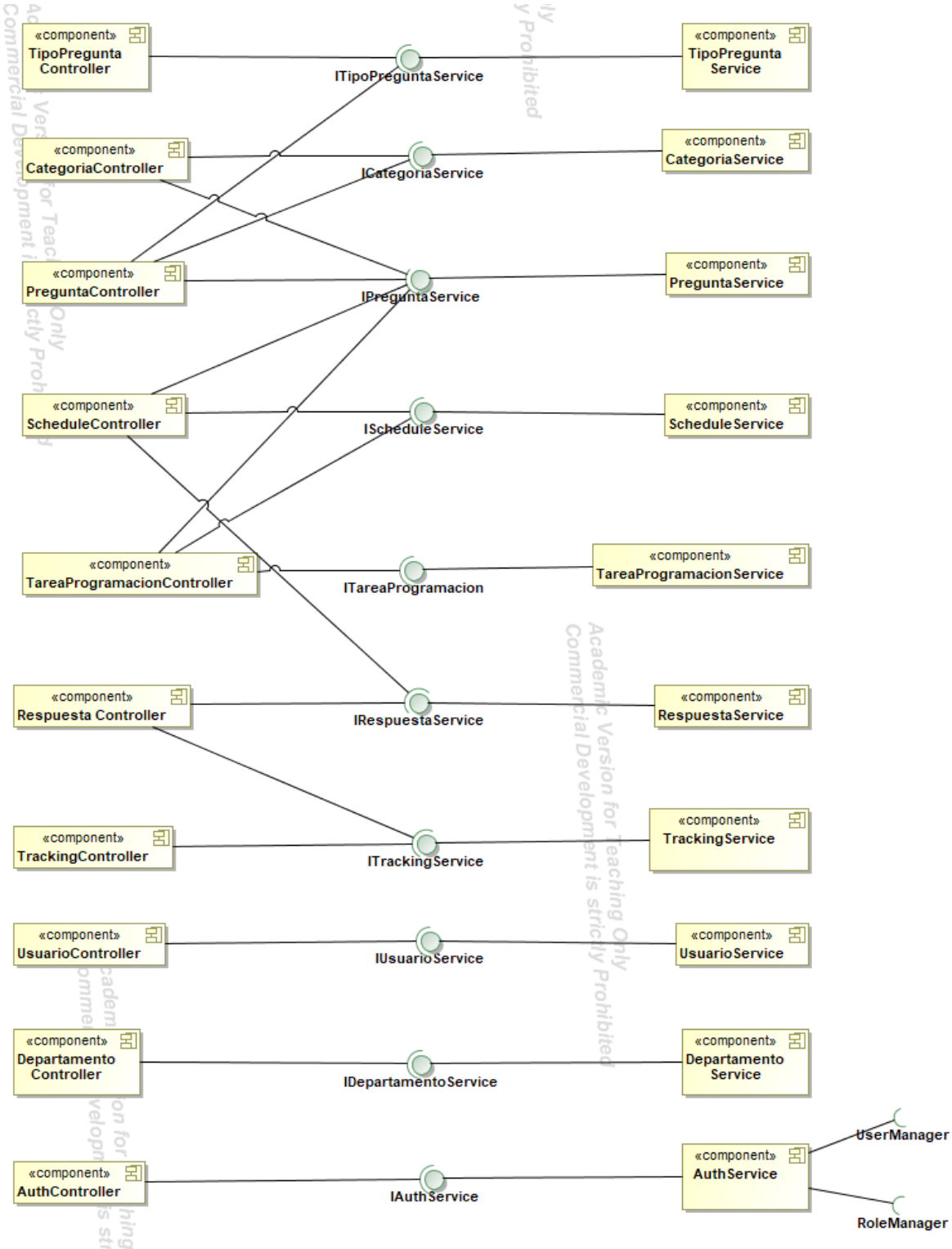
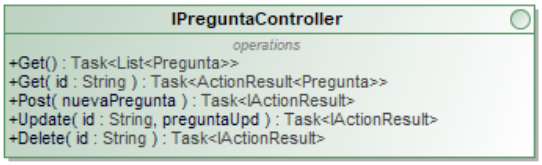


Ilustración 5: Arquitectura del servicio REST

Se muestran también las interfaces a implementar para las distintas capas. Dado que la estructura es similar entre los distintos elementos, en la Ilustración 5 se muestran como representación las interfaces correspondientes a la gestión de las preguntas, al ser uno de los recursos principales del sistema.



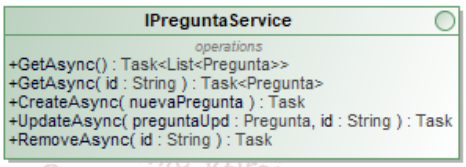


Ilustración 6: Ejemplo de interfaces CRUD para controladores y servicios

4.1.3 Diseño de la aplicación web

A la hora de desarrollar la aplicación web, se ha hecho un listado de las distintas páginas que deben estar alojadas, la ruta de acceso a cada una y una breve descripción de la funcionalidad que debe incluir, conforme a la siguiente tabla.

Página	**Ruta**	**Descripción**
Inicio	/inicio	Página de inicio que muestra la fecha actual y una serie de botones para acceder a algunas de las páginas del sistema.
Login	/login	Gestión de login de usuarios. Solo permite el acceso a los usuarios con rol de administrador.
Preguntas	/preguntas	Gestión de preguntas y categorías. Muestra las preguntas y categorías existentes y permite modificarlas, crearlas y borrarlas.
Programación	/programacion	Creación de tareas de programación con una o más preguntas para un rango de fechas dado y un horario. Visualización de las tareas creadas.
Estadísticas	/estadisticas	Visualización de los resultados de las preguntas mediante gráficos.
Usuarios	/usuarios	Gestión y visualización de los usuarios básicos del sistema. Permite al administrador añadir nuevos usuarios.

Tabla 8: Routing de la aplicación web

Además de estas rutas, se ha decidido redirigir cualquier URL que no coincida con las indicadas a la página de login.

Para comenzar a diseñar la interfaz de la aplicación web se han tenido en cuenta los diseños de otras páginas similares estilo *dashboard* disponibles en la web. Todas estas páginas comparten tres elementos en común:

- Appbar. Barra superior con opciones de búsqueda y datos relacionados con el usuario.
- Sidebar. Barra lateral que aloja el acceso a las distintas páginas de la aplicación, accediendo a través de botones.
- Panel general. Donde se muestran los datos referentes a cada página.

Decididos los elementos principales de la aplicación, que aparecen en cada página del sistema, se han realizado distintos mockups en papel para crear un diseño básico de cada página, probando distintas configuraciones de elementos hasta dar con una versión definitiva que pudiese implementarse.

En lo que al diseño arquitectónico representa, cada página de la aplicación está representada por un componente (que incluye otros componentes en su interior para gestionar tablas, ventanas emergentes, etc.). Todos los componentes se gestionan desde un componente central que carga la aplicación web y gestiona el *routing* entre las distintas páginas.

4.1.4 Diseño de la aplicación de escritorio

Para el diseño de la aplicación de escritorio se ha decidido usar una arquitectura MVVM *(Model View ViewModel)*. Su objetivo principal es el de separar al máximo posible los elementos de la vista de la lógica de negocio y el funcionamiento del sistema.

En una arquitectura MVC, como la implementada en el servicio, son los elementos de la vista quienes invocan las respectivas llamadas sobre el controlador, quien accede a los datos mediante su lógica interna. La vista recoge la respuesta y modifica la interfaz gráfica para mostrar los cambios, según se haya definido.

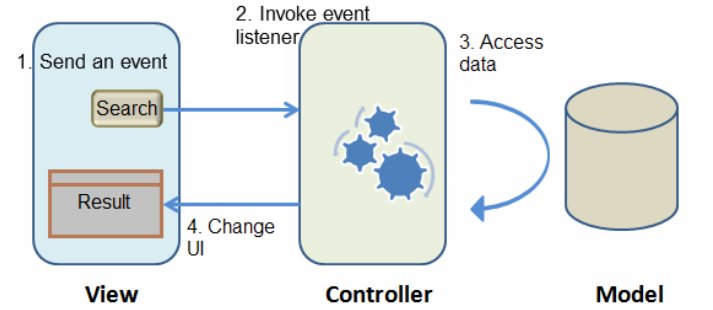


Ilustración 7: Potix Corporation. MVC Model [Figura]. Recuperado de https://www.zkoss.org/zkdemo/getting_started/mvc

Sin embargo, la arquitectura MVVM introduce un cambio. La relación entre la vista y quien anteriormente era el controlador (ahora *ViewModel*, si bien la relación no es del todo equivalente) está representada por un binder, un sistema de enlazado que une el estado de los diferentes elementos de la vista con el *ViewModel*, de forma que se actualice automáticamente.

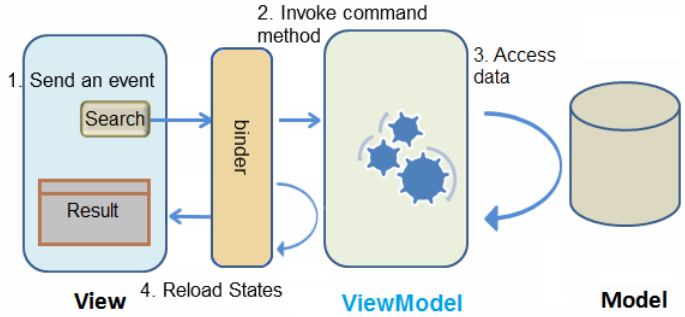


Ilustración 8: Potix Corporation. MVVM Model [Figura]. Recuperado de https://www.zkoss.org/zkdemo/getting_started/mvvm

El modelo MVVM cuenta con tres capas, similares a la arquitectura MVC:

- **Modelo**. Representación del modelo de datos de la aplicación, mediante clases y DTOs. Orientado a la conexión con un *backend* del que extraer la información (en este caso, el servicio web desde el que se recuperan los datos).
  A diferencia del modelo MVC, es en el Modelo donde se sitúa la lógica de negocio de la aplicación, comunicándose con el *ViewModel* para realizar cambios en la vista.
- **View**. Interfaz gráfica y definición de los elementos que se van a mostrar en pantalla. Carece de lógica de negocio de ningún tipo y se limita a mostrar la información designada.
- **ViewModel**. Encargado de gestionar el estado de la vista y modificarlo según la lógica de negocio procedente del modelo.

La principal ventaja de la arquitectura MVVM frente a la MVC es el desacople entre los componentes. Cada capa tiene una función muy concreta y la lógica de negocio del sistema se encuentra dentro del modelo, lo cual facilita la realización de *tests* unitarios que comprueben el funcionamiento de dicha lógica (al ser totalmente independiente de la vista).

En una arquitectura MVC existe además una relación entre todas las capas. La vista es dependiente de los modelos para mostrar información y estos son a su vez gestionados por el controlador, quien debe modificar el modelo y actualizar la vista. En el caso de que un modelo se actualice, la vista necesita conocer esta actualización para cambiar los datos mostrados. Esta "actualización" se realiza en el modelo MVVM de forma automática, a través del *binding*.

De esta forma, en la Ilustración 8 se muestra la arquitectura resultante de la aplicación de escritorio, obviando las clases que representan los datos del modelo por simplicidad. Los modelos utilizados son equivalentes a los del servicio, cuyo diseño se detalla en la sección 4.4.

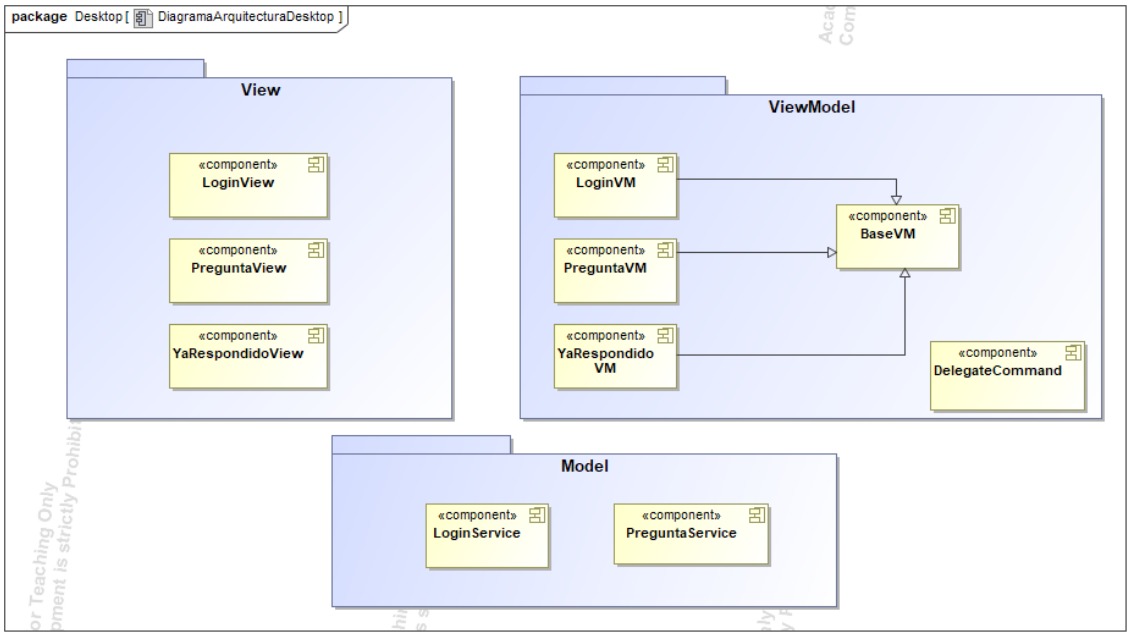


Ilustración 9: Arquitectura de la aplicación de escritorio

A partir de este análisis de la arquitectura se han diseñado los elementos básicos de la aplicación en forma de ventanas, similar a como ya se hizo anteriormente con la aplicación web.

Ventana	Descripción
Login	Permite el acceso al sistema a los usuarios. Valida el email y contraseña del usuario.
Pregunta del día	Muestra la pregunta del día según la programación existente. Permite responder en base al tipo de la pregunta, omitir o posponer la pregunta.
Ya Respondido	Indica que ya se ha realizado la respuesta diaria y se puede cerrar la aplicación.

Tabla 9: Ventanas de la aplicación de escritorio

4.3 Diseño de despliegue

Por el momento el proyecto no sea ha desplegado ninguna aplicación en un servidor de producción, sino que está todo alojado en el equipo local de desarrollo. En un entorno de producción real sería necesario desplegar el servicio REST en un servidor real, así como la aplicación web.

Aunque existen infinidad de posibles configuraciones, en la Ilustración 10 se muestra un posible despliegue para el sistema, diferenciando los siguientes elementos:

- Cliente. Ordenador portátil o de escritorio de cada empleado de la empresa con sistema operativo Windows, que hace uso de la aplicación de escritorio instalada localmente.
- Cliente web. Se separa del cliente de escritorio por simplicidad en el diagrama, aunque podría tratarse del mismo equipo. Accede a la URL en la que se encuentra desplegada la aplicación web a través de un navegador Google Chrome.
- Servidor y base de datos. Desplegados de forma local en un equipo de la empresa.

- Aplicación web. Desplegada en un servidor en la nube (por ejemplo, Microsoft Azure) y accesible mediante su URL.

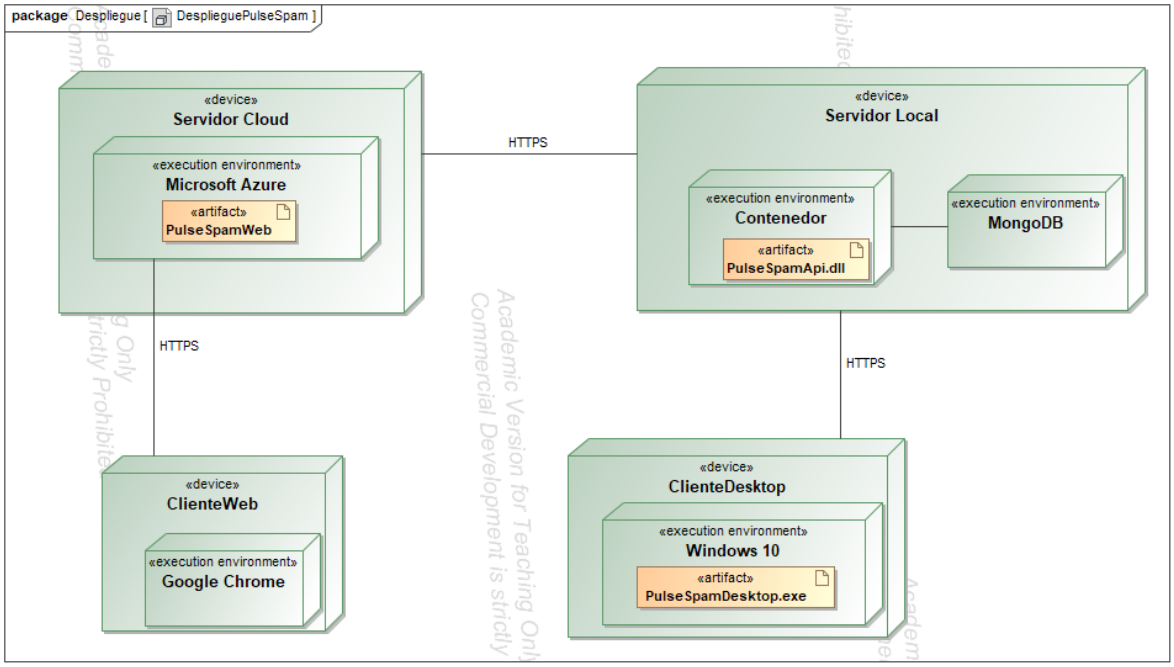


Ilustración 10: Diagrama de despliegue del sistema

5. Implementación

5.1 Implementación del servicio REST

Tal y como se explicaba en la sección 4, el servicio REST está compuesto por tres capas principales y sus respectivos componentes.

Modelos

Son las clases C# que representan los documentos disponibles en la base de datos. Cada clase está identificada por un elemento Id de tipo BsonId que permite identificar el objeto en la base de datos y realizar el mapeado a través del *driver* para MongoDB de .NET, así como sus respectivos atributos y referencias a otros tipos según lo indicado en el diagrama de clases de dominio indicadas en la Ilustración 6.

En base a este diseño se va a analizar la implementación realizada para las preguntas. En la Ilustración 11 se muestra como ejemplo la implementación de la clase Pregunta, que modela las preguntas del sistema.

```
public class Pregunta
{
    [BsonId]
    [BsonRepresentation(BsonType.ObjectId)]
    49 referencias
    public string? Id { get; set; }

    [BsonElement("pregunta")]
    19 referencias
    public string PreguntaTxt { get; set; }

    [BsonRepresentation(BsonType.ObjectId)]
    [BsonElement("categoria_id")]
    10 referencias
    public string CategoriaId { get; set; }

    [BsonRepresentation(BsonType.ObjectId)]
    [BsonElement("tipo_id")]
    8 referencias
    public string TipoId { get; set; }

    0 referencias
    public override bool Equals(object? obj)
    {
        return obj is Pregunta pregunta &&
               Id == pregunta.Id &&
               PreguntaTxt == pregunta.PreguntaTxt &&
               CategoriaId == pregunta.CategoriaId &&
               TipoId == pregunta.TipoId;
    }

    0 referencias
    public override int GetHashCode()
    {
        return HashCode.Combine(Id, PreguntaTxt, CategoriaId, TipoId);
    }
}
```

Ilustración 11: Código de la clase Pregunta

Los elementos de la clase Pregunta cuentan con dos atributos principales además de su id: el texto de la pregunta y el id de la categoría y el tipo al que pertenecen. Dado que se está usando una base de datos no relacional no existe la posibilidad de utilizar técnicas ORM (Object-Relational Mapping) que permitan cargar dinámicamente los objetos desde su referencia.

Esto proponía dos posibles alternativas para resolverse: utilizar técnicas ODM, el equivalente no relacional de los ORM o gestionarlo a través del propio código. A fin de simplificar el código y dado que no todas las clases necesitan este tipo de carga, se ha optado por realizar a través del código.

De esta forma, además de los modelos, se han creado clases DTO para facilitar la transferencia de datos. Un DTO (Data Transfer Object) es un objeto que tiene como objetivo el transporte de datos, tomando por ejemplo datos procedentes de distintas entidades y encapsulándolos en una única clase.

Continuando con el ejemplo de Pregunta, en el diseño del servicio se cuentan con dos representaciones para los objetos pertenecientes a este tipo. Pregunta, encargada del mapeado entre la base de datos y el servicio y PreguntaTransfer, una clase que encapsula los atributos propios de una pregunta y de sus correspondientes categoría y tipo. Esto permite acceder a los datos completos desde la aplicación web, simplificando la gestión de datos.

El código de la clase PreguntaTransfer se muestra en la Ilustración 12

```
public class PreguntaTransfer
{
    [BsonRepresentation(BsonType.ObjectId)]
    [BsonElement("pregunta_id")]
    1 referencia
    public string? Id { get; set; }

    [BsonRepresentation(BsonType.ObjectId)]
    [BsonElement("categoria_id")]
    1 referencia
    public string? CategoriaId { get; set; }

    [BsonRepresentation(BsonType.ObjectId)]
    [BsonElement("tipo_id")]
    1 referencia
    public string? TipoId { get; set; }

    [BsonElement("pregunta")]
    1 referencia
    public string PreguntaTxt { get; set; }

    [BsonElement("categoria")]
    2 referencias
    public string CategoriaCat { get; set; }

    [BsonElement("tipo")]
    2 referencias
    public string Tipo { get; set; }

    2 referencias
    public PreguntaTransfer(Pregunta pregunta, Categoria cat, TipoPregunta tipo)
    {
        Id = pregunta.Id;
        CategoriaId = pregunta.CategoriaId;
        TipoId = pregunta.TipoId;
        PreguntaTxt = pregunta.PreguntaTxt;
        CategoriaCat = cat.CategoriaCat;
        Tipo = tipo.Tipo;
    }
}
```

Ilustración 12: Código de la clase PreguntaTransfer

Los objetos de la clase PreguntaTransfer se crean desde el propio controlador de preguntas para los métodos GET, cargando la pregunta, categoría y tipo de pregunta asociados.

Servicios

Encargados del acceso a los distintos repositorios de datos y su gestión, según los distintos métodos disponibles.

Todos los servicios cuentan con una estructura similar, según lo definido en su respectiva interfaz, disponible en la Ilustración 13.

```
public interface IPreguntaService
{
    2 referencias
    public Task<List<Pregunta>> GetAsync();

    15 referencias
    public Task<Pregunta?> GetAsync(string id);

    2 referencias
    public Task<List<Pregunta>> GetByCategory(string categoria);

    2 referencias
    public Task CreateAsync(Pregunta nuevaPregunta);

    3 referencias
    public Task UpdateAsync(string id, Pregunta preguntaUpd);

    2 referencias
    public Task RemoveAsync(string id);
}
```

Ilustración 13: Código de la interfaz IPreguntaService

De esta forma, cada servicio ofrece métodos para realizar operaciones CRUD sobre los datos que maneja. Esto ha permitido posteriormente poder desarrollar pruebas unitarias de algunos servicios de la aplicación, mediante el uso de mocks.

Cada servicio cuenta con un atributo IMongoCollection<T> que funciona como repositorio de los elementos del tipo correspondiente, por lo que cada servicio gestiona un tipo de datos único. Este repositorio ofrece una serie de métodos que permiten buscar los elementos en su interior bajo una serie de parámetros. Por ejemplo, para encontrar una determinada pregunta en base a su id.

```
public async Task<Pregunta?> GetAsync(string id) =>
            await _preguntaCollection.Find(x => x.Id ==
id).FirstOrDefaultAsync();
```

Controladores

Los controladores conforman la capa más externa del servicio, permitiendo la conexión entre aplicaciones externas y el propio servicio REST.

Cada controlador ofrece los métodos HTTP disponibles para cada recurso, en base a lo indicado en su diseño. Si bien el mapeado de algunas clases es directo, cabe destacar como ejemplo el controlador para preguntas, PreguntasController, que accede tanto al servicio asociado a las preguntas como a las categorías y los tipos para realizar la carga de datos en los objetos DTO.

Cuando se realiza una operación de creado, modificación o eliminación, se trabaja con un objeto básico de la clase Pregunta. Para los métodos GET, sin embargo, se hace uso de la

clase PreguntaTransfer. Esto permite encapsular los datos tanto de preguntas como de sus categorías y tipos en una única clase, de forma que sea más sencillo de gestionar desde las aplicaciones externas.

A fin de no penalizar el rendimiento del servicio ni el diseño, se ha decidido limitar el alcance de cada servicio a un tipo de objetos concreto, de forma que sea el propio controlador el que cree el objeto DTO en base a los datos del objeto de la clase Pregunta y realizando las llamadas necesarias a los otros servicios que gestionan el tipo y las categorías respectivamente.

Todos los controladores cuentan con una estructura similar en base a los métodos HTTP que se ofrecen, según la tabla de la sección 4.1.2. La cabecera de los métodos es la siguiente, con X indicando la clase de la que se encarga el controlador en cuestión.

public async Task<List<XTransfer>> Get()
public async Task<ActionResult<XTransfer>> Get(string id)
public async Task<IActionResult> Post(X nuevoX)
public async Task<IActionResult> Update(string id, X updX)
public async Task<IActionResult> Delete(string id)

Tabla 10: Métodos de los controladores del servicio REST

Tal y como se indicaba en la implementación de los servicios, todos los métodos son asíncronos para mejorar el rendimiento general del sistema. Los métodos asíncronos tienen como principal ventaja la liberación del thread en uso mientras se procesa su petición, lo que evita posibles cuellos de botella innecesarios en el acceso a la base de datos, pudiendo dedicar el tiempo necesario a esperar este acceso en otras cuestiones de la aplicación.

En .NET, esta gestión asíncrona se realiza a través del uso de Tasks y la palabra reservada await. Además, se utilizan elementos ActionResult. Estos objetos son representaciones del resultado de una llamada sobre el controlador, pudiendo contener objetos C# según se haya definido en la lógica de negocio, distintos estados de error HTTP u objetos JSON, entre otros.

Los elementos IActionResult son similares, pero no especifican el tipo del objeto a devolver, permitiendo retornar múltiples resultados sin necesidad de implementar diferentes métodos, lo que permite mayor flexibilidad.

En la Ilustración 13 se muestra la implementación para el método Get de la clase TareaProgramacionController.

```
[HttpGet]
2 referencias
public async Task<List<TareaProgramacionTransfer>> Get()
{
    List<TareaProgramacion> tareas = await _tareaService.GetAsync();
    List<TareaProgramacionTransfer> tareasTransfer = new List<TareaProgramacionTransfer>();

    foreach (TareaProgramacion tarea in tareas)
    {
        string[] tareaPreguntasIds = tarea.PreguntasId;
        List<Pregunta> preguntasTarea = new List<Pregunta>();

        foreach (string idPregunta in tareaPreguntasIds)
        {
            Pregunta p = await _preguntaService.GetAsync(idPregunta);
            preguntasTarea.Add(p);
        }
        tareasTransfer.Add(new TareaProgramacionTransfer(tarea, preguntasTarea));
    }
    return tareasTransfer;
}
```

Ilustración 14: Código del método Get de TareaProgramacionController

5.2 Implementación de la aplicación web

React es una librería Javascript que permite diseñar interfaces de usuario a través del uso de componentes, elementos prediseñados que pueden insertarse en páginas web. Aunque Javascript es el lenguaje de referencia en lo que a las aplicaciones web, para mejorar el control de errores se decidió programar los componentes en Typescript, que ofrece tipado de elementos entre otras ventajas.

Para estructurar la aplicación web se ha dividido el código entre componentes reutilizables entre páginas (*components*), tipos de datos (*types*) y elementos propios de cada página, agrupados por página (*pages* y sus subcarpetas).

Existe además una carpeta *constants*, que cuenta con un fichero urls.tsx en el que se describen todas las URLs de acceso al servicio REST. Esto facilita el posible despliegue del servicio en otro servidor, teniendo que cambiar únicamente estas direcciones a las nuevas URLs.

Además, cuenta la clase authInfo.tsx, que extrae el JWT asociado al usuario desde el localStorage o un objeto vacío en caso de no existir. La definición de seguridad en la aplicación web y su implementación de JWT se explica en más detalle en la sección 6.

En lo que a la estructura de la aplicación respecta, se ha seguido la Tabla 8 para definir el *routing* del sistema. El *routing* define las distintas URLs existentes en la aplicación y los elementos asociados a ellas, de tal forma que un acceso a la página /login redireccione al componente encargado de la gestión del login.

En la clase principal de la aplicación, App.tsx, se especifica la estructura jerárquica de la aplicación y las distintas rutas que existen para acceder a sus páginas, a través de los componentes Routes y Route de la libería React Router. Para poder ocultar todas las páginas salvo la de login, se ha creado un componente PrivateRoutes que determina, a través de una función privada hasJWT, si el usuario está o no logueado en el sistema comprobando si existe un JWT en el localStorage.

Si no existe un token, se redirecciona al usuario a la página de login. En caso contrario, se permite que el sistema renderice el listado de rutas anidadas según están definidas en la página inicial. La página asociada al login no forma parte de las rutas hijas de PrivateRoutes dado que siempre debe ser creada.

```
function App() {
  return (
    <LocalizationProvider dateAdapter={AdapterDayjs}>
      <div className="app">
        <main className="main">
              <Routes>
              <Route element={<PrivateRoutes/>}>
                <Route element={<LayoutSistema />}>
                  <Route path="/inicio" element={<Inicio />} />
                  <Route path="/preguntas" element={<Preguntas />} />
                  <Route path="/programacion" element={<Programacion />} />
                  <Route path="/estadisticas" element={<Estadisticas />} />
                  <Route path="/usuarios" element={<Usuarios />} />
                  <Route path="*" element={<Navigate to="/login" replace />} />
                </Route>
              </Route>
              <Route path="/login" element={<Login />} />
              </Routes>
        </main>
      </div>
    </LocalizationProvider>
  );
}

export default App;
```

Ilustración 15: Clase principal de la aplicación web, App.tsx

Las URLs de la página pueden agruparse en dos tipos: la página de *login* y el resto de las páginas, a las que un usuario administrador no puede acceder sin antes haber iniciado sesión. Todas estas páginas comparten además una serie de elementos comunes y fijados como son la barra de la parte superior y la barra lateral, que han de ocultarse en la página de inicio de sesión.

Como referencia de como se ha realizado la implementación de la aplicación web, se analiza el código de la página /programación, representado por el componente Programacion.

La estructura de cada componente de la aplicación web puede definirse en tres segmentos: estados, funciones y render.

```
const [tab, setTab] = useState(0);

const [fechaInicio, setFechaInicio] = useState<Dayjs | null>(today);
const [fechaFin, setFechaFin] = useState<Dayjs | null>();
const [hora, setHora] = useState<Dayjs | null>(today);

const [preguntasChecked, setPreguntasChecked] = useState<string[]>([]);
```

Ilustración 16: Estados del componente Programacion

Los estados controlan el funcionamiento de ciertos elementos destacados de la página, registrando tanto acciones del usuario como propias del sistema que modifican los datos a mostrar. En la Ilustración 8 se observa el elemento que controla la pestaña

El estado de cada elemento de la página está controlado a través de una pareja [variable, setVariable], mediante el *hook* useState de React. Esta es una funcionalidad propia de React que permite controlar el estado de un elemento y modificarlo. Las parejas están controladas por un método de tipo *listener*, handle{Variable}Change, que actualiza el valor guardado en el estado cada vez que se detecta una modificación en la página (por ejemplo, se escoge una determinada fecha).

En la Ilustración 17 se muestra un ejemplo de listener que crea una nueva programación en base a los datos introducidos por el usuario en la ventana.

```
const handleProgramacionOk = () => {
    var fechaIniDate = fechaInicio?.toDate();
    var fechaFinDate = fechaFin?.toDate();
    var horaString = hora?.toString();
    const programacion = {
        fechaIni: fechaIniDate, fechaFin: fechaFinDate, hora: horaString,
        preguntasId: preguntasCheckedExport
    } as TareaProgramacion;
    creaProgramacion(programacion);
    //Limpiar preguntas
    setPreguntasChecked([]);
};
```

Ilustración 17: Listener para la creación de programación

Para obtener las categorías, tipos de datos e incluso las propias preguntas para mostrarlas en la tabla, se utiliza la función *fetch*. *Fetch* permite realizar peticiones HTTP indicando la URL, método, cabeceras y cuerpo de la petición. Estos datos se rellenan en base a lo elegido por el usuario, con la cabecera *Authorization* para permitir el acceso al servicio y los datos referentes a la petición en el cuerpo.

En la Ilustración 18 se muestra un ejemplo de petición HTTP mediante React.

```
function creaProgramacion(tarea: TareaProgramacion) {
    return fetch(TAREA_PROGRAMACION_GET, {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json',
            'Authorization': `Bearer ${token}`,
        },
        body: JSON.stringify({
            'FechaInicio': tarea.fechaIni,
            'FechaFin': tarea.fechaFin,
            'Hora': tarea.hora,
            'PreguntasId': tarea.preguntasId
        })
    })
        .then(response => {
            if (!response.ok) {
                throw new Error('ERROR');
            } else {
                setState({
                    showFailAlert: false,
                    showSuccessAlert: true
                  })
                return response.json();
            }
        })
        .catch(err => {
            setState({
                showSuccessAlert: false,
                showFailAlert: true
            })
        })
}
```

Ilustración 18: Petición HTTP para crear una programación

Además de la obtención de datos, también se realiza la gestión de errores de cada petición HTTP en base al diseño del servicio mostrado anteriormente. El resultado de cada proceso (creación o error, para la tarea de la Ilustración 18) se muestra por pantalla a través de una alerta en la zona inferior de la página.

En lo que al aspecto gráfico de cada componente respecta, se trata de componentes reutilizables de MaterialUI (MUI), repartidos en distintas estructuras de *layout*.

De entre los componentes utilizados cabe destacar por su complejidad DataGrid, un modelo de tabla customizable para la visualización de datos, como se muestra en la Ilustración 19. Se ha diseñado de forma que se carguen los datos de los elementos, incluyendo los ids necesarios para las posteriores modificaciones sobre las preguntas, pero invisibles para el usuario.

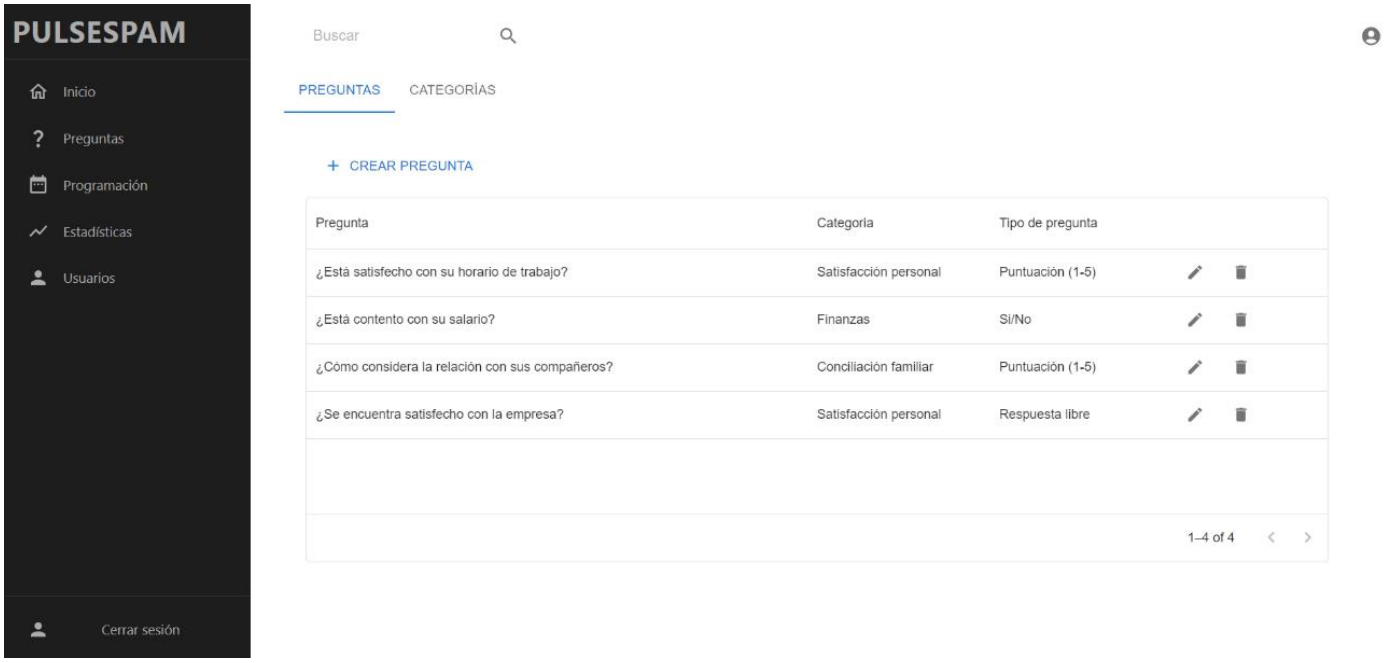


Ilustración 19: Interfaz gráfica de la aplicación web

5.3 Implementación de la aplicación de escritorio

La aplicación de escritorio tiene un conjunto de funcionalidades muy limitadas: permitir a los usuarios hacer *login* en la aplicación y registrar sus respuestas diarias, asegurando una única respuesta por cada usuario.

Tras investigar con respecto a las distintas tecnologías existentes para el desarrollo de aplicación de escritorio, se optó por WPF (*Windows Presentation Foundation*) debido a su versatilidad y sencillez. WPF combina la programación del sistema a través de clases C# con el diseño de las interfaces gráficas, utilizando documentos XAML que pueden ser editados tanto a través de texto como mediante una paleta gráfica.

Partiendo de la arquitectura definida en la Ilustración 8, cada página de la aplicación se implementó realizando primero su vista y luego su *ViewModel*, así como los posibles modelos que necesite para el mapeado de datos.

La implementación de las vistas es sencilla y está basada en una organización basada en elementos de tipo *Grid*, definiendo filas y columnas en las que situar los elementos de cada página similar a como se realizaría un diseño básico en HTML. Además, siguiendo el patrón MVVM, se han enlazado los elementos de la vista con los del *ViewModel* a través de *Bindings*.

Un Binding puede realizarse tanto con una propiedad de C# (para representar su estado y actualizarlo) o a un comando que ha de ser ejecutado, por ejemplo, al pulsar un determinado elemento.

```
<TextBox Name="txtEmail"
         Text="{Binding Email, UpdateSourceTrigger=PropertyChanged}"
         FontSize="14"
         Height="24"
         VerticalAlignment="Center"
         BorderThickness="0,0,0,2">
</TextBox>
            <Button Name="btnLogin"
                    Command="{Binding LoginCommand}"
                    Content="Iniciar sesión"
                    Width="150"
                    Height="20"
                    Cursor="Hand"
                    Margin="0,40,0,0"
                    Grid.Column="2"/>
```

Ilustración 20: Ejemplos de Binding a una propiedad y a un comando en LoginView.xaml

Para los View Models se han definido dos clases principales:

- BaseVM. Clase que implementa INotifyPropertyChanged, para gestionar las notificaciones de los cambios en las propiedades. La interfaz ofrece un único método que notifica que una propiedad ha cambiado, sirviendo de clase raíz para la herencia del resto de View Models para asegurar la actualización de los elementos de la vista en base al VM.
- DelegateCommand. Clase que implementa ICommand y permite ejecutar los comandos y comprobar si pueden ejecutarse.

```
public class DelegateCommand : ICommand
{
    4 referencias
    public Action CommandAction { get; set; }

    3 referencias
    public Func<bool>? CanExecuteAction { get; set; }

    0 referencias
    public void Execute(object? parameter)
    {
        if (CommandAction != null)
        {
            CommandAction();
        }
    }

    0 referencias
    public bool CanExecute(object? parameter)
    {
        return CanExecuteAction == null || CanExecuteAction();
    }

    public event EventHandler? CanExecuteChanged
    {
        add { CommandManager.RequerySuggested += value; }
        remove { CommandManager.RequerySuggested -= value; }
    }

    1 referencia
    public DelegateCommand(Action commandAction, Func<bool> canExecuteAction)
    {
        CommandAction = commandAction;
        CanExecuteAction = canExecuteAction;
    }

    2 referencias
    public DelegateCommand(Action commandAction)
    {
        CommandAction = commandAction;
    }
}
```

Ilustración 21: Código de la clase DelegateCommand

Tal y como se observa en la Ilustración 21, la clase DelegateCommand cuenta con dos propiedades que encapsulan el método a ejecutar por el comando y una comprobación que validar si se puede o no ejecutar el método (como comprobar que se han introducido datos en la vista y los campos no están vacíos). Estas propiedades se lanzan desde los métodos Execute y CanExecute respectivamente. Además, hay un método CanExecuteChanged que gestiona si se ha cambiado la condición de ejecución del comando, a través del gestor de comandos de Windows.

De esta forma, cada VM hereda de BaseVM y define sus atributos privados y las respectivas propiedades que permiten acceder a ellos, con una pareja *getter/setter* trivial pero que además notifica que se ha cambiado la propiedad a través del método OnPropietyChanged (heredado de BaseVM).

Además de estas propiedades, se definen los comandos de los que hace uso cada vista. Por ejemplo, para LoginVM se define un comando que permite iniciar sesión.

```
        public ICommand LoginCommand { get; }

        public LoginVM()
        {
            LoginCommand = new DelegateCommand(ExecuteLoginCommand,
CanExecuteLoginCommand);
        }
```

Este comando realiza dos funciones. En primer lugar, valida que los campos de email y contraseña contengan un texto válido, bloqueando el elemento asociado de no ser así. Cuando los dos campos contienen un texto correcto y se llama al comando de login al clicar el botón para iniciar sesión, se ejecuta el funcionamiento interno del comando según lo definido en el modelo, que conecta con el controlador de autentificación definido en el servicio web, permitiendo o no el inicio de sesión en el sistema.

Tras el proceso de *login*, se comprueba si el usuario ha contestado ya a su pregunta diaria y se le redirige a la ventana indicada según lo indicada en la Tabla 9.

Una vez completada la implementación de la aplicación de escritorio, se valoraron las distintas posibilidades para poder programar su lanzamiento de forma automática. Idealmente, se planeaba desarrollar un servicio de Windows que mantuviese la aplicación de escritorio activa en segundo plano y, al llegar la hora indicada por el usuario, lanzase el ejecutable permitiendo la respuesta diaria.

Sin embargo, la realización de un servicio de Windows no es trivial. Implica el desarrollo del servicio como tal y su posterior configuración dentro del sistema, asegurando en este caso su correcta conexión tanto con la aplicación de escritorio como el servicio. Debido a la falta de experiencia en este campo se optó por una alternativa más sencilla: tareas programadas de Windows.

El programador de tareas de Windows es una herramienta que permite definir tanto horarios como conjuntos de restricciones que, en caso de cumplirse, permiten que se ejecute de forma automática una o varias aplicaciones en el equipo.

En el caso de este proyecto, se trata de una tarea que ejecuta la aplicación de escritorio todos los días a una hora determinada (las 12 de la mañana). Esto podría generar posibles

problemas de escalabilidad en el ámbito empresarial dado que no es una alternativa tan completa como la realización de un servicio, habiendo de programarse manualmente en cada equipo que quiera hacer uso de la aplicación. Sin embargo, sería posible generar un script mediante comandos que cree la tarea programada de forma idéntica en todos los equipos de la empresa, por ejemplo, si están conectados a una red propia. Para ello, Windows ofrece los comandos de la familia schtasks [3].

De esta forma, se define la tarea según lo indicado en la ilustración 22.

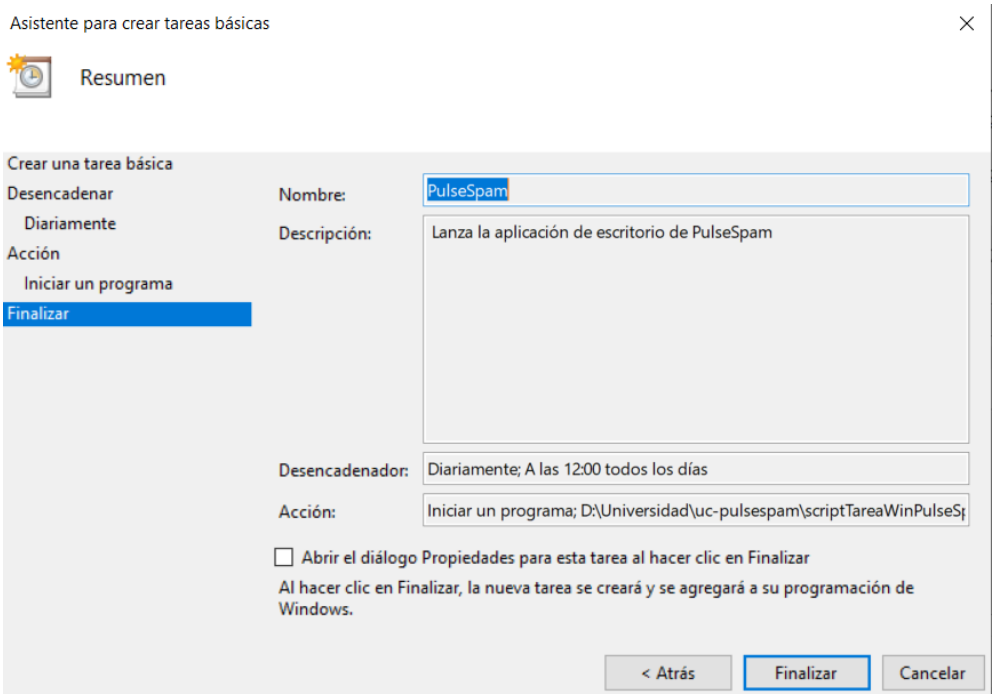


Ilustración 22: Tarea para la aplicación de escritorio

6. Seguridad

Al tener dos aplicaciones de usuario distintas cuyo acceso es dependiente del tipo de usuario, es necesario contar con un sistema para gestionar el registro y *login* de estos usuarios en el sistema. Dado que el sistema de inicio de sesión y permisos entre las aplicaciones y el servicio están íntimamente ligados, en esta sección se detalla cómo se ha realizado la seguridad y validación de usuarios para el sistema.

Si bien existen múltiples formas de gestionar el acceso, se ha optado por utilizar JWT *(Javascript Web Token)* por ser uno de los estándares de seguridad más extendidos actualmente.

JWT está basado en token, un *string* identificativo que gestiona el acceso al sistema de un usuario, incluyendo distintos datos como sus posibles roles, la fecha de creación del mismo o su caducidad.

El funcionamiento de JWT se puede observar a continuación

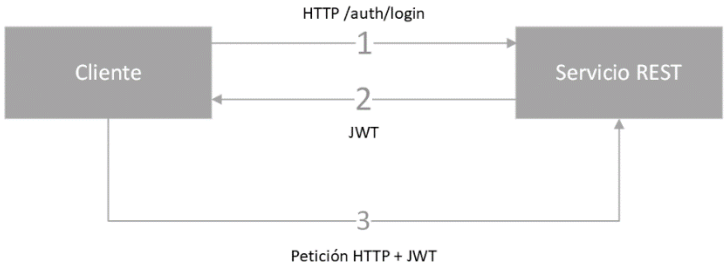


Ilustración 23: Diagrama JWT

El cliente solicita el token al sistema (en este caso, al mismo servicio que gestiona el resto de la aplicación). El sistema comprueba que el usuario está registrado con el usuario y contraseña indicados y, según su rol, permite o no el acceso a cada aplicación: los usuarios administradores acceden a la aplicación web y los usuarios convencionales a la de escritorio.

Poseyendo este token, el usuario ya puede acceder al resto de páginas y opciones del sistema, siempre dentro de las limitaciones que establece su rol. Para ello, todas sus peticiones al servicio deberán incluir el token recibido en la cabecera *Authorization* de HTTP.

Cuando se produce un cierre de sesión en el sistema, se borra el token asociado al usuario, quien deberá volver a iniciar sesión para obtener un nuevo token.

6.1 Servicio

Para la seguridad del servicio se ha realizado una implementación de JWT basada en uso de Microsoft Identity para la conexión con la base de datos. Se trata de una API disponible en ASP.NET que permite gestionar cuestiones relacionadas con el login y el registro de usuarios, así como sus distintos roles.

Dado que la implementación original de Microsoft está diseñada para su uso con Microsoft SQL Server se ha utilizado una adaptación de Identity llamada AspNetCore.Identity.MongoDbCore, que permite conectar el servicio a una base de datos

MongoDB, asegurándose de encriptar datos relevantes como la contraseña de cada usuario.

A nivel del servicio web, la seguridad en el sistema está gestionada a través de AuthController. Este controlador ofrece métodos para gestionar el login, registro en el sistema y la creación de roles para usuarios.

Los usuarios están gestionados en base a dos clases:

- **UsuarioLogin**. Clase reducida DTO que cuenta con el nombre de usuario y la contraseña y que, como su nombre indica, gestiona el sistema de login.
- **ApplicationUser**. Estas clases forman parte de la implementación de Identity realizada. Se encarga del mapeo con la base de datos, con todos los campos con los que cuenta un usuario de la aplicación y, adicionalmente, los definidos en la implementación básica de Identity.

Adicionalmente, la clase ApplicationRole es el equivalente para roles de ApplicationUser, encargada de incluir el rol de usuario (USER, ADMIN).

Cuando el usuario trata de iniciar sesión en el sistema, se verifica que el usuario realmente existe en el sistema a través de la clase UserManager de Identity, así como que su contraseña es correcta. Hecho esto se comprueba si el usuario que está accediendo es un administrador a través de un booleano incluido en la propia petición, para gestionar la división de acceso entre usuarios básicos y administradores y se genera el respectivo JWT a través los *claims* indicados, incluyendo los roles.

El registro funciona de forma similar, creando un ApplicationUser con todos los datos relevantes cuando el registro es correcto e incluyendo el rol indicado.

Una vez realizada la implementación de seguridad con JWT, se ha incluido la anotación [Authorize] en todos los controladores para asegurar que solo se puedan acceder cuando se cuente con el token adecuado. La única excepción a esto es el controlador encargado de registrar e iniciar sesión, cuyo acceso es libre a todos los usuarios.

6.2 Aplicación web

Nada más acceder a la aplicación web, se muestra una página de inicio de sesión a los usuarios. Mediante su nombre de usuario y contraseña se realiza una petición HTTP para validar al usuario mediante el servicio, obteniendo su token en caso de que sea correcto y redireccionando a la página inicial.

Este token se guarda localmente en *localStorage*. LocalStorage es un mecanismo para guardar datos localmente dentro del navegador local, permitiendo acceder al token en cualquier momento. Para cerrar sesión, únicamente es necesario eliminar el token de este *localStorage*, forzando al usuario de vuelta a la página de inicio de sesión para que vuelva a introducir sus credenciales.

De igual forma, todas las páginas de la aplicación web están ocultas para los usuarios que no estén logueados en el sistema. Para conseguir esto, se ha modificado el archivo inicial de la aplicación, App.tsx, según lo indicado en el apartado 5.3.
De esta forma, independientemente de a qué página del sistema se trate de acceder, si el usuario no está autenticado siempre será redirigido a la página de *login*.

Cuando el usuario inicia sesión correctamente, el sistema se introduce su token en el *localStorage* del sistema, permitiendo el acceso al resto del sistema. Esto puede comprobarse a través de las Chrome DevTools, en la pestaña *Application*. Accediendo al *localStorage* asociado a la página de PulseSpam, se puede comprobar que el usuario ha iniciado sesión correctamente, mostrando su email y el JWT asociado.

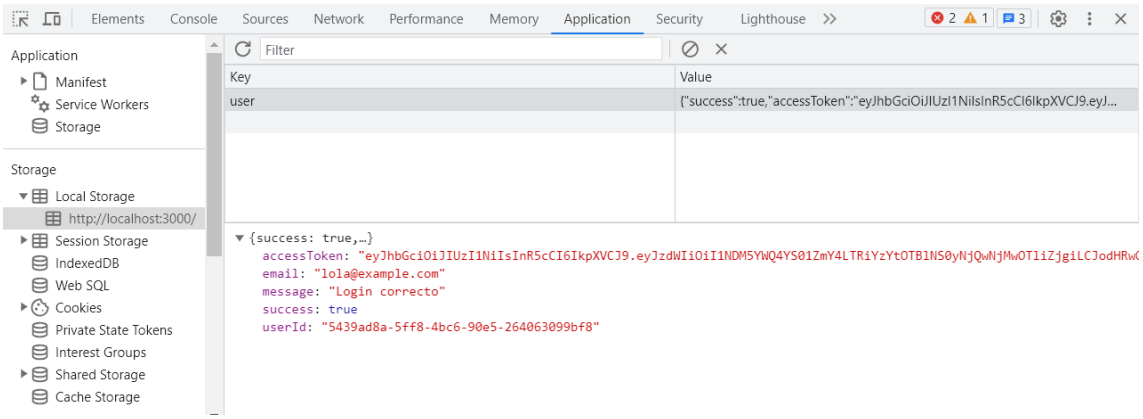


Ilustración 24: JWT asociado al usuario

6.3 Aplicación de escritorio

De forma similar a la aplicación web, la aplicación de escritorio está diseñada en base al *login* del usuario en el sistema. En el formulario de inicio de sesión se solicitan los datos del usuario (email y contraseña) y se valida su acceso mediante el servicio REST, tal y como se ha explicado anteriormente.

De esta forma, solo se permite mostrar una nueva ventana en la que introducir la respuesta para los usuarios que hayan iniciado sesión. Tras esto, el usuario puede escoger si responder o no la pregunta y se muestra una pantalla que marca el fin de la aplicación.

Tal y como la aplicación web hacía uso del *localStorage* para almacenar el token obtenido del servicio, la aplicación de escritorio guarda dicho token y el email del usuario en la configuración de la aplicación. Se trata de un fichero que almacena parejas clave valor según un tipo de dato definido, a las que se les da valor desde la propia aplicación, tras un *login* exitoso.

```
//Se guarda el token y el usuario en los settings locales
Properties.Settings.Default.Token = token;
Properties.Settings.Default.Usuario = usuario.Email;
Properties.Settings.Default.Save();
```

Ilustración 25: Extracto de ExecuteLoginCommand, de LoginVM

Una vez el usuario ha respondido a la pregunta diaria y cierra la aplicación, estas propiedades se borran automáticamente, sirviendo como cierre de sesión e instando al usuario a que vuelva a iniciar sesión en la próxima ejecución diaria.

7. Pruebas

Antes de comenzar a realizar las pruebas se realizó un plan de pruebas (disponible en el Anexo 1) en el que se describe todos los casos de prueba a llevar a cabo para considerar que el proyecto es correcto.

Puesto que los distintos elementos del sistema utilizan *frameworks* muy distintos para la fase de pruebas, esta sección se ha estructurado en base a los dos elementos más destacados del sistema (servicio REST y aplicación web) y sus respectivas pruebas unitarias y de integración, así como las pruebas de aceptación definidas para el proyecto en su totalidad.

7.1 Pruebas del servicio

Para probar el servicio web se han definido pruebas a nivel unitario y de integración. Dada la extensión del propio servicio, realizar pruebas unitarias de todos los métodos de todos los controladores existentes resulta complejo y tedioso por lo que se ha optado por hacer pruebas unitarias de los controladores con mayor complejidad técnica del sistema.

En la Ilustración 21 se muestra un ejemplo del *test* que verifica la correcta obtención de una tarea mediante su id.

```
public void GetIdTest()
{
    //Arrange
    Pregunta pregunta1 = new Pregunta();
    pregunta1.Id = ObjectId.GenerateNewId().ToString();
    pregunta1.PreguntaTxt = "Pregunta1";

    Pregunta pregunta2 = new Pregunta();
    pregunta2.Id = ObjectId.GenerateNewId().ToString();
    pregunta2.PreguntaTxt = "Pregunta2";

    TareaProgramacion tarea = new TareaProgramacion();
    tarea.Id = "T001";
    tarea.FechaInicio = DateTime.Now;
    tarea.FechaFin = DateTime.Now.AddDays(1);
    tarea.PreguntasId = new string[] { pregunta1.Id, pregunta2.Id };

    List<Pregunta> preguntas = new List<Pregunta>
    {
        pregunta1,
        pregunta2
    };
    TareaProgramacionTransfer tareaTransfer = new TareaProgramacionTransfer(tarea, preguntas);

    mockTareaService.Setup(x => x.GetAsync(tarea.Id)).Returns(Task.FromResult(tarea));
    mockPreguntaService.Setup(x => x.GetAsync(pregunta1.Id)).Returns(Task.FromResult(pregunta1));
    mockPreguntaService.Setup(x => x.GetAsync(pregunta2.Id)).Returns(Task.FromResult(pregunta2));

    //Act
    ActionResult<TareaProgramacionTransfer> programacionAction = sut.Get(tarea.Id).Result;
    TareaProgramacionTransfer programacion = programacionAction.Value;

    //Assert
    var object1Json = JsonConvert.SerializeObject(programacion);
    var object2Json = JsonConvert.SerializeObject(tareaTransfer);

    output.WriteLine("Esperado: " + object1Json);
    output.WriteLine("Actual: " + object2Json);

    Assert.Equal(object1Json, object2Json);
}
```

Ilustración 26: Pruebas unitarias del servicio

Para probar el servicio web a nivel de integración con la base de datos se ha utilizado Postman, cubriendo los casos de prueba definidos en el plan de pruebas disponible en el Anexo 1. Las pruebas en Postman están realizadas a través de distintas peticiones HTTP al servidor, incluyendo en la respectiva cabecera *Authorization* de cada una el token que permite el acceso al servicio.

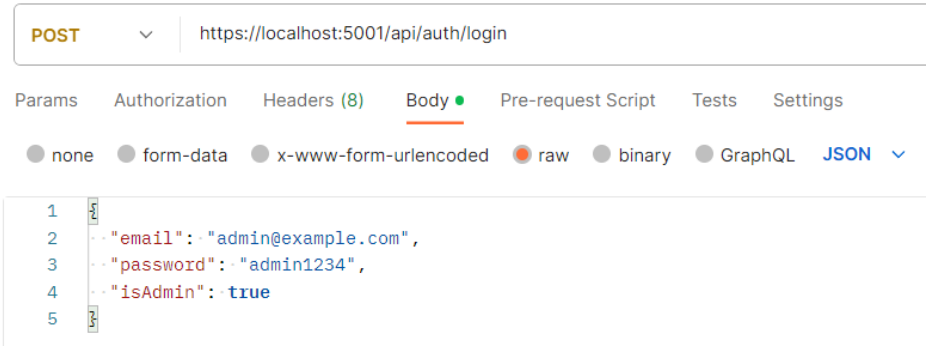


Ilustración 27: Pruebas de integración con Postman

Estas pruebas han permitido solucionar, entre otros, un error que solo permitía el uso del servicio por usuarios de tipo administrador. A fin de no complicar innecesariamente el servicio con las verificaciones de rol para todos los métodos definidos, se ha optado por gestionar el rol desde el propio login, de forma que las peticiones procedentes de la aplicación web incluyan un booleano isAdmin que indica al servicio que debe comprobar el rol del usuario.

7.2 Pruebas de la aplicación web

Debido a la propia naturaleza de la aplicación web y la fuerte conexión entre las capas del sistema, no cabe la posibilidad de realizar pruebas unitarias aisladas de la web, por lo que se ha optado por testear su integración con el servicio REST (y, consecuentemente, con la base de datos).

Las pruebas de integración de la aplicación web han sido realizadas mediante SeleniumIDE, capturando la pantalla y comprobando los elementos que aparecen en base a los casos de prueba definidos en el plan de pruebas del Anexo 2. En la Ilustración 23 se muestra un ejemplo de prueba que verifica el correcto inicio de sesión de un usuario administrador.

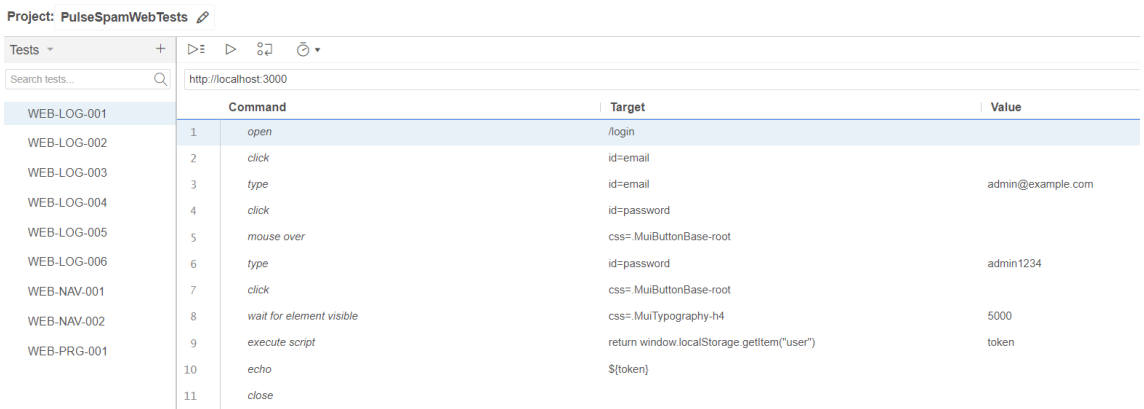


Ilustración 28: Pruebas de integración de la aplicación web con SeleniumIDE

Los errores más destacados que se han encontrado son los siguientes:

- Se permitía el login a los usuarios normales. Esto ha sido corregido, forzando a que el servicio verifique que el usuario que trata de iniciar sesión desde la web es siempre un administrador.
- Se podía crear elementos con campos vacíos. Por ejemplo, un usuario sin email. El usuario no llegaba a crearse como tal en el sistema dado que no puede contener elementos vacíos, pero no se notificaba visualmente al usuario del error. Esto se ha solucionado a través de un mensaje de error y validando los campos de cada formulario.

7.3 Pruebas de aceptación

Las pruebas de aceptación del sistema se han llevado a cabo probando todas las funcionalidades esperadas por el cliente, atendiendo a las historias de usuario definidas. De esta forma, se ha comprobado manualmente mediante ambas aplicaciones el buen funcionamiento de sistema.

8. Conclusiones y trabajo futuro

El objetivo de este proyecto era desarrollar una aplicación para la gestión y realización de preguntas diarias en una organización. Considero que los requisitos principales del sistema han sido cumplidos puesto que existen todas las funcionalidades básicas que se buscaban y, además, se han identificado posibles mejoras para el futuro.

Mediante la aplicación de una metodología ágil se ha creado un *backlog* del proyecto en el que, en forma de historias de usuario, se han identificado las necesidades del sistema. En *sprints* de dos semanas, se han llevado a cabo los distintos diseños e implementaciones del proyecto de forma iterativa, comenzando por el análisis del dominio del sistema.

A partir de este dominio se ha creado una base de datos no relacional en MongoDB, accesible por un servicio REST encargado de la gestión de los datos. En lo que a las aplicaciones respecta se ha desarrollado un cliente de escritorio que recoge las respuestas diarias de un usuario y una aplicación web administrativa que permite gestionar el sistema, añadiendo preguntas y programándolas.

Cabe destacar la complejidad de proyecto por las tecnologías empleadas para su desarrollo. Al comienzo de este TFG, la única tecnología que conocía y era capaz de usar era .NET, si bien siempre lo había utilizado de cara a aplicaciones orientadas a objetos muy sencillas. De igual manera, mi experiencia con servicios es muy limitada, especialmente para uno del tamaño y la complejidad del definido en este proyecto.

Ha sido mi primer acercamiento tanto al desarrollo de aplicaciones de escritorio en Windows con una arquitectura MVVM, así como al desarrollo web mediante React. Este último, así como me ha resultado especialmente interesante a nivel personal, también ha tenido una complejidad mucho más alta de lo que había pensado inicialmente, debido tanto a la sintaxis de Typescript como al sistema basado en componentes. Además, nunca había utilizado bases de datos no relacionales y esto ha resultado una dificultad adicional para el manejo de los datos.

Como posibles ampliaciones sobre el proyecto y trabajo futuro cabría principalmente la implementación de ciertas funciones claves para el sistema, como el registro de usuarios desde la aplicación web a través de un formulario de registro, similar al de *login*.

Otra funcionalidad que considero esencial para mejorar el sistema sería una mejora de la página de estadísticas sobre las respuestas, mostrando mejores gráficos que permitan resumir los resultados, así como observar distintas métricas relevantes sobre los usuarios. Poder filtrar a los usuarios según el departamento de la empresa al que pertenecen para poder localizar cuales son los departamentos con mejores o peores respuestas podrían permitir a la empresa determinar donde necesita centrar su atención y actuar en consecuencia y considero que sería muy positivo.

Por último, la versión actual de PulseSpam gestiona la aparición de la aplicación de escritorio para las respuestas en base a una tarea de Windows, con una hora fija todos los días (12:00). Puesto que la página web permite definir una hora concreta para cada tarea, podría cambiarse esta planificación sobre la aplicación a un servicio de Windows que controle la aparición de las preguntas según ese horario definido, para hacer al sistema mucho más flexible y adaptable.

A nivel personal y académico, este proyecto ha sido el contacto real con la ingeniería del software que considero que necesitaba para dar por completada mi formación. He podido experimentar con gran cantidad de tecnologías actuales que se utilizan en el mundo laboral con las que no tenía ninguna experiencia y que de seguro me serán útiles en el futuro, así como probar distintas arquitecturas en el diseño.

No es hasta que he tenido la oportunidad de desarrollar un proyecto de este tamaño que realmente me he dado cuenta de la importancia de todas las fases del proceso de desarrollo, así como del coste que tiene cada una. Comparando los requisitos iniciales de la aplicación y las ideas que yo misma quería incluir en el sistema con la que es su versión final me ayuda a afianzar más que nunca la importancia de saber establecer límites entre lo que ha sido pactado para un determinado proyecto y lo que, idealmente, con tiempo y recursos infinitos, se querría hacer.

Ha sido una experiencia muy positiva que me ha hecho valorar y poner en perspectiva los conocimientos adquiridos durante el Grado, así como mejorar mis habilidades.

Bibliografía

[1] J. P. a. M. V. V. Karina Van De Voorde, «Predicting business unit performance using employee surveys: monitoring HRM-related changes,» *Human Resource Management Journal*, vol. 20, nº 1, pp. 44-63, 2010.

[2] IBM, «Propiedades ACID,» [En línea]. Available: https://www.ibm.com/docs/en/cics-ts/5.4?topic=processing-acid-properties-transactions.

[3] Microsoft, «Comandos de Windows. Schtasks,» [En línea]. Available: https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/schtasks.

[4] MUI, «Página de Material UI,» [En línea]. Available: https://mui.com/material-ui/getting-started/overview/.

[5] SeleniumDev, «Documentación oficial de SeleniumIDE,» [En línea]. Available: https://www.selenium.dev/selenium-ide/docs/en/introduction/getting-started.

[6] Microsoft, «Documentación oficial de ASP.NET,» [En línea]. Available: https://learn.microsoft.com/en-us/aspnet/core/.

[7] React, «Página oficial de React,» [En línea]. Available: https://react.dev/.

[8] MongoDB, «Página oficial de MongoDB,» [En línea]. Available: https://www.mongodb.com/.

[9] Mozilla Corporation, «Documentación de Mozilla Corporation para desarrollo web,» [En línea]. Available: https://developer.mozilla.org/en-US/docs/Web.

[10] Microsoft, «Documentación oficial de Entity Framework,» [En línea]. Available: https://learn.microsoft.com/en-us/ef/.

[11] «Página de HappyForce,» [En línea]. Available: https://myhappyforce.com/es/mide-2/.

[12] Microsoft, «The Model-View-ViewModel Pattern,» [En línea]. Available: https://learn.microsoft.com/en-us/xamarin/xamarin-forms/enterprise-application-patterns/mvvm.

Anexo 1: Diagrama de Gantt del proyecto

Ilustración 29: Diagrama de Gantt I

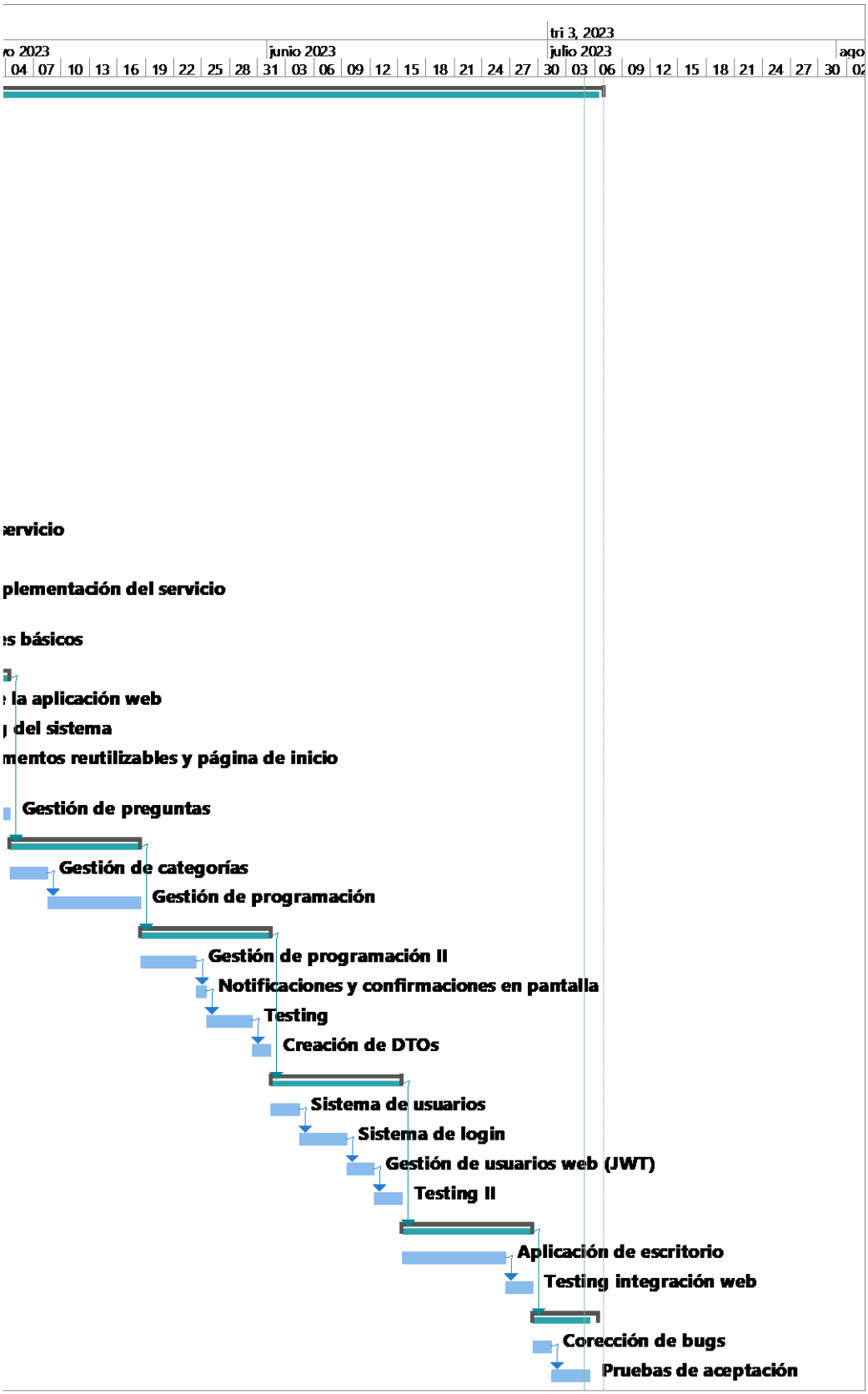


Ilustración 30: Diagrama de Gantt II

Anexo 2: Plan de Pruebas

Este plan de pruebas está estructurado en base a los tipos de pruebas principales en el desarrollo software: pruebas unitarias, de integración y de aceptación.

Debido a la extensión de las pruebas y a fin de no alargar innecesariamente este documento, se muestran en las Tablas 11-15 algunos de los *tests* definidos para el sistema.

Pruebas unitarias
Se han realizado pruebas unitarias en el servicio para los siguientes casos:

- Controlador de TareaProgramacion.

Para las pruebas unitarias se ha utilizado el framework xUnit de C# junto a la librería Moq para la creación de mocks.

Pruebas de integración
Se han realizado pruebas de integración en el servicio para los siguientes casos:

- Registro, login (usuario, administrador), creación de rol
- Controladores de Usuario, Schedule, Respuesta, Tracking, Categoria, Pregunta y Tarea

Además, se han realizado pruebas de integración para la aplicación web para los siguientes casos:

- Login
- Routing
- Creación de preguntas
- Creación de usuarios

Pruebas de aceptación
Aunque estas pruebas han de ser realizadas por el cliente, también se han llevado a cabo tras el proceso de desarrollo para verificar que todas las implementaciones funcionaban correctamente.

Se han aplicado pruebas de aceptación al sistema completo, usando tanto la aplicación de escritorio como la aplicación web, en base a los siguientes casos:

1. Pruebas unitarias

TareaProgramacionController

Se han realizado pruebas unitarias para probar el funcionamiento de los métodos descritos en TareaProgramacionController.

2. Pruebas de integración

Servicio web

Las pruebas de integración del servicio web han sido realizadas mediante Postman.

Aplicación web

Para las pruebas de integración de la aplicación web se parte de la existencia de dos usuarios administrador y usuario normal según se define a continuación.

Email	**Contraseña**	**Rol**
Admin@example.com	Admin1234	ADMIN
lola@example.com	Lola1234	USER

Tabla 11: Usuarios de prueba

Todas las pruebas de integración están identificadas con un id tal que WEB-{página}-000, definidas en distintas colecciones según la página de la aplicación que se está probando. El código creado para las pruebas puede consultarse en el fichero PulseSpamWebTests.side

Id	**Entrada**	**Resultado**
WEB-NAV-001	Acceder a /preguntas sin login	Se redirecciona a /login
WEB-NAV-002	Acceder a /preguntas después de login	Se redirecciona a /preguntas

Tabla 12: Pruebas de integración (Navegación)

Id	**Entrada**	**Resultado**
WEB-LOG-001	Login correcto	Se muestran la página de inicio de PulseSpam
WEB-LOG-002	Login incorrecto (usuario no admin)	Se muestra una alerta con el texto “Permisos insuficientes. Contacta con un administrador”
WEB-LOG-003	Login incorrecto (email de admin incorrecto)	Se muestra una alerta con el texto “Email o contraseña incorrectos”
WEB-LOG-004	Login incorrecto (contraseña de admin incorrecta)	Se muestra una alerta con el texto “Email o contraseña incorrectos”
WEB-LOG-005	Login incorrecto (ambas credenciales incorrectas)	Se muestra una alerta con el texto “Email o contraseña incorrectos”
WEB-LOG-006	Login correcto con cierre de sesión	Se muestran la página de inicio de PulseSpam. Tras cerrar sesión, se muestra la página de inicio.

Tabla 13: Pruebas de integración (Navegación)

Id	Entrada	Resultado
WEB-P-001	Añadir pregunta	Se muestran los datos introducidos en la tabla de preguntas
WEB-P-002	Añadir pregunta sin categoría	Se muestra una alerta de error en la parte inferior
WEB-P-003	Añadir pregunta sin tipo	Se muestra una alerta de error en la parte inferior
WEB-P-004	Actualizar categoría de pregunta	Tras refrescar la página, se muestra la pregunta actualizada con la nueva categoría
WEB-P-005	Actualizar tipo de pregunta	Tras refrescar la página, se muestra la pregunta actualizada con el nuevo tipo
WEB-P-006	Borrar pregunta	Se muestra un mensaje de confirmación indicando que se ha realizado el borrado

Tabla 14: Pruebas de integración (Navegación)

Id	Entrada	Resultado
WEB-USR-001	Crear nuevo usuario	Tras refrescar la página, se muestran los datos introducidos en la tabla de usuarios
WEB-USR-002	Crear nuevo usuario administrador	Tras refrescar la página, se muestran los datos introducidos en la tabla de usuarios
WEB-USR-003	Crear usuario incorrecto (sin contraseña)	Se muestra una alerta de error en la ventana de creación de usuario
WEB-USR-004	Crear usuario incorrecto (sin email)	Se muestra una alerta de error en la ventana de creación de usuario
WEB-USR-005	Crear usuario incorrecto (sin nombre de usuario)	Se muestra una alerta de error en la ventana de creación de usuario

Tabla 15: Pruebas de integración (Navegación)