Escuela Técnica Superior de Ingeniería Informática

GRADO EN INGENIERÍA INFORMÁTICA

CURSO ACADÉMICO 2020/2021

TRABAJO DE FIN DE GRADO

DESAROLLO DE UNA PLATAFORMA DE ANÁLISIS AUTOMÁTICO DE CÓDIGO

Autor: Pablo López Parrilla
Tutores: Jesús Sánchez-Oro Calvo
Isaac Lozano Osorio

Agradecimientos

Agradecer a la universidad por esta etapa de mi vida, a mis compañeros de grado, a mis profesores, a mis compañeros de trabajo del departamento de informática de Alcorcón y finalmente a los tutores que han hecho posible este TFG. Agradecer a mi familia, a mis amigos y a todas las personas que han estado para aguantar todas mis tonterías.

Siglas

BBDD. Bases de Datos.

API. Interfaz de programación de aplicaciones o en inglés *Application Programming Interfaces*.

IT. Tecnología de la información o en inglés *Information Technology*.

TFG. Trabajo de Fin de Grado.

HW. Componentes físicos de un sistema informático o en inglés *Hardware*.

SW. Componente lógico de un sistema informático o en inglés *Software*.

PDF. Fichero de texto portable o en inglés *Portable Document Format*.

IDE. Entorno de desarrollo integrado o en inglés.

URJC. Universidad Rey Juan Carlos.

SO. Sistema Operativo.

HTTP. Protocolo de transferencia de hipertexto o en inglés *Hypertext Transfer Protocol*.

RAM. Memoria de acceso aleatorio o en inglés *Random Access Memory*.

Resumen

En búsqueda de la modernización de la educación, un poco forzados por la situación epidemiológica actual y tendencia a la "tele-vida", se ha necesitado crear nuevas técnicas que evalúen de forma digital los conocimientos de los alumnos.

Centrándonos en el sector de la informática, las pruebas relacionadas con programación son recurrentes. Habitualmente, se realizaban sobre un folio en blanco, en el cual se construía una solución a un problema que debía ser corregida manualmente por un profesor. Este proceso es lento debido a múltiples causas: caligrafía del estudiante, dificultad del ejercicio, desarrollo del código del estudiante, etc.

En este TFG hemos creado un sistema informático que almacena problemas a modo de repositorio, los cuales pueden ser solucionados por los alumnos evaluándose la solución propuesta de manera automática, pudiendo obtener los resultados en cuestión de segundos y quedando todo ello registrado en una BBDD. Esto permitirá tener control absoluto al profesor para realizar pruebas, pudiendo obtener resultados en tiempo real, además de dar la posibilidad al alumno de comprobar si su código es correcto.

Para la realización del sistema se ha utilizado un esquema de servicios y módulos, separando la parte dedicada a la ejecución de código de la parte dedicada al repositorio de los mismos.

- Para la parte del ejecutor de códigos se han utilizado tecnologías de virtualización mediante contenedores, específicamente Docker.
- Para la parte del repositorio se utilizan diferentes tecnologías de programación, utilizando ampliamente Spring, BBDD relacionales y comunicación por paso de mensajes mediante RabbitMQ.

En esta memoria se hará un repaso de los diferentes jueces de programación ya existentes, desde el punto de vista de la aplicación a la educación. Se mencionarán los objetivos que se quieren conseguir, y las funcionalidades del sistema. Se realizaran diferentes explicaciones sobre el desarrollo del sistema, así como la arquitectura de la misma y metodología utilizada, poniendo ejemplos mediante el uso de diagramas de flujo y pseudocódigo.Además contamos con los métodos disponibles para interactuar con el juez mediante una API.

Finalizará demostrando los resultados obtenidos con el juez y las funcionalidades obtenidas.Además, cuenta con una guía para la instalación del juez.

Palabras clave

Juez automático, Docker, orquestador de contenedores, RabbitMQ, ejecución automática de código, virtualización, contenedores, Spring.

Índice general

Capítulo 1

Introducción

La realización de este proyecto ha consistido en el desarrollo de una plataforma de ejecución y corrección de código automatizada, utilizando diferentes tecnologías y buscando el uso de buenas prácticas de programación para conseguir un producto robusto, escalable y modular.

Este proyecto está dirigido al sector que se ocupa de la programación, particularmente a estudiantes y profesionales de la misma. La utilización del proyecto es viable y eficaz durante concursos de programación, prácticas de clases, procesos de selección de personal en una empresa o exámenes.

En diferentes ocasiones, existe la necesidad de comprobar la correcta ejecución de un código, bien sea durante la corrección de un examen o la selección de personal como hemos dicho antes. En estas situaciones, en el mejor de los casos tendremos un fichero digital que podremos ejecutar manualmente y comprobar el resultado. Sin embargo, si el problema a corregir cuenta con múltiples casos de prueba, la ejecución demorará una gran cantidad de tiempo. Si además el usuario realiza varias soluciones a un mismo problema, el número de ejecuciones manuales se multiplicará dificultando la corrección de un problema.

Este proyecto explotará la necesidad de conseguir una plataforma en la que guardar y mostrar los diferentes problemas, así como la relación entre los participantes y los problemas, además de la corrección automática de los diferentes códigos propuestos por los usuarios.

En este capítulo, definiremos brevemente que es un juez de programación y explicaremos la estructura que va a seguir esta memoria.

Un juez de programación es un sistema informático encargado de validar las soluciones propuestas por usuarios para los problemas disponibles [1]. El sistema debe de contener una serie de problemas que podrán estar organizados de diferentes maneras en secciones, concursos, clases, etc. Cada problema contará con un enunciado, así como de las entradas y salidas que debe obtener de la ejecución del problema, además, podrá configurar la ejecución con una serie de restricciones como pueden ser

tiempo máximo de ejecución o memoria máxima para la ejecución.

Un juez de programación consta de dos partes claramente diferenciadas:

- El sistema encargado de la ejecución de un código en un entorno controlado y seguro.
- El sistema que contiene toda la arquitectura para el guardado de los diferentes problemas, así como de la interfaz y las diferentes funcionalidades del juez.

La primera parte se tratará en la memoria del TFG "Diseño e implementación de una arquitectura basada en contenedores", dejando para este la encargada de la plataforma de análisis automático de código.

El juez utilizara diferentes tecnologías y tendrá varias funcionalidades añadidas al mismo, siguiendo algunos estándares previamente creados como pueden ser el del formato de problema [2]. Además, contará con una API definida para su utilización por interfaces externas u otros sistemas.

Comenzando con una breve introducción en el capítulo 1, este TFG desarrollara y explicará las técnicas utilizadas para la realización del proyecto del juez de ejecución automática de código. Continuaremos con una investigación sobre proyectos similares al que se va a realizar en el capitulo 3, destacando aspectos importantes para el mismo. Seguiremos con el punto de Objetivos 2 en el cual se desarrollan los propósitos que se quieren obtener a la finalización de este proyecto. Pasaremos al punto más extenso, la descripción informática 4. Aquí explicaremos de forma extendida todo el funcionamiento del código, así como la forma en la que se interactuará con el y la documentación pertinente para lo mismo. Expondremos los resultados obtenidos en el aparatado de resultados 5, mostrando diferentes ejecuciones y finalizaremos con las conclusiones en el punto 6. Se incluye finalmente un apéndice que consta de una pequeña guía de instalación A para hacer funcionar el proyecto en cualquier máquina.

Capítulo 2

Objetivos

El objetivo principal de este TFG es desarrollar una plataforma de juez en linea para códigos informáticos, almacenando problemas, organizándolos por concursos y permitiendo interactuar a los usuarios con ellos. El objetivo por lo tanto es el crear un espacio en el que profesores puedan proponer ejercicios a alumnos y estos puedan ser evaluados en tiempo real, devolviendo en segundos los resultados de la ejecución de su código sobre el problema.

Los objetivos secundarios que se quieren cumplir con la realización de este proyecto son:

- Ampliar conocimientos de programación en Java.
- Ampliar conocimientos sobre BBDD y sus relaciones.
- Ampliar conocimientos de Spring y complementos como controladores o servicios.
- Aprender el funcionamiento de RabbitMQ.
- Aprender el desarrollo de microservicios y una arquitectura independiente mediante el paso de mensajes.
- Generar una API que trabajará con la aplicación desde una interfaz gráfica independiente.
- Ampliar conocimientos sobre programación web.
- Aprender el uso de repositorios como GitHub.
- Crear una documentación para la API utilizando Swagger.
- Ampliar conocimientos de redes.

Capítulo 3

Marco teórico y estado del arte

En este capítulo se explicarán las diferentes alternativas que existen en los jueces de programación así como las tareas básicas, funcionalidades y características que deben cumplir.

3.1. Juez de programación

Un juez, como se ha definido antes, es un sistema informático encargado de almacenar diferentes problemas/ejercicios capaz de comprobar, que las propuestas por parte de los usuarios para resolver dicho problemas mediante un código de programación sean correctos. Se compone de las siguientes características:

- Interfaz gráfica: Necesaria para la interacción entre el usuario y el juez de programación. A su vez permitirá añadir problemas y recibir nuevos envíos de códigos de los distintos usuarios.
- Sistema de almacenamiento persistente: Necesitará guardar los resultados de los diferentes usuarios y los intentos de resolver los diferentes ejercicios que los usuarios realicen. Además, tendrá que guardar los distintos problemas, concursos, perfiles de usuario y equipos.
- Servicio de ejecución de código: Un juez necesitará como una de sus partes más importantes y complejas un servicio dedicado a ejecutar código. El código que se ejecute puede contener partes maliciosas, para evitar esto se utilizan distintas técnicas, esto esta tratado en el siguiente punto 3.2. Además el tiempo de ejecución debe de ser consistente, dando valores justos aunque la carga del servidor sea alta.
- Servicio de revisión de la salida del código: El juez tendrá que revisar si la salida obtenida es la esperada, pudiendo marcar el resultado con:

- *accepted*: Si la respuesta es correcta.
- *wrong_answer*: Si la respuesta es incorrecta.
- *time_limit_exceded*: Si se ha superado el tiempo máximo disponible para la ejecución.
- *run_time_exception*: Si se ha producido algún error durante la ejecución.

Estas son las principales características que debe de definir, pero además de ellas se suelen encontrar otras como:

- Sistemas de exportación e importación de problemas. Se trata de una función por la cual se podrá importar o exportar un problema desde un fichero. Para esto existen algunos estándares que se pueden utilizar.
- Sistemas de estructuración de problemas. Característica por la cual se podrán generar concursos o clases en las que existirá una organización de los diferentes problemas, permitiendo, por ejemplo, no poder visualizar uno antes de completar otro.
- Sistemas de clasificación. Sistema por el cual se puntúa los diferentes envíos en base a la dificultad del ejercicio, tiempo que se ha tardado en realizarlo, tiempo de ejecución etc, creando clasificaciones con los distintos usuarios, problemas o concursos.

3.2. Ejecución de código en un entorno seguro

La ejecución de un código es uno de los capítulos más críticos que tiene un juez. El principal problema existente a la hora de la ejecución del mismo es la posibilidad de que el código sea malicioso y pueda crear perjuicios en el entorno donde se ejecute. De tal forma un código podría ejecutar instrucciones que vulneraran el sistema, inspeccionar el sistema de ficheros en búsqueda de otras soluciones o podría incluso bloquear el sistema y dejarlo inutilizado . Debido a esto, existen varias técnicas para proteger al sistema de la ejecución de código. Estas son las siguientes [3]:

- Filtrar por palabras el código: Se trata de leer el código en búsqueda de comandos o patrones que indiquen una posible vulnerabilidad como podría ser que haga una llamadas al sistema o busque un patrón que reserve memoria de forma masiva. Esta técnica, aunque es la más rápida y fácil de implementar, no es efectiva. Esto es debido a que en ocasiones no se pueden encontrar estos códigos o se limita en exceso los comandos que no se pueden utilizar dando excesivas complicaciones a los usuarios que quieran enviar un código. Un ejemplo de un código malicioso que pudiera ser introducido sin ser detectado

seria uno generado en una función dinámica que se obtenga desde un servidor en internet.

- Acotar el entorno donde se ejecuta: Se trata de limitar las capacidades del entorno en el que se va a ejecutar. Esto consiste en controlar cada uno de los puntos en los que un código pudiera generar problemas. Algunos ejemplos de lo que se suele realizar son:

  - Limitar la memoria utilizada. En algunos lenguajes la propia función de llamada al mismo puede contener un límite de memoria a utilizar por el mismo como es Java.
  - Limitar los permisos de utilización del sistema de ficheros. Se puede crear un usuario exclusivo para la ejecución del código el cual esté atrapado en un directorio cerrado y con permisos muy limitados.
  - Limitar el uso de interfaces de red y de salida a internet. Esto puede acarrear en ocasiones problemas si se utilizan librerías no instaladas previamente en el sistema, sin embargo son múltiples las opciones de introducir programas maliciosos utilizando conexiones al exterior del entorno.
  - Limitar el tiempo de ejecución de un código. Utilizando diferentes comandos, se puede llegar a enviar una finalización del proceso que ejecuta un código.

  Es el método más utilizado pero puede tener fallos de seguridad con códigos que bloqueen momentáneamente el sistema reduciendo el rendimiento del mismo y afectando a otros códigos que se estén ejecutando al mismo tiempo.

- Utilizar virtualización: Se trata de la utilización de tecnologías de virtualización para crear entorno en el que el código pueda ejecutar todo lo que quiera dentro de ese sistema controlado. Este entorno se generará nuevo cada vez que se quiera ejecutar un código y posteriormente se deberá eliminar. Es más costoso en términos de recursos necesarios para la ejecución, y además, será más lento por la necesidad de crear dichos entornos. Sin embargo es el sistema más seguro y el más escalable gracias a la facilidad de crear nuevos entornos para la ejecución de nuevos lenguajes. Para la virtualización de estos entornos se podrán utilizar máquinas virtuales o contenedores.

  Esta técnica es la más moderna y requiere de un sistema encargado de organizar o orquestar los diferentes entornos de virtualización. Será la técnica elegida para este proyecto y se desarrollará un servicio independiente en el otro TFG. A este servicio se le llamará orquestador de contenedores y como bien dice el nombre utilizará la tecnología de contenedores para la ejecución de códigos.

3.3. Diferentes jueces de programación

En esta sección, mostraremos los jueces de programación más populares así como sus características. Además se ha realizado una tabla comparativa con algunas de las características que nos resultan más importantes. Se trata de la figura 3.1 en la que podemos observar los distintos jueces evaluando si son gratuitos, si se trata de código libre, si permite organizar competiciones libremente, el número de lenguajes que tiene disponibles para la ejecución y su disponibilidad para utilizarlo como medio educativo en las clases. Destacar que en el caso de Iudex (juez en latín), el juez que aquí desarrollamos, dispone de 2 lenguajes de programación y tiene la funcionalidad de poder ampliar según se requiera en el entorno en el que se va a ejecutar. Esto se realiza de de una manera muy sencilla la cual se explica en el punto A.2.

3.3.1. Kattis

Kattis es un importante sistema de evaluación que consta de varios subsistemas dedicados cada uno especialmente a un fin. Está desarrollado por un amplio equipo de programadores y se encuentra en servicio actualmente [4].

Funcionamiento: El sistema de ejecución del código *online*, primero compila el archivo en busca de fallos en compilación, una vez pasado este punto ejecuta el código con la primera entrada del problema. Se comprobara que la salida es correcta y solamente si es así se ejecutará la siguiente, así hasta que no queden entradas en el problema.

Lenguajes: Kattis soporta 21 lenguajes de programación, entre los que encontramos: C, C#, C++, Java, PHP, Python o Node.js entre otros.

Tecnología de ejecución: Se desconoce el sistema de ejecución de código que utiliza Kattis pues este datos no es público.

Vista del problema: Los problemas muestran una gran cantidad de información. Primeramente se incluye el texto junto a alguna imagen del problema en cuestión, además incluye las características con las que se va a ejecutar dicho problemas tanto en tiempo de ejecución como en memoria. También cuenta con un sistema que le atribuye una dificultad y el autor, fuentes y licencia del problema.

Kattis está divido en 3 productos:

- Para empresas: Esta dedicado a la selección de personal, ofreciendo un sistema automático de evaluación de problemas a la vez que una interfaz para el envió de las invitaciones a los diferentes candidatos. Este sistema es de pago y cuenta con funcionalidades como un sistema de anticopia.

- Para desarrolladores: Se trata de un sistema diseñado para el amplio público. Es gratuito, consta de diferentes desafíos y concursos en los que los usuarios pueden trabajar. Tiene además algunos sistemas de posicionamiento de los

diferentes usuarios en varios *rankings* divididos en países, universidades, general, etc. Los propietarios del juez son los únicos con permisos para crear concursos y problemas.

- Para educación. Dedicado a su uso en la educación, permite crear problemas y corregirlos automáticamente. Proporciona además una interfaz de estadísticas y contiene también un sistema antiplagio. Es un sistema de pago y sus precios se encuentran entre 0 y 35 euros por estudiante y curso.

3.3.2. Codeforces

Codeforces es un importante sistema de evaluación, patrocinado por Telegram [5]. Esta diseñado para realizar concursos de programación competitivas y se mantiene mediante algunos programadores de la universidad ITMO de San Petersburgo [6]. Fue creado por un grupo de estudiantes de la universidad Saratov State University en 2010. Hasta ahora se han realizado más de 650 concursos, manteniéndose como de un servicio gratuito.

Funcionamiento: La ejecución de código en Codeforces se realiza en entornos controlados. Además, Codeforces implementa una API para poder ser utilizado con otras interfaces. Permite además el envío de códigos desde grupos.

Lenguajes: Codeforces soporta 17 lenguajes entre los que se encuentran: JavaScript, Scala, Haskell, Python, Java, C, etc.

Tecnología de ejecución: Utilizan el método de acotar el entorno donde se ejecuta. Anuncia que respetan los tiempos de ejecución de los códigos para obtener un rendimiento justo.

Vista del problema: Los problemas en Codeforces están organizados dentro de concursos. Los problemas están formados por un texto que admite imágenes en el cual se describe la tarea. También cuenta con ejemplos de entrada y salida y con algunas pistas o explicaciones extra.

Codeforces esta trabajando para permitir la creación de concursos privados, de manera libre por los usuarios, pudiendo añadir problemas ya existentes en el sistema, pero no pudiendo subir problemas propios.

3.3.3. Topcoder

Topcoder es un sistema de evaluación de código basado en la venta de talentos a empresas [7]. Esta enfocado a poner en contacto a programadores y empresas mediante la realización de pruebas y problemas propuestos por Topcoder o las empresas. Topcoder organiza concursos mucho más complejos que no solo tratan de resolver un problema, si no de construir una arquitectura. Topcoder nace en 2001 y hasta el día de hoy es el sistema de evaluación de código más utilizado.

Funcionamiento: Topcoder tiene tanto una interfaz web como una interfaz de programa. La ejecución de los diferentes códigos se realiza en entornos controlados con un HW muy específico.

Lenguajes: Topcoder soporta 6 lenguajes, estos son: C++, Java, Python, C# , .NET y VB.NET.

Tecnología de ejecución: Utilizan el método de acotar el entorno donde se ejecuta. Utilizan un HW antiguo muy específico para la ejecución de los códigos individualmente [8].

Vista del problema: Muestra una descripción del problema, y los limites que se van a tener en la ejecución. Puede incluir algunas notas además de ejemplos de entrada y salida. El envío de códigos se puede hacer mediante texto directamente.

Topcoder es una herramienta con grandes posibilidades, pero cuenta con una interfaz muy compleja de entender que implica una gran dificultad para usarlo por un usuario ocasional.

3.3.4. SPOJ

SPOJ es un sistema de juez online preparado para realización de problemas y concursos [9]. Fue creado con le fin de tener una plataforma en la que profesores pudieran proponer ejercicios a sus estudiantes[10]. La pagina además cuenta con un importante enfoque social, teniendo un foro bien estructurado.

Funcionamiento: SPOJ utiliza un servicio llamado Sphere Engine para la ejecución de los códigos. Este a su vez, primero compila el programa, después ejecuta y comprueba el resultado por cada una de las ejecuciones. Devuelve el tiempo de ejecución, uso de memoria y salidas.

Lenguajes: SPOJ cuenta con soporte para más de 60 lenguajes, entre los que se encuentran: Java, Pascal, C, C++, Ruby, Python, etc.

Tecnología de ejecución: La ejecución en el caso de SPOJ se realiza mediante un sistema de acotar el entorno donde se ejecuta, limitando el uso de ciertos comandos. Los códigos se ejecutan en un servidor dedicado.

Vista del problema: Los problemas están formados por el texto del problema además de entradas y salidas de ejemplo.También incluyen los limites con los que se va a ejecutar el problema de tiempo, memoria, tamaño del código y lenguajes permitidos. En cada problema existe una zona de comentarios en los que los usuarios pueden opinar sobre este.

3.3.5. UVa Online Judge

UVa Online Judge es un juez automático desarrollado por la universidad de Valladolid [11, 12]. En 1995 se comenzó el desarrollo del juez por el profesor Miguel Ángel Revilla, siendo 1999 el primer año en el que se realizaba un concurso utilizando

el juez, la plataforma actual sobre la que funciona data de 2005. En la actualidad la universidad de Valladolid no financia el mantenimiento del sistema, este es realizado por una asociación sin animo de lucro. UVa Cuenta con paginas complementarias para buscar problemas o añadir otras funcionalidades al juez. Actualmente no cuenta con nuevo desarrollo. UVa cuenta con una herramienta para poder buscar entre los diferentes problemas almacenados en el sistema, esta se llama uHunt [13].

Funcionamiento: No se conoce el funcionamiento del envío de código al sistema.

Lenguajes: UVa Online Judge soporta 4 lenguajes pero las versiones de los mismos son antiguas, estos son: Java, Pascal, C y C++.

Tecnología de ejecución: Utiliza la tecnología de filtrado del código antes de la ejecución del mismo. Es muy restrictivo con lo que permite para cada lenguaje y es importante leer las especificaciones del lenguaje antes de enviar un código.

Vista del problema: Los problemas están formados por un PDF en el cual esta incluido las entradas y salidas de ejemplo, además, existe un limite de tiempo para la ejecución del problema.

Este sistema, el cual fue uno de los más populares e incluyen múltiples problemas utilizados en libros de programación, se encuentra prácticamente en desuso debido a la cantidad de errores y caídas del servicio que tiene.

3.3.6. DOMjudge

DOMjudge es un sistema de evaluación de código automático de forma local. Es decir, se debe ejecutar el juez en un servidor. Se trata de un proyecto de código libre y gratuito y cuenta con una API definida para ampliar el juez. DOMjudge está enfocado a la realización de concursos.

Funcionamiento: DOMjudge permite la creación de concursos de forma local, además de la configuración de las máquinas en las que se ejecutaran los códigos. La ejecución de los códigos es realizada por el judgehost. Este es un servidor independiente del juez y puede funcionar con múltiples códigos al mismo tiempo.

Lenguajes: DOMjudge admite un total de 8 lenguajes, entre los que se encuentran C, Haskell, Java, Pascal, Prolog, etc.

Tecnología de ejecución: DOMjudge utiliza el sistema de acotar el entorno donde se ejecuta para ejecutar los códigos haciendo uso del comando “chroot”. El script que prepara el entorno se encuentra de forma pública en el Github de DOMjudge [14].

Vista del problema: Los problemas en DOMjudge se muestran para la descarga, igualmente sucede con las entradas y salidas de ejemplo. También cuenta con el limite de tiempo de ejecución y de memoria.

DOMJudge ha tenido numerosas vulnerabilidades debido al método que tiene para ejecutar los códigos. Estas se han ido corrigiendo gracias a la colaboración de los distintos usuarios.

3.3.7. Acepta el reto

¡Acepta el reto! es un sistema de tipo juez de programación creado en el año 2013 por la Universidad Complutense de Madrid. Nace con la necesidad de poder generar un sistema en el que almacenar todos los problemas que utilizaban para los concursos de programación, buscaban una alternativa al juez de programación UVa. El juez es creado por varios profesores y estudiantes, dividiéndose la carga del proyecto entre *frontend* y *backend*.

En la actualidad el juez no permite crear problemas por usuarios exteriores a la organización, y el contenido en materia de problemas es muy limitado. Además, aunque ha sido utilizado en ocasiones para realizar concursos, no es habitual debido la imposibilidad de organizarlos correctamente.

Funcionamiento: Acepta el reto contiene una serie de problemas generados por los administradores y organizados por categorías. Tiene una arquitectura que separa el *frontend* y el *backend* teniendo además *demoneos* encargados de realizar diferentes tareas como la de ejecución de código o actualización de la clasificación [15].

Lenguajes: Tiene un total de 3 lenguajes disponibles con versiones muy antiguas: C, C++ y Java.

Tecnología de ejecución: Utiliza la técnica de filtrado del código antes de la ejecución y la de acotar el entorno, según el lenguaje utilizado filtrará en búsqueda de amenazas potenciales sobre el sistema, más tarde preparará la ejecución en un entorno encapsulado utilizando el comando "chroot", la ejecución se realizará sobre una máquina fija con un procesador Intel 80x86 [16, 15].

Vista del problema: La vista que otorga el juez sobre el problema está formada por un enunciado en los que se encuentran algunas entradas y salidas de ejemplo además de un menú en el que se pueden seleccionar otras opciones como son: enviar una propuesta de solución, ver las estadísticas de envío sobre ese problema tuyas y de otros usuarios, los créditos sobre los creadores del problema e incluso una ayuda para solucionar el mismo.

El problema que tiene este juez es la imposibilidad de preparar concursos o ser estos muy complejos de participar o crear, se utiliza para practicar en programación. La utilización sería mucho mayor si permitieran crear problemas y concursos a gente externa al juez. Este sistema de juez de programación ha sido realizado encadenando numerosos TFGs de distintos estudiantes de la UCM.

3.3.8. Tabla de jueces

Con toda la información recogidas de los diferentes jueces se ha realizado esta tabla con las características más interesantes para este proyecto.

Juez	Gratuito	Código libre	Organizar concursos	Lenguajes	Uso educativo
Open.Kattis	SI	NO	NO	21	NO
Kattis	NO	NO	SI, de pago	21	NO
Universities.Kattis	NO	NO	SI, de pago	21	SI
SPOJ	SI	NO	SI, con permisos	+60	SI
Uva Online Judge	SI	NO	NO	4	NO
Codeforces	SI	NO	Si, con restricciones	17	NO
DOMJudge	SI	SI	SI	8	SI
TOPCoder	SI	NO	SI, de pago	5	NO
Acepta el reto	SI	NO	NO	3	NO
Iudex	SI	SI	SI	2, ampliables	SI

Figura 3.1: Tabla de características de los jueces

Capítulo 4

Descripción Informática

En este capítulo se definirá más técnica y detenidamente el proceso de creación del proyecto, incluyendo metodologías utilizadas, herramientas, arquitectura, diagramas de flujo y diagramas de navegación.

4.1. Metodología utilizada

Durante la realización del proyecto se han seguido los conceptos de la metodología Agile, esto se ha debido a la creación de requisitos de manera dinámica. El juez se ha dividido en dos partes claramente diferenciadas: la parte propia del juez, donde se manejan todas las relaciones entre las diferentes entidades y los usuarios, y la parte de ejecución que es donde se prueban los códigos en entornos controlados. En este TFG nos centraremos en la parte del juez.

Agile es una metodología que existe como concepto desde el año 2001 [17]. Nace de una reunión entre las principales empresas del sector de IT en el estado de Utah en Estados Unidos, en la cual ponía en común las diferentes buenas practicas que tenía cada compañía. Gracias a esto nacía el manifiesto Agile. Mediante el manifiesto Agile, se publicaban procesos de planificación, creación y testing de cada pequeña implementación. En el caso de este TFG, se han utilizado algunas características de la metodología Scrum [18], permitiendo la modificación de los diferentes módulos del proyecto sin necesitar modificar el resto, teniendo pequeñas funcionalidades encapsuladas. Además, se ha utilizado la figura de Scrum Master para revisar la arquitectura y evitar bloqueos. Siguiendo estas técnicas, se realizaron numerosas reuniones con los tutores, tanto físicamente como de forma remota, en las cuales se discutían las distintas funcionalidades que se querían obtener o se añadían nuevas, se repasaban el diseño de la arquitectura, los tutores explicaban nuevas tecnologías aplicables al proyecto o se presentaban los problemas encontrados en el desarrollo del juez.

Centrándose en el desarrollo del juez, se definieron algunos hitos a conseguir.

1. Generar un sistema de clases y entidades para la organización del juez en concursos, problemas y envíos.
2. Conexión del juez con el orquestador encargado de ejecutar los códigos.
3. Revisión del resultado obtenido en los envíos generados por los usuarios.
4. Generar la funcionalidad de importado de problemas desde un archivo ZIP con un formato estándar.
5. Generar una interfaz gráfica para realizar las diferentes pruebas.
6. Generar una API para la conexión desde una interfaz al juez.

De forma particular, desgrane algunos de los hitos en otros más pequeños y añadir otros quedando la lista así:

1. Definir las bases del sistema, necesidades en relación a las diferentes entidades a crear.
2. Realizar las pruebas de conexión entre el juez y el orquestador mediante paso de mensajes por RabbitMQ.
3. Generar una primera estructura básica de un problema con envíos desde archivos de texto y enviar al ejecutor de códigos.
4. Crear un servicio de revisión del resultado obtenido en la revisión de los envíos.
5. Crear un servicio de importado de problemas desde un fichero ZIP con una estructura estándar utilizada por otro jueces y explicada en el apartado 5.
6. Trabajar en las relaciones en BBDD respecto a las diferentes entidades de usuarios, concursos, problemas, envíos...
7. Añadir la funcionalidad de guardar y descargar PDFs en los problemas.
8. Añadir todos los servicios y funciones relacionados con la asignación de problemas a concursos, usuarios a problemas, usuarios a concursos, borrado de objetos, etc.
9. Generar una pequeña interfaz para comprobar el correcto funcionamiento de todos los servicios previamente creados.
10. Crear entidades simplificadas de los objetos previamente definidos para su uso en las interfaces.
11. Generar una API para la conexión desde una interfaz externa al juez.

4.2. Requisitos

Dentro del desarrollo del proyecto se ha utilizado metodologías ágiles, específicamente Scrum. Para ello, se ha decido obtener los requisitos del sistema mediante un sistema de historias de usuario. Con ello se consigue obtener todas las funcionalidades que el sistema debe de contener.

Como definición [19], una historia de usuario es una explicación informal y general de una función de un SW escrita desde el punto de vista del usuario final que sigue el esquema: Como (usuario), quiero (acción a realizar), para (finalidad de la misma). El proyecto cuenta con las siguientes historias de usuario:

- Como estudiante, quiero poder ver los concursos disponibles, para poder ver los problemas que lo conforman.
- Como estudiante, quiero poder ver un problema, para estudiar la forma de resolverlo.
- Como estudiante. quiero poder enviar mi código, para comprobar los resultados.
- Como usuario general, quiero ver los resultados de los intentos de los diferentes usuarios y mios.
- Como profesor, quiero poder crear un problema, para que os estudiantes puedan solucionarlos.
- Como profesor, quiero poder modificar un problema, para cambiar los atributos del mismo.
- Como profesor, quiero poder borrar un problema de un concursos, para que mis estudiantes de una clase no lo solucionen.
- Como profesor, quiero poder crear un problema importándolo desde un archivo estándar, para poder utilizar problemas exportados de otros jueces o creados manualmente.
- Como profesor, quiero poder borrar un problema de todo el sistema, para que ningún estudiante pueda solucionarlo.
- Como profesor, quiero poder crear un concurso, para que poder organizar a los estudiantes en clase.
- Como profesor, quiero poder modificar un concurso, para poder cambiar de nombre o añadir cosas.

- Como profesor quiero poder borrar un concurso, para poder liberar a los estudiantes de esa clase.
- Como profesor, quiero probar la correcta ejecución de mi código, para asegurar que las entradas y salidas son correctas.
- Como profesor, quiero poder organizar los problemas en concursos, para llevar un orden con las distintas clases.
- Como profesor, quiero poder crear un equipo, para que los estudiantes trabajen juntos.
- Como profesor, quiero poder modificar un equipo, para cambiar los integrantes del mismo.
- Como profesor, quiero poder borrar un equipo.

4.3. Herramientas software y tecnologías utilizadas

En esta sección repasaremos brevemente las herramientas utilizadas divididas en secciones.

Los lenguajes utilizados en este proyecto son:

- Java 11: Como base del proyecto y del código.
- HTML: Con Mustache para la realización de la interfaz gráfica de pruebas.

Las tecnologías utilizadas en el proyecto dentro del código son:

- Docker: Como tecnología de virtualización de contenedores.
- Docker-Java: Como librería que nos permite conectar con Docker desde código Java y controlar la vida de los contenedores de una manera más sencilla.
- Spring: Como base del proyecto, es un framework que se compone de herramientas y utilidades de las cuales vamos a hacer uso como pueden ser las conexiones y gestiones de la BBDD además de la generación de una interfaz gráfica y una API.
- RabbitMQ: Es un SW de encolado de mensajes el cual utilizaremos para comunicarnos entre el Sistema de juzgado y el sistema de ejecución de códigos.

- Spring AMQP: Es un protocolo de mensajes de Spring que utilizaremos en conjunción con RabbitMQ.
- Swagger: Es una herramienta que permite interactuar con APIs de forma gráfica, además, genera una pagina interactiva con todos los métodos disponibles en tu API.
- Postman: Es una herramienta que permite interactuar con APIs.
- SLF4J: Es una herramienta que permite realizar operaciones con el log en códigos que usan el lenguaje Java.

El IDE que se ha utilizado para el proyecto es:

- IntelliJ IDEA.

El Sistema sobre el que se ha desarrollado el proyecto ha sido:

- Una máquina virtual con Ubuntu 18.04, virtualizada con VMWare Workstation, con 8 cores y 8Gb de RAM.

4.4. Arquitectura

La arquitectura que se ha utilizado para la realización de este proyecto se apoya sobre los principios de abstracción y encapsulación. El principio de abstracción y encapsulación [20] están motivados por la dificultad de desarrollar sistemas sin dividir e independizar partes y funcionalidades de los mismos.

Gracias al principio de abstracción podemos aislar conceptualmente las diferentes funciones que deberá proporcionar nuestros sistema, dividiéndolas en servicios y módulos. Gracias al principio de encapsulación podremos independizar los distintos módulos del sistema pudiendo trabajar en un módulo sin necesidad de afectar al resto del sistema.

Un servicio es una colección de recursos relacionados entre si que presentan una funcionalidad completa al usuario o a las aplicaciones [20]. Cada servicio estará formado por al menos un módulo. Un módulo es un conjunto de acciones denominadas funciones que comparten un conjunto de datos comunes llamados atributos [20]. Cada módulo contará con una descripción abstracta de la funcionalidad que desarrolla.

Para el desarrollo de la arquitectura del proyecto, se comenzó realizando una definición de servicios de forma muy abstracta, definiendo más detalladamente los módulos que los formarían.

Los servicios y módulos que forman el sistema son:

- Interfaz gráfica: La interacción entre los usuarios y el sistema se realizará mediante una interfaz web. Para ello existen dos módulos encargados de esto:

  - Interfaz de pruebas: Se trata de una interfaz destinada a la realización de pruebas de las funcionalidades desarrolladas en el sistema, además de la realización de demostraciones.
  - API: Se trata del módulo diseñado para la funcionalidad completa del sistema. Está enfocado a la independencia completa de la interfaz respecto al resto del sistema, pudiendo desarrollar múltiples interfaces sin necesidad de conocer el proyecto en profundidad.
  - Generación de POJOs: Es el módulo encargado de la generación objetos tipo POJO, los cuales son objetos simplificados en los cuales se proporciona solamente la información que el sistema considera que el usuario debe conocer.

- Juez: Es el servicio principal del sistema, compone toda la funcionalidad propia del juez menos la ejecución de los códigos. Los módulos que la componen son los siguientes:

  - BBDD: El sistema contará con una BBDD de tipo relacional. Estará formada y diseñada con las tablas y relaciones necesarias para persistir la información de una manera eficiente.
  - Importador de problemas desde archivos ZIP: Es un módulo encargado de generar un problema desde un archivo comprimido. Acepta las distintas entradas y salidas, códigos de prueba para validarlo y varios archivos de configuración de la ejecución de un problema.
  - Funcionalidades de un problema: Es el módulo encargado de todas las tareas relacionadas con la creación, modificación, borrado y enlazado de problemas.
  - Funcionalidades de una *Submission*: Es el módulo encargado de todas las tareas relacionadas con la creación, modificación, borrado y enlazado de un envío o *Submission*.
  - Funcionalidades de un concurso: Es el módulo encargado de todas las tareas relacionadas con la creación, modificación, borrado y enlazado de concursos.
  - Funcionalidades de un equipo: Es el módulo encargado de todas las tareas relacionadas con la creación, modificación, borrado y enlazado de problemas.
  - Funcionalidades de un lenguaje: Es el módulo encargado de la creación y almacenamiento de los distintos lenguajes de programación y su configuración.

  - Envío de un *Result*: Módulo encargado del envío de un *Result* al servicio de ejecución.
  - Recepción de un *Result*: Módulo encargado la recepción de un *Result*.
  - Corrección de una *Submission*: Módulo encargado de la corrección de una *Submission* una vez todos los *Results* han sido ejecutados.
  - Validador de un problema: Es el módulo encargado de validar las entradas y salidas del problema respecto al código del profesor, para asegurar que todos los casos se cumplen y son correctos antes de publicar el problema.

- RabbitMQ: Es el servicio encargado de comunicar el juez con el ejecutor de código y viceversa.

  - Exchanges, colas y mensajes: Se trata de los módulos internos con los que cuenta la herramienta RabbitMQ y que permite configurar la comunicación entre los dos puntos.

- Ejecutor de código: Este servicio obtendrá el resultado de la ejecución del código propuesto para el problema. Se realizará utilizando sistemas de virtualización. En este caso contenedores. Esta formado por:

  - Recepción del *Result*: El *Result* se recibirá como un mensaje, y este módulo se encargará de seleccionar el ejecutor correcto en base al lenguaje seleccionado.
  - Ejecutor: Ejecutará el código propuesto con la entrada seleccionada, obteniendo la salida, códigos de error y otros parámetros. Esto lo realizará creando un contenedor independiente el cual se creará y se borrará dentro del módulo por cada ejecución.
  - Envío del *Result*: Una vez ejecutado el código y obtenido los resultados se volverá a enviar el mensaje de vuelta al juez para su corrección.

En la figura 4.1 se encuentra un esquema simplificado de la arquitectura y diseño del sistema en servicios y módulos. La arquitectura se ha diseñado con el objetivo de conseguir un sistema robusto y de fácil mantenimiento,buscando la escalabilidad del sistema, tanto en la parte del ejecutor de códigos, pudiendo escalar el número de contenedores que puedan ejecutarse paralelamente, como en la propia distribución del juez.

La arquitectura finalmente se ha cumplido parcialmente, pudiendo tener una interfaz gráfica externa al sistema y el sistema de paso de mensajes independiente.

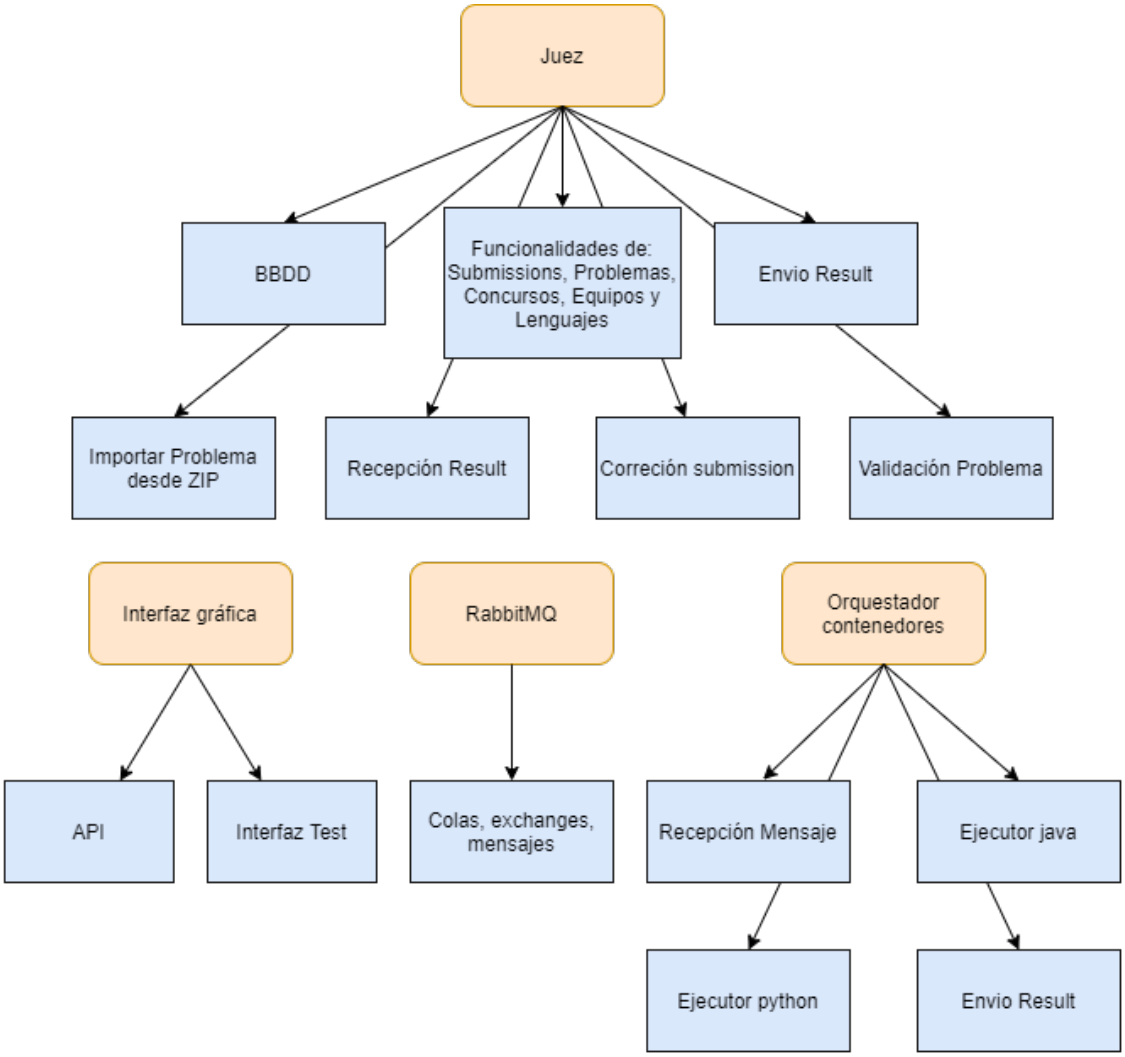

Figura 4.1: Esquema de elaboración propia de arquitectura de módulos y servicios de la aplicación de un *Result*

4.5. API

Con el fin de permitir la utilización de este proyecto por parte del usuario se ha desarrollado una API. Una API es un servicio informático que actúa como intermediario entre dos *softwares* [21], en la mayoría de ocasiones las transmisiones se efectúan desde una aplicación con interfaz o un servidor web con dirección a una aplicación que procese las peticiones como en el ejemplo de la figura 4.2. En el caso del TFG, hemos realizado un juez al cual se le realizarán peticiones desde una interfaz independiente.

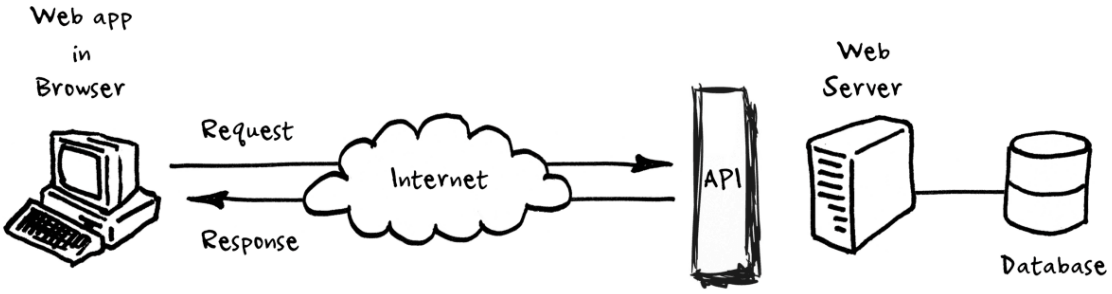

Figura 4.2: Esquema de peticiones de una API [21].

Características:

- Web API: La API que se ha realizado esta diseñada para ser una API de tipo web, la cual a diferencia de otro tipos devuelve respuestas legibles utilizando el protocolo de transferencia de hipertexto o HTTP.

- API Pública: Gracias a la protección de las respuestas del servidor, mediante la limitación de la información que el servidor responde, el sistema permite que cualquier usuario interactúe con el juez de una forma segura.

- Arquitectura REST: Se trata de un protocolo establecido para permitir la comunicación entre diferentes sistemas utilizando estructurar datos. Suele utilizar formatos de tipo XML o JSON los cuales son fácilmente interpretables por ambos puntos de la transmisión.

- Terminología REST: El diseño de la API tiene una definición de verbos muy especificas. Se utilizarán los verbos GET, POST, PUT y DELETE con su significado habitual, obtener, crear, modificar y borrar respectivamente.

- Documentación: Para la documentación de la API y pruebas con ella se ha utilizado Swagger UI, esta aplicación obtiene automáticamente todas las llamadas previamente definidas en el código de los controladores. La documentación permite a los desarrolladores que fácilmente obtengan el conocimiento necesario para trabajar con ella.

De forma resumida, las llamadas disponibles están indicadas en las siguientes tablas 4.3, 4.4, 4.5, 4.6, 4.7:

REST	Path	Definicion	Out
GET	/result/resultId	Obtiene un Result completo	Result

Figura 4.3: Tabla de llamadas a la API del controlador *Administrador*.

REST	Path	Definicion	Out
GET	/contest	Devuelve todos los concursos	Lista<ContestAPI>
GET	/contest/contestId	Devuelve un concurso	ContestAPI
GET	/contest/page	Devuelve un Page de todos los concursos	Page<ConcursoAPI>
POST	/contest	Añade un concurso nuevo	ContestAPI
POST	/contest/contestId/problemId	Añade un problema existente a un concurso	ContestAPI
PUT	/contest/contestId	Actualiza un concurso	ContestAPI
DELETE	/contest/contestId	Borra un concurso	—
DELETE	/contest/contestId/problemId	Borra un problema de dentro de un concurso	—

Figura 4.4: Tabla de llamadas a la API del controlador *Concurso*.

REST	Path	Definicion	Out
GET	/problem	Devuelve todos los problemas	List<ProblemAPI>
GET	/problem/page	Devuelve un Page de todos los problemas	Page<ProblemAPI>
GET	/problem/problemId	Devuelve un problema	ProblemAPI
GET	/problem/problemId/getPDF	Devuelve el pdf del problema	byte[]
POST	/problem	Crea un problema	ProblemAPI
POST	/problem/fromZip	Crea un problema desde un ZIP	ProblemAPI
PUT	/problem/problemId	Actualiza un problema	ProblemAPI
PUT	/problem/problemId/fromZip	Actualiza un problema desde un ZIP	ProblemAPI
DELETE	/problem/problemId	Borra un problema	—-

Figura 4.5: Tabla de llamadas a la API del controlador *Problema*.

REST	Path	Definicion	Out
GET	/submission(?problemId)&(?contes	Devuelve todas las submissions, las de un problema o las de un concurso o las de un problema y concurso	List<SubmissionAPI>
GET	/submission/submissionId	Devuelve una submission	SumbissionAPI
GET	/submission/page	Devuelve un Page de todas las submissions	Page<SubmissionAPI>
POST	/submission	Crea una submission	SubmissionAPI
DELETE	/submission/submissionId	Borrar una submission	—

Figura 4.6: Tabla de llamadas a la API del controlador *Submission*.

REST	Path	Definicion	Out
GET	/team	Devuelve todos los equipos	List<TeamAPI>
GET	/team/teamId	Devuelve un equipo	TeamAPI
POST	/team	Crea un equipo	TeamAPI
POST	/team/teamId/userId	Añade un usuario a un team	TeamAPI
PUT	/team/teamId	Actualiza un team	TeamAPI
DELETE	/team/teamId	Borra un equipo	—
DELETE	/team/teamId/userId	Borra un usuario de un team	—

Figura 4.7: Tabla de llamadas a la API del controlador *Equipo*.

4.5.1. SwaggerUI

Para la realización de la documentación y las pruebas pertinentes, se ha utilizado Swagger-UI. Swagger es un SW de código libre que contiene varias herramientas diseñadas para construir, documentar e interactuar con servicios web de tipo REST [22]. Fue creada en 2011 y desarrollada por la empresa SmartBear Software.

En el caso de este TFG utilizaremos la herramienta Swagger-UI, la cual puede añadirse a un proyecto Java mediante una dependencia. Esta herramienta nos proporciona una interfaz gráfica en un navegador web mediante la cual podremos ejecutar todas las llamadas disponibles de la API. Además, nos proporciona un menú automatizado y documentado de cada una de las llamadas posibles, así como de los objetos que recibiremos mediante las respuestas y posibles códigos HTTP.

El menú principal de la interfaz organizará las llamadas disponibles por controladores. Las pequeñas descripciones que se encuentran en los laterales de cada llamada

es personalizable por el desarrollador. La interfaz del juez seria la de la figura 4.8.

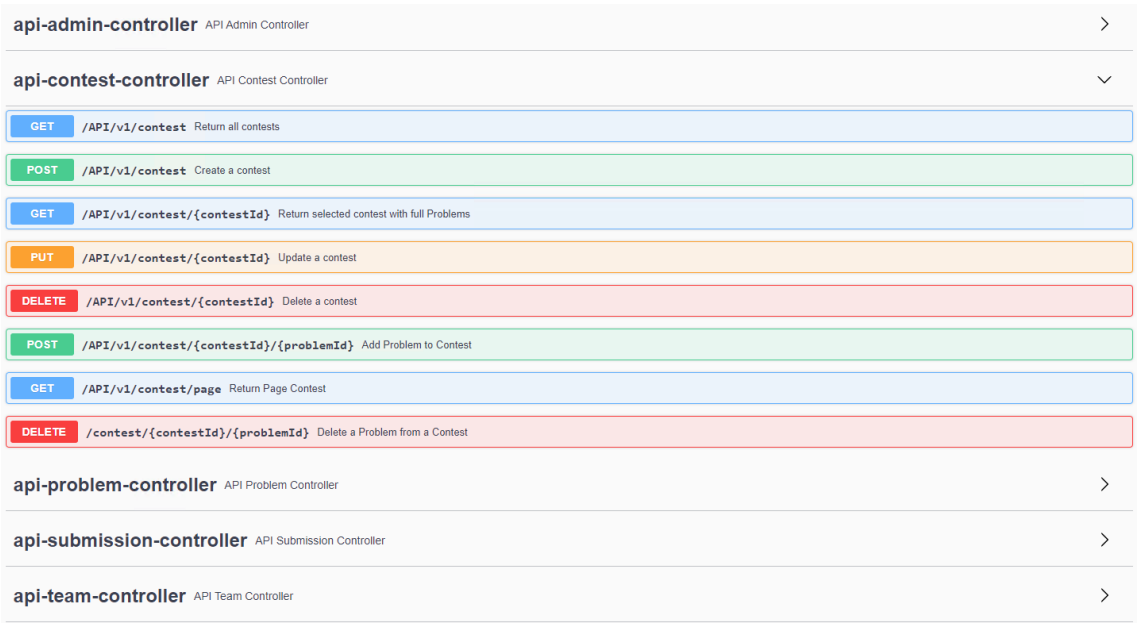

Figura 4.8: Ejemplo de la interfaz de Swagger y métodos disponibles para el controlador de concursos.

En la figura 4.9 se puede observar como es la interfaz de Swagger a la hora de realizar una petición POST al servidor. En este caso todos los campos son requeridos y ninguno opcional, pero si no fuera así se marcaría correctamente.

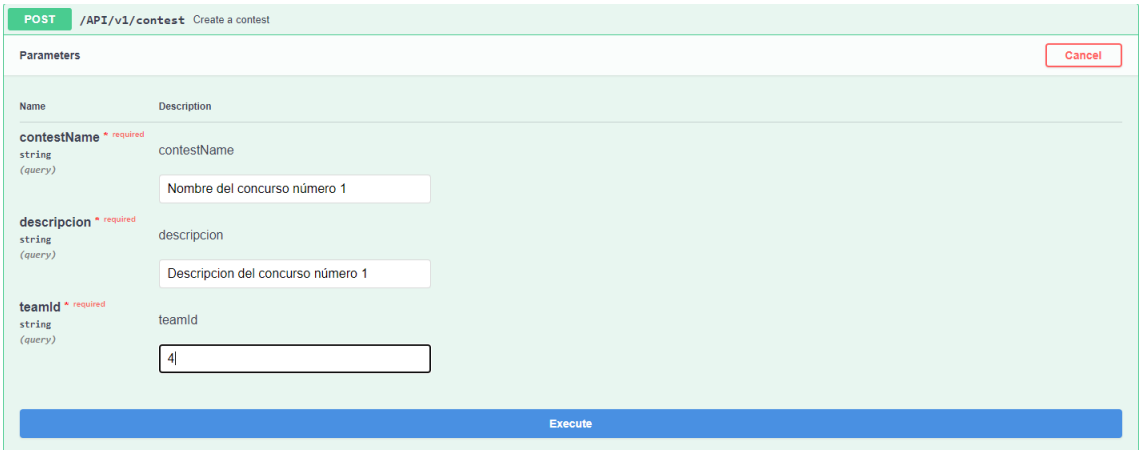

Figura 4.9: Ejemplo de la creación de un concurso utilizando la interfaz de Swagger

En la siguiente figura 4.10 podemos observar la petición que realiza el sistema en

las dos primeras cajas de texto. Además en la siguiente cajas podemos observar la respuesta del servidor. En este caso obtenemos un código HTTP 201 el cual significa creado, también nos devuelve como respuesta el objeto creado en un JSON.

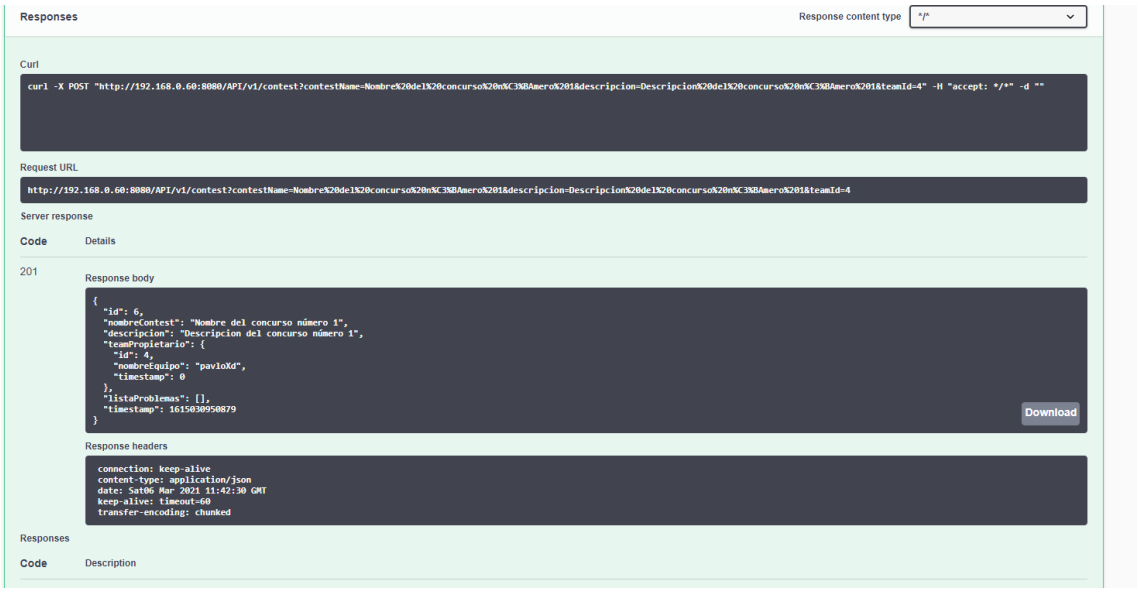

Figura 4.10: Ejemplo de la respuesta a la creación de un concurso utilizando la interfaz de Swagger

En el siguiente texto 4.11 obtenemos un ejemplo de la respuesta que devuelve la API al pedirle que nos envíe todos los problemas guardados en el sistema. En este caso solo hay un problema, el cual tiene una entrada visible y una salida visible, además de dos *Submissions* cada una resumidamente incluye el resultado y equipo que la envío. También incluirá información propia del problema como si este ha sido validado, el *timeout* configurado para el mismo, el límite de memoria y el enlace para poder obtener el documento del problema.

```
[
  {
    "id": 1,
    "nombreEjercicio": "primavera",
    "entradaVisible": [{
        "id": 6,
      }],
    "salidaVisible": [{
        "id": 7,

      }],
    "submissions": [
      {
        "id": 1,
        "team": {
          "id": 4,
          "nombreEquipo": "pavloXd",
        },
        "resultado": "accepted",
      },
      {
        "id": 2,
        "team": {
          "id": 4,
          "nombreEquipo": "pavloXd",
        },
        "resultado": "accepted",
      }
    ],
    "equipoPropietario": {
      "id": 4,
      "nombreEquipo": "pavloXd",
    },
    "valido": true,
    "timeout": "1",
    "memoryLimit": "1000",
    "color": "yellow",
    "problemURLpdf": "/API/v1/problem/1/getPDF"
  }
]
```


Figura 4.11: Ejemplo de respuesta JSON a la petición “GET /problem”.

Al preguntar directamente por la primera *Submission* creada al problema “primavera” obtenemos el siguiente resultado 4.12. En la respuesta encontraremos la lista de *Results* generados, dentro de los cuales podemos observar el resultado del mismo además de otros datos. La *Submission* también contendrá el lenguaje utilizado y el equipo que la ha realizado. Contará con varios parámetros que indicaran en que estado se encuentra la ejecución, el número de *Results* corregido y finalmente los datos globales de los resultados, los cuales serán la suma de todos los *Results*.

```
{
"id": 1,
"results": [
  {
    "id": 9,
    "codigo": "class Isaac {\n    public static void main(String[] args) {\n\n          System.out.printf(\"Hola Abri
    "numeroCasoDePrueba": 0,
    "execTime": 0.06,
    "execMemory": 116640,
    "revisado": true,
    "resultadoRevision": "accepted",
  }
],
"team": {
  "id": 4,
  "nombreEquipo": "pavloXd"
},
"corregido": true,
"numeroResultCorregidos": 1,
"resultado": "accepted",
"language": {
  "id": 1,
  "nombreLenguaje": "java",
  "timestamp": 1615049194379
},
"execSubmissionTime": 0.06,
"execSubmissionMemory": 116640,
"timestamp": 1615049196790
}
```


Figura 4.12: Ejemplo de respuesta JSON a la petición “GET /submission/1”.

4.6. Diagrama de flujo

En esta sección se desarrollarán algunos de los diagramas de flujo o diagramas de actividades de algunas de las tareas que hace el juez. Con esto conseguiremos, de forma esquemática, mostrar el orden de las funciones que se completan con estas tareas.

4.6.1. Diagrama de ejecución y corrección de un envío

Para la ejecución de un código dada una entrada concreta se utiliza un servicio externo al juez. Para ello hacemos uso de un sistema de mensajes concretamente RabbitMQ.

En la figura 4.13 se puede ver el recorrido que sigue un envío para su corrección. El proceso es el siguiente:

1. De un objeto tipo *Submission* se tienen los *Results* correspondientes a la *Submission* creados en el punto 4.6.2. Se recorrerá la lista de *Results* y se envían a la cola de ejecución.

2. La cola de ejecución llama a la función "ejecucion()" de la clase *ResultHandler*. En esta se seleccionara el lenguaje utilizado y lanzara la clase correspondiente para la ejecución con Docker-Java.

3. La clase *DockerContainer* creará un contenedor con los datos correspondientes a la ejecución de la misma, además copiará el código y las entradas a ejecutar. Ejecutará el contenedor y esperará a su finalización obteniendo los resultados correspondientes a la salida estándar, salida error, salida del compilador y señal interna.

4. Una vez obtenido el resultado de la ejecución se envía el objeto *Result* de vuelta mediante RabbitMQ. Se hará utilizando la cola de revisión.

5. La cola revisión lanzará la función "revisar()" de la clase *ResultReviser*. Mediante esta función se revisarán los resultados obtenidos en la ejecución del código. Primero, se mirará si han ocurrido errores en la compilación, después se mirara si existen errores en la ejecución. Si es así se diferenciará entre los errores por *timeout* y los errores en ejecución. Si no ha dado ninguno de estos errores se compararán la salida obtenida con la salida deseada. Para la comparación se eliminarán algunos caracteres especiales. Si las salidas coinciden se marcará como *accepted* en caso contrario como *wrong_answer*.

6. Finalmente se comprobará el numero de *Results* ya corregidos de la *Submission* en búsqueda de ver si se ha completado la ejecución y corrección de una *Submission* entera. Si este es el caso, se recorrerán todos *Results* en búsqueda de ponerte un resultado a la *Submission*.

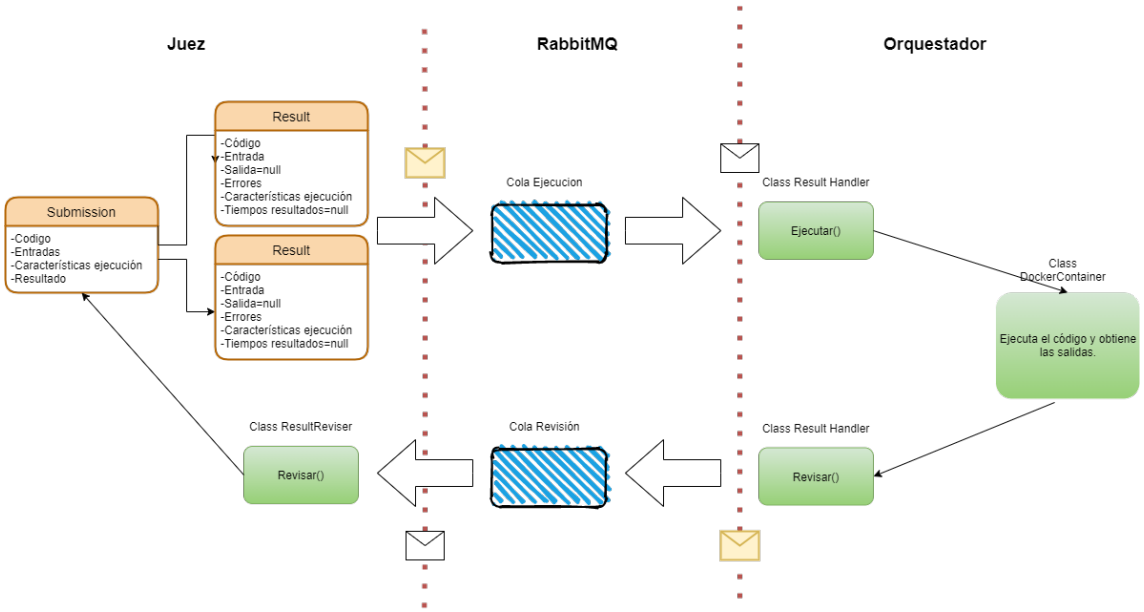

Figura 4.13: Diagrama de elaboración propia de la ejecución de un envío.

4.6.2. Diagrama de creación de un envío

El objeto de tipo *Submission* que hemos creado, es un objeto formado por todas las propiedades del envío realizado por algún usuario del sistema más las relaciones que tiene este con los problemas, equipos, concursos y otros elementos del sistema.

Para la creación de este objeto se parte de la necesidad de tener todas las relaciones de la BBDD comprobadas. Una vez realizado esto, se creará la *Submission* con una lista de objetos *Results* que son el producto obtener una entrada y salida correspondiente junto al código a ejecutar. Además el *Result* contendrá aquellos parámetros de ejecución extraídos del problema como pueden ser el tiempo máximo o memoria para la ejecución. El proceso seguido se esquematiza en la figura 4.14

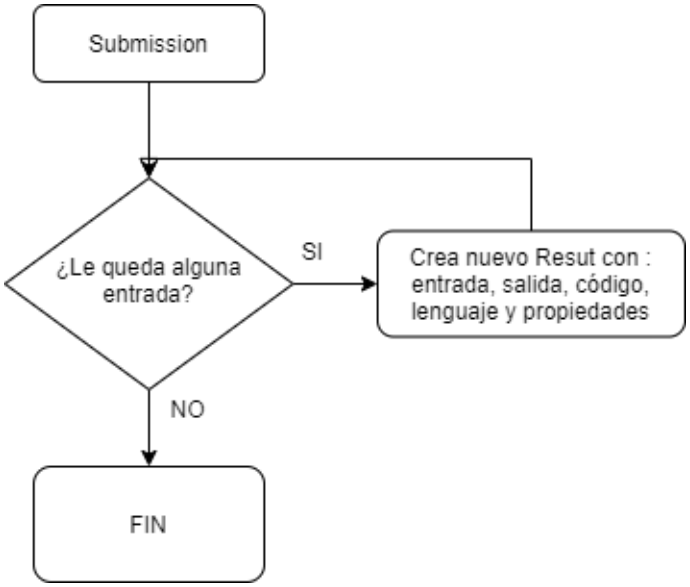

Figura 4.14: Diagrama de elaboración propia de la creación de una *Submission* o envío.

4.6.3. Diagrama de creación y validación de un problema

El proceso de creación de un problema es de las funciones más complejas que existen en el proyecto. Un problema, o como se le ha llamado al objeto en el proyecto *Problem*, debe de contener las relaciones con el equipo que lo publico, así como en el concurso en el que se publicó. Además, contara con las entradas cada una con su salida correspondiente las cuales pueden ser publicas o secretas de tal forma que se muestre o no al usuario. De manera dinámica se irán construyendo las relaciones que existen entre el problema y los concursos a los que pertenezca, o entre el problema y los envíos realizados, o entre el problema y los usuarios que se han apuntado.

La parte más complicada de un problema es su validación. La validación de un problema es la acción por la cual un profesor puede automáticamente comprobar si las salidas y entradas del problema son correctas dado un código a ejecutar. Es decir, se incluirán ciertos envíos con distintos resultados esperados los cuales se ejecutarán con cada entrada y salida del problema. Estos envíos no tiene por que esperar siempre una ejecución correcta, pueden esperar un *timeout* o una respuesta incorrecta.

En la figura 4.15 se puede observar el proceso de creación y el de validación de un problema. El orden es el siguiente:

1. Se crea el problema, se le asignan todas sus propiedades y atributos, así como entradas y salidas, diferenciando en listas según si son entrada o salida y si es pública o secreta. Además, se comprobará que el número de entradas y salidas es el mismo.

2. Si en la creación del problema se le han añadido códigos para validar que los casos de prueba son correctos, se crearán por cada código un objeto de tipo envío o *Submission*. Este seguirá el mismo procedimiento que el punto 4.6.2 con la salvedad de incluir un resultado esperado, que será el desenlace propuesto para ese código (*accepted*, *wrong_answer* o *timeout*.

3. Se procederá a la validación todas las *Submissions*. Esto se realizará recorriendo cada una de ellas y aplicando el punto 4.6.1.

4. Una vez se haya ejecutado todas las *Submissions*, cada vez que la corrección de un envío llegue a su fin, se buscará si todos los envíos han sido corregidos. Solo en el momento en el que todos estén corregidos se marcará el problema como validado o erróneo.

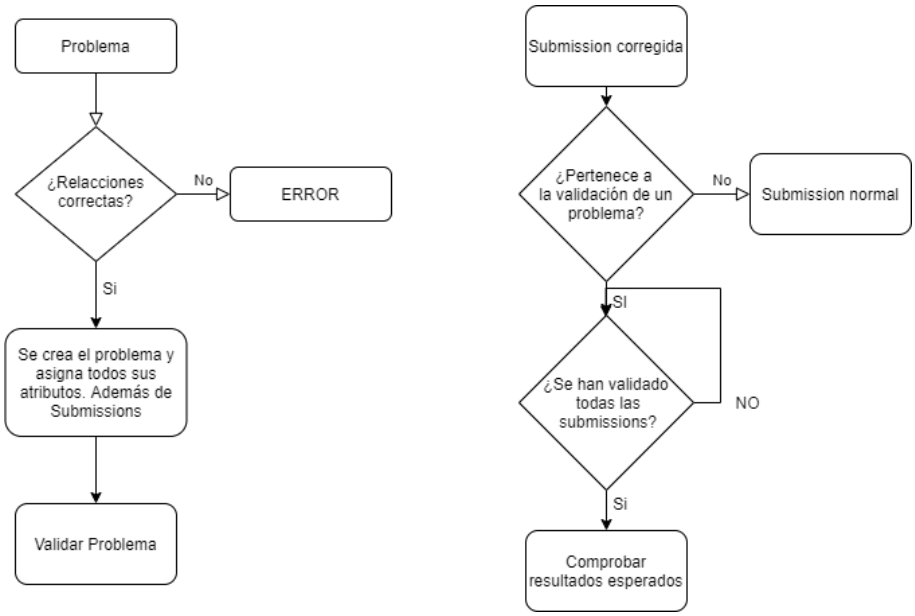

Figura 4.15: Diagrama de elaboración propia de la creación y validación de un problema.

4.7. Decisiones técnicas de diseño

En esta sección hablaremos de forma más detallada de ¿Cómo? y ¿Por qué? se han diseñado algunas funcionalidades específicas.

4.7.1. API y POJOs

En el diseño de la API se ha insistido en simplicidad de todas las llamadas disponibles. Unificando todos los nombres de las variables a utilizar y simplificando la utilización de las llamadas conseguimos una API robusta sobre la que poder crear una interfaz.

API

Para simplificar las llamadas, se ha intentado seguir un patrón por el cual utilizando algunas reglas del lenguaje podemos realizar todas las operaciones relacionadas con la función de la llamada. Las normas utilizadas son las siguientes:

- GET: El método de petición HTTP implicará que lo que se busca es solicitar una información de uno o varios objetos.
- POST: El método de petición HTTP implicara que lo que se busca es crear uno o varios nuevos objetos.
- PUT: El método de petición HTTP implicara que lo que se busca es modificar un objeto ya creado.
- DELETE: El método de petición HTTP implicara que lo que se busca es borrar uno o varios objetos ya creados.
- *Optionals*: El uso de atributos opcionales implicará que si este se encuentra presente será un factor limitante para la función que se quiere realizar.

Un ejemplo de la aplicación de estas normas sería el siguiente:

- "GET /submissions (*Optional=ContestId, ProblemId*)": Al no pasarle ningún parámetro obligatorio en la propia llamada y ser un GET, esperamos obtener una lista. Además tiene como parámetros opcionales el concurso y el problema al que se refieren. Con esto obtenemos el siguiente comportamiento:
  - Si la llamada es vacía, solamente a "/submissions": Obtendremos una lista de todas las *Submissions* almacenadas en el sistema.
  - Si en la llamada se completa el *ContestId*: Obtendremos la lista de todas las *Submissions* realizadas en el concurso.
  - Si en la llamada se completa el *ProblemId*: Obtendremos la lista de todas las *Submissions* realizadas en el problema.
  - Si en la llamada se completa el *ContestId* y el *ProblemId*: Obtendremos la lista de todas las *Submissions* que pertenezcan al problema y al concurso seleccionado.

- "GET /submissions/page": Con el elemento siguiente a la llamada principal "/submissions" sabemos que se busca obtener una opción especifica. Esta estará definida en la siguiente opcion, en este ejemplo busca devolver un objeto de tipo *Pageable*.
- " GET /submission/{submissionId}: Llamada que se le pasa un valor identificativo visible dentro de la misma por el que esperamos la respuesta de un único objeto.
- "POST /submission" Llamada en singular, creará un nuevo objeto único *Submission*.
- "DELETE /submission/{submissionId}: Como en la llamada anterior, conservará el mismo texto cambiando únicamente el método HTTP. Borrará el objeto indicado.

Además el servidor devolverá siempre una respuesta según el resultado de la función. En el caso de que la ejecución sea correcta se devolverá un "200 OK", en el caso de que exista algún error se enviará el código correspondiente al error además de un pequeño mensaje en el cuerpo de la petición. Por ejemplo: "ERROR IN FILE 415(Unsuported Media Type)" en el caso de un error producido por que esperaba un archivo de una extensión y recibe otro; o "SUBMISSION NOT FOUND 404(Not Found)" error que se devuelve cuando se envía una variable con un identificador de *Submission* que no está registrado en el sistema. Al añadir los distintos cuerpos además de errores HTTP permitimos al servicio de interfaz gráfica saber que ha fallado de su petición y realizar las tareas que crea conveniente con ello.

POJOs

Un POJO, en ingles *Plain Old Java Object*, es una instancia de una clase que no extiende ni implementa nada en especial [23]. Es decir, es un objeto simple con pequeñas funcionalidades simples que no depende de un framework.

En nuestro caso, los POJOs se utilizaran como entidades simplificadas para el uso en las respuestas de la API. Se utilizarán por lo tanto como una clase simplificada de las diferentes entidades que tenemos, es decir, una clase POJO de una entidad estándar contará con los atributos necesarios para la visualización por parte de la API, sin incluir otros no deseados. Además, también se utilizan para devolver objetos compuestos con más de un objeto o una salida de error.

Por ejemplo, cuando la interfaz pide a la API una llamada para conseguir un objeto de tipo problema, la API buscará en la BBDD el problema deseado y lo devolverá. Si no controlásemos que atributos devuelve la API podríamos incluir en la respuesta entradas y salidas secretas o información sensible como contraseñas u otros datos.

Por lo tanto, para el uso de POJOs a diferencia de un código normal se ha creado lo siguiente:

- Clases: Las clases generadas han sido diseñadas partiendo de la clase del objeto original, quitando o añadiendo nuevos atributos y retirando relaciones con otros objetos. Estas clases se encuentran en el directorio raíz en la carpeta POJOs.
- Constructores: Las clases POJOs se construyen siempre desde la clase original, pudiendo tener varias funciones en relación a los atributos que queramos que obtenga la clase POJO resultante.
- Llamadas: La creación de los objetos POJOs se realiza siempre partiendo de un objeto o lista de objetos principales, recorriéndolas con un *for* y lanzando en cada uno de estos objetos la función constructor elegida. Esto se ha hecho con el fin de simplificar el código y hacerlo mejor mantenible.

4.7.2. Importado de problemas

El importado de problemas desde archivos comprimidos tipo ZIP fue una de las tareas más complejas de implementar. Se tenía que conseguir descomprimir un archivo desde java; recorrerlo en búsqueda de los diferentes ficheros y configuraciones; asegurar que todos estos eran correctos y su número también lo era; y finalmente proteger lo máximo posible la funcionalidad de este servicio.

Todo el código se encuentra en la clase *ZipHandlerService* y el pseoudocódigo es el siguiente 4.16:

1. Obtenemos el *stream* de datos del fichero comprimido con la clase *ZipInputStream*.
2. Obtenemos el primer documento del *stream* de datos del fichero con la clase **ZipEntry**.
3. Creamos un bucle en el que recorreremos todos los documentos con el objeto **ZipEntry**.
    a) Utilizando la librería de expresiones regulares *Matcher*, podremos filtrar la ruta completa del archivo pudiendo obtener la dirección en la que se encuentra el documento, así como el nombre y extensión que tenga.
    b) Si el documento es una entrada o salida, se organizará según si su carpeta es “sample”, si es una entrada de ejemplo o “secret” si es una entrada secreta. Después obtendremos el texto del documento y lo guardaremos en el objeto problema previamente creado. Además, se escribirá en una estructura de tipo *HashMap* que controlará que todas las entradas tienen una salida con el mismo nombre.

c) Si el documento está dentro de la carpeta *submissions*, se diferenciará entre las carpetas en búsqueda de *accepted, wrong_answer o timeout* que serán las posibles soluciones que puedan generar estas *Submissions*. Con la extensión del documento se buscará el lenguaje al que pertenece dando error si este no existiera en la BBDD. Finalmente se generará una clase derivada de la clase *Submission* llamada *SubmissionProblemValidator* que tendrá el resultado esperado para dicho código añadido a la clase.

d) Se buscará la existencia de un objeto PDF, y se añadirá al problema si este existiera.

e) Finalmente, se buscarán objetos de tipo *.yaml o .ini* los cuales corresponden a archivos de configuración que siguen un estándar. Son de tipo JSON y se tratarán todos los atributos guardándolos en el objeto problema. Si el nombre del problema no estuviese definido por estos archivos se definirá como nombre del problema el que tuviera el archivo comprimido.

4. Una vez acabado de recorrer el documento se continuará con la creación del problema, creando todo lo referente al punto 4.6.3.

5. Se validará que el objeto *HashMap* que se ha utilizado cuando se obtenían las entradas y salidas sea completo, es decir, que para cada clave tenga un valor.

6. Finalmente, se generará un *Hash* del estado del problema en ese momento y se guardará en la BBDD lanzando el proceso de validación 4.6.3.

```
class ZipHandlerService{
    ZipInputStream = getZipStream()
    ZipEntry = ZipInputStream.next()

    while(ZipEntry != null){
        aux = parse(ZipEntry)
        if(aux == "data")
            createEntrada-Salida(aux)
        if(aux == "submission")
            createSubmission(aux)
        if(aux == "configuracion")
            createConfig(aux)
    }
    close(ZipInputStream)
    Problem = createProblem()
    checkProblem(Problem)
    save(Problem)
}
```

Figura 4.16: Pseoudocódigo de la clase ZipHandlerService.

4.7.3. Independencia del servicio de juez y ejecutor. Uso de RabbitMQ

Desde el comienzo del proyecto, se pensó en las ventajas que generaría la separación del servicio de ejecución de código del del juez. Uno de los motivos principales es la escalabilidad. La escalabilidad es la propiedad de un sistema informático de adecuar los recursos utilizados con respecto al rendimiento demandado [24]. En este caso, adecuar el numero de ejecutores a la cola de códigos a ejecutar.

Para conseguir un sistema escalable, se ha buscado independizar el ejecutor de código del juez. Con esto se consigue por una parte, cumplir el esquema de arquitectura de servicios y módulos desarrollado en el punto 4.4: y por otra parte, permitir la escalabilidad de ejecución de problemas.

Para la comunicación entre juez y ejecutor se ha utilizado el SW RabbitMQ, que es un sistema de comunicación mediante mensajes encolados. Con esto obtenemos varias ventajas:

- Gracias a la utilización de mensajes, se independiza completamente la utilización de la BBDD, quedando libre el ejecutor de la necesidad de estar conectado al juez.

- Con la utilización de colas, conseguimos generar una ejecución asíncrona, no teniendo que esperar el juez una respuesta del ejecutor hasta que este termine la ejecución.

- Abrimos la puerta a la paralelización de ejecutores. Gracias a la posibilidad de tener varios consumidores por cada cola, se podría configurar el número de ejecutores que se quiere tener arrancados en función del numero de mensajes encolados, la carga o las capacidades del HW sobre el que corren.

- Independencia física del juez. El ejecutor no necesitará estar ejecutándose en la misma maquina que el juez, ni siquiera en el mismo *host*. Gracias al uso de mensajes, cualquier sistema informático con conectividad mediante red con el juez tendría la posibilidad de convertirse en un ejecutor.

Dentro del esquema de paso de mensajes y especialmente de RabbitMQ contamos con 4 elementos:

- Productor: Es el extremo que genera un mensaje y lo envía al exchange.

- Consumidor: Es el extremo que recibe un mensaje y lo procesa.

- Exchange: Es el servicio que distribuye el mensaje emitido desde el productor a la cola correspondiente, utilizando la configuración indicada.

- Cola: Es el componente que almacena los mensajes de forma ordenada hasta que el consumidor los retira.

Gracias al uso de los diferentes elementos que engloban al paso de mensajes se ha desarrollado un diseño especifico en búsqueda de implementar un servicio escalable. El diseño que se ha pensado cuenta con dos colas:

- Cola de ejecución: Se trata de la cola que guarda los mensajes con los códigos antes de ejecutar. Los mensajes cuentan con la entrada, el lenguaje y el código correspondiente, además de otras opciones de ejecución.

    - Productor: El productor es el juez de programación. Genera un mensaje por cada *Result* que se manda ejecutar.
    - Consumidor: El consumidor es el ejecutor de códigos. Obtiene el mensaje y genera un contenedor con las características y datos adecuados. Debido a que está desconectado del juez, se podría implementar paralelismo en la consumición de las colas. Esto se realizaría mediante la inclusión de múltiples consumidores y el arranque de múltiples instancias del ejecutor.

- Cola de revisión: La cola de revisión es la dedicada a guardar los mensajes de los *Results* después de ejecutar. Además de entrada, lenguaje y código, estos mensajes contienen la salida obtenida por el ejecutor.

    - Productor: El productor en este caso es el ejecutor de códigos. El mensaje se produce una vez el contenedor ha terminado su ejecución y se han obtenido las salidas correspondientes.
    - Consumidor: El consumidor es el juez. Con la salida obtenida, comparará con la salida esperada en búsqueda de igualdad. Además, actualizará la BBDD para contar el resultado obtenido. Este proceso en un principio no está preparado para paralelizar pues se requieren algunos elementos de la BBDD que implicarían bloqueos entre ellos. Sin embargo no es necesario pues la velocidad de consumición es mucho mayor que la de producción.

4.7.4. Uso de Spring: servicios, controladores, repositorios y entidades

Spring Framework es un conjunto de herramientas de código abierto destinada al a la creación de aplicaciones Java [25]. Permite generar distintos tipos de arquitectura gracias a su estructura modular, ayudando y simplificando a los desarrolladores el desarrollo de aplicaciones.

Spring Framework está compuesto por un gran numero de módulos 4.17, entre los que destacamos por su uso dentro de este TFG:

- Core y Beans: Estos módulos son la base del *framework* de Spring. Son los encargados de permitir la inyección de dependencias permitiendo con ello proporcionar modularidad, evitar dependencias entre clases y escalabilidad [26]. Lo utilizaremos con notaciones como "@Autowired" para añadir la dependencias de una clase dentro de otra. Como por ejemplo, añadir una clase de servicios dentro de una clase de un controlador web. Se utilizarán también notaciones como "@Service" para indicar que una objeto se trata de una clase de servicios y la notación "@Repository" para indicar que se trata de una clase de acceso a datos.

- Acceso a datos: JPA es el módulo encargado de facilitarnos el acceso a datos de tipo relacional. Es decir, simplifica las búsquedas de tipo *query* en el esquema de BBDD que tenemos definido, permitiendo con las notaciones "@Repository" poder guardar, buscar o eliminar datos de una forma mucho más simple y evitando errores.

- Web: Es uno de los módulos que más facilitan el desarrollo al programador, utiliza el el patrón Modelo-Vista-Controlador [27]. Con esto conseguimos

desarrollar controladores que se encargarán de responder a peticiones HTTP producidas desde un cliente. También se encargará de toda la conexión con el cliente, pudiendo mantener sesiones e implementar seguridad en ellas. En este proyecto se utilizará ampliamente con notaciones como "@Controller" o "@RestController".

- Mensajes: Utilizaremos RabbitMQ como proveedor de una plataforma de comunicación mediante mensajes, sin embargo, para la utilización de esta plataforma usamos un módulo de Spring llamado Spring AMQP, este proporciona una capa de abstracción facilitando la creación y consumición de mensajes por parte de los extremos de la comunicación así como por la configuración. Este apartado esta ampliamente explicado en el TFG del orquestador o ejecutor.

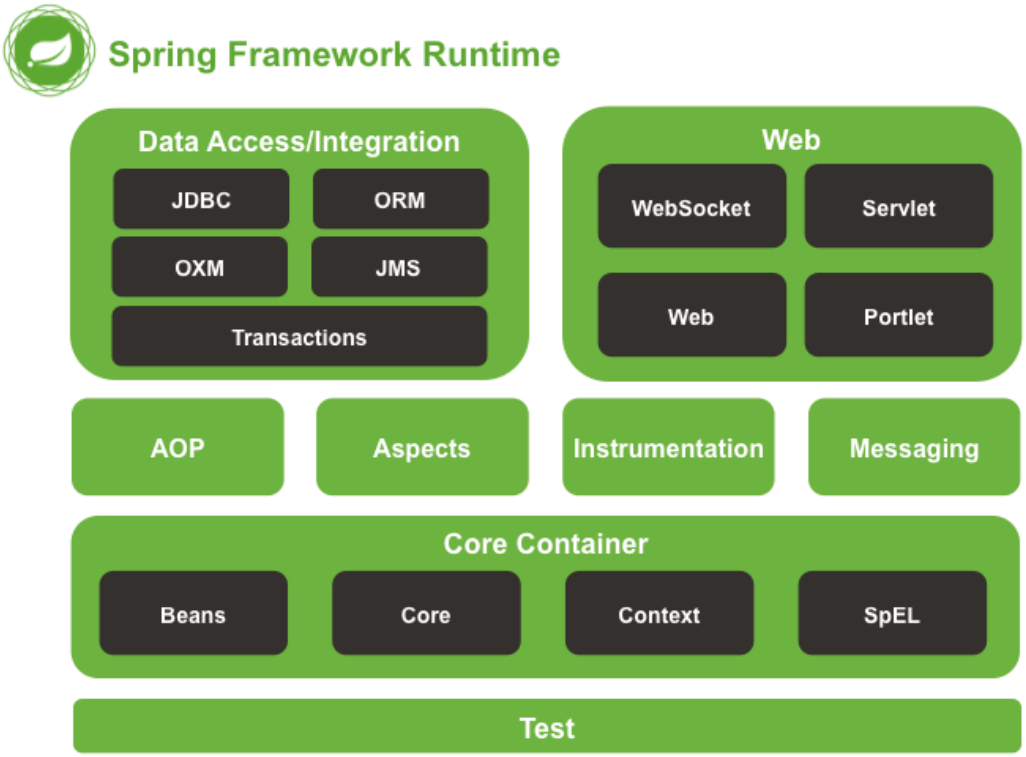

Figura 4.17: Diferentes módulos que comprenden Spring. Obtenido de la pagina principal de documentación de Spring [28]

Capítulo 5

Resultados

A la finalización de esta primera versión del proyecto, se ha conseguido obtener un sistema sencillo de mantener a la vez que completo, de fácil ampliación y mejora, teniendo además una gran escalabilidad a la hora de ejecutar códigos. Se han implementado un gran numero de funcionalidades concretas como son:

- Importar problemas desde archivos comprimidos basados en un estándar definido.
- Comprobar que los casos de prueba de un problema son correctos cuando este es generado.
- Generar y documentar una API para el manejo del sistema.
- Independizar los diferentes servicios como el juez del ejecutor de códigos y comunicarlos mediante el paso de mensajes.
- Generar un esquema de BBDD relacional para la gestión de todos los datos del juez.
- Permitir el tratamiento de archivos PDF en los problemas.

Además de otras más generales como:

- La gestión relacionada con la creación de concursos, el borrado en cascada de los mismos, la modificación de estos, la asignación de problemas a un concurso, el eliminado de problemas de un concurso y la obtención de los diferentes datos de un concurso.
- La gestión relacionada con la creación de un problema, borrado del mismo y modificación utilizando métodos como el paso de un problema en un JSON o utilizando archivos comprimidos.

- La gestión y funcionalidad relacionada con la creación de objetos de tipo concurso, problema, *Submission*, *Result*s y equipos.
- La gestión y funcionalidad relacionada con la modificación de las distintas entidades.
- La gestión y funcionalidad relacionada con el borrado de entidades, implementando y controlando el borrado en cascada para las mismas. Además del enlazado de problemas con concursos.
- La funcionalidad relacionada con el tratamiento de las *Submissions*, la creación de *Results* correspondientes y el envío de estos.
- La funcionalidad relacionada con la corrección de los *Results* corregidos, así como la recepción de los mismos y el control de resultados y errores.

Al haber implementado el estándar en el importado de problemas desde otros jueces conseguimos compatibilizar los problemas ya disponibles en DOMJudge con nuestro sistema. Esta funcionalidad es de gran importancia debido a que el grupo de la URJC que organiza los concursos de programación utiliza el sistema DOMJudge, y al poder importar dichos problemas sobre esta nueva plataforma facilitará la migración de los concursos a Iudex.

5.1. Ejemplos

Con el fin de demostrar el funcionamiento del juez, vamos a utilizar un ejemplo de un problema creado por el grupo de programación competitiva de la URJC.

1. Empezaremos con la estructura del fichero a seguir. El problema deberá de tener una estructura similar a la mostrada en la figura 5.1. Contendrá una carpeta “data” donde se encuentran los ficheros de entrada y salida y una carpeta “submissions” donde se encuentran lo códigos que debe de validar. En ambos casos existen subcarpetas con opciones como si una entrada es pública o secreta o si una *Submission* debe de ser correcta o fallida. Además, el directorio raíz contendrá el PDF del problema así como diferentes archivos de configuración.
2. Al crear el problema tiene 3 *Submissions* en la carpeta *accepted* y 3 *Submissions* en la carpeta *wrong_answer*. Sin embargo, como C++ no es un lenguaje soportado ahora mismo, solo creará 4 *Submissions* en total. El problema tiene en total 7 casos de prueba, por lo que generará un total de 7x4=28 *Results*. Cada *Result* implica generar un contenedor independiente.

3. Los *Results* se envían al ejecutor de código el cual devolverá la respuesta de a los mismos. En la figura 5.2 podemos observar el log generado por el juez y la corrección realizada a cada *Result*. En este caso el problema se valida correctamente, pues las *Submissions* 3 y 4 a las que pertenecen los *Results* con identificador 43 y 51 han fallado siendo este el resultado esperado al estar presente dicho código en la carpeta *wrong_answer*.

4. Observando los resultados obtenidos, la figura 5.3 consiste en un *Result* correcto. En este caso se puede observar como el parámetro *text* del apartado *salidaEstandarCorrectaInO*, que corresponde a los casos de prueba del problema, es el mismo que el parámetro que *salidaEstandar*. En el caso de la figura 5.4 justo al contrario no corresponden dichos parámetros. Además se pueden observar otros datos como los tiempos de ejecución, la memoria utilizada, el momento en el que se ejecutó y el lenguaje utilizado entre otros.

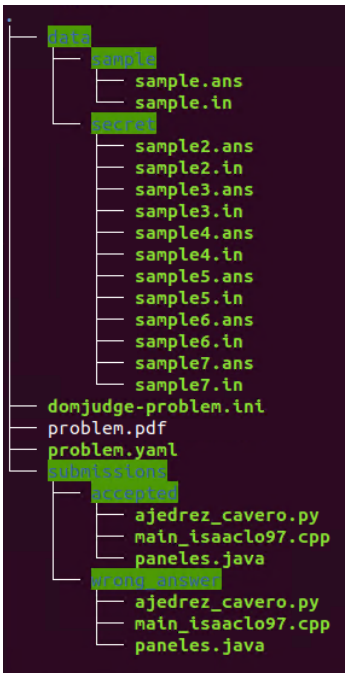

Figura 5.1: Ejemplo de la estructura de un problema.

```
17:30:14.542  INFO 31041 --- [nio-8080-exec-2] c.e.a.services.ProblemValidatorService   :  El result 51 de la submission 4 se manda a ejecutar
17:30:16.805  INFO 31041 --- [ntContainer#0-1] c.e.aplicacion.services.ResultReviser    : Result 21 corregido con resultado accepted
17:30:19.178  INFO 31041 --- [ntContainer#0-1] c.e.aplicacion.services.ResultReviser    : Result 22 corregido con resultado accepted
17:30:21.295  INFO 31041 --- [ntContainer#0-1] c.e.aplicacion.services.ResultReviser    : Result 23 corregido con resultado accepted
17:30:23.387  INFO 31041 --- [ntContainer#0-1] c.e.aplicacion.services.ResultReviser    : Result 24 corregido con resultado accepted
17:30:25.524  INFO 31041 --- [ntContainer#0-1] c.e.aplicacion.services.ResultReviser    : Result 25 corregido con resultado accepted
17:30:27.610  INFO 31041 --- [ntContainer#0-1] c.e.aplicacion.services.ResultReviser    : Result 26 corregido con resultado accepted
17:30:29.699  INFO 31041 --- [ntContainer#0-1] c.e.aplicacion.services.ResultReviser    : Result 27 corregido con resultado accepted
17:30:29.728  INFO 31041 --- [ntContainer#0-1] c.e.a.services.ProblemValidatorService   : COMPROBANDO PROBLEMA: La comprobacion del problema julio se ha comenzado
17:30:30.469  INFO 31041 --- [ntContainer#0-1] c.e.aplicacion.services.ResultReviser    : Result 29 corregido con resultado accepted
17:30:31.217  INFO 31041 --- [ntContainer#0-1] c.e.aplicacion.services.ResultReviser    : Result 30 corregido con resultado accepted
17:30:31.960  INFO 31041 --- [ntContainer#0-1] c.e.aplicacion.services.ResultReviser    : Result 31 corregido con resultado accepted
17:30:32.698  INFO 31041 --- [ntContainer#0-1] c.e.aplicacion.services.ResultReviser    : Result 32 corregido con resultado accepted
17:30:33.422  INFO 31041 --- [ntContainer#0-1] c.e.aplicacion.services.ResultReviser    : Result 33 corregido con resultado accepted
17:30:34.158  INFO 31041 --- [ntContainer#0-1] c.e.aplicacion.services.ResultReviser    : Result 34 corregido con resultado accepted
17:30:34.899  INFO 31041 --- [ntContainer#0-1] c.e.aplicacion.services.ResultReviser    : Result 35 corregido con resultado accepted
17:30:34.916  INFO 31041 --- [ntContainer#0-1] c.e.a.services.ProblemValidatorService   : COMPROBANDO PROBLEMA: La comprobacion del problema julio se ha comenzado
17:30:36.999  INFO 31041 --- [ntContainer#0-1] c.e.aplicacion.services.ResultReviser    : Result 37 corregido con resultado accepted
17:30:39.163  INFO 31041 --- [ntContainer#0-1] c.e.aplicacion.services.ResultReviser    : Result 38 corregido con resultado accepted
17:30:41.237  INFO 31041 --- [ntContainer#0-1] c.e.aplicacion.services.ResultReviser    : Result 39 corregido con resultado accepted
17:30:43.413  INFO 31041 --- [ntContainer#0-1] c.e.aplicacion.services.ResultReviser    : Result 40 corregido con resultado accepted
17:30:45.612  INFO 31041 --- [ntContainer#0-1] c.e.aplicacion.services.ResultReviser    : Result 41 corregido con resultado accepted
17:30:47.879  INFO 31041 --- [ntContainer#0-1] c.e.aplicacion.services.ResultReviser    : Result 42 corregido con resultado accepted
17:30:50.008  INFO 31041 --- [ntContainer#0-1] c.e.aplicacion.services.ResultReviser    : Result 43 corregido con resultado wrong_answer
17:30:50.023  INFO 31041 --- [ntContainer#0-1] c.e.a.services.ProblemValidatorService   : COMPROBANDO PROBLEMA: La comprobacion del problema julio se ha comenzado
17:30:50.746  INFO 31041 --- [ntContainer#0-1] c.e.aplicacion.services.ResultReviser    : Result 45 corregido con resultado accepted
17:30:51.484  INFO 31041 --- [ntContainer#0-1] c.e.aplicacion.services.ResultReviser    : Result 46 corregido con resultado accepted
17:30:52.234  INFO 31041 --- [ntContainer#0-1] c.e.aplicacion.services.ResultReviser    : Result 47 corregido con resultado accepted
17:30:52.967  INFO 31041 --- [ntContainer#0-1] c.e.aplicacion.services.ResultReviser    : Result 48 corregido con resultado accepted
17:30:53.700  INFO 31041 --- [ntContainer#0-1] c.e.aplicacion.services.ResultReviser    : Result 49 corregido con resultado accepted
17:30:54.452  INFO 31041 --- [ntContainer#0-1] c.e.aplicacion.services.ResultReviser    : Result 50 corregido con resultado accepted
17:30:55.192  INFO 31041 --- [ntContainer#0-1] c.e.aplicacion.services.ResultReviser    : Result 51 corregido con resultado wrong_answer
17:30:55.198  INFO 31041 --- [ntContainer#0-1] c.e.a.services.ProblemValidatorService   : COMPROBANDO PROBLEMA: La comprobacion del problema julio se ha comenzado
17:30:55.204  INFO 31041 --- [ntContainer#0-1] c.e.a.services.ProblemValidatorService   : El problema julio ha sido validado
```

Figura 5.2: Ejemplo de la ejecución de un problema.

```
{
  "id": 21,
  "codigo": "import java.util.Scanner;\n\npublic class panales {\n\n    public static void main
    (String[] args) {\n        char a = '#';\n        char m = '+';\n        Scanner sc = new Scanner
    (System.in);\n        int n = sc.nextInt(); sc.nextLine();\n        String line = sc.nextLine
    ();\n        char c = line.charAt(line.length() - 1);\n        final boolean b = c == m && line
    .charAt(line.length() - 2) == m;\n        if (n % 2 == 0) {\n            if (c == a) {\n
           System.out.println(\"0-1\");\n            } else if (b) {\n                System.out
    .println(\"0-1\");\n            } else {\n                System.out.println(\"1/2-1/2\");\n
           }\n        } else {\n            if (c == a) {\n                System.out.println(\"1-0\"
    );\n            } else if (b) {\n                System.out.println(\"1-0\");\n            } else
    {\n                System.out.println(\"1/2-1/2\");\n            }\n        }\n    }\n\n}",
  "entradaInO": {
    "id": 12,
    "name": "sample",
    "timestamp": 1615134614416,
    "text": "5\ne4 e5 Dh5 Re7 D*e5#"
  },
  "salidaEstandarCorrectaInO": {
    "id": 19,
    "name": "sample",
    "timestamp": 1615134614416,
    "text": "1-0"
  },
  "salidaEstandar": "1-0\n",
  "salidaError": "",
  "salidaCompilador": "",
  "timestamp": 1615134614416,
  "numeroCasoDePrueba": 0,
  "salidaTime": "0.10 130000\n",
  "signalCompilador": "0\n",
  "signalEjecutor": "0\n",
  "execTime": 0.1,
  "execMemory": 130000,
  "revisado": true,
  "resultadoRevision": "accepted",
  "language": {
    "id": 1,
    "nombreLenguaje": "java",
    "imgenId": "eb85ca7e7c0e",
```


Figura 5.3: Ejemplo *Result* correcto

```
{
  "id": 51,
  "codigo": "n = int(input())\nline = input().strip()\n\nif n % 2 == 0:\n    if line.endswith(\"#\")
    or line.endswith(\"+\"):\n        print(\"0-1\")\n    else:\n        print(\"1/2-1/2\")\nelse:\n
       if line.endswith(\"#\") or line.endswith(\"+\"):\n        print(\"1-0\")\n    else:\n
    print(\"1/2-1/2\")",
  "entradaIn0": {
    "id": 11,
    "name": "sample7",
    "timestamp": 1615134614416,
    "text": "1\ne4+"
  },
  "salidaEstandarCorrectaIn0": {
    "id": 18,
    "name": "sample7",
    "timestamp": 1615134614416,
    "text": "1/2-1/2"
  },
  "salidaEstandar": "1-0\n",
  "salidaError": "",
  "salidaCompilador": "",
  "timestamp": 1615134614416,
  "numeroCasoDePrueba": 6,
  "salidaTime": "0.04 28752\n",
  "signalCompilador": "0",
  "signalEjecutor": "0\n",
  "execTime": 0.04,
  "execMemory": 28752,
  "revisado": true,
  "resultadoRevision": "wrong_answer",
  "language": {
    "id": 2,
    "nombreLenguaje": "python3",
    "imgenId": "678410a5e78b",
```


Figura 5.4: Ejemplo *Result* no correcto

5.2. Tiempos de ejecución

Uno de los principales objetivos en el proyecto era mantener un buen rendimiento a la hora de corregir problemas y envíos a la vez que se mantenía un sistema seguro. La principal forma de medir el rendimiento de los jueces de programación es comprobar cuanto tiempo tardar en validar un problema y corregir una *submission*.

En este caso vamos a comparar DOMJudge con el juez propuesto, utilizando un solo core para la ejecución de ambos sistemas. DOMJudge tiene la ventaja de estar ejecutándose en un entorno más potente, instalado en un servidor sobre un hipervisor de tipo 1, a diferencia, Iudex esta instalado en un ordenador personal sobre un hipervisor de tipo 2.

Las pruebas se van a realizar utilizando el problema “Julio” el cual está explicado en el apartado anterior.

- Tiempo de validación de un problema:

- Iudex: 96 segundos. Tiempo calculado utilizando el log desde que el sistema recibe el problema en un archivo comprimido hasta que todas las submissions están ejecutadas. Este log se puede ver en la figura 5.5.
- DOMJudge: 105 segundos. Este tiempo tiempo se ha calculado de forma teórica debido a no tener acceso al log de DOMJudge. Es la suma de la ejecución de todas las *submissions* y 2 segundos teóricos que tarde en recibir el archivo, descomprimirlo y crear el problema. En la figura 5.7 se observa una de las *submissions*.

- Tiempo de corrección de una *submission* en Python:
  - Iudex: 11 segundos. Tiempo obtenido observando el log que se muestra en la figura 5.6.
  - DOMJudge: 16 segundos. Tiempo obtenido mediante la interfaz gráfica, disponible en la figura 5.7.

Con esto observamos que el rendimiento de Iudex es ligeramente superior al obtenido por DOMJudge. Estos resultados podrían mejorar gracias a la futura paralelización que permitirá realizar a Iudex ejecuciones de código en varios orquestadores de forma simultanea.

```
14:57:04.801  INFO 29660 --- [io-8080-exec-10] c.e.a.services.ZipHandlerService        : ZIPUNCOMRESS: Se ha anyadido un nuevo zip para descomprimir con el nombre: julio
14:57:04.990  INFO 29660 --- [io-8080-exec-10] c.e.a.services.ProblemValidatorService  : La submision 1 del problema 1 se empieza a recorrer
14:58:40.715  INFO 29660 --- [ntContainer#0-1] c.e.a.services.ProblemValidatorService  : COMPROBANDO PROBLEMA: La comprobacion del problema julio se ha comenzado
14:58:40.729  INFO 29660 --- [ntContainer#0-1] c.e.a.services.ProblemValidatorService  : El problema julio ha sido validado
```

Figura 5.5: Ejemplo del tiempo de ejecución del problema “Julio”.

```
16:46:56.226  INFO 29660 --- [io-8080-exec-10] c.e.aplicacion.services.SubmissionService    : Comienza la Submission 10
16:47:07.208  INFO 29660 --- [io-8080-exec-10] c.e.aplicacion.services.SubmissionService    : Submission 10 corregida
```

Figura 5.6: Tiempo que tarda Iudex en corregir una *submission* para el problema “Julio”.

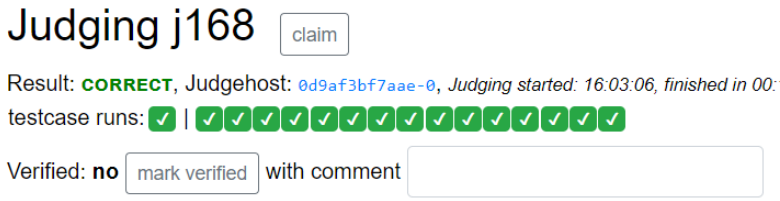

Figura 5.7: Tiempo que tarda DOMjudge en corregir una *submission* para el problema “Julio”.

Capítulo 6

Conclusiones y trabajos futuros

A la finalización de este proyecto, se ha obtenido un sistema que, de forma autónoma, es capaz de ejecutar y corregir códigos a la vez que sirve de repositorio para organizar problemas y concursos. El sistema se decidió dividir en dos proyectos, pudiendo desarrollar independientemente la parte de ejecución del código propiamente dicha de la parte de gestión, corrección y toda la interfaz referente al juez.

Gracias al proyecto he conseguido formarme en dos campos en los que mis conocimientos era muy básicos, como son el mundo de la programación en Java con Spring y el mundo de los contenedores y otras tecnologías de virtualización, ambas pudiéndolas aplicar en mi sector laboral. Todo esto ha sido gracias a la orientación obtenida por parte de los tutores, así como todas las ideas dadas y organización utilizando distintas técnicas de programación.

A la finalización del proyecto todos los objetivos planteados inicialmente quedan resueltos, además de múltiples funcionalidades adicionales que se han implementado como compatibilidad con otros jueces o independencia de los servicios. El resultado obtenido queda en forma de código libre, preparado para recibir una interfaz web y una mejora en el juez, confiando en que otros estudiantes encuentren el desarrollo sobre el sistema sencillo e intuitivo, pudiendo realizar sus TFGs mejorando el juez.

Debido a la necesidad de mejorar el juez y gracias a la arquitectura utilizada, este TFG tienen una gran facilidad a la hora de desarrollar nuevos módulos o mejorar los ya existentes. Actualmente ya se encuentran realizando dos TFGs adicionales que utilizan como base este. Estos son:

- Interfaz Web: Se trata de la creación de una interfaz que permita interactuar de manera sencilla con el juez. Aplicando diferentes técnicas de desarrollo *frontend* utilizando la API previamente definida para el juez.

- Mejora del juez: Actualmente también existe otro proyecto que tiene como fin la mejora del sistema, incluye la implementación de nuevas funcionalidades así como añadir nuevos lenguajes disponibles o arreglar algunos fallos de seguridad.

También se espera implementar una serie de sesiones y diferentes roles de los usuarios que interactúen con la aplicación.

Además de estos trabajos que ya están siendo realizados existen algunos aspectos en los que se puede completar y mejorar el juez:

- Integración dentro del aula virtual, utilizando el proyecto de Spring-LTI [29] de Raúl Martín Santamaría, permite derivar la gestión de roles y usuarios a una plataforma previamente creada como es la plataforma del aula virtual.

Bibliografía

[1] *Online judge*. URL: https://code.fandom.com/wiki/Online_judge#.

[2] *Problem format*. URL: https://clics.ecs.baylor.edu/index.php/Problem_format.

[3] Ahmad Faiyaz. *How does Online Judge Work?* 2017. URL: https://www.linkedin.com/pulse/how-does-online-judge-works-ahmad-faiyaz/.

[4] *Kattis*. URL: https://www.kattis.com/.

[5] *Codeforces*. URL: https://codeforces.com/.

[6] Wikipedia contributors. *Codeforces — Wikipedia, The Free Encyclopedia*. [Online; accessed 6-February-2021]. 2021. URL: https://en.wikipedia.org/w/index.php?title=Codeforces&oldid=1001432488.

[7] *Topcoder*. URL: https://www.topcoder.com/.

[8] URL: https://www.topcoder.com/community/competitive-programming/how-to-compete.

[9] URL: https://www.spoj.com/.

[10] Wikipedia contributors. *SPOJ — Wikipedia, The Free Encyclopedia*. [Online; accessed 6-February-2021]. 2020. URL: https://en.wikipedia.org/w/index.php?title=SPOJ&oldid=994387753.

[11] Miguel A. REVILLA. *Competitive Learning in Informatics: The UVaOnline Judge Experience*. URL: https://ioinformatics.org/journal/INFOL035.pdf.

[12] Wikipedia contributors. *UVa Online Judge — Wikipedia, The Free Encyclopedia*. [Online; accessed 6-February-2021]. 2021. URL: https://en.wikipedia.org/w/index.php?title=UVa_Online_Judge&oldid=1000362248.

[13] URL: https://onlinejudge.org/index.php.

[14] URL: https://github.com/DOMjudge/domjudge/blob/3f8c8b1f8d69c142ea90d34653c55a2c
judge/chroot-startstop.sh.in.

[15] Pedro Pablo Gómez-Martın y Marco Antonio Gómez-Martın. «¡ Acepta el reto!: juez online para docencia en español». En: *Actas de las Jornadas sobre Enseñanza Universitaria de la Informática* 2 (2017), págs. 289-296.

[16] *Acepta el reto documentación*. URL: https://www.aceptaelreto.com/doc/faq.php.

[17] Maria Tena. *¿Qué es la metodología 'agile'?* URL: https://www.bbva.com/es/metodologia-agile-la-revolucion-las-formas-trabajo/.

[18] *WHAT IS SCRUM?* URL: https://www.scrum.org/resources/what-is-scrum.

[19] MAX REHKOPF. *Historias de usuario con ejemplos y plantilla*. URL: https://www.atlassian.com/es/agile/project-management/user-stories.

[20] *Arquitectura de sistemas de información*. URL: https://www.mastergeoinformatica.es/wp-content/uploads/2016/06/FSI-SI-T2-ArquitecturaSI.pdf.

[21] *What is API: Definition, Types, Specifications, Documentation*. Jun. de 2019. URL: https://www.altexsoft.com/blog/engineering/what-is-api-definition-types-specifications-documentation/.

[22] Wikipedia. *Swagger (software) — Wikipedia, La enciclopedia libre*. [Internet; descargado 20-marzo-2021]. 2020. URL: https://es.wikipedia.org/w/index.php?title=Swagger_(software)&oldid=130505934.

[23] Carlos Pesquera. *¿QUÉ ES UN POJO, EJB Y UN BEAN?* URL: https://carlospesquera.com/que-es-un-pojo-ejb-y-un-bean/.

[24] *Conceptos sobre la escalabilidad*. URL: http://www.juntadeandalucia.es/servicios/madeja/contenido/recurso/220#:~:text=Se%20entiende%20por%20escalabilidad%20a,complejo%20e%20importante%20del%20dise%C3%B1o..

[25] Luis Miguel López Magaña. *Qué es Spring framework*. URL: https://openwebinars.net/blog/que-es-spring-framework/.

[26] *¿Qué es Spring Framework? Características I*. URL: https://www.atsistemas.com/blog/qu-es-spring-framework-caractersticas-i.

[27] David Valverde. *¿Qué es Spring?* URL: https://www.davidvalverde.com/blog/que-es-spring/.

[28] *Overview of Spring Framework*. URL: https://docs.spring.io/spring-framework/docs/5.0.0.RC2/spring-framework-reference/overview.html.

[29] Raul Martin. *Spring LTI*. URL: https://github.com/rmartinsanta/spring-lti.

Apéndice A

Guía de instalación

A.1. Instalación del juez

Para la instalación del juez deberemos de cumplir algunos requisitos del sistema básicos:

1. Sistema operativo: El SO soportado por el sistema es de tipo GNU/Linux, concretamente Ubuntu 18.6. El funcionamiento con windows implicaría la modificación de un valor referente a la conexión con el Docker-Engine desde el código. Se deberá modificar dentro de la clase *ResultHandler* la linea que hace referencia a *dockerClient.getInstance(ünix:///var/run/docker.sock")* debiéndose modificar el texto que se encuentra en comillas por la localización del *socket* del motor.

2. Sistema de ficheros XFS: El sistema operativo debe de estar basado en un sistema ficheros XFS, esto es debido a una funcionalidad de seguridad que tiene el orquestador. En caso de que no se disponga de dicha característica, se puede bloquear la funcionalidad del orquestador borrando de la linea referente a la configuración del objeto *hostConfig* la opción que hace referencia a *withStorageOpt()*

3. Instalación de Docker-Engine. En cualquier caso la instalación de docker engine viene ampliamente explicada en la pagina web oficial https://docs.docker.com/engine/install/

4. RabbitMQ: RabbitMQ es un motor que deberá de ser instalado de forma local en las maquinas que se utilicen para la ejecución. Podemos aprovechar que ya tenemos instalado docker-engine pudiendo levantar un contenedor que se dedique a contener una instancia de RabbitMQ. El código en linux para ello sería:

```
docker run -it --rm --name rabbitmq -p 5672:5672
-p 15672:15672 rabbitmq:3-management
```

A.2. Guía añadir un nuevo lenguaje

Para añadir un lenguaje necesitaremos seguir los siguientes pasos:

1. Crear un DockerFile. Sera necesario crear una carpeta dentro del directorio DOCKERS cuyo nombre sea el lenguaje a implementar. Dentro de esta carpeta crearemos el fichero DockerFile, el cual nos basaremos en el resto ya existentes cambiando el ENTRYPOINT y la imagen de la que derivamos.

2. Añadir al basicController la inicialización del DockerFile así como la inclusión del lenguaje en la BBDD.

3. Crear la clase correspondiente en la carpeta Docker, llamándola DockerLenguaje (cambiando Lenguaje por el lenguaje a tratar). Tendremos que implementar las diferentes características y variables de entorno del lenguaje en cuestión. Las entradas y salidas no debería hacer falta modificarlas, si acaso añadir o retirar según el lenguaje.

4. Añadir en la clase ResultHandler el "case" para el lenguaje.

5. Para que también lea los códigos que entran desde los problemas mediante los ZIPs, tendremos q modificar la clase ZIPHandlerService la función SelectLenguaje añadiendo la extensión y traduciéndola al lenguaje que toca.