Juez multilenguaje para el aprendizaje de la programación
Language-agnostic online judge for programming learning

Trabajo de Fin de Grado
Curso 2021–2022

Autores
Marta Estévez Bravo (Grado de Ingeniería Informática)
Pablo Morientes Lavín (Grado de Ingeniería del Software)
Víctor Manuel Cavero Gracia (Grado de Ingeniería Informática)

Directores
Enrique Martín Martín
Manuel Montenegro Montes

Facultad de Informática
Universidad Complutense de Madrid

Juez multilenguaje para el aprendizaje de la programación
Language-agnostic online judge for programming learning

Trabajo de Fin de Grado
Departamento de Sistemas Informáticos y Computación

Autores
Marta Estévez Bravo (Grado de Ingeniería Informática)
Pablo Morientes Lavín (Grado de Ingeniería del Software)
Víctor Manuel Cavero Gracia (Grado de Ingeniería Informática)

Directores
Enrique Martín Martín
Manuel Montenegro Montes

Convocatoria: Junio 2022

Facultad de Informática
Universidad Complutense de Madrid

Agradecimientos

A Manuel y Enrique por ayudarnos y mostrarnos el camino durante todo el desarrollo. A Marco y Pedro por haber creado esta plantilla. A nuestros familiares por apoyarnos en los momentos más difíciles a lo largo de la carrera, sin ellos no estaríamos aquí ahora mismo.

Resumen

Juez multilenguaje para el aprendizaje de la programación

ScholarJudge es un juez de programación que permite a los estudiantes practicar cualquier lenguaje de programación que estén aprendiendo. A diferencia de otros jueces tradicionales, en *ScholarJudge* la solución se ejecuta en el lado del cliente y, posteriormente, se envía al servidor para validar su corrección. Durante este proyecto se han desarrollado un servidor, un cliente de escritorio y una aplicación web. El cliente de escritorio permite a los estudiantes obtener los problemas y enviar sus soluciones, y la aplicación web permite a los profesores crear nuevos problemas y almacenarlos en el servidor. La comunicación entre clientes y servidor se realiza mediante una API REST. Para acceder a las aplicaciones web y de escritorio es necesario registrarse primero.

En los siguientes repositorios se encuentra el código del cliente de escritorio y del servidor:

`https://github.com/ScholarJudge/ClientApp`

`https://github.com/ScholarJudge/ServerApp`

Palabras clave

Juez automático, aprendizaje de la programación, resolución de problemas, lenguajes de programación, Node.js, aplicación web, interfaz de línea de comandos, CLI

Abstract

Language-agnostic online judge for programming learning

ScholarJudge is a programming judge that allows to students to practice any programming language they are learning. Unlike other traditional judges, in *ScholarJudge* the solution is executed on the client side and then sent to the server to validate its correctness. During this project, a server, a desktop client and a web application have been developed. The desktop client allows students to get problems and send their solutions, and the web application allows teachers to create new problems and save them in the server. Comunication between clients and server is done through a REST API. To access the web and desktop applications you need to register first.

The server and desktop client code is located in the following repositories:

`https://github.com/ScholarJudge/ClientApp`

`https://github.com/ScholarJudge/ServerApp`

Keywords

Automatic judge, programming learning, problems solving, programming languages, Node.js, web application, command line interface, CLI

Índice

1. Introducción **1**

1.1. Motivación . . . 1

1.2. Objetivos . . . 2

1.3. Plan de trabajo . . . 2

1.4. Estructura de la memoria . . . 4

2. Introduction **5**

2.1. Motivation . . . 5

2.2. Objectives . . . 5

2.3. Workplan . . . 6

2.4. Memory structure . . . 6

3. Preliminares **9**

3.1. Los jueces de programación . . . 9

3.2. Tecnologías utilizadas . . . 10

3.2.1. Node.js . . . 10

3.2.2. Express . . . 11

3.2.3. API REST . . . 11

3.2.4. MongoDB . . . 11

3.2.5. JWT . . . 12

3.2.6. HTML y CSS . . . 12

3.2.7. JavaScript y jQuery . . . 12

3.2.8. Bootstrap . . . 12

3.2.9. Git y GitHub . . . 13

3.2.10. LaTeX . . . 13

3.2.11. Heroku . . . 13

3.2.12. MTCaptcha . . . 13

4. Organización del sistema **15**

4.1. Definición de *ScholarJudge* . . . 15

4.1.1. Problemas . . . 15

4.1.2. Usuarios . . . 15

4.2. Arquitectura de *ScholarJudge* . . . 16

4.2.1. Cliente de escritorio . . . 17

4.2.2. Cliente Web 17
4.2.3. Servidor 17
4.2.4. Conexión cliente-servidor 18

5. Servidor **29**
5.1. Base de datos 29
5.1.1. Mongo Atlas 29
5.1.2. Mongoose 30
5.2. Comunicación con el Cliente web 31
5.2.1. Express 31
5.2.2. Manejo de sesiones 32
5.2.3. *Middlewares* 32
5.2.4. Roles 33
5.3. Comunicación con la App del Cliente 34
5.3.1. JSON Web Tokens - JWT 34
5.4. Creación de plantillas y renderizado dinámico 34
5.5. Corrección de envíos 35
5.6. Generación de checksums 35

6. Clientes de interfaz de línea de comandos **37**
6.1. Desarrollo de los clientes 37
6.2. Aspectos comunes a los clientes de línea de comandos 37
6.2.1. El fichero *scholarJudge.json* 38
6.2.2. Gestión del *JWT* 39
6.3. Interfaz de linea de comandos 39
6.3.1. Funcionalidades 39
6.3.2. Descripción técnica 42
6.4. Interfaz de linea de comandos interactivo 42
6.4.1. Funcionalidades 42
6.4.2. Descripción técnica 43
6.5. Historia de usuario básica común 43
6.6. Llamadas al servidor 43

7. Aplicación web **45**
7.1. *Hosting* 45
7.2. Librerías utilizadas 46
7.3. Vistas 46
7.3.1. Menú de navegación 46
7.3.2. Vistas principales 47

8. Conclusiones y trabajo futuro **55**
8.1. CLI 55
8.2. Aplicación web 55
8.3. Autenticación 56
8.4. Roles 56
8.5. Desafíos encontrados 56
8.6. Trabajo futuro 57

9. Conclusions and Future Work **61**
9.1. CLI . . . 61
9.2. Web application . . . 61
9.3. Authentication . . . 62
9.4. Roles . . . 62
9.5. Challenges . . . 62
9.6. Future work . . . 63

10. Contribuciones personales **67**
10.1. Marta Estévez Bravo . . . 67
10.1.1. Investigación . . . 67
10.1.2. Diseño . . . 67
10.1.3. Implementación . . . 68
10.1.4. Pruebas . . . 68
10.1.5. Memoria . . . 69
10.2. Pablo Morientes Lavín . . . 69
10.2.1. Desarrollo del CLI interactivo . . . 69
10.2.2. Envío y corrección de problemas . . . 69
10.2.3. Autenticación de usuarios en la web . . . 69
10.2.4. Diseño de la interfaz web . . . 70
10.2.5. Desarrollo de *scripts* de ayuda . . . 70
10.2.6. Testing . . . 71
10.3. Víctor Manuel Cavero Gracia . . . 71
10.3.1. Creación del proyecto e instalación de los paquetes . . . 71
10.3.2. Construcción y definición del CLI . . . 72
10.3.3. Gestión de las versiones de los paquetes utilizados . . . 72
10.3.4. El fichero *scholarJudge.json* . . . 72
10.3.5. Gestión de JWT . . . 72
10.3.6. Aportación al cliente web . . . 73
10.3.7. Añadir nuevos problemas . . . 73

Bibliografía **75**

Índice de figuras

1.1. Diagrama de Gantt: planificación primer cuatrimestre 3
1.2. Diagrama de Gantt: planificación segundo cuatrimestre 3

2.1. Gantt Chart: first quarter planning 7
2.2. Gantt Chart: second quarter planning 7

3.1. Subcategorías de problemas de ¡Acepta el reto! 10

4.1. Arquitectura de ScholarJudge 16

5.1. MongoAtlas: gráfico 30
5.2. Checksum: Script generador de checksums 35

6.1. CLI Comando init 40
6.2. CLI Comando contests 40
6.3. CLI Comando problems 40
6.4. CLI Comando next 41
6.5. CLI Comando stats 41
6.6. CLI Comando problem 41
6.7. CLI Comando check 41
6.8. CLI Texto de ayuda 42
6.9. Terminal menú principal 43

7.1. Plan de *hosting* gratuito de *Heroku* 45
7.2. ScholarJudge: vista del menú en modo administrador o profesor 46
7.3. ScholarJudge: vista del menú desplegable 47
7.4. ScholarJudge: vista del menú colapsado 47
7.5. ScholarJudge: vista del inicio de sesión 48
7.6. ScholarJudge: vista de registro 48
7.7. ScholarJudge: vista de concursos 49
7.8. ScholarJudge: vista de administrador y profesor de concursos 49
7.9. ScholarJudge: vista de administrador y profesor de añadir concurso 50
7.10. ScholarJudge: vista de administrador y profesor de editar concurso 50
7.11. ScholarJudge: vista de problemas 50
7.12. ScholarJudge: vista de administrador y profesor de problemas 51
7.13. ScholarJudge: vista de administrador y profesor de añadir problema 51

7.14. ScholarJudge: vista de administrador y profesor de editar problema 52
7.15. ScholarJudge: vista de estadísticas 52
7.16. ScholarJudge: vista de usuarios 52
7.17. ScholarJudge: vista de editar perfil 53

10.1. Interfaz web: página de concursos 70
10.2. Interfaz web: página de concursos rediseñada 70
10.3. Paleta de colores de ScholarJudge generada mediante Huemint 73

Capítulo 1

Introducción

En este capítulo se pondrá en contexto la motivación, objetivos y plan de trabajo para el desarrollo del *Juez multilenguaje para el aprendizaje de la programación*, al cual se le ha bautizado con el nombre de *ScholarJudge*.

1.1. Motivación

El aprendizaje de la programación, al igual que muchas otras materias, suele ser un camino arduo y con mucha incertidumbre para aquellos que son nuevos en este sector. En concreto, la programación exige abstracción, cierta creatividad y oficio. Una definición soslayada del *Software* sería aquella que la define como un conjunto de instrucciones que permiten a la computadora realizar determinadas tareas. Este punto es en el que nos vamos a centrar en este TFG.

El camino tradicional que se presenta a un alumno para que aprenda y refuerce sus conocimientos, es realizar ejercicios sobre un tema específico. Estos ejercicios resuelven un *tipo de problema* con unos *datos específicos*. No obstante, la programación permite resolver un *tipo de problema* para *cualquier conjunto válido de datos*. Por consiguiente, un estudiante de programación, para poder validar su solución a un problema, deberá comprobar su programa para cualquier conjunto válido de datos. Esto último, en general, no es posible y aunque hay maneras formales de determinar si una porción de código resuelve de manera exitosa el problema para cualquier conjunto de datos, entendemos que a un alumno novel se le escapa de las manos. Una alternativa sería escoger un conjunto de datos o casos de prueba suficientemente fiables, es decir, que contenga un conjunto razonablemente exhaustivo de casos de prueba, incluyendo casos límite. De nuevo, la tarea de escoger casos de prueba suficientemente fiables requiere mucho tiempo y puede escaparse de las capacidades del alumno.

Para determinar la validez de un programa existen los llamados jueces de programación. No obstante, estos tienen una limitación importante. Esta limitación consiste en que dichos jueces solo podrán emitir veredictos a soluciones programadas en los lenguajes para los que están preparados. Esto supone que para cada lenguaje, e incluso para diferentes versiones del mismo, se necesite desarrollar la funcionalidad para poder corregir las soluciones programadas en dicho lenguaje. De la necesidad de poder aprender a programar en cualquier lenguaje haciendo uso de un juez, nace la idea de *ScholarJudge*.

1.2. Objetivos

El objetivo principal de este proyecto es desarrollar un *Juez multilenguaje de programación* para que los estudiantes puedan reforzar y ampliar sus conocimientos sobre la programación de manera práctica. Las metas concretas de ScholarJudge son:

- Creación de una aplicación de escritorio de uso sencillo, para que el estudiante pueda descargar problemas desde un servidor que contenga un catálogo de problemas, ver sus estadísticas y comprobar sus soluciones indiferentemente del lenguaje de programación que se use.
- Creación de una aplicación web con diseño adaptable tanto a ordenadores como a dispositivos tipo tablet, para que los profesores puedan gestionar los concursos, problemas y usuarios. A su vez, los estudiantes podrán visualizar de una manera más atractiva la información referente a los concursos, problemas y estadísticas propias.
- Disponer de un sistema de autenticación para que cada estudiante pueda llevar un registro de sus envíos y estadísticas.
- Establecer un sistema de roles que permita a los profesores y administradores acceder a funcionalidades de gestión de contenido a las cuales los estudiantes no deberían poder tener acceso.

1.3. Plan de trabajo

El desarrollo de este proyecto abarca desde septiembre de 2021 hasta junio de 2022. Para la planificación de este proyecto se ha seguido una metodología de trabajo ágil, adaptando la forma de trabajo a las condiciones del proyecto según se requiera en cada momento. Dado que todos los integrantes de este proyecto han estado trabajando parcial o totalmente durante el transcurso de este, la metodología escogida ha otorgado más flexibilidad y una gestión del tiempo mas eficiente.

Se han ido realizando *sprints* durante todo el proceso de desarrollo del proyecto con duraciones que abarcan desde 1 hasta 3 semanas. Estos finalizan con una reunión en la que todos los participantes del proyecto están presentes. En dichas reuniones los estudiantes muestran el trabajo realizado durante el *sprint*, los tutores revisan las tareas y generan un *feedback*. Acto seguido se analiza la situación actual del proyecto y se determinan las siguientes tareas a realizar así como que tecnologías nuevas investigar, sí asi se requiere.

En cada *sprint* se han realizado de manera conjunta las siguientes tareas:

- Análisis: elección de las tareas a desarrollar en el nuevo *sprint*.
- Investigación: elección e hincapié en nuevas tecnologías, ya sean nuevas o utilizadas anteriormente, para poder resolver las tareas del nuevo *sprint*.
- Implementación: desarrollo a la par de la aplicación de escritorio y de la aplicación web.
- Pruebas: en su mayoría pruebas de aceptación y en determinados casos, pruebas unitarias.

En la siguientes figuras se muestran dos diagramas de Gantt correspondientes a las tareas realizadas en el primer y segundo cuatrimestre respectivamente.

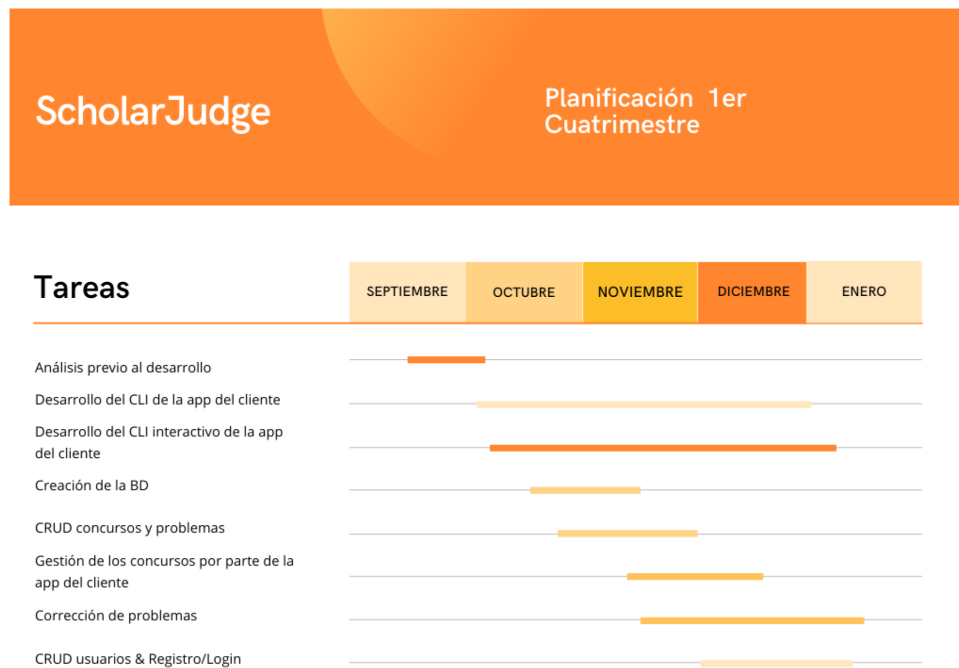


Figura 1.1: Diagrama de Gantt: planificación primer cuatrimestre

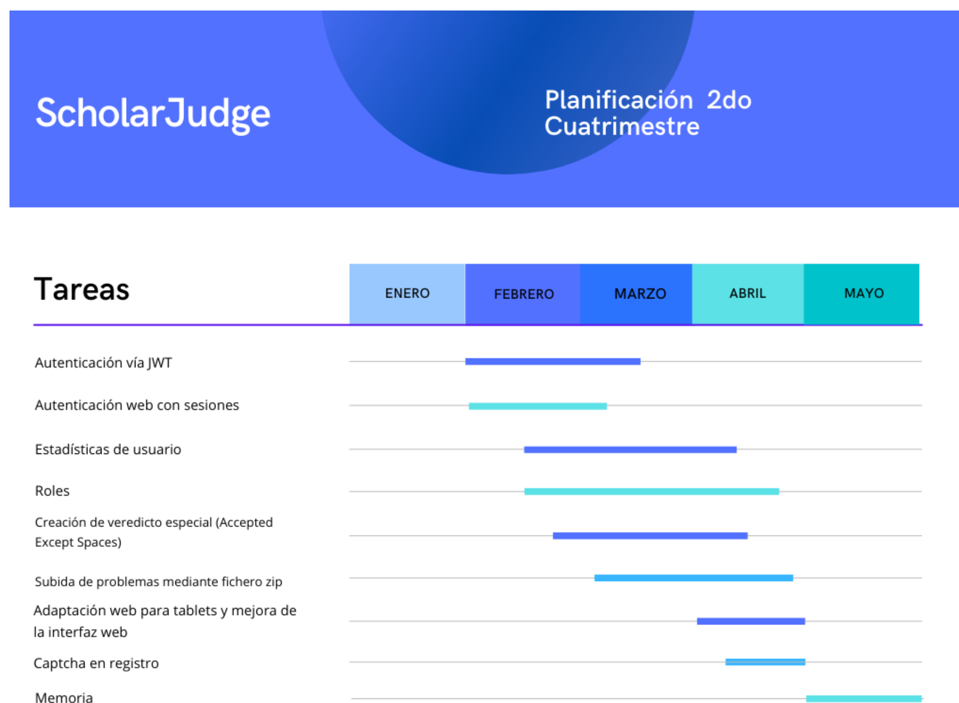


Figura 1.2: Diagrama de Gantt: planificación segundo cuatrimestre

1.4. Estructura de la memoria

El resto de la memoria se estructura de la siguiente manera:

- En el capítulo 3 se hablará acerca de los jueces de programación, su importancia en el aprendizaje y las limitaciones para soportar nuevos lenguajes. También se presentan las principales tecnologías utilizadas en el proyecto.
- En el capítulo 4 estudiaremos la organización del sistema, de qué componentes está compuesto y cómo están interconectados.
- En el capítulo 5 se detallan las funciones del servidor y su estructura interna.
- En el capítulo 6 se detallan las funciones de la aplicación del cliente, su estructura interna y como el usuario puede interactuar con el.
- En el capítulo 7 se estudiará en detalle la aplicación web.
- En el capítulo 8 se hablará de las conclusiones y trabajos futuros para este proyecto.

Chapter 2

Introduction

This chapter will put into context the motivation, objectives and work plan for the development of the Language-agnostic Judge for learning programming, which has been named ScholarJudge.

2.1. Motivation

Learning programming, like many other subjects, is often an arduous and uncertain path for those new to the industry. In particular, programming requires abstraction, a certain amount of creativity and skill. An overlooked definition of software would be the one that defines it as a set of instructions that allow the computer to perform certain tasks. This is the point we are going to focus on this TFG.

The traditional way for a student to learn and reinforce his knowledge is to perform exercises on a specific topic. These exercises solve a type of problem with specific data. However, programming makes it possible to solve a type of problem for any valid set of data. Therefore, a student of programming, in order to validate his solution to a problem, must test his program for any valid set of data. The latter is not possible, and although there are formal ways to determine whether a piece of code successfully solves the problem for any data set, we understand that it is out of the hands of a novice student. An alternative would be to choose a sufficiently reliable data set or test cases, i.e., one that contains a reasonably comprehensive set of test cases, including edge cases. Again, the task of choosing sufficiently reliable test cases is time-consuming and may be beyond the capabilities of the learner.

In order to determine the validity of a program, there are so-called programming judges. However, they have an important limitation. This limitation consists in the fact that these judges can only issue verdicts to solutions programmed in the languages for which they are prepared. This means that for each language, and even for different versions of it, it is necessary to develop the functionality to be able to correct the solutions programmed in that language. From the need to be able to learn to program in any language using a judge, the idea of ScholarJudge was born.

2.2. Objectives

The main goal of this project is to develop a language-agnostic programming judge so that students can reinforce and extend their programming knowledge in a practical way.

The specific goals of ScholarJudge are:

- Creation of a user-friendly desktop application, so that the student can download problems from a server containing a catalog of problems, view their statistics and check their solutions regardless of the programming language used.
- Creation of a web application with a design adaptable to both computers and tablet devices, so that teachers can manage contests, problems and users. At the same time, students will be able to visualize in a more attractive way the information related to contests, problems and their own statistics.
- Provide an authentication system so that each student can keep track of their submissions and statistics.
- Establish a role system that allows teachers and administrators to access content management features that students should not be able to access.

2.3. Workplan

The development of this project covers from September 2021 to June 2022. For this project scheduling, an agile methodology has been followed, adapting the way of working to the conditions of the project as required at all times. Since all members of this project have been working partial or full time during the project course, the chosen methodology has given more flexibility and a more efficient time management.

Sprints have been carried out during the project development course with durations ranging from 1 week to 3 weeks. The sprints end with a meeting in which all project participants are present. In these meetings the students show the work done during the sprint, the tutors review the assignments and give feedback. Immediately afterwards, the project current situation is analyzed and the following assignments to do are determined just like the new technologies to investigate, if so required.

In each *sprint* the following tasks have been carried out jointly:

- Analysis: selection of the tasks to carry out in the new *sprint*.
- Research: selection and emphasis in new technologies, either new or previously used, to resolve the tasks of the new *sprint*.
- Implementation: development of the desktop application and the web application at the same time.
- Testing: mostly checking tests and, in some cases, unit tests.

The following figures show two Gantt charts corresponding to the tasks carried out in the first and second quarter respectively.

2.4. Memory structure

The rest of the memory is structured as follows:

- Chapter 3 will talk about programming judges, their importance in learning and their limitations to support new languages. The main technologies used in the project are also presented.

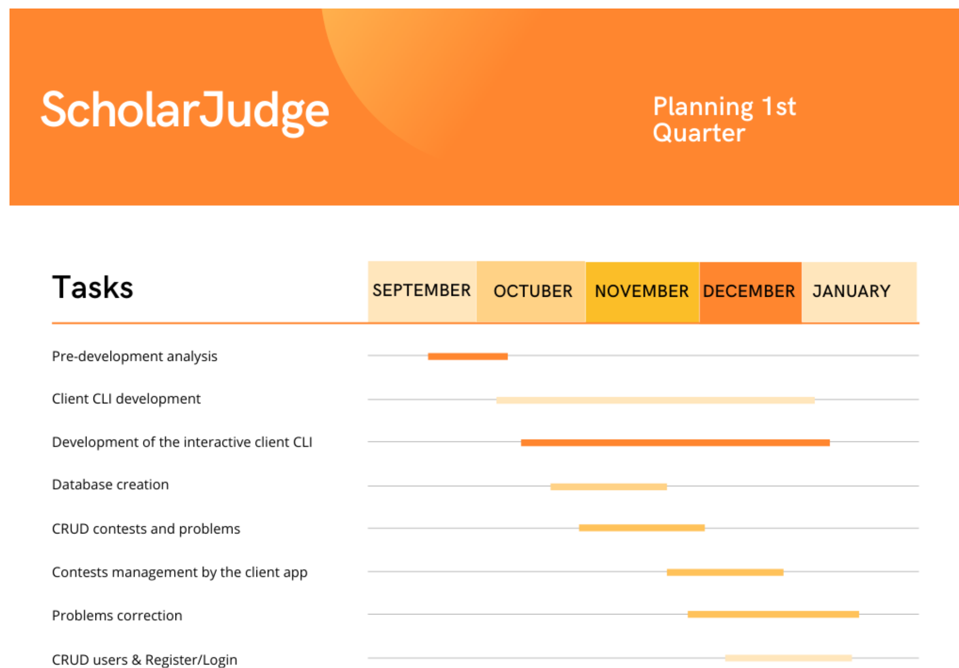


Figure 2.1: Gantt Chart: first quarter planning

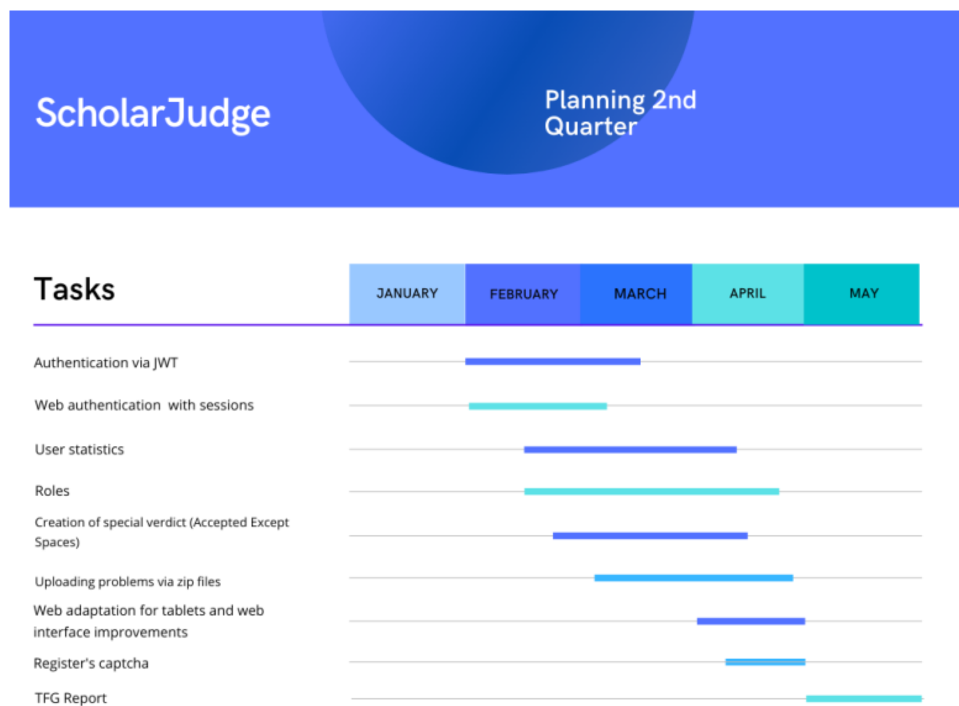


Figure 2.2: Gantt Chart: second quarter planning

- Chapter 4 will study the system organization, what components is it composed of and how are they interconnected.
- Chapter 5 details the functions of the server and its internal structure.
- Chapter 6 details the functions of the client application, its internal structure and how the user can interact with it.
- In chapter 7 the web application will be studied in detail.
- In chapter 8 the conclusions and future work for this project will be discussed.

Capítulo 3

Preliminares

En este capítulo se presentarán los jueces de programación y su importancia en el aprendizaje y las tecnologías utilizadas en el desarrollo del proyecto.

3.1. Los jueces de programación

Un juez de programación es un sistema informático que, tras proponerle un problema al usuario, evalúa si la solución aportada es correcta.

Para los estudiantes de informática, los jueces de programación resultan ser una herramienta muy útil a la hora poner a prueba sus conocimientos y progresos. La idea de que un sistema informático pueda proponer un problema y decidir si la solución propuesta es o no correcta resulta atractiva y motiva a probar y practicar. La práctica es una parte fundamental en el aprendizaje de la programación, y disponer de un sistema al que los alumnos puedan recurrir en busca de nuevos problemas y retos como complemento a los aportados por su profesores es beneficioso para ellos.

Una de las limitaciones que tiene la gran mayoría de los jueces es que están limitados por los lenguajes de programación que están programados para entender. De modo que, si un estudiante está aprendiendo un lenguaje distinto a los que el juez tiene en su sistema, la herramienta deja de ser útil para él.

Otro problema es que un juez suele recibir el código que resuelve el problema propuesto y lo compila en su servidor en ese momento. Si el código es muy ineficiente o no termina, aparte del tiempo que tarda en devolverle el resultado al usuario, se malgastarían los recursos de memoria o de velocidad de procesamiento del servidor. Para un solo envío esto no debería suponer mayor problema, pero para muchos envíos simultáneos, se podría llegar a saturar el servidor.

Uno de los jueces de programación ya existentes que pone de manifiesto estas limitaciones es *¡Acepta el reto!* Este juez esta preparado sólo para C, C++ y Java, por lo que solo se puede practicar con estos tres lenguajes utilizando este juez. Sus problemas están agrupados por volúmenes, con unos 100 problemas cada uno; o por categorías (figura 3.1) El usuario sube el código de su solución y el sistema hace la compilación y obtiene el resultado. En los casos en los que no se llega a una solución porque se produce algún bucle infinito, el usuario tarda bastante en obtener una respuesta. También dispone de una página de estadísticas, tanto general de todos los usuarios, que solo muestra los últimos envíos; como en particular del usuario conectado.

Como se ha mencionado anteriormente en el capítulo 1, el objetivo de este TFG es

¡Acepta el reto! Problemas Estadísticas Documentación marE92 Buscar

Estás en: Inicio / Problemas / Por categorías / Programación

Por volúmenes
Por categorías

Categoría Programación

Categorías relacionadas con conceptos de programación.

Subcategorías

Nombre	Descripción	Nº de problemas	AC/Envíos
Construcciones de programación	Problemas categorizados según las construcciones de programación más relevantes usadas en las soluciones.	276	155544/424992
Estructuras de datos	Problemas que necesitan almacenar en memoria colecciones de datos estructurados para poder dar la solución.	202	79270/231759
Algoritmia	Problemas categorizados según la familia a la que pertenece el algoritmo que se necesita para resolverlos.	202	65434/202145
Matemáticas	Problemas en los que se necesita programar rutinas con cálculos matemáticos.	77	41752/125649
Grafos	Problemas donde o bien se deben utilizar grafos para almacenar los datos leídos, o bien se necesitan algoritmos sobre grafos.	31	3997/11845
Geometría	Problemas relacionados directamente con geometría o requieren algoritmos de geometría computacional.	7	6223/16533

¿Qué significan estos números?

(c) Acepta el reto, 2013 - 2022

Figura 3.1: Subcategorías de problemas de ¡Acepta el reto!

proponer un solución al problema de limitación por lenguaje y al problema de tiempo de ejecución.

Para ello, se ha creado un nuevo juez de programación, llamado *ScholarJudge*, que, tras proporcionar al usuario el problema propuesto, recibe solo la salida producida por la solución obtenida por el estudiante y la compara con una solución esperada. La compilación o interpretación de la solución del usuario depende en exclusiva de la máquina de este último, sin consumir los recursos del servidor y pudiendo utilizar para obtener dicha solución el lenguaje que desee.

3.2. Tecnologías utilizadas

Para el desarrollo del proyecto se han utilizado las siguientes tecnologías:

3.2.1. Node.js

Node.js es un entorno de ejecución para JavaScript construido con V8, motor de JavaScript de Chrome[1]. Dispone de una gran cantidad de librerías y su facilidad a la hora de mantener el código de cara a futuro es lo que hizo que se convirtiera en la tecnología principal del desarrollo del proyecto.

Algunas de las características principales de Node.js:

- Es multiplataforma.
- Permite la ejecución de JavaScript en el lado del servidor.
- Su ejecución es asíncrona y basada en eventos.

[1] `https://nodejs.org/es/`

3.2.2. Express

Express es un framework de Node.js que facilita el *renderizado* de las vistas y la gestión de peticiones HTTP. Express facilita la descripción de los endpoints REST en *Routers*. En ellos se indica la URI y el método HTTP (GET, POST, PATCH, DELETE...) de cada *endpoint* en función de cómo se responda a las peticiones de los clientes. Estos *endpoints* llaman a una función *middleware*, que ejecuta las acciones solicitadas por el cliente y devuelve un objeto con la respuesta a la petición.
Ejemplo de uno de los endpoints de *ScholarJudge*:

```
1  const path = require("path");
2  const express = require("express");
3  const userRouter = express.Router();
4  //Middlewares & config
5  const accessControl = require(path.join("..", "middlewares", "access"))
   ;
6
7  const UserController = require(path.join(
8      "..",
9      "controllers",
10     "UserController"
11 ));
12
13 userRouter.post("/", UserController.createUser);
```

En este caso, "/" sería la URI, `UserController.createUser` sería la función *middleware* y `post` el método HTTP que describe el comportamiento del *endpoint*.
Para el desarrollo del TFG se utilizó Express tanto para las peticiones REST de los clientes web y de escritorio, como para gestionar las peticiones HTTP del cliente web para la obtención de las vistas (Krause, 2017).

3.2.3. API REST

API de REST es una interfaz de programación que permite la comunicación entre servidor y cliente de escritorio a través de peticiones HTTP. Esta tecnología permite la solicitud de datos o de ejecución de operaciones por parte del cliente en el lado del servidor.

Cada una de las peticiones REST es independiente entre sí, ahorrando recordar estados previos para poder ejecutar las distintas operaciones. Se eligió esta tecnología frente a otros protocolos, como SOAP, por su sencillez y ligereza.

Más específicamente, en este desarrollo se han utilizado los verbos HTTP `GET`, para recuperar información contenida en la base de datos; `POST`, para introducir nueva información en la base de datos o en el sistema de ficheros; `PATCH`, para modificar información ya existente; y `DELETE`, para eliminar información de la base de datos.

Por los objetivos marcados para este TFG, se han identificado numerosos recursos que se agrupan en función de si afectan a usuarios, concursos, problemas e intentos de envío. Principalmente se enfocan a la introducción y modificación de datos en la base de datos o en la recuperación de dichos datos para mostrarlos. Esto se ampliará en el capítulo 4.

3.2.4. MongoDB

MongoDB es un sistema de base de datos no relacional, de código abierto y con licencia Apache GNU AGPL v3.0. Se eligió MongoDB para este TFG por su flexibilidad y rapidez

de acceso a los datos.

3.2.5. JWT

JSON Web Token (JWT) es un estándar basado en JSON que permite la autenticación en el sistema de forma segura creando un token con las credenciales necesarias.

En el proyecto se utiliza para la identificación de un usuario en el sistema desde el cliente de escritorio. Su objetivo es, por un lado, no hacer que el usuario tenga que introducir sus credenciales en cada operación que hace y, por otro lado, que esas credenciales viajen por las peticiones REST lo mínimo posible, haciendo más segura la identificación del usuario.

Inicialmente, el usuario se identifica en el cliente de escritorio, introduciendo su email y su contraseña. Se envían al servidor y, si son correctas, el servidor devuelve un token con la información codificada y una fecha de caducidad de ese token, que se almacena en un archivo del sistema de ficheros de la máquina del usuario. Mientras el token esté activo, el usuario no tiene que volver a identificarse. En nuestra aplicación hemos utilizado JSONWebToken, que es una librería de manejo de JWT para Node.js.

3.2.6. HTML y CSS

HTML es un lenguaje de marcado que permite la definición de la estructura de las vistas del cliente web.

CSS (Cascade Style Sheets) es el lenguaje que permite definir la presentación de los diferentes elementos creados en el HTML de las vistas.

En *ScholarJudge* se utilizan estos lenguajes para la creación de las vistas del cliente web y su aspecto visual.

3.2.7. JavaScript y jQuery

JavaScript es un lenguaje de programación del lado del cliente que permite incluir interactividad en las páginas web.

jQuery es una librería de JavaScript que simplifica el manejo del DOM de las vistas. Es multiplataforma y de código abierto.

Ambas tecnologías se utilizan en todo el proyecto, en el lado de servidor mediante Node.js para la definición del *middleware* que ejecutan las operaciones solicitadas por el cliente de escritorio. En el lado del cliente web, por un lado, se utiliza para definir algunos comportamientos de componentes de las vistas y, por otro lado, para validar información introducida en los distintos formularios y, finalmente, llamar a los servicios REST necesarios para registrar la información validada.

3.2.8. Bootstrap

Bootstrap es una librería que facilita el diseño de páginas y aplicaciones web. Dispone de plantillas con componentes como botones, formularios, tablas, menús... basados en CSS y HTML. Es multiplataforma y de código abierto.

En este TFG se utiliza para hacer la aplicación web *responsive*, con un menú colapsable cuando la pantalla disminuye de un determinado tamaño, y para la gestión de la tabla de usuarios, que permite ordenar la información mostrada en la tabla, paginarla y tener un buscador predictivo.

3.2.9. Git y GitHub

Git es una herramienta de control de versiones, que facilita el registro de todas las modificaciones que se hacen a los ficheros que componen un proyecto y quién realiza dichas modificaciones. También ayuda a evitar conflictos entre el código que aportan diferentes programadores.

GitHub es un repositorio que utiliza Git como sistema de control de versiones. Es el repositorio y sistema de control de versiones utilizados para este TFG.

3.2.10. LaTeX

LaTeX es una herramienta de código abierto para la creación de documentos. Es muy utilizada en el sector científico por su calidad tipográfica, su facilidad de uso y la capacidad de definir funcionalidades propias. Es el sistema de composición de textos utilizado para editar esta memoria.

3.2.11. Heroku

Heroku es una plataforma en la nube que permite alojar aplicaciones, así como datos, que proporciona a los desarrolladores una gran cantidad de herramientas para el crecimiento de proyectos. Es la plataforma seleccionada para alojar la aplicación desarrollada en este TFG.

3.2.12. MTCaptcha

Los CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart) son pruebas utilizadas como medidas de seguridad para evitar el acceso fraudulento de *bots* a las páginas web.

Hay diferentes tipos de captcha, pero todos deben cumplir las mismas características: que sean fáciles de resolver por humanos, que sean fáciles de generar y evaluar y que no sean fáciles de resolver por ordenadores.

MTCaptcha es el servicio captcha elegido para evitar el registro de usuarios fantasma o falsos provocados por *bots*.

Capítulo 4

Organización del sistema

Para la implementación del juez de programación *ScholarJudge* se ha recurrido a una arquitectura cliente-servidor. En este capítulo se describirán las entidades involucradas de *ScholarJudge* y su arquitectura.

4.1. Definición de *ScholarJudge*

La herramienta *ScholarJudge* está pensada para su uso en el ámbito académico. Esto se refleja en la organización de los problemas que puede proponer el sistema y en los tipos de usuarios que puede tener.

4.1.1. Problemas

En el contexto de este TFG, un *Problema* es cada uno de los ejercicios que un estudiante tiene disponibles para resolver. Estos ejercicios cuentan con un enunciado, un par de ficheros con la entrada y salida esperada de los casos de prueba visibles en el enunciado y una cantidad variable de pares de ficheros correspondientes a los casos de prueba almacenados en el servidor.

ScholarJudge tiene depositados en su servidor un número indefinido de problemas que los estudiantes pueden resolver. La complejidad de los problemas y la forma de resolverlos es variada. Para dar un poco de orden a los problemas se dispone de unos contenedores llamados *Concursos*, que los agrupan por temática o, en el contexto académico, por asignatura. Cada uno de los problemas puede estar en más de un concurso.

4.1.2. Usuarios

En *ScholarJudge* puede haber tres tipos distintos de usuarios:

- *Usuario/User*: Solo puede descargar problemas y enviar potenciales soluciones de los mismos.
- *Profesor/Teacher*: Tiene la capacidad de crear y editar concursos y problemas. También puede descargar y resolver problemas.
- *Administrador/Admin*: A mayores de crear y editar concursos y problemas, descargarlos y resolverlos, gestiona los usuarios. Puede convertir usuarios en profesores y

profesores en usuarios, cambiar la contraseña de un usuario que haya olvidado la suya para no obligarlo a registrarse de nuevo, además de eliminar usuarios.

Todos los usuarios pueden registrarse por sí mismos y modificar la información de su perfil y su contraseña.

4.2. Arquitectura de *ScholarJudge*

Para este proyecto se ha recurrido a una arquitectura cliente-servidor, conectados mediante servicios REST para facilitar la comunicación entre las partes. En la figura 4.1 se aprecia la arquitectura del sistema.

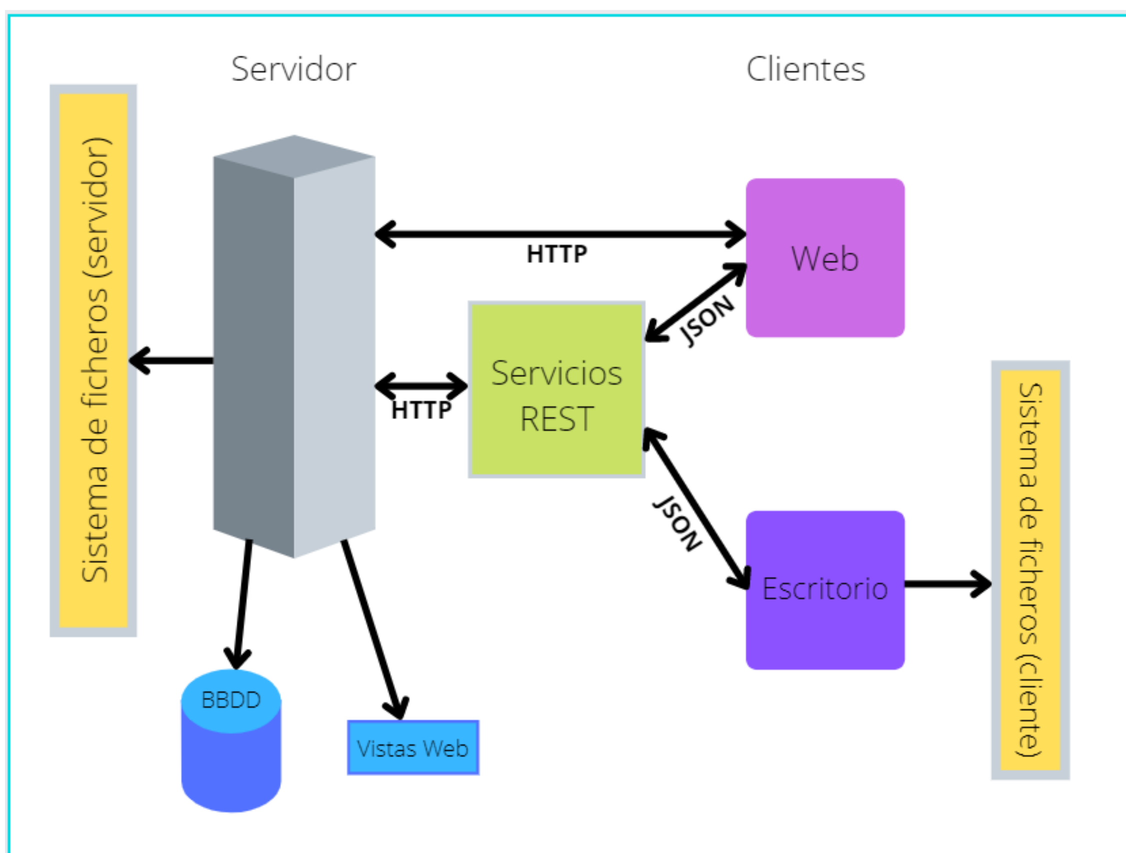


Figura 4.1: Arquitectura de ScholarJudge

El usuario podrá desde su aplicación cliente de escritorio solicitar datos relativos a los diferentes concursos y problemas o enviar sus soluciones a los problemas invocando a los servicios REST correspondientes. Estos servicios se comunicarán con el servidor, que ejecutará las operaciones solicitadas y devolverá al cliente, de nuevo a través del protocolo HTTP, la información requerida.
Desde la aplicación web se podrá dar de alta y modificar usuarios; crear y editar concursos y problemas; y revisar las estadísticas de aciertos y fallos de un forma más visual. Estas operaciones también se hacen a través de servicios REST.
A continuación se describirá brevemente cada una de las partes.

4.2.1. Cliente de escritorio

El cliente de *ScholarJudge* es una aplicación de escritorio controlada mediante línea de comandos. Mediante esta aplicación, el usuario podrá inscribirse en concursos, descargar problemas, enviar sus intentos de solución a dichos problemas y comprobar cuáles tiene ya resueltos.
Las tareas del cliente de escritorio son:

- Solicitar al servidor los datos de los problemas.
- Almacenar en el sistema de ficheros del equipo del usuario los datos del problema recibidos.
- Enviar al servidor la salida generada por la solución del usuario para los casos de prueba del problema, y recibir el veredicto por parte del servidor.

La funcionalidad del cliente de escritorio se tratará más en profundidad en el capítulo 6.

4.2.2. Cliente Web

El cliente web de *ScholarJudge* es una aplicación web. Mediante esta aplicación, todos los usuarios podrán registrarse, modificar su datos y revisar sus estadísticas de forma detallada. Los profesores y administradores podrán crear concursos y problemas y, en el caso de los administradores, gestionar los usuarios.
El cliente web realiza las siguientes tareas:

- Registro y gestión de usuarios.
- Creación y gestión de concursos y problemas.
- Mostrar estadísticas de participación de los usuarios.

La funcionalidad del cliente web se tratará más en profundidad en el capítulo 7.

4.2.3. Servidor

El servidor es el núcleo del sistema. En *ScholarJudge* el servidor aloja la base de datos, los problemas, las vistas de la aplicación web y compara la salida esperada del problema con lo aportado por el usuario.
Las tareas del servidor incluyen:

- Conectar con la base de datos y alojar los problemas.
- Alojar las vistas de la aplicación web.
- Servir al cliente la información y los datos de los problemas.
- Servir al navegador las vistas de la aplicación web.
- Comparar la salida esperada de un problema con la generada por la solución del usuario.
- Gestionar usuarios.

La funcionalidad del servidor se tratará más en profundidad en el capítulo 5.

4.2.4. Conexión cliente-servidor

Para hacer posible la conexión entre el cliente y el servidor se ha desarrollado una serie de servicios REST (Nandaa, 2018). Para facilitar su comprensión, se dividirán en función de las entidades del sistema involucrados en su operativa:
A lo largo de la descripción de los siguientes endpoints se va a hacer referencia a las entidades User, Contest y Problem.

El modelo de User contiene la siguiente información:

- `id`: String. Identificador único para cada usuario. Se genera automáticamente.
- `name`: String. Nombre de usuario.
- `email`: String. Email que el usuario utiliza para identificarse en el sistema.
- `password`: String. Contraseña del usuario. Se almacena codificada utilizando la librería `bycript`. Se sigue el método en dos pasos: primero se genera una sal y luego se codifica la contraseña con la sal previamente generada.
- `role`: String. Rol del usuario en el sistema. Puede ser `user`, `teacher` o `admin`.

El modelo de Contest contiene la siguiente información:

- `id`: String. Identificador único para cada concurso. Se genera automáticamente.
- `name`: String. Nombre del concurso.
- `desc`: String. Descripción del concurso.
- `categories`: [String]. Lista de categorías a las que pertenece un concurso.
- `problems`: [String]. Lista con los identificadores únicos de los problemas que compone un concurso.

El modelo de Problem contiene la siguiente información:

- `id`: String. Identificador único para cada problema. Se genera automáticamente.
- `title`: String. Nombre del problema.
- `numFiles`: Number. Número de casos de prueba que tiene el problema.
- `categories`: [String]. Lista de categorías a las que pertenece un problema.
- `checkSums`: [String]. Lista con las soluciones esperadas codificadas.
- `noSpaceCheckSums`: [String]. Lista con las soluciones esperadas codificadas tras eliminar los espacios en blanco.

El modelo de Attempt contiene la siguiente información:

- `user_id`: String. Identificador único del usuario que hizo el envío.
- `problem_id`: String. Identificador único del problema que usuario intentó resolver.
- `correct`: Boolean. Indicador que informa si se han pasado todos los casos de prueba del problema `problem_id`.

- `status`: [String]. Lista con los posibles veredictos de un problema: `AC` (solución correcta), `AC-ES` (solución correcta salvo los espacios), `WA` (solución incorrecta).
- `veredict`: String. Veredicto de la solución a un problema.
- `date`: Date. Fecha en la que el usuario envió su intento de solución del problema.

4.2.4.1. Endpoints

Usuarios

- URL Endpoint: `/user` Método: `POST`
  Input:
  - `name`: String
  - `email`: String
  - `password`: String

  Output:
  - `error`: String
  - `ok`: Boolean
  - `users`: User
  - `Código de estado`: Integer

  Rol que interviene: `Usuario/User`
  Descripción: Crea e incluye en la base de datos un nuevo usuario desde la aplicación web. El usuario se crea con el rol `user` por defecto. Si el proceso acaba correctamente, devuelve el usuario creado al cliente, la salida `ok` con el valor `"true"` y el valor `"false"` en `error`. Si se produce algún error, devuelve el código de error, `ok` con el valor `"false"` y el mensaje correspondiente al error que se haya producido en `error`.

- URL Endpoint: `/user`
  Método: `PATCH`
  Input:
  - `name`: String
  - `newMail`: String
  - `newName`: String
  - `newRole`: String
  - `newPass`: String
  - `oldPass1`: String
  - `action`: String

  Output:
  - `Código de estado`: Integer
  - `error`: String
  - `ok`: Boolean

Rol que interviene: `Usuario/User, Profesor/Teacher y Administrador/Admin`
Descripción: Actualiza la información del usuario en la base de datos. En función del valor de `action`, se actualizan unos campos u otros:

- Si `action` es `"newRole"`, se actualiza el rol del usuario con el valor de `newRole`.
- Si `action` es `"newNameMail"`, se actualizan el valor del email y del nombre de usuario con el valor contenido en `newMail` y `newMail`.
- Si `action` es `"newPass"`, se actualiza la contraseña del usuario con el valor de `newPass`, comprobando previamente si la contraseña en `oldPass1` es la que ya estaba almacenada. En este caso, la operación la inicia el usuario que está identificado en el sistema.
- Si `action` es `"newPassAdmin"`, se actualiza la contraseña del usuario con nombre `name` por la introducida en `newPass`. En este caso, la operación la inicia el administrador del sistema.

Si el proceso acaba correctamente, devuelve 200 en el código de estado. Si se produce algún error, devuelve el código y el mensaje de error.

- URL Endpoint: `/user`
  Método: `DELETE`
  Input:
  - `name`: String

  Output:
  - `Código de estado`: Integer
  - `error`: String
  - `ok`: Boolean

  Rol que interviene: `Administrador/Admin`
  Descripción: Elimina toda la información del usuario de nombre `name` de la base de datos. Si el proceso acaba correctamente, devuelve 200 en el código de estado. Si se produce algún error, devuelve el código y el mensaje de error.

- URL Endpoint: `/user/login`
  Método: `POST`
  Input:
  - `email`: String
  - `password`: String

  Output:
  - `Código de estado`: Integer
  - `err`: String

  Rol que interviene: `Usuario/User, Profesor/Teacher y Administrador/Admin`
  Descripción: Valida las credenciales del usuario, `email` y `password`, introducidas en el formulario de autenticación del cliente web. Si es todo correcto, devuelve 200 como código de estado. En caso contrario, devuelve un mensaje de error con su código correspondiente.

- URL Endpoint: `/user/<email>/token`
  Método: `POST`
  Input:
  - `email`: String
  - `password`: String

  Output:
  - `Código de estado`: Integer
  - `err`: String
  - `token`: String

  Rol que interviene: `Usuario/User`
  Descripción: Valida las credenciales del usuario, `email` y `password`, y devuelve un token JWT de autenticación al cliente de escritorio. Esto se explicará con mayor detalle en el capítulo 6. En caso de que las credenciales no sean correctas, devuelve un mensaje de error con su correspondiente código de estado.

- URL Endpoint: `/user/logout`
  Método: `POST`
  Output:
  - `Código de estado`: Integer

  Rol que interviene: `Usuario/User, Profesor/Teacher y Administrador/Admin`
  Descripción: Elimina el usuario almacenado en sesión. Si la operación finaliza correctamente, se devuelve 200 en el código de estado. En caso contrario, se devuelve el código de error correspondiente.

Concursos

- URL Endpoint: `/contest`
  Método: `POST`
  Input:
  - `title`: String
  - `description`: String

  Output:
  - `error`: String
  - `Código de estado`: Integer

  Rol que interviene: `Profesor/Teacher y Administrador/Admin`
  Descripción: Crea e introduce en la base de datos un nuevo concurso a partir del título `title` y la descripción `description` aportados por el usuario a partir del formulario web. En caso de error, se devuelve el código y el mensaje de error.

- URL Endpoint: `/contest/<id>`
  Método: `PATCH`
  Input:

- `id`: String
- `name`: String
- `desc`: String
- `problems`: [String]
- `categories`: [String]

Output:

- `error`: String
- `Código de estado`: Integer

Rol que interviene: `Profesor/Teacher y Administrador/Admin`
Descripción: Actualiza en la base de datos la información del concurso con identificador `id`. A parte del título y la descripción del concurso, también se puede modificar la lista de problemas que componen el concurso (`problems`) y las categorías a las que pertenece (`categories`). En caso de error, se devuelve el código y el mensaje de error.

- URL Endpoint: `/contest/<id>`
Método: `DELETE`
Input:

  - `id`: String

  Output:

  - `error`: String
  - `Código de estado`: Integer

  Rol que interviene: `Profesor/Teacher y Administrador/Admin`
  Descripción: Elimina de la base de datos la información del concurso con identificador `id`. Si la operación termina correctament, se devuelve 200 como código de estado. En caso de error, se devuelve el código y el mensaje de error correspondientes.

- URL Endpoint: `/contest/`
Método: `GET`
Output:

  - `error`: String
  - `contests`: [Contest]
  - `Código de estado`: Integer

  Rol que interviene: `Usuario/User`
  Descripción: Devuelve la lista entera de concursos registrados en la base de datos. En caso de error, se devuelve el código y el mensaje de error.

- URL Endpoint: `/contest/<id>/problems`
Método: `GET`
Input:

  - `id`: String

Output:

- `error`: String
- `data`: [Problem]
- `Código de estado`: Integer

Rol que interviene: `Usuario/User`
Descripción: Devuelve la lista completa de problemas de un concurso con identificador `id` junto con la indicación de si el usuario que la solicita ha resuelto ya cada uno de los problemas. En caso de error, se devuelve el código y el mensaje de error.

Problemas

- URL Endpoint: `/problem/`
  Método: `POST`
  Input:

  - `files`: FILE

  Output:

  - `msg`: String
  - `Código de estado`: Integer

  Rol que interviene: `Profesor/Teacher y Administrado/Admin`
  Descripción: Crea un nuevo problema en la base de datos y en el sistema de ficheros del servidor a partir de un fichero comprimido con extensión .zip, que contiene el fichero con los metadatos del problema; y una serie de ficheros como el enunciado y los casos de prueba. En caso de fallar, se devuelve el código y el mensaje de error.

- URL Endpoint: `/problem/<id>`
  Método: `PATCH`
  Input:

  - `id`: String
  - `tags`: String
  - `title`: String
  - `files`: [File]
  - `deletes`: [Boolean]

  Output:

  - `msg`: String
  - `Código de estado`: Integer

  Rol que interviene: `Profesor/Teacher y Administrado/Admin`
  Descripción: Actualiza el problema con identificador `id` en la base de datos y en sistema de ficheros del servidor. En caso de fallar, se devuelve el código y el mensaje de error.

- URL Endpoint: `/problem/<id>`
  Método: `DELETE`
  Input:

- `id`: String

Output:

- `msg`: String
- `Código de estado`: Integer

Rol que interviene: `Profesor/Teacher y Administrado/Admin`
Descripción: Elimina el problema con identificador `id` de la base de datos y sus referencias en los concursos. Si la operación se completa correctamente, devuelve 200 como código de estado. En caso de fallar, se devuelve el código y el mensaje de error.

- URL Endpoint: `/problem/<id>/statement`
  Método: `GET`
  Input:

  - `id`: String

  Output:

  - `error`: String
  - `data`: String
  - `Código de estado`: Integer

  Rol que interviene: `Usuario/User`
  Descripción: Recupera el enunciado del problema con identificador `id` del sistema de archivos del servidor, lo lee y convierte a formato texto y se lo envía al cliente. En caso de fallar, se devuelve el código y el mensaje de error.

- URL Endpoint: `/problem/<id>/input/<num>`
  Método: `GET`
  Input:

  - `id`: String
  - `num`: String

  Output:

  - `data`: String
  - `error`: String
  - `Código de estado`: Integer

  Rol que interviene: `Usuario/User`
  Descripción: Recupera los datos de entrada del caso de prueba `num` del problema con identificador `id` del sistema de archivos del servidor, lo lee y convierte a formato texto y se lo envía al cliente. En caso de fallar, se devuelve el código y el mensaje de error.

- URL Endpoint: `/problem/<id>/output/<num>`
  Método: `GET`
  Input:

  - `id`: String

- `num`: String

Output:

- `data`: String
- `error`: String
- `Código de estado`: Integer

Rol que interviene: `Usuario/User`
Descripción: Recupera los datos de salida del caso de prueba `num` del problema con identificador `id` del sistema de archivos del servidor, lo lee y convierte a formato texto y se lo envía al cliente. En caso de fallar, se devuelve el código y el mensaje de error.

- URL Endpoint: `/problem/<id>/metadata`
  Método: `GET`
  Input:

  - `id`: String

  Output:

  - `id`: String
  - `name`: String
  - `numFiles`: String
  - `error`: String
  - `Código de estado`: Integer

  Rol que interviene: `Usuario/User`
  Descripción: Recupera la información almacenada del problema con identificador `id` de la base de datos y se la envía al cliente. En caso de fallar, se devuelve el código y el mensaje de error.

- URL Endpoint: `/problem/<id>/stats`
  Método: `GET`
  Input:

  - `id`: String

  Output:

  - `correct`: Integer
  - `fail`: Integer
  - `total`: Integer
  - `error`: String
  - `Código de estado`: Integer

  Rol que interviene: `Usuario/User`
  Descripción: Recupera el número de intentos de un usuario para resolver el problema con identificador `id` de la base de datos, incluido el número de aciertos (`correct`) y de fallos (`fail`) y se los envía al cliente. En caso de fallar, se devuelve el código y el mensaje de error.

- URL Endpoint: `/problem/next`
  Método: `GET`
  Input:
  - `problems`: [String]
  - `contest`: String

  Output:
  - `id`: Integer
  - `error`: String
  - `Código de estado`: Integer

  Rol que interviene: `Usuario/User`
  Descripción: Para el usuario actual, recupera el identificador del problema siguiente a resolver de un determinado concurso con identificador `contest`. En caso de fallar, se devuelve el código y el mensaje de error.

Intentos de solución

- URL Endpoint: `/attempt/`
  Método: `POST`
  Input:
  - `problemId`: String
  - `status`: [String]
  - `veredict`: String
  - `correct`: Boolean

  Output:
  - `error`: String
  - `attempt`: Attempt
  - `Código de estado`: Integer

  Rol que interviene: `Usuario/User`
  Descripción: Crea e incluye en la base de datos un nuevo intento del usuario con la fecha del intento. Si funciona correctamente, se devuelve el código del estado y el intento al cliente. En caso contrario, se devuelve el código y el mensaje de error.

- URL Endpoint: `/attempt/check/<id>/<num>/<solution>`
  Método: `GET`
  Input:
  - `id`: String
  - `num`: String
  - `solution`: String

  Output:
  - `id`: String

- `num`: String
- `status`: String
- `error`: String
- `Código de estado`: Integer

Rol que interviene: `Usuario/User`
Descripción: Comprueba la solución `solution` aportada para el caso de prueba `num` del problema con identificador `id`. Si la operación ocurre con éxito, se devuelve el identificador del problema, con el número de caso de prueba y el resultado. En caso de fallar, se devuelve el código y el mensaje de error.

Capítulo 5

Servidor

El servidor de ScholarJudge ha sido desarrollado usando Node.js como principal tecnología. Su propósito puede dividirse en dos. Por un lado, servir el contenido de las páginas web, así como las funcionalidades que estas ofrecen a los usuarios que naveguen por la aplicación web. Por otro lado, proporcionar al usuario de la aplicación de escritorio los servicios necesarios para poder descargar concursos, problemas, corregir sus soluciones y ver sus estadísticas.

Para poder servir al usuario final de dichas funcionalidades, el servidor será el único encargado de comunicarse con la base de datos, la cual almacenará toda la información relativa al modelo del dominio, a excepción de los enunciados y casos de prueba de los problemas. Estos se alojarán directamente en el sistema de ficheros del propio servidor debido a motivos de eficiencia.

5.1. Base de datos

Se ha optado por escoger una base de datos NoSQL ya que son bastante versátiles a la hora de agregar nueva información o hacer cambios en la propia estructura de los datos. Se prevén posibles cambios en la estructura de los datos posteriores a la entrega de este TFG, en función de las necesidades de los profesores y/o asignaturas, lo cual refuerza la necesidad de que nuestra base de datos sea escalable.

Otra característica a mencionar de las bases de datos NoSQL es que pueden realizar y recibir peticiones en formato JSON, lo cual hace muy fácil el tratamiento de los datos.

5.1.1. Mongo Atlas

La base de datos que se ha usado ha sido MongoDB, en concreto, su versión en la nube llamada Mongo Atlas. Este es un servicio de bases de datos totalmente gestionado. MongoAtlas se encarga de las tareas de administración laboriosas y costosas para permitirnos como usuario acceder a los recursos de la base de datos que necesitemos.

MongoAtlas consta de varios planes de contratación que ofrecen diferentes funcionalidades. Para este TFG se ha escogido el Plan básico gratuito, el cual proporciona las siguientes características:

- 512MB a 5GB de almacenamiento
- RAM compartida

- Clústers con funcionalidad limitada
- Acceso al soporte en la aplicación de lunes a viernes
- Gráficas con información relativa al nº de lecturas, escrituras, etc.

En la figura 5.1 se muestra una imagen de un dashboard con información relativa al número de lecturas/escrituras en la BD.

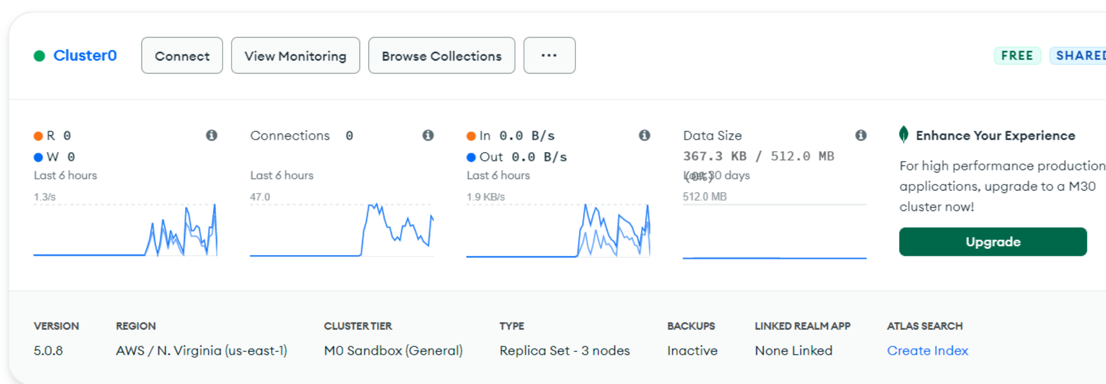


Figura 5.1: MongoAtlas: gráfico

5.1.2. Mongoose

Para acceder a la base de datos desde nuestro servidor, se ha usado Mongoose, un marco de JavaScript que se usa comúnmente en aplicaciones desarrolladas en Node.js que usan MongoDB como base de datos. Mongoose es un ODM *(Object Document Mapper)*. Esto significa que Mongoose nos permite definir objetos con un esquema(Mongoose, 2011a) fuertemente tipado que se asigna a un documento MongoDB. Mongoose actualmente contiene ocho tipos para definir los datos. Estos son:

- String (Cadena)
- Number (Número)
- Buffer
- Boolean (Booleano)
- Mixed (Mixto)
- ObjectId
- Array (Matriz)

Estos tipos encajan con nuestras necesidades de almacenamiento. El tipo ObjectId nos permite almacenar las claves primarias generadas por MongoDB. Las bases de datos NoSQL, a diferencia de las SQL, nos permiten almacenar arrays, lo cual es conveniente, entre otros casos, para almacenar en un concurso un listado de sus problemas. El resto de tipos nos permiten guardar tipos primitivos como números, cadenas de texto, etc. A continuación se muestra un fragmento de código usado para definir el esquema de un concurso.

```
const mongoose = require("mongoose");
const { Schema } = mongoose;

const contestSchema = new Schema({
    name: String,
    desc: String,
    categories: [{ type: mongoose.Types.ObjectId, ref: "categories" }],
    problems: [mongoose.ObjectId]
});

const Contest = mongoose.model("contests", contestSchema);
```

5.2. Comunicación con el Cliente web

En esta sección se explicará en profundidad como se comunica el Servidor con el Cliente web, entrando en detalle en las tecnologías y herramientas utilizadas para dicho fin.

5.2.1. Express

Express (Express, 2010a) es un framework de Node.js, usado en nuestro TFG, que proporciona numerosas funciones a la hora de construir nuestra aplicación web. Los principales mecanismos que nos proporciona son:

- Creación de gestores de peticiones HTTP para diferentes URLs
- Motores de renderización de vistas para generar respuestas mediante la introducción de datos en plantillas
- Añadir procesamiento de peticiones, mediante los llamados *middlewares*.
- Algunos ajustes para aplicaciones web como el puerto a usar, localización de plantillas, etc.

Cabe destacar que se ha hecho uso de `express.router()`(Express, 2010b) para crear subaplicaciones, teniendo así una división lógica de los controladores de ruta que hace mas mantenible y entendible el código de los manejadores. A continuación proporcionamos unas imágenes, correspondientes al código del servidor, para entender mejor lo explicado con anterioridad.

```
app.use("/contest", accessControl, contestRouter);
app.use("/problem", accessControl, problemRouter);
app.use("/attempt", accessControl, attemptRouter);
app.use("/user", userRouter);
```

```
contestRouter.post("/", ContestController.createContest);
contestRouter.get("/:id", ContestController.getContest);
contestRouter.patch("/:id", ContestController.editContest);
contestRouter.get("/", ContestController.listContests);
contestRouter.get("/:id/problems", ContestController.listProblemsByContest)
```

5.2.2. Manejo de sesiones

Para conservar información sobre un usuario al pasar de una página a otra de la web, se han usado las sesiones. Dicho de manera mas formal, las sesiones mantienen un estado asociado a un usuario a través de múltiples peticiones para las mismas o distintas páginas web. Para poder implementar el mecanismo de sesiones se han hecho uso de las librerías express-session y connect-mongo(Mongoose, 2011b), la cual require de una sencilla configuración.

```
1  const session = require("express-session");
2  const MongoStore = require("connect-mongo")(session);
3
4  app.use(
5      session({
6          secret: "topsecret",
7          resave: true,
8          saveUninitialized: true,
9          store: new MongoStore({
10             url: replaceAll(config.DB_URL, "$", [
11                 config.DB_USER,
12                 config.DB_PASSWORD,
13                 config.DB_NAME
14             ]),
15             autoReconnect: true
16         })
17     })
18 );
```

5.2.3. *Middlewares*

Como se ha mencionado anteriormente, Express nos permite incorporar *middlewares*. El *middleware* es el software que brinda servicios y funciones comunes a los manejadores de rutas de la aplicación, además de lo que ofrece el sistema operativo. En nuestro caso, el *middleware* que hemos desarrollado gestiona el acceso a las páginas y servicios.

En el caso de la aplicación del cliente, el *middleware* deniega el uso de los servicios del servidor si no se proporciona un token válido.

Si se esta navegando por la interfaz web, impide que usuarios que no han iniciado sesión puedan entrar en páginas que requieran de un login previo.

```
1  const accessControl = (req, res, next) => {
2      if (req.session.user) {
3          if (!validatePath(req)) return res.redirect("/index");
4          res.locals.user = req.session.user;
5          next();
6      } else {
7          let bearerToken = req.headers["authorization"];
8          if (!bearerToken) {
9              res.redirect("/login");
10             return;
11         }
12         let token = bearerToken.split(" ")[1];
```

```
13
14         if (!token)
15             return res.status(504).send({
16                 msg: "No token provided"
17             });
18
19         jwt.verify(token, config.key, (err, decoded) => {
20             if (err) {
21                 return res.json({ msg: "Invalid token" });
22             } else {
23                 req.decoded = decoded;
24                 const user = User.findOne({ _id: decoded.sub });
25                 if (!user) return res.json({ msg: "Invalid token" });
26                 res.locals.user = decoded.sub; // save current id_user in
       the request to be used by controllers
27                 // decoded.sub <- id user
28                 // decoded.exp <- expiration time unix
29                 next();
30             }
31         });
32     }
33 };
```

5.2.4. Roles

En nuestra aplicación existen tres roles diferenciados, los cuales solo tienen sentido si se usa la web. En caso de usar la aplicación de escritorio, el rol no tendrá repercusión en qué acciones puede o no realizar. Eso estará regulado por el JWT.

Los diferentes roles que existen son:

- Admin
- Teacher
- User

Estos roles ya se definieron con anterioridad en el capítulo 4. La gestión de permisos mediante roles puede tener mucha profundidad y complejidad. En nuestro caso se trata de una gestión sencilla que impide a los usuarios ver páginas a las que no tienen acceso. En la imagen que proporcionamos a continuación, mostramos como con *EJS* (*mirar en detalle en la siguiente sección)* y con roles, impedimos que a un usuario con rol User pueda ver la porción de HTML que contiene el botón de editar problemas.

```
1  <% if (user.role === "admin" || user.role == "teacher") { %>
2      <div class="edit">
3          <a
4              href="/problem/edit/<%= data[i].id %>"
5              class="btn btn-primary"
6              role="button"
7              >Edit</a
8          >
9      </div>
10   <% }; %>
```

5.3. Comunicación con la App del Cliente

En esta sección se explicará brevemente como se comunica el Servidor con el Cliente web. En el capítulo 6 se entrara más en detalle acerca del papel que asume el cliente y de las tecnologías usadas.

5.3.1. JSON Web Tokens - JWT

De forma parecida a la comunicación con el cliente web, el usuario realizará peticiones para solicitar ciertos servicios del servidor. No obstante, ahora se requerirá de un token válido para poder acceder a dichos servicios. La tarea del servidor será extraer de la cabecera de las llamadas el token enviado por el usuario. En caso de que la cabecera contenga el token, se verificará si este es válido. Esta tecnología se explicará con detalle en las sección 6.2.2 de la memoria.

5.4. Creación de plantillas y renderizado dinámico

En las aplicaciones web, es frecuente que existan fragmentos de HTML que aparezcan repetidos en numerosas páginas. Un ejemplo claro sería la cabecera de la aplicación. Para este cometido existen numerosas tecnologías y frameworks, no obstante, nuestra aplicación web, al no ser una web muy grande, no requiere de muchas plantillas ni de funcionalidad extra. Tras una previa investigación, decidimos usar *EJS* como motor de plantillas debido a su sencillez.

Para poder crear una plantilla lo único que se necesita es almacenar dichas porciones de HTML en una carpeta llamada *partials* y luego incluirlo en la página que necesitemos. En la siguiente porción de código se muestra un ejemplo de como insertar nuestra plantilla.

```
1  <body>
2        ...
3
4        <%- include("partials/header.ejs", {active:"problems"}) %>
5
6        ...
7  </body>
```

Otra funcionalidad que nos proporciona EJS es la de poder renderizar contenido de manera dinámica. En nuestro proyecto se usa principalmente para el listado de concursos y problemas y para restringir el acceso a diferentes funcionalidades según el rol. En el siguiente fragmento de código se puede ver el renderizado dinámico de los concursos usando EJS.

```
1  <% } %> <% for (var i = 0; i < data.length; i++) { %>
2            <li data-index="<%= data[i].id %>">
3                <div class="contest-info flex">
4                    <div>
5                        <h2><%= data[i].name %></h2>
6                        <p><%= data[i].desc %></p>
7                        <p>
```

```
8                                   <strong>Number of problems:</strong> <%=
9                                   data[i].nproblems %>
10                              </p>
11                            ...
12                       </div>
13                  ...
14             </li>
15             ...
16   <% }; %>
```

5.5. Corrección de envíos

El servidor entra en acción en el momento en el que el usuario envía su solución para el primer lote de prueba. Este primer lote es accesible por el usuario y coincide con los casos de prueba descritos en el enunciado del problema. La solución que el usuario envía es un checksum. El servidor lo recoge y lo compara con el checksum correcto correspondiente al problema y al lote de prueba específico, el cual estará alojado en la BD. En caso de no coincidir también se compara con otro checksum que corresponde a la solución sin tener en cuenta saltos, espacios y tabulaciones. Esto se conoce en otros jueces de programación como "Presentation Error", aunque en nuestra aplicación somos más optimistas y daremos la solución por válida .^{A}ccepted Except Spaces". Una vez que ya se han comparado los checksum, y por tanto se ha obtenido el veredicto para dicho lote (AC, WA, AC-ES), el servidor irá enviando uno a uno los demás lotes de pruebas. El cliente compilará su solución y enviará de nuevo su checksum al servidor, todo esto automáticamente, sin que el usuario haga nada. Este proceso se repetirá hasta tener todos los checksums de todos los lotes de prueba. En ningún momento se dejarán de seguir comprobando los lotes de prueba, aunque se detecte que la solución de uno de ellos sea errónea. Esto permite tanto a profesores como alumnos tener una idea de qué casos de prueba han fallado.

5.6. Generación de checksums

Por último, para que los profesores puedan obtener de manera sencilla los checksums, tanto de las soluciones correctas como de las soluciones correctas salvo espacios, se ha servido de un script para facilitar la tarea.

Este script recibirá como argumentos el nombre del problema en *camelcase* y el número de lote de caso de prueba del que se quiere generar el checksum. La salida del script será un par con el checksum correspondiente a la solución correcta y otro checksum correspondiente a la solución correcta salvo espacios.

```
PS C:\Users\pmori\Desktop\TFG\ServerApp> node .\scripts\checkSums.js AprendiendoASumar 1
[
  '2076aca85a970f40fbef21743d3bf472ca9b719659ffcca70b1cadd57bdd2df2',
  'ecba28bb827265d9a9a86be4e4e8fb684404ae8bf385c2ae367fa9dd57e1e6e6'
]
```

Figura 5.2: Checksum: Script generador de checksums

Capítulo 6

Clientes de interfaz de línea de comandos

Los clientes son una de las partes principales de cualquier sistema, ya que no puede existir un servidor que ofrezca utilidades sin un sistema que la consuma. En este capítulo se detalla todo lo relacionado con el cliente, su evolución y en concreto las funcionalidades ofrecidas por la interfaz de línea de comandos y el cliente de terminal.

6.1. Desarrollo de los clientes

Como primera aproximación, y para probar el correcto funcionamiento del servidor, se utilizó un cliente para *API* llamado *Postman*, que permite generar todo tipo de peticiones del protocolo *HTTP* como *GET*, *POST*, *PUT*, *PATCH* o *DELETE*, además de dar libertad de poder gestionar las cabeceras de cada llamada y la autenticación.

Postman es conocido por ser uno de los software de *testing* modernos más utilizados en la actualidad con más de 20 millones de usuarios, y cuenta con una amplia gama de herramientas y utilidades para facilitar la construcción de aplicaciones (Postman, 2014). Durante el desarrollo de los clientes de *ScholarJudge* lo utilizamos de la forma más sencilla posible y nos ayudó a ir añadiendo funcionalidades al servidor conforme fuimos necesitándolo. Sirvió para ir probando su validez, que las llamadas a los *endpoints* se realizaran de la forma correcta evitando parámetros innecesarios y gestionando los resultados.

Contamos con tres clientes principales que han sido desarrollados de forma paralela durante el transcurso del TFG, cada uno con sus ventajas e inconvenientes y características que definiremos a continuación.

- Cliente de interfaz de linea de comandos
- Cliente de interfaz de línea de comandos interactivo
- Cliente web

6.2. Aspectos comunes a los clientes de línea de comandos

En este capitulo de la memoria quedan detallados tanto la interfaz de línea de comandos como el cliente de terminal. La aplicación web será desarrollada en el siguiente capítulo debido a su complejidad.

6.2.1. El fichero *scholarJudge.json*

Tanto la interfaz de línea de comandos normal como la interactiva hacen uso de un fichero de configuración común presente en todos los concursos que se crean (inicializando con el comando *init*), llamado *scholarJudge.json*. En concreto, se utilizo la extensión de fichero *JSON* debido a su facilidad para acceder a los campos, al ser tratado como un diccionario en la librería de desarrollado usada, *Node.js*. Un ejemplo de la estructura que presenta el mismo queda mostrado en el siguiente fragmento de código.

```
1  {
2    "id": "62701a2af6cadf2042232372",
3    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiI2MjcwMWEyYWY2Y2
4    FkZjIwNDIyMzIzNzYiLCJpYXQiOjE2NTIxNzM2NDksImV4cCI6MTY1MjIxNjg0OX0.
       c1_TUUa_RxeG89ojzKudt9DiwQ-6XUT
5    -u_yygJuA0",
6    "name": "Strings",
7    "desc": "Basics strings problems",
8    "problems": [
9      {
10       "id": "62701a2af6cadf204223236e",
11       "name": "Greetings",
12       "numFiles": 2,
13       "metadata": true,
14       "statement": true,
15       "in": true,
16       "out": true
17     }
18   ]
19 }
```

Este fichero consta de diversos campos:

- *id*: *hash* único definido por *MongoDB* que identifica el concurso.
- *token*: *JWT* que permite identificar al usuario e interaccionar con el sistema.
- *name*: nombre del concurso.
- *desc*: descripción del concurso.
- *problems*: listado de los problemas que ya han sido descargados en la computadora local.
    - *id*: hash único definido por *MongoDB* que identifica el problema.
    - *name*: nombre del problema.
    - *numFiles*: número de ficheros de prueba disponibles.
    - *metadata*: valor booleano para determinar si se han descargado los datos del problema.
    - *statement*: valor booleano para determinar si se ha descargado el fichero *PDF* con el enunciado.
    - *in*: valor booleano para determinar si se ha descargado el fichero de ejemplo de entrada.

- *out*: valor booleano para determinar si se ha descargado el fichero de ejemplo de salida.

Tanto en la interfaz de linea de comandos normal como en la interactiva se genera este fichero al inicializar un concurso.

6.2.2. Gestión del *JWT*

Al comenzar con el proyecto seguimos una primera aproximación al sistema donde dejábamos las credenciales del usuario (email y contraseña) escritas directamente en el fichero *JSON*. Esto daba grandes problemas de seguridad, ya que cualquiera que tuviese acceso al sistema del cliente podría saber sus credenciales de un vistazo al no estar codificadas. Debatimos diversas opciones, entre ellas crear un token temporal que se almacenase en nuestro *MongoDB*, pero esto esto resulta bastante engorroso, ya que implica añadir mas datos a los modelos de nuestra base de datos y gestionar la temporalidad de los *tokens*.

Al final recurrimos a utilizar un acceso mediante la tecnología JWT (Auth0, 2010a). Esta se implementó con un *middleware* en el servidor para autentificar todas las rutas que fueran necesarias y para evitar todos los problemas mencionados anteriormente, ya que no es necesario guardar los *tokens* en nuestra base de datos. La temporalidad es gestionada de forma intrínseca al *token* y permite generar tantos como hagan falta. El *token* creado tiene una duración de doce horas, una vez pasadas las cuales es necesario iniciar sesión de nuevo en nuestro sistema. En concreto, esto se lleva acabo haciendo un llamada *POST* al servidor con el email y contraseña del usuario en cuestión. Tal y como se puede ver en el fichero *scholarJudge.json* de la figura **??** el *JWT* es el parámetro *token*.

6.3. Interfaz de linea de comandos

Una interfaz de linea de comandos o también conocida como *CLI (Command line interface)* es un tipo de cliente que permite interaccionar con el usuario mediante líneas de texto llamadas comandos que se ejecutan en una terminal o *shell*. Esta clase de clientes tienen una componente histórica ya que se han utilizado desde los inicios de las computadoras, pero esto no implica que no se sigan utilizando ampliamente en la informática moderna. Normalmente y gracias a su sencillez brindan más funcionalidades que los *GUI (Graphic user interface)*, que pueden sacrificar ciertas funcionalidades por mejorar la forma de interactuar con el usuario, además de incrementar la complejidad y por lo tanto sus vulnerabilidades. Los comandos en *CLI* más populares en *GNU/Linux* podrían ser *cd*, *ls* o *gcc*, entre otros.

6.3.1. Funcionalidades

En esta primera subsección se explican los comandos definidos en la interfaz de línea de comandos de *ScholarJudge*.

- Comando inicializar (*init* figura 6.1): Inicializa el directorio del nuevo concurso y realiza el inicio sesión en el servidor, creando la carpeta y el fichero de configuración *scholarJudge.json*. Recibe como parámetro el nombre de la carpeta y permite seleccionar el concurso entre los disponibles en el sistema.

- Comando concursos (*contests* figura 6.2): Muestra la lista del concursos disponibles.

```
MacBook-Pro:ClientApp victorcavero$ scholarjudge init Matrix
Log in with the email and password...
Email: victor@ucm.es
Password: *********
- Contest 1
   Contest name: Iterative
   Description: Basic iterative problems. Suitable for b...
   Problems: Aprendiendo a sumar,Un buen inicio

- Contest 2
   Contest name: Matrix
   Description: Matrix Searching. This contest contains ...
   Problems: The magic matrix

- Contest 3
   Contest name: Strings
   Description: Basics strings problems...
   Problems: Greetings

Select contest: 2
```

Figura 6.1: CLI Comando init

```
MacBook-Pro:Matrix victorcavero$ scholarjudge contests
- Contest 1
   Contest name: Iterative
   Description: Basic iterative problems. Suitable for b...
   Problems: Aprendiendo a sumar,Un buen inicio

- Contest 2
   Contest name: Matrix
   Description: Matrix Searching. This contest contains ...
   Problems: The magic matrix

- Contest 3
   Contest name: Strings
   Description: Basics strings problems...
   Problems: Greetings
```

Figura 6.2: CLI Comando contests

- Comando problemas (*problems* figura 6.3): Muestra la lista de problemas para el concurso definido en el directorio actual (determinado por el fichero *scholarJudge.json*) indicando también si ya ha sido resuelto.

```
MacBook-Pro:Matrix victorcavero$ scholarjudge problems
- Problem 1 ~ Unsolved
   Problem title: The magic matrix
```

Figura 6.3: CLI Comando problems

- Comando siguiente (*next* figura 6.4): Descarga los ficheros necesarios del siguiente problema que le corresponde al usuario actual. Como parámetro de entrada permite seleccionar tres opciones: descargar solo el *PDF* del enunciado (opción 1), solo el fichero de entrada (opción 2) o solo el de salida (opción 3). Además añade al fichero de configuración *scholarJudge.json* información de los datos descargados.

```
MacBook-Pro:Matrix victorcavero$ scholarjudge next 1
Downloading your next problem: The magic matrix...
Done!
```

Figura 6.4: CLI Comando next

- Comando estadísticas (*stats* figura 7.15): Muestra las estadísticas del concurso de cada uno de los problemas, en particular, el número de intentos correctos y fallidos.

```
MacBook-Pro:strings victorcavero$ scholarjudge stats
- Problem 1 : Greetings
   Correct: 7
   Fail: 2
   Total: 9
```

Figura 6.5: CLI Comando stats

- Comando problema (*problem* figura 6.6): Descarga un problema en concreto del concurso. Permite seleccionar tres opciones: descargar solo el *PDF* del enunciado (opción 1), solo el fichero de entrada (opción 2) o solo el de salida (opción 3). Al igual que el comando *next*, añade al fichero de configuración *scholarJudge.json* información de los datos descargados.

```
MacBook-Pro:Matrix victorcavero$ scholarjudge problem 1
Available problems:
- 1: The magic matrix
Select the problem you want to get (1-1):1
```

Figura 6.6: CLI Comando problem

- Comando solución (*check* figura 6.7): Envía el *checksum* del fichero solución para que sea comprobado el intento. Recibe como argumentos el nombre del archivo solución y el número del problema que se intenta solucionar (número mostrado por el comando *problems*).

```
MacBook-Pro:Greetings victorcavero$ scholarjudge check 1 ./a.out
Correct
[ 'AC', 'AC-ES' ]
```

Figura 6.7: CLI Comando check

6.3.2. Descripción técnica

ScholarJudge se puede instalar de forma global con las herramientas *npm* o *yarn* para que pueda ser utilizado desde cualquier directorio en nuestro sistema. Después de esto, cualquiera de los comandos mencionados anteriormente se podría ejecutar con el programa *scholarjudge* seguido del comando y los argumentos del mismo. Esto se aprecia con más detalle en la figura 6.8. Se ha utilizado como librería de desarrollo de interfaces de línea de comandos el paquete *Yargs* (Yargs, 2013), que sirve para crear *CLI* interactivas ofreciendo funcionalidades ampliamente utilizadas en estas como son los comandos (como *init* o *next*), parámetros, opciones (como *-v*), argumentos y mensajes de ayuda por defecto (*–help*). Incluso es capaz de ofrecer un texto de ayuda automático al definir cada uno de los comandos, tal y como se muestra en la figura 6.8.

```
MacBook-Pro:ClientApp victorcavero$ scholarjudge

Usage: scholarjudge <contests|problems|init|stats|next|problem|check> [args]

Comandos:
  scholarjudge init      Login and init a contest.
  scholarjudge contests  List all contests.
  scholarjudge problems  List all problems.
  scholarjudge next      Get the next problem.
  scholarjudge stats     Get stats of all problems in the contest.
  scholarjudge problem   Get a problem.
  scholarjudge check     Check a problem.

Opciones:
  --version  Muestra número de versión                            [booleano]
  --help     Muestra ayuda                                        [booleano]

Please provide at least one command _
```

Figura 6.8: CLI Texto de ayuda

6.4. Interfaz de linea de comandos interactivo

Por otro lado y de forma paralela también se ha desarrollado un cliente de terminal interactivo. Este ofrece las mismas características presentes de la interfaz de línea de comandos pero afrontadas desde otra perspectiva, la ejecución continua de la aplicación. La aplicación de terminal permite seleccionar con las flechas de dirección o el teclado numérico la opción que queremos ejecutar. Esto nos lleva a ver que una de las principales ventajas respecto al *CLI* es la sencillez de su utilización ya que a diferencia de la aplicación de terminal, si no tenemos experiencia previa utilizando comandos de *CLI* la curva de aprendizaje es más inclinada para el no interactivo.

6.4.1. Funcionalidades

Como hemos comentado en la subsección anterior, el cliente de terminal ofrece las mismas utilidades que el *CLI* pero presentadas de otra manera. Un ejemplo de la interfaz gráfica que nos encontraremos se muestra en la figura 6.9.

```
========================
  * Scholar Judge! *
========================

? Choose an option (Use arrow keys)
❯ 1. Log in
  2. List contests
  3. List problems
  4. Init contest
  5. Get stats
  6. Next problem
  7. Get problem
(Move up and down to reveal more choices)
```

Figura 6.9: Terminal menú principal

6.4.2. Descripción técnica

La aplicación de terminal ha sido desarrollada utilizando el paquete *Inquirier* (Boudrias, 2013). Este permite generar menús interactivos pudiendo: leer datos de entrada (con diversos tipos, ya sea para contraseñas o texto plano), gestionar colores, validar respuestas, definir preguntas y opciones mediante diccionarios, utilizar las flechas de nuestro teclado para movernos entre estas opciones, etcétera.

6.5. Historia de usuario básica común

El recorrido que haría un usuario normal, por orden, durante su primer uso de *ScholarJudge* con el objetivo de afrontar un problema y solucionarlo sería:

1. *Init*: Iniciar sesión en nuestro sistema y crear la carpeta del concurso.
2. *Contests*: Obtener la lista de concursos por si el usuario decide inicializar otro.
3. *Problems*: Obtener la lista de problemas para elegir cuál se desea resolver.
4. *Next* o *Problem*: Obtener el problema que se quiere solucionar. Cualquiera de los dos comandos puede servir, ya sea obtener el siguiente del concurso o un problema en concreto.
5. *Check*: Comprobar la solución al problema que se intenta resolver.

6.6. Llamadas al servidor

Las conexiones con el servidor son gestionadas en la carpeta *helpers* de nuestro proyecto. En ella tenemos un fichero por cada tipo de acción que queramos tratar (concursos, problemas, intentos y usuarios). En ellos hacemos las llamadas al servidor necesarias para cada comando y gestionamos las respuestas según lo necesite el cliente. Un ejemplo de llamada básica es la mostrada en el código inferior, en la cual se puede observar un método

para obtener la lista de concursos presentes en el sistema, recibiendo como entrada el *JWT* que autentica al usuario en nuestro servidor. La librería *node-fetch* realiza llamadas de tipo *GET* por defecto y hemos añadido a las cabeceras el campo *Authorization Bearer* con el *JWT* de acceso. Esto último lo hace el método *getRequest*.

```
const readContest = async (contest_id, token) => {
    const result = await fetch(
        urljoin(url, String(contest_id)),
        getRequest(token)
    );
    if (result.error) throw new FetchError(result.error);

    const parseResult = await result.json();

    const contest = new Contest(
        parseResult.contest._id,
        parseResult.contest.name,
        parseResult.contest.desc,
        parseResult.contest.problems
    );

    return contest;
};
```

Capítulo 7

Aplicación web

En este capítulo queda definida al completo la página web, la plataforma de *hosting*, las librerías utilizadas y sus principales vistas.

7.1. *Hosting*

La plataforma de *hosting* que hemos utilizado para mantener nuestra plataforma es Heroku, seleccionando, en concreto, el plan gratuito para evitar incurrir en costes para el proyecto. El plan es el presente en la figura 7.1. La aplicación es accesible a través de la página web *https://scholarjudge.herokuapp.com*.

Heroku nos ha permitido integrar de una forma muy sencilla el repositorio de GitHub, donde tenemos alojado el código, con su plataforma de manera que cualquier *git push* que se haga en la rama *master* es compilado y desplegado automáticamente a producción. Esto se conoce como integración continua o *CI*.

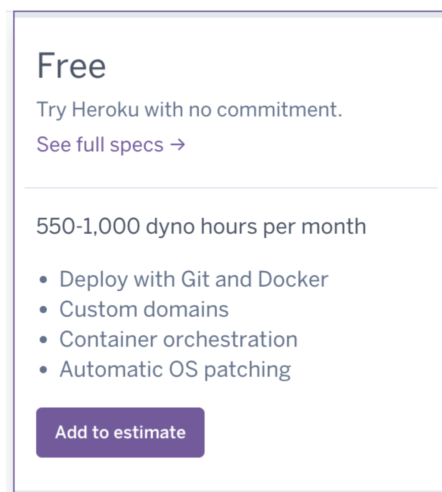


Figura 7.1: Plan de *hosting* gratuito de *Heroku*

7.2. Librerías utilizadas

- *Bootstrap*: Es una biblioteca multiplataforma que ofrece multitud de componentes, cuadrículas, formularios y en definitiva facilita el desarrollo de interfaces gráficas (Bootstrap, 2011). Permite personalizar cada uno de los componentes, ya sean sus botones o funciones, además de hacerlos dinámicos y adaptables.

- *Node-sass (Syntactically Awesome Style Sheets)*: Es una librería que permite utilizar *CSS (Cascade Style Sheet)* extendido con comandos nuevos y funcionalidades agregadas que ofrecen gran utilidad, por ejemplo selectores anidados o módulos (SASS, 2012). Bootstrap permite el uso de *SASS* (SASS, 2006) para crear un tema personalizado definiendo los colores globales a todos los componentes que ofrece. A continuación se define el fichero *custom.scss* utilizado. En nuestra aplicación se usó una paleta de colores azules donde cada uno queda especificado en codificación hexadecimal.

```
1         $white: #fffeff;
2 
3         $theme-colors: (
4             "light": #ddedf8,
5             "dark": #000500,
6             "primary": #22379d,
7             "secondary": #2473ef,
8             "info": #4091f2,
9             "accent1": #004576,
10            "accent2": #0070c9,
11            "accent3": #54b0db,
12            "success": #00a35e,
13            "warning": #e1cb00,
14            "danger": #f01c00
15        );
16
17        @import "../../node_modules/bootstrap/scss/bootstrap";
18
```

7.3. Vistas

7.3.1. Menú de navegación

Existe un menú común a todas las vistas que es el que se muestra en la figura 7.2. Este varía dependiendo si el usuario es o no administrador. En caso de no serlo la opción de *users* no sería visible ni accesible.

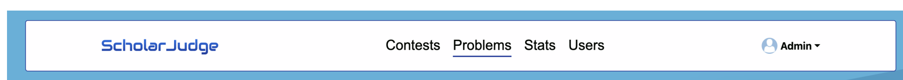


Figura 7.2: ScholarJudge: vista del menú en modo administrador o profesor

Además, el menú consta de un desplegable en el lateral que permite editar la información del usuario y salir de la sesión. Esto se puede observar con más detalle en la figura 7.3.

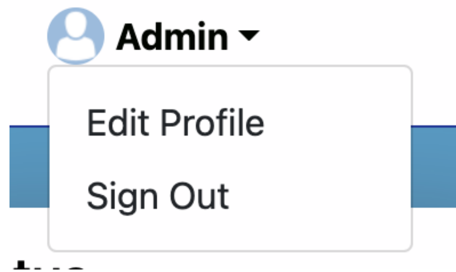


Figura 7.3: ScholarJudge: vista del menú desplegable

También se incorporó la opción de colapsado, como se observa en la figura 7.4, que ofrece *Bootstrap*, de las opciones disponibles en el menú para permitir que en dispositivos con pantallas de menor anchura no se solaparan las opciones.

Figura 7.4: ScholarJudge: vista del menú colapsado

7.3.2. Vistas principales

Las vistas accesibles al entrar en la página web son:

- Inicio de sesión (*Log in*) figura 7.5: Permite utilizar las credenciales del usuario (email y contraseña) para acceder a la plataforma. Se comprueba la validez tanto del parámetro email como el de la contraseña. En caso de error muestra un mensaje para informar al usuario.

- Registro (*Sign Up*) figura 7.6: Permite registrarse en el sistema, es decir, almacenar el nuevo usuario en nuestra base de datos (*MongoDB*) para permitir al nuevo usuario iniciar sesión. Los parámetros que debe rellenar para registrarse son el email, nombre, contraseña y captcha. Esto último permite evitar ataques de *bots scrappers* que pretendan inundar nuestra BBDD con multitud de usuarios falsos pretendiendo corromper así nuestro sistema. Al igual que en el inicio de sesión, si alguno de los parámetros es incorrecto se muestra al usuario un mensaje de error para informarle al respecto.

Una vez que se ha realizado el inicio de sesión en *ScholarJudge* y dependiendo del tipo de usuario que sea, tiene acceso a determinadas vistas. Los roles que se contemplan en el sistema actualmente son administrador, profesor y usuario normal. Los administradores y profesores pueden acceder a las mismas vistas ya que la única diferencia entre estos es que el administrador puede crear nuevos administradores o profesores pero el profesor no puede gestionar los administradores y sí crear otros profesores.

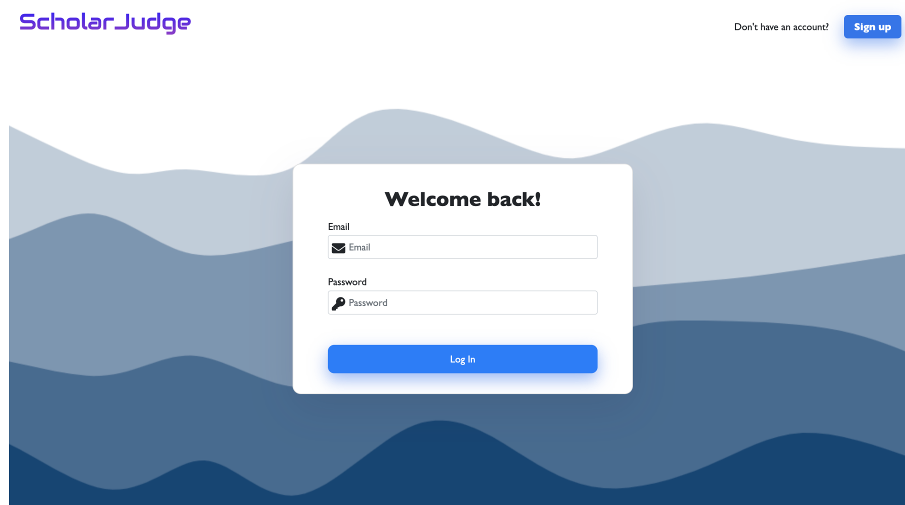


Figura 7.5: ScholarJudge: vista del inicio de sesión

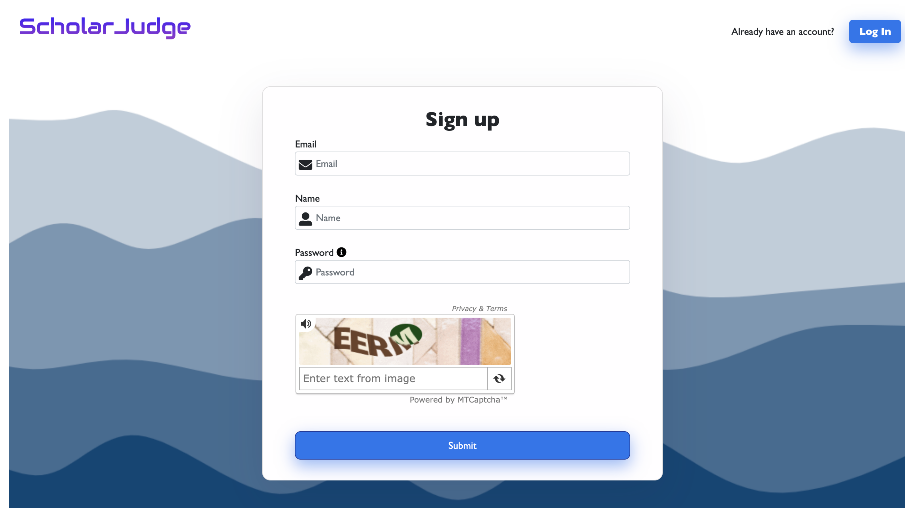


Figura 7.6: ScholarJudge: vista de registro

Las vistas de *ScholarJudge* son:

- Concursos (figura 7.7): muestra todos los concursos disponibles en la base de datos. Además consta de un buscador que permite filtrar los resultados por la categoría que tienen asignados.

- Concursos en vista de administrador o profesor (figura 7.8): aparte de mostrar lo mismo que en la vista de usuario normal, aparece también un botón *Add contest* que permite añadir un nuevo concurso y un botón *Edit* que permite editar los actuales.

    - Añadir concurso (figura 7.9): permite añadir un nuevo concurso a nuestra base de datos mediante un formulario que recibe como parámetros el nombre y la descripción.

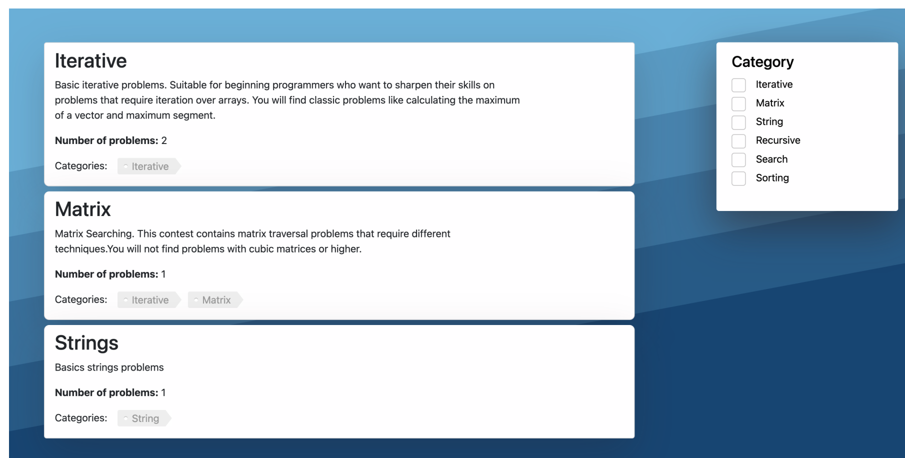


Figura 7.7: ScholarJudge: vista de concursos

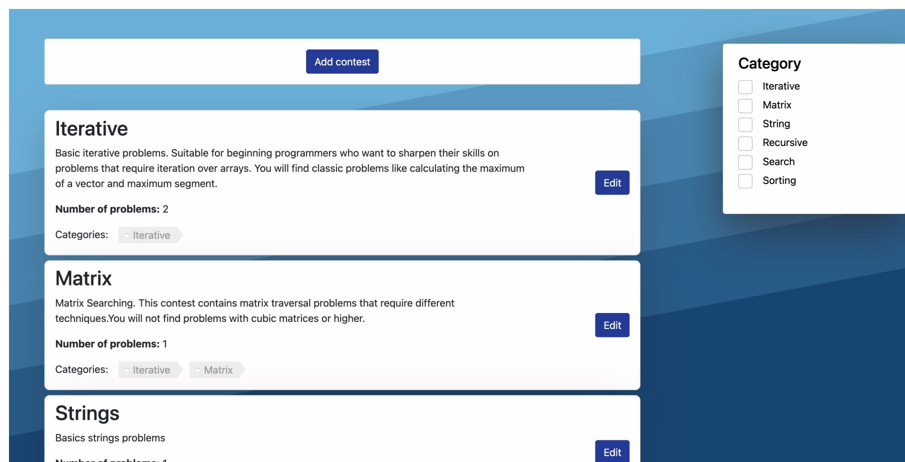


Figura 7.8: ScholarJudge: vista de administrador y profesor de concursos

    - Editar concurso (figura 7.10): vista que permite editar un concurso determinado en la vista de concursos. Se pueden modificar del concurso: el nombre, la descripción, la categorías y los problemas y su orden. Además, en caso de error o de haberse realizado correctamente se muestra un mensaje para informar al usuario.

- Problemas (figura 7.11): muestra todos los problemas disponibles en la base de datos. Además consta de un buscador que permite filtrar los resultados por la categoría que tienen asignados y por si han sido resueltos o no.

- Problemas en vista de administrador o profesor (figura 7.12): aparte de mostrar lo mismo que en la vista de usuario normal, aparece también un botón *Add problem* que permite añadir un nuevo problema y botón *Edit* que permite editar los actuales.

    - Añadir problema (figura 7.13): permite añadir un nuevo problema a nuestra base de datos mediante un fichero ZIP que siga la estructura definida en la

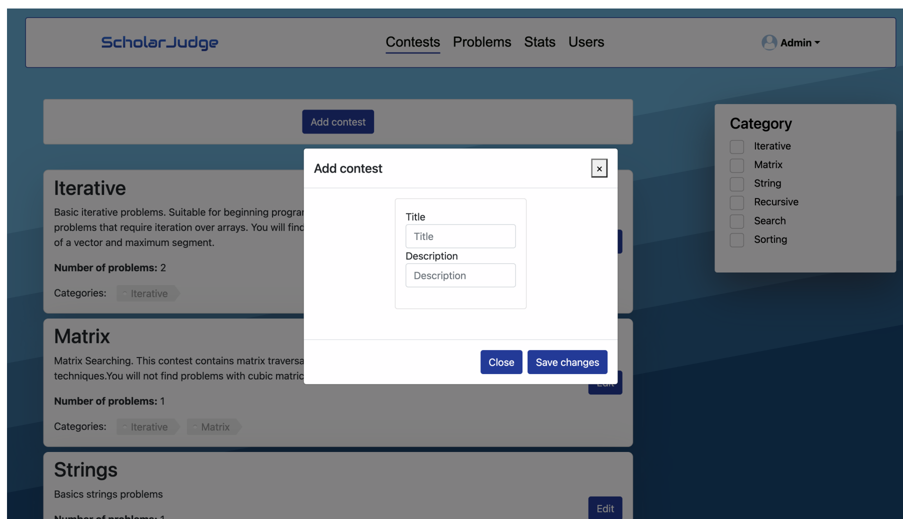


Figura 7.9: ScholarJudge: vista de administrador y profesor de añadir concurso

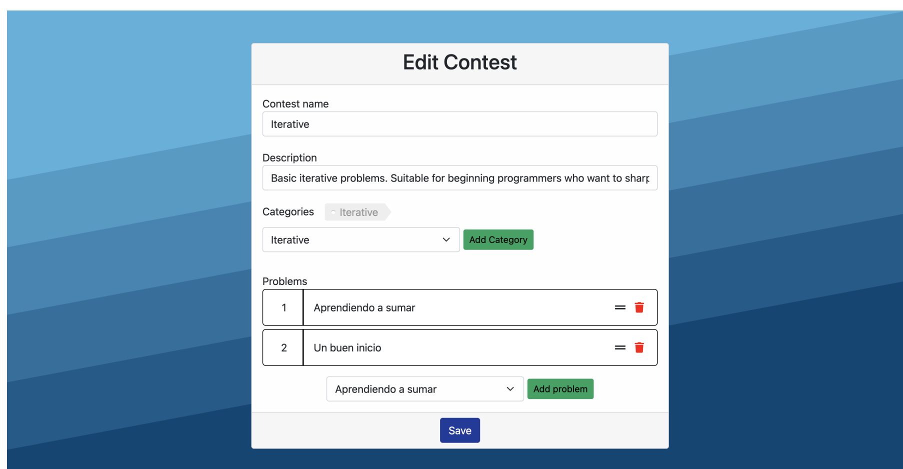


Figura 7.10: ScholarJudge: vista de administrador y profesor de editar concurso

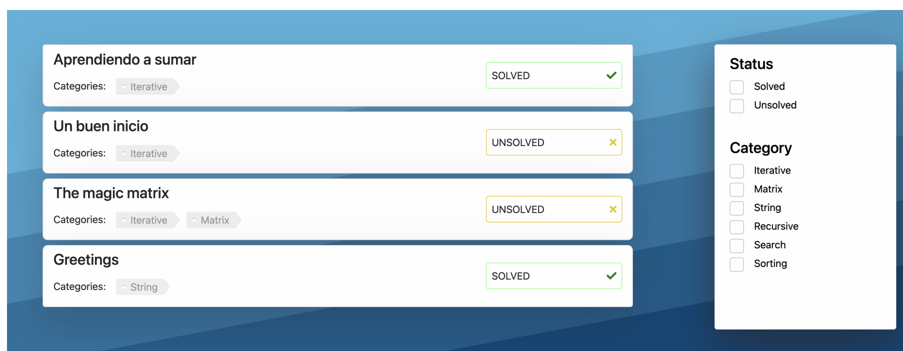


Figura 7.11: ScholarJudge: vista de problemas

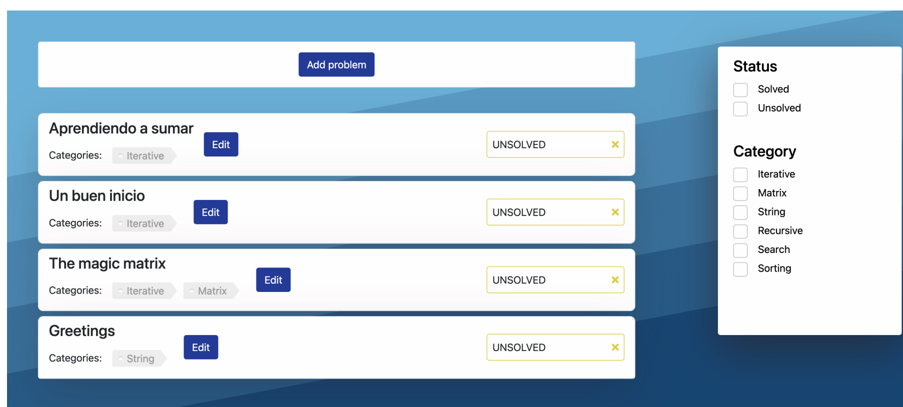


Figura 7.12: ScholarJudge: vista de administrador y profesor de problemas

figura 7.13 y con los documentos necesarios (casos de prueba de entrada, salida y el documento del enunciado).

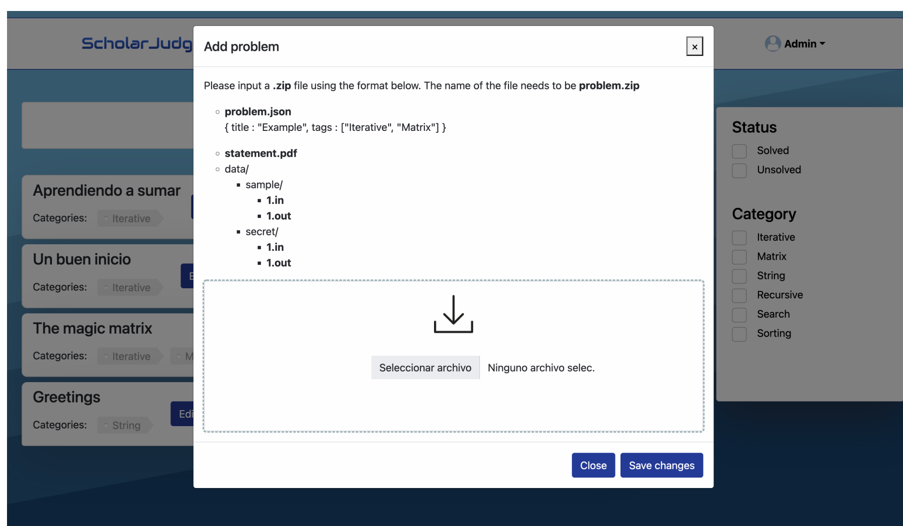


Figura 7.13: ScholarJudge: vista de administrador y profesor de añadir problema

  - Editar problema (figura 7.14): permite editar el problema seleccionado en la vista de problemas, pudiendo modificar del mismo: el nombre, categorías y los ficheros de entrada y salida.

- Estadísticas (figura 7.15): muestra las estadísticas para el usuario que ha iniciado sesión en el sistema, con porcentajes de las entregas de sus soluciones y una tabla a modo de resumen.

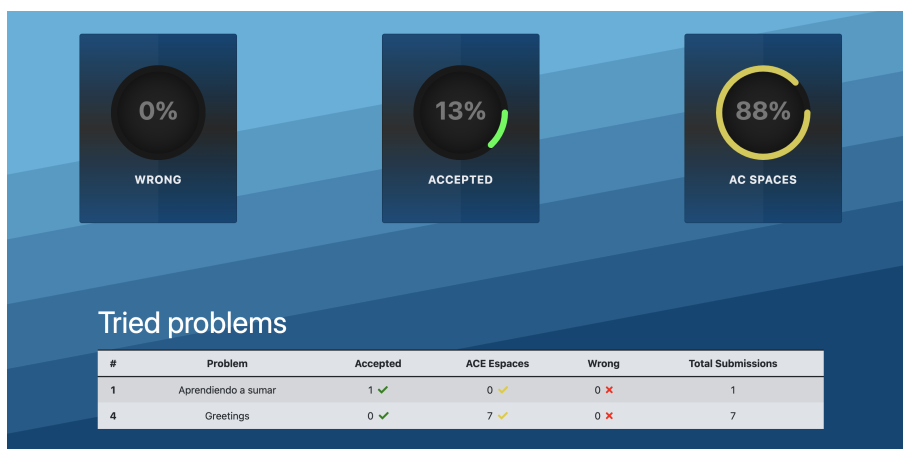
- Lista de usuarios en vista de administrador o profesor (figura 7.16): en esta vista aparece una lista de los usuarios presentes en el sistema, con sus datos y permitiendo modificar sus contraseñas y cambiar sus roles.

- Editar perfil (figura 7.17): vista que permite al usuario cambiar sus datos (nombre, email y contraseña).

Edit Problem

Problem title
Aprendiendo a sumar
Tags
Samples
1.IN Seleccionar archivo Ninguno archivo selec.
1.OUT Seleccionar archivo Ninguno archivo selec.
Secret
1.IN Seleccionar archivo Ninguno archivo selec.
1.OUT Seleccionar archivo Ninguno archivo selec.
2.IN Seleccionar archivo Ninguno archivo selec.
2.OUT Seleccionar archivo Ninguno archivo selec.
3.IN Seleccionar archivo Ninguno archivo selec.
3.OUT Seleccionar archivo Ninguno archivo selec.
Save

Figura 7.14: ScholarJudge: vista de administrador y profesor de editar problema



#	Problem	Accepted	ACE Espaces	Wrong	Total Submissions
1	Aprendiendo a sumar	1 ✓	0 ✓	0 ✗	1
4	Greetings	0 ✓	7 ✓	0 ✗	7

Figura 7.15: ScholarJudge: vista de estadísticas

Manage Users

Show 10 entries Search:

Name (email) Role	
Admin (admin@ucm.es) *admin*	Edit password
Manuel Montenegro (mmontene@ucm.es) *user*	Upgrade to teacher · Edit password
Marta (marta@ucm.es) *user*	Upgrade to teacher · Edit password
Pablo (pablo@ucm.es) *user*	Upgrade to teacher · Edit password
Test (test@ucm.es) *user*	Upgrade to teacher · Edit password
Victor (victor@ucm.es) *user*	Upgrade to teacher · Edit password

Showing 1 to 6 of 6 entries Previous 1 Next

Figura 7.16: ScholarJudge: vista de usuarios

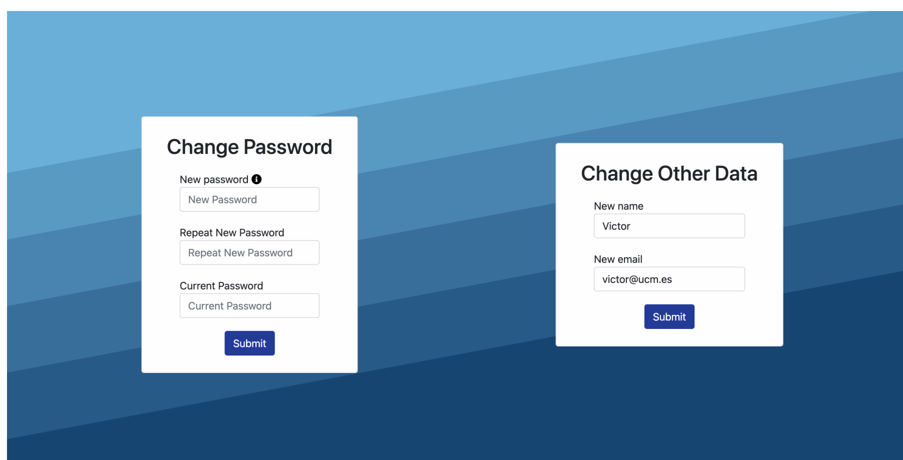


Figura 7.17: ScholarJudge: vista de editar perfil

Capítulo 8

Conclusiones y trabajo futuro

En este capítulo se detallan las conclusiones finales al desarrollo del proyecto, la discusión crítica de los objetivos logrados, las dificultades afrontadas y las líneas de trabajo futuras.

8.1. CLI

En lo que respecta al primer objetivo propuesto, este fue de los primeros en ser abordado. En cuanto tuvimos una API con las funcionalidades básicas pudimos consumir los *endpoints* y crear una interfaz de línea de comandos. Este CLI nos sirvió para probar la validez del servidor al completo y otorgar a la interfaz de línea de comandos las funcionalidades mínimas relacionadas con la gestión de problemas, como son: inicializar un concurso (comando *init*), descargar un problema (comando *problem*) y comprobar una solución (comando *check*). Una vez desarrollado el esqueleto del juez, pudimos ir iterando sobre la versión inicial e ir añadiendo nuevas características al sistema, como la gestión de usuarios, obtención de estadísticas (comando *stats*) o descarga del siguiente problema asignado a un usuario (comando *next*). Uno de los puntos positivos de *ScholarJudge* con respecto a otros jueces en línea es que la entrega de soluciones es indiferente del lenguaje de programación. Esto se ve reflejado en el CLI, que a su vez permite al servidor abstenerse de contar con el *software* para procesar los lenguajes de cada entrega. Este apartado ha sido desarrollado al completo y hemos cumplido los objetivos esperados, dejando abierta la posibilidad de añadir en un futuro nuevos comandos o funcionalidades, como podrían ser: diferenciar roles, subir un problema nuevo al sistema o gestionar usuarios.

8.2. Aplicación web

La aplicación web fue desarrollada en las últimas etapas del proyecto pero, a pesar de ello, fuimos capaces de permitir que los profesores puedan gestionar los concursos, problemas y usuarios. A su vez, los estudiantes pueden visualizar de una manera más llamativa la información referente a los concursos, problemas y estadísticas propias.

El único objetivo que no pudo ser afrontado de forma completa fue crear un diseño adaptable tanto a ordenadores como tablets o móviles. Es decir, un diseño que fuera dinámico y actualizase las vistas en función del tamaño de cada una de las pantallas. Si accedemos a la aplicación a través de un dispositivo móvil, se puede observar cómo los elementos de la interfaz quedan descuadrados o se solapan los unos con los otros. Se podría

mejorar en esta línea de diseño, en concreto, haciendo uso de las características y parámetros de los componentes de Bootstrap (la librería gráfica utilizada) pero por falta de tiempo, no se pudo llevar a cabo esta extensión en el proyecto. Esta línea de trabajo futura se ha comentado con más detalle, esbozando una posible implementación, al final de este mismo capítulo.

8.3. Autenticación

Inicialmente, y durante el desarrollo de nuestro sistema, no existía el sistema de autenticación. Los envíos eran recibidos y gestionados por el servidor sin saber su procedencia. Esto fue suficiente como primera aproximación para probar la validez del sistema, aunque decidimos que sería interesante que cualquier usuario, ya sea profesor, administrador o alumno, pudiese llevar un seguimiento exhaustivo de todas sus interacciones con el sistema. El objetivo principal fue permitir que cada estudiante pueda llevar un registro de sus envíos, estadísticas y problemas realizados, y que los profesores y administradores pudieran gestionar la plataforma pudiendo editar problemas, añadir nuevos problemas o añadir concursos, entre otros. Este objetivo fue desarrollado al completo ya que cualquier usuario puede iniciar sesión en nuestro sistema con sus credenciales (email y contraseña) y automáticamente se le reconoce su rol y sus estadísticas asociadas. Sin embargo, no fue tan sencillo como cabría esperar, porque tuvimos dificultades a la hora de integrar el inicio de sesión de los tres clientes (CLI, CLI interactivo y cliente web) y en concreto la autenticación de usuarios por JWT fue uno de los mayores desafíos del sistema. Esto último se comentará con más detalle en la siguiente sección.

8.4. Roles

Tal y como se ha comentado en la subsección anterior, el sistema de autenticación nos permitió crear un sistema de roles (administrador, profesor y usuario) cada cual con funcionalidades únicas. El objetivo principal era permitir a los profesores y administradores acceder a funcionalidades de gestión de contenido (añadir problemas, crear concursos, etcétera) a los que los estudiantes no pudieran tener acceso. Esto fue desarrollado al completo sin tener que afrontar demasiados problemas durante el desarrollo y dejando espacio para líneas de trabajo futuras, pudiendo añadir nuevos roles como podría ser el rol de delegado, director o incluso dar la posibilidad de crear grupos por clases.

8.5. Desafíos encontrados

- Comprobar soluciones: el comando *check* fue uno de los más complicados de implementar debido a su complejidad técnica, y al ser la pieza principal de *ScholarJudge*, era el que más atención necesitaba. El comando ejecuta de forma local los casos de prueba y manda al servidor el *checksum* de los resultados para comprobar si la solución es correcta o incorrecta. Implementar estas comprobaciones fue una tarea laboriosa puesto que tuvimos que aprender a manejar ficheros con *Node.js* y sobre todo a ejecutar los ficheros solución y que estos fuesen independientes del sistema instalado en el cliente local.

- Autenticación: Encontrar la manera óptima de realizar la autenticación en el sistema, sobre todo mediante la aplicación de CLI, fue una tarea complicada que nos llevó

bastantes pruebas y errores hasta encontrar la solución. Fue necesario incorporar un sistema de gestión de JWT que permitiese a cualquier usuario iniciar sesión en nuestro sistema sin exponer sus credenciales en su computador local ni consumiendo almacenamiento extra en nuestra base de datos MongoDB. Tuvimos que familiarizarnos con la tecnología JWT y con la librería *jsonwebtoken* (Auth0, 2010b), que es la que se utiliza en el servidor para generar los *tokens* de inicio de sesión. También tuvimos que decidir un tiempo de validez para los *tokens* que fuese justo y que permitiese al usuario interactuar con el sistema sin tener que introducir constantemente sus credenciales, pero evitando también el concederle acceso ilimitado. Llegamos a la conclusión de escoger una duración de 12 horas.

- Migrar y eliminar librerías desfasadas: Las llamadas del protocolo *HTTP* desde el cliente al servidor las hacíamos con una librería llamada *request*, la cual actualmente no se continua su desarrollo y se recomienda no utilizar, tal y como indica su web, por lo que tuvimos que migrar el sistema a la librería *node-fetch*. Esto que podría parecer un cambio efímero es uno de los grandes riesgos de usar software libre y depender de otros desarrolladores para el correcto funcionamiento de tu aplicación, ya que nunca sabes en que momento puede cambiar su licencia de uso, dejar de mantenerlos o cambiar el repositorio al completo. Esto último ya ha ocurrido en otros proyectos de gran importancia como por ejemplo con el paquete *faker* o el de *colors*, en los cuales el desarrollador eliminó todo su código como motivo de protesta al estar siendo utilizado de forma indiscriminada por grandes multinacionales sin retribuir recompensa económica para el creador. Ante esto *GitHub*, la empresa que tenia hospedado en su dominio el código, revocó el acceso al propio creador. Esto nos lleva pensar, ¿hasta que punto es nuestro el código que desarrollamos?

8.6. Trabajo futuro

ScholarJudge es un sistema completo, ya que ofrece una amplia gama de utilidades, pero aún existen ramas de trabajo futuro posibles que se puede desarrollar para complementar las funcionalidades actuales. Los puntos posibles de mejora de nuestro sistema son:

- Añadir contenedores: se podría hacer uso de un sistema de contenedores tanto en el cliente como en el servidor, utilizando *Docker*. Los contenedores permitirían, entre otros, ejecutar *ScholarJudge* en cualquier sistema sin tener instaladas todas las dependencias necesarias como serían una version concreta de *Node.js* o de *Bootstrap*. Además, *Docker* ofrece a los desarrolladores una manera fiable de crear, enviar y ejecutar aplicaciones distribuidas en cualquier escala. Para incorporar Docker deberíamos crear un archivo *Dockerfile* en el cual se tuviese la configuración necesaria común y los comandos de ejecución. La configuración debería contener las siguientes sentencias: el sistema operativo o imagen base (normalmente se suele utilizar una imagen de *GNU/Linux* optimizada en espacio y las características mínimas necesarias), los comandos de instalación de una version de *Node.js* y que contase con los administradores de paquetes npm o yarn, los comandos de uso de npm o yarn para instalar los paquetes necesarios en la imagen anterior y, por último, el comando para iniciar la aplicación. Un ejemplo de archivo *Dockerfile* podría ser:

```
1 # build environment
2 FROM node:16.14.0-alpine3.14 as build
3 WORKDIR /app
4 ENV PATH /app/node_modules/.bin:$PATH
```

```
5 COPY . ./
6 RUN yarn install
7 EXPOSE 3000
8 CMD ["yarn", "start"]
```

- Diseño adaptable y dinámico: Mejorar el diseño de la aplicación web para que este pueda ser lo más adaptable posible a los distintos dispositivos. La librería gráfica utilizada, *Bootstrap*, permite crear configuraciones para diseños adaptables que se podrían incorporar a *ScholarJudge*. En concreto, tendríamos que definir los conocidos como *breakpoints*, que son marcas de tamaño que se utilizan en el CSS para modificar sus valores dependiendo del tamaño de la pantalla. Un ejemplo de configuración de *breakpoints* se muestra a continuación:

```
1     // Extra small devices (portrait phones, less than 576px)
2     // No media query since this is the default in Bootstrap
3
4     // Small devices (landscape phones, 576px and up)
5     @media (min-width: 576px) { ... }
6
7     // Medium devices (tablets, 768px and up)
8     @media (min-width: 768px) { ... }
9
10    // Large devices (desktops, 992px and up)
11    @media (min-width: 992px) { ... }
12
13    // Extra large devices (large desktops, 1200px and up)
14    @media (min-width: 1200px) { ... }
```

- Añadir un domino y mejorar el *hosting*: comprar un dominio personalizado para otorgar de mayor entidad a *ScholarJudge* y añadir un *hosting* personalizado a las necesidades de la web. Esto incurriría en gastos extra para el desarrollo, teniendo que realizar pagos periódicos mensuales o anuales para mantener el dominio y el hosting. En nuestro caso, *ScholarJudge* ha sido desarrollado de forma totalmente gratuita, por lo que el presupuesto sería un factor limitante para esta mejora.

- Creación de *tests*: durante el desarrollo de nuestro sistema nos encontramos en situaciones repetitivas donde tuvimos que ejecutar el mismo método con los mismos parámetros una y otra vez para probar su validez. Esto, a parte de ser sumamente ineficiente en términos de tiempo y esfuerzo, puede ser programable mediante tests que permitan reconstruir la situación completa. Con ello se mejoraría la experiencia de desarrollo, pudiendo centrar los esfuerzos en otros aspectos del sistema. Se ha incluido una pequeña parte de tests en *ScholarJudge* usando la librería *chai*. Un ejemplo concreto para obtener el JWT de acceso es el siguiente, donde se comprueba que se recibe de forma correcta el *token* correspondiente para el usuario con email *test@ucm.es* y contraseña *Test123*.

```
1 before(function (done) {
2     this.timeout(TIME_OUT);
3     chai.request(app)
4         .post("/user/test@ucm.es/token")
5         .send({ password: "Test123" })
6         .end((error, res) => {
```

```
7                 global.token = "Bearer " + res.body.token;
8                 token = "Bearer " + res.body.token;
9                 done();
10            });
11    });
```

Habría que extender estos tests para cubrir la funcionalidad de todos los comandos que se contemplan en nuestro sistema como son: descargar un problema, ver los concursos y sobre todo comprobar si una solución es correcta ya que es de los procesos más tediosos para comprobar de forma manual.

- Mejorar la integración continua: este paso puede ir de la mano del anterior, puesto que los tests deben ser parte del proceso de integración continua. Podríamos usar un *software* como *Jenkins* que comprobase los cambios en el repositorio de código y sea capaz de construir y desplegar el proyecto de forma automática, ejecutando los tests creados y dando información a los desarrolladores sobre el resultado de cada una de estas etapas.
- Documentación del código: mejorar la documentación de cada uno de los métodos, tanto del servidor como del cliente, para permitir a nuevos desarrolladores incorporarse al proyecto de una forma más sencilla, indicando el objetivo de cada uno de los parámetros de entrada y salida.

Chapter 9

Conclusions and Future Work

This chapter details the final conclusions to the development of the project, the critical discussion of the objectives achieved, the difficulties faced and future lines of work.

9.1. CLI

Regarding the first proposed objective, this was addressed among the first. As soon as we had an API with the basic functionalities we were able to consume the endpoints and create a command line interface. This CLI was used to test the validity of the entire server and grant to the command line interface the minimum functionalities related to problem management, such as: initialize a contest (init command), download a problem (problem command) and check a solution (check command). Once the structure of the judge was developed, we could iterate over the initial version and add new features to the system, such as user management, obtaining statistics (stats command) or downloading the next problem assigned to a user (next command). One of the positive points of *ScholarJudge* compared with other online judges is that the solution does not depend on a specific programming language. This is reflected in the CLI, which allows the server to avoid having the software to process the languages of each solution. This section has been fully developed and we have met the expected objectives, leaving open the possibility of adding new commands or functionalities in the future, such as: differentiate between roles, upload a new problem to the system or manage users.

9.2. Web application

The web application was developed in the last stages of the project but, despite this, we were able to allow teachers to manage contests, problems and users. At the same time, students can visualize in a more eye-catching way the information regarding contests, problems and their own statistics.

The only objective that could not be fully addressed was to create a design adaptable to both computers and tablets or mobiles. A responsive design updates the views depending on the size of each of the screens. If we access the application through a mobile device, you can see how the elements of the interface are out of space or overlap with each other. The system could be improved in this line of design, specifically, making use of the characteristics and parameters of the components of Bootstrap (the graphic library used) but due to lack of time, this extension could not be carried out in the project. This line

of future work has been discussed in more detail, outlining a possible implementation, at the end of this chapter.

9.3. Authentication

Initially, and during the development of our system, the authentication system did not exist. The HTTP calls were received and managed by the server without knowing their origin. This was enough as a first approximation to test the validity of the system, although we decided that it would be interesting for any user, whether teacher, administrator or student, could keep track of all their interactions with the system. The main objective was to allow each student to keep a record of their submissions, statistics and problems made, and that teachers and administrators could manage the platform being able to edit problems, add new problems or add contests, among others. This goal was fully developed since any user can log in to our system with their credentials (email and password) and automatically his role and statistics are recognized. However, it was not as simple as expected, because we had difficulties developing the login into the three clients (CLI, interactive CLI and web client) and specifically the user authentication by JWT was one of the biggest challenges of the system. It will be discussed in more detail in the next section.

9.4. Roles

As discussed in the previous subsection, the authentication system allowed us to create a system of roles (administrator, teacher and user) each with unique functionalities. The main objective was to allow teachers and administrators to access to content management functionalities (add problems, create contests, etc.) that students could not have access to. This was fully developed without having to deal with too many difficulties during development and leaving space for future lines of work, for example being able to add new roles such as the role of delegate, director or even giving the possibility of creating groups by classes.

9.5. Challenges

- Check solutions: The check command was one of the most difficult to implement due to its technical complexity, and since it is the main piece of ScholarJudge, it was the one that needed the most attention. The command runs the test cases locally and send the *checksums* of the results to the server to check if the solution is correct or incorrect. Implementing this process was a laborious task since we had to learn to handle files with *Node.js* and especially to execute the solution files and ensure that these were independent of the system installed on the local client.

- Authentication: Finding the optimal way to authenticate to the system, especially using the CLI application, was a complicated task that took us a lot of test and error until we found the solution. It was necessary to incorporate a JWT management system that would allow any user to log into our system without exposing their credentials on their local computer or consuming extra storage in our MongoDB. We had to get familiar with JWT technology and with the *jsonwebtoken* library, which is the one used on the server to generate the login tokens. We also had to decide on

a expiry time for the tokens that was fair and that allowed the user to interact with the system without having to constantly enter your credentials, but also avoiding giving you unlimited access. We came to the conclusion of choosing a duration of 12 hours.

- Migrate and remove outdated libraries: We made the *HTTP* protocol calls from the client to the server with a library called *request*, which currently is not being developed and it is recommended not to use it, as indicated on their own website, so we had to migrate the system to the *node-fetch* library. This, which might seem like an ephemeral change, is one of the risks of using free software and depend on other developers for the proper functioning of your application, since you never know at which point they are going to change your license to use them, stop maintaining them or change the entire repository. This has already happened in other projects of great importance, such as the *faker* package or the *colors* package, in which the developer removed all their code as a reason for protesting that it was being used in an indiscriminate way by large multinationals without even paying to the creator. After this *GitHub*, the company that had the code hosted on its domain, revoked access to the creator himself. This leads us to think: Is the code that we develop our property?

9.6. Future work

ScholarJudge is a complete system, as it offers a wide range of utilities, but still there are possible future work paths that can be developed to complement the current functionalities. The possible points of improvement of our system are:

- Add containers: a container system could be used both on the client and on the server side, using *Docker*. The containers would allow, among others, to run ScholarJudge on any system without having all the necessary dependencies installed, such as a specific version of *Node.js* or Bootstrap. Additionally, Docker offers developers a reliable way to build, ship, and run distributed applications at any scale. To incorporate Docker we should create a file *Dockerfile* in which we have the necessary common configuration and commands of execution. The configuration should contain the following statements: the operating system or base image (usually a space-optimized *GNU/Linux* image with minimum features), the installation commands of a version of *Node.js* and the npm or yarn package managers, the commands to use npm or yarn to install the necessary packages in the previous image and finally the command to start the application. An example of a *Dockerfile* file could be:

```
1 # build environment
2 FROM node:16.14.0-alpine3.14 as build
3 WORKDIR /app
4 ENV PATH /app/node_modules/.bin:$PATH
5 COPY . ./
6 RUN yarn install
7 EXPOSE 3000
8 CMD ["yarn", "start"]
```

- Responsive and dynamic design: Improve the design of the web application so that it can be as responsive as possible to the different devices. The graphical library

used, Bootstrap, allows one to create settings for responsive layouts that could be incorporated into ScholarJudge. Specifically, we would have to define what are known as breakpoints, which are size marks that are used in the CSS to modify their values depending on the size of the screen. An example of configuring breakpoints is shown below:

```
1     // Extra small devices (portrait phones, less than 576px)
2     // No media query since this is the default in Bootstrap
3
4     // Small devices (landscape phones, 576px and up)
5     @media (min-width: 576px) { ... }
6
7     // Medium devices (tablets, 768px and up)
8     @media (min-width: 768px) { ... }
9
10    // Large devices (desktops, 992px and up)
11    @media (min-width: 992px) { ... }
12
13    // Extra large devices (large desktops, 1200px and up)
14    @media (min-width: 1200px) { ... }
```

- Add a domain and upgrade the *hosting*: buy a custom domain to give ScholarJudge and add a customized hosting that fits the requirements of the web. This would incur extra expenses for the development, having to make periodic payments monthly or annually to maintain the domain and the hosting. In our case, ScholarJudge has been developed completely free of charge, so the budget would be a limiting factor for this improvement.

- Creation of tests: During the development of our system we found ourselves in repetitive situations where we had to run the same method with the same parameters over and over again to test its validity. This, apart from being extremely inefficient in terms of time and effort, it can be programmable through tests that allow to simulate the entire situation. This would improve the development experience, allowing to focus your efforts on other aspects of the system. We have created a small part of tests in ScholarJudge using the *chai* library. A concrete example to obtain the access JWT is the following, where it is checked that the corresponding *token* is received correctly for the user with email *test@ucm.es* and password *Test123*.

```
1  before(function (done) {
2      this.timeout(TIME_OUT);
3      chai.request(app)
4          .post("/user/test@ucm.es/token")
5          .send({ password: "Test123" })
6          .end((error, res) => {
7              global.token = "Bearer " + res.body.token;
8              token = "Bearer " + res.body.token;
9              done();
10         });
11 });
```

These tests should be extended to cover the functionality of all the commands that are contemplated in our system, such as: download a problem, view the contests and

check if a solution is correct since it is one of the most tedious processes to check manually.

- Improve continuous integration: this step can continue the previous one, since tests must be part of the process of continuous integration. We could use software like *Jenkins* that checks the code repository for changes and is able to build and deploy the project automatically, executing the tests we have created and giving information to the developers about the result of each of these stages.
- Code documentation: improve the documentation of each of the methods, both server and client, to allow new developers to join the project in a simpler way by indicating the purpose of each inputs and outputs parameters.

Capítulo 10

Contribuciones personales

10.1. Marta Estévez Bravo

Mis principales aportaciones al proyecto están relacionadas con el diseño, la implementación del API REST y las vistas de la aplicación web.

10.1.1. Investigación

En primer lugar tuve que investigar las principales tecnologías del proyecto: Node.js y MongoDB. Aunque tenía conocimientos de JavaScript, Node.js era desconocida para mí y tuve que empezar de cero. También tuve que investigar cómo implementar APIs en Node.js, la apertura de ficheros y cómo codificar una contraseña para guardarla en una base de datos.

Sobre la web, investigué el uso de *Captchas* para evitar el registro provocado por bots y cómo paginar tablas y crear un buscador predictivo para el contenido de una tabla.

10.1.2. Diseño

Uno de los primeros pasos dados en el desarrollo del proyecto una vez ya decididas las tecnologías, fue el diseño de la base de datos y de los primeros *endpoints* del API REST.

Para la base de datos hicimos un análisis preliminar de la información que queríamos guardar y decidimos los modelos con sus correspondientes atributos. Nos centramos principalmente en concursos y problemas, ya que la funcionalidad relacionada con estas entidades del proyecto era lo primero que íbamos a implementar, pero sin olvidarnos de los usuarios. Posteriormente, decidimos los *endpoints*, con sus URLs, entradas y salidas. Empezamos definiendo aquellos que recuperan información de la base de datos, como la lista de concursos o la de problemas. También decidimos cómo íbamos almacenar los ficheros relacionados con problemas (enunciado y casos de prueba). En esta parte del diseño participé a partes iguales con el resto de mis compañeros.

Durante el desarrollo del TFG se han ido haciendo las modificaciones necesarias e incluyendo más *endpoints* y entidades en la base de datos a medida que fueron necesarios para la implementación.

Cuando empezamos con el desarrollo de la aplicación web, la decisión de la estética final de la web la tomamos entre todos. Finalmente, yo diseñé la página de registro, la de edición de datos de los usuarios y la de creación de un nuevo concurso, aunque finalmente se descartó en favor de un *popup*.

10.1.3. Implementación

Al principio de la implementación del proyecto, decidimos dividir responsabilidades y hacer que un miembro del equipo desarrollara el cliente de escritorio, otro integrante el servidor, y el tercer integrante trabajara en ambos componentes según necesidad. Yo me encargué principalmente del servidor, de modo que mi aportación al cliente de escritorio es escasa.

Cliente de escritorio

En mi breve aportación a la implementación de este componente, me encargué de hacer la primera versión para mostrar la lista de los problemas correspondientes a cada concurso. Esto supuso añadir la opción de recuperar la lista de problemas en el menú, llamar al *endpoint* del API REST correspondiente y mostrar la información que devuelve dicho *endpoint*.

10.1.3.1. Servidor y aplicación web

Durante la fase de creación del cliente de escritorio se fueron implementando los *endpoints* REST necesarios. Por mi parte, me encargué del desarrollo del *endpoint* que recupera la lista de problemas de un determinado concurso de la base de datos y de los que devuelven la información de un problema concreto desde el sistema de ficheros del servidor hasta el cliente.

Cuando empezamos a implementar una primera versión de la página web, desarrollé la página de registro. Para esto, aparte de crear la vista, en la que incluí un captcha para evitar registros fraudulentos, hice el *endpoint* de registro desde la web. También me encargué de la desconexión de usuarios, con un enlace en la cabecera de todas las páginas más un *endpoint*.

Una vez teníamos una versión inicial con soporte para usuarios, decidimos que tenían que tener oportunidad de modificar sus datos, como la contraseña, su nombre de usuario o su email. Para esto, hice una nueva página y dos *endpoints*, uno que modificara la contraseña y otro que modificara el nombre de usuario y el email. A la par, propuse introducir roles en el modelo de usuario en la base de datos. Para el administrador, implementé una vista con la lista entera de usuarios dados de alta. Desde esta página, puede modificar el rol y la contraseña de los usuarios y darlos de baja. Para cada una de estas funcionalidades, implementé un nuevo *endpoint*.

También llegué a implementar una página para crear nuevos concursos y un *endpoint* para poder realizar la operación. La página fue descartada pero el *endpoint* no.

En toda la web, colaboré con la gestión de usuarios, haciendo que ciertas pantallas y botones solo los puedan ver profesores o administradores. También hice el control de que, si un estudiante intenta entrar en una página en la que no está autorizado, el sistema lo devuelva a la página de inicio. Finalmente, refactorizamos el API REST, e hice las modificaciones necesarias para que correspondiera con la nueva API.

10.1.4. Pruebas

A lo largo de todo el desarrollo he ido haciendo pruebas, tanto de mi código individualmente, como de forma integrada con el cliente de escritorio.

Para las pruebas individuales de los *endpoints*, especialmente de los problemas y de los usuarios, utilicé la herramienta *Postman*.

10.1.5. Memoria

Decidimos dividir la escritura de la memoria entre los tres. Yo escribí los capítulos "Preliminares", dónde cuento qué es un juez de programación, su importancia en el aprendizaje de la programación y sus limitaciones, y las tecnología utilizadas para el desarrollo del TFG; y "Organización del sistema", donde explico la arquitectura del sistema y los *endpoints* del API. También redacté el resumen, las palabras clave y traduje al inglés parte de la introducción.

10.2. Pablo Morientes Lavín

Mis principales contribuciones han estado relacionadas de manera equitativa tanto con el componente del cliente, como con el componente del servidor, en las cuales he estado trabajando de manera continua a lo largo del proyecto. Como consecuencia, he ido informando a mis compañeros de la situación actual del proyecto en todo su conjunto, consiguiendo así que todos tengamos una visión global y fidedigna del proyecto. En las siguientes subsecciones se explicará en profundidad cada una de las aportaciones realizadas.

10.2.1. Desarrollo del CLI interactivo

Como alternativa al CLI, se optó por destinar parte del tiempo al desarrollo de un CLI interactivo. Esta alternativa esta pensada para aquellos estudiantes que no estén familiarizados con una interfaz de línea de comandos básica. Tras una investigación previa, se optó por usar la librería Inquirer, que nos permitió crear en *Node.js* la interfaz interactiva deseada. La implementación del CLI interactivo fue desarrollado al completo cumpliendo con la misma funcionalidad que el CLI básico.

10.2.2. Envío y corrección de problemas

Esta funcionalidad requirió un análisis amplio, en el que, tanto Enrique como Manuel, estuvieron guiándonos continuamente. Lo primero fue decidir el flujo de ejecución, que tiene presencia tanto en la app del cliente como en el servidor. En segundo lugar se desarrolló la funcionalidad necesaria en el cliente, tanto del CLI interactivo como en el CLI básico. Esta funcionalidad consiste en pedir al usuario los datos necesarios para el envío del problema, ejecutar el fichero de la solución, generar los *checksums* para los distintos lotes de prueba y enviar los *checksums* al servidor. Por último, se desarrolló la funcionalidad que reside en el servidor, la de generar el veredicto. En un primer lugar solo se tuvieron en cuenta dos posibles veredictos: Accepted y Wrong Answer. Más tarde se implementó un nuevo veredicto Accepted Except Spaces, que contemplaba la opción de enviar una solución correcta a excepción de espacios y saltos de línea de más o de menos. Finalmente se crea el envío con su veredicto asociado al usuario.

10.2.3. Autenticación de usuarios en la web

Como tarea previa a la incorporación de roles, lo primero fue crear un proceso de autenticación. Este consistió en diseñar la página web de inicio de sesión y desarrollar el proceso de autenticación mediante el manejo de sesiones.

10.2.4. Diseño de la interfaz web

Al igual que el resto de compañeros, una de las tareas fue el diseño de las páginas que conforman la aplicación web. El desarrollo de las páginas en las que participé fueron la página de *login*, estadísticas de usuario y edición de concursos.

Entre las tareas de diseño de la interfaz web, cabe destacar mi aportación al re-diseño de la interfaz, en la cual se revisaron todas las páginas, que pasaron de tener un diseño simple a uno mucho más atractivo y usable. En las siguientes figuras se muestra cómo era la página web de concursos en sus inicios y una vez rediseñada.

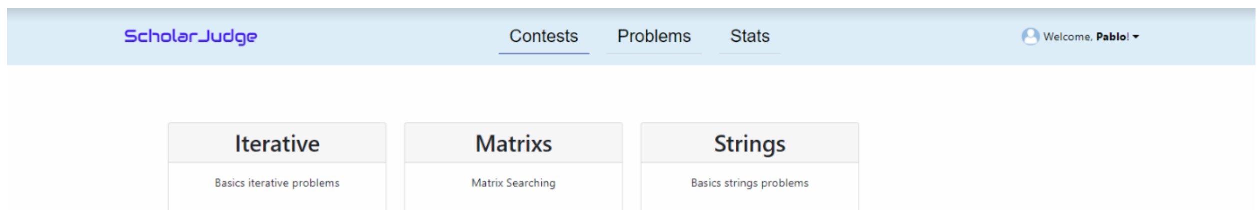


Figura 10.1: Interfaz web: página de concursos

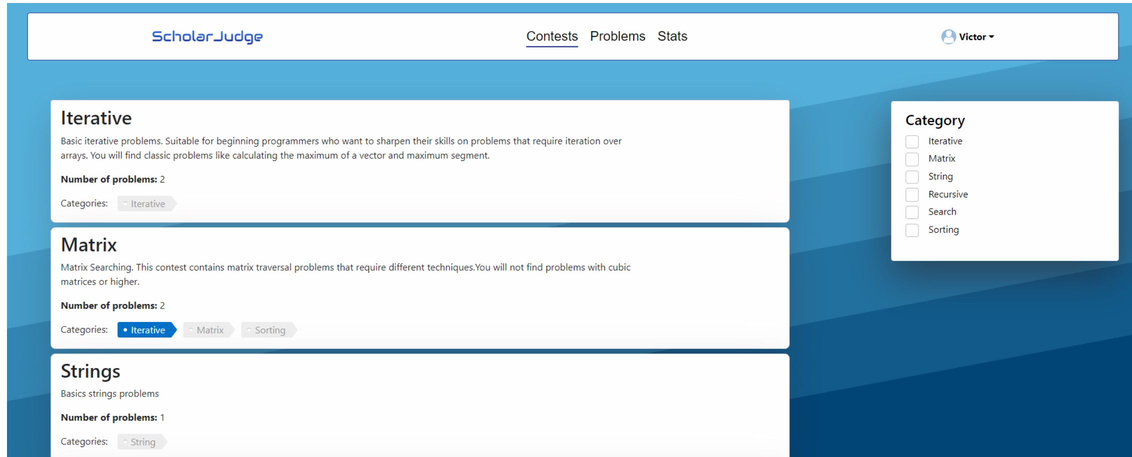


Figura 10.2: Interfaz web: página de concursos rediseñada

10.2.5. Desarrollo de *scripts* de ayuda

Para agilizar algunas tareas, me encargué del desarrollo de diferentes *scripts*. En concreto creé un *script* para la creación de la base de datos. Este *script* inserta varios usuarios con diferentes roles, concursos, problemas y envíos asociados a algunos usuarios. Otro *script* fue un generador de *checksums*, el cual, dado el nombre de un problema y un número de lote de prueba, genera un checksum para la solución correcta y otro para la solución correcta salvo espacios. Por último, creé un *script* que permitía la ejecución de una solución programada en *Java*.

10.2.6. Testing

De la necesidad de comprobar el correcto funcionamiento de algunas funcionalidades susceptibles a cambios, se opto por pruebas de integración. Tras una investigación previa se optó por usar las librerías Mocha y Chai-Http para este cometido. Estas librerías nos permitieron realizar pruebas de integración con llamadas HTTP. No obstante, el alcance de estas pruebas quedó limitada unicamente al testing de la funcionalidad de envío y corrección de problemas.

10.3. Víctor Manuel Cavero Gracia

Mis principales aportaciones han estado relacionadas con el cliente, tanto el CLI como el web. Entre los desafíos encontrados caben destacar: la creación de la interfaz de línea de comandos, las gestión del fichero *scholarJudge.json* o la selección de las librerías de desarrollo necesarias. En la siguientes subsecciones se explica con más profundidad cada una de las aportaciones realizadas.

10.3.1. Creación del proyecto e instalación de los paquetes

Para poder comenzar con el desarrollo de ScholarJudge fue necesario inicializar primero los repositorios de código en GitHub, añadiendo a cada uno de los integrantes y utilizar el comando *npm init* para poder crear la estructura básica de una aplicación de *Node.js*. Este comando crea, entre otros, el fichero *package.json* en el cual se lleva constancia de todas las versiones y paquetes utilizados en el proyecto. Entre estos paquetes utilizados cabe destacar:

- *Prettier*: da formato al código de manera automática y permite así que todos los integrantes del proyecto utilicen un estilo consistente, pudiendo definir los espacios, tamaños de línea máximos o las comillas que se usan por defecto, entre otros (Prettier, 2017).

- *Eslint*: es una herramienta que permite identificar e informar sobre patrones encontrados en el código de JavaScript. Por ejemplo, cuando se introduce una variable que no ha sido utilizada en ninguna parte del código, se genera un aviso (Eslint, 2013).

- *Bootstrap*: librería gráfica seleccionada para crear la aplicación web. Esta ofrece componentes básicos como botones o barras de navegación, entre otros, que permitieron agilizar el desarrollo del cliente web.

- *Jsonwebtoken*: se utiliza para la generación y gestión de los JWT. Esta gestión se describe con más detalle a lo largo de este capítulo.

- *Node-sass*: necesario para generar los estilos personalizados de los componentes de Bootstrap.

- *Unzipper*: permite gestionar los archivos zip para que se puedan añadir nuevos problemas al sistema. Esto se comenta con más detalle a lo largo de este capítulo (Bootstrap, 2016).

- *Nodemon* y *livereload*: permiten reiniciar la aplicación de *Node.js* cuando se producen cambios en alguno de los ficheros de código. Antes de su incorporación, cuando

se modificaba cualquier archivo ya sea código CSS o JS, era necesario reiniciar manualmente el servidor para poder ver los cambios. Sobre todo era problemático para el CSS, donde es más común realizar cambios constantemente hasta encontrar el estilo que más se adapte a nuestra necesidades. Para evitar reiniciar manualmente el servidor, en el archivo *package.json* antes mencionado, se pueden definir, a parte de las librerías utilizadas, comandos nuevos, por lo que se añadió el comando *watch* que hace uso de nodemon (Nodemon, 2011) y livereload (Livereload, 2014) para hacer seguimiento de los ficheros de código y ante cualquier cambio reiniciar el servidor y refrescar el navegador automáticamente.

10.3.2. Construcción y definición del CLI

Una vez definida la API con el resto del integrantes del proyecto solo fue necesario construir la interfaz de línea de comandos. Para ello se ha utilizado la librería *yargs* que gracias a la utilidades que te ofrece te permite crear los comandos utilizados como son: check, stats, problem, etcétera. La librería Yargs consta de una documentación bastante extensa por lo que llevó un periodo de aprendizaje elevado poder llegar utilizarla de la forma correcta, no obstante, la implementación de los comandos fue desarrollada al completo.

10.3.3. Gestión de las versiones de los paquetes utilizados

Como se ha comentado en capítulos anteriores, las llamadas del protocolo HTTP al servidor se realizaban con la librería *request* que debido a su desuso, tuvimos que realizar una migración a la librería *node-fetch*. Por lo tanto, queda demostrado que es necesario ir revisando de forma periódica las librerías que componen el proyecto para validar si siguen siendo soportadas y que no han cambiado tanto en lo relativo a sus licencias como a los sistemas soportados.

10.3.4. El fichero *scholarJudge.json*

Decidir qué información guardar en el fichero común al CLI normal y al interactivo (*scholarJudge.json*) fue una decisión que hubo que meditar bastante, y fue recibiendo cambios durante todas las etapas del desarrollo hasta encontrar las configuración correcta, la cual se muestra en el capítulo 5. Se almacena toda la información relacionada con la carpeta del concurso, desde el *hash* identificador de este concurso, pasando por el JWT que identifica al usuario, hasta incluso la lista de los problemas que ya se han descargado en el sistema local.

10.3.5. Gestión de JWT

Cabe destacar la mejora en el sistema al incorporar la gestión del JWT. Sobre todo, en el fichero *scholarJudge.json* mencionado en la subsección anterior, donde en un primer momento se guardaba el email y contraseña de forma directa, generando grandes problemas de seguridad. Para implementar esta mejora hubo que crear el *endpoint* necesario en el servidor que permite obtener un JWT con validez de doce horas mediante un email y contraseña dados en el cuerpo de la llamada. También se incorporó un *middleware* en el servidor para comprobar que las llamadas del cliente CLI contasen con el JWT en sus cabeceras.

10.3.6. Aportación al cliente web

Se realizó una selección de la paleta de colores mediante la plataforma Huemint[1], que permite generar de forma automática colores y verlos de forma práctica en cada uno de los componentes de Bootstrap. La paleta utilizada se puede observar en la figura 10.3.
También participé en el desarrollo cliente web, creando componentes como:

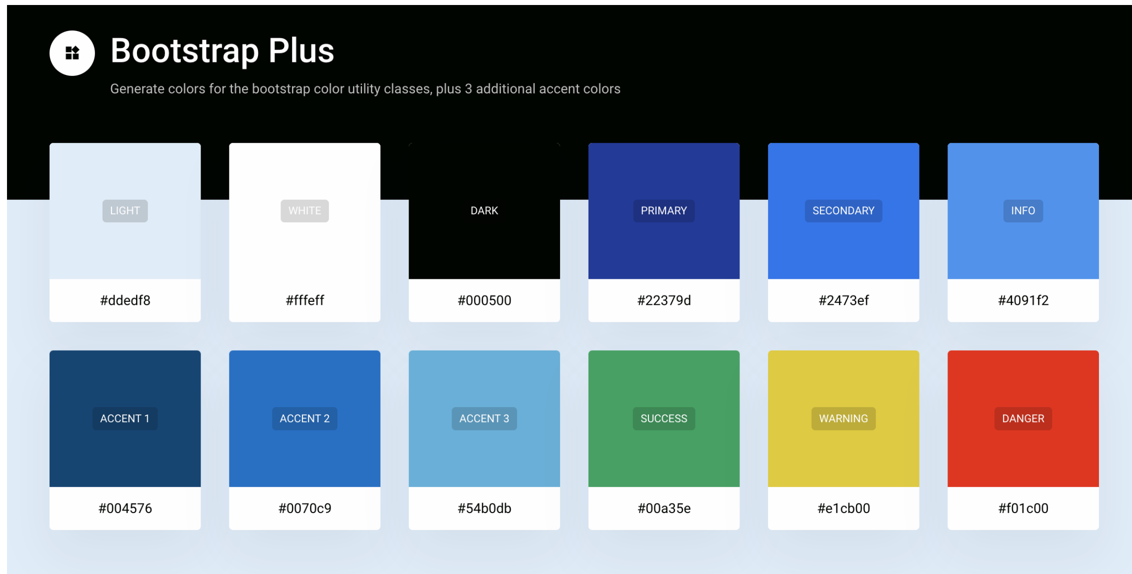


Figura 10.3: Paleta de colores de ScholarJudge generada mediante Huemint

- Barra de navegación: para cambiar entre las distintas vistas de la página web. El mayor desafío afrontado fue que dependiendo del rol del usuario se pudiesen acceder a unas rutas u otras. Por ejemplo, el administrador puede acceder a lista de usuarios registrados pero el usuario normal no.
- Vista concursos: mostrar la lista de concursos presentes en el sistema. Además, dependiendo del rol del usuario se puede acceder a añadir nuevos concursos o editar los concursos actuales.
- Vista problemas: mostrar la lista de problemas indicando cuales ha resuelto el usuario actual. Además, dependiendo del rol del usuario se puede acceder a añadir nuevos problemas mediante un zip o editar los problemas actuales.

10.3.7. Añadir nuevos problemas

Permitir a los administradores o profesores añadir nuevos problemas al sistema mediante ficheros zip. *Node.js* no soporta por defecto la gestión de ficheros comprimidos por lo que fue necesario familiarizarse con la librería *Unzipper* para poder hacer uso de los mismos. También tuvimos que comprobar toda la estructura del archivo que se sube al sistema es la correcta y notificar al usuario con mensajes de error en el caso de que no lo sea.

[1] `https://huemint.com/bootstrap-plus/`

Bibliografía

AUTH0. Documentación de jwt. https://jwt.io/, 2010a. Acceso: 12-02-2022.

AUTH0. Documentación de la librería de node.js: Node-jsonwebtoken. https://github.com/auth0/node-jsonwebtoken, 2010b. Acceso: 12-02-2022.

BOOTSTRAP. Documentación de bootstrap. https://getbootstrap.com/docs/5.2/getting-started/introduction/, 2011. Acceso: 04-03-2022.

BOOTSTRAP. Documentación de unzipper. https://github.com/ndeet/unzipper, 2016. Acceso: 20-03-2022.

BOUDRIAS, S. Documentación de la librería de node.js: Inquirer. https://github.com/SBoudrias/Inquirer.js#documentation, 2013. Acceso: 20-11-2021.

ESLINT. Documentación de eslint. https://eslint.org/, 2013. Acceso: 12-09-2021.

EXPRESS. Documentación de express. https://expressjs.com/es/4x/api.html, 2010a. Acceso: 29-11-2021.

EXPRESS. Documentación de redireccionamiento express. https://expressjs.com/es/guide/routing.html, 2010b. Acceso: 29-11-2021.

KRAUSE, J. *Programming web applications with Node, Express and Pug*. Apress, 2017.

LIVERELOAD. Documentación de livereload. https://github.com/napcs/node-livereload, 2014. Acceso: 03-10-2021.

MONGOOSE. Documentación de mongoose. creación de esquemas. https://mongoosejs.com/docs/guide.html, 2011a. Acceso: 01-10-2021.

MONGOOSE. Documentación de mongoose. sesiones. https://www.npmjs.com/package/connect-mongo, 2011b. Acceso: 15-09-2021.

NANDAA, A. *Beginning API Development with Node.js : Build Highly Scalable, Developer-Friendly APIs for the Modern Web with JavaScript and Node.js*. Birmingham: Packt Publishing Ltd, 2018.

NODEMON. Documentación de nodemon. https://nodemon.io/, 2011. Acceso: 03-10-2021.

POSTMAN. Documentación de postman. https://www.postman.com/product/what-is-postman/, 2014. Acceso: 20-09-2021.

PRETTIER. Documentación de prettier. https://prettier.io/, 2017. Acceso: 12-09-2021.

SASS. Documentación de sass. https://sass-lang.com/documentation, 2006. Acceso: 20-02-2022.

SASS. Documentación de la librería de node.js: Node-sass. https://github.com/sass/node-sass, 2012. Acceso: 20-02-2022.

YARGS. Documentación de yargs. https://yargs.js.org/docs/, 2013. Acceso: 24-09-2021.