ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
INGENIERÍA DEL SOFTWARE

Herramienta visual educativa para la resolución de problemas

Herramienta del profesor

Visual learning tool for problem solving

Professor tool

Realizado por
Antonio Pareja Jaén

Tutorizado por
Eduardo Guzmán de los Riscos

Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, JUNIO DE 2019

Fecha defensa:

Fdo. El/la Secretario/a del Tribunal

Resumen:

En este Trabajo de Fin de Grado se ha realizado el diseño e implementación de una aplicación web cuyo objetivo es que sirva como herramienta didáctica para ser utilizada en clase por profesores y alumnos, en la que se pretende enseñar los fundamentos básicos de programación a alto nivel, utilizando bloques arrastrables y anidables unos con otros. Además la aplicación podrá evaluar automáticamente a los alumnos mediante un algoritmo de corrección.

La aplicación usa un sistema de roles de profesor y alumno. El profesor creará problemas utilizando un entorno donde aparecen bloques con distinta funcionalidad, que corresponden a líneas de código pero dándole un aspecto visual e intuitivo. El alumno usará la aplicación para resolver los problemas propuestos por el profesor. El sistema cuenta con un algoritmo de corrección que devolverá una nota propuesta para cada problema, que servirá al profesor de ayuda a la hora de evaluar un ejercicio y decidir la nota definitiva.

Palabras clave: Herramienta visual para la resolución de problemas**,** AngularJS, MongoDB, Spring.

Abstract:

This work describes the design and implementation of a web application that serves as a didactic tool to be used in class by teachers and students, in which it is intended to teach the basics of programming at a high level, using draggable and nestable blocks. In addition, the application can automatically evaluate students through a correction algorithm.

The application uses a system of roles, which are teachers and students. The teachers will create problems using an environment where blocks with different functionality appear, corresponding to lines of code but giving it a visual and intuitive appearance. The students will use the application to solve the problems proposed by the teacher. The system has a correction algorithm that will return a proposed note for each problem, which will help the teachers to help evaluate an exercise and decide the final grade.

Keywords: Visual tool for problem solving activities, AngularJS, MongoDB, Spring.

Índice de contenidos

1. Introducción

1.1. Descripción y objetivos

Este trabajo de fin de grado surge de la necesidad de una aplicación que permita:

- Resolver problemas a alto nivel de forma visual, mediante el uso de bloques.
- Definir bloques propios personalizables según las necesidades de cada usuario/profesor.
- Un sistema con roles profesor/alumno donde el profesor puede definir bloques y problemas mientras que el alumno podrá usar los bloques definidos para solucionar problemas.

El proyecto consiste en la creación de una herramienta web pensada para ser utilizada por profesores y sus respectivos alumnos, en un ámbito académico en el que el docente pueda, mediante el uso de bloques de colores con distinta función, crear problemas que contengan un enunciado y una composición de bloques que se llamará solución propuesta por el profesor, mientras que el alumno pueda plantear soluciones a estos problemas, utilizando igualmente los bloques de colores. La herramienta además cuenta con la función de analizar las soluciones de los alumnos, y de devolver una nota, que será luego interpretada por el profesor, el cual podrá ver las soluciones que han sido enviadas por sus alumnos, y podrá también cambiar la nota si lo cree necesario, debido a que no esté de acuerdo con la nota propuesta por la herramienta.

Actualmente existen herramientas que permiten la resolución de problemas de programación basados en bloques (Scratch, Snap, AppInventor), pero ninguna cumple con todos los requisitos de esta herramienta.

El objetivo del proyecto es proporcionar una herramienta que facilite a los alumnos la forma en que aprenden programación, haciéndolo más visual e intuitivo, a la vez que se facilita la tarea del profesor de explicar los fundamentos básicos de programación. Claramente es una herramienta bastante simple, con la que no se pueden crear estructuras muy complejas, ni está enfocada a usarse como entorno de programación real, pero la forma en la que está diseñada pretende hacer más intuitiva

la enseñanza de las nociones básicas de programación, las cuales, en muchas ocasiones en las escuelas de ingeniería se dan por sabidas desde el primer día, y conlleva a que los alumnos se frustren y abandonen las asignaturas tras perder interés en éstas.

1.2. Tecnologías y herramientas

Para realizar la aplicación se han utilizado las siguientes aplicaciones, herramientas y tecnologías:

- **Spring:** es un framework para el desarrollo de aplicaciones y contenedor de inversión de control, de código abierto para la plataforma Java.
- **Spring Tools:** Conjunto de herramientas basadas en el framework Spring para el desarrollo de aplicaciones. Se ha utilizado en el entorno de programación Eclipse.
- **Eclipse IDE:** es un entorno de desarrollo software compuesto por un conjunto de herramientas de programación de código abierto multiplataforma, y es el IDE de Java más utilizado.
- **Java:** Java es un lenguaje de programación de propósito general, concurrente y orientado a objetos.
- **Javascript:** es un lenguaje de programación interpretado, dialecto del estándar ECMAScript. Se define como orientado a objetos, basado en prototipos, imperativo, débilmente tipado y dinámico. Permite mejoras en la interfaz de usuario y páginas web dinámicas.
- **AngularJS:** es un framework de JavaScript de código abierto que se utiliza para crear y mantener aplicaciones web de una sola página. Su objetivo es aumentar las aplicaciones basadas en navegador con capacidad de Modelo Vista Controlador (MVC), en un esfuerzo para hacer que el desarrollo y las pruebas sean más fáciles.
- **Drag & Drop Lists:** es una librería de AngularJS que nos permite arrastrar y soltar bloques.
- **HTML:** es un lenguaje de programación que se utiliza para el desarrollo de páginas de Internet.

- **CSS:** es un lenguaje de diseño gráfico para definir y crear la presentación de un documento estructurado escrito en un lenguaje de marcado. Es muy usado para establecer el diseño visual de los documentos web, e interfaces de usuario.
- **Bootstrap:** es una biblioteca multiplataforma o conjunto de herramientas de código abierto para diseño de sitios y aplicaciones web. Contiene plantillas de diseño con tipografía, formularios, botones, cuadros, menús de navegación y otros elementos de diseño basado en HTML y CSS, así como extensiones de JavaScript adicionales.
- **Node.js:** es un entorno en tiempo de ejecución multiplataforma, para la capa del servidor.
- **MongoDB:** es un sistema de base de datos NoSQL orientado a documentos de código abierto. En lugar de guardar los datos en tablas, tal y como se hace en las bases de datos relacionales, MongoDB guarda estructuras de datos BSON (una especificación similar a JSON) con un esquema dinámico, haciendo que la integración de los datos en la aplicación sea más fácil y rápida.
- **MongoDB Compass:** es una interfaz gráfica de usuario para MongoDB. Sirve para explorar los datos visualmente, ejecutar consultas ad hoc en segundos e interactuar con los datos gracias a todas las funciones CRUD.
- **Sublime Text:** es un editor de texto y editor de código fuente.
- **Advance Rest Client para Google Chrome:** es una interfaz gráfica que sirve para probar aplicaciones REST. Permite realizar las distintas operaciones que usa una API Rest y ver e interpretar las salidas de forma fácil y limpia.
- **Trello:** es un software de administración de proyectos con interfaz web y con cliente para iOS y android para organizar proyectos.
- **Magic Draw:** es una herramienta de modelado visual UML. Diseñada para analistas de negocios, analistas de software, programadores e ingenieros de control de calidad, esta herramienta de desarrollo dinámica y versátil facilita el análisis y el diseño de sistemas y bases de datos orientados a objetos.
- **Pencil Project:** herramienta que se ha utilizado para realizar las maquetas de la interfaz gráfica de la aplicación.

1.3. División del trabajo

Este proyecto ha sido desarrollado por dos personas y, por tanto, se ha dividido en dos partes: la herramienta del profesor, y la herramienta del alumno. Este documento se centra en la herramienta del profesor, mientras que la otra miembro del grupo se encarga de desarrollar la herramienta del alumno. Además, como la herramienta del profesor conlleva hacer el algoritmo de corrección, el cual es una tarea bastante extensa, hemos separado la parte común de la aplicación (menú de inicio, inicio de sesión, configuración de cuentas, etc.) para que se encargue de ello una sola persona.

Sin embargo, las dos partes tienen muchos elementos en común, y ambos miembros hemos compartido los conocimientos que íbamos adquiriendo. Por esta razón hemos ido trabajando casi a la par, y hemos estado al tanto de lo que ha hecho tanto uno como otro durante gran parte del tiempo.

1.4. Estructura de la memoria

1. **Introducción**: en este punto inicial se explicará en qué consiste toda la aplicación, proporcionando una descripción detallada de la misma, así como también se expondrán los objetivos del proyecto.
2. **Especificación y análisis**: se empezará describiendo a los participantes y usuarios de la aplicación, luego se listarán los requisitos de la aplicación, que están divididos en funcionales y no funcionales. Los funcionales serán los de la herramienta del profesor, mientras que los no funcionales son comunes en los dos miembros del proyecto. Por último, se realizará un análisis de los casos de uso.
3. **Diseño**: se mostrarán y se explicarán varios diagramas, entre ellos el diagrama de clases o el diagrama de la arquitectura de la aplicación.
4. **Implementación**: en este apartado se explicará la estructura de ficheros de la aplicación. También se detallarán las partes del código consideradas más relevantes.
5. **Pruebas**: se hablará sobre las distintas pruebas realizadas para comprobar el correcto funcionamiento de la aplicación y se comprobará que se cumplen todos los requisitos.

6. **Conclusiones**: se expondrán los objetivos cumplidos, así como las dificultades encontradas a lo largo del proyecto. También se hablará de las posibles ampliaciones que podría llegar a tener la aplicación.
7. **Referencias**: se listará el conjunto de enlaces y ficheros a los que hemos acudido para ayudarnos a realizar el proyecto.

2. Especificación y análisis

2.1 Participantes y usuarios

Nombre	Alumno
Descripción	Usuario que realiza soluciones para problemas existentes
Tipo	Usuario registrado
Responsabilidades	Crear una solución y guardarla en el sistema para su posterior evaluación
Criterio de éxito	Capacidad de ver una nota como resultado a la solución que realizó previamente

Nombre	Profesor
Descripción	Usuario que crea bloques personalizados y problemas, y que corrige soluciones que le llegan de los alumnos.
Tipo	Usuario registrado.
Responsabilidades	Crear problemas que puedan ser realizados por alumnos para su posterior evaluación, y crear bloques que se necesiten para la realización de esos problemas. Evaluar a los alumnos guardando una nota en el sistema.
Criterio de éxito	Capacidad de usar los bloques que ha creado él mismo previamente, y de recibir soluciones de sus alumnos listas para ser evaluadas.

2.2 Requisitos de la aplicación

2.2.1 Requisitos funcionales profesor (RFP)

RFP 01 - Crear bloque personalizado

Un profesor puede crear bloques personalizados que podrán ser utilizados en sus problemas

RFP 02 - Listar bloques personalizados

Un profesor puede ver una lista de bloques que él ha creado

RFP 03 - Editar bloque personalizado

Un profesor puede editar un bloque que él haya creado

RFP 04 - Borrar bloque personalizado

Un profesor puede borrar un bloque que él haya creado

RFP 05 - Crear problema

Un profesor puede crear un problema dándole un título, una descripción y un conjunto de bloques que compondrán la solución propuesta

RFP 06 - Listar problemas

Un profesor puede ver una lista problemas que él ha creado

RFP 07 - Editar problema

Un profesor puede editar un problema creado por él, y podrá modificar tanto el título y la descripción, como la solución propuesta

RFP 08 - Borrar problema

Un profesor puede borrar un problema que él haya creado

RFP 09 - Listar soluciones

Un profesor puede ver una lista de las soluciones existentes de cada problema, creadas previamente por alumnos

RFP 10 - Corregir solución

Un profesor puede ponerle una nota a la solución realizada por un alumno

2.2.2 Requisitos no funcionales (RNF)

RNF 01 - La aplicación utilizará RESTful con Spring Tool para implementar el back-end

RNF 02 - La aplicación utilizará MongoDB como base de datos no relacional

RNF 03 - La aplicación utilizará AngularJS para implementar el front-end de la aplicación

2.3 Casos de uso

RFP 01 - Crear bloque personalizado

Título	Crear bloque personalizado
Descripción	Un profesor puede crear bloques personalizados que podrán ser utilizados en sus problemas, tanto por él como por sus alumnos
Pre-condición	Un usuario, con rol de profesor, debe estar autenticado en la aplicación
Post-condición	El bloque se almacena en la base de datos y aparecerá en el listado de bloques personalizados
Escenario principal	
1. El profesor pulsa sobre el botón “Custom blocks” de la pestaña Home. 2. El profesor pulsa sobre el botón “Create block” que aparece al final del listado. 3. El profesor introduce el nombre (title) del nuevo bloque. 4. El profesor añade tantos campos “inputs” como necesite pulsando sobre el botón “+”. 5. El profesor selecciona el tipo de los campos (int, double, etc..). 6. El profesor selecciona el tipo de salida del bloque (output). 7. El profesor pulsa sobre el botón “Save”. 8. El sistema crea el bloque personalizado en la base de datos. 9. Se muestra listado de bloques personalizados, en el que aparecerá el bloque que acaba de crear.	

Escenario alternativo	
7b. El profesor intenta pulsar sobre el botón “Save” sin seleccionar el tipo de algún “input” o el de la salida, o habiendo dejado el nombre vacío. 8b. El sistema no le deja pulsar hasta que no rellene lo que le falta.	
Clases de análisis	
A. Clases de entidad	CustomBlock.java
B. Clases de control	CustomBlockController.java teacherController.js
C. Clases de interfaz	custom-block-edit.html
Maquetas de interfaz	

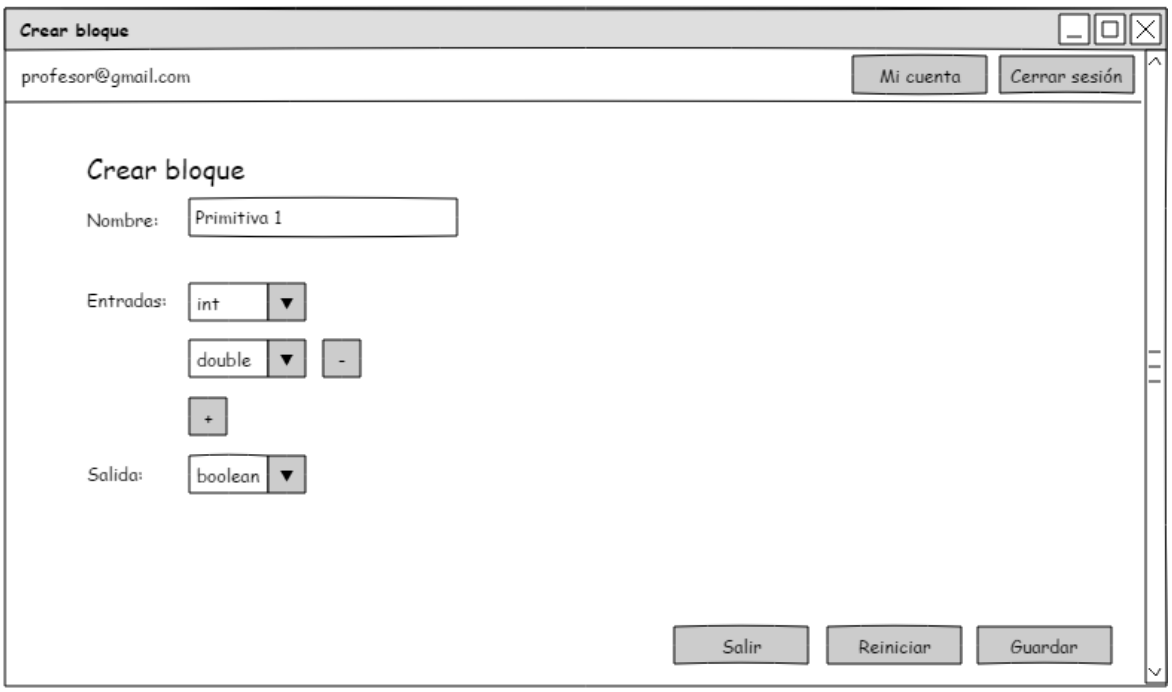

Figura 1 - Maqueta de crear bloque personalizado

RFP 02 - Listar bloques personalizados

Título	Listar bloques personalizados

Descripción	Un profesor puede ver una lista de bloques que él ha creado
Pre-condición	Un usuario, con rol de profesor, debe estar autenticado en la aplicación. Debe haber al menos un bloque creado.
Post-condición	El profesor visualiza una lista de bloques personalizados
Escenario principal	
1. El profesor pulsa sobre el botón “Custom blocks” de la pestaña Home. 2. Se muestra un listado con todos los bloques personalizados creados previamente por ese profesor, con botones de editar y borrar al lado de cada bloque.	
Escenario alternativo	
2b. El sistema no encuentra ningún bloque y se muestra un listado vacío junto al botón “Create block".	
Clases de análisis	
A. Clases de entidad	CustomBlock.java
B. Clases de control	CustomBlockController.java teacherController.js
C. Clases de interfaz	custom-block-list.html
Maquetas de interfaz	

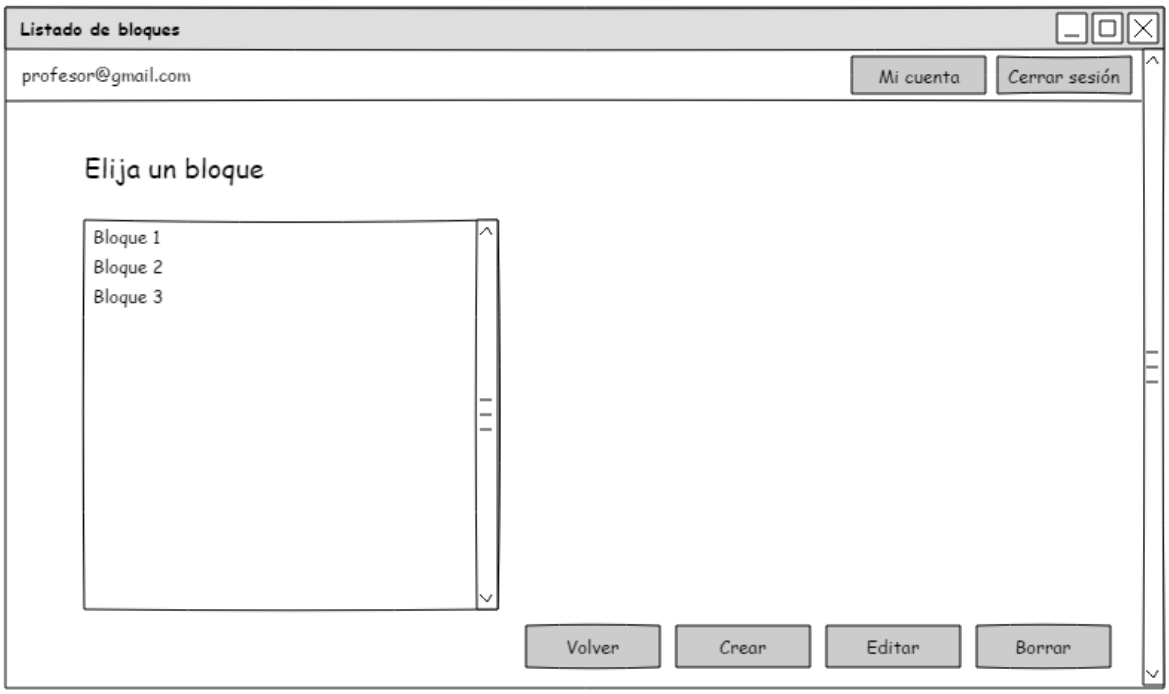

Figura 2 - Maqueta de listar bloques personalizados

RFP 03 - Editar bloque personalizado

Título	Editar bloque personalizado
Descripción	Un profesor puede editar un bloque que él haya creado
Pre-condición	Un usuario, con rol de profesor, debe estar autenticado en la aplicación. Debe existir al menos un bloque creado.
Post-condición	El bloque personalizado se actualiza con los cambios realizados.
Escenario principal	
1. El profesor pulsa sobre el botón “Custom blocks” de la pestaña Home. 2. El profesor pulsa sobre el botón “Edit” que aparece al lado del bloque que	

quiere editar.

3. Se muestra una ventana similar a la de “crear bloque personalizado”.
4. El profesor edita los campos que crea necesarios.
5. El profesor pulsa sobre el botón “Save”.
6. El sistema actualiza el bloque en la base de datos.
7. Se muestra listado de bloques personalizados, en el que aparecerá el bloque que se acaba de editar.

Escenario alternativo	
5b. El profesor intenta pulsar sobre el botón “save” sin seleccionar el tipo de algún “input” o el de la salida, o habiendo dejado el nombre vacío. 6b. El sistema no le deja pulsar hasta que no rellene lo que le falta.	
Clases de análisis	
A. Clases de entidad	CustomBlock.java
B. Clases de control	CustomBlockController.java teacherController.js
C. Clases de interfaz	custom-block-edit.html
Maquetas de interfaz	

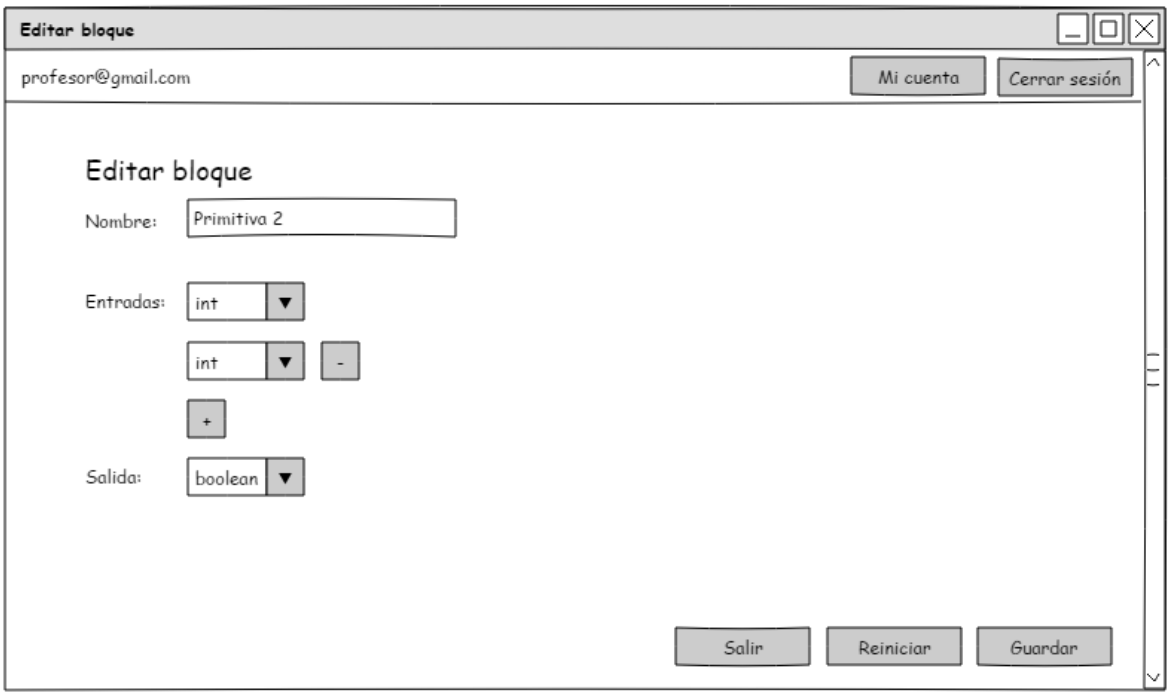

Figura 3 - Maqueta de editar bloque personalizado

RFP 04 - Borrar bloque personalizado

Título	Borrar bloque personalizado
Descripción	Un profesor puede borrar un bloque que él haya creado previamente.
Pre-condición	Un usuario, con rol de profesor, debe estar autenticado en la aplicación. Debe existir al menos un bloque creado.
Post-condición	El profesor visualiza una lista de bloques personalizados, en la que no aparece el bloque que acaba de ser eliminado
Escenario principal	
1. El profesor pulsa sobre el botón “Custom blocks” de la pestaña Home.	

2. El profesor pulsa sobre el botón “Delete” que aparece al lado del bloque que quiere borrar.
3. Aparece un mensaje preguntando al usuario si está seguro de que quiere borrar el bloque.
4. El profesor pulsa sobre “Aceptar”.
5. El bloque es eliminado de la base de datos.
6. Se actualiza el listado de bloques sin el bloque que se acaba de borrar.

Escenario alternativo

4b. El profesor pulsa sobre “Cancelar”.
5b. No se borra el bloque seleccionado.

Clases de análisis

A. Clases de entidad	CustomBlock.java
B. Clases de control	CustomBlockController.java teacherController.js
C. Clases de interfaz	custom-block-list.html

Maquetas de interfaz

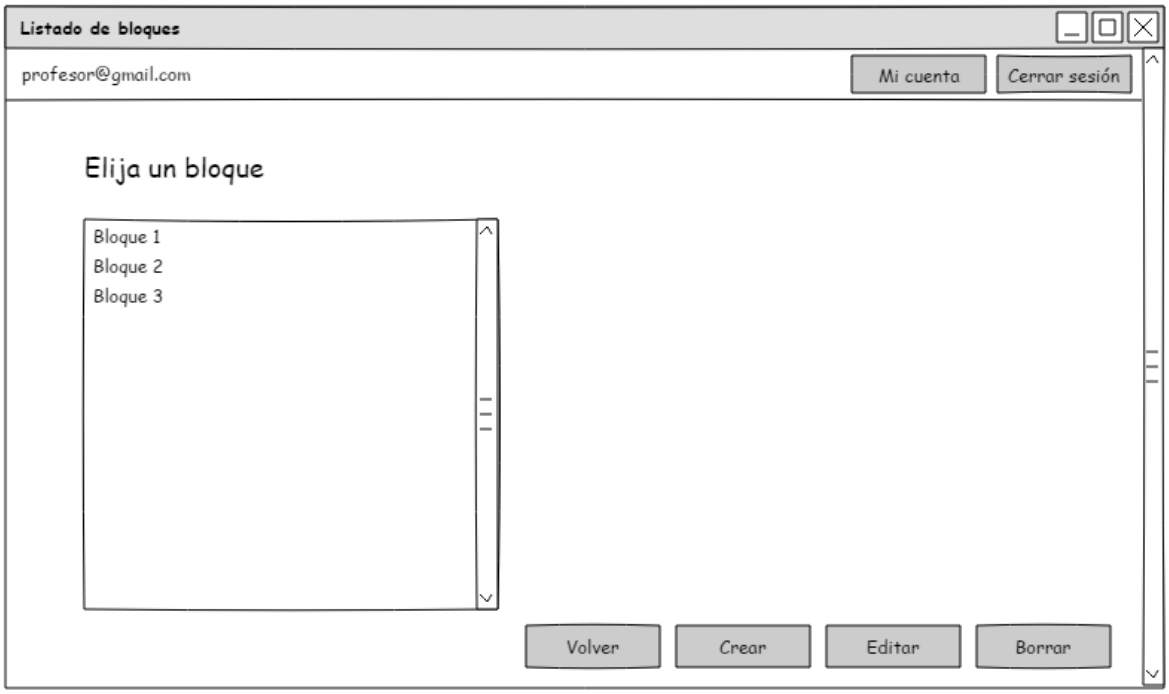

Figura 4 - Maqueta de borrar bloque personalizado

RFP 05 - Crear problema

Título	Crear problema
Descripción	Un profesor puede crear un problema dándole un título, una descripción y un conjunto de bloques que compondrán la solución propuesta
Pre-condición	Un usuario, con rol de profesor, debe estar autenticado en la aplicación
Post-condición	El problema se almacena en la base de datos, aparecerá en el listado de problemas de profesor, y podrá ser resuelto por usuario que sean alumnos
Escenario principal	
1. El profesor pulsa sobre el botón “Problems” de la pestaña Home. 2. El profesor pulsa sobre el botón “Create problem” que aparece al final del listado. 3. El profesor introduce el título del nuevo problema. 4. El profesor escribe el código del nuevo problema. 5. El profesor redacta una descripción para el problema. 6. El profesor pulsa sobre el botón “Save”. 7. Se muestra una nueva ventana de edición en la que el profesor crea, utilizando bloques, una solución al problema. 8. El profesor pulsa sobre el botón “Save & Finish”. 9. Se muestra listado de problemas, en el que aparecerá el problema que	

acaba de crear.

Escenario alternativo

6b. El profesor pulsa sobre el botón “Save” sin rellenar alguno de los campos.

7b. El sistema no le deja avanzar hasta que no rellene lo que le falta.

4c. El profesor escribe un código que ya existe.

6c. El profesor pulsa sobre el botón “Save”.

7c. Aparece un mensaje indicando que ya existe un problema con ese código.

Clases de análisis

A. Clases de entidad	Problem.java
B. Clases de control	ProblemController.java problemController.js
C. Clases de interfaz	problem-create.html problem-edit.html

Maquetas de interfaz

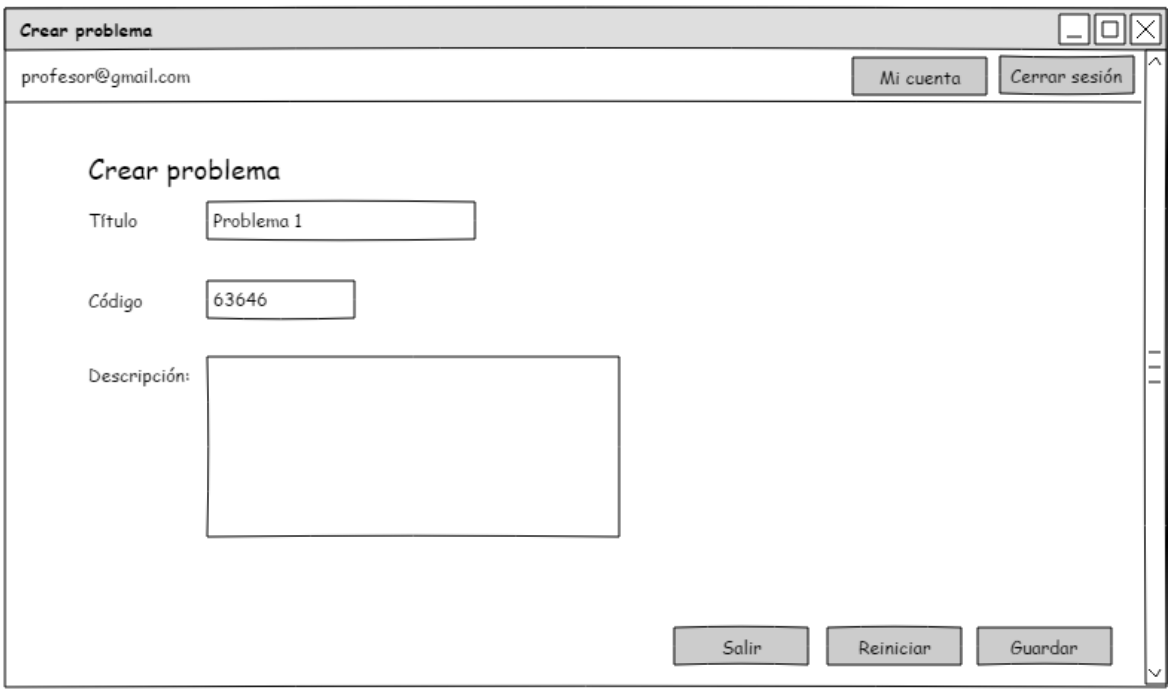

Figura 5.1 - Maqueta de crear problema, introduciendo datos

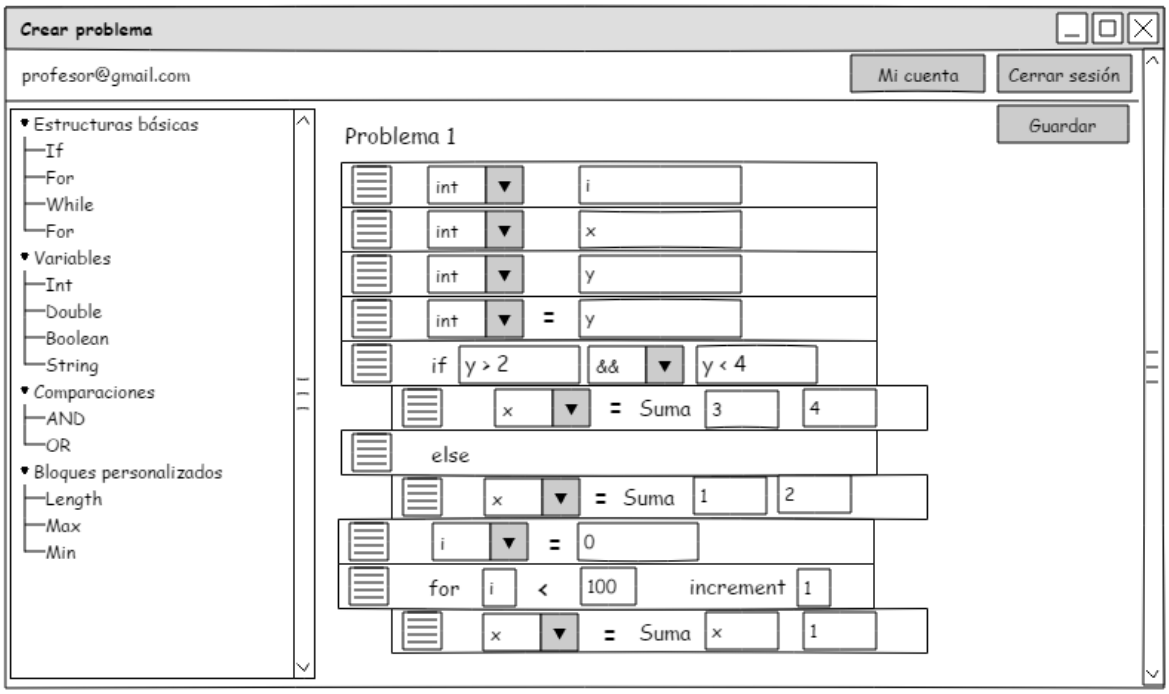

Figura 5.2 - Maqueta de crear problema, creando bloques

RFP 06 - Listar problemas

Título	Listar problemas
Descripción	Un profesor puede ver una lista problemas que él ha creado
Pre-condición	Un usuario, con rol de profesor, debe estar autenticado en la aplicación. Debe existir al menos un problema creado.
Post-condición	El profesor visualiza una lista de problemas
Escenario principal	
1. El profesor pulsa sobre el botón “Problems” de la pestaña Home.	

2. Se muestra un listado con todos los problemas creados previamente por ese profesor, con botones de editar y borrar al lado de cada problema.

Escenario alternativo

2b. El sistema no encuentra ningún problema y se muestra un listado vacío junto al botón “Create problem".

Clases de análisis

A. Clases de entidad	Problem.java
B. Clases de control	ProblemController.java teacherController.js
C. Clases de interfaz	problem-list.html

Maquetas de interfaz

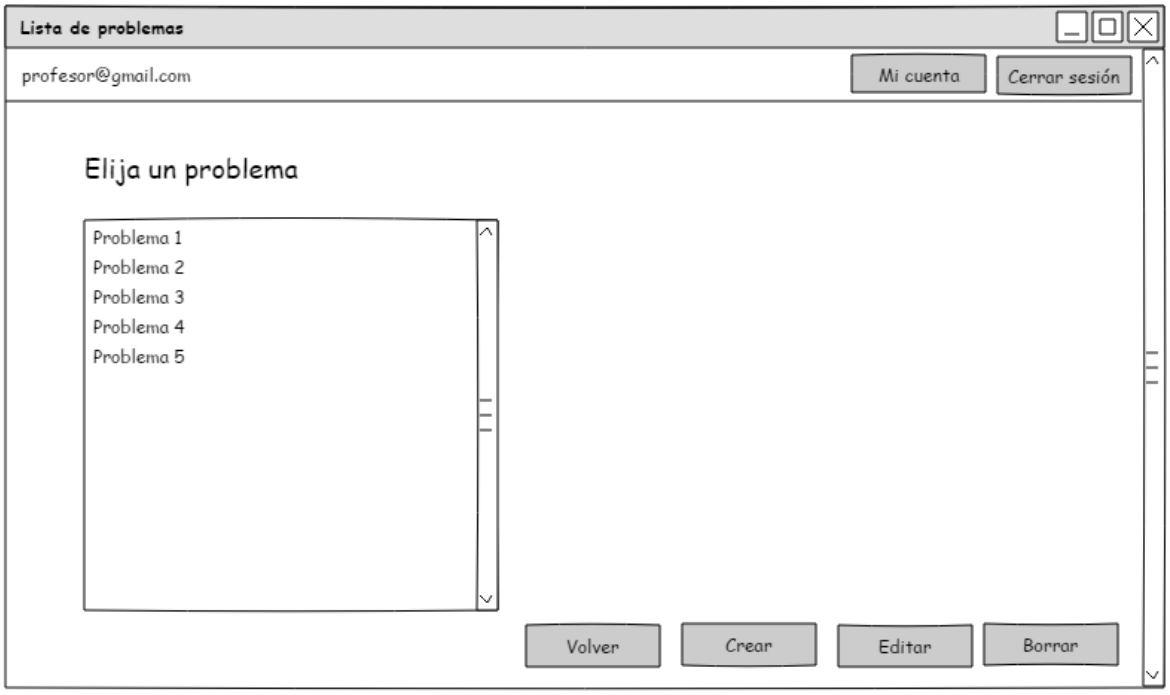

Figura 6 - Maqueta de listar problemas

RFP 07 - Editar problema

Título	Editar problema
Descripción	Un profesor puede editar un problema creado por él, y podrá modificar tanto el título y la descripción, como la solución propuesta
Pre-condición	Un usuario, con rol de profesor, debe estar autenticado en la aplicación. Debe existir al menos un problema creado.
Post-condición	El problema se actualiza con los cambios realizados.
Escenario principal	
1. El profesor pulsa sobre el botón “Problems” de la pestaña Home. 2. El profesor pulsa sobre el botón “Edit problem” que aparece al lado del problema que quiere editar. 3. Se muestra una ventana similar a la de “crear problema”. 4. El profesor edita los campos que crea necesarios. 5. El profesor pulsa sobre el botón “Save”. 6. Se muestra una ventana de edición para que el profesor edite la solución propuesta. 7. El profesor pulsa sobre “Save & Finish”. 8. El sistema actualiza el problema en la base de datos. 9. Se muestra listado de problemas, en el que aparecerá el problema que se acaba de editar.	
Escenario alternativo	
2b. El profesor pulsa sobre el botón “Edit solution” que le llevará directamente a editar la solución propuesta.	

<table>
<tr><td colspan="2">3b. Se muestra una ventana de edición para que el profesor edite la solución propuesta.
4b. El profesor pulsa sobre “Save & Finish”.
5b. El sistema actualiza el problema en la base de datos.
6b. Se muestra listado de problemas, en el que aparecerá el problema que se acaba de editar.

5c. El profesor intenta pulsar sobre el botón “Save” sin rellenar alguno de los campos.
6c. El sistema no le deja avanzar hasta que no rellene lo que le falta.

4d. El profesor escribe un código que ya existe.
5d. El profesor pulsa sobre el botón “Save”.
6d. Aparece un mensaje indicando que ya existe un problema con ese código.</td></tr>
<tr><td colspan="2">Clases de análisis</td></tr>
<tr><td>A. Clases de entidad</td><td>Problem.java</td></tr>
<tr><td>B. Clases de control</td><td>ProblemController.java
problemController.js
teacherController.js</td></tr>
<tr><td>C. Clases de interfaz</td><td>problem-create.html
problem-edit.html</td></tr>
<tr><td colspan="2">Maquetas de interfaz</td></tr>
</table>

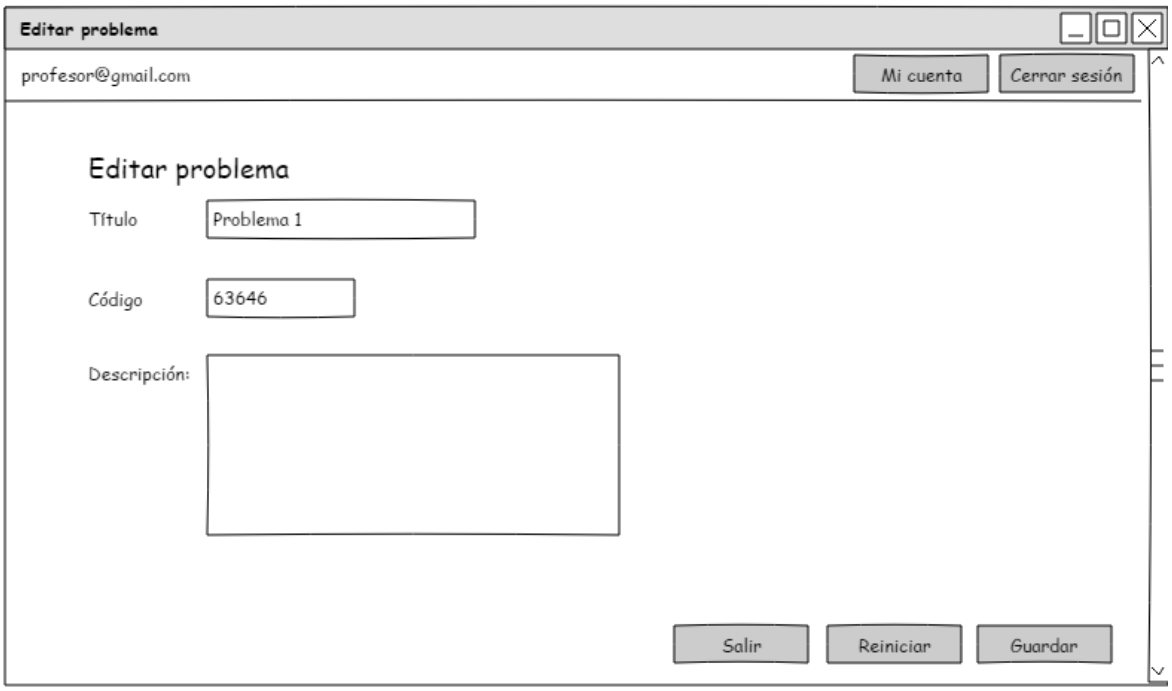

Figura 7.1 - Maqueta de editar problema, modificando datos

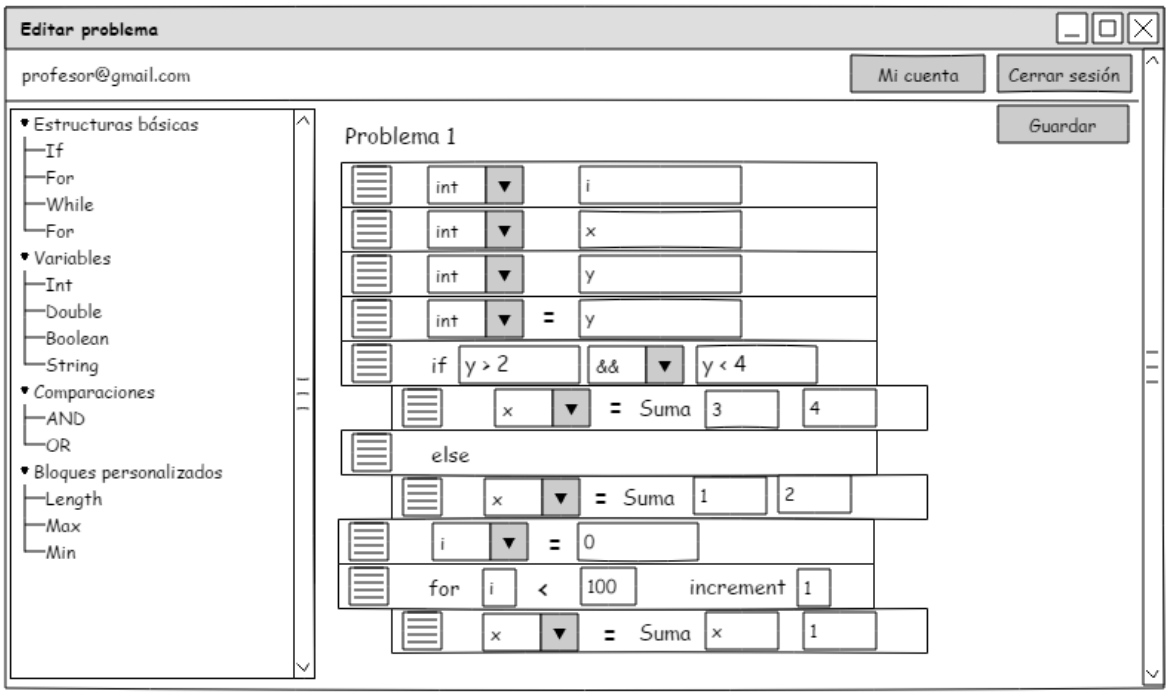

Figura 7.2 - Maqueta de editar problema, modificando bloques

RFP 08 - Borrar problema

Título	Borrar problema
Descripción	Un profesor puede borrar un problema que él haya creado previamente.
Pre-condición	Un usuario, con rol de profesor, debe estar autenticado en la aplicación. Debe existir al menos un problema creado.
Post-condición	El profesor visualiza una lista de problemas, en la que no aparece el problema que acaba de ser eliminado
Escenario principal	
1. El profesor pulsa sobre el botón “Problems” de la pestaña Home. 2. El profesor pulsa sobre el botón “Delete” que aparece al lado del problema que quiere borrar. 3. Aparece un mensaje preguntando al usuario si está seguro de que quiere borrar el problema. 4. El profesor pulsa sobre “Aceptar”. 5. El problema es eliminado de la base de datos. 6. Se actualiza el listado de problemas sin el problema que se acaba de borrar.	
Escenario alternativo	
4b. El profesor pulsa sobre “Cancelar”. 5b. No se borra el problema seleccionado.	
Clases de análisis	
A. Clases de entidad	Problem.java

B. Clases de control	ProblemController.java teacherController.js
C. Clases de interfaz	problem-list.html
Maquetas de interfaz	

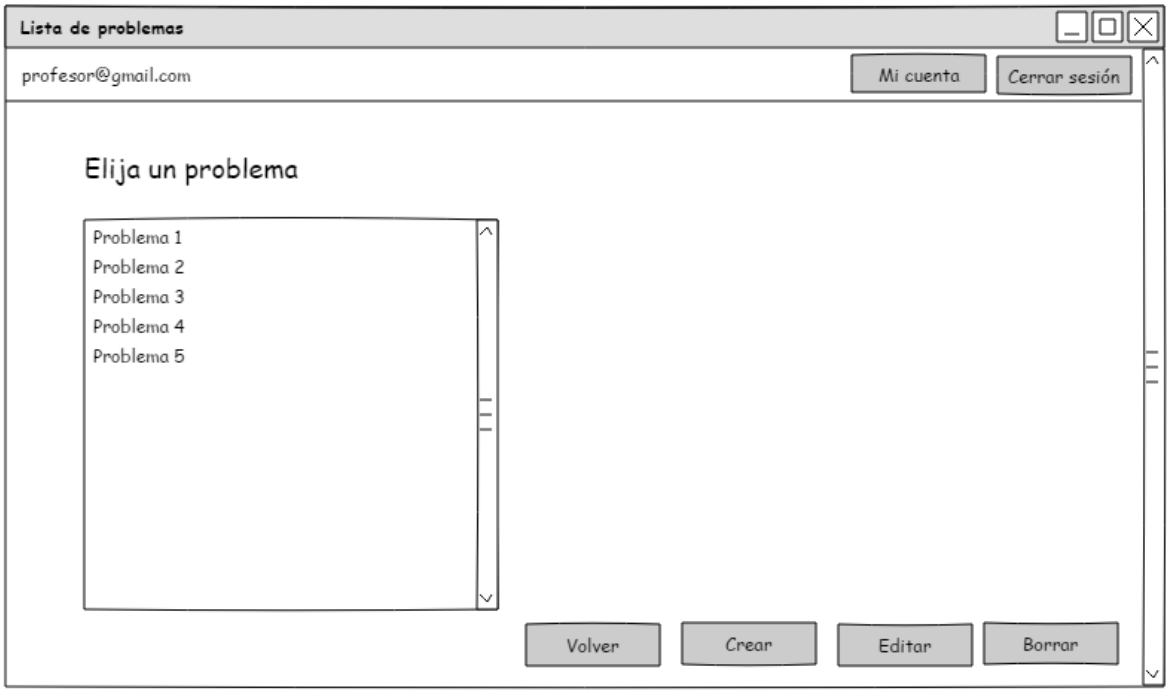

Figura 8 - Maqueta de borrar problema

RFP 09 - Listar soluciones

Título	Listar soluciones
Descripción	Un profesor puede ver una lista de las soluciones existentes de cada problema, creadas previamente por alumnos. En cada entrada de la lista también aparece el alumno al que pertenece cada solución y el problema al que hacen referencia.
Pre-condición	Un usuario, con rol de profesor, debe

	estar autenticado en la aplicación
Post-condición	El profesor visualiza una lista de soluciones
Escenario principal	
1. El profesor pulsa sobre el botón “Review problems” de la pestaña Home. 2. Se muestra un listado de problemas creados por el profesor. 3. El profesor pulsa sobre un problema de la lista. 4. Aparece en la misma ventana, debajo de los problemas, un listado con las soluciones existentes para problema seleccionado. Cada solución de la lista aparece con el nombre del alumno a la que pertenece.	
Escenario alternativo	
3b. El profesor pulsa sobre un problema que no tiene soluciones. 4b. Aparece un mensaje indicando que no existen aún soluciones para ese problema.	
Clases de análisis	
A. Clases de entidad	Solution.java
B. Clases de control	SolutionController.java teacherController.js
C. Clases de interfaz	revise-problems-list.html
Maquetas de interfaz	

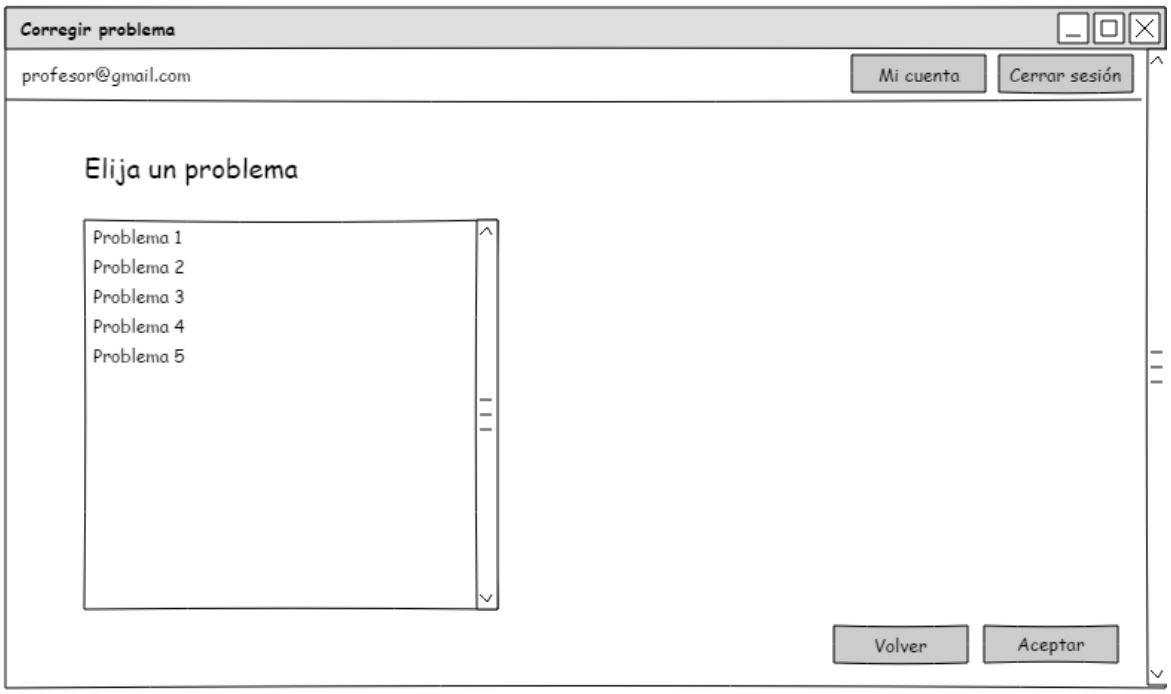

Figura 9.1 - Maqueta de listar soluciones, eligiendo problema

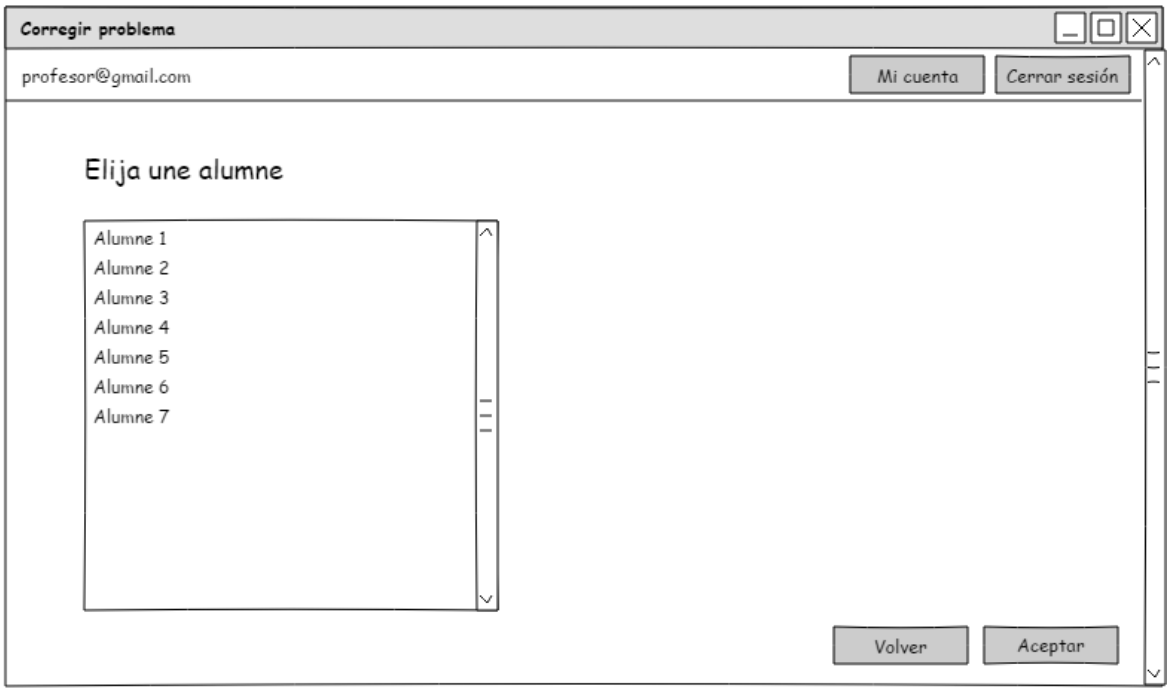

Figura 9.2 - Maqueta de listar soluciones, eligiendo alumno

RFP 10 - Corregir solución

Título	Corregir solución

Descripción	Un profesor puede ponerle una nota a la solución realizada por un alumno
Pre-condición	Un usuario, con rol de profesor, debe estar autenticado en la aplicación
Post-condición	

Escenario principal

1. El profesor pulsa sobre el botón “Review problems” de la pestaña Home.
2. Se muestra un listado de problemas creados por el profesor.
3. El profesor pulsa sobre un problema de la lista.
4. Aparece en la misma ventana, debajo de los problemas, un listado con las soluciones existentes para problema seleccionado. Cada solución de la lista aparece con el nombre del alumno a la que pertenece.
5. El profesor pulsa sobre el botón “Review” de una solución.
6. Aparece en pantalla la solución del alumno, junto a una nota calculada por el algoritmo de corrección
7. El profesor escribe la nota que cree que debe tener el alumno basándose en la nota del algoritmo y en su propio punto de vista
8. El profesor pulsa sobre el botón “save” para guardar la nota de esa solución del alumno
9. Se actualiza el listado de soluciones, con la solución que se acaba de corregir junto con la nota que se le ha puesto

Escenario alternativo

3b. El profesor pulsa sobre un problema que no tiene soluciones.
4b. Aparece un mensaje indicando que no existen aún soluciones para ese problema.

Clases de análisis

A. Clases de entidad	Solution.java

B. Clases de control	SolutionController.java problemController.java teacherController.js
C. Clases de interfaz	revise-problems-list.html

Maquetas de interfaz

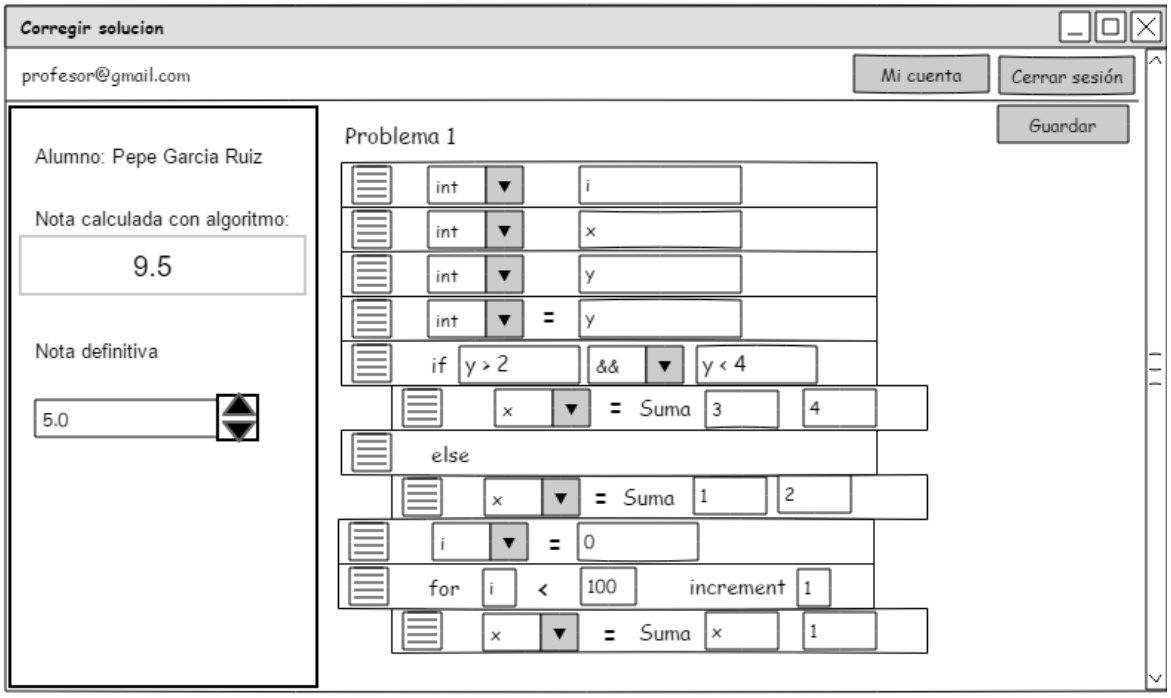

Figura 10 - Maqueta de corregir solución

3. Diseño

3.1 Diagrama arquitectónico

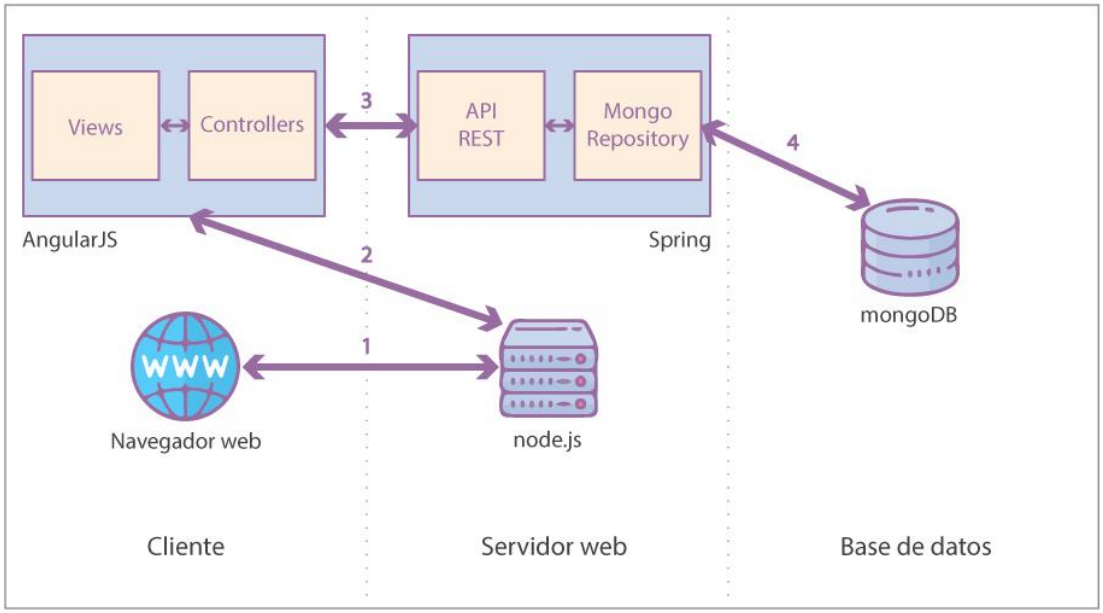

Figura 11 - Diagrama arquitectónico

Para tener una visión global de cómo está construida la aplicación, se ha creado un diagrama que contenga las tecnologías más representativas que se utilizan y las tareas de las que se encargan, y se han dividido de forma que podamos diferenciar mejor el papel de cada una.

Por un lado tenemos la base de datos, donde guardaremos toda la información. Se ha elegido utilizar MongoDB porque se necesitaba una base de datos NoSQL. Esto se debe a que la información respectiva a los bloques era demasiado compleja y se decidió utilizar el formato JSON para guardarla.

En medio tenemos la parte del servidor web. Para el desarrollo se ha creado una API REST implementada en Spring. Hemos creado un proyecto maven con las dependencias necesarias para usar MongoDB, y gracias a ello tenemos Mongo Repository, que nos sirve para conectar la base de datos con la aplicación y para poder hacer todo el CRUD de los datos. Por otro lado, tenemos Node.js, que se usa

para servir el servidor en local en nuestro equipo, utilizando el frontend de la aplicación y un navegador web.

Por último, está la parte del cliente. Todo el frontend de la aplicación ha sido desarrollado utilizando AngularJS. Existen una serie de controladores que conectan con la API REST utilizando los métodos de petición http (GET, POST, etc..) para guardar, leer y modificar datos. También están las vistas, que son las interfaces gráficas con las que interactúan los usuarios. Finalmente está el navegador web, que nos servirá para conectarnos a través de URLs con la aplicación.

Para entender de forma más clara cómo funcionan todas las partes entre sí, a continuación se explica cómo sería el proceso a seguir para llegar de un extremo a otro de la aplicación. Una vez esté la base de datos arrancada, la aplicación REST ejecutándose en el servidor y Node.js la esté sirviendo, desde el navegador se escribirá la URL de la aplicación (http://localhost:8000), que nos mostrará una vista inicial. El usuario interactúa con la vista, que estará vinculada a un controlador, a través del cual, según las órdenes de este usuario, se comunicará mediante una petición http a la API REST. La API utilizará un repositorio que accederá a la base de datos para hacer las operaciones oportunas con los datos (leer, guardar, etc.).

3.2 Diagrama de clases

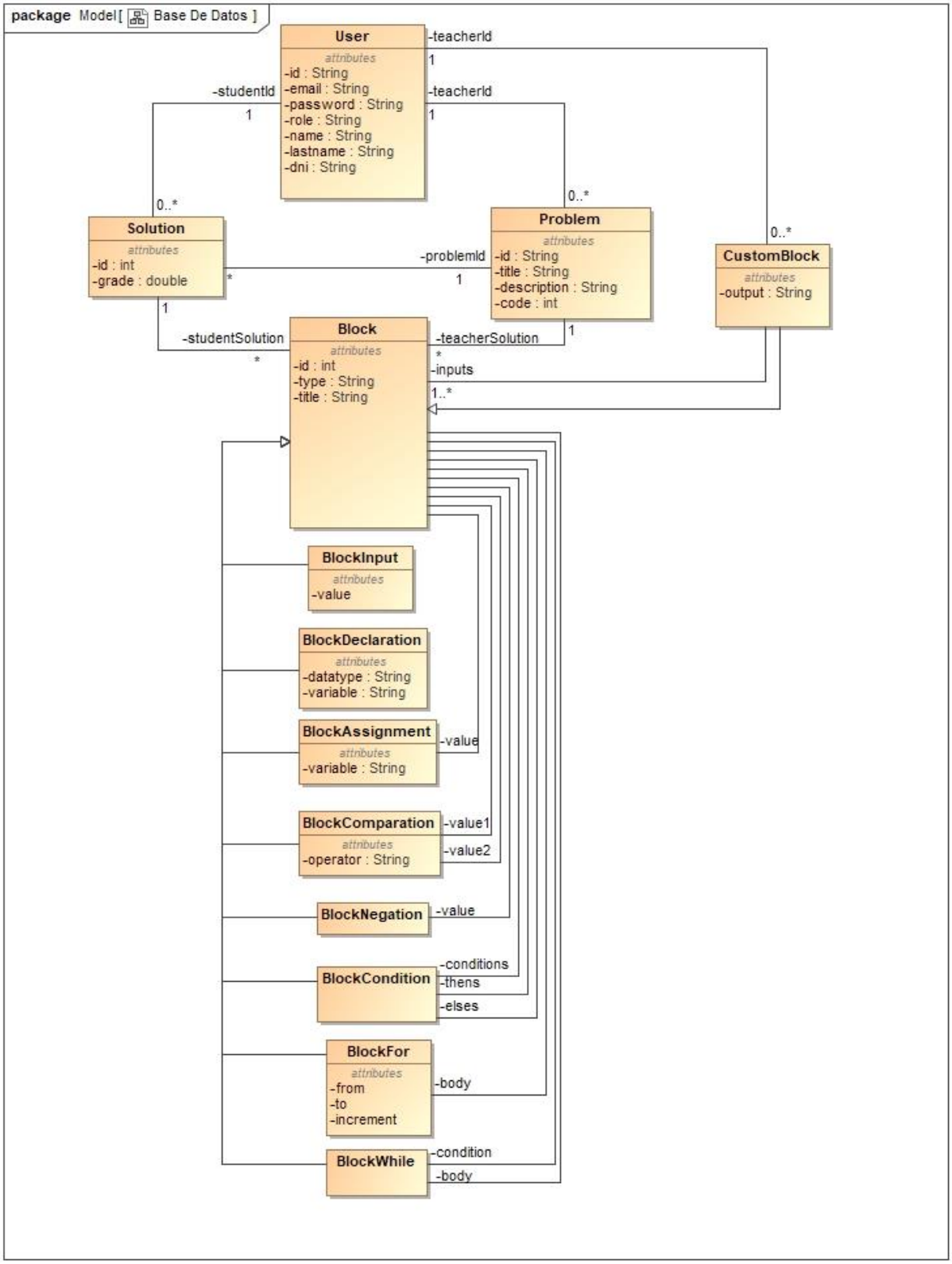

Figura 12 - Diagrama de clases

Para tener una idea más clara de la estructura de la aplicación, se ha creado un diagrama de clases UML donde se muestran las principales entidades y la

interacción entre ellas. Este diagrama ha sido pensado para utilizarlo en una base de datos NoSQL, como en este caso, que se ha utilizado MongoDB.

A continuación, se explicarán con detalle las clases del diagrama:

- **User:** es la entidad que define a un usuario de la aplicación. Sus atributos más importantes son el *email* y la *contraseña*, que le sirven para identificarse, además del *rol*, que puede tomar el valor de profesor o alumno. Dependiendo del rol, la aplicación tendrá un aspecto distinto para cada tipo de usuario. Además, todos los usuarios pueden tener un *nombre*, *apellido* y *dni.*
- **Problem:** es la entidad que representa un problema creado por un profesor. Sus atributos son un *título*, una *descripción* a modo de enunciado del problema, y un *código* que servirá para facilitárselo a los alumnos que vayan a resolver ese problema. El código es único, no puede haber dos problemas con el mismo código. Cuenta también con un atributo *teacherId,* que hace referencia al profesor que lo ha creado. También contiene un atributo llamado *teacherSolution,* que no es más que una estructura que contiene el conjunto de bloques que componen la solución propuesta por el profesor.
- **Solution:** representa una solución a un problema creada por un alumno. Tiene un atributo *nota*, que guarda la calificación obtenida una vez el profesor ha corregido el ejercicio, y otro atributo *studentId*, que guarda el id del alumno que ha creado la solución. Al igual que la entidad Problem, también tiene una estructura para guardar los bloques de la solución, llamada *studentSolution*. Y por último tiene un atributo para guardar el problema al que hace referencia dicha solución, llamado *problemId*.
- **Block:** es una entidad muy general de la que heredan todos los tipos de bloque que existen en la aplicación, ya que todos comparten una serie de atributos. Esta entidad no se encuentra como tal en la base de datos, ya que los bloques son estructuras de tipo JSON que se van a ir anidando unos con otros. Para crearse se utilizan unas plantillas que veremos en el apartado 4 de este documento que trata sobre implementación. Estas estructuras JSON serán las que se guardan en los atributos *teacherSolution* y *studentSolution* que se han explicado anteriormente en las entidades problem y solution. Los atributos que

contienen todos los bloques son *tipo* y *título*, que hacen referencia al tipo de bloque ya sea un if, una asignación, etc...

- **CustomBlock:** son los bloques personalizados que crean los profesores. Cuenta con un *título* a modo de nombre para ese bloque, el tipo de salida *output,* y el conjunto de bloques que contiene, *inputs.* Cada input tiene un tipo (entero, cadena de caracteres, booleano, etc.) y el bloque en sí. Tiene un *teacherId* para saber el id del profesor que lo ha creado.
- **BlockInput:** es un tipo de bloque muy simple que solo contiene un valor, que puede ser de tipo numérico (int y double), cadena de caracteres (string), booleano o una variable genérica (var).
- **BlockDeclaration:** es un bloque que sirve para declarar variables. Consiste en elegir un tipo para la variable, *datatype*, y en un nombre para la variable, que se guarda en el atributo *variable*.
- **BlockAssignment:** es un bloque que se utiliza para asignar valores a las variables. Tiene la *variable* a la que hace referencia, y la relación *value* es la estructura donde se guarda el valor asignado a la variable (es una lista con solo un elemento, que será un bloque de tipo input).
- **BlockComparation**: es un bloque que sirve para comparar dos bloques. Estos bloques se guardan en *value1* y *value2*, y también cuenta con un atributo *operator* que guarda el operador de comparación (==, >=, ...).
- **BlockNegation**: es un bloque que se utiliza para negar otro bloque. En *value* se guarda el bloque que queremos negar.
- **BlockCondition**: es un bloque que simula una estructura condicional (if) en programación. Cuenta con una relación de bloques *condition* que guarda el bloque o los bloques que componen la condición del if, otra relación llamada *thens* que contiene los bloques que van después de la condición si ésta es cierta, y *elses,* que contiene los bloques que irían después de no cumplirse la condición.
- **BlockFor**: es un bloque que simula una estructura de control tipo for en programación. Tiene los atributos *from*, *to* e *increment* que tomarán los valores o variables desde donde queremos que empiece, hasta donde va a acabar, y el incremento que se hará en cada iteración. Cuenta con la estructura *body* donde se guardan los bloques que forman el cuerpo de bucle.

- **BlockWhile**: es un bloque que simula la estructura de control tipo while en programación. Tiene un bloque o conjunto de bloques que forman la condición, *condition*, y otro conjunto de bloques que forman el cuerpo, *body*.

Como puede observarse en el diagrama, todos los bloques heredan de la clase Block, ya que todos contienen los atributos *type* y *title,* y además pueden contener bloques de cualquier tipo, por eso están relacionados con Block con el nombre de las estructuras que guardan estos bloques. Por ejemplo, en el caso de BlockWhile, vemos que tiene una relación con Block que recibe el nombre de body, donde se guardan los bloques que forman el cuerpo del while.

Como usamos una base de datos NoSQL, aquí no hay claves foráneas ni nada por el estilo, simplemente cada entidad guarda como un atributo normal las relaciones con otras entidades. Por ejemplo, la entidad Problem está relacionada con la entidad User, a través de teacherId, esto quiere decir que cada problema tendrá un atributo que guarda el número de id del profesor que lo creó, pero no está haciendo una referencia directamente a la id del profesor. Si ésta id cambia, la teacherId de problema no va a cambiar, pues no es una base de datos relacional.

3.3 Diagrama de navegación del rol de profesor

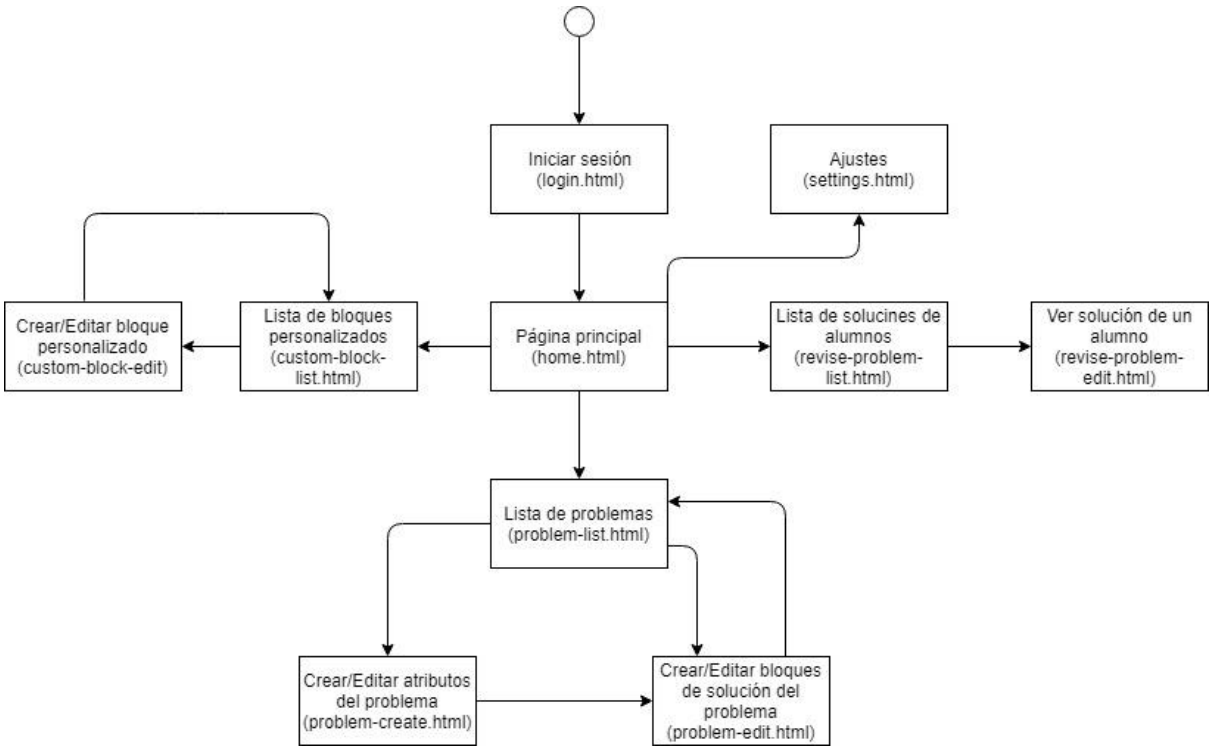

Figura 13 - Diagrama de navegación del profesor

En este diagrama se muestra cuáles son los caminos que puede recorrer un usuario de la aplicación, con el rol de profesor, por las diferentes vistas. En él se plasma la estructura de la navegación por la aplicación.

La primera página a la que se accede es la de inicio de sesión, donde se le pedirán las credenciales al usuario. En este caso va a ser un profesor. Tras identificarse, la aplicación se dirige a la página principal, donde se muestran las distintas opciones que puede realizar un profesor:

- Si decide ir a los *bloques personalizados*, se le redirigirá a la página con la lista de bloques personalizados ya creados. Si decide editar uno o crear uno nuevo, se le mostrará la siguiente página de crear/editar un bloque y tras darle a finalizar, la aplicación volverá a redirigirle a la lista de bloques.
- Si decide ir a los *problemas,* se le redirigirá a la página con la lista de problemas ya creados. Si decide crear uno nuevo, se le mostrará la siguiente página de crear atributos de un problema y tras darle a siguiente, pasará a la página para crear bloques de solución del problema. Después de pinchar en finalizar, la aplicación volverá a redirigirle a la lista de problemas. También puede elegir si editar sus atributos, o editar los bloques directamente.
- Si decide *revisar problemas,* se le redirigirá a la lista de soluciones, y tras pinchar en una, aparecerá la página para mostrar esa solución y ponerle una nota.

La aplicación cuenta con un header o cabecera, desde la cual siempre podrá volver a la página principal, si pincha sobre la pestaña *home.* En la cabecera hay una segunda pestaña llamada *settings*, la cual redirigirá al profesor a una página de configuración donde podrá modificar sus datos personales o su contraseña.

3.4 Diagrama de actividad del algoritmo de corrección

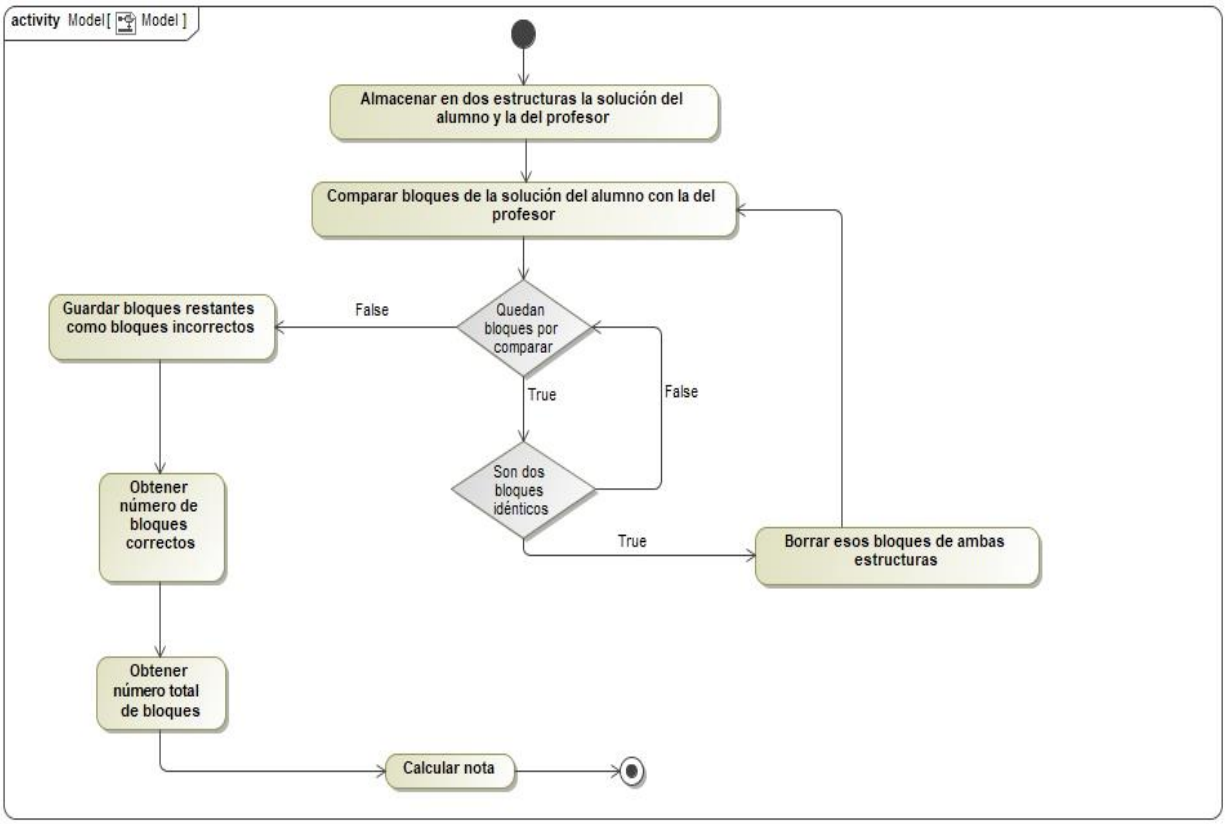

Figura 14 - Diagrama de actividad del algoritmo de corrección.

El diagrama de la figura muestra el flujo del algoritmo de corrección. El algoritmo se explica detalladamente en el apartado 4.2 que trata sobre la implementación del frontend, pero principalmente se trata de un algoritmo que compara bloque por bloque de las soluciones de alumno y profesor, contando como correcto los bloques idénticos, e incorrectos los bloques que sobren o falten en la solución del alumno, además de los que tengan algún atributo erróneo. Tras realizar la comparación se realizan una serie de cálculos para obtener la nota.

4. Implementación

4.1 Backend

A continuación, se muestra la estructura de los ficheros que componen el backend de la aplicación. Se ha utilizado el entorno de programación Eclipse, utilizando Spring Tools 4. Como se puede observar, para identificar al proyecto le hemos puesto el nombre rainblocks, haciendo referencia a la palabra en inglés *rainbow*, que significa arcoíris, ya que los bloques de la aplicación tienen diversos colores que hacen recordar a un arcoíris.

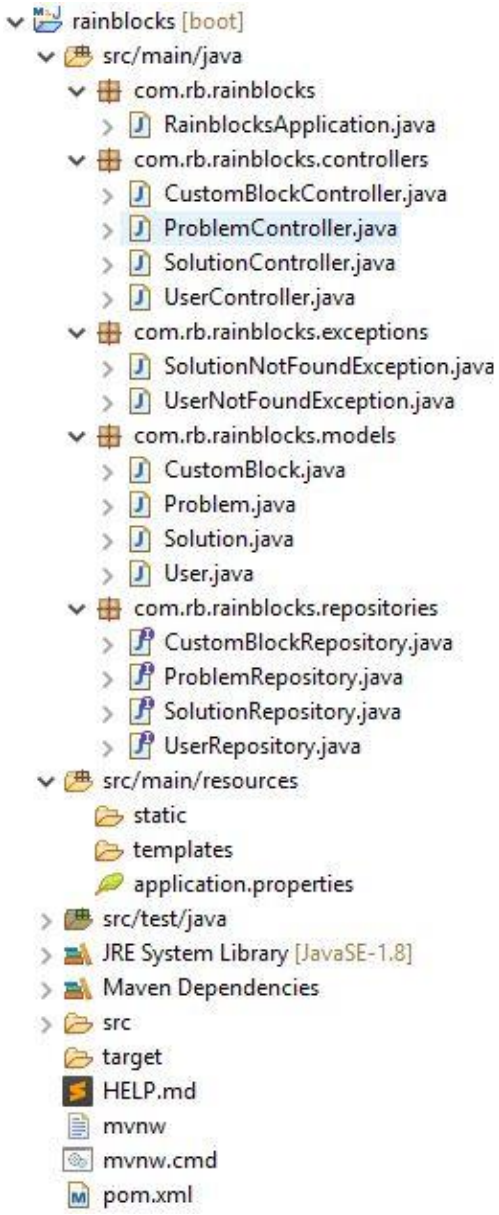

Figura 15 - Estructura de ficheros de backend

Como puede observarse, se trata de un proyecto maven que contiene un fichero pom.xml. Este fichero, además de la versión de Spring y el nombre del proyecto, también contiene las dependencias usadas:

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-mongodb</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>

</dependencies>
```

Figura 16 - Dependencias pom.xml

Estas dependencias son **Spring Boot Data MongoDB Starter** para utilizar la base de datos orientada a documentos MongoDB y Spring Data MongoDB, **Spring Boot Web Starter** para la creación de aplicaciones web, incluidas las aplicaciones REST, que utilizan Spring MVC, y **Spring Boot Test Starter** para probar aplicaciones Spring Boot con bibliotecas que incluyen JUnit, Hamcrest y Mockito.

Lo siguiente importante es el directorio “src/main/java” que es donde está todo lo que ha sido implementado desde cero. Tenemos el paquete “com.rb.rainblocks” con sus diferentes subcarpetas y el ejecutable de la aplicación *RainblocksApplication.java* , que no es más que una simple línea de código que ejecuta el proyecto.

RainblocksApplication.java

```java
package com.rb.rainblocks;

import org.springframework.boot.SpringApplication;

@SpringBootApplication
public class RainblocksApplication {

    public static void main(String[] args) {
        SpringApplication.run(RainblocksApplication.class, args);

    }

}

```

Figura 17 - RainblocksApplication.java

La subcarpeta **models** contiene las entidades que se usan para crear objetos (o documentos en este caso) en la base de datos. Todas comienzan con la etiqueta @Document porque es como se deben etiquetar al utilizar MongoDB al ser una base de datos orientado a documentos de código abierto, además todas tiene un id etiquetado con @Id. A continuación, se muestra el código del fichero *User.java*, aunque el resto (*CustomBlock.java, Problem.java* y *Solution.java*) tienen la misma estructura. La estructura no es más que la declaración de sus atributos, el constructor y los getters y setters de los atributos.

```java
package com.rb.rainblocks.models;

import org.bson.types.ObjectId;

@Document
public class User {

    @Id
    public ObjectId _id;

    public String email;
    public String password;
    public String role;

    public String name;
    public String lastname;
    public String dni;


    public User() {
        super();
        // TODO Auto-generated constructor stub
    }



    public User(ObjectId _id, String email, String password, String role, String name, String lastname, String dni) {
        super();
        this._id = _id;
        this.email = email;
        this.password = password;
        this.role = role;
        this.name = name;
        this.lastname = lastname;
        this.dni = dni;
    }



    public String get_id() {
        return _id.toHexString();
    }

    public void set_id(ObjectId _id) {
        this._id = _id;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
```

Figura 18 - User.java

La subcarpeta **repositories** contiene interfaces que extienden de *MongoRepository*. Estas interfaces consultan los documentos de las diferentes entidades en la base de datos para extraer o guardar información, es decir, son un conector entre los modelos y mongoDB. A continuación, se muestra por ejemplo *ProblemRepository* que extiende de la interfaz de *MongoRepository* y conecta el tipo de valores y el id con el que trabaja: *Problem* y *String*. Esta interfaz viene con muchas operaciones, incluidas las de un CRUD estándar (crear-leer-actualizar-eliminar). Por defecto viene *findBy_id* que devuelve el problema que tiene el Id que se le pasa por parámetro. Se pueden definir otras consultas según sea necesario, simplemente declarando su método y escribiendo el nombre exacto del atributo por el que se

pretende buscar. En este caso, se ha agregado *findByCode*, que esencialmente busca documentos del tipo *Problem* y encuentra el que coincide con el *code*. También tiene *findByTeacherId* para encontrar una lista de profesores por *teacherId*.

```
1  package com.rb.rainblocks.repositories;
2
3  import java.util.List;
9
10 public interface ProblemRepository extends MongoRepository<Problem, String>{
11     Problem findBy_id(ObjectId _id);
12     Problem findByCode(String code);
13     List<Problem> findByTeacherId(String teacherId);
14 }
15
```

Figura 19 - ProblemRepository.java

En una aplicación Java típica, hay que escribir una clase que implementa ProblemRepository y crear las consultas manualmente. Lo que hace que Spring Data MongoDB sea tan útil es el hecho de que no hay que crear esta implementación. Spring Data MongoDB lo crea sobre la marcha cuando ejecuta la aplicación.

La subcarpeta **controllers** contiene los controladores de cada una de las entidades. Cada fichero contiene los métodos básicos para realizar las operaciones REST y todos están etiquetados con @RestController. Cada controller utiliza una instancia del correspondiente repositorio etiquetada con @Autowired para hacer las consultas a la base de datos. Como antes hemos visto el archivo repository de Problem, vamos a echarle un vistazo a *ProblemController.java.*

```java
package com.rb.rainblocks.controllers;

import java.util.ArrayList;

@RestController
@RequestMapping("/problems")
public class ProblemController {

    @Autowired
    private ProblemRepository repository;

    @CrossOrigin(origins = "http://localhost:8000")
    @RequestMapping(value = "/", method = RequestMethod.GET)
    public List<Problem> getAllProblems() {
        return repository.findAll();
    }

    @CrossOrigin(origins = "http://localhost:8000")
    @RequestMapping(value = "/{id}", method = RequestMethod.GET)
    public Problem getProblemById(@PathVariable("id") ObjectId id) {
        return repository.findBy_id(id);
    }

    @CrossOrigin(origins = "http://localhost:8000")
    @RequestMapping(value = "/byTeacher/{teacherId}", method = RequestMethod.GET)
    public List<Problem> getProblemByTeacherId(@PathVariable("teacherId") String teacherId) {
        return repository.findByTeacherId(teacherId);
    }

    @CrossOrigin(origins = "http://localhost:8000")
    @RequestMapping(value = "/byCode/{code}", method = RequestMethod.GET)
    public Problem getProblemByCode(@PathVariable("code") String code) {
        return repository.findByCode(code);
    }

    @CrossOrigin(origins = "http://localhost:8000")
    @RequestMapping(value = "/{id}", method = RequestMethod.PUT)
    public void modifyProblemById(@PathVariable("id") ObjectId id, @Valid @RequestBody Problem problem) {
        problem.set_id(id);
        repository.save(problem);
    }

    @CrossOrigin(origins = "http://localhost:8000")
    @RequestMapping(value = "/", method = RequestMethod.POST)
    public Problem createProblem(@Valid @RequestBody Problem problem) {
        problem.set_id(ObjectId.get());
        List<Object> ts = new ArrayList<>();
        problem.setTeacherSolution(ts);
        repository.save(problem);
        return problem;
    }

    @CrossOrigin(origins = "http://localhost:8000")
    @RequestMapping(value = "/{id}", method = RequestMethod.DELETE)
    public void deleteProblem(@PathVariable ObjectId id) {
        repository.delete(repository.findBy_id(id));
    }

}
```

Figura 20 - ProblemController.java

Por todo el fichero aparece la anotación @RequestMapping, que especifica la URL base que el controlador manejará, por lo que cualquier solicitud al host que comience con "/problems" se dirigirá a este controlador. También aparece la etiqueta @CrossOrigin, que se encarga de habilitar CORS, el cual es un mecanismo que utiliza cabeceras HTTP adicionales para permitir que un user agent obtenga permiso para acceder a recursos seleccionados desde un servidor, en un origen distinto (dominio)

al que pertenece. En este caso solo permitimos que http://localhost:8000 envíe solicitudes de origen cruzado.

La primera función que aparece en la línea 29 se llama *getAllProblems()* y lo que hace es, como su propio nombre indica, devolver una respuesta con todos los problemas que haya almacenados en la base de datos. Puede observarse como en el RequestMapping aparece un parámetro llamado *method* que contiene el valor *RequestMethod.GET*, haciendo referencia a que es una solicitud http de tipo GET para leer información de la base de datos.

Otra función algo más compleja es la de la línea 41, *getProblemByTeacherId()*. El Request Mapping en este caso tiene como value la url “/byTeacher/{teacherId}” para diferenciarla del resto de peticiones. El parámetro teacherId es una variable que se le pasa en la url desde el frontend, que será necesaria para luego utilizarla en el repository a través del método que hemos comentado antes, *findByTeacherId(teacherId).* En este caso se devolverá una lista de problemas que contengan esa Id de profesor.

Además de las solicitudes GET, también hay funciones para las demás. POST para crear, utilizando repository.save(). PUT para modificar, tambíen con repository.save(). DELETE para eliminar, utilizando repository.delete().

El resto de controladores tienen una estructura similar a la de ProblemController, cada uno con sus funciones particulares que han sido creadas según las necesidades que han surgido.

La subcarpeta **exceptions** contiene dos ficheros que se usan como excepción. *SolutionNotFoundException* se utiliza cuando se intenta hacer una solicitud GET de una solución a partir del id del problema al que hace referencia, y del id del alumno que la ha creado, y ésta no existe. UserNotFoundException se usa cuando se intenta acceder a la aplicación con un email o una contraseña incorrectos. Ambas excepciones heredan de RuntimeException.

El último fichero que se va a mencionar es **application.properties**, ubicado en la carpeta "src/main/resources".. No es más que un fichero donde está la información necesaria para decirle a Spring la información de conexión para nuestra base de datos de MongoDB. Esta es toda la información que Spring necesitará para conectarse a la base de datos.

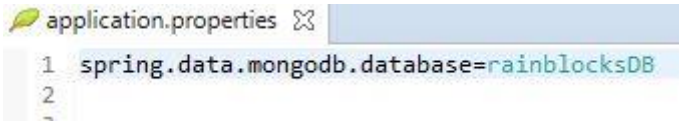

Figura 21 - application.properties

4.2 Frontend

En este apartado se comentará la implementación de la parte más visual de la aplicación, que ha supuesto el mayor porcentaje de ésta. El código está escrito en JavaScript, AngularJS, HTML y CSS. A continuación, se muestra la estructura de ficheros que componen el frontend.

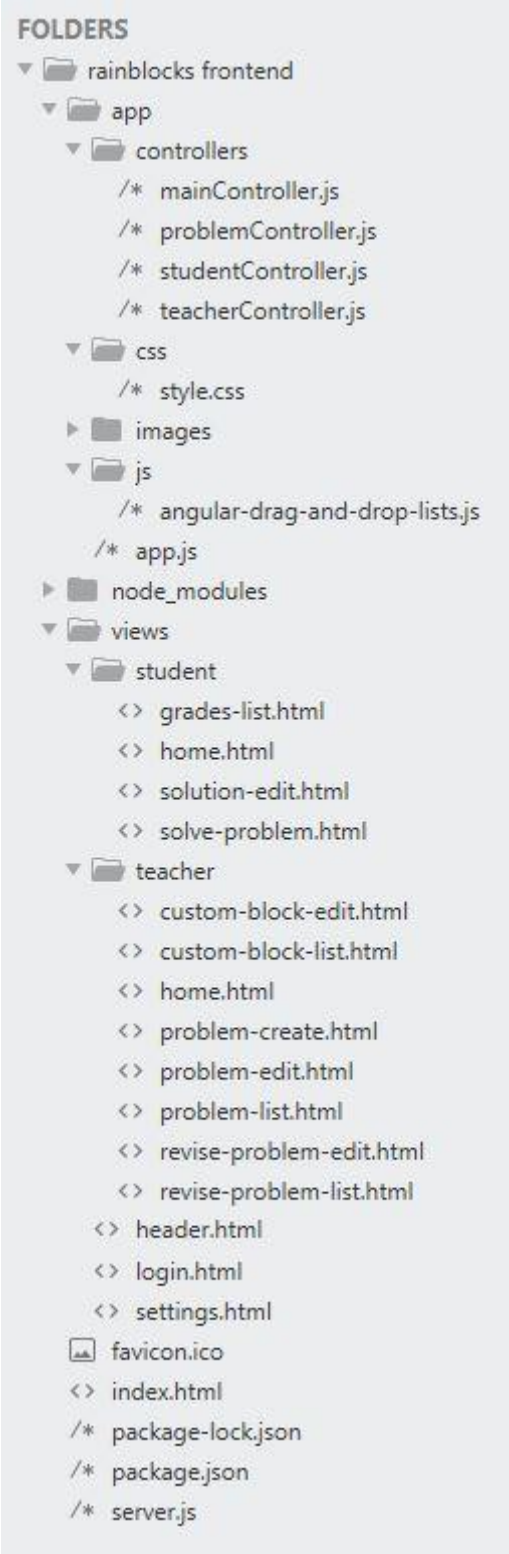

Figura 22 - Estructura de fichero de frontend

Existen dos carpetas principales, **app** y **views**. En **app** se encuentran los controladores, el fichero de estilo css, las imágenes utilizadas, los javascript importados y el fichero *app.js* (que pertenece a la parte común de la aplicación y

corresponde al otro miembro del grupo). Este fichero, app.js, es bastante importante porque se encarga de instanciar la aplicación, de importar las librerías, de establecer las rutas y de asociar vistas con controladores. En **views** se encuentran todas las vistas en formato HTML. Fuera de estas carpetas están los ficheros de Node.js y el *index.html.*

App

Dentro de la carpeta **app** se encuentra la carpeta controllers, que contiene cuatro ficheros que actúan de controladores en la aplicación. El fichero *mainController.js* se encarga de operaciones generales como del inicio y cierre de sesión de los usuarios en la aplicación, del cambio de contraseña y de la actualización de los datos personales de cada usuario. El fichero *studentController.js* pertenece al rol de estudiante, así que pertenece al otro miembro del grupo explicarlo, pero se da por entendido que realiza operaciones que sólo tienen que ver con el estudiante. El fichero *problemController.js* es común a los dos roles y en él es donde se encuentra todo lo que tiene que ver con la composición de bloques para crear tanto problemas como soluciones, incluido el algoritmo de corrección de la aplicación. Y el fichero *teacherController.js* realiza las operaciones que tienen que ver con el rol de profesor.

problemController.js

El fichero *problemController.js* contiene lo relacionado con los bloques, incluso su creación desde cero. Para entender mejor a qué nos referimos, recordamos que la entidad *block* no se encuentra como tal en la base de datos, ya que los bloques son estructuras de tipo JSON que se van a ir anidando unos con otros. Para crearse se utilizan unas plantillas, que están implementadas en este fichero.

```
1  app.controller("ProblemController", function($scope, $rootScope, $route, $routeParams, $location, $http) {
2      $scope.$route = $route;
3      $scope.$location = $location;
4      $scope.$routeParams = $routeParams;
5
6      var currentId = $routeParams.id;
7
8      $scope.types = $rootScope.variableTypes;
9      $scope.booleantypes = ['true','false'];
10     $scope.operators = ['&&','||', '==', '!=', '<=', '<', '>=', '>'];
11     $scope.variables = [];
12     $scope.templates = [
13         {title:"Inputs", icon:"keyboard", id:"inputTemplates",
14         blocks:[{type: "inputboolean", title: "boolean", id:1, value: ""},
15                 {type: "inputvariable", title: "variable", id:2, value: ""},
16                 {type: "inputstring", title: "string", id:3, value: ""},
17                 {type: "inputnumeric", title: "numeric", id:4, value: ""},
18         ]},
19         {title:"Basics", icon:"pen", id:"basicTemplates",
20         blocks:[{type: "declaration", title: "declaration",  id:5, datatype: "", variable:""},
21                 {type: "assignment", title: "assignment",  id:6,  variable: "", value:[]},
22         ]},
23         {title:"Logic structures", icon:"not-equal", id:"logicTemplates",
24         blocks:[{type: "comparation", title: "compare", id:7, value1:[], operator:"", value2:[]},
25                 {type: "negation", title: "negate",  id:8, value:[]},
26         ]},
27         {title:"Control structures", icon:"project-diagram", id:"controlTemplates",
28         blocks:[{type: "conditional", title: "if",  id:9, condition: [], thens:[], elses:[] },
29                 {type: "loopfor", title: "for",  id:10,  from: "", to: "", increment: "", body:[]},
30                 {type: "loopwhile", title: "while",  id:11,  condition: [], body:[]},
31         ]},
32         {title:"Customs", icon:"star", id:"customTemplates",
33         blocks:[]},
34     ];
```

Figura 23 - Plantillas de bloques en problemController.js

En la línea 12 de la figura 23 tenemos una variable llamada *$scope.templates*, que contiene la estructura en formato JSON de todos los tipos de bloques. Como se puede observar, dentro de templates hemos separado los bloques en cinco tipos generales:

- *Inputs* contiene los bloques que simplemente sirven para introducir valores. Se trata de cuatro bloques, **inputboolean** para variables booleanas, **inputstring** para cadenas de caracteres, **inputnumeric** para números e **inputvariable** para utilizar cualquier variable creada en el problema.

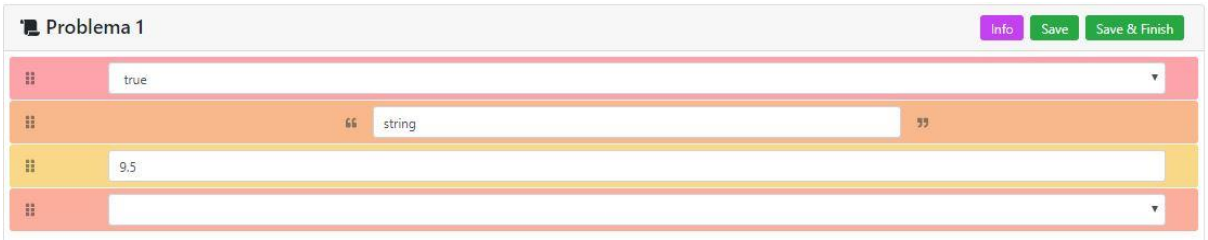

Figura 24 - Plantillas de bloques Inputs

- *Basics* contiene el bloque **declaración** y el bloque **asignación**. El bloque declaración sirve para declarar variables, eligiendo el tipo del que queramos que sea desde el menú desplegable, y escribiendo el nombre de la variable en

el campo de texto. El bloque asignación sirve para darle valor a una variable, eligiendo la variable desde el menú desplegable, y arrastrando un bloque de tipo input al hueco de la derecha.

Figura 25 - Plantillas de bloques Basics

- *Logic structures* contiene un bloque de **comparación** y otro de **negación**. El bloque comparación sirve para comparar dos bloques, y el de negación para negarlo. En las cajas blancas es donde van situados los bloques, y en el caso del bloque comparación, hay un desplegable con los distintos operadores de comparación (==, >=, <=, ect…).

Figura 26 - Plantillas de bloques Logic structures

- *Control structures* contiene el bloque **if**, el bloque **for** y el bloque **while**. El bloque if cuenta con una primera caja blanca donde va situado el bloque que actúe como condición del if. Las dos siguientes cajas corresponden al cuerpo del if, una para el caso en el que se cumpla la condición (then) y otra para el caso contrario (else). El bloque for contiene en primer lugar dos desplegables, uno para elegir desde donde empieza el bucle, y otro hasta donde debe llegar (from - to), en segundo lugar, un campo para indicar el incremento en cada iteraccion del bucle, y por último contiene una caja donde irán situados los bloques de formen el cuerpo del bucle. El bloque while cuenta con una caja para la condición y otra para el cuerpo.

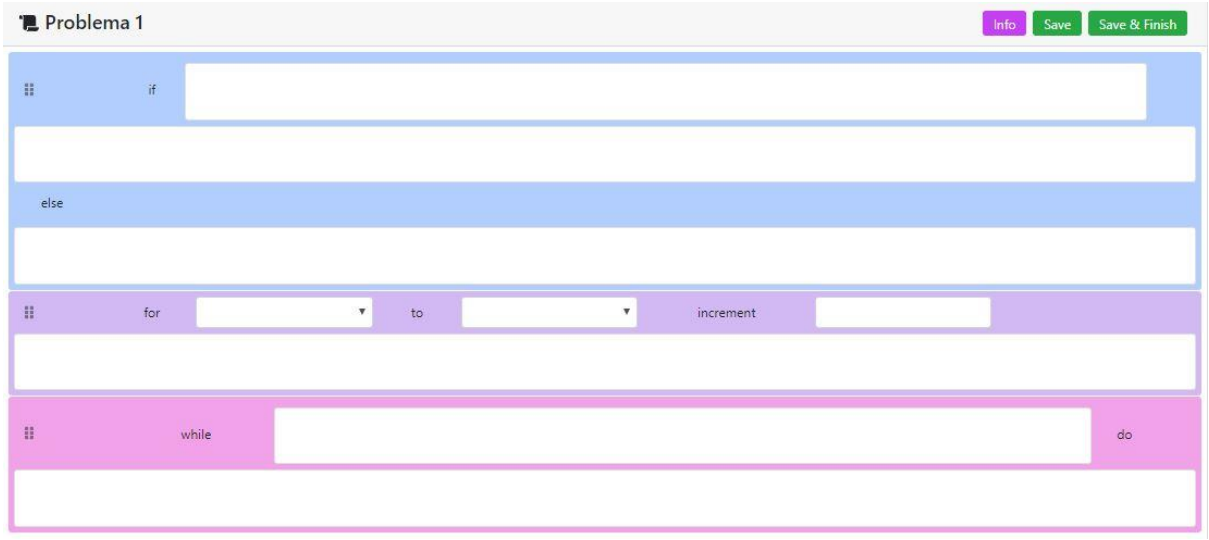

Figura 27 - Plantillas de bloques Control structures

- *Customs* contendrá los bloques personalizados una vez creados por el profesor. Estos tendrán tantas cajas de bloques como inputs le haya indicado el profesor.

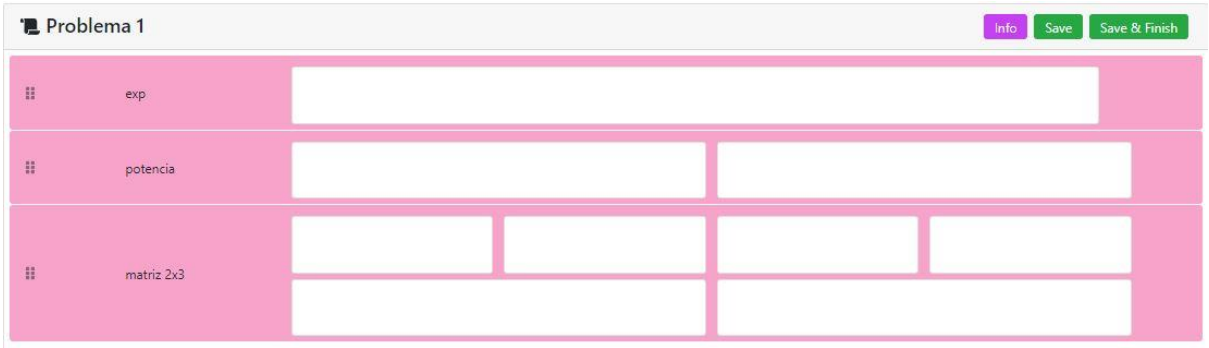

Figura 28 - Plantillas de bloques Customs

El fichero *problemController.js* también contiene las operaciones para guardar un problema, guardar una solución del estudiante y el algoritmo de corrección, que se explicará más adelante.

teacherController.js

Vamor a ver un ejemplo para cada operación relacionada con el sistema REST, son cuatro: POST (crear), GET (leer y consultar), PUT (editar) y DELETE (eliminar). En *teacherController.js* tenemos la función *updateProblem()*, que lo que hace es crear un problema si éste no existía, o actualizar sus valores si ya existía anteriormente. A continuación, vemos cómo se realizan las operaciones POST y PUT en AngularJS.

```
29      $scope.updateProblem = function(problem){
30          problem.teacherId = $rootScope.userId;
31          if(angular.isUndefined(currentProblemId)){
32              $http.post("http://localhost:8080/problems/", problem).then(
33              function successCallback(response) {
34                  $location.path("/teacher/edit-solution/"+ response.data._id);
35              },
36              function errorCallback(response) {
37                  console.log(response.data);
38                  $rootScope.problemCodeError = "There is already a problem using that code"
39              }
40              );
41          }else{
42              $http.put("http://localhost:8080/problems/"+currentProblemId, problem).then(
43              function successCallback(response) {
44                  $location.path("/teacher/edit-solution/"+ currentProblemId);
45              },
46              function errorCallback(response) {
47                  console.log(response.data);
48                  $rootScope.problemCodeError = "There is already a problem using that code"
49              }
50              );
51          }
52      }
```

Figura 29 - Fichero teacherController.js, función updateProblem()

Primero se comprueba si el id de problema existe:

- En caso de que no exista pasamos a crearlo, utilizando **$http**, que es un servicio de AngularJS para leer datos desde servidores remotos. En la línea 32 de la figura 29 aparece *$http.post* donde se le pasa como parámetro la ruta asociada al controlador de *Problem* en el servidor, y el objeto problem que queremos crear. Tras esto esperamos una respuesta por parte del servidor. Si es correcta, se entra por la línea 33 que contiene la función *successCallback()*, y si hay algún error, entramos por la línea 36 con la función *errorCallback().*
- En caso de que el problema ya exista, utilizamos esta vez *$http.put*, pasando como argumento la ruta asociada a la función de editar problema en el servidor y el objeto problem en sí.

En el mismo fichero, *teacherController.js*, hay otras dos funciones que realizan un GET y un DELETE. La primera es *updateProblemList()* que se encarga de leer todos los problemas creados por un profesor concreto. La segunda es *deleteProblem()* que lo que hace es, como su propio nombre indica, eliminar un problema de la base de datos.

```
7     $scope.updateProblemList = function () {
8         $http.get("http://localhost:8080/problems/byTeacher/"+$rootScope.userId).then(
9         function successCallback(response) {
10            $scope.problemList = response.data;
11        },
12        function errorCallback(response) {
13            console.log(response.statusText);
14        }
15        );
16    }
17
18    $scope.deleteProblem = function(id){
19        if (confirm('Are you sure you want to delete this?')) {
20            $http.delete("http://localhost:8080/problems/"+id).then(
21                function successCallback(response) {
22                    $scope.updateProblemList();
23                }
24            );
25        }
26    }
```

Figura 30 - Fichero teacherController.js, funciones para get y delete

Como puede verse, se vuelve a utilizar el servicio $http, esta vez con las funciones *$http.get* y *$http.delete*. La estructura es similar a post y put, explicadas anteriormente.

Algoritmo de corrección

En el fichero *problemController.js* se encuentra el algoritmo de corrección, *reviseSolution()*. Este algoritmo se basa en la comparación de las estructuras JSON de la solución propuesta por el profesor y la solución que realiza el alumno. En la comparación de las dos estructuras, se van a tener en cuenta los bloques que faltan (*missingBlocks*) y los bloques que sobran (*excessiveBlocks*), incluyendo los bloques que estén incorrectos.

```
221   $scope.reviseSolution = function(){
222
223       var problemAux = [];
224       angular.copy($scope.problem.teacherSolution, problemAux);
225       var solutionAux = [];
226       angular.copy($scope.solution.studentSolution, solutionAux);
227
228       $scope.missingBlocks = [];
229       $scope.excessiveBlocks = [];
230
231       var numCorrectBlocks = $scope.compareBlocks(problemAux, solutionAux);
```

Figura 31 - Fichero problemController.js, algoritmo de corrección I

La comparación se hace bloque por bloque, comparando cada bloque de una de las estructuras con todos los bloques de la otra estructura, hasta encontrar uno idéntico.

Cuando se encuentran dos bloques idénticos, se eliminan ambos de sus estructuras correspondientes. Cuando la comparación termina, los bloques que han quedado en la solución del profesor serán los bloques que faltan, y los bloques que han quedado en la solución del alumno serán los bloques que sobran. Este algoritmo de comparación es recursivo, puesto que los bloques pueden contener estructuras de bloques dentro de sí, por ejemplo, el caso de un bloque for, que cuenta con un *body* donde se encuentra todos los bloques que hay dentro del cuerpo del bucle for.

```
253     $scope.compareBlocks = function(problem, solution){
254         var numCorrectBlocks = 0;
255         //var allChildrenCorrect = true;
256         for(var i = 0; i < problem.length; i++) {
257             for(var j = 0; j < solution.length; j++){
258                 if(angular.equals(problem[i].type, solution[j].type)){
259                     switch (problem[i].type) {
260                         case 'assignment':
261                             numCorrectBlocks += $scope.compareBlocks(problem[i].value, solution[j].value);
262                             break;
263                         case 'comparation':
264                             numCorrectBlocks += $scope.compareBlocks(problem[i].value1, solution[j].value1);
265                             numCorrectBlocks += $scope.compareBlocks(problem[i].value2, solution[j].value2);
266                             break;
267                         case 'negation':
268                             numCorrectBlocks += $scope.compareBlocks(problem[i].value, solution[j].value);
269                             break;
270                         case 'conditional':
271                             numCorrectBlocks += $scope.compareBlocks(problem[i].condition, solution[j].condition);
272                             numCorrectBlocks += $scope.compareBlocks(problem[i].thens, solution[j].thens);
273                             numCorrectBlocks += $scope.compareBlocks(problem[i].elses, solution[j].elses);
274                             break;
275                         case 'loopfor':
276                             numCorrectBlocks += $scope.compareBlocks(problem[i].body, solution[j].body);
277                             break;
278                         case 'loopwhile':
279                             numCorrectBlocks += $scope.compareBlocks(problem[i].condition, solution[j].condition);
280                             numCorrectBlocks += $scope.compareBlocks(problem[i].body, solution[j].body);
281                             break;
282                         case 'custom':
283                             numCorrectBlocks += $scope.compareBlocks(problem[i].inputs, solution[j].inputs);
284                             break;
285                         default:
286                     }
287                     if(blockStructureEquals(problem[i],solution[j])){
288                         problem.splice(i, 1);
289                         solution.splice(j, 1);
290                         i--;
291                         j--;
292                         numCorrectBlocks++;
293                         break;
294                     }
295                 }
296             }
297         }
```

```
298         if(problem.length > 0){
299             angular.forEach(problem, function(value,key){
300                 $scope.missingBlocks.push(angular.copy(value));
301             });
302         }
303         if(solution.length > 0){
304             angular.forEach(solution, function(value,key){
305                 $scope.excessiveBlocks.push(angular.copy(value));
306             });
307         }
308         return numCorrectBlocks;
309     }
310
```

Figura 32 - Fichero problemController.js, algoritmo de corrección II

El algoritmo de comparación devuelve un número de bloques que están correctos. Una vez tenemos el número de bloques correctos del alumno, el número

de bloques incorrectos y el número total de bloques que había en la solución del profesor, hacemos una serie de cálculos para obtener la nota, ponderando los bloques de forma equilibrada para que los correctos sumen y los incorrectos resten.

```
231         var numCorrectBlocks = $scope.compareBlocks(problemAux, solutionAux);
232
233         var numTotalBlocks = getNumTotalBlocks($scope.problem.teacherSolution);
234
235         if(numTotalBlocks > 0){
236             var correctBlockValue = 10/numTotalBlocks;
237             var incorrectBlockValue = correctBlockValue/10;
238             var grade = numCorrectBlocks*correctBlockValue -
239             ($scope.missingBlocks.length+$scope.excessiveBlocks.length)*incorrectBlockValue;
240
241             grade = Math.round(grade * 100) / 100;
242             if (grade > 10) {
243                 grade = 10;
244             } else if (grade < 0) {
245                 grade = 0;
246             }
247         }else{
248             var grade = 0;
249         }
250         $scope.suggestedGrade = grade;
251     }
```

Figura 33 - Fichero problemController.js, algoritmo de corrección III

Views

La carpeta **views** contiene todas las vistas de la aplicación, separadas en las de profesor y en las de alumnos, además de las comunes que son *login.html*, *setting.html* y *header.html*. Como se puede observar en la figura 22, dentro de la carpeta *teacher* están las vistas relacionadas con profesor. Están escritas en código HTML y AngularJS.

- **home.html**

  Es la página principal, contiene tres opciones para el profesor: Custom blocks, Problems y Revise problems. Tanto en esta como en el resto de páginas, aparece siempre arriba un encabezado, que corresponde con el fichero header.html, desde el cual siempre podemos volver a home.html, a settings.html o cerrar nuestra sesión en la aplicación.

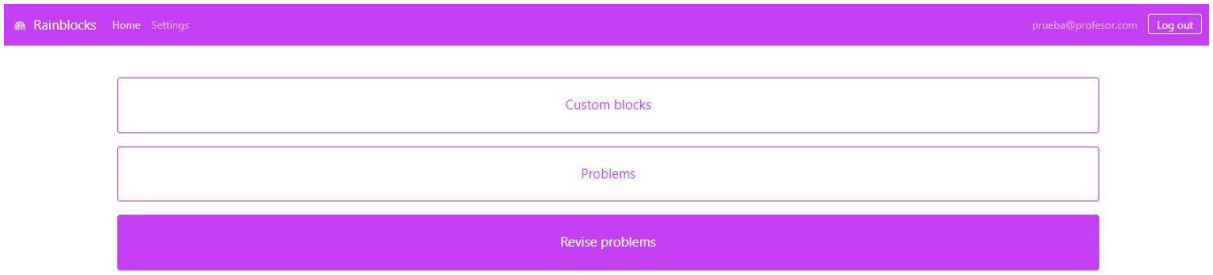

Figura 34. home.html

- **custom-block-list.html**

  Muestra la lista de bloques personalizados del profesor.

Figura 35. custom-block-list.html

- **custom-block-edit.html**

  Es la vista desde donde se crea un bloque personalizado

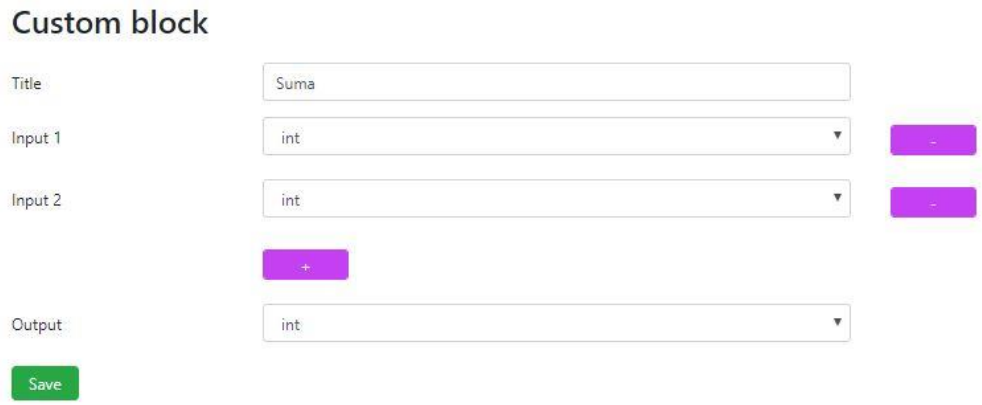

Figura 36. custom-block-edit.html

- **problem-list.html**

  Muestra la lista de problemas creador por el profesor.

Figura 37. problem-list.html

- **problem-create.html**

  Es la vista desde donde se crea o edita los parámetros de un problema: titulo, codigo y descripción.

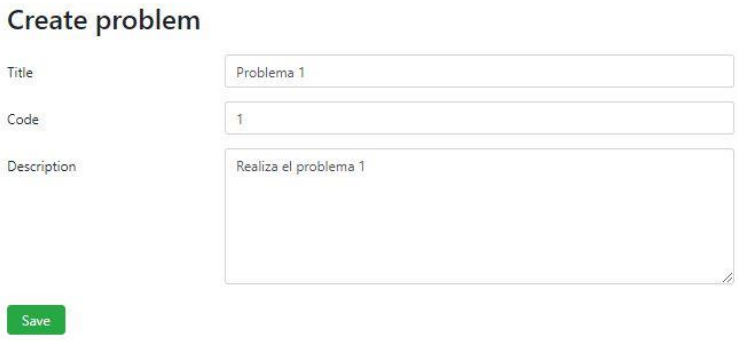

Figura 38. problem-create.html

- **problem-edit.html**

  Es la vista desde donde se crea o edita la solución de un problema.

Figura 39. problem-edit.html

- **revise-problem-list.html**

  Es la vista donde se pueden ver la lista de soluciones pendientes de revisar.

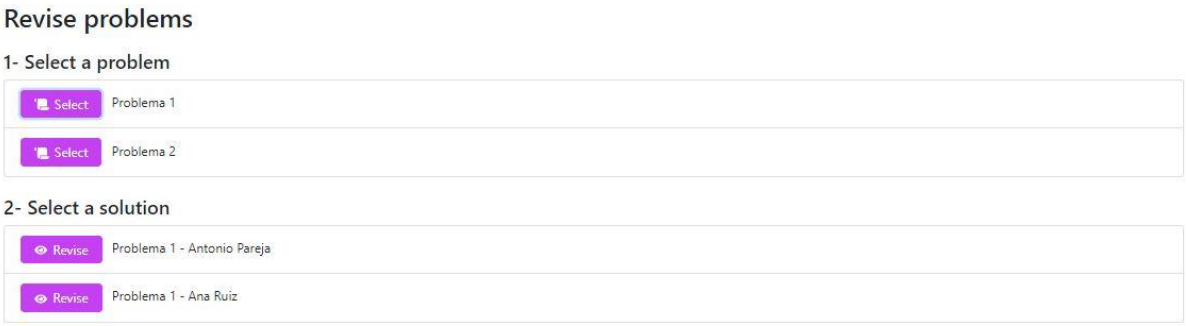

Figura 40. revise-problem-list.html

- **revise-problem-edit.html**

  Es la vista desde donde se visualiza la solución de un alumno, la nota propuesta por el algoritmo de corrección, y donde el profesor puede escribir la nota del problema.

Figura 41. revise-problem-edit.html

Librería Drag & Drop Lists

La librería Drag & Drop Lists es una librería de AngularJS que hemos utilizado para realizar las operaciones que tienen que ver con el arrastre de los bloques. A continuación, se explican las operaciones que hemos utilizado en nuestra aplicación, en este caso en el fichero problem-edit.html.

Figura 42. problem-edit.html

Lo que aparece en la imagen es el código HTML y angular que corresponde con la zona habilitada para arrastrar bloques desde el menú lateral de la izquierda (puede observarse en la Figura 39). En la línea 53, la directiva *dnd-list* sirve para convertir la lista de elementos en una dropzone o zona donde se arrastran bloques. Todo lo que arrastremos y soltemos aquí, se guardará en *problem.teacherSolution*. El atributo *dnd-allowed-types* de la línea 54 sirve para indicar el tipo de elementos que permitimos que pueden arrastrarse y soltarse en la dropzone. En este caso allowedTypesMain es un array que contiene los tipos de los bloques que queremos que se puedan arrastrar en la dropzone principal. El atributo *dnd-dragable* sirve para indicar que el elemento será arrastrable. El atributo *dnd-type* se combina con *dnd-allowed-types*, y en él se indica el tipo del cual es el elemento, para que la aplicación sepa si se puede arrastrar o no. Por último, *dnd-moved* contiene una función que se invocará cuando el elemento esté arrastrándose. En este caso, cada vez que cogemos un elemento y lo empezamos a arrastrar, se eliminará de la lista (función *splice*), aunque en el momento que lo soltemos en la dropzone se volverá a añadir.

index.html

El fichero *index.html* contiene la estructura de todas las vistas. En él se importan todas las librerías y ficheros necesarios. También es aquí donde se establece la aplicación y un controlador común a todas las vistas (*MainController*).

Además, se incluye una cabecera común (*header.html*) y se indica dónde se debe mostrar el HTML correspondientes a la ruta actual (línea 30, figura 38).

```
<html>
    <head>
        <base href="/">
        <title>Rainblocks</title>
        <link rel='shortcut icon' type='image/x-icon' href='/app/images/favicon.ico'/>

        <!-- JS -->
        <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>
        <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.5/angular.min.js"></script>
        <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.5/angular-route.min.js"></script>
        <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.5/angular-cookies.min.js"></script>
        <script src="http://marceljuenemann.github.io/angular-drag-and-drop-lists/angular-drag-and-drop-lists.js"></script>
        <script src="https://cdnjs.cloudflare.com/ajax/libs/twitter-bootstrap/4.3.1/js/bootstrap.bundle.min.js"></script>
        <script src="https://use.fontawesome.com/releases/v5.7.0/js/all.js" data-auto-replace-svg="nest"></script>
        <script type='text/javascript' src="app/app.js"></script>
        <script type='text/javascript' src="app/controllers/mainController.js"></script>
        <script type='text/javascript' src="app/controllers/teacherController.js"></script>
        <script type='text/javascript' src="app/controllers/problemController.js"></script>
        <script type='text/javascript' src="app/controllers/studentController.js"></script>

        <!-- CSS -->
        <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css" integrity="
        sha384-Gn5384xqQ1aoWXA+058RXPxPg6fy4IWvTNh0E263XmFcJlSAwiGgFAW/dAiS6JXm" crossorigin="anonymous">
        <link rel="stylesheet" type="text/css" href="app/css/style.css">
    </head>

    <div ng-app="blocksApp">
        <div ng-controller="MainController">
            <div ng-include="'views/header.html'" ></div>
            <body>
                <ng-view ></ng-view>
            </body>
        </div>
    </div>
</html>
```

Figura 43. index.html

Node.js

Respecto a Node.js tenemos la carpeta *node_modules* que contiene los módulos que podemos utilizar, y el fichero *server.js*, que se encarga de la configuración de node.js.

```
var express = require('express');
const app = express();

app.use('/app', express.static(__dirname + '/app'));
app.use('/views', express.static(__dirname + '/views'));

app.get('*', (req, res) => {
    res.sendfile('./index.html');
});

app.listen(8000, () => {
  console.log('Server started!');
});
```

Figura 44. server.js

Primero se importa el módulo que se necesita y se instancia la aplicación. En las líneas 4 y 5 se indica que los archivos correspondientes a las carpetas *app* y *views* se sirvan directamente, es decir, que AngularJS no interprete la ruta. En las líneas 7, 8 y 9 se indica que el resto de rutas sí sean interpretadas por AngularJS, mediante el fichero *index.html*. Por último, en las líneas 11, 12 y 13 se arranca la aplicación en el puerto 8000.

5. Pruebas

Durante todo el tiempo de desarrollo, la aplicación ha sido continuamente probada a medida que avanzábamos en la implementación. Una vez estaba acabada, se le han realizado pruebas a todos los casos de uso para comprobar que se satisfacen (cada miembro del grupo ha realizado pruebas a los casos de uso de su documento). Para ello se han creado varios usuarios de prueba con los roles de profesor y alumno, y hemos simulado el comportamiento normal de la aplicación. Como los errores se han ido corrigiendo durante la implementación, al realizar estas pruebas finales no hemos encontrado errores de navegación, ni de integración, etc. Simplemente hemos añadido un par detalles que no estaban implementados pero que hemos considerado útiles al usar la aplicación, como por ejemplo que al revisar una solución y ponerle una nota, esta nota aparezca luego en el listado de soluciones.

Sin embargo, nos hubiese gustado realizar pruebas más exhaustivas a la aplicación, utilizando alguna herramienta de pruebas de servicios web, que realice pruebas de integración (mocking), de rendimiento, de carga, de navegación y de todas las que hubiesen sido útiles para una aplicación como ésta.

6. Conclusiones

6.1 Dificultades encontradas

- **Aprender las tecnologías**

  La primera dificultad que nos encontramos fue elegir las tecnologías y aprender a utilizarlas, ya que casi todas han sido nuevas para nosotros. Hemos realizado tutoriales para AngularJS, para Spring Boot con MongoDB, para crear una API REST en Spring y para Node.js.

- **Arrastrar y soltar bloques**

  Se necesitó un tiempo hasta dar con la librería adecuada que nos fuese útil para lo que queríamos hacer. Una vez la elegimos, tuvimos que aprender a utilizarla usando la documentación de Github y varios ejemplos implementados en su sitio web. Nos llevó más tiempo del debido conseguir que la aplicación hiciese lo que queríamos.

- **Diseñar la base de datos**

  Inicialmente pensamos utilizar una base de datos relacional, puesto que eran las que mejor conocíamos. Pero cuando intentamos hacer el diseño, nos dimos cuenta de que aquello iba a ser demasiado complejo como para llevarlo a cabo, porque teníamos demasiadas entidades, relaciones, claves foráneas, etc. Finalmente descubrimos las bases de datos NoSQL, como MongoDB, que sería la solución perfecta a nuestro problema de diseño. Aun así, el diagrama de clases ha sido editado un número considerable de veces.

- **Crear bloques que representen arrays**

  Imaginar los tipos de bloques básicos que tendría la aplicación no fue tarea fácil. Y menos aún lo fue intentar implementar bloques más complejos como uno que representase una estructura de tipo array, donde pudieras elegir su tamaño, el tipo de elemento, y que guardara todos los elementos dentro de sí mismo, además todo esto de manera visual. Al final pensamos que con los bloques que ya teníamos podíamos hacer tareas suficientes para que la

aplicación fuera funcional, y decidimos dejar este tipo de bloques para futuras ampliaciones de la herramienta.

- **Estructura de los custom blocks**
  Algo que ha supuesto un poco de dificultad a la hora de programar ha sido el diseño de los bloques personalizados. En un principio no hubo problemas, pero cuando tocó trabajar con ellos en el algoritmo de corrección, nos dimos cuenta de que quizá el diseño era algo enrevesado.

- **Algoritmo de corrección**
  Es quizás la parte más compleja de la aplicación, por lo que hizo falta la participación de los dos miembros del grupo para diseñarlo. Lo complicado era elegir cómo se iban a comparar dos estructuras y a partir de esto calcular una nota. El resultado no es perfecto, aunque sí bastante satisfactorio, y entendemos que con el tiempo podría mejorarse para hacerlo más preciso.

6.2 Posibles ampliaciones

- **Añadir seguridad a las contraseñas y a las URLs**
  Actualmente la aplicación no cuenta con seguridad de ningún tipo. Cualquier usuario puede teclear URLs y probar a navegar a páginas que no le corresponde ver, por ejemplo, un usuario con el rol de alumno podría averiguar la id de un problema mediante la URL, y así ver la solución. También cualquier programador puede obtener la contraseña de un usuario y mostrarla tal cual por pantalla, ya que ésta no está encriptada de ninguna forma.

- **Añadir bloques de estructuras más complejas**
  Respecto a la hora de utilizar los bloques, sería interesante que se pudieran crear estructuras más complejas utilizando bloques, como por ejemplo arrays. Esto haría posible que se pudiese usar la aplicación para más tipos de problemas más matemáticos y complejos.

- **Mejorar el sistema de corrección automática para que sea más preciso con la nota devuelta**

  Aunque estamos bastantes satisfechos con el algoritmo de corrección, pensamos que con algo más de desarrollo se podría mejorar para que ofreciera una nota más precisa, teniendo en cuenta pequeños aspectos que actualmente no tiene, como los bloques cambiados de orden.

- **Mejorar el sistema de corrección automática para que muestre visualmente de alguna forma los bloques que están mal en la solución**

  También sería interesante que en la página donde se le muestra al profesor la solución del alumno junto a la nota calculada por el algoritmo, se pudiese ver de alguna forma cuáles son los fallos de la solución, por ejemplo, marcando en rojo los bloques incorrectos.

- **Crear un usuario administrador para que cree más usuarios**

  La versión actual de la aplicación no cuenta con un sistema de creación de usuarios, sino que se han creado manualmente sobre la base de datos. Sería necesario crear la figura del administrador, que sería un usuario con privilegios para gestionar a los demás usuarios de la aplicación, pudiendo crearlos, darles una contraseña, eliminarlos, etc.

- **Añadir más idiomas**

  Desde el inicio se decidió crear la aplicación en inglés, para que fuese más internacional, y para que concordase el código con los menús. Además, si se subiese a internet, estando en inglés llegaría a más desarrolladores que la quieran ampliar. Pero sería interesante crear una traducción al español, y quizás a más idiomas.

- **Que sea adaptativa (responsive)**

  Por último, sabemos que esta aplicación solo está pensada para ser vista en la pantalla de un ordenador, pero en un futuro se pueden realizar los cambios necesarios para tener una interfaz gráfica adaptativa (responsive), para que pueda verse correctamente en distintos dispositivos.

6.3 Conclusión

El resultado final ha sido bastante satisfactorio, y se ha conseguido cumplir con los objetivos propuestos. Con este proyecto hemos conseguido aprender cosas nuevas, y obtener experiencia de las que ya habíamos estudiado durante el grado. Por ejemplo, AngularJS era totalmente nuevo para mí, y gracias al proyecto he obtenido bastante experiencia utilizándolo, que no hubiese obtenido por mi propia cuenta. Respecto a conocimientos que ya tenía como por ejemplo la arquitectura REST, he conseguido comprender mejor su funcionamiento y tener más soltura a la hora de utilizarla.

Haber realizado el trabajo con otra persona no ha supuesto ningún problema, al contrario, el trabajo en grupo ha supuesto que avanzásemos más rápido en lo que estábamos haciendo y que la aplicación sea más completa.

Referencias

- Documentación de AngularJS
  https://docs.angularjs.org/api
- Tutorial de AngularJS
  https://www.w3schools.com/angular/
- Librería Drag & Drop Lists de Angular
  https://github.com/marceljuenemann/angular-drag-and-drop-lists
- MongoDB
  https://www.mongodb.com/
- Spring Tools Suite
  https://spring.io/tools3/sts/all
- Tutorial para crear una REST API utilizando Spring Boot con MongoDB
  https://www.codementor.io/gtommee97/rest-api-java-spring-boot-and-mongodb-j7nluip8d
- Extensión para Chrome Advanced REST Client
  https://chrome.google.com/webstore/detail/advanced-rest-client/hgmloofddffdnphfgcellkdfbfbjeloo/related