VNiVERSiDAD D SALAMANCA

TRABAJO DE FIN DE GRADO

GRADO EN INGENIERÍA INFORMÁTICA

FACULTAD DE CIENCIAS

2Code: Aplicación web para evaluar conocimientos de programación

MEMORIA DEL PROYECTO

Autor:

Fabián Villalobos Cayoja

Tutoras:

Alicia García Holgado

Andrea Vázquez Ingelmo

SALAMANCA

Septiembre de 2022

Dª. Alicia García Holgado, profesora del Departamento de Informática y Automática de la Universidad de Salamanca

Dª. Andrea Vázquez Ingelmo, profesora del Departamento de Informática y Automática de la Universidad de Salamanca

CERTIFICAN:

Que el trabajo titulado "2Code: Aplicación web para evaluar conocimiento de programación" ha sido realizado por D. Fabián Andrés Villalobos Cayoja, con Pasaporte 4881147 y constituye la memoria del trabajo realizado para la superación de la asignatura Trabajo de Fin de Grado de la Titulación Grado en Ingeniería Informática de esta Universidad.

Y para que así conste a todos los efectos oportunos.

En Salamanca, a 7 de Septiembre de 2022

GARCIA HOLGADO ALICIA - 70893684W

Firmado digitalmente por GARCIA HOLGADO ALICIA - 70893684W
Fecha: 2022.09.07 10:05:27 +02'00'

Dª. Alicia García Holgado
Dpto. Informática y Automática
Universidad de Salamanca

Firmado digitalmente por VAZQUEZ INGELMO ANDREA - 70918831X
Fecha: 2022.09.07 10:44:35 +02'00'

Dª. Andrea Vázquez Ingelmo
Dpto. Informática y Automática
Universidad de Salamanca

Resumen

Hoy en día es muy fácil encontrar cientos de recursos e información sobre programación en internet. Es ligeramente más difícil encontrar un compilador de código en línea, pero los hay. Y es aún más difícil encontrar un servicio web que no sólo tenga un repositorio de ejercicios de programación pero que te permita escribir el código en la misma página web y evalúe su resultado. Sin embargo, sí existen algunas páginas web que proponen un servicio similar como, por ejemplo, Leetcode [1] o Hackerrank [2].

Este tipo de aplicaciones web son populares por contener clásicos ejercicios de programación que demandan y fortalecen conocimientos muy requeridos por las empresas, sin embargo, no suelen ser de acceso libre y gratuito debido a la dificultad que conlleva crear y mantener una aplicación de este tipo. En ambos ejemplos mencionados, la mayoría del contenido se encuentra detrás de una barrera de pago o es de acceso exclusivo. Es muy difícil encontrar una aplicación web que tenga estas características y sea gratuita, y es así que nace la idea de este proyecto.

Con 2Code se busca crear una aplicación web que permita a sus usuarios resolver problemas de programación en distintos lenguajes de programación y, poder compilarlos y probarlos en la misma página sin necesidad de ejecutarlos localmente.

El objetivo de este trabajo es construir una aplicación web que permita a sus usuarios poner a prueba sus conocimientos de programación evaluando sus soluciones a diferentes problemas de programación. Para ello, se pondrá a disposición de los usuarios una lista de problemas clasificados por nivel de dificultad. Los usuarios escogen el problema que desean resolver y escriben su solución en uno de los lenguajes de programación disponibles en la aplicación, que será enviada a un servidor remoto para ser compilada y testeada. Para que la solución propuesta sea aceptada, tendrá que pasar una serie de pruebas unitarias definidas previamente. Además, el servidor guardará la información de los problemas de programación y las soluciones propuestas por el usuario en una base de datos relacional.

Para poder hacer frente a un proyecto como este, se ha utilizado una mezcla entre los marcos de desarrollo software del Proceso Unificado y la metodología ágil de desarrollo de aplicaciones, obteniendo y adaptando de estos los mejores aspectos para un proyecto de desarrollo web *fullstack*.

Con la creación de 2Code se obtiene un software que podría llegar a utilizarse como una base a partir de la cual empresas, universidades, organizaciones o individuos puedan tener su propia aplicación web que evalúe el código enviado por sus usuarios y que incluya su propia base de datos de preguntas.

Mantener una aplicación de este tipo no es tarea fácil y tiene todavía algunas características que extender para poder manejar los diferentes lenguajes de programación existentes.

Palabras clave: Problemas de programación, evaluador de código, servidor remoto, compilador, testeo, pruebas unitarias, aplicación web, uso público, software libre, desarrollo *fullstack*.

Abstract

Nowadays it is very easy to find hundreds of programming resources and information on the Internet. It is slightly more difficult to find an online code compiler, but they do exist. And it is even more difficult to find a web service that not only has a repository of programming exercises but allows you to write the code on the same web page and evaluate the result. However, there are some websites that offer a similar service, such as Leetcode [1] or Hackerrank [2].

These types of web applications are popular because they contain classic programming problems that highlight knowledge highly demanded by companies, however, they are not usually free and openly available due to the difficulty involved in creating and maintaining such a type of application. In both examples, most of the content is either behind a paywall or exclusive to access. It is very difficult to find a web application that has these features and remains free, and that is how the idea of this project was born.

2Code aims to become a web application that allows its users to solve programming problems in different programming languages, and to compile and test them on the same page without having to run them locally.

The objective of this work is to build a web application that allows its users to test their programming knowledge by evaluating their solutions to different programming problems. For this purpose, a list of problems classified by level of difficulty will be made available to users. Users choose the problem they wish to solve and write their solution in one of the programming languages available in the application, which will be sent to a remote server to be compiled and tested. For the proposed solution to be accepted, it will have to pass a series of previously defined unit tests. In addition, the server will store the information of the programming problems and the solutions proposed by the user in a relational database.

To tackle a project like this, a mix between the Unified Process software development framework and the Agile application development methodology has been used. Extracting and adapting from these frameworks the best aspects for a full-stack web development project.

With the creation of 2Code we obtain a software that could be used as a base from which companies, universities, organizations, or individuals can have their own web application that evaluates the code submitted by its users and includes its own database of questions.

Maintaining such an application is no easy task and it still has some features to extend in order to handle all the different programming languages available.

Keywords: programming problems, code evaluator, judging engine, remote server, compiler, testing, unit testing, web application, public use, free software, full-stack development.

Índice

Índice de figuras

1. INTRODUCCIÓN

En los últimos años se han popularizado aplicaciones web como Leetcode [1] o AlgoExpert [3], que se hicieron conocidas por exponer clásicos problemas de programación que sirven para aprender, no solo sobre programación, sino también sobre conceptos más específicos como la programación orientada a objetos, programación dinámica, recursividad, estructuras de datos, algoritmos, *machine learning*, entre otros. Estos sitios web pertenecen a empresas privadas, por lo que mantienen en secreto el funcionamiento de su evaluador de código. Además, tanto el acceso a los ejercicios de programación como al resto de contenido depende del tipo de membresía que uno tiene, dando una disponibilidad muy limitada o ninguna a aquellos usuarios que no están dispuestos a pagar.

2Code se inspira en la existencia de páginas web que recopilan librerías de código abierto y de uso público, y también del *software* libre y sin restricciones. La idea es que cualquier usuario tenga un acceso completo a la aplicación web, evitando membresías o cualquier otro coste por su parte, que limite su acceso a los ejercicios de programación. 2Code también podría llegar a ser un *software* base a partir del cual empresas, universidades o individuos pueden crear su propia aplicación web que evalúe el código enviado por sus usuarios.

En una primera parte se realizará un análisis del problema que se intenta resolver, seguido los conceptos teóricos necesarios para abordarlo. Posteriormente, se verán las técnicas y herramientas escogidas para realizar el proyecto. También, se explicará la metodología de trabajo escogida y utilizada, además de la organización y temporización del proyecto.

A modo de anexo, se incluirá toda la documentación técnica asociada a 2Code, indicando el proceso de desarrollo de la aplicación web, el análisis del problema y las especificaciones de diseño. Asimismo, se incluirá el manual de usuario y el manual del programador.

2. OBJETIVOS

2Code tiene como objetivo principal el ser capaz de evaluar código que los usuarios envían para resolver los problemas/ejercicios de programación. Se busca que la aplicación sea capaz de compilar y evaluar el código mediante distintas pruebas unitarias que comprueban que la solución propuesta responde completamente el problema de programación. También se busca crear una plataforma que almacene, muestre y ponga a disposición de los usuarios una librería de problemas de programación. Se creará un registro de usuarios únicamente con el propósito de almacenar los intentos de resolución de problemas por su parte, creando así un historial con el resultado de las ejecuciones.

2.1. Objetivos funcionales

En resumen, los objetivos funcionales son:

- **Principal - Evaluación de soluciones**: Cada intento de solución enviada por el usuario tendrá que ser compilado en el servidor y será sometido a pruebas unitarias para comprobar que el código es una posible solución del problema. Los resultados obtenidos serán comparados con los resultados esperados definidos en las pruebas unitarias para determinar si la solución resuelve el problema.
- **Gestión de problemas**: Se tiene que manejar el alta, baja y modificación de los problemas. Estos contendrán toda la información de los problemas de programación y un conjunto de pruebas que deberán superar las soluciones a evaluar.
- **Gestión de usuarios**: Se deberá poder dar de alta un usuario y permitir que realice el inicio de sesión para utilizar la aplicación.
- **Gestión de soluciones**: Los intentos de solución a un problema deberán guardarse en el sistema, relacionando el autor de la solución y el problema que se intenta resolver para poder mantener un historial de intentos de solución de cada problema.
- **Soporte de distintos lenguajes de programación**: el evaluador de soluciones deberá estar diseñado para soportar más de un lenguaje de programación. La idea es que, con el progreso del proyecto, incluso fuera del alcance de este TFG, se soporte múltiples lenguajes de programación.

2.2. Objetivos no funcionales

En cuanto a objetivos no funcionales, se tienen los siguientes:

- **Compatibilidad**: La aplicación web deberá ser compatible con los navegadores más utilizados.
- **Usabilidad**: La aplicación web mantendrá un aspecto y navegación sencillos e intuitivos.
- **Tiempo de respuesta**: El tiempo en el que, tanto la página web como el servidor remoto, tardan en responder no debe afectar el uso de la aplicación para permanecer dentro de lo aceptable.
- **Concurrencia**: El número de usuarios que utilizan la aplicación web no debe afectar ni detener el servicio.

2.3. Objetivos personales

En la actualidad, la demanda de ingenieros informáticos que tengan conocimientos de metodologías ágiles, como Scrum [4], es muy alta y por tanto extender los conocimientos en esta metodología, así como aprender y utilizar Javascript [5] en conjuntos con librerías y marcos de trabajo para crear una aplicación web, es una buena combinación de conocimientos para acceder con más fuerza al mercado laboral. Aún si TypeScript [6] se ha convertido en el nuevo lenguaje predominante en la programación web, Javascript es el lenguaje subconjunto del cual TypeScript fue creado y por tanto el cambio de uno al otro no debería presentar mucha dificultad. Por otro lado, React [7] también es considerado una buena base para la cual continuar la adopción de marcos de trabajo más modernos, como NEXT.js [8] o Vue.js [9].

3. CONCEPTOS TEÓRICOS PREVIOS

Toda aplicación web se divide en al menos dos grandes partes: la aplicación frontal (frontend) con la que el usuario interactúa con el programa y el servidor (backend) que recibe lo datos, los procesa y/o los almacena. No todas las aplicaciones necesitan almacenar datos o procesarlos, sin embargo, este sí será el caso para 2Code.

El frontend está compuesto por todo el código que genera la interfaz gráfica de una aplicación y permite la correcta interacción entre el usuario y el programa. Aunque previamente se entendía frontend únicamente como el conjunto de ficheros HTML y CSS que generaban la interfaz gráfica en un navegador, hoy en día puede abarcar más elementos. Desde la creación de V8, el "motor de código abierto de Google para JavaScript y WebAssembly de alto rendimiento, escrito en C++" [10], es posible la ejecución de código JavaScript en todos los navegadores que la incluyen. A pesar de que V8 ha abierto la posibilidad de ejecutar toda una aplicación web del lado del cliente, los datos que se manejan son volátiles y su procesamiento es local.

Una solución a los límites de una aplicación web ejecutada directamente en el lado del cliente es la creación de un servidor con el que el frontend se comunica a través de una API (Application Programming Interface en inglés).

El backend es el conjunto de ficheros que contienen el código que procesa las solicitudes y datos, generalmente enviados desde el frontend. Este puede tener múltiples componentes: la API es uno de ellos y la base de datos es otro.

En una aplicación web, una API permite la comunicación entre dos componentes software mediante un conjunto de definiciones y protocolos. En esta interfaz se define cómo se comunican entre sí a través de solicitudes y respuestas. Además, la API se convierte en una capa de abstracción que evita que el cliente tenga que conocer la implementación de los servicios propuestos por el servidor [11], e igualmente el servidor recibirá solicitudes por el/los clientes que siguen un formato específico por lo que no necesita conocer cómo están programados.

La base de datos aloja toda la información no volátil de la aplicación web. En el caso de 2Code esto incluye información sobre los usuarios, los problemas de programación, las respuestas realizadas por los usuarios, entre otros. Existen dos tipos de bases de datos: las relacionales

y las no relacionales. La principal diferencia entre ambas es que las relacionales almacenan la información en forma de tablas que permite optimizar el tiempo que tarda encontrar un conjunto de datos que tienen algún tipo de relación entre ellos, mientras que las no relacionales almacenan los datos en un cierto formato en función del tipo de dato que se quiere guardar, facilitando su almacenaje, aunque a veces a costa de aumentar el tiempo de búsqueda. Dependiendo del tipo de aplicación web se escoge el tipo de base de datos más apropiado, aunque no se está limitado a uno de ellos ya que una aplicación puede utilizar múltiples bases de datos.

Actualmente, una aplicación que contenga estos tres componentes puede realizarse de distintas formas: gran parte de las decisiones de lenguajes de programación, marcos de trabajo y dependen del tipo de arquitectura con la que se diseña la aplicación.

Una RESTful API es una de las posibles arquitecturas con las que se puede desarrollar el backend de la aplicación. Una de las principales características de esta arquitectura es que "Las solicitudes de los clientes al servidor son similares a las URL que se escriben en el navegador para visitar un sitio web. La respuesta del servidor son datos simples, sin la típica representación gráfica de una página web" [11]. Las solicitudes del cliente se envían como transacciones HTTP y en ellas se pueden enviar datos, especificar la operación que se quiere realizar o indicar los datos que se quieren obtener. GET, POST, PUT y DELETE son cuatro de las operaciones más utilizadas que permiten obtener, crear, actualizar y borrar datos en un servidor. En una arquitectura REST, cada sección de una solicitud puede manejarse por distintos módulos, dando mucha flexibilidad a la hora diseñar la API.

Los datos procesados por una REST API pueden seguir distintos formatos de representación, aunque JSON es el que actualmente más se ha popularizado ya que facilita la transferencia de datos mediante HTTP. JSON está basado en la sintaxis de Javascript, es decir que, a pesar de ser una cadena de texto, ésta sigue el formato de presentación de objetos javascript y, por lo tanto, cualquier servidor o navegador que utilice el motor de código Javascript "V8" de Google puede, mediante una simple función, transformar un objeto javascript a una línea de caracteres en formato JSON y viceversa.

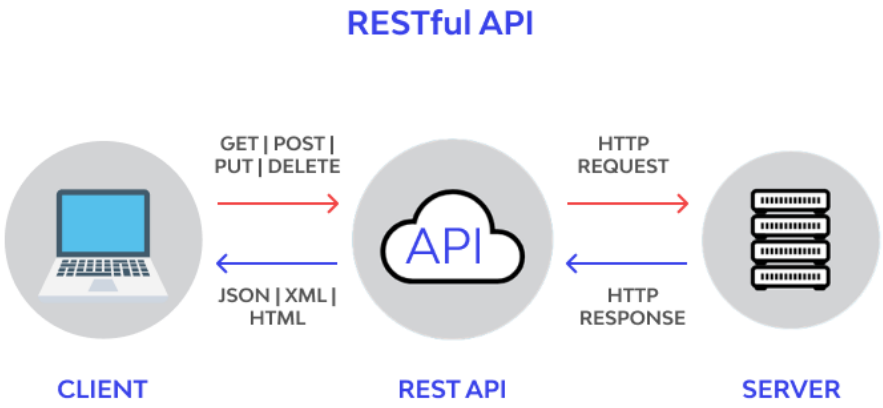


Ilustración 1: Arquitectura de una REST API - www.hakin9.org

4. INTRODUCCIÓN A LA APLICACIÓN

2Code es una aplicación *fullstack*. Esto quiere decir que la aplicación está compuesta por el frontal o interfaz de usuario, mejor conocido como *front end*, y el código que maneja el procesamiento de información y las interacciones entre la interfaz y la base de datos en el servidor, mejor conocido como *back end*. Para introducir la aplicación, es mejor comenzar por el *front end* ya que siendo un componente gráfico de la aplicación, es más fácil entender lo que se consigue realizar con este proyecto, mientras que el *back end* permite explicar el *cómo* se llega al resultado del proyecto.

Ilustración 2: Página con la librería de problemas de 2Code.

Para empezar, en la captura de pantalla anterior se puede ver la página que contiene la lista de problemas que ya se tienen en la aplicación. Cada fila de la tabla los principales datos que se muestran son el título del problema, una estimación de su dificultad y, según el usuario, si ha resuelto el problema o no.

Al realizar clic en "Resolver" o en la misma fila del problema, se accede al "*Playground*" o zona de prueba. En esta página se tiene el panel con información del problema en la parte izquierda de la pantalla y el editor de código que se extiende desde el centro hasta la parte derecha de la pantalla.

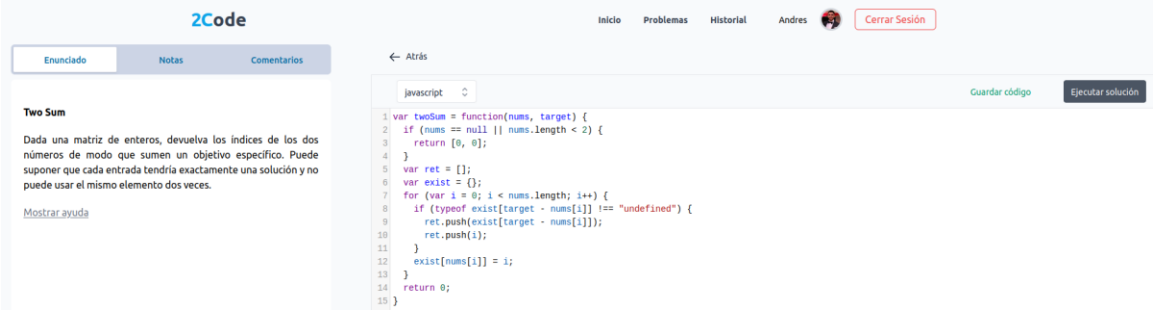


Ilustración 3: Página del problema y el editor de código.

La zona de prueba es el elemento principal de esta aplicación ya que en esta es que el usuario, a partir de la descripción del problema, selecciona el lenguaje de programación que prefiera y escribe una solución que se envía al servidor para su evaluación haciendo clic en el botón "Ejecutar solución".

El servidor realiza distintas consultas para relacionar la solución enviada con el problema y el usuario que la escribe. Si se validan estas consultas, se procede a compilar el código dependiendo del lenguaje escogido y se realizan las pruebas unitarias. Al finalizar las pruebas o al obtener un resultado distinto al esperado en una de ellas, se devuelve el resultado de la evaluación a la interfaz de usuario, que a su turno informa al usuario si el código enviado es una solución del problema o no.

Finalmente, a través de una consulta al servidor, se enseña al usuario un historial con los resultados de todos sus intentos enviados a evaluar.

Ilustración 4: Página del historial de ejecuciones del usuario.

5. TÉCNICAS Y HERRAMIENTAS

Para el desarrollo de la aplicación web se ha escogido una mezcla de técnicas de desarrollo y organización extraídas del Proceso Unificado y del Desarrollo Ágil. La mezcla entre estos dos se debe a que en general, las distintas metodologías ágiles que se han desarrollado suelen aplicarse en proyectos en los que se tiene un equipo de desarrollo con múltiples participantes. Pese a ello, existen muchas técnicas definidas en estas metodologías que se pueden extraer para un proyecto de un solo desarrollador y que permiten establecer parte de los valores y principios del desarrollo ágil de software.

En cuanto las herramientas técnicas de desarrollo, se ha escogido implementar el denomido "PERN S*tack*". Esto consiste en PostgreSQL como base de datos, Express como infraestructura de aplicaciones web, React como librería web y Node como entorno de ejecución de la aplicación. Esta es una estructura muy conocida en el desarrollo de aplicaciones web ya que utiliza software libre muy conocido y establecido.

5.1. Técnicas de desarrollo

2Code se ha desarrollado utilizando extractos de las metodologías ágiles y del Proceso Unificado, apoyándose en técnicas y herramientas de gestión de proyectos como Kanban, historias de usuario y planificación temporal "*top-down"*.

Scrum es una metodología ágil destinada a equipos de desarrollo. Pese a ello, entre los desarrolladores e ingenieros se debate si es posible aplicarlo a proyectos con un solo desarrollador. No hay una respuesta única, sin embargo, la posición general indica que no exactamente ya que gran parte de su ideología está en la gran interacción que tienen los desarrolladores entre ellos y con los *stakeholders*. En lo que muchos están de acuerdo es que sí es posible adoptar una parte de Scrum a lo que sería simplemente una metodología ágil inspirada en Scrum, y eso es lo que se realiza con 2Code.

En las metodologías ágiles se tiene la mentalidad de que los requerimientos de un proyecto son volátiles y por tanto van a existir cambios impredecibles por parte del cliente, el equipo de desarrollo o el mismo producto, por lo que una metodología basada en la predicción y planificación no va a estar suficientemente preparada para afrontar eficientemente estos

cambios. De acuerdo con Scrum, no es necesario entender el motivo de los cambios, sino esperarlos, aceptarlos y enfocar las habilidades del equipo a adaptarse al cambio de situación.

Para afrontar estos principios de impredecibilidad y adaptación, se toma de Scrum la idea de dividir el proyecto en múltiples objetivos que se pueden alcanzar mediante iteraciones de trabajo de corta y media duración. En general se intenta que estas iteraciones duren entre dos a cuatro semanas de trabajo ya que en estos cortos periodos pueden surgir problemas con la tareas o mejores ideas sobre cómo alcanza los objetivos. Dándose el tiempo de analizar el avance tras cada iteración, se puede evitar perder tiempo en modificar e incluso rehacer partes de la aplicación. Esto es el *sprint review*.

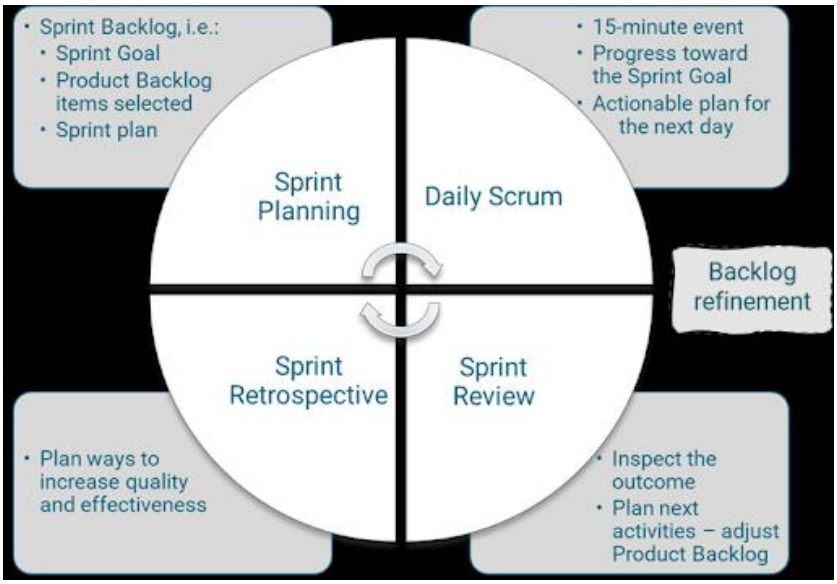


Ilustración 5: Eventos de un sprint, extraído de The 2020 Scrum Guide

Por otro lado, en Scrum se define un control diario del progreso hacia el objetivo de la iteración mediante reuniones (*daily scrum).* En un proyecto de una sola persona como 2Code, pierde el sentido realizar un control diario puesto que sólo se tiene una línea de avance hacia el objetivo de la iteración. Sin embargo, es posible reemplazar este control de progreso hacia el objetivo mediante una planificación de tareas de cada iteración. Tanto en el Proceso Unificado como en Scrum, se suele realizar una planificación temporal con las tareas de la iteración. Cada metodología tiene su propia forma de realizar esta planificación, pero la cual se ha utilizado para 2Code es la denominada "arriba hacia abajo" o "*top-down"* [12] ya que

con esta se pueden adaptar e incluir cambios inesperados de la planificación del proyecto ya que la planificación de tareas se realiza a cada iteración y no solo una única vez para todo el proyecto. Scrum también fomenta la baja dependencia y modularidad de las tareas, evitando que se tengan cascadas de dependencias entre las tareas y por tanto dando espacio a modificaciones inesperadas en la planificación temporal.

Las tareas mencionadas anteriormente se extraen a partir de historias de usuario, una técnica también extraída de las metodologías ágiles en general. El enfoque en casos de uso que se utiliza en el Proceso Unificado es una alternativa válida para este proyecto, sin embargo, teniendo en cuenta que su proceso de descripción es mucho más preciso y genera mucha documentación, es preferible seguir las directivas del manifestó ágil y limitarse a un mínimo de documentación técnica a un inicio ya que se espera que esta progrese a lo largo de proyecto. Las historias de usuario permiten definir los requerimientos del producto software manteniendo un enfoque de desarrollo centrado en el usuario y mantienen la modularidad de las tareas que conforman los entregables de cada iteración.

Una vez definidas las historias de usuario, los puntos de historia que se les asignan permiten realizar una planificación temporal basada en la estimación de puntos de usuario que el equipo de desarrollo puede realizar durante una iteración. Esta estimación nos permite calcular, y adaptar después de cada iteración, el tiempo estimado para finalizar el producto.

Tanto en Scrum como en las metodologías ágiles en general, una iteración se descompone de la siguiente forma:

Ilustración 6: Agile Methodology in System Development [13]

Uno de los aspectos más importantes de las metodologías ágiles es que no son metodologías sumamente estrictas. El equipo tiene que adaptar la metodología al proyecto, a las circunstancias y al mismo equipo, dando paso a añadir, cambiar o eliminar aspectos de éste que se vean necesarios. Es por ello por lo que la metodología de desarrollo descrita anteriormente ha sido complementada con otras técnicas de organización de proyectos como, por ejemplo, el uso de Kanban para la organización de tareas de una iteración.

Kanban es una metodología que permite visualizar el flujo de trabajo de un proyecto, o en este caso de una iteración y del proceso de desarrollo del producto [14]. Tener constantemente a la vista el flujo de trabajo a realizar permite mantener el número de tareas a abordar durante una iteración a un número asequible, en especial si se mezcla a con la técnica de asignación de puntos de historia a estas tareas. La fácil lectura de un tablero en Kanban también permite mantener en mente los objetivos de una iteración ya que al pasar mucho tiempo en la resolución de tareas más largas de lo habitual (con puntos de historia más altos), se puede perder el enfoque de las metas del *sprint* actual.

Finalmente, existen ciertas técnicas de desarrollo que son más comúnmente utilizadas en el Proceso Unificado pero que, siendo 2Code una aplicación web, permiten mayor facilidad a la hora de entender y diseñar las llamadas entre funciones que ocurren entre el cliente y el

servidor y dentro del servidor mismo. El diagrama de clases, los diagramas de secuencia, el diagrama de paquetes, la definición de la API, entre otros, son los tipos de documentos que se han utilizado para describir los datos de la aplicación, la arquitectura de la aplicación y las interacciones entre componentes. La totalidad de estos se encuentran en el Anexo I.

5.2. Lenguajes de programación

5.2.1. HTML

HyperText Markup Language es el estándar para el desarrollo de interfaces gráficas de páginas web y actualmente se sigue el estándar HTML5. En este proyecto no se utiliza directamente HTML para realizar la interfaz, pero como con otros *frameworks* de desarrollo web, React genera HTML a partir los elementos JSX para que pueda ser interpretable por el motor de V8 de Google, o equivalentes, que los navegadores web utilizan.

5.2.2. CSS

Cascading Style Sheet es el código que describe cómo los elementos HTML van a verse en el navegador. Actualmente se sigue el estándar CSS3. Para ello se crean ficheros (o hojas de estilo) en las que se definen las propiedades visuales de los elementos HTML. Estos ficheros se han utilizado en algunos de los componentes React más complejos como, por ejemplo, en las distintas tablas que enseñan las listas de problemas, las soluciones enviadas o la lista de usuarios del administrador. Para el resto de los componentes se ha utilizado Tailwindcss, un *framework* de bajo nivel que nos permitirá modificar el estilo de los elementos rápidamente en la misma propiedad de "*classNames*" de un elemento JSX sin tener que generar una hoja de estilo separada del mismo código JavaScript.

5.2.3. JavaScript

JavaScript es el lenguaje de programación principal en 2Code. Este permite implementar funciones complejas que permiten añadir un control lógico y un valor dinámico a una página web obteniendo o actualizando su contenido en tiempo de ejecución [15]. Es un lenguaje moderno que actualmente se encuentra entre los más establecidos para la programación de aplicaciones web. Habiendo sido inicialmente un lenguaje destinado al desarrollo de código que sería leído por navegadores web, hoy en día, gracias a la introducción de Node.js en 2009,

Javascript es un lenguaje para desarrollar tanto el *back end* como el *front end*. A pesar de que Javascript no está tan optimizado como PHP o GOLANG para realizar cálculos, siendo un lenguaje *fullstack*, su adopción genera menos fricción ya que elimina la necesidad de aprender otro lenguaje de programación para el *back end*.

En este proyecto se ha decido utilizar JavaScript tanto para el *front end* como para el *back en* puesto que encajan muy bien con el *stack* tecnológico escogido. En especial, Node en el *back end*, a pesar de tener TypeScript bien incorporado, tiene más documentación e información de ayuda disponible para JavaScript en internet. Además, Node utiliza el motor V8 creado por Google para compilar código Javascript, un motor optimizado y rápido.

5.2.4. JSX

Pese a parecerse a una mezcla de simple HTML con JavaScript, JSX (JavaScript XML) es una extensión de JavaScript que permite describir lo que sería la interfaz de usuario. Puede ser utilizado para producir "elementos" React de los cuales después se han creado complejos componentes de los cuales la mayoría son reutilizables (Ilustración 7). Al final, cuando se compila la aplicación, React transforma estos elementos JSX a JavaScript.

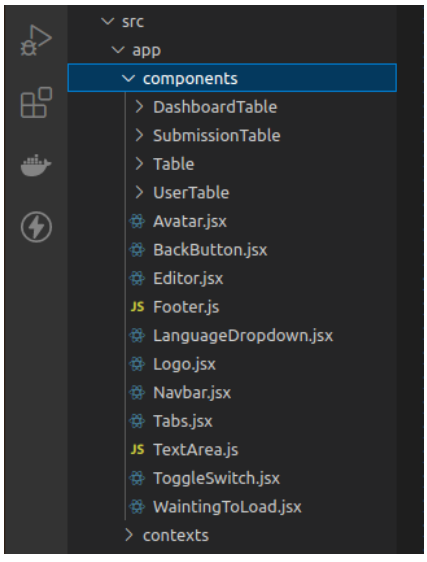


Ilustración 7: Lista de componentes de 2Code en VSCode

5.2.5. SQL

Structured Query Language es un lenguaje de programación para comunicarse con una base de datos relacional. Para la base de datos, MongoDB y PostgreSQL son dos sistemas de bases de datos muy utilizados y establecidos para la creación de aplicaciones web. Viendo que los datos de 2Code estarán relacionados entre ellos, se ha escogido PostgreSQL ya que es una base de datos relacional que soporta el estándar SQL en su gran mayoría. Para 2Code se ha creado una base de datos en PostgreSQL que contiene tres tablas que contienen la información del usuario, de los problemas de programación y de las soluciones enviadas por usuarios. Gracias a SQL es posible utilizar su sentencia "*JOIN*" para obtener información que combina datos de las tres tablas. 2Code ha dado muchas oportunidades de realizar distintas operaciones sobre la información recuperada como, por ejemplo, en la obtención de la última solución enviada por el usuario para cada problema para saber si ya ha resuelto el problema previamente.

5.3. Herramientas CASE

5.3.1. Diagrams.net

Es una aplicación web de código abierto y gratuita para generar todo tipo de diagramas, incluyendo aquellos que utilizan el lenguaje UML de modelado. Se ha utilizado para generar todos los diagramas de clase, de secuencia (Ilustración 8), de paquetes, entre otros, que se encuentra en el Anexo I.

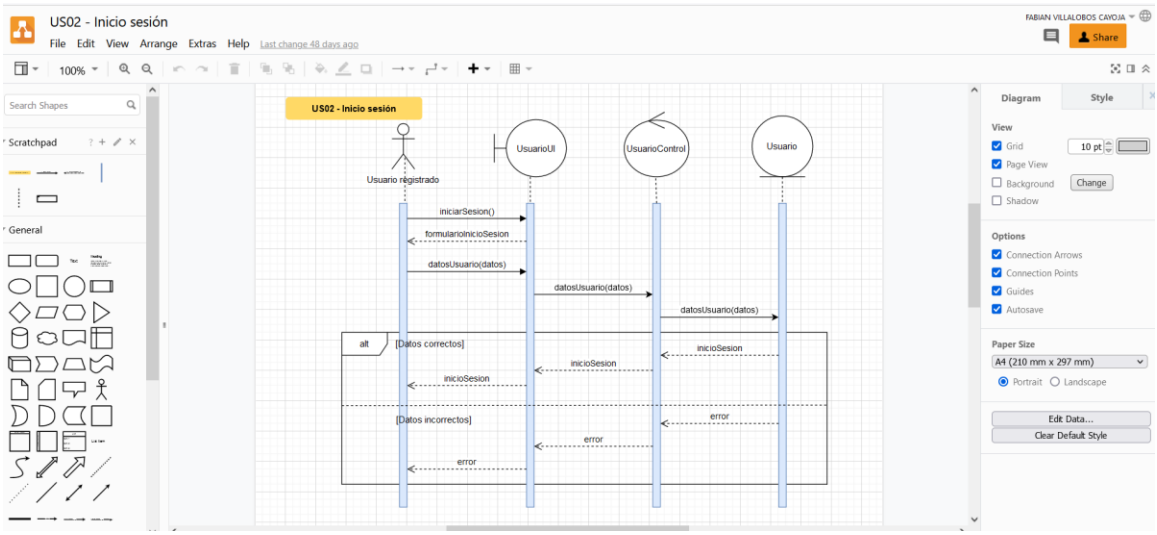


Ilustración 8: Realización de un diagrama de secuencia en Diagrams.net

5.3.2. Jira

Aplicación web para manejar la administración de un proyecto software. Permite la gestión ágil de proyectos, en específico, tiene plantillas adaptadas a proyectos que utilicen Scrum u otras metodologías ágiles.

Se han utilizado distintas herramientas integradas en la aplicación. Como ejemplos más importantes se tiene la creación de un *backlog del producto* y la creación del *backlog* de cada *sprint* (Ilustración 9), la hoja de ruta con la organización y realización temporal de las tareas, la organización las historias de usuario que se deben cumplir en cada versión, la asignación de los puntos de historia y la división las historias de usuario por módulos.

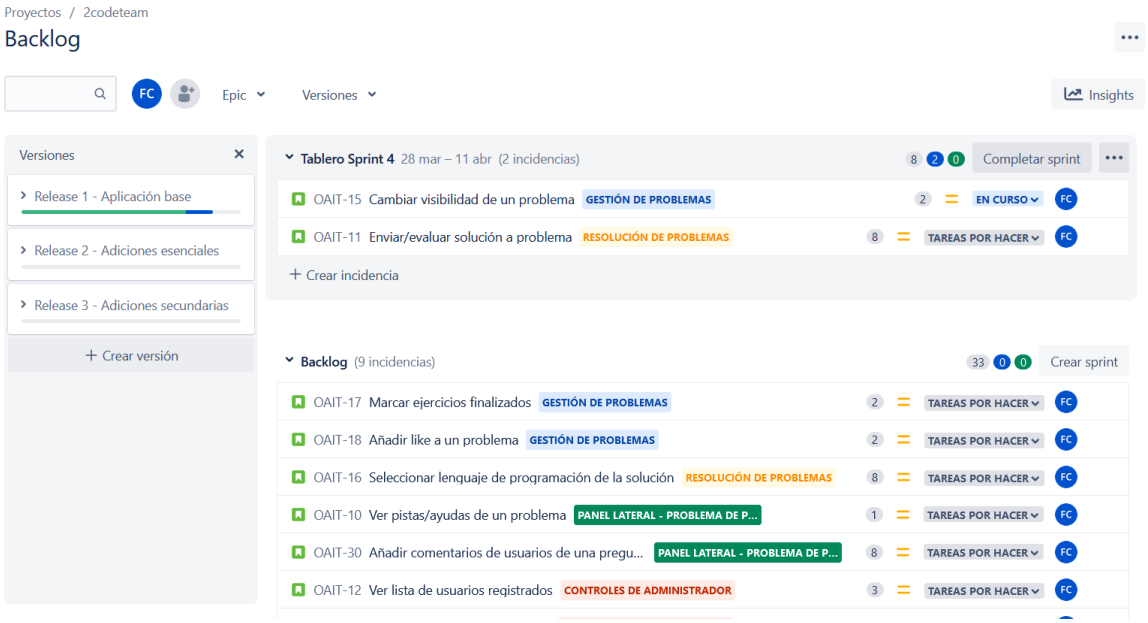


Ilustración 9: Backlog del sprint 4 y backlog del producto en Jira

5.3.3. Trello

Como Jira, es utilizada en aspectos de gestión de proyectos, pero sin capacidad de seguimiento de tiempo o generación de informes. El beneficio de esta herramienta es que posee una interfaz gráfica e interacciones más amigables con el usuario.

La página web Trello fue utilizada a un principio para mantener el *backlog* del producto y para gestionar las tareas de cada *sprint* en un tablero siguiendo la metodología Kanban (Ilustración 10). Eventualmente se trasladaron a Jira.

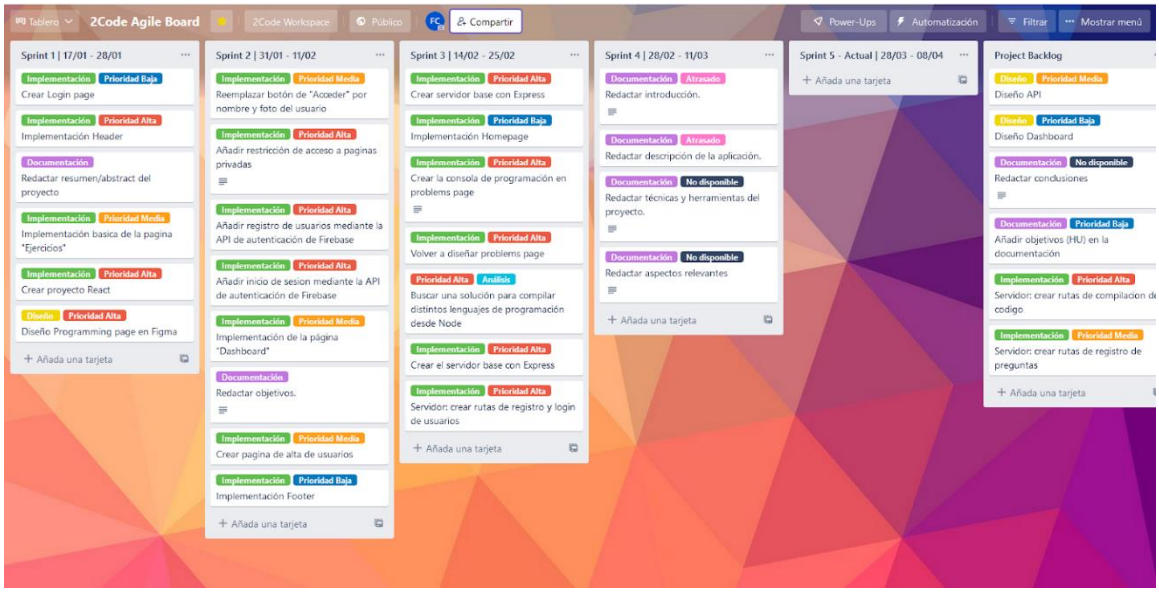


Ilustración 10: Tablero Kanban de 2Code en Trello

5.4. Herramientas de desarrollo

5.4.1. PostgreSQL

Es una base de datos relacional muy capaz y establecida, con más de 30 años de continuas mejoras, haciéndola una de las bases de datos más robustas y con buen rendimiento. Es de código abierto y se adhiere al estándar SQL en su mayoría [16]. Como se ha mencionado antes, se ha utilizado para crear las tablas de datos de la aplicación. Se ha incluído una librería adaptada a Node.js llamada “node-postgres” y que se ha utilizado para generar las consultas e ingresar datos a la base de datos. Esta librería tiene un modulo que permite generar múltiples consultas al mismo tiempo a una misma base de datos desde muchos clientes, permitiendo mantener concurrencia entre los procesos ejecutados desde diferentes clientes al único servidor de una forma segura.

5.4.2. Node.js

Además de ser el entorno de ejecución de la aplicación en el servidor, sirve como herramienta de desarrollo ya que durante todo el proceso de implementación del proyecto permite compilar y ejecutar el código en un servidor local de desarrollo. Como se ha mencionado antes, Node es también utilizado por Chrome para ejecutar código JavaScript del cliente.

5.4.3. Express

Express es una infraestructura para aplicaciones web que utilizan Node.js. Funciona como una capa de *middleware* que permite a una aplicación recibir peticiones HTTP y responder a ellas con diversos tipos de datos y mediante una gran cantidad de metodos que pone a disposición mediante su API [17]. De igual forma, permite implementar metodos propios para manejar y procesar las peticiones. En 2Code se ha utilizado Express para crear un servidor REST mediante el cual se reciben las peticiones que inician los procesos para guardar información, realizar consultas en la base de datos y para evaluar las soluciones de problemas de programación enviadas desde el *front end* de la aplicación. Express es muy robusto y permite soportar una gran cantidad de peticiones concurrentes en un servidor de producción.

5.4.4. Git

Es un software de control de versiones muy establecido. Viendo que 2Code tiene el potencial de convertirse en un software bastante grande y complejo, se ha utilizado GIT como control de versiones ya que tiene la capacidad de crear ramas o *branches* que sirven para desarrollar una nueva característica en un entorno aislado del resto del código, y pudiendo revertir los cambios al último punto estable de la aplicación si fuese necesario. GIT permite que estas nuevas características, una vez finalizadas, puedan ser fusionadas de vuelta a la rama principal del proyecto.

5.4.5. GitHub

Una aplicación web que permite guardar repositorios de control de versiones creados con Git en la nube. Puesto que 2Code está separado en dos aplicaciones de menor tamaño, una para la aplicación cliente que compone el *front end* y otra para la lógica del servidor y la base de datos que compone el *back end*, se tiene dos repositorios individuales en GitHub para cada uno. Además de mantener una copia del código de la aplicación segura en la nube, a la hora de desplegar la aplicación es posible enlazar el repositorio de GitHub del cliente con el servidor de Netlify, que se ocupa mantener la aplicación accesible mediante internet. De esta forma, cada cambio realizado a la rama principal del repositorio en GitHub desencadena un nuevo despliegue de la aplicación con los nuevos cambios.

Aplicación cliente: https://github.com/Fabianvc88/2code-client

Aplicación del servidor: https://github.com/Fabianvc88/2Code-server

5.4.6. Visual Studio Code

El editor de código más utilizado actualmente para el desarrollo de aplicaciones web, y el que se utiliza para el desarrollo de 2Code, es Visual Studio Code (VSC). Fue creado por Microsoft y se ha popularizado por ser muy ligero, pero a la vez muy potente gracias a la gran cantidad de extensiones o *plugins* que han permitido una mayor integración al utilizar ciertos lenguajes de programación, herramientas, tecnologías y frameworks, siendo el PERN *Stack* uno de los más integrados a VSC.

5.5. Librerías y *frameworks* relevantes

5.5.1. React

Inicialmente, las aplicaciones web generaban todo el código del *front end* desde el mismo servidor y este se enviaba al usuario mediante respuestas a peticiones HTTP. Sin embargo, en la era moderna de JavaScript se popularizó lo que es ahora conocido como "*client side rendering*" o CSR. CSR se popularizó gracias a *frameworks* y librerías que lo han implementado. Entre estos los más populares son Angular, React y Vue.js. React es un caso particular ya que es una librería y no realmente un *framework* como los otros dos. Esto significa que muchos de los módulos que podría utilizar una aplicación web ya vienen incluidos en Angular y Vue.js, como, por ejemplo, el sistema de enrutamiento, y no lo están en React. Por tanto, React da una cierta libertad al programador de escoger las librerías de terceros que se quieran incluir. El lado negativo de este aspecto es que es necesario estar bien informado de cuáles librerías cumplen con las necesidades del proyecto y estén suficientemente bien documentadas. En 2Code, React se ha utilizado para generar la totalidad de la aplicación cliente.

5.5.2. Tailwind CSS

Más conocido como Tailwind, es un *framework* de CSS que permite generarlo de una forma más sencilla y de bajo nivel. A diferencia de librerías como Bootstrap, no tiene generado un

conjunto de clases predefinidas para los elementos (botones, tablas, etc), sino que tiene un conjunto de clases de "bajo nivel" que, en general, sólo modifican una o un par de características CSS de un elemento [18]. Casi como pseudo etiquetas, se añaden directamente sobre la propiedad "className" de los elementos JSX. Pese a tener un menor nivel de reutilización que generando ficheros CSS, se pueden generar diseños mucho más avanzados en muy poco tiempo. Además, su post procesador disminuye el tamaño de la versión de producción del proyecto eliminando todas las etiquetas que no influyen en el aspecto final de los elementos.

Todos los componentes, menos las tablas, y páginas de 2Code se han estilizado utilizando las etiquetas de Tailwind.

5.6. Herramientas de despliegue

5.6.1. Heroku

"Heroku es una plataforma como servicio (PaaS) de computación en la Nube que soporta distintos lenguajes de programación." [19]. Heroku permite desplegar aplicaciones en distintos lenguajes de programación, entre los cuales está Node.js. Heroku tiene distintas versiones de Ubuntu para ejecutar las aplicaciones y es capaz de utilizar un gran rango de versiones de Node.js indicándolo en el fichero "package.json" que se genera en todas la aplicaciones de Node. Para desplegar 2Code se ha creado una aplicación en Heroku y se le ha añadido una extensión de PostgreSQL que viene también con Heroku.

Un punto muy positivo de Heroku es que utiliza Git para desplegar la aplicación. Esto se hace de una forma similar a como funciona GitHub ya que simplemente se tiene que subir la rama principal del código al repositorio de la aplicación en Heroku. Es el servidor de Heroku que se ocupa de generar la versión de producción y ejecutarlo.

5.6.2. Netlify

Al igual que Heroku, Netlify es un PaaS y permite alojar aplicaciones web en la Nube. A diferencia de Heroku, soporta aplicaciones *front end* sin la necesidad de tener un servidor como Express que envíe las páginas ya que Netlify lo hace por nosotros. Por otro lado, Netlify soporta cierta librería que Heroku no, como por ejemplo React Router que ha sido utilizado en 2Code.

Para realizar el despliegue de la aplicación, se ha enlazado el repositorio de GitHub de la aplicación del servidor con una aplicación en Netlify. De esta forma, Netlify clona el repositorio en su propio servidor, genera la versión de producción y la despliega de una forma similar a Heroku.

Enlace a la página alojada en los servidores de Netlify: https://resonant-nasturtium-032ce8.netlify.app/

5.7. Herramientas de documentación

5.7.1. Especificación OpenAPI

OpenAPI es una especificación no propietaria que ha sido creada y sigue siendo mantenida por un consorcio de empresas que busca crear un estándar para facilitar la descripción de APIs. Consiste en utilizar YAML, un lenguaje de programación de serialización de datos similar a XML o JSON, para definir, bajo una estructura única, un mínimo de información sobre la API en un fichero descriptivo.

5.7.2. Swagger

Swagger es una aplicación web para generar rápidamente documentación de una API siguiendo la Swagger Specification o la OpenAPI Specification. Su opción de pago también permite generar una página web interactiva que incluye toda la documentación generada.

Para este Proyecto se ha generado la documentación de la API utilizando la especificación OpenAPI 3.0.0. En esta documentación se han definido las posibles respuestas de todas las solicitudes que la API de 2Code podría recibir. Cada respuesta tiene un código para indicar si la operación fue exitosa o no. OpenAPI también permite definir los tipos de datos que se responden y dar ejemplos de los objetos o datos de estos.

5.7.3. Swagger to PDF

Es un sitio web que genera documentación de una API a partir de un fichero JSON que sigue la especificación Swagger 2.0 o OpenAPI 3.0.0 [20]. Desde Swagger se ha descargado el fichero JSON resultante y a partir de el se ha generado varias hojas con la especificación de la API de 2Code en PDF. Estas hojas están adjuntas en el Anexo I.

6. ASPECTOS RELEVANTES

Este apartado comenta las partes más importantes de la totalidad del proyecto software.

6.1. Metodología del trabajo

El Proceso Unificado (PU) es un marco de desarrollo software dirigido por los casos de uso. A diferencia de las metodologías clásicas, el Proceso Unificado ha presentado ser muy efectivo y flexible a la hora de integrar o modificar requisitos [21]. Sin embargo, PU puede llegar a ser un proceso lento y complejo, en el que se tiene que planificar mucho por adelantado, por lo que se considera que los miembros del equipo deben ser expertos en su campo para poder desarrollar software bajo esta metodología.

Actualmente, muchas compañías de software han pasado a utilizar metodologías ágiles y aunque PU sigue siendo utilizado, se considera que las metodologías ágiles han reemplazado al Proceso Unificado. En general, las metodologías ágiles son aún más flexibles y adaptables a cambios en los requisitos del software, además de ser más fáciles de adoptar.

Para el desarrollo de este proyecto se ha adoptado una mezcla de técnicas típicamente utilizadas en metodologías ágiles, como Scrum, y también otras típicamente utilizadas en el Proceso Unificado.

6.1.1. Organización de tareas

Para empezar, con 2Code se ha seguido una planificación basada en la obtención de artefactos entregables que se han denominado “versiones” de la aplicación. Para ello se han organizado las historias de usuario del *backlog* del producto en versiones, empezando con aquellas que se han determinado de mayor prioridad para el proyecto. Iniciando por la primera versión, se extraen de ella las historias de usuario al *backlog* del *sprint*. Cada *sprint* está dividido en marcos de tiempo de 2 a 4 semanas según los puntos de historia que contiene.

En resumen, las técnicas más importantes para la organización de las tareas del proyecto son las siguientes:

- División del proyecto en múltiples objetivos de menor tamaño (versiones):

    - Mediante el análisis inicial de las historias de usuario, crear el primer *product backlog*.
    - Asignar las historias de usuario a las distintas versiones según prioridad de las características que se quieran integrar con preferencia en la aplicación.
    - Definir los puntos de historia de las tareas en el *product backlog*.
    - Calcular y ajustar continuamente al finalizar cada *sprint* la estimación de tiempo del proyecto.
- Planificación del *sprint*:
    - Planificar el objetivo del *sprint*.
    - Crear el *sprint backlog* seleccionando las tareas que lleven a la versión en la que se trabaja, manteniendo el tiempo para completarlo entre dos a cuatro semanas.
    - Planificación de las tareas para el *sprint*.
- Ajuste diario del tablero del *sprint* (Kanban):
    - Tiempo diario para actualizar el estado del *sprint* con las últimas tareas finalizadas.
    - Incluir en el tablero problemas o ideas para *sprints* futuros.
- *Sprint*:
    - Para cada historia de usuario en el *backlog* del *sprint*:
        - Se realiza el análisis de la historia de usuario, se profundiza los requerimientos y su(s) solución(es) a partir del análisis inicial.
        - Se diseña la solución escogida o se mejora el diseño inicial y se define las interacciones entre componentes, clases o con la API.
        - Se crea e implementa la solución.
        - Se prueba la solución y se revisa el correcto funcionamiento.
- Revisión del *sprint*:
    - Evaluación del producto obtenido en el *sprint*.
    - Según el progreso de las tareas hacia el objetivo, planificar y ajustar las actividades en el *backlog* del producto.
    - Probar, revisar y corregir posibles efectos secundarios de la implementación a los entregables previos.

6.1.2. Especificación de historias de usuario

Una historia de usuario es una descripción informal de una funcionalidad que se desea tener en el producto software. Para este proyecto se ha utilizado el lenguaje natural para describir el requisito y su motivación, según la perspectiva del usuario final, y sigue una estructura mínima para describir el tipo de usuario, una breve descripción del requisito y su motivación. Aunque en principio se busca que el usuario final aporte las historias de usuario, en teoría, el desarrollador de la aplicación también puede aportar historias de usuario.

Una historia de usuario en un marco de trabajo ágil no define el trabajo exacto que se tiene que realizar, sino que ayuda a generar una reflexión sobre las posibles soluciones para cumplir con el requisito del usuario o dueño del producto. De esta manera no se limita al equipo a trabajar con un requerimiento y solución estáticos durante meses, sino que permite cambios y ajustes a la solución.

Las historias de usuario implican por tanto una reducción de la documentación típica ya que, a diferencia del Proceso Unificado donde se tiene un diseño específico de las soluciones, se tiene sólo suficiente información para producir una solución que podría cambiar con el tiempo, adaptándose a los requisitos cambiantes.

Para 2Code, se han realizado las historias de usuario siguiendo la estructura “Quién, qué, por qué”, o mejor conocido en inglés como “las tres W”: “Who, what, why”. Esta estructura facilita la lectura de las historias de usuario ya que todas siguen la misma estructura.

A partir de las historias de usuario se ha calculado los puntos de historia y realizado una primera reflexión sobre la posible solución y sus correspondientes tareas (Ilustración 11, Ilustración 12).

Historias de usuario		PH	Solución		Tarea	
US1	**Como** un usuario no registrado **quiero** poder crear una cuenta **para** que el sistema pueda autenticarme.	3	SO1	Crear un sistema de registro de usuarios.	TA1	Crear una página de registro de usuario
					TA2	API: añadir una interfaz que almacene los datos de usuario en la BBDD
					TA3	API: añadir una interfaz que modifique los datos de usuario en la BBDD
					TA4	API: añadir una interfaz que elimine los datos de usuario en la BBDD
US2	**Como** un usuario registrado **quiero** poder iniciar sesión con mi email y contraseña **para** que el sistema pueda autenticarme.	2	SO2	Crear un sistema de inicio de sesión de usuarios.	TA5	Crear una página de inicio de sesión
			SO3	Crear un sistema de autenticación de usuarios para restringir el acceso a páginas protegidas.	TA6	Proteger las páginas reservadas a usuarios registrados: autenticación
US3	**Como** usuario **quiero** poder explorar la lista de problemas disponibles **para** escoger el que quiero intentar resolver.	1	SO4	Crear una página con la lista de problemas.	TA7	Crear página de problemas
US4	**Como** usuario registrado **quiero** que la aplicación marque los ejercicios completados **para** saber que ejercicios ya he hecho y cuales me falta por hacer.	2	SO5	En la página de la lista de problemas, identificar visualmente cuales se han completado y cuales se han iniciado.	TA8	Añadir indicador a la página de problemas que indique cuando un problema ya haya sido resuelto.
					TA9	Añadir indicador a la página de problemas que indique cuando se haya comenzado a resolver un problema.
			SO6	Almacenar en la base de datos una relación entre el estado de los ejercicios y el usuario.	TA10	Añadir atributo a la clase solución para indicar el estado (progreso) de la solución. Sin estado = problema no iniciado, en curso = problema iniciado, finalizado = problema resuelto.
US5	**Como** usuario registrado **quiero** poder indicar que ejercicios me han gustado **para** poder recomendar el problema a otros usuarios.	2	SO7	Añadir un atributo de "me gusta" o "recomiendo" a los problemas.	TA11	Añadir un atributo de comendación a la clase problema en la BBDD.
US6	**Como** usuario registrado **quiero** poder escoger el lenguaje de programación **para** responder en el lenguaje que se me resulte más cómodo.	8	SO8	Añadir un selector de lenguaje de programación en el que se resolverá el problema.	TA12	Añadir un boton deplegable a la página de resolución del ejercicio para seleccionar el lenguaje de programación.
			SO9	En el backend, utilizar el compilador correspondiente al lenguaje de programación escogido por el usuario para lanzar la prueba.	TA13	Crear una clase que lance diferentes procesos según el lenguaje de programación que se escoje.
US7	**Como** usuario registrado **quiero** tener la opción de obtener pistas **para** poder resolver problemas muy difíciles.	1	SO10	Incluir una opción para obtener pistas sobre cómo resolver el problema.	TA14	Añadir un botón a la página de resolución del ejercicio para poder obtener pistas de cómo resolver el problema.
US8	**Como** usuario registrado **quiero** poder guardar soluciones **para** visualizar las más interesantes en un futuro.	2	SO11	Guardar las soluciones realizadas por el usuario en la BBDD.	TA15	Crear una clase "soluciones" en la BBDD.
					TA16	Relacionar la solución con el usuario.
					TA17	API: añadir una interfaz que registre soluciones en la BBDD
					TA18	API: añadir una interfaz que modifique soluciones en la BBDD
					TA19	API: añadir una interfaz que elimine soluciones en la BBDD
US9	**Como** usuario registrado **quiero** que la aplicación indique el estado de resolución de los problemas **para** poder saber cuales no he terminado.	5	SO12	Añadir un atributo de "solución iniciada" o que indique el estado actual de la solución para saber que problemas se han iniciado pero no terminado.		TA10

Ilustración 11: Definición historias de usuario 1-9

US10	**Como** usuario registrado **quiero** poder ejecutar/evaluar código **para** saber si responde al problema.	13	SO13	Compilar y ejecutar la función de la solución. Se evalua comprobando los casos de prueba especificados para el problema.	TA20	API: añadir una interfaz que intente compilar la solución al problema.
					TA21	Pasar los casos de prueba a la función compilada para evaluar su lógica.
US11	**Como** admin **quiero** poder crear problemas de programación **para** que los usuarios puedan intentar resolverlos.	5	SO14	Añadir una forma de almacenar problemas en la base de datos.	TA22	API: añadir una interfaz que registre problemas en la BBDD
					TA23	API: añadir una interfaz que modifique problemas en la BBDD
					TA24	API: añadir una interfaz que elimine problemas en la BBDD
US12	**Como** admin **quiero** poder elegir si un problema está visible o oculto **para** poder guardar un problemas incompletos en lo que se finaliza su creación.	2	SO15	Añadir atributo a la clase problema de la BBDD para que sea visible o no en la página de selección de problemas.	TA22	
					TA25	Añadir atributo a la clase problema de la BBDD para indicar su visibilidad.
US13	**Como** admin **quiero** ver la lista de usuarios registrados **para** poder tener acceso a su información individual.	3	SO16	Crear página, únicamente visible para admins, que muestre la lista de usuarios de la aplicación.	TA26	Crear página con la lista de usuarios.
					TA27	API: añadir una interfaz GET que devuelva la lista de usuarios.
			SO17	Crear página de usuario, únicamente visible por admins, que muestre sus datos, que permita modificarlos y que permita eliminar el usuario.	TA28	Crear página de usuario con sus datos y rol. Incluir botones para modificalos.
					TA29	API: añadir una interfaz GET que devuelva la información de un usuario.
					TA30	API: añadir una interfaz PUT que actualice la información de un usuario
					TA31	API: añadir una interfaz DELETE que elimine la cuenta de un usuario.
US14	**Como** admin **quiero** poder modificar los permisos de los usuarios en la aplicación **para** añadir otros administradores o quitarlos.	2	SO18	Añadir botón en la página de la lista de usuarios para añadir un nuevo usuario.	TA32	Añadir botón en la página de la lista de usuarios para añadir un nuevo usuario.
			SO19	Añadir botón en la página de usuario para modificar el rol del usuario.	TA33	TA28
US15	**Como** admin **quiero** poder modificar todos los problemas **para** controlar que sean apropiados y que su contenido esté correcto.	3	SO20	Tener los botones para modificar los problemas visibles a los admins.	TA34	Añadir botón para modificar problema en la página de lista de problemas.

Ilustración 12: Definición de historias de usuario 10-15

6.1.3. Estimación del esfuerzo

En la metodología ágil, a las historias de usuario se les asigna un valor que representa el esfuerzo o dificultad que tomaría completarla. Este valor es determinado según algunas características definidas por el equipo, aunque en general suelen ser las siguientes:

- Riesgo: Se determina según la falta de detalle del requerimiento, la cantidad de dependencias y de los posibles cambios que podría sufrir.
- Complejidad: Está directamente relacionada a la dificultad de desarrollo de la funcionalidad.

- Repetición: Se determina según la familiaridad del equipo con la funcionalidad a desarrollar, además de que tan monótona son las tareas que la componen.

Para asignar los valores a cada historia de usuario se utiliza comúnmente la secuencia de Fibonacci. Es decir que se utilizan los valores 1, 2, 3, 5, 8, 13 y 21. Por encima de 21 puntos de historia se considera que la historia de usuario puede ser dividida de forma que las resultantes tengan un valor igual o menor a 21. La razón por la que no se utiliza una escala lineal es que sería muy difícil decidir cuál valor se asigna entre dos números seguidos. Por ejemplo, la diferencia entre un requerimiento de dificultad 7 y otro de dificultad 8 es muy pequeña y no sería fácil decidir a cuáles historias de usuario se les debería asignar un valor u otro. Por tanto, se utilizan valores que se distancian mucho más entre sí, facilitando la asignación a cada historia de usuario.

Inicialmente se ha escogido una historia de usuario a la que se puede determinar su dificultad fácilmente. Esta serve de referencia para asignar valores al resto de historias de usuario; son asignaciones relativas a la primera. En el caso de 2Code, se utiliza la historia de usuario US3 como el caso más simple.

Utilizando este método en 2Code, obtenemos las asignaciones de valores de la columna PH (Puntos de Historia) en la Ilustración 13.

Siguiendo la metodología ágil, tanto en proyectos que utilizan Scrum o Extreme Programming, se utiliza un método "de arriba a abajo" o "*top-down*", para calcular progresivamente el tiempo estimado que tardaría en completarse el proyecto [12]. En otras metodologías se utiliza un método inverso, "de abajo para arriba" o "*bottom-up*", para estimar el tiempo de desarrollo. Por ejemplo, en PU se utiliza el método *bottom-up* en la cual se determina el esfuerzo para completar las tareas que componen los casos de uso, y se calcula a partir de ellas una estimación de tiempo para completar la tarea. A partir de ahí, se añaden las tareas a un diagrama de Gantt con el cual se obtiene una estimación total del tiempo para completar el proyecto.

Con el método *top-down*, el equipo de desarrollo asigna los puntos de historia a las historias de usuario y con conocimiento previo de la velocidad de trabajo del equipo se sabe cuántos puntos de historia pueden completar en un *sprint* [12]. En caso de no tener esta información previa, el equipo puede realizar un primer *sprint* intentando completar una cierta cantidad

de historias de usuario. Con el resultado de ese primer *sprint*, según cuantas historias de usuario se han logrado completar (sin tomar en cuenta historias de usuario comenzadas y no finalizadas), se obtiene la velocidad del equipo con la que se puede comenzar a estimar el resto de *sprints*.

El gran beneficio de este método es que con cada *sprint* se obtiene una mejor estimación de la velocidad del equipo y no sólo la velocidad de un programador. Además, con esta información se puede también predecir un rango con la cantidad de puntos de historia que un equipo puede completar, tanto en el peor de los casos, como en el mejor.
La desventaja de este método de estimación de tiempo es que, a diferencia del método *bottom-up* que utiliza el PU, es difícil estimar el tiempo que toma un proyecto completo si el equipo de desarrollo no tiene información de su velocidad. En este caso es mejor que un desarrollador con experiencia previa haga la estimación hasta que se pueda calcular la velocidad del equipo con el primer *sprint*.

Tras realizar un primer *sprint* de duración estándar de 5h por día por 2 semanas se logra completar las siguientes historias de usuario:

ID	PH	Terminado
US1	3	x
US2	2	x
US3	1	
US10	13	
US6	8	
US7	1	
US4	2	
US9	5	
US5	2	
US8	2	
US11	5	

Ilustración 13: Determinación de velocidad - sprint 1

Se obtiene una velocidad de 5 puntos de historia por *sprint*, con lo que podemos realizar la siguiente estimación provisional del proyecto: [13]

Número total de puntos de historia inicial para todo el proyecto (TPH): 54
Tomando un estimado de 5 puntos de historia por sprint:

$$\frac{54\ PH}{5\ PH/sprint} \approx 10.8\ sprints$$, ó 11 sprints si se redondea el resultado.

Suponiendo que todos los *sprints* son de duración estándar (5h por semana por 2 semanas), entonces obtenemos:

$$11\ sprints\ \times 2\ semanas = 22\ semanas\ \approx 5.1\ meses$$

La primera estimación que obtenemos para la realización de 2Code, con un total provisional de 54 puntos de historia, es de 5.1 meses.

6.1.4. Planificación temporal

Como se ha visto, en las metodologías ágiles no existe realmente una conversión exacta de puntos de historia a horas o a cualquier otro valor temporal. Sin embargo, una vez que se tiene una estimación inicial de la velocidad de desarrollo, se puede realizar una planificación temporal utilizando la velocidad de desarrollo para cada *sprint*. Con cada *sprint*, la velocidad y por tanto la planificación temporal se van refinando hasta obtener una estimación de tiempo de desarrollo de proyecto cada vez más cercano a la realidad.

Para el siguiente *sprint*, se realiza una priorización de las historias de usuario que aportan más valor y que tengan sentido de ser realizadas antes. Para ello, las historias de usuario se asignan a distintas versiones de la aplicación. Cada versión agrupa las historias de usuario que cumplen una etapa de la aplicación. Por ejemplo, la versión base es la que incluye las funciones más importantes para cumplir con los objetivos principales del proyecto.

Tras haber obtenido una estimación inicial de la velocidad de desarrollo, la planificación temporal para el segundo sprint es la siguiente (Ilustración 14):

	Historias de usuario	PH	Pronóstico	Sprint
US1	**Como** un usuario no registrado **quiero** poder crear una cuenta **para** que el sistema pueda autenticarme.	3	finalizado	1
US2	**Como** un usuario registrado **quiero** poder iniciar sesión con mi email y contraseña **para** que el sistema pueda autenticarme.	2	finalizado	
US11	**Como** admin **quiero** poder crear problemas de programación **para** que los usuarios puedan intentar resolverlos.	5	Más probable	2
US3	**Como** usuario **quiero** poder explorar la lista de problemas disponibles **para** escoger el que quiero intentar resolver.	1		
US10A	**Como** usuario registrado **quiero** poder visualizar la plantilla del problema para poder escribir mi solución**.**	3	Optimista	
US7	**Como** usuario registrado **quiero** tener la opción de obtener pistas **para** poder resolver problemas muy difíciles.	1		
US10B	**Como** usuario registrado **quiero** poder ejecutar/evaluar código **para** saber si responde al problema.	8		
US13	**Como** admin **quiero** ver la lista de usuarios registrados **para** poder tener acceso a su información individual.	3		
US14	**Como** admin **quiero** poder modificar los permisos de los usuarios en la aplicación **para** añadir otros administradores o quitarlos.	2		
US15	**Como** admin **quiero** poder modificar todos los problemas **para** controlar que sean apropiados y que su contenido esté correcto.	3		
US12	**Como** admin **quiero** poder elegir si un problema está visible o oculto **para** poder guardar un problema incompleto en lo que finaliza su creación.	2		
US6	**Como** usuario registrado **quiero** poder escoger el lenguaje de programación **para** responder en el lenguaje que se me resulte más cómodo.	8		
US4	**Como** usuario registrado **quiero** que la aplicación marque los ejercicios completados **para** saber que ejercicios ya he hecho y cuales me faltan por hacer.	2		
US5	**Como** usuario registrado **quiero** poder indicar que ejercicios me han gustado **para** poder recomendar el problema a otros usuarios.	2		
US8	**Como** usuario registrado **quiero** poder guardar soluciones **para** visualizar las más interesantes en un futuro.	2		
US9	**Como** usuario registrado **quiero** que la aplicación indique el estado de resolución de los problemas **para** poder saber cuales no he terminado.	5		

Ilustración 14: Pronóstico del sprint 2

Se puede observar que el pronóstico indica, de una forma más abstracta, la probabilidad de completar un conjunto de historias de usuario.

Inicialmente, para mantener la organización de la distribución de tareas por sprint se utilizó Trello (Ilustración 15). En él se creó el *project backlog*, el *sprint backlog* y los *logs* de cada *sprint* con las tareas completadas.

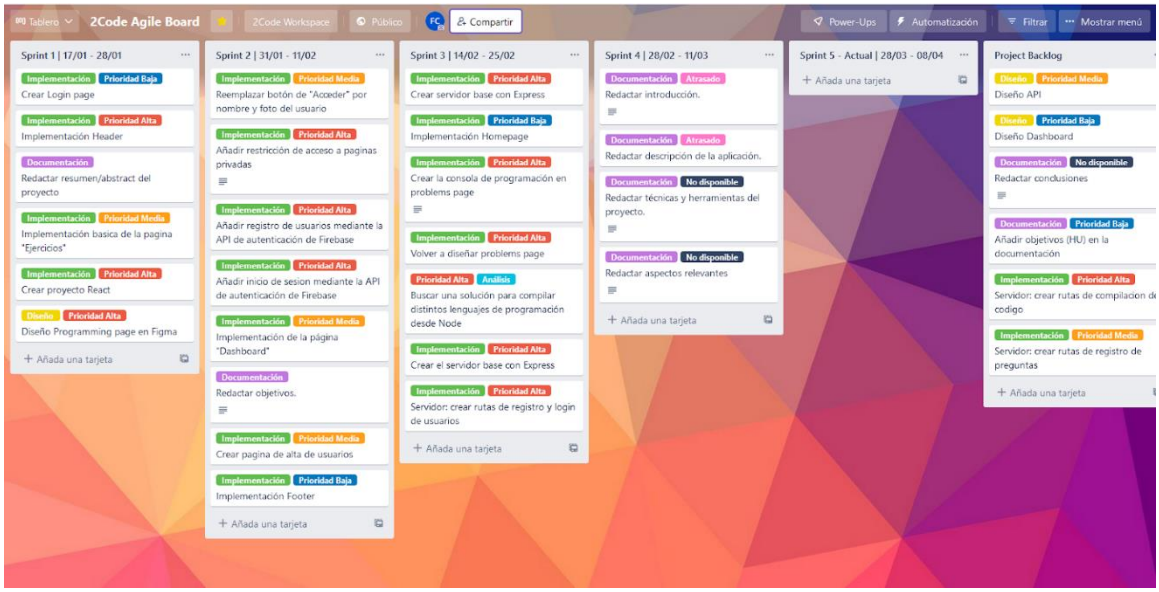


Ilustración 15: Captura del backlog en Trello

Sin embargo, la aplicación web Jira se vio mucho más adaptada al manejo de proyecto ágiles, facilitando la generación de documentación y distintas tablas/vistas con el seguimiento de cada *sprint* y de la totalidad del proyecto en general

Jira, la aplicación web para la administración de proyectos software ágiles, permite visualizar la distribución de historias de usuario por versiones de una forma más intuitiva e interactiva (Ilustración 16, Ilustración 17).

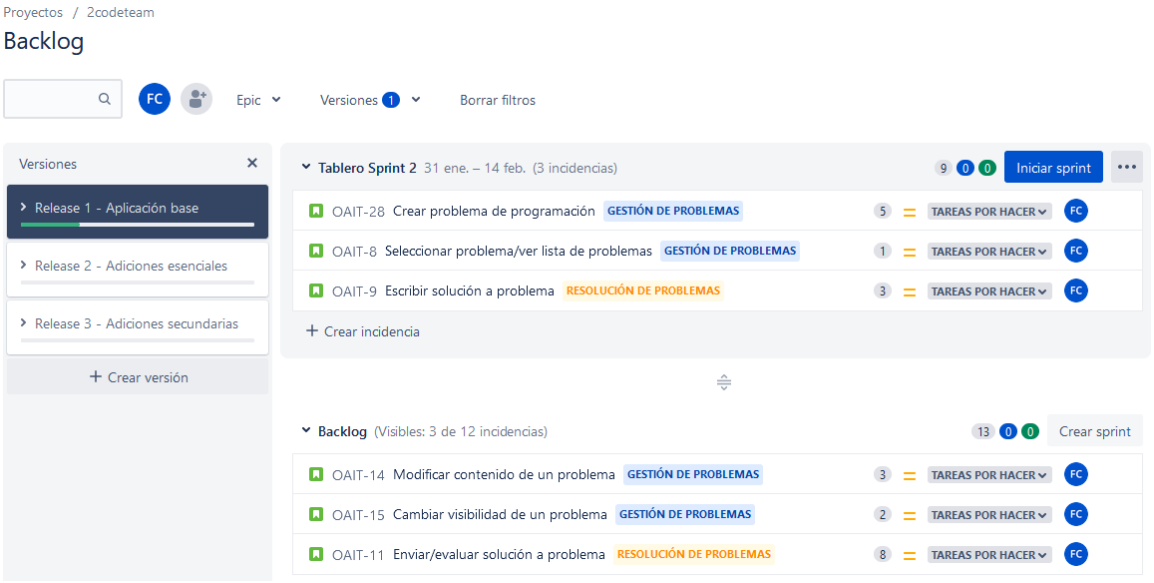


Ilustración 16: Captura del backlog en Jira

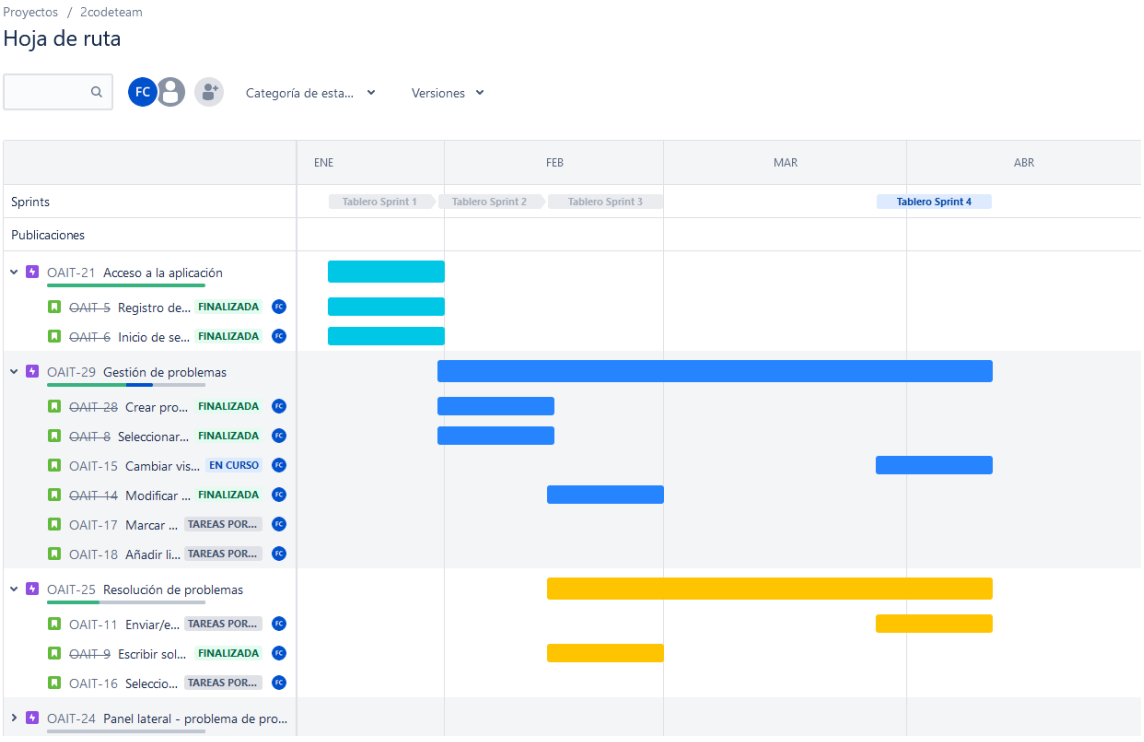


Ilustración 17: Captura de la hoja de ruta en Jira

Una explicación más detallada de la planificación temporal del proyecto se encuentra en el Anexo 1 – Planificación temporal.

6.1.5. Realización de historias de usuario en análisis

Las soluciones propuestas para cada historia de usuario permiten la creación de diagramas de secuencias que especifican el proceso de interacción entre componentes abstractos del sistema. Aún si no se conoce con exactitud que componentes se terminan construyendo, se puede tener una idea inicial de las funciones y llamadas entre estos.

En el siguiente diagrama (Ilustración 18) de secuencia se puede ver el funcionamiento de una de las historias de usuario más importantes de sistema.

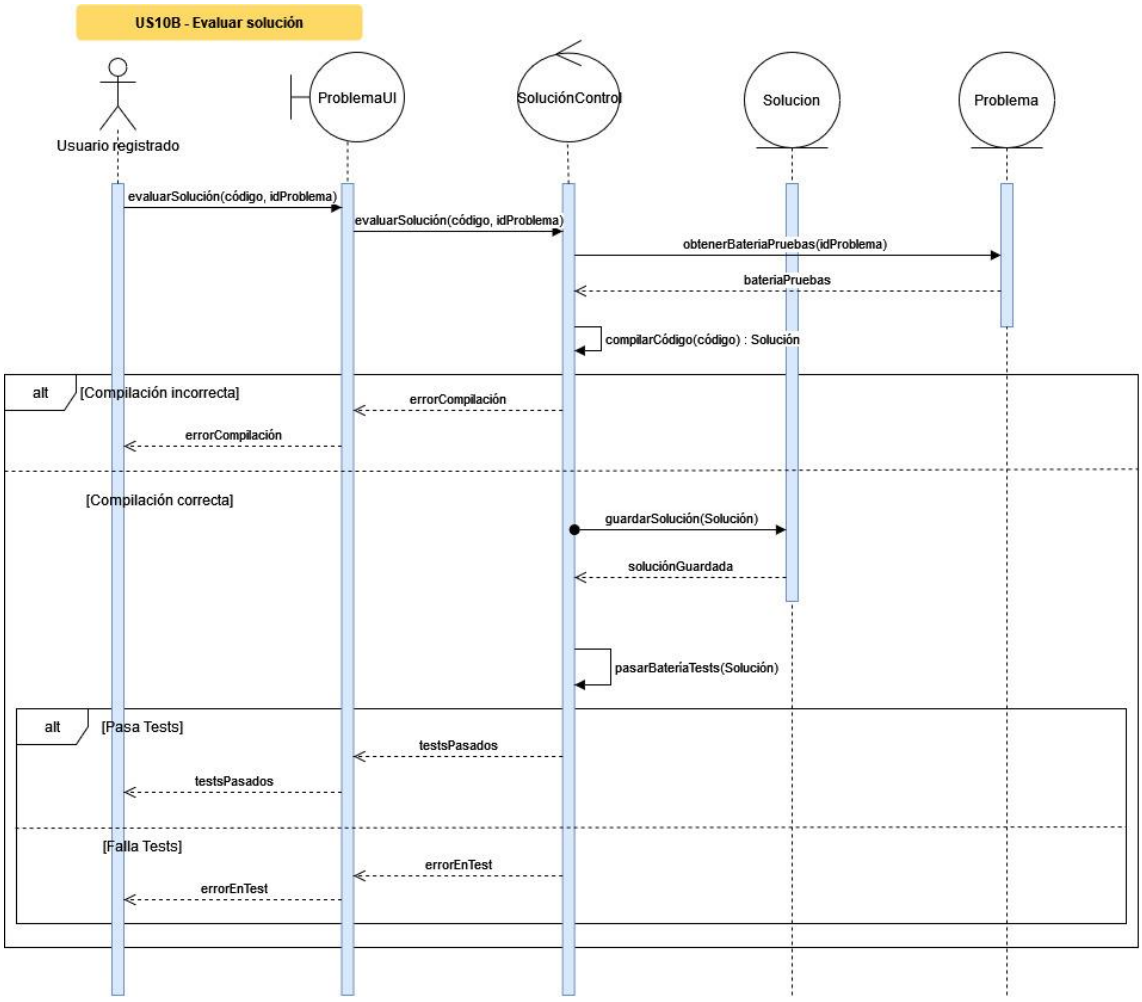


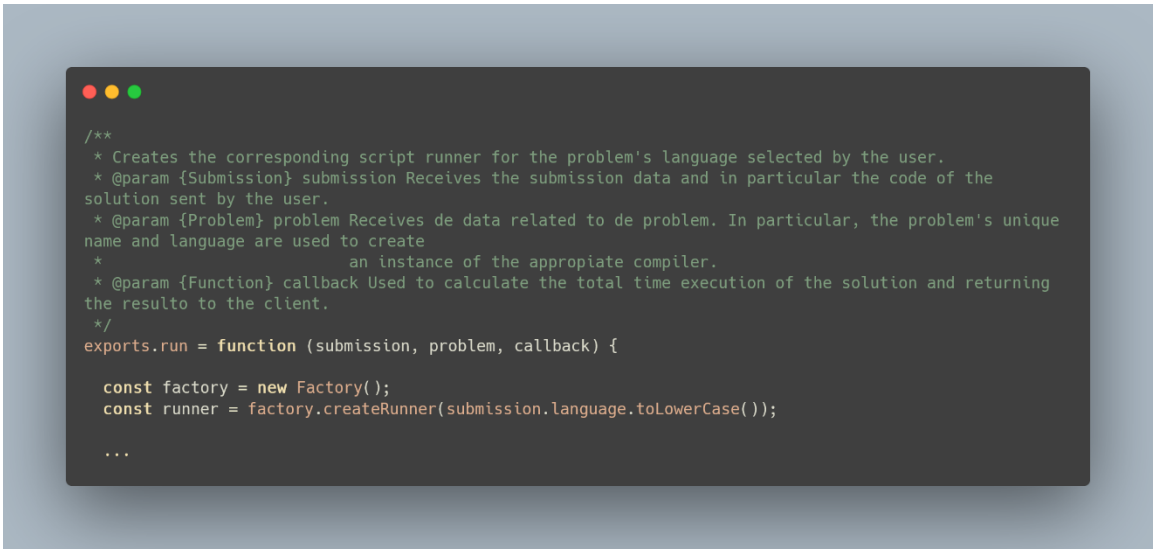
Ilustración 18: Diagrama de secuencia de análisis la historia de usuario US10B - Evaluar solución

Como se puede ver en el diagrama anterior, el controlador recibe la solicitud del usuario a través de la interfaz. Con el identificador del problema "idProblema" puede extraer de la entidad Problema toda la información del problema que se intenta resolver. El controlador intenta compilar el código enviado por el usuario y en caso de que sea compilado exitosamente se procede a guardar la solución y a probar los tests del problema para asegurarse que el código del usuario lo resuelve correctamente. Ya sea que el código resuelve correctamente el problema o no, se notifica al usuario del resultado.

A partir de este diagrama de secuencia se puede visualizar mucho mejor las interacciones entre componentes y clases que se tiene, aunque para ciertas partes más técnicas de la implementación, como por ejemplo la evaluación de la solución enviada por el usuario, el diagrama de secuencia no es más que una vista de alto nivel que sirve como guía de la lógica de la función.

```
function run(res, submission, problem) {
  //console.log("**Starting run function in submission.js**");

  // 1. Start runtime timer
  let start = DateTime.now();

  // 2. Then, run the solution to get the test result
  RunnerManager.run(submission, problem, async function (status, message) {
    //console.log("--Controller callback--");
    let result = {
      status,
      message,
    };
    console.log("status:", status);
    if (status == "pass" || status == "fail") {
      let end = DateTime.now();
      let ms = end.diff(start, ["seconds", "milliseconds"]);
      console.log("Runtime: ", ms.toObject());

      // 3. Find the submission
      submission.status = status;
      submission.runtime = ms.seconds + ms.milliseconds / 1000;

      // 4. Update the submission
      try {
        await submission.updateToDB();
      } catch (err) {
        return res.status(422).json({ errors: [err] });
      }

      // 5. Send results to client
      return res.status(200).json(result);
    } else {
      return res.status(500).json(result);
    }
  });
}
```

Ilustración 19: Parte de la implementación de la función que evalua el código de la solución.



```
/**
 * Creates the corresponding script runner for the problem's language selected by the user.
 * @param {Submission} submission Receives the submission data and in particular the code of the
solution sent by the user.
 * @param {Problem} problem Receives de data related to de problem. In particular, the problem's unique
name and language are used to create
 *                          an instance of the appropiate compiler.
 * @param {Function} callback Used to calculate the total time execution of the solution and returning
the resulto to the client.
 */
exports.run = function (submission, problem, callback) {

  const factory = new Factory();
  const runner = factory.createRunner(submission.language.toLowerCase());

  ...
```

Ilustración 20: Inicio de la función "run" que crea el compilador adecuado para el lenguaje de programación escogido para la solución.

6.2. Gestión de los datos

Al igual que en el apartado anterior, partiendo de las historias de usuario y de las tareas que se crean a partir de ellas se ha podido determinar una gran parte de los datos que se tiene que almacenar para las distintas clases. Las clases conceptuales de los elementos más importantes de la aplicación, y de los cuales vamos a necesitar almacenar datos en forma de atributos, son el usuario, los problemas y las soluciones.

En un análisis inicial se propone el modelo de dominio de la Ilustración 21.

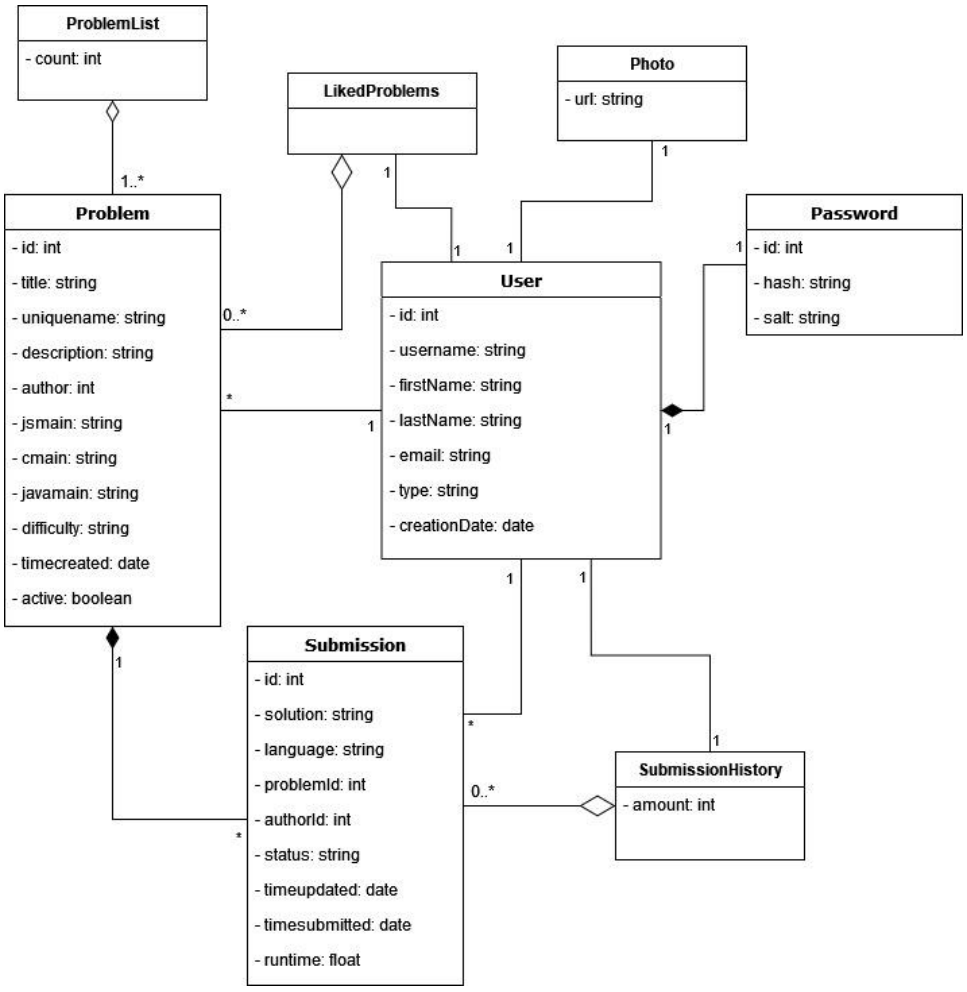


Ilustración 21: Diagrama de clases de análisis

Uno de los principales problemas de un modelo de dominio es que no está particularmente claro si una clase a implementar en la aplicación es equivalente a una clase en el modelo de diagrama de clases. Como se sabe, no está bien definida la diferencia en ciertas relaciones y una propiedad de una clase en este tipo de modelos. Por ejemplo, la relación entre *User* y *Photo* puede ser reemplazada por una propiedad en la clase *User*. Pasa lo mismo con la relación entre *Password* y *User*. En este caso, para 2Code, se ha decidido que, en estos casos, las relaciones que puedan estar como propiedades de otra clase, pero no lo están, es porque la información misma no está alojada en la clase. En los casos mencionados anteriormente, ambos *Password* y *Photo* se han alojado en la base de datos de Firebase y no están en la base de datos PostgreSQL de 2Code. Por lo tanto, en la implementación, cuando se realizan

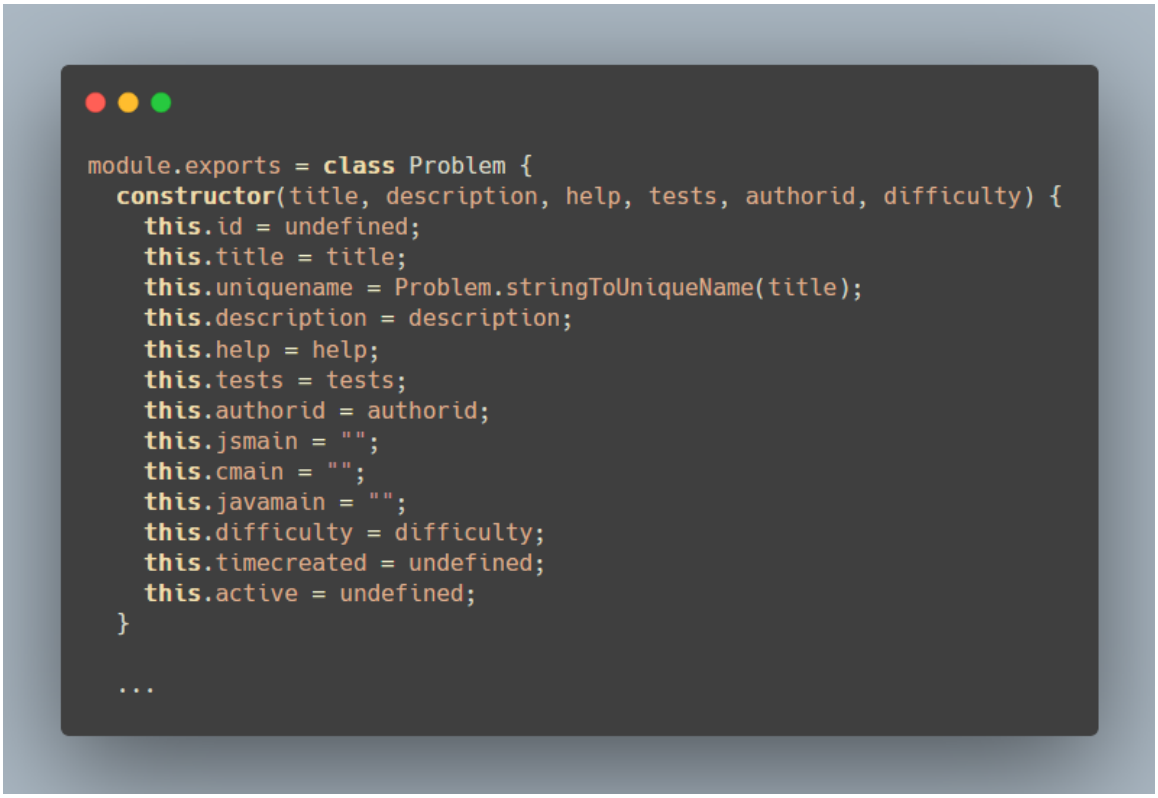
operaciones con objetos de la clase *User*, no se tiene acceso ni a la contraseña ni a la foto del usuario como propiedades de la clase.

Por lo tanto, a partir de este diagrama de clase, se ha creado un fichero escrito en SQL (Ilustración 22) para generar las tablas que alojan las clases descritas anteriormente.

```sql
CREATE TABLE "problem" (
    "id" SERIAL PRIMARY KEY,
    "title" VARCHAR(255) NOT NULL,
    "uniquename" VARCHAR(255) UNIQUE NOT NULL,
    "description" TEXT,
    "help" TEXT,
    "tests" TEXT,
    "authorid" INTEGER NOT NULL,
    "jsmain" TEXT,
    "cmain" TEXT,
    "javamain" TEXT,
    "difficulty" VARCHAR(255) NOT NULL,
    "active" BOOLEAN NOT NULL DEFAULT true,
    "timecreated" TIMESTAMP(3) NOT NULL DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE "submission" (
    "id" SERIAL PRIMARY KEY,
    "problemId" INTEGER NOT NULL,
    "language" TEXT NOT NULL,
    "authorId" INTEGER NOT NULL,
    "solution" TEXT,
    "status" VARCHAR(255) NOT NULL DEFAULT 'pending',
    "runtime" REAL,
    "timeupdated" TIMESTAMP(3) NOT NULL DEFAULT CURRENT_TIMESTAMP,
    "timesubmitted" TIMESTAMP(3)
);
```

Ilustración 22: Parte del documento "database.sql" del proyecto de 2Code.

Una vez definidas las columnas de las tablas en la base de datos, se procede a generar las clases que se utilizan en el servidor para implementar y aprovechar de la programación orientada a objetos que JavaScript soporta (Ilustración 23).



```
module.exports = class Problem {
  constructor(title, description, help, tests, authorid, difficulty) {
    this.id = undefined;
    this.title = title;
    this.uniquename = Problem.stringToUniqueName(title);
    this.description = description;
    this.help = help;
    this.tests = tests;
    this.authorid = authorid;
    this.jsmain = "";
    this.cmain = "";
    this.javamain = "";
    this.difficulty = difficulty;
    this.timecreated = undefined;
    this.active = undefined;
  }

  ...
```

Ilustración 23: Propiedades de la clase "Problem" en la aplicación Node del servidor.

A través de la implementación de las clases como objetos JavaScript, se ha creado en cada uno una gran cantidad de métodos para consultar, insertar o modificar sus correspondientes tablas en la base de datos.

6.3. Arquitectura de la aplicación

El diseño del sistema profundiza las primeras instigaciones realizas en el modelo de análisis. En él se tiene un mayor detalle de como se realizan los componentes de sistema y con eso de toma decisiones de la arquitectura más apropiada. Se tiene mayor detalle en el Anexo I – Análisis del Sistema software.

6.3.1. Patrón arquitectónico

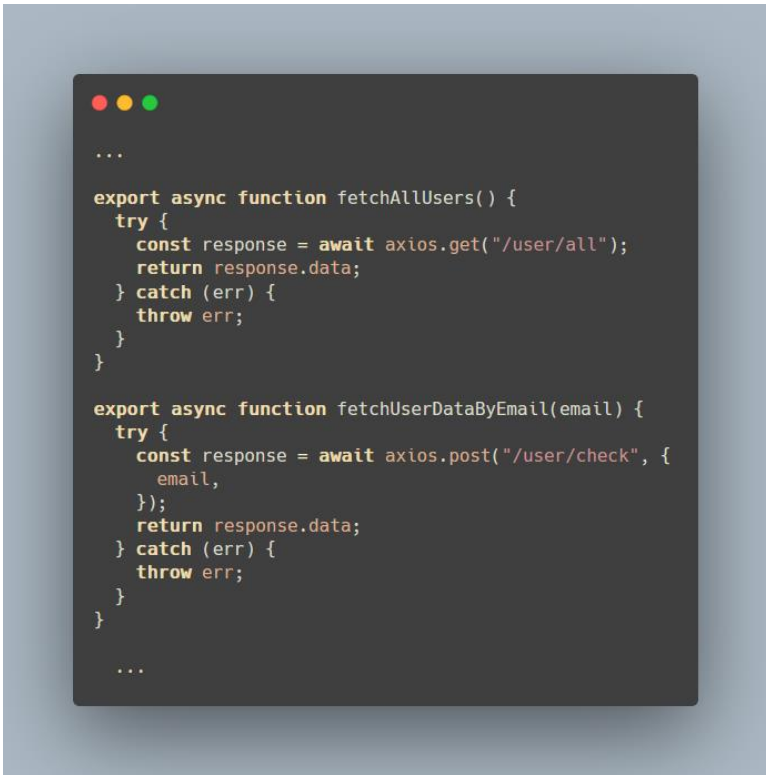
El patrón Modelo-Vista-Controlador (MVC) es uno de los patrones que da mayor libertad de implementación al programador ya que está menos definido que otras arquitecturas. A pesar de ello, sigue siendo la arquitectura más importante en el desarrollo de aplicaciones ya que

permite separar los datos entre la interfaz de usuario, la lógica de control del programa y el modelo de datos. La variación MVC mediado (Ilustración 24) es muy utilizada para aplicaciones web que tengan que almacenar información en bases de datos [22] ya que permite alojar toda la lógica del programa en el Controlador, permitiendo una mayor separación e independencia entre la Vista y el Modelo [23]; es por ello por lo que es una gran opción para el desarrollo de 2Code.

Ilustración 24: Patrón de diseño MVC mediado [22]

En esta versión del patrón MVC, el controlador tiene que realizar un mayor trabajo ya que es el intermediario de todos lo mensajes entre la Vista y el Modelo. Al mismo tiempo, el controlador realiza todo el procesamiento de datos necesarios. Por ejemplo, si desde la Vista se solicita la evaluación de código es el Controlador el que realiza la compilación del código y ejecuta las pruebas, extrayendo los casos de prueba para un determinado problema desde Modelo.

Tanto la aplicación del Cliente como la del servidor implementan en el patrón MVC mediado. La aplicación cliente es un poco más particular ya que sólo maneja objetos JavaScript genéricos a diferencia de la aplicación del servidor en el que se tienen clases bien definidas en el Modelo. En el Cliente también se ha modificado lo que sería un controlador en el servidor, teniendo más bien un fichero que sirve como mediador (Ilustración 25) entre las peticiones de información de los componentes React y el servidor.



```
...

export async function fetchAllUsers() {
  try {
    const response = await axios.get("/user/all");
    return response.data;
  } catch (err) {
    throw err;
  }
}

export async function fetchUserDataByEmail(email) {
  try {
    const response = await axios.post("/user/check", {
      email,
    });
    return response.data;
  } catch (err) {
    throw err;
  }
}

  ...
```

Ilustración 25: Ejemplo de funciones intermedias entre la lógica de la interfaz de usuario (componentes React) y el servidor.

Este tipo de funciones son llamadas desde la lógica de los componentes que definen la interfaz de usuario, y que por tanto serían la Vista dentro de la aplicación cliente.

6.3.2. Subsistema de diseño

La vista del subsistema de diseño muestra la arquitectura del sistema siguiendo el patrón MVC mediado desde un punto de vista de más alto nivel (Ilustración 26).

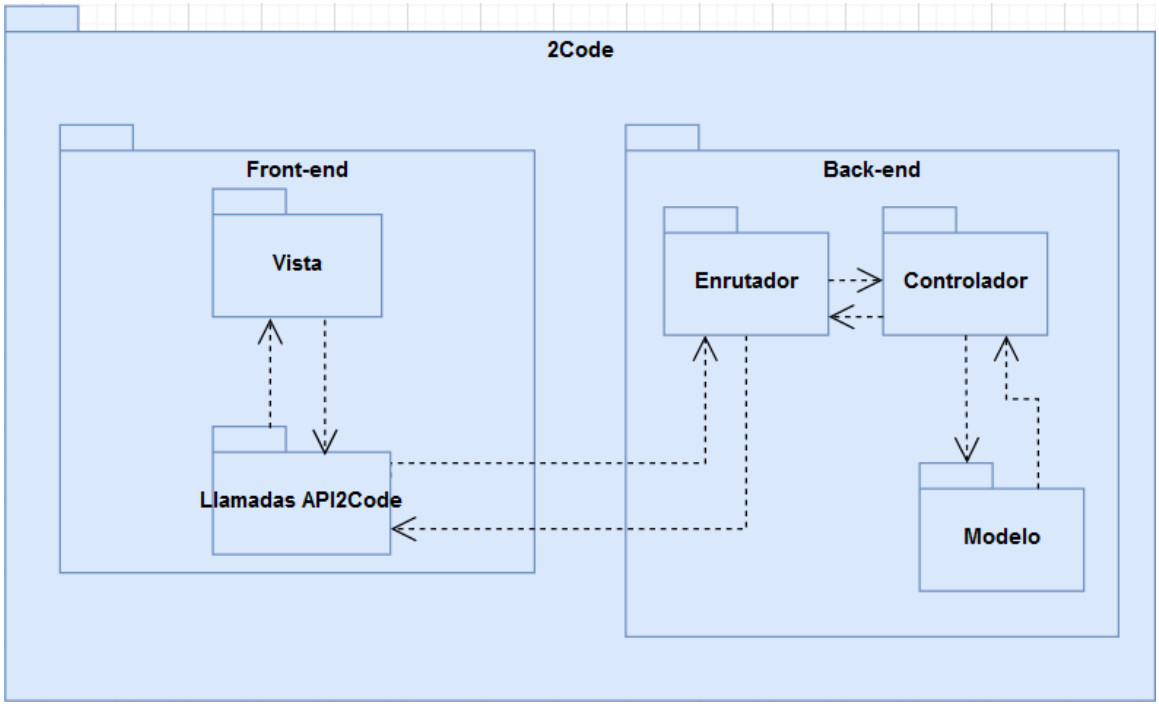


Ilustración 26: Vista arquitectónica de 2Code

En esta vista arquitectónica se puede ver la división y organización de los módulos más relevantes de la aplicación.

6.3.3. Diseño de las historias de usuario implementando el patrón MVC mediado

Con la arquitectura de 2Code definida, se pueden adaptar los diagramas de secuencia, definidos durante el análisis (Anexo I), al patrón MVC mediado. Con esto se obtienen los diagramas de historias de usuario del diseño (Ilustración 27) que definen la implementación durante el desarrollo del código. Estos diagramas de secuencia servirán para ir defiendo a grandes rasgos las interacciones de los distintos componentes que conforman el *front end* y el *back end.* La interfaz de la API se encontraría justo antes del enrutador o *router* de cada módulo definido en los diagramas de secuencia.

Para más definir a más detalle las rutas o *endpoints* de la API se utiliza el estándar de OpenAPI (Ilustración 28) que se explica más adelante.

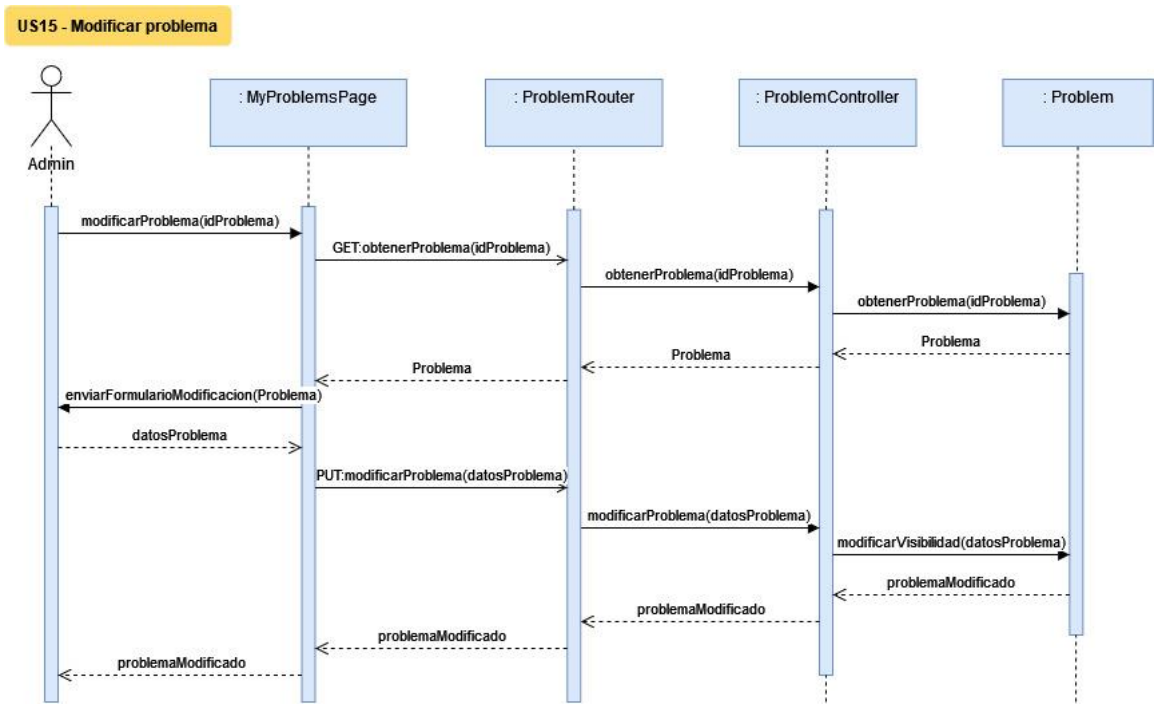


Ilustración 27: Diagrama de secuencia de diseño de la historia de usuario US15 – Modificar problema

6.3.4. Diseño e implementación de una REST API

Existen distintas arquitecturas para realizar APIs. Entre ellas se tiene *Remote Procedure Call* (RPC), *Simple Object Access Protocol* (SOAP), *Representational State Transfer* (REST) y otras más modernas como GraphQL. Algunas como RPC datan de los años 80 y otras, como GraphQL, tienen menos de una década desde que se crearon. Sin embargo, no hay duda de que, hasta hace poco REST fue la arquitectura por defecto para construir las APIs de servicios web. Existen múltiples motivos de su popularidad, entre ellos, la facilidad para entender e implementar una RESTful API, una alta transferencia de datos en poco tiempo al utilizar un formato compacto como JSON para enviar datos mediante HTTP, la no necesidad de tener un estado de la aplicación y por tanto poder procesar de forma independiente cada solicitud [24], entre otros. Igualmente, REST se ha extendido mucho por ser la arquitectura

implementada en Express, el *framework* para desarrollo de aplicaciones web. Por estas razones, se ha decidido desarrollar el *back end* de 2Code como una RESTful API.

```yaml
openapi: "3.0.0"
info:
  title: 2Code REST API
  description: Herramienta para evaluar ejercicios de programación.
  contact:
    name: Fabian Villalobos
    email: fvillalobos@usal.es
  version: 1.0.0
servers:
  - url: http://localhost:5000/api
paths:
  /problem:
    description: Operaciones relacionadas con los problemas de programación.
    get:
      summary: Solicita todas las preguntas en una versión reducida
      description: Solicita la lista actual de todas las preguntas que estan marcadas como visibles en
una versión reducida de parametros.
      responses:
        "200":
          description: "OK"
          content:
            application/json:
              schema:
                type: array
                minItems: 0
                items:
                  type: object
                  properties:
                    id:
                      type: integer
                      minimum: 1
                      description: id del problema
                    title:
                      type: string
                      example: problem1
                      description: titulo del problema
                    difficulty:
                      type: string
                      example: "10"
                      description: dificultad del problema
                example: [{
                  id: 1,
                  title: "Problema1",
                  difficulty: "10"
                },
                {
                  id: 2,
                  title: "Problema2",
                  difficulty: "30"
                },
                {
                  id: 3,
                  title: "Problema3",
                  difficulty: "10"
                }]
        "4XX":
          description: Error
```

Ilustración 28: Parte del fichero descriptivo de la API de 2Code siguiendo OAS v3.0.0

Utilizando los diagramas de secuencia y en conjunto con el diagrama de clases del Anexo I, se ha ido especificando distintos datos de los *endpoints* de la API. Siguiendo la OpenAPI Specification 3.0.0 (OAS3), para cada "camino" o *path/endpoint* de la API se puede especificar

el tipo de datos de la respuesta, el mensaje y código de la respuesta según para cada posible resultado de la operación (Ilustración 29), los tipos y nombres de las propiedades que se reciben en la solicitud e incluso se pueden proporcionar ejemplo de conjuntos de datos que se reciben en el cuerpo de la solicitud o que se envían en el cuerpo de la respuesta.

El proceso es largo, obteniendo un fichero yaml con la descripción inicial de la API de más de 600 líneas.

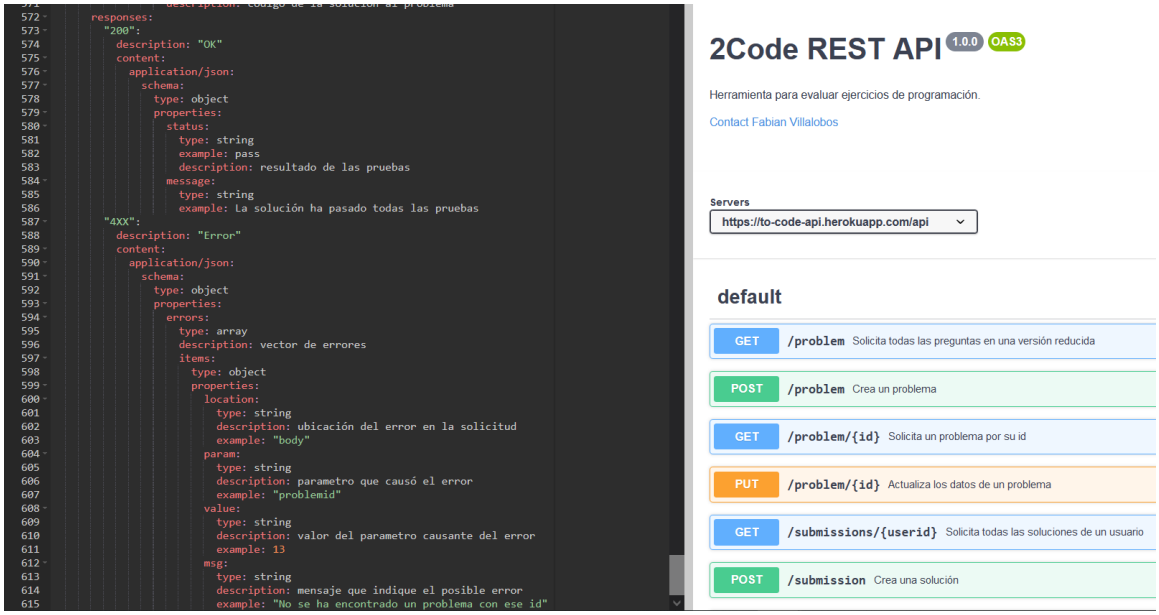


Ilustración 29: Especificación de tipos de respuestas para una solicitud en un endpoint de la API.

Este fichero no es la opción más legible para los humanos ya que está pensado en que sea interpretable por procesadores de código. Gracias a esto, se facilita la creación de distintos formatos y herramientas que pueden convertir este fichero en páginas web, documentación más legible por humanos como páginas PDF e incluso es posible obtener de algunas herramientas como Swagger, una implementación de una primera versión de la interfaz de la API en distinto lenguajes de programación.

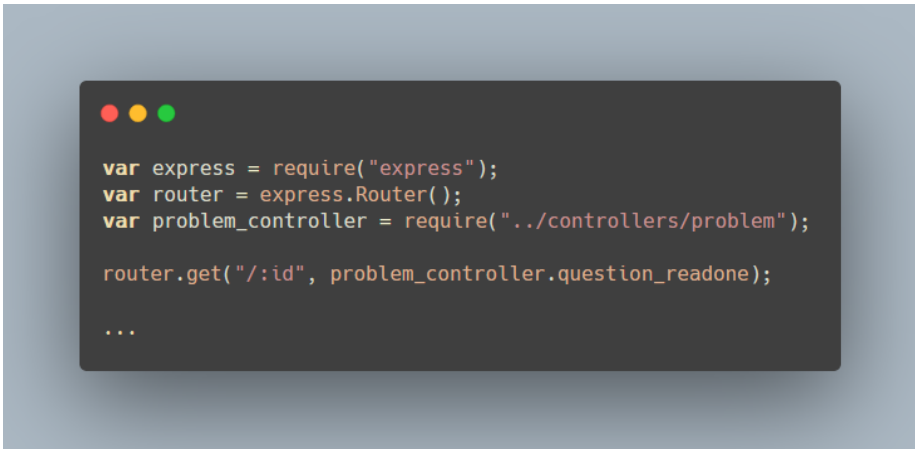
Gracias a que OAS se ha convertido en el estándar más ampliamente adoptado por la industria para describir APIs modernas [25], cualquier programador podría continuar utilizando y extendiendo la especificación de la API para mantenerla con mayor facilidad.

Uno de los beneficios de una RESTful API mencionados anteriormente, es la facilidad para manejar distintos conjuntos de solicitudes mediante la creación de módulos de rutas. En la implementación del enrutador, se ha divido las interacciones con la API en 5 módulos: autenticación, gestión de problemas, gestión de soluciones, gestión de usuario y funciones de administrador (Ilustración 30).

```
...

// test route to make sure everything is working (accessed through GET http://localhost:5000/api)
router.get("/", function (req, res) {
  res.json({ message: "Welcome to 2Code api!" });
});

router.use("/authentication", authentication);
router.use("/problem", problem);
router.use("/submission", submission);
router.use("/user", user);
router.use("/admin", admin);
}

  ...
```

Ilustración 30: División por módulo de las rutas en el enrutador de la API.

Dentro de cada módulo, el enrutador deja la lógica de las operaciones al controlador (Ilustración 31). A su vez, el controlador hace uso de los métodos y propiedades de las clases que representan los modelos de datos (Ilustración 32).



```
var express = require("express");
var router = express.Router();
var problem_controller = require("../controllers/problem");

router.get("/:id", problem_controller.question_readone);

...
```

Ilustración 31: El enrutador delega la lógica para consultar los datos de un problema al controlador.

```
...

exports.question_readone = async function (req, res) {
  try {
    const problem = await Problem.findProblemById(req.params.id);
    if (problem) {
      res.status(200).json(problem);
    } else {
      res.status(404).json({ errors: ["No problem for given id"] });
    }
  } catch (err) {
    return res.status(422).json({ errors: [err] });
  }
};

  ...
```

Ilustración 32: Función de controlador del módulo Problem que obtiene los datos de un problema y devuelve el objeto a la vista.

No se construye una clase únicamente con sus propiedades, sino que, como se mencionó antes, estas también poseen métodos para interactuar con la base de datos (Ilustración 33).

```
// /models/problem.js

// Find problem by ID
  static async findProblemById(id) {
    try {
      const data = await db.query(
        "SELECT * FROM public.problem WHERE id = $1",
        [id]
      );
      if (data.rowCount != 0) {
        const problem = Problem.createProblemFromObject(data.rows[0]);
        return problem;
      } else {
        return null;
      }
    } catch (err) {
      throw err;
    }
  }

  ...
```

Ilustración 33: Método dentro de la clase Problem para consultar información de una entrada en la base de datos.

6.4. Evaluación de soluciones

La evaluación de las soluciones enviadas por el usuario es probablemente una de las funcionas más importantes de 2Code. Inicialmente, en el Anexo I, se ha creado un diagrama de secuencia de análisis y posteriormente, al establecer la arquitectura de la aplicación, un diagrama de secuencia de diseño que muestran a desde una perspectiva de alto nivel el funcionamiento de la función. Sin embargo, el funcionamiento detallado de la implementación es mucho más complejo puesto que se ha tenido que considerar un diseño que soporte la posible inclusión de todo tipo de lenguajes de programación en el futuro.

Para las primeras versiones de la aplicación se ha decidido empezar por soportar soluciones escritas en JavaScript. Viendo que el entorno de ejecución de la aplicación del servidor es Node.js, no es necesario tener instalado un compilador de lenguaje de programación en el entorno de producción.

Al igual que el resto de las solicitudes que llegan a la API, la evaluación de una solución se recibe en el módulo “Submission” del enrutador (Ilustración 30). Desde este se pasa los parámetros de la solicitud al controlador, que a su vez separa la lógica de comprobación de parámetros necesarios para empezar la evaluación del problema y la misma función que evalua la solución.

6.4.1. Comprobación de parámetros

La comprobación de parámetros en el cuerpo de la solicitud es necesaria para evitar que el cliente no envíe una solicitud de evaluación para un problema que no existe en la base de datos, que no soporta el lenguaje de programación escogido o que el problema no se encuentre disponible. Metodos de la clase *Submission* en el modelo de datos se ocupan de las verificaciones de las propiedades del problema que se intenta resolver (Ilustración 34).

```
// /controllers/submission.js

exports.submission_run = async function (req, res) {
  try {
    // Search user id:
    const user = await User.findUserByEmail(req.body.email);
    if (!user) {
      // If user doesn't exist
      var error = new ValidationError(
        "body",
        "email",
        req.body.email,
        "USER_NOT_FOUND"
      );
      return res.status(404).json({ errors: [error] });
    }

    // Find submission's problem
    let problem = await
Problem.findProblemById(parseInt(req.body.problemId));
      var error = new ValidationError(
        "body",
        "problemId",
        req.body.problemId,
        "PROBLEM_NOT_FOUND"
      );
      return res.status(404).json({ errors: [error] });
    }

    // Search existing submission
    let submission = await Submission.findSubmissionByLanguageAndStatus(
      req.body.problemId,
      req.body.language,
      user.id,
      "pending"
    );

    ...
```

Ilustración 34: Comprobación de parámetros de la solicitud de evaluación de una solución.

6.4.2. Compilación de la solución

Como se ha mencionado antes, es necesario que durante la implementación se tenga en cuenta la posibilidad de soportar múltiples lenguajes de programación. Para ello es necesario poder utilizar distintos compiladores dependiendo el lenguaje de la solución [26].

Para resolver este problema se ha implementado un patrón de diseño de bajo nivel llamado *Factory Method*. Este patrón de diseño se utiliza comúnmente en la programación orientada a objetos ya que permite crear, a partir de una clase constructora, una subclase determinada según sea necesario.

```
// RunnerManager.js

class Factory {
  constructor() {
    this.createRunner = function createRunner(lang) {
      let runner;

      if (lang === "javascript") {
        runner = new JavaScriptRunner();
      } else if (lang === "c") {
        runner = new CRunner();
      } else if (lang === "go") {
        runner = new GoRunner();
      } else if (lang === "java") {
        runner = new JavaRunner();
      } else if (lang === "python") {
        runner = new PythonRunner();
      }

      return runner;
    };
  }
}

...
```

Ilustración 35: Implementación del patrón Factory Method para los distintos compiladores.

Los compiladores de los lenguajes de programación más utilizados no varían mucho entre ellos, es por ello por lo que la clase que se ocupa de realizar la compilación y la evaluación de las soluciones para cada lenguaje de programación puede implementarse como un subtipo de la clase constructora (Ilustración 35).

En la clase constructora se incluye una función "*Evaluate*" que conforma la parte inicial del proceso de evaluación (Ilustración 36). Esta función escribe los datos de la solución a evaluar y del problema en ficheros, lanza la función de evaluación con el subtipo adecuado para el lenguaje de programación y recibe de ella el resultado de la evaluación. Según si la solución ha compilado o no y si ha pasado todos los casos de prueba o no, devuelve un resultado correspondiente a la aplicación cliente.

```
// RunnerManager.js

exports.evaluate = function (submission, problem) {
  const factory = new Factory();
  const runner = factory.createRunner(submission.language);

  // ...

  runner.run(testFile,targetDir,testFileName,extension)
  .then(
    (status, message) => {
      if (status == "ok") { //ok => code compiled
        if (message.startsWith("[Success]")) {
          return {
            status: "pass",
            message: message.slice(9)
          }

        } else {
          return {
            status: "fail",
            message: message.slice(6)
          }
        }
      } else { // -> not ok => i.e. error during
compilatreturn {
          status,
          message
        }
      }
    }
  )
  .catch(err){
    throw err;
  }

// ...
```

Ilustración 36: Líneas relevantes de la función "Evaluate" compartida entre los subtipos de "Runner".

Cada evaluación de una solución creará un directorio temporal para alojar la solución y el fichero con la función que prueba la solución. Estos se deben crear nuevamente a cada nueva evaluación puesto que el código del usuario puede cambiar.

6.4.3. Prueba de la solución

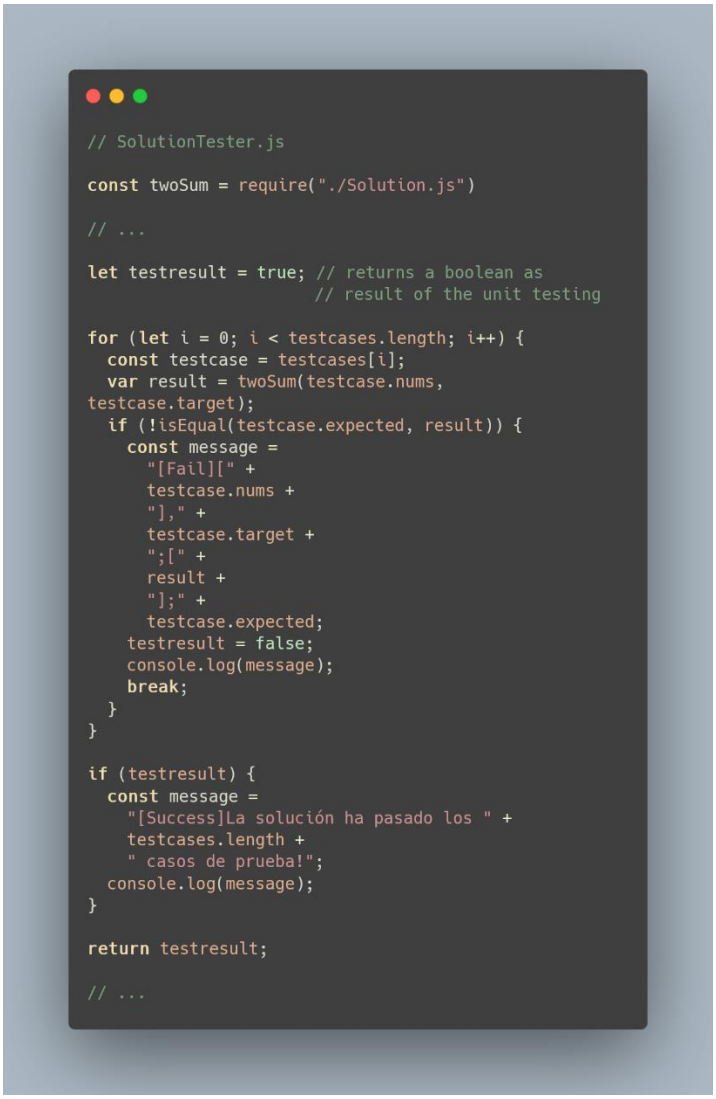
Para el evaluador de código de JavaScript se tiene la función “run” que recibe la URL de todos los ficheros relevantes para la evaluación de la solución. En esta función, se lanza un proceso hijo que compila y ejecuta el programa que prueba la solución del usuario (Ilustración 37).

```
// JavaScriptRunner.js

class JavaScriptRunner extends Runner {

  // ...

  run(file, directory, filename, extension) {
    if (extension.toLowerCase() !== ".js") {
      console.error(`${file} is not a javascript file.`);
    }
    // set solution directory for child process
    const options = { cwd: directory };
    const argsRun = [];
    argsRun[0] = file;

    const executor = spawn("node", argsRun, options);
    executor.stdout.on("data", (output) => {
      const out = String(output);
      if (out.startsWith("[Success]") || out.startsWith("[Fail]")) {
        return {
          status: "ok", // ok -> no error
          message: String(output)
        };
      }
    });
    executor.stderr.on("data", (output) => {
      return {
        status: "error_exec", // err -> execution failure
        message: String(output)
      };
    });
  }
}
```

Ilustración 37: Función Run en el subtipo que soporta soluciones en JavaScript

Como se puede ver en la Ilustración 37, el ejecutor del programa de prueba utiliza la orden "node [direcciónFichero] [direcciónDirectorioTemporal]".

El programa de prueba de la solución, *SolutionTester.js,* lee el fichero de pruebas unitarias, importa la función con la solución del usuario, pasa los conjuntos de datos de entrada a la solución de usuario y finalmente compara los resultados obtenidos con cada conjunto de entradas con los resultados esperados (también definidos en el fichero de pruebas unitarias) (Ilustración 38).



```
// SolutionTester.js

const twoSum = require("./Solution.js")

// ...

let testresult = true; // returns a boolean as
                       // result of the unit testing

for (let i = 0; i < testcases.length; i++) {
  const testcase = testcases[i];
  var result = twoSum(testcase.nums,
testcase.target);
  if (!isEqual(testcase.expected, result)) {
    const message =
      "[Fail][" +
      testcase.nums +
      "]," +
      testcase.target +
      ";[" +
      result +
      "];" +
      testcase.expected;
    testresult = false;
    console.log(message);
    break;
  }
}

if (testresult) {
  const message =
    "[Success]La solución ha pasado los " +
    testcases.length +
    " casos de prueba!";
  console.log(message);
}

return testresult;

// ...
```

Ilustración 38: Evaluación de la solución del usuario comparando resultados obtenidos y esperados.

El resultado de las pruebas es de tipo Booleano y se le asigna a la variable “testresult” el valor *false* si uno de los resultados de las pruebas no es igual al resultado esperado. Al mismo tiempo, dependiendo del valor de esta variable, se imprime un mensaje con los detalles de la ejecución que se puede leer desde el hilo padre. Al finalizar la ejecución del programa de prueba de la solución, el proceso hijo devuelve el resultado al proceso principal y muere.

Un objeto con el mensaje del resultado de la prueba es eventualmente devuelto hasta la función de evaluación del controlador y este a su vez devuelve el resultado a la aplicación cliente.

6.5. Despliegue de la aplicación

El modelo de despliegue consiste en la representación de las diferentes partes software y hardware que componen el total del sistema.

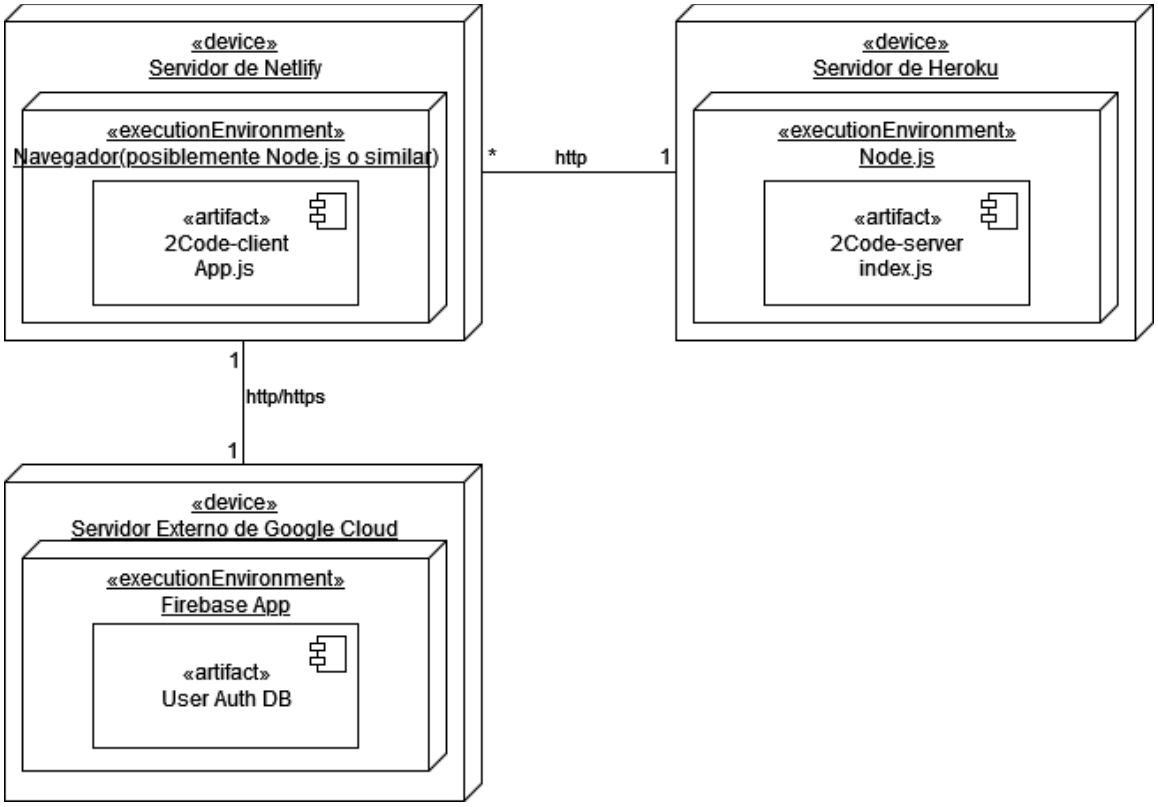


Ilustración 39: Diagrama de despliegue

En este diagrama se puede ver la relación entre los distintos dispositivos hardware, el software que contiene cada uno de ellos y su entorno de ejecución.

Se encuentra una descripción a más detalle de cada nodo en Anexo I – Modelo de despliegue.

6.5.1. Autenticación con Firebase

Hasta la versión actual de la aplicación, la autenticación es en su mayoría realizada desde la misma aplicación cliente que, a través de la API de Firebase para aplicaciones con Node.js,

realiza el alta de usuario, el inicio de sesión, la confirmación de dirección email y la autenticación en la aplicación.

Integrar el servicio de autenticación de Firebase es fácil ya que solo hace falta activarlo desde la consola del proyecto e instalar las librerías de firebase en la aplicación cliente. La gran ventaja de Firebase frente a una autenticación propia es que se puede añadir distintos métodos para iniciar sesión en la aplicación, como por ejemplo Iniciar Sesión con la cuenta de GitHub, simplemente activándolo desde la consola de Firebase. Por ahora, 2Code mantiene una autenticación únicamente por email y contraseña (Ilustración 40).

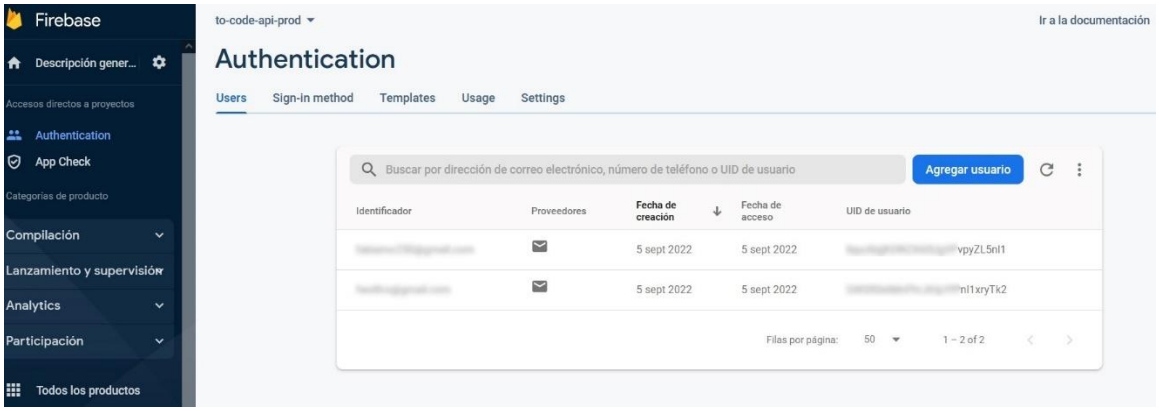


Ilustración 40: Menú del servicio de autenticación de Firebase. Lista de usuario registrados.

6.5.2. Aplicación servidor en Heroku

Para desplegar la aplicación que contiene el servidor se utiliza Heroku, una *Platform as a Service* que aloja la aplicación del *back end* simplemente clonando mediante Git la rama principal del proyecto al repositorio remoto que nos proporciona Heroku.

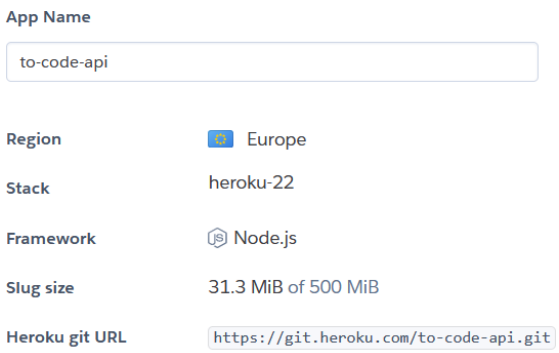


Ilustración 41: Datos del framework y repositorio remoto proporcionados por Heroku

Para permitir únicamente la conexión de la aplicación cliente a la API en el servidor de Heroku se han bloqueado, mediante el mecanismo de *Cross-Origin Resource Sharing* (CORS), todas las solicitudes que puedan llegar de otras direcciones de dominio que no provengan de la dirección de la aplicación cliente.

Para evitar almacenar datos de configuración importantes en el repositorio Git de 2Code, se pasan las variables de configuración mediante variables de entorno a las que se pueden asignar el valor desde la consola de Heroku (Ilustración 42).

Ilustración 42: Lista de variables de entorno para producción en Heroku

Finalmente, al clonar la rama principal al repositorio remoto de Heroku, se inicia la construcción de la versión de producción y se inicial su ejecución.

6.5.3. Aplicación cliente en Netlify

Netlify es similar a Heroku puesto que es una PaaS que aloja aplicaciones de Node.js, entre otros lenguajes. Sin embargo, Netlify está mucho más optimizado para aplicaciones web *front*

end ya que permite incluir el código JavaScript, independientemente del *framework* o librería de interfaz de usuario escogida, en un servidor que únicamente se encarga de enviar las páginas a los usuarios. Muchos *frameworks* como Angular o Next.js no necesitan esta función, pero al utilizar React, es una forma de simplificar el despliegue de la aplicación.

Al igual que Heroku, es posible conectar un repositorio en GitHub del cual Netlify clona la rama principal a su propio repositorio remoto, o bien se puede empujar manualmente la rama principal utilizando Git.

Para modificar la URI a la que la aplicación cliente se conecta con la API dinámicamente, se ha utilizado, al igual que en el *back end*, variables de entorno que determinan si se está ejecutando la aplicación en un servidor de desarrollo o de producción. React modifica la variable que determina el tipo de entorno a la hora de producir la versión de producción. Otras variables de entorno se pueden especificar desde la consola de Netlify (Ilustración 43).

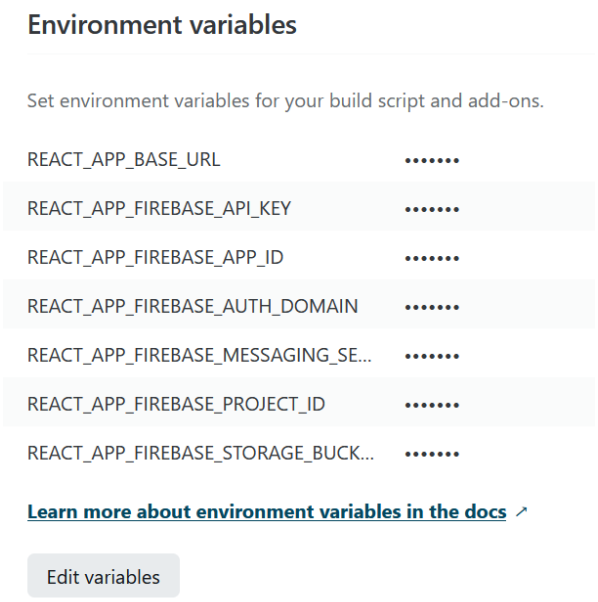


Ilustración 43: Variables de entorno en Netlify

La URL que nos proporciona Netlify para acceder a la aplicación cliente es: https://resonant-nasturtium-032ce8.netlify.app/

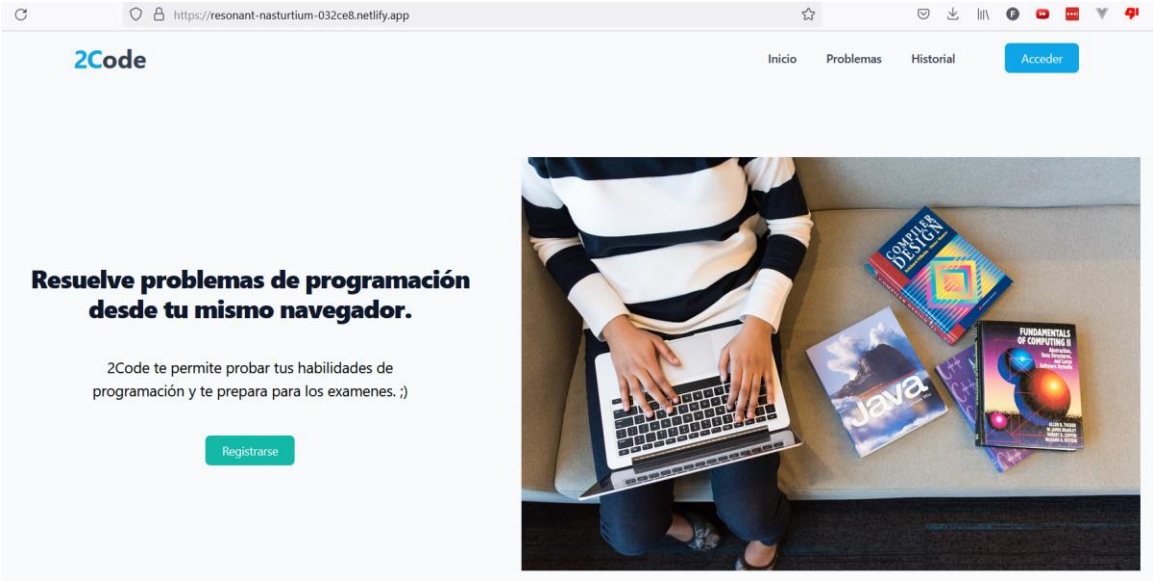


Ilustración 44: Aplicación cliente alojada y desplegada en Netlify

7. CONCLUSIONES

A diferencia de un compilador, un evaluador de código es un *software* muy particular ya que para cada posible ejercicio de programación que se incluye en la librería de 2Code, es necesario poder determinar si la función escrita por el usuario es una solución del ejercicio de programación. La gran problemática está en que probablemente existen cientos de formas de resolver un ejercicio de programación y por tanto cientas de posibles soluciones que el usuario podría escribir. 2Code es una propuesta de cómo afrontar este desafío, y para ello se ha explicado la metodología de trabajo utilizada para su desarrollo, las herramientas utilizadas en la organización del proyecto y para su implementación, y una arquitectura que puede soportar un evaluador de código en la web similar al de las grandes empresas.

Las tecnologías utilizadas en el desarrollo de 2Code no son extrañas a aquellos que han trabajado en programación de aplicaciones web. Muchas de ellas, si es que no la mayoría, están bien establecidas en la industria y por tanto permiten alcanzar mayores estándares de calidad, optimización y mantenimiento. A pesar de ello, existen nuevas herramientas que podrían facilitar mucho más ciertos aspectos de este trabajo, como, por ejemplo, en la elección de React como *framework* de interfaz de aplicación web. Ahora se tienen opciones como Angular o NEXT.js que facilitan mucho más a la hora desplegar la aplicación ya que llevan incluidas funciones como *server-side rendering*.

La arquitectura implementada en 2Code es también una elección frecuente en el desarrollo de aplicaciones web, ya que permite un operar con una gran cantidad de datos. La arquitectura MVC mediada ha permitido realizar operaciones en ambos sentidos del flujo de información ya que esta pasa tanto por la entrada como en la salida del servidor a través del Controlador. Esto permite alojar toda la lógica importante de la aplicación del servidor en el mismo Controlador.

Durante el desarrollo de los metodos que se ocupan de la evaluación de las soluciones se ha constatado que no hay una forma práctica para permitir que los usuarios puedan crear sus propios problemas de programación y que 2Code pueda utilizarlos para evaluar soluciones. Esto se debe a que es necesario conocer cómo es que se prueba una solución en el servidor. Cada función de prueba tiene que leer los casos de prueba y pasar a la función evaluada el

parámetro correcto. El problema de este acercamiento para crear el evaluador es que cada problema de programación puede necesitar distintas cantidades y distintos tipos de parámetros de entrada, y a su vez, distintos tipos de valores de salida. Por tanto, cada función de evaluación tiene que estar adaptado al problema de programación mismo y no se puede crear una función de evaluación genérica para todos los problemas.

La necesidad de tener que conocer el funcionamiento del evaluador de código para crear nuevos problemas es un problema que incluso las mismas grandes empresas del sector no saben cómo abordar. Aplicaciones web como Leetcode [1] o AlgoExpert [3] que dominan el mercado de evaluadores de código, tampoco permiten a sus usuarios de base crear sus propios problemas de programación. Por otro lado, HackerRank [2] está destinado a que empleadores creen sus propios problemas de programación para utilizarlos en entrevistas de trabajo, sin embargo, su aproximación a este problema es un secreto industrial y no se tiene mucha información en internet sobre cómo lo hacen, por lo que puede ser que también sea una aproximación poco práctica.

En 2Code se han creado dos posibles soluciones a este problema. Con la primera se permite a los administradores de la página crear nuevos problemas de programación desde la misma interfaz gráfica de la aplicación, sin embargo, es necesario ingresar manualmente un archivo en el código fuente del programa que contenga la función de evaluación específica para el problema de programación. Con la segunda solución se puede subir este el fichero con la función de evaluación directamente en la interfaz gráfica de la aplicación, pero por detrás, igualmente el archivo es incluido en el código fuente de la aplicación de una forma automática. Las soluciones se diferencian a la hora de corregir errores sobre la función de evaluación, ya que si se incluye un fichero con errores de compilación se podría desencadenar errores de ejecución en la versión desplegada. En cambio, al subir los archivos manualmente, se puede crear mediante Git una rama de prueba de la aplicación y así verificar que el nuevo ejercicio de programación no ocasiona errores de ejecución antes de incluirlo en la rama principal.

Finalmente, 2Code ha sido un proyecto multidisciplinar ya que ha sido necesario abordar un gran rango de aspectos de un sistema informático. Es por ello por lo que se lo califica de un trabajo *full-stack*, que en muchas empresas se dividen entre varios empleados. Pese a ello es

muy enriquecedor poder conocer toda la complejidad y profundidad que hay detrás de la interfaz gráfica de una página web.

Personalmente, he iniciado este proyecto sin tener conocimientos de JavaScript, React, Node, Express y muchas de las otras tecnologías utilizadas. Sin embargo, este proyecto ha sido un ejemplo de lo que se puede construir a través de la información gratuita que internet pone a disposición de todos.

8. LÍNEAS DE TRABAJO FUTURAS

Durante el desarrollo de 2Code han surgido muchas posibles mejoras de la aplicación. Entre estas destaca la que se ha mencionado previamente: la aproximación utilizada para añadir problemas de programación a la aplicación.

Asimismo, sería una buena opción adaptar este proyecto a TypeScript ya que los objetos JavaScript no discriminan el tipo de dato que se asignan a sus propiedades. En 2Code, por ejemplo, si se asigna un valor del tipo equivocado a una propiedad de una instancia de la clase Usuario, es posible que el error pase desapercibido hasta la ejecución del programa. TypeScript por otro lado, posee *strong typing definition*, por lo que se pueden evitar errores de ese tipo.

El mantenimiento de 2Code puede simplificarse al utilizar TypeScript junto con NEXT.js ya que tanto funciones como clases tendrían una interfaz que describe exactamente el esquema que poseen. NEXT por otro lado, trae consigo librerías para el enrutamiento de las URLs de las páginas de la aplicación y un servidor para el despliegue de la aplicación.

Finalmente, 2Code utiliza Firebase para realizar la autenticación de los usuarios en la aplicación, sin embargo, los metodos de la API de Firebase tiene efectos secundarios que no son siempre deseables. Por ejemplo, el método de su API para registrar un nuevo usuario hace que este inicie sesión automaticamente. Esto complica la lógica al añadir la comprobación de la propiedad de la dirección email ya que da acceso inmediato del usuario a la aplicación. Es por esto que una posible mejor sería mantener la autenticación a la

aplicación de una forma local, almacenando las contraseñas en la base de datos de 2Code y, por ejemplo, utilizando JWT Auth para mantener la autenticación en la aplicación cliente.

9. BIBLIOGRAFÍA

[1] LeetCode, «LeetCode,» 2022. [En línea]. Available: https://leetcode.com/.

[2] HackerRank, «HackerRank,» 2022. [En línea]. Available: https://www.hackerrank.com/. [Último acceso: Septiembre 2022].

[3] A. LLC, «AlgoExpert,» 2022. [En línea]. Available: https://www.algoexpert.io/product. [Último acceso: Septiembre 2022].

[4] ProyectosAgiles.org, «proyectosagiles.org,» [En línea]. Available: https://proyectosagiles.org/que-es-scrum/. [Último acceso: Septiembre 2022].

[5] Mozilla, «Mdn web docs,» [En línea]. Available: https://developer.mozilla.org/es/docs/Web/JavaScript. [Último acceso: Septiembre 2022].

[6] Microsoft, «TypeScript,» [En línea]. Available: https://www.typescriptlang.org/. [Último acceso: Septiembre 2022].

[7] I. Meta Platforms, «React,» [En línea]. Available: Microsoft. [Último acceso: Septiembre 2022].

[8] I. Vercel, «NEXT.js,» [En línea]. Available: https://nextjs.org/. [Último acceso: Septiembre 2022].

[9] E. You, «Vue.js,» [En línea]. Available: https://vuejs.org/. [Último acceso: Septiembre 2022].

[10] Google LLC, «V8 JavaScript Engine,» 2008. [En línea]. Available: https://v8.dev/. [Último acceso: Febrero 2022].

[11] Amazon Web Services, Inc., «Servicios de integración de aplicaciones,» [En línea]. Available: https://aws.amazon.com/es/what-is/api/. [Último acceso: Junio 2022].

[12] M. Sliger, «Agile estimation techniques,» de *PMI® Global Congress 2012*, North America, Vancouver, British Columbia, Canada. Newtown Square, 2012.

[13] B. Grepon, N. Baran, K. Gumonan, A. Martinez y M. Lacsa, «Designing and implementing e-schools system: An information systems approach to school management of a community college in northen Mindanao, Philippines.,» *International Journal of Computing Sciencese Research, in press,* 2021.

[14] S. Labs, «kabantool,» [En línea]. Available: https://kanbantool.com/es/metodologia-kanban. [Último acceso: Septiembre 2022].

[15] Mozilla. [En línea].

[16] PostgreSQL, «PostgreSQL Documentation,» 1996. [En línea]. Available: https://www.postgresql.org/docs. [Último acceso: 16 Julio 2022].

[17] Expressjs, «Express documentation,» 2017. [En línea]. Available: https://expressjs.com/. [Último acceso: 16 Julio 2022].

[18] Tailwindcss, «Tailwindcss,» [En línea]. Available: https://tailwindcss.com/. [Último acceso: Septiembre 2022].

[19] Wikipedia, «Heroku,» [En línea]. Available: https://es.wikipedia.org/wiki/Heroku. [Último acceso: Septiembre 2022].

[20] Swagger To Pdf, «Swagger To Pdf,» [En línea]. Available: https://www.swdoc.org/. [Último acceso: 16 Julio 2022].

[21] J. Harich, «Unified Process Introduction,» 25 Agosto 1999. [En línea]. Available: https://www.thwink.org/soft/info/process/unified/UnifiedIntroduction.html. [Último acceso: 7 Julio 2022].

[22] J. Bucanek, «Model-View-Controller Pattern,» de *Learn Objective-C for Java Developers*, Apress, 2009, pp. 353-357.

[23] S. Borini, «stefanoborini,» de *Understanding Model View Controller*, 2019, pp. Sección 2.5.1 - Model-View-Adapter (MVA, Mediated MVC, Model-Mediator-View).

[24] freeCodeCamp, «Benefits of Going RESTful,» [En línea]. Available: https://www.freecodecamp.org/news/benefits-of-rest/. [Último acceso: Septiembre 2022].

[25] OpenAPI Initiative, «OpenAPI Documentation,» OpenAPI Initiative, [En línea]. Available: https://oai.github.io/Documentation/introduction.html. [Último acceso: 15 Julio 2022].

[26] R. Zhuang, «Online Judge - Judging System,» [En línea]. Available: https://jojozhuang.github.io/tutorial/online-judge-judging-system/. [Último acceso: Mayo 2022].

[27] H. W. Gellersen y M. Gaedke, «Object-oriented web application development,» *IEEE Internet Computing, 3(1),* pp. 60-68, 1999.

[28] O. Foundation, «Express,» [En línea]. Available: https://expressjs.com/. [Último acceso: Septiembre 2022].