ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
INGENIERÍA DEL SOFTWARE

Herramienta visual educativa para la resolución de problemas

Herramienta del alumno

Visual learning tool for problem solving

Student tool

Realizado por
María del Carmen Arjona Gómez

Tutorizado por
Eduardo Guzmán de los Riscos

Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, JUNIO DE 2019

Fecha defensa: Julio de 2019

Fdo. El/la Secretario/a del Tribunal

Resumen: En este proyecto se ha desarrollado una aplicación web para resolver problemas de tipo procedimental, cuya solución consiste en un conjunto de pasos ordenados. Para resolverlos, se utiliza un entorno visual, con bloques que se pueden arrastrar y soltar en cualquier orden. Estos bloques también permiten entradas de datos mediante entradas de texto, desplegables o incluso introduciendo bloques dentro de bloques. El objetivo es que sirva como herramienta didáctica entre profesores y alumnos de cualquier disciplina. También existe una separación de roles de manera que el profesor y el alumno realizan funciones distintas, pero complementarias.

Esta aplicación se ha realizado en grupo y en este proyecto se desarrolla tanto la parte del alumno como la parte común a todos los roles de usuario.

Las tecnologías que hemos utilizado para el frontend de la aplicación son principalmente AngularJS, HTML, Bootstrap y CSS. Por otro lado, para el backend hemos utilizado Spring y MongoDB.

Palabras clave: herramienta visual educativa, Angularjs, Mongodb, Spring

Abstract: In this project we have developed a web application to solve procedural problems, which solution follows a step by step structure. To solve them, we can use a visual environment with blocks that can be dragged and dropped in any order. These blocks can also contain data within themselves in the form of text inputs, dropdowns or even other blocks. The goal is to use this application as a learning tool for teachers and students from any discipline. We have also separated the users in two roles, since teachers and students do different functions, even though they complement each other.

The application has been developed by teamwork. This project contains the part of the student role but also the common part of the application to both roles, student and teacher.

The technologies we have used in the frontend are mainly AngularJS, HTML, Bootstrap y CSS. Otherwise, in the backend we have used Spring and MongoDB.

Keywords: visual learning tool, Angularjs, Mongodb, Spring

Índice de contenidos

1. Introducción

1.1. Descripción

El objetivo de este proyecto es crear una herramienta didáctica visual, que no requiera de un conocimiento muy profundo para su uso y que utilice una dinámica profesor/alumno que permita crear, solucionar y evaluar problemas de tipo procedimental.

Cada solución a un problema se entiende como un conjunto de pasos ordenados a seguir y cada paso, a su vez, puede estar compuesto de otros pasos. Estos pasos los hemos traducido a bloques, por lo tanto, cada paso es un bloque y cada bloque puede contener a su vez más bloques.

Mediante el uso de bloques, se pretende que el alumno no necesite conocer la sintaxis exacta de cada paso (como se lo exigiría por ejemplo un entorno de programación), sino que su atención se centre exclusivamente en el orden y la composición de los mismos.

Este enfoque de reducir un problema a bloques permite que con una misma aplicación se puedan resolver problemas de múltiples disciplinas, incluso algunas para las que no exista un lenguaje de programación equivalente.

1.2. Motivación, antecedentes y objetivos

Este proyecto surge de la necesidad de una herramienta de resolución de problemas de tipo procedimental, donde la solución es un conjunto de pasos ordenados y que, además:

1. Sea visual, de alto nivel.
2. Permita crear bloques personalizados con el objetivo de no estar limitado a una disciplina.
3. Distinga entre los roles de profesor y alumno, donde uno puede crear y corregir problemas mientras que el otro puede resolverlos, respectivamente.

Actualmente existen herramientas similares como Scratch, Snap!, AppInventor, etc. pero ninguna cumple todos los requisitos mencionados. Éstas son herramientas orientadas principalmente a la programación, sin embargo, esta metodología de resolución de problemas puede ser útil también en otras áreas de la

educación. Solo con añadir una funcionalidad que permita crear bloques personalizados es posible ampliar de manera significativa la cantidad de problemas de distintas disciplinas que pueden ser resueltos.

Aunque algunas de las herramientas mencionadas permiten crear bloques personalizados, no es muy útil si las tiene que crear un alumno mientras soluciona el problema. Esto creará inconsistencia en las soluciones por los diferentes alumnos que podrían diferir mucho unas de las otras, pues las opciones al crear bloques son muy amplias. Por lo tanto, es mucho más interesante que sea el profesor el que pueda crear bloques personalizados según necesiten los problemas que vaya a crear y que los alumnos tengan un número limitado de bloques para solucionar el problema propuesto por el profesor.

1.3. Tecnologías y herramientas utilizadas

Para el desarrollo de esta aplicación se han utilizado las siguientes tecnologías y herramientas:

- **JavaScript** es un lenguaje de programación interpretado. Se define como orientado a objetos, basado en prototipos, imperativo, débilmente tipado y dinámico. Se utiliza principalmente en su forma del lado del cliente, implementado como parte de un navegador web permitiendo mejoras en la interfaz de usuario y páginas web dinámicas.
- **jQuery** es una biblioteca multiplataforma de JavaScript que permite simplificar la manera de interactuar con los documentos HTML, manipular el árbol DOM, manejar eventos, desarrollar animaciones y agregar interacción con la técnica AJAX a páginas web.
- **AngularJS** es un framework de JavaScript de código abierto, mantenido por Google, que se utiliza para crear y mantener aplicaciones web de una sola página.
- **Drag & Drop Lists** es una librería de AngularJS que permite trabajar con bloques, arrastrarlos y soltarlos. En ella hemos basado los bloques utilizados en nuestra aplicación.
- **HTML** es un lenguaje de marcado para la elaboración de páginas web. Es un estándar que define una estructura básica y un código para la definición de

contenido de una página web, como texto, imágenes, videos, juegos, entre otros.

- **CSS** es un lenguaje de diseño gráfico para definir y crear la presentación de un documento estructurado escrito en un lenguaje de marcado, como HTML o XHTML.
- **Bootstrap** es un framework de CSS orientado al diseño adaptativo (*responsive*) que contiene una biblioteca de diseños de tipografías, formularios, botones y otros componentes de interfaz.
- **Java** es un lenguaje de programación de propósito general, concurrente, orientado a objetos.
- **Spring** es un framework para el desarrollo de aplicaciones y contenedor de inversión de control para la plataforma Java.
- **Spring Tools Suite** es un entorno de desarrollo basado en Eclipse adaptado para desarrollar aplicaciones Spring.
- **MongoDB** es un sistema de base de datos NoSQL orientado a documentos.
- **MongoDB Compass** es una interfaz gráfica de usuario para MongoDB.
- **Node.js** es un entorno en tiempo de ejecución multiplataforma, para la capa del servidor.
- **Advanced Rest Client** es una extensión para Google Chrome que permite realizar peticiones a APIs REST detalladamente.
- **Sublime Text** es un editor de texto que permite añadir un paquete de AngularJS para colorear el código correctamente.
- **MagicDraw UML** es una herramienta de modelado que permite diseñar y planificar de manera visual múltiples aspectos del ciclo de vida de desarrollo del software.
- **Pencil Project** es una herramienta de creación de mockups de aplicaciones.
- **Trello** es un software de administración de proyectos con interfaz web para organizar proyectos.

1.4. División del trabajo

Este proyecto está dividido en dos partes: la parte del profesor por un lado y la parte del alumno más la parte común por otro. Cada miembro del grupo ha desarrollado la parte perteneciente a uno de los roles (profesor o alumno). Sin

embargo, como la parte del profesor incluye un algoritmo de corrección automática, se ha decidido sumar la parte común de la aplicación (menú, rutas, sesión, etc.) a la parte del alumno para así repartir el trabajo lo más equitativamente posible.

Cabe destacar que ambos miembros hemos trabajado siempre simultáneamente, compartiendo los conocimientos que íbamos adquiriendo y revisando el proyecto en totalidad con el objetivo de que sea lo más completo y consistente posible. Nuestro objetivo como grupo no ha sido solo repartir el trabajo, sino también crear un proyecto conjunto.

1.5. Estructura de la memoria

- **1. Introducción**: comenzaremos explicando de dónde surge la necesidad de esta aplicación y qué objetivos queremos alcanzar con ella. También hablaremos un poco de qué tecnologías nos ayudarán a conseguirlo. Por último, comentamos cómo hemos distribuido y gestionado la división del trabajo.
- **2. Especificación y análisis**: listaremos y explicaremos los distintos tipos de usuario, así como los requisitos de la aplicación. Los requisitos están divididos en requisitos funcionales y no funcionales. Los funcionales se dividen a su vez en dos tipos: requisitos del usuario con rol de alumno y requisitos comunes a todos los tipos de usuarios. Por último, desarrollaremos el caso de uso correspondiente a cada requisito.
- **3. Diseño de la aplicación**: detallaremos algunos aspectos de la aplicación, como el diagrama de clases y la estructura que la compone.
- **4. Implementación**: explicaremos la estructura de los ficheros que forman el proyecto (tanto frontend como backend), así como partes del código que consideremos más relevantes o significativas. También se incluirán imágenes de ejemplo de la aplicación en funcionamiento.
- **5. Pruebas**: indicaremos las pruebas que hemos realizado para comprobar que el funcionamiento de la aplicación es correcto y qué resultado hemos obtenido.
- **6. Conclusiones y posibles mejoras**: desarrollaremos una conclusión final del proyecto, explicando cómo ha sido el proceso, qué partes nos han resultado más complicadas y qué mejoras se podrían implementar para completarlo o pulirlo.
- **7. Referencias**: listaremos los sitios webs que nos han servido de ayuda para documentarnos.

2. Especificación y análisis

2.1. Usuarios y roles

Nombre	Rol
Profesor	Usuario que gestiona los problemas, los bloques y las correcciones a los problemas.
Alumno	Usuario que realiza soluciones a problemas.

2.2. Perfiles de usuario

Nombre	Responsabilidades	Criterio de éxito
Profesor	Gestiona (crea, lee, modifica y elimina) problemas y bloques. Corrige las soluciones a los problemas realizadas por los alumnos.	Evalúa de manera eficiente la capacidad de resolución de problemas de los alumnos.
Alumno	Crea y edita soluciones a los problemas creados por el profesor. Accede al listado de las notas de los problemas resueltos.	Demuestra su capacidad de resolución de problemas de manera cómoda y visual.

2.3. Requisitos funcionales

Requisitos funcionales comunes (RFC)

RFC 01 - Iniciar sesión

Un usuario puede iniciar sesión en la aplicación

RFC 02 - Cerrar sesión

Un usuario puede cerrar sesión en la aplicación

RFC 03 - Modificar contraseña

Un usuario puede modificar su contraseña

RFC 04 - Modificar datos personales

Un usuario puede modificar sus datos personales

Requisitos funcionales alumno (RFA)

RFA 01 - Resolver problema

Un alumno puede crear una solución para un problema existente

RFA 02 - Ver notas

Un alumno puede ver las notas que el profesor le ha asignado en cada problema

2.4. Requisitos no funcionales

Requisitos no funcionales (RNF)

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

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

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

2.5. Casos de uso

RFC 01 - Iniciar sesión

Título	Iniciar sesión
Descripción	Un usuario puede iniciar sesión en la aplicación
Pre-condición	El usuario existe y no está autenticado
Post-condición	El usuario está autenticado en la aplicación
Escenario principal	
1. El usuario introduce su email y su contraseña. 2. El usuario pulsa el botón “Log in”. 3. La aplicación verifica que los datos introducidos son correctos. 4. La aplicación muestra la página principal relativa al rol del usuario autenticado.	
Escenario alternativo	
3b. La aplicación comprueba que el email introducido no existe. 4b. La aplicación muestra en la página de inicio de sesión el error “The user does	

not exist”. 3c. La aplicación comprueba que la contraseña introducida es errónea. 4c. La aplicación muestra en la página de inicio de sesión el error “Invalid password”.
Clases de análisis

A. Clases de entidad	User.java
B. Clases de control	UserController.java mainController.js
C. Clases de interfaz	login.html

Maquetas de interfaz

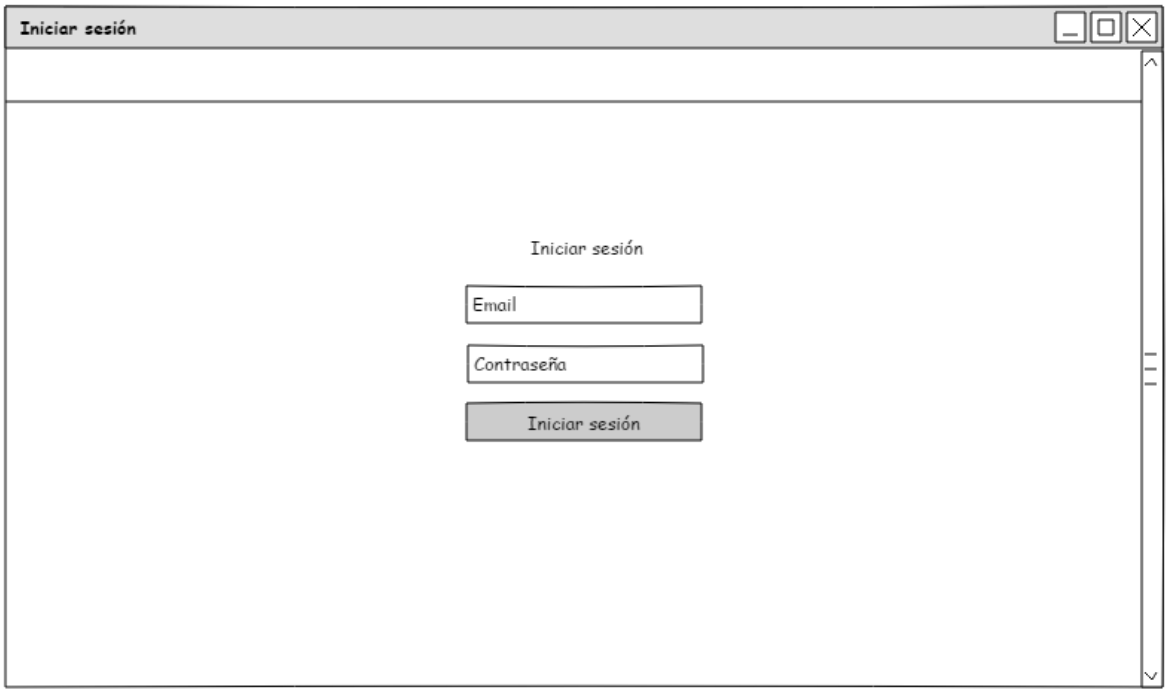

Figura 1 - Mockup de inicio de sesión

RFC 02 - Cerrar sesión

Título	Cerrar sesión
Descripción	Un usuario puede cerrar sesión en la aplicación

Pre-condición	El usuario está previamente autenticado
Post-condición	El usuario no está autenticado en la aplicación
Escenario principal	
1. Desde cualquier parte de la aplicación, el usuario pulsa el botón “Log out”. 2. La aplicación muestra la página de inicio de sesión.	
Clases de análisis	
A. Clases de entidad	-
B. Clases de control	mainController.js
C. Clases de interfaz	header.html
Maquetas de interfaz	

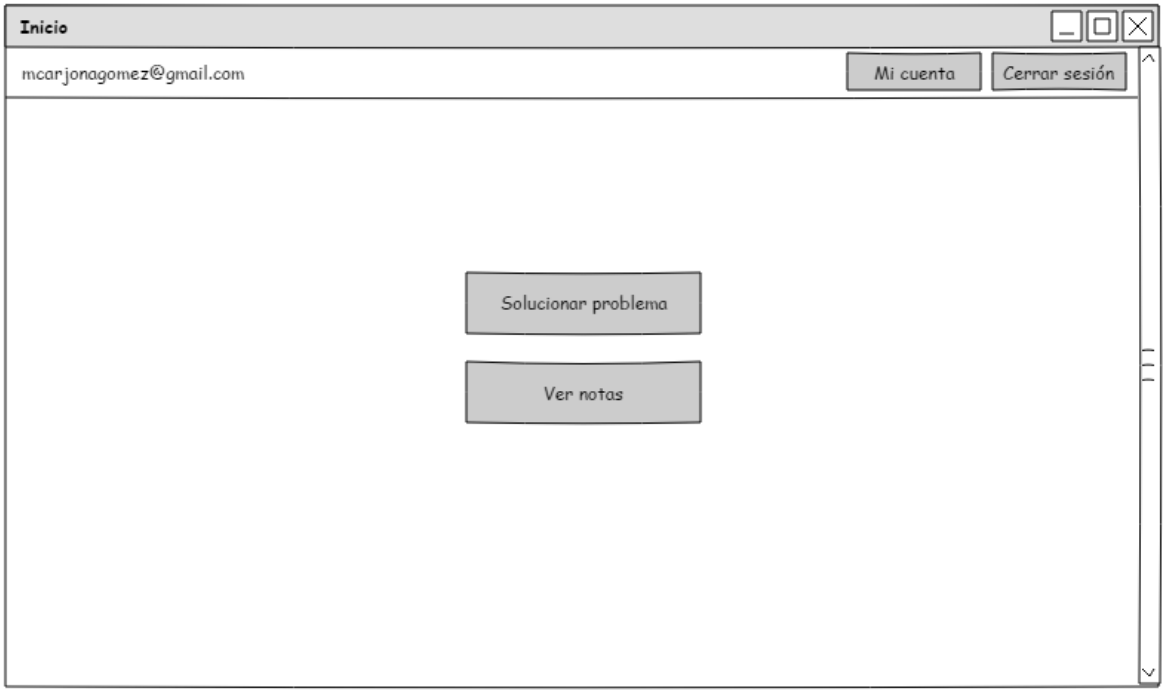

Figura 2 - Mockup de página que contiene la barra de navegación, donde está el botón de cerrar sesión

RFC 03 - Modificar contraseña

Título	Modificar contraseña
Descripción	Un usuario puede modificar su contraseña
Pre-condición	El usuario está previamente autenticado
Post-condición	La contraseña del usuario ha sido modificada en la base de datos de la aplicación
Escenario principal	
1. Desde cualquier parte de la aplicación, el usuario pulsa la pestaña “Settings”. 2. El usuario introduce su actual contraseña en el campo vacío “Password”. 3. El usuario introduce su nueva contraseña en el campo vacío “New password”. 4. El usuario pulsa el botón “Change password”. 5. La aplicación comprueba que los datos introducidos son correctos. 6. La aplicación actualiza la contraseña del usuario en la base de datos. 7. La aplicación muestra el mensaje de éxito “Password updated”.	
Escenario alternativo	
5b. La aplicación comprueba que la contraseña actual es errónea. 6b. La aplicación muestra el mensaje de error “Wrong password”. 5c. La contraseña no cumple los requisitos mínimos de seguridad. 6c. La aplicación muestra el mensaje de error indicando que es necesario introducir al mínimo una contraseña de 8 caracteres.	
Clases de análisis	
A. Clases de entidad	User.java

B. Clases de control	UserController.java mainController.js
C. Clases de interfaz	settings.html
Maquetas de interfaz	

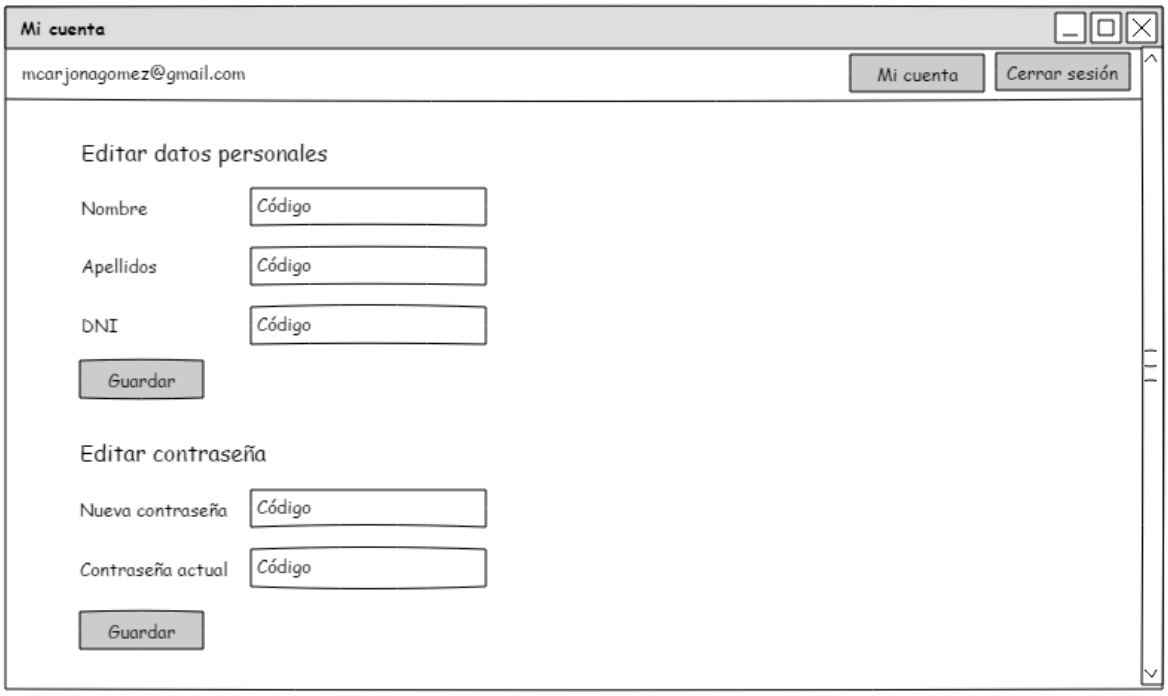

Figura 3 - Mockup de modificar contraseña

RFC 04 - Modificar datos personales

Título	Modificar datos personales
Descripción	Un usuario puede modificar sus datos personales: nombre, apellido y DNI
Pre-condición	El usuario está previamente autenticado
Post-condición	Los datos del usuario se modifican en la base de datos de la aplicación

Escenario principal

1. Desde cualquier parte de la aplicación, el usuario pulsa la pestaña “Settings”.
2. El usuario modifica los datos de los campos “Name”, “Last Name” y “DNI”.
3. El usuario pulsa el botón “Update user”.
4. La aplicación actualiza los datos del usuario en la base de datos.
5. La aplicación muestra el mensaje de éxito “User updated”.

Clases de análisis

A. Clases de entidad	User.java
B. Clases de control	UserController.java mainController.js
C. Clases de interfaz	settings.html

Maquetas de interfaz

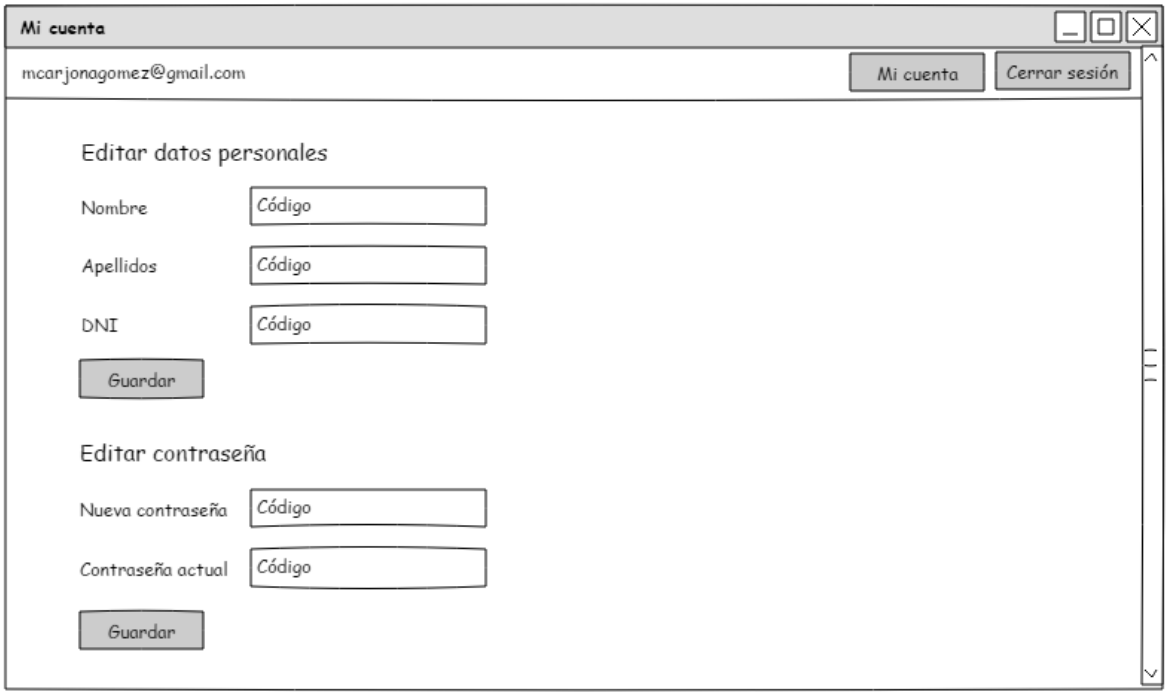

Figura 4 - Mockup de modificar datos personales

RFA 01 - Resolver problema

Título	Resolver problema
Descripción	Un alumno puede crear una solución para un problema existente, previamente creado por un profesor
Pre-condición	El usuario tiene rol de alumno, está previamente autenticado y el problema está habilitado.
Post-condición	La solución al problema sugerida por el alumno se almacena en la base de datos de la aplicación.
Escenario principal	
1. Desde la página principal de la aplicación (“Home”), el usuario pulsa el botón “Solve problem”. 2. EL usuario introduce en el campo “Enter code” el código relativo al problema que quiere resolver. 3. La aplicación comprueba que existe un problema con ese código. 4. El usuario crea una solución al problema, añadiendo todos los bloques que crea convenientes. 5. El usuario pulsa el botón “Save & Finish”. 6. La aplicación guarda la solución propuesta por el alumno en la base de datos. 7. La aplicación redirige al alumno a la página principal “Home”.	
Escenario alternativo	
3b. La aplicación comprueba que no existe ningún problema con ese código. 4b. La aplicación devuelve el error “This code doesn't belong to any problem”. 5c. El usuario pulsa el botón “Save”. 6c. La aplicación guarda la solución propuesta por el alumno en la base de datos.	
Clases de análisis	
A. Clases de entidad	Problem.java Solution.java

B. Clases de control	ProblemController.java SolutionController.java studentController.js
C. Clases de interfaz	solve-problem.html solution-edit.html

Maquetas de interfaz

Figura 5 - Mockup de introducir código del problema

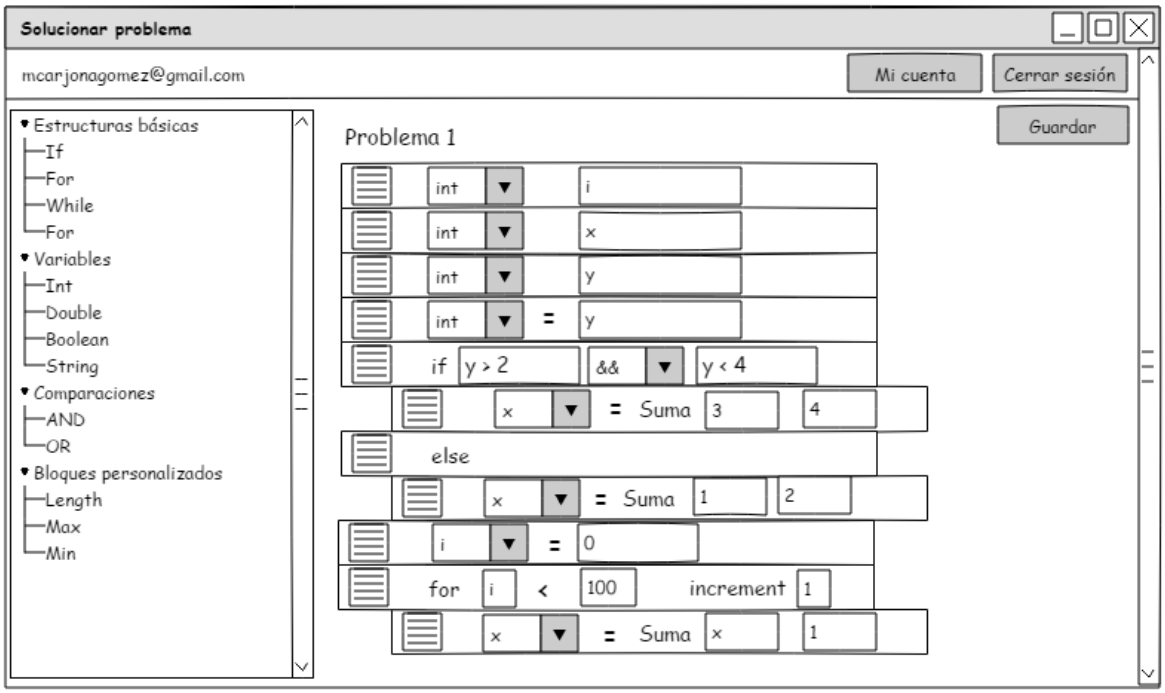

Figura 6 - Mockup de solucionar problema

RFA 02 - Ver notas

Título	Ver notas
Descripción	Un alumno puede ver las notas que cada profesor le ha asignado en los problemas que solucionó con anterioridad.
Pre-condición	El usuario tiene rol de alumno, está autenticado y existe algún problema corregido por el profesor correspondiente
Post-condición	El usuario ve su listado de notas
Escenario principal	
1. Desde la página principal de la aplicación (“Home”), el usuario pulsa el botón “My grades”. 2. La aplicación muestra una lista de notas de los problemas resueltos de ese alumno y su nota.	
Clases de análisis	
A. Clases de entidad	Solution.java Problem.java
B. Clases de control	SolutionController.java ProblemController.java studentController.js
C. Clases de interfaz	grades-list.html

Maquetas de interfaz

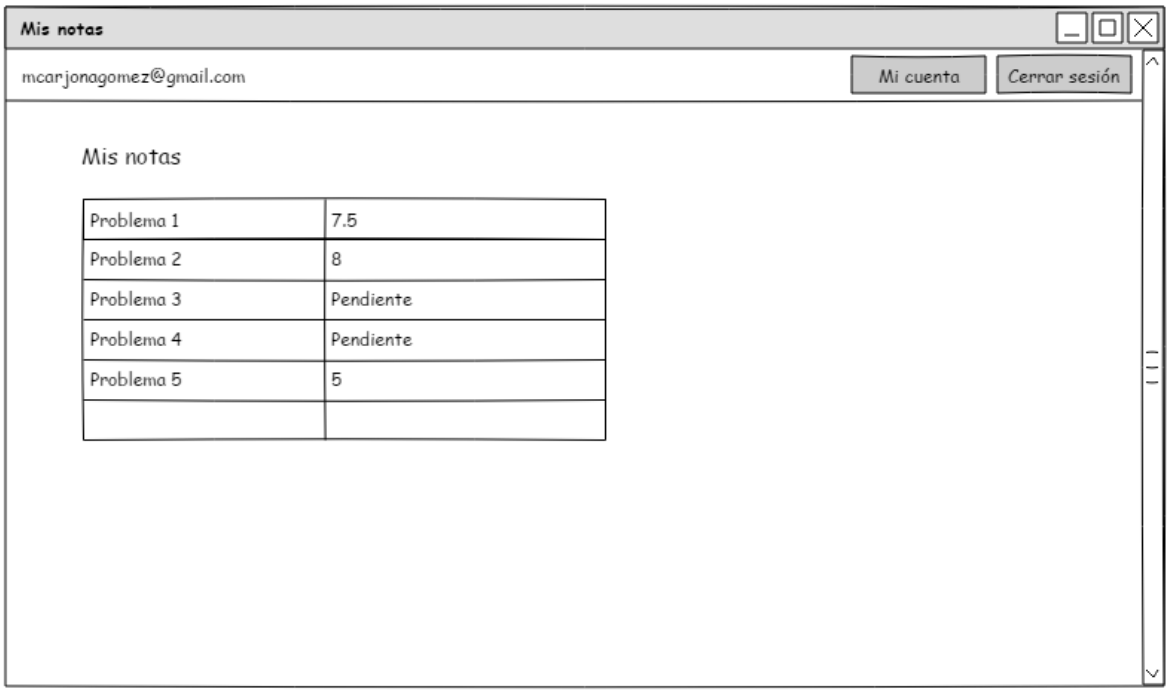

Figura 7 - Mockup de ver notas

3. Diseño de la aplicación

3.1. Diagrama de arquitectura

En la figura 8 podemos observar el diagrama de la arquitectura que constituye la aplicación.

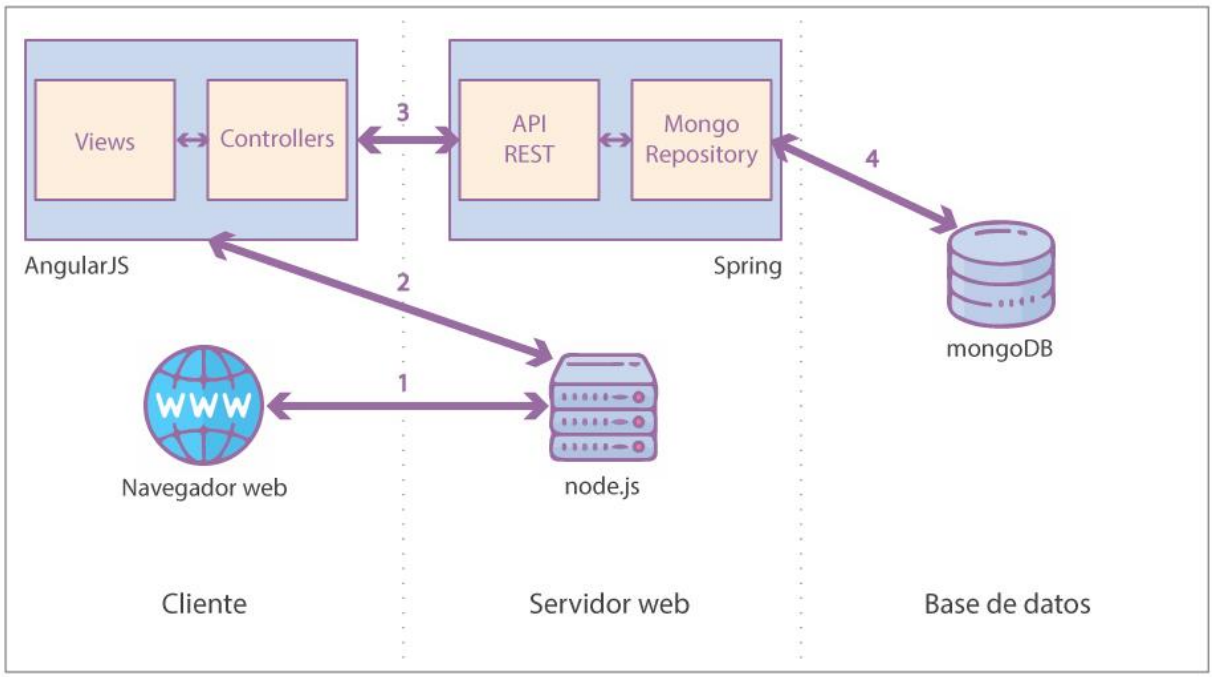

Figura 8 - Diagrama de arquitectura

El diagrama está dividido en tres partes:

- En la parte del **cliente** tenemos el navegador web que utiliza el usuario y AngularJS. El navegador se encarga de pedir la aplicación al servidor y AngularJS se encarga del funcionamiento general de la web (enrutamiento, vistas, controladores, peticiones a la API REST para obtener datos, etc.).
- En la parte del **servidor web** tenemos Node.js y Spring. Por un lado, Node.js se encarga de servir la aplicación (ya que la hemos desarrollado en local) y atender las peticiones HTTP que realiza el cliente a la aplicación web (AngularJS). Por otro lado, Spring se encarga de interpretar los datos almacenados en la base de datos mediante Mongo Repository para poder responder a las peticiones HTTP a la API REST que realiza la aplicación web.
- En la parte de la **base de datos** tenemos la base de datos con MongoDB. Elegimos esta tecnología porque necesitábamos basar nuestra aplicación en un

modelo no relacional, ya que los datos que se almacenan no mantienen siempre el mismo tipo de estructura. En el siguiente apartado se explicará con más detalle.

Basándonos en el diagrama de la figura 8, vamos a explicar como sería el flujo normal de la aplicación, donde los pasos se corresponden con las flechas de la figura 8:

- **Paso 1**: el navegador web hace una petición HTTP al servidor. En nuestro caso http://localhost:8000.
- **Paso 2**: node.js devuelve los ficheros correspondientes a la carpeta del proyecto. Por ejemplo, el fichero index.html. AngularJS muestra la aplicación.
- **Paso 3**: AngularJS hace una petición HTTP a la API REST que variará en función de los datos de la base de datos que necesite. Por ejemplo, http://localhost:8080/users/1 para obtener los datos del usuario con id=1.
- **Paso 4**: la API REST accede a la base de datos mediante el Mongo Repository para interpretarlos, ejecutar la sentencia solicitada y devolver el resultado como respuesta.

3.2. Diagrama de clases

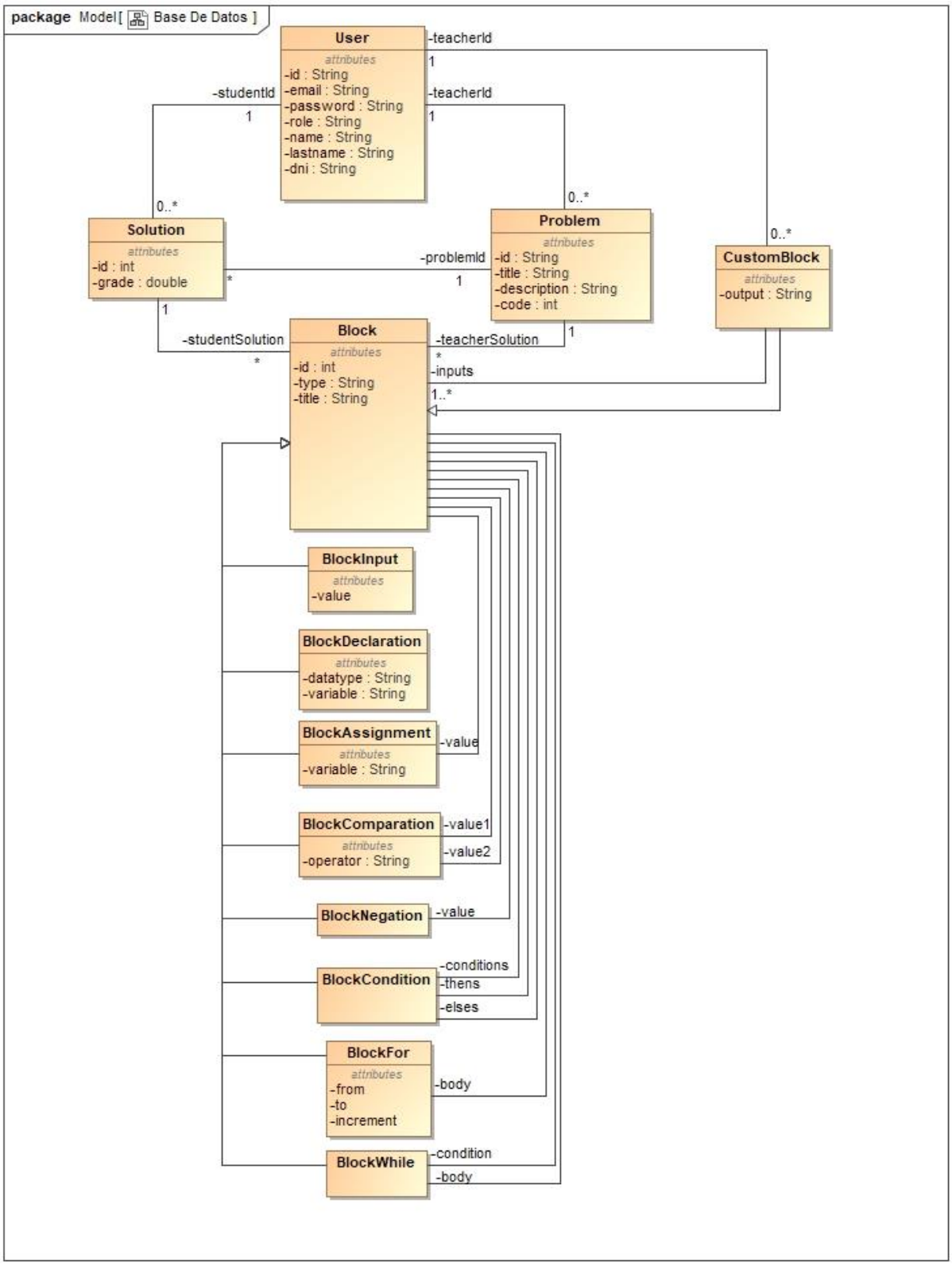

Figura 9 - Diagrama de clases

Aunque, como hemos mencionado previamente, los datos de nuestra aplicación no mantienen una estructura relacional, hemos diseñado un diagrama de clases equivalente, en la medida de lo posible, a la versión relacional. Este diagrama se puede observar en la figura 9.

Las clases que componen el diagrama son:

- **Usuario** (*User*) representa a los usuarios que utilizan el sistema. Éstos tienen como atributos *id*, *email*, *contraseña*, *rol*, *nombre* y *apellidos*. El rol del usuario define las partes de la aplicación a las que puede acceder y por lo tanto, las acciones que puede realizar. Un usuario puede tener rol de profesor o rol de alumno.
- **Problema** (*Problem*) representa los problemas que son creados por un profesor. Tiene como atributos *id*, *título*, *descripción* y *código*. Mientras que título y descripción definen y explican el problema, código se utiliza para que el alumno pueda acceder a él a la hora de crear una solución para ese problema. El alumno solo puede solucionar un problema si sabe cuál es su código correspondiente, por lo tanto, este código es único e irrepetible. Por último, cada problema está relacionado con un conjunto de bloques que forman la solución ideal a este problema y que la aplicación comparará con la solución del alumno para determinar si es correcto o no.
- **Solución** (*Solution*) representa una solución a un problema creada por un alumno. Tiene como atributos *id* y *nota*. La nota la asigna el profesor manualmente tras revisar si es correcta. Cada solución está relacionada con un conjunto de bloques, que son los bloques que el alumno ha utilizado para resolver el problema.
- **Bloque** (*Block*) representa los bloques que forman cada solución a cada problema. Es la clase de la que heredarán todos los tipos de bloques de la aplicación, ya que todos tienen en común los atributos *id*, *tipo* y *título*.
- **Tipos de bloques**:
  - Entradas (*BlockInput*): bloques de datos con un *valor* que pueden ser de cuatro tipos: *boolean, variable, string o numérica.* Las de tipo variable permitirán utilizar una variable que previamente haya sido declarada mediante un bloque de declaración.
  - Declaraciones (*BlockDeclaration*): bloques que permiten declarar variables indicando su *tipo* y su *nombre*.

- Asignaciones (*BlockAssignment*): bloques que permiten asignar el valor de un bloque a una *variable* previamente declarada.
- Comparaciones (*BlockComparation*): bloques que permiten comparar dos bloques entre sí, mediante el uso de un *operador* de comparación o lógico.
- Negaciones (*BlockNegation*): bloques que niegan el valor de un bloque.
- Condiciones (*BlockCondition*): bloques que simulan la estructura de control if/then/else y puede contener bloques en la *condición*, en el *else* y en el *then*.
- Bucles for (*BlockFor*): bloques que simulan la estructura de control for/next, tiene como atributos *desde, hasta* e *incremento* y puede contener bloques en el *body*.
- Bucles while (*BlockWhile*): bloques que simulan la estructura de control while/do y puede contener bloques en la *condición* y en el *body.*
- Personalizados (*BlockCustom*): bloques personalizados que puede crear cada profesor. Tiene como atributo *tipo de salida* y está relacionado con uno o varios *tipos de entradas*. Al crearlos, el profesor especificará su *título* y de qué tipo tiene que ser cada entrada y la salida.

Mientras diseñábamos este diagrama, fue cuando nos dimos cuenta de que necesitábamos un modelo no relacional, pues sino habríamos tenido que crear muchas clases distintas para los diferentes tipos de bloques, la mayoría con apenas atributos, pues lo que las diferencia son sus relaciones principalmente. Por esto, llegamos a la conclusión de que era mejor utilizar una estructura de datos JSON con MongoDB.

3.3. Diagrama de navegación

En el diagrama de la figura 10 podemos observar el diagrama de navegación de la aplicación, correspondiente al rol del alumno. En él podemos observar tanto el flujo de navegación como los ficheros HTML que se corresponden con cada vista.

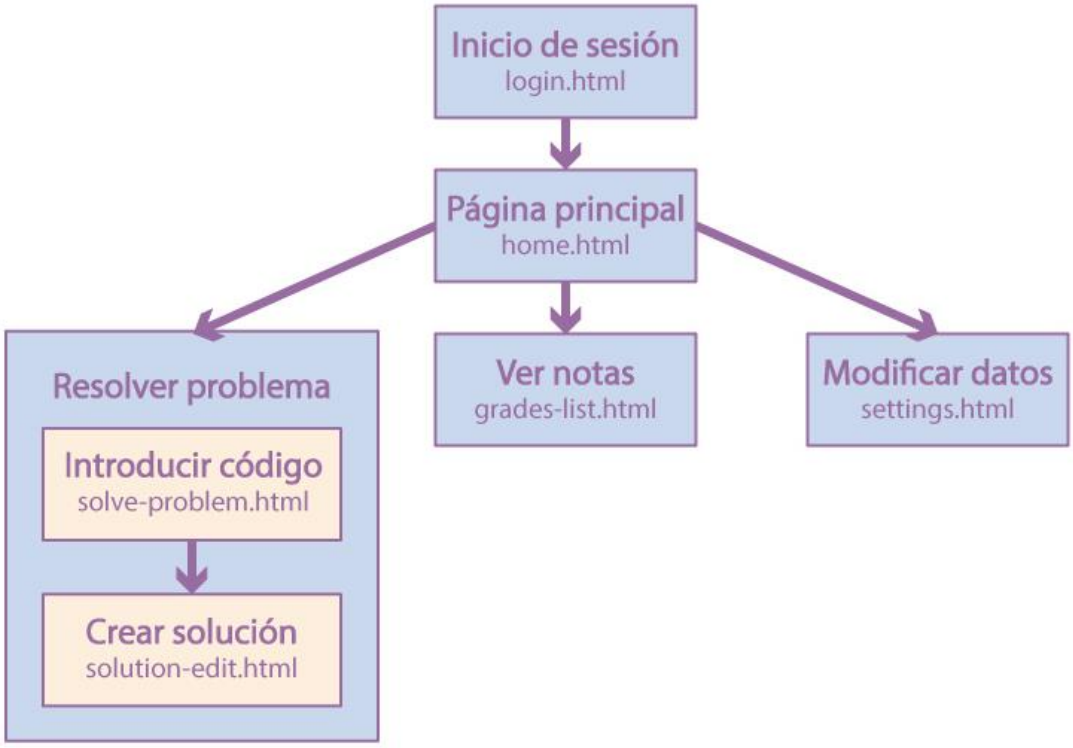

Figura 10 - Diagrama de navegación

Usualmente, el usuario iniciará sesión, y después accederá a una de las tres principales funcionalidades de su rol: “Resolver problema”, “Ver notas” o “Modificar datos”. La primera se divide a su vez en “Introducir código” y “Crear solución”.

Mientras que un usuario que no ha iniciado sesión todavía tendrá que comenzar por la vista “Inicio de sesión” obligatoriamente, otro que sí lo haya hecho podrá acceder a la URL de la página que quiera visitar directamente. Sin embargo, normalmente se accederá a una de las tres funcionalidades básicas de la aplicación desde la página principal.

La aplicación dispone de una barra de navegación superior que permite, desde cualquier vista, acceder a la “Página principal” y a “Modificar datos”.

4. Implementación

4.1. Evolución

En este apartado vamos a explicar cómo hemos desarrollado la aplicación. Pero antes mostraremos brevemente cómo ha ido evolucionando la parte relativa a los bloques. Esta parte fue la primera que desarrollamos. No estábamos seguros de si la librería que encontramos nos sería de utilidad, así que hasta que no nos aseguramos de que nos serviría, no desarrollamos el resto de la aplicación. Esta es la base que posteriormente ambos miembros del grupo utilizamos para nuestros respectivos proyectos.

En las figuras 11, 12, 13 y 14 se puede observar la evolución de los bloques. Comenzamos con muy pocos tipos de bloques y después fuimos añadiendo más tipos poco a poco, aunque lo primero fue cerciorarnos de que era posible introducir un bloque dentro de otro bloque. También fuimos mejorando la apariencia y el funcionamiento de la acción de arrastrar.

Al principio la aplicación se limitaba a leer el valor del JSON estático que nosotros habíamos introducido en una variable. Más adelante, creamos un panel vacío al que se le pudieran agregar los tipos bloques que nos ofrecía el panel de la izquierda. También añadimos una papelera donde arrastrar los bloques que quisiéramos eliminar.

Una vez alcanzado el prototipo de la figura 14, ya comenzamos a desarrollar el resto de la aplicación y a integrarla con la base de datos.

Figura 11 - Prototipo de aplicación 1

Figura 12 - Prototipo de aplicación 2

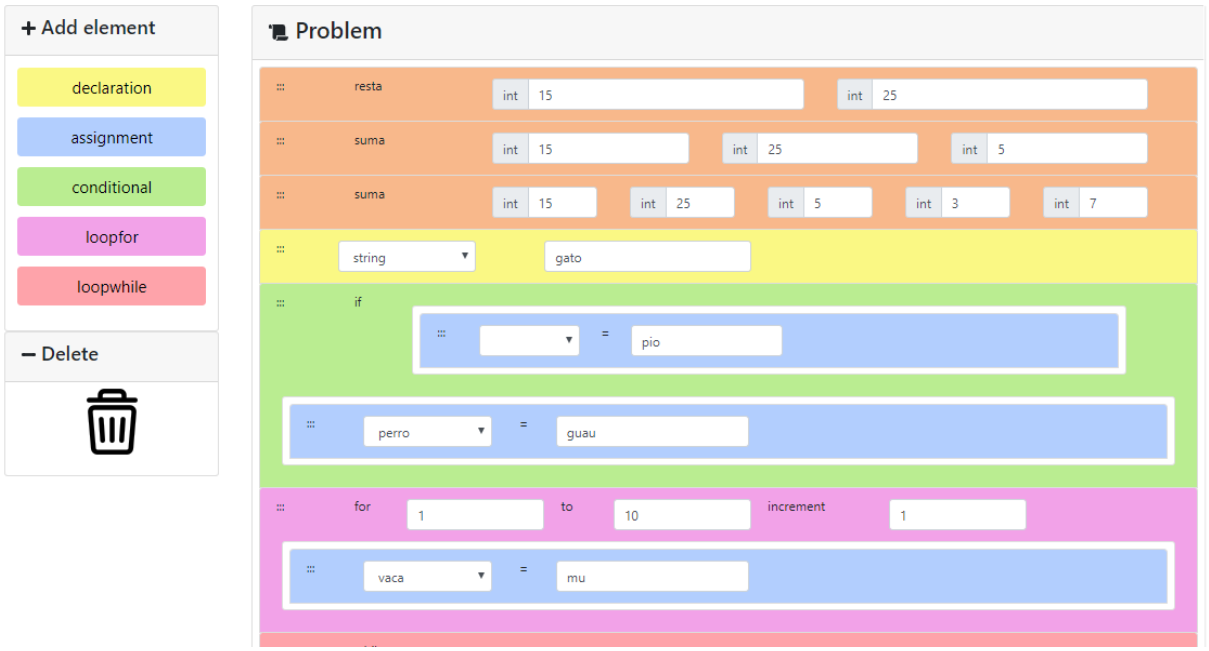

Figura 13 - Prototipo de aplicación 3

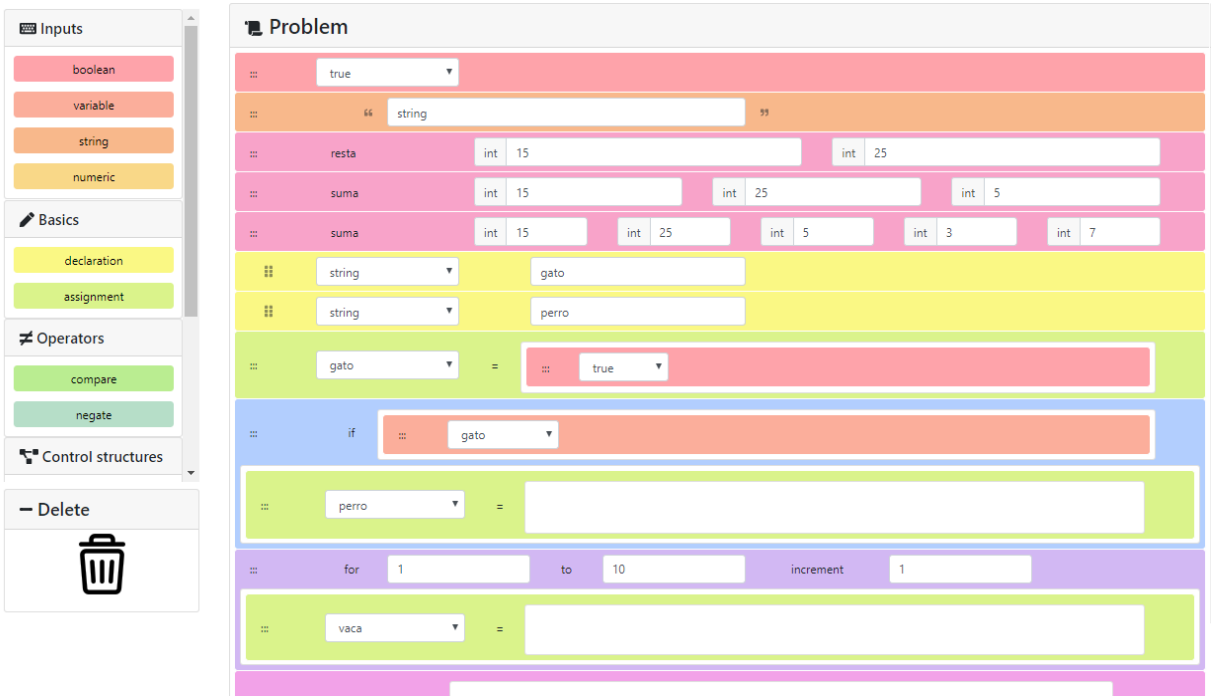

Figura 14 - Prototipo de aplicación 4

4.2. Backend

Como hemos visto previamente, el backend de la aplicación se ha desarrollado principalmente con Spring y MongoDB. Antes de empezar a explicar el código, lo primero será comentar brevemente la estructura que siguen los ficheros (figura 15).

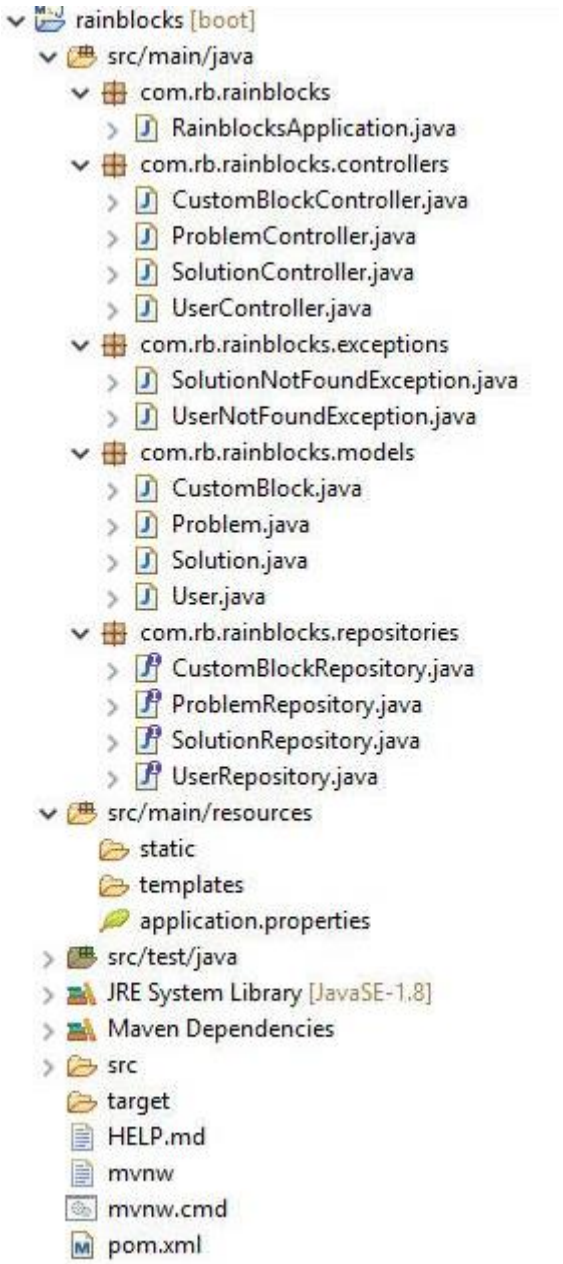

Figura 15 - Estructura de ficheros del backend

Los ficheros principales están divididos en cuatro paquetes:

- Repositorios: se encargan de acceder y modificar la base de datos.
- Modelos: se corresponden con las clases de la base de datos.
- Controladores: responde a las peticiones HTTP, realizando antes las acciones que necesite.
- Excepciones: errores controlados de la aplicación.

Los ficheros de configuración más importantes son *pom.xml* y *application.properties*. En *pom.xml* se añaden algunos datos básicos de la aplicación, como el nombre, y las dependencias necesarias. Se rellena automáticamente al crear el proyecto, pero de necesitar alguna dependencia adicional hay que añadirla aquí. En *application.properties* hay que indicar los datos de acceso a la base de datos (figura 16).

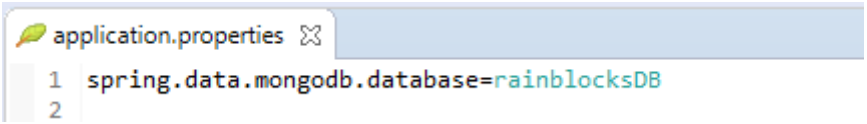

Figura 16 - Fichero application.properties

El fichero que arranca la aplicación al desplegarla es RainblocksApplication.java (figura 17).

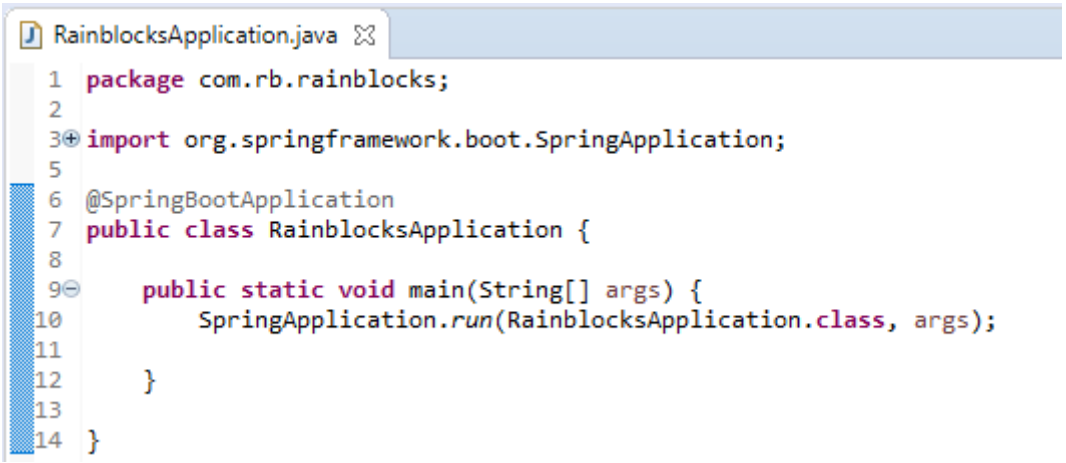

Figura 17 - Fichero RainblocksApplication.java

A continuación, vamos a explicar un fichero de cada uno de los cuatro tipos mencionados previamente. Con esto será suficiente, pues los ficheros de cada paquete son similares entre sí, aunque varíen según los atributos de la clase con la que se corresponden.

Repositorio

Los repositorios extienden de MongoRepository, que ya contiene las funciones básicas para crear, leer, editar y borrar. Sin embargo, si queremos otro tipo de consultas personalizadas, como búsquedas, tenemos que añadirlas. Para añadir una búsqueda solo tenemos que seguir el formato que se observa en la figura 18, líneas 11, 12, y 13. Es importante que el nombre de la función y del parámetro que le pasamos coincidan con los atributos del modelo, pues es lo que utiliza para saber qué buscar. Simplemente con esto ya tenemos implementada una función de búsqueda a partir de un atributo de la clase.

SolutionRepository.java

```
1  package com.rb.rainblocks.repositories;
2
3  import java.util.List;
9
10 public interface SolutionRepository extends MongoRepository<Solution, String> {
11     Solution findBy_id(ObjectId _id);
12     List<Solution> findByProblemId(String problemId);
13     List<Solution> findByStudentId(String studentId);
14 }
```

Figura 18 - Fichero SolutionRepository..java

Modelo

Los modelos son similares a los de Java, son una clase que contiene atributos, constructor, funciones getters y funciones setters. Lo más destacable es que hay que añadir las anotaciones @Document e @Id (figura 19, líneas 9 y 12) a la clase y al identificador, respectivamente, para indicar la correspondencia de éstos con la entidad y la clave primaria en MongoDB.

```
1  package com.rb.rainblocks.models;
2
3  import java.util.List;
8
9  @Document
10 public class Solution {
11
12     @Id
13     public ObjectId _id;
14
15     public String grade;
16     public List<Object> studentSolution;
17     public String studentId;
18     public String problemId;
19
20     public Solution(ObjectId _id, String grade, List<Object> studentSolution, String studentId, String problemId) {
21         super();
22         this._id = _id;
23         this.grade = grade;
24         this.studentSolution = studentSolution;
25         this.studentId = studentId;
26         this.problemId = problemId;
27     }
28     public String get_id() {
29         return _id.toHexString();
30     }
31     public void set_id(ObjectId _id) {
32         this._id = _id;
33     }
34     public String getGrade() {
35         return grade;
36     }
37     public void setGrade(String grade) {
38         this.grade = grade;
39     }
40     public List<Object> getStudentSolution() {
41         return studentSolution;
42     }
43     public void setStudentSolution(List<Object> studentSolution) {
44         this.studentSolution = studentSolution;
45     }
46     public String getStudentId() {
47         return studentId;
48     }
49     public void setStudentId(String studentId) {
50         this.studentId = studentId;
51     }
52     public String getProblemId() {
53         return problemId;
54     }
55     public void setProblemId(String problemId) {
56         this.problemId = problemId;
57     }
58
59
60
61 }
```

Figura 19 - Fichero Solution.java

Controlador

El controlador recibe las peticiones HTTP y devuelve una respuesta. Para indicarle a Spring que esta clase desempeña esta tarea, hay que usar la anotación @RestController (figura 20, línea 21). Por otro lado, la anotación @RequestMapping (figura 20, línea 22) indica que esta clase recibirá las peticiones cuya URL empiece por “/solutions”. Dentro de cada controlador, tenemos también la anotación @Autowired (figura 20, línea 25) que indica el repositorio que va a utilizar esta clase para leer y escribir datos. Por lo general, habrá un controlador y un repositorio por cada modelo con el que queramos trabajar, y cada controlador utilizará su correspondiente repositorio.

El resto del controlador contiene las distintas funciones que podemos llamar utilizando la URL, el método HTTP y variables indicadas en su correspondiente @RequestMapping. El método HTTP utilizado dependerá de la acción que queramos realizar: crear (POST), leer (GET), actualizar (PUT) o borrar (DELETE).
Por ejemplo, podemos obtener los datos de la solución con id 5 mediante la URL /solutions/5 con el método GET (ver figura 20, línea 35).

La mayoría de las veces la función recibirá como parámetros el id y/o los datos de la entidad que queremos leer/actualizar. Luego, utilizaremos el repositorio para leerlos/actualizarlos.

La función de la figura 20 línea 71 es un poco más compleja que el resto, así que vamos a analizarla. Necesitamos leer (GET) la solución a un problema concreto (problemId) por un alumno concreto (studentId). Para hacer esto primero obtenemos todas las soluciones de este alumno y después buscamos si entre ellas alguna es la del problema que queremos encontrar. Si no encontramos la solución que buscamos, devolvemos un mensaje de error.

Por último, mencionar que la anotación @CrossOrigin (figura 20, línea 28) sirve para darle permiso al frontend a que realice estas peticiones HTTP, ya que lo hemos ejecutado siempre en localhost puerto 8000. Si cambiase la URL o el puerto habría que actualizar estos datos.

SolutionController.java

```java
package com.rb.rainblocks.controllers;
import java.util.ArrayList;

@RestController
@RequestMapping("/solutions")
public class SolutionController {

    @Autowired
    private SolutionRepository repository;

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

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

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

    @CrossOrigin(origins = "http://localhost:8000")
    @RequestMapping(value = "/", method = RequestMethod.POST)
    public Solution createSolution(@Valid @RequestBody Solution solution) {
        solution.set_id(ObjectId.get());
        List<Object> ss = new ArrayList<>();
        solution.setStudentSolution(ss);
        repository.save(solution);
        return solution;
    }

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

    @CrossOrigin(origins = "http://localhost:8000")
    @RequestMapping(value = "/byProblem/{problemId}", method = RequestMethod.GET)
    public List<Solution> getSolutionByProblemId(@PathVariable("problemId") String problemId) {
        return repository.findByProblemId(problemId);
    }

    @CrossOrigin(origins = "http://localhost:8000")
    @RequestMapping(value = "/byProblemStudent/{problemId}/{studentId}", method = RequestMethod.GET)
    public Solution getSolutionByProblemIdStudentId(@PathVariable("problemId") String problemId, @PathVariable("studentId") String studentId) {
        List<Solution> byStudent = repository.findByStudentId(studentId);
        Solution res = null;
        Iterator<Solution> iterator = byStudent.iterator();
        while(iterator.hasNext()) {
            Solution s = iterator.next();
            if(s.getProblemId().equals(problemId)) {
                res = s;
            }
        }
        if(res == null) {
            throw new SolutionNotFoundException("Solution not found");
        } else {
            return res;
        }
    }

    @CrossOrigin(origins = "http://localhost:8000")
    @RequestMapping(value = "/byStudent/{studentId}", method = RequestMethod.GET)
    public List<Solution> getSolutionByStudentId(@PathVariable("studentId") String studentId) {
        return repository.findByStudentId(studentId);
    }
}
```

Figura 20 - Fichero SolutionController.java

Excepción

Las excepciones siguen siempre la estructura de la figura 21. Solo hay que introducir como argumento el mensaje que queremos mostrar cuando llamamos a la función SolutionNotFoundException.

SolutionNotFoundException.java

```
1  package com.rb.rainblocks.exceptions;
2
3  public class SolutionNotFoundException extends RuntimeException {
4
5      private static final long serialVersionUID = 1L;
6
7      public SolutionNotFoundException(String exception) {
8          super(exception);
9        }
10
11 }
```

Figura 21 - Fichero SolutionNotFoundException.java

4.3. Frontend

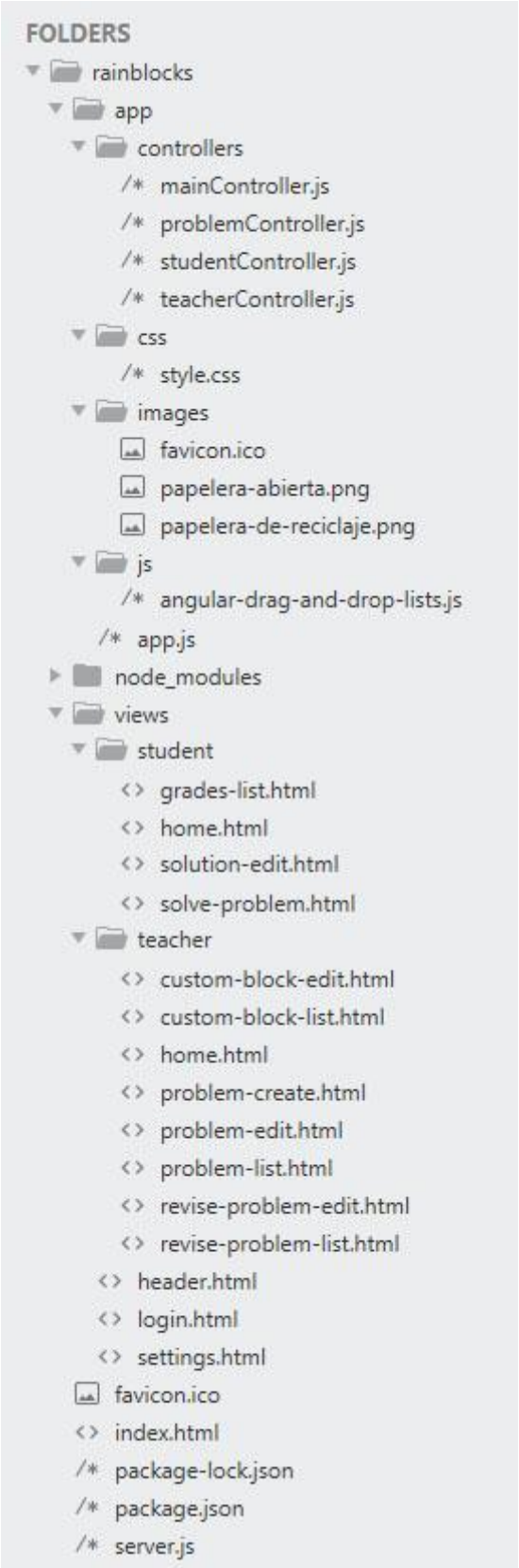

Figura 22 - Estructura de ficheros del frontend

El frontend de la aplicación se ha desarrollado fundamentalmente con AngularJS, aunque también hemos necesitado HTML, JS, CSS, etc. Antes de empezar a explicar el código, comentaremos la estructura que siguen los ficheros (figura 22).

Para ejecutar la aplicación hemos utilizado Node.js. En la carpeta *node_modules* están las librerías de node que podremos utilizar, para lo cual primero tendremos que importarlas en el fichero *server.js*.

Por otro lado, en la carpeta *views* están las vistas de la aplicación mientras que en la carpeta *app* está el resto de la aplicación: controladores, css, imágenes, librerías y el fichero principal de AngularJS *app.js*. En la carpeta raíz de la aplicación se encuentra la vista principal *index.html*.

Node.js

La configuración de node.js para este proyecto es muy sencilla, pero vamos a explicar el fichero *server.js* (figura 23) brevemente. En las líneas 1 y 2 importamos la librería que necesitamos e instanciamos la aplicación. En las líneas 4 y 5 indicamos 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 indicamos que el resto de rutas sí sean interpretadas por AngularJS, mediante el fichero index.html. Por último, en las líneas 11, 12 y 13 arrancamos la aplicación en el puerto 8000.

server.js

```
1  var express = require('express');
2  const app = express();
3
4  app.use('/app', express.static(__dirname + '/app'));
5  app.use('/views', express.static(__dirname + '/views'));
6
7  app.get('*', (req, res) => {
8      res.sendfile('./index.html');
9  });
10
11 app.listen(8000, () => {
12   console.log('Server started!');
13 });
```

Figura 23 - Fichero server.js

AngularJS

El fichero *app.js* (figura 24 y 25) es uno de los principales de AngularJS. En él se instancia la aplicación junto con las librerías que necesite (línea 3). Luego, entre las líneas 5 y 73 se establecen las rutas de la aplicación (aunque en la imagen no se muestran las relativas al rol del profesor). Para cada path se establece una vista (templateUrl) y, si lo necesita, un controlador específico (controller). En caso de introducir un path que no coincida con ninguno de los anteriores, nos llevará a la página inicial (línea 31).

app.js

```
1  'use strict';
2
3  var app =  angular.module('blocksApp',['dndLists','ngRoute','ngCookies'])
4
5  .config(function($routeProvider,$locationProvider) {
6    $locationProvider.html5Mode(true);
7    $routeProvider
8    // COMMON
9    .when('/login', {
10     templateUrl : 'views/login.html'
11   })
12   .when('/settings', {
13     templateUrl : 'views/settings.html'
14   })
15   // STUDENT
16   .when('/student/home', {
17     templateUrl : 'views/student/home.html'
18   })
19   .when('/student/solve-problem', {
20     templateUrl : 'views/student/solve-problem.html',
21     controller: 'StudentController'
22   })
23   .when('/student/solve-problem/:id', {
24     templateUrl : 'views/student/solution-edit.html',
25     controller: 'ProblemController'
26   })
27   .when('/student/grades', {
28     templateUrl : 'views/student/grades-list.html',
29     controller: 'StudentController'
30   })
31   .otherwise({redirectTo : '/home'})
32 })
```

Figura 24 - Fichero app.js

En el bloque run (figura 25, líneas 74-96) se inicializa la aplicación y se añaden unas funciones que se ejecutarán cada vez que se inicie la aplicación o se recargue la página. Estas funciones son: redirigir al login si no hemos iniciado sesión (líneas 78-80), redirigir a la página de inicio del rol específico si es necesario y redirigir a la página de inicio si se intenta acceder a cualquier vista del otro rol (líneas 89-93).

```
74  .run(['$rootScope', '$location', '$cookies', '$http',
75    function ($rootScope, $location, $cookies, $http) {
76      $rootScope.$on('$locationChangeStart', function (event, next, current) {
77        // Redirect to login page if not logged in
78        if ($location.path() !== '/login' && !$rootScope.isAuthenticated) {
79          $location.path('/login');
80        }
81
82        // Redirect to specific role home
83        var role = $rootScope.userRole;
84        if ($location.path() == '/home') {
85          $location.path('/'+role+'/home');
86        }
87
88        // Redirect to home page if role isn't allowed
89        var pathStart = $location.path().split("/")[1]||"none";
90        if (role!==undefined && pathStart!=='none'
91            && ((role=='teacher' && pathStart=='student') || (role=='student' && pathStart=='teacher'))) {
92            $location.path('/'+role+'/home');
93        }
94      });
95    }
96  ]);
```

Figura 25 - Fichero app.js

```
index.html
1   <html>
2       <head>
3           <base href="/">
4           <title>Rainblocks</title>
5           <link rel='shortcut icon' type='image/x-icon' href='/app/images/favicon.ico'/>
6
7           <!-- JS -->
8           <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>
9           <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.5/angular.min.js"></script>
10          <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.5/angular-route.min.js"></script>
11          <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.5/angular-cookies.min.js"></script>
12          <script src="http://marceljuenemann.github.io/angular-drag-and-drop-lists/angular-drag-and-drop-lists.js"></script>
13          <script src="https://cdnjs.cloudflare.com/ajax/libs/twitter-bootstrap/4.3.1/js/bootstrap.bundle.min.js"></script>
14          <script src="https://use.fontawesome.com/releases/v5.7.0/js/all.js" data-auto-replace-svg="nest"></script>
15          <script type='text/javascript' src="app/app.js"></script>
16          <script type='text/javascript' src="app/controllers/mainController.js"></script>
17          <script type='text/javascript' src="app/controllers/teacherController.js"></script>
18          <script type='text/javascript' src="app/controllers/problemController.js"></script>
19          <script type='text/javascript' src="app/controllers/studentController.js"></script>
20
21          <!-- CSS -->
22          <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css" integrity="
            sha384-Gn5384xqQ1aoWXA+058RXPxPg6fy4IWvTNh0E263XmFcJlSAwiGgFAW/dAiS6JXm" crossorigin="anonymous">
23          <link rel="stylesheet" type="text/css" href="app/css/style.css">
24      </head>
25
26      <div ng-app="blocksApp">
27          <div ng-controller="MainController">
28              <div ng-include="'views/header.html'" ></div>
29              <body>
30                  <ng-view></ng-view>
31              </body>
32          </div>
33      </div>
34  </html>
```

Figura 26 - Fichero index.html

El fichero *index.html* (figura 26) es la estructura de todas las vistas. En él se importan todas las librerías y ficheros necesarios (líneas 7-23). También es aquí donde se establece la aplicación (línea 26) y un controlador común a todas las vistas (línea 27). Además, se incluye una cabecera común (línea 28) y se indica dónde se debe mostrar el HTML correspondientes a la ruta actual (línea 30).

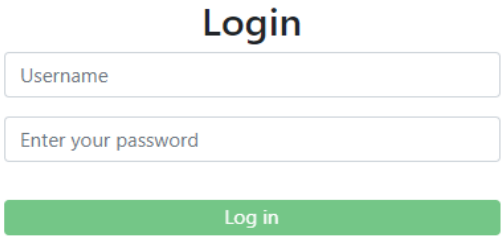

Figura 27 - Página de inicio de sesión

El fichero *login.phtml* (figura 28) muestra un formulario de inicio de sesión. El botón de envío solo estará disponible (línea 28, ng-disabled) cuando los campos email y contraseña sean válidos. Al enviar el formulario se enviará el objeto usuario, formado por email y contraseña (líneas 6 y 11, ng-model) a la función login (línea 3, ng-submit). Si los datos son incorrectos se activarán (ng-show) los mensajes de error (líneas 14-25). Visualmente se corresponde con la figura 27.

```
1  <div class="login-container">
2      <h2>Login</h2>
3      <form name="loginForm" ng-submit="login(user)">
4          <div class="row">
5              <div class="col-md-12 form-group">
6                  <input type="email" class="form-control" placeholder="Username" name="email" ng-model="user.email" required>
7              </div>
8          </div>
9          <div class="row">
10             <div class="col-md-12 form-group">
11                 <input type="password" placeholder="Enter your password" class="form-control" name="password" ng-model="
                   user.password" required>
12             </div>
13         </div>
14         <div class="mb-3">
15             <span style="color:red" ng-show="loginForm.email.$dirty && loginForm.email.$invalid">
16                 <span ng-show="loginForm.email.$error.required">Email is required.</span>
17                 <span ng-show="loginForm.email.$error.email">Invalid email address.</span>
18             </span>
19             <span style="color:red" ng-show="loginForm.password.$dirty && loginForm.password.$invalid">
20                 <span ng-show="loginForm.password.$error.required">Password is required.</span>
21             </span>
22             <span style="color:red" ng-show="loginError">
23                 <span ng-show="loginError">{{loginError}}</span>
24             </span>
25         </div>
26         <div class="row">
27             <div class="col-md-12 form-group">
28                 <input type="submit" value="Log in" class="btn btn-block btn-success" ng-disabled="
                   loginForm.password.$invalid || loginForm.email.$invalid">
29             </div>
30         </div>
31     </form>
32 </div>
```

Figura 28 - Fichero login.phtml

El fichero *mainController.js* contiene las funciones comunes a todos los usuarios, es decir, las relativas a iniciar y cerrar sesión, así como las de modificar los datos personales. Lo más destacable de este fichero es el proceso de inicio de sesión (figura 29). Al cargar la aplicación se comprueba si existe un usuario que ya haya iniciado sesión (líneas 13-19) ya que en ese caso existiría una variable almacenada en localStorage (en el navegador). Esta información la usa *app.js*. Por otro lado, la función *login* (líneas 21-39) realiza una petición POST a la API REST para comprobar si los datos de inicio de sesión son correctos. Si lo es, guarda el usuario localStorage y redirige a la página de inicio (figura 30) y, si no, muestra un mensaje de error.

```
13      var user = JSON.parse(localStorage.getItem("UserLoggedIn"));
14      if(user !== null) {
15          $rootScope.isAuthenticated = true;
16          $rootScope.userEmail = user.email;
17          $rootScope.userRole = user.role;
18          $rootScope.userId = user._id;
19      }
20
21      $scope.login = function (user) {
22          $http.post("http://localhost:8080/users/login",user).then(
23          function successCallback(response) {
24              $scope.response = response.data;
25              $rootScope.isAuthenticated = true;
26              $rootScope.userEmail = response.data.email;
27              $rootScope.userRole = response.data.role;
28              $rootScope.userId = response.data._id;
29              localStorage.setItem('UserLoggedIn', JSON.stringify({"_id":response.data._id,
30                  "email":response.data.email, "role":response.data.role}));
31              $location.path('/home');
32              $rootScope.error = '';
33          },
34          function errorCallback(response) {
35              console.log(response.data.message);
36              $rootScope.loginError = response.data.message;
37          }
38          );
39      };
```

Figura 29 - Fichero mainController.js

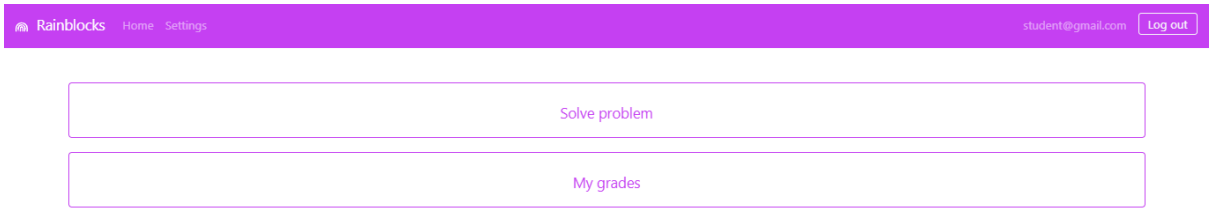

Figura 30 - Página de inicio del rol alumno

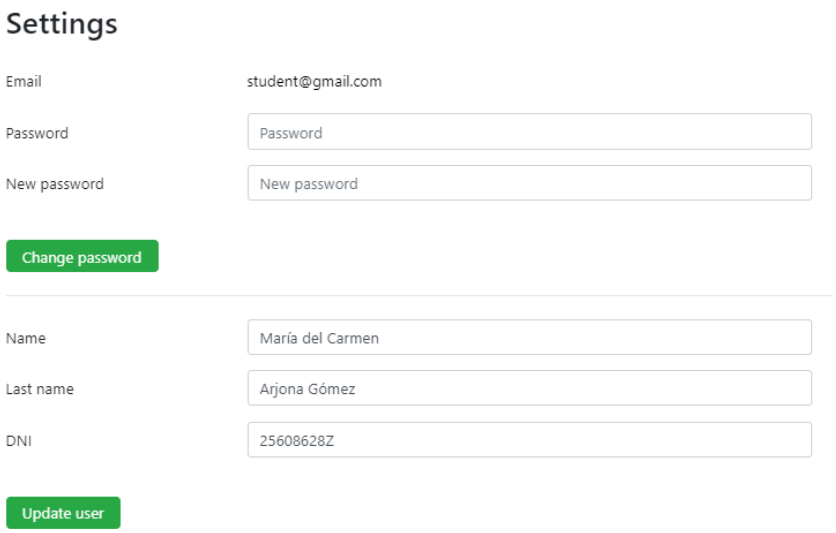

Figura 31 - Página de modificación de datos personales

El fichero *studentController.js* contiene funciones relativas exclusivamente al rol de alumno, pero ninguna de especial complejidad. Simplemente realizan peticiones GET a la API REST y la respuesta que reciben se la asignan a una variable de AngularJS con el objetivo de mostrar los datos recibidos. Las variables de AngularJS están vinculadas, es decir, se actualizan constantemente en la vista en cuanto se modifican en el controlador. Por ejemplo, en la figura 32 vemos cómo se cargan las notas de un alumno mediante una petición GET y luego, de cada problema se cargan sus datos y se almacenan en una variable que la vista mostrará.

```
50    $scope.loadSolutionList = function () {
51      $http.get("http://localhost:8080/solutions/byStudent/"+$rootScope.userId).then(
52          function successCallback(response) {
53            $scope.studentSolutions = response.data;
54            $scope.solutionProblems = [];
55            for(var i = 0; i < $scope.studentSolutions.length; i++) {
56                  $scope.aux_i = i;
57                  $http.get("http://localhost:8080/problems/"+$scope.studentSolutions[i].problemId).then(
58                      function successCallback(response_problem) {
59                        $scope.solutionProblems[response_problem.data._id] = response_problem.data.title;
60                      },
61                      function errorCallback(response_problem) {
62                          console.log(response_problem.statusText);
63                      }
64                  );
65              }
66          },
67          function errorCallback(response) {
68            console.log(response.statusText);
69          }
70        );
71    }
```

Figura 32 - Fichero studentController.js

En el fichero *grades-list.html* (figura 33) se muestra el listado de notas del alumno. Justo al cargarlo (línea 1, ng-init), se llama a la función loadSolutionList para que obtenga los datos necesarios y se actualicen las variables que queremos mostrar. Cuando esto ocurra la vista iterará sobre las notas obtenidas (línea 4, ng-repeat) y mostrará los datos necesarios de cada una (línea 7). Visualmente se corresponde con la figura 34.

```
1  <div class="container my-5" ng-init="loadSolutionList()">
2      <h2>My grades</h2>
3      <ul class="list-group mt-4">
4          <li ng-repeat="solution in studentSolutions"
5              class="list-group-item justify-content-between align-items-center"
6          >
7              <span class="mx-2">{{solutionProblems[solution.problemId]}} - {{solution.grade||'Pending'}}</span>
8          </li>
9        </li>
10     </ul>
11 </div>
```

Figura 33 – Fichero grades-list.html

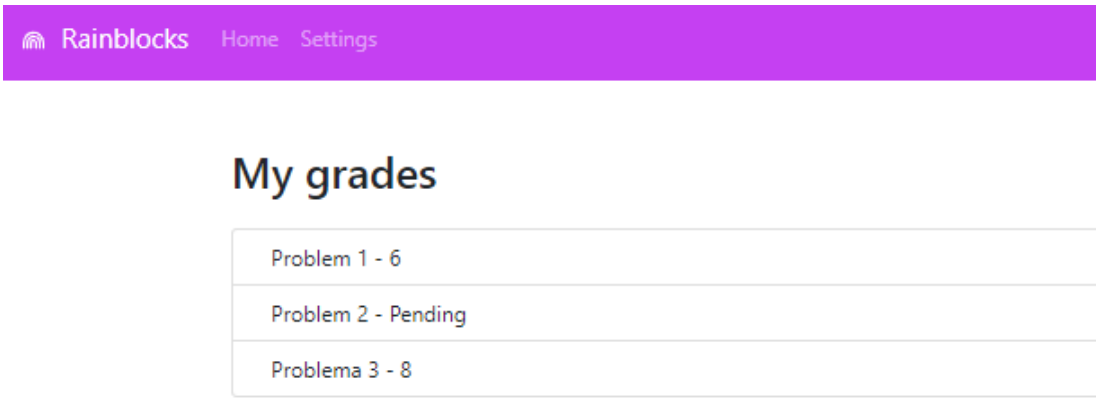

Figura 34 - Página de ver listado de notas

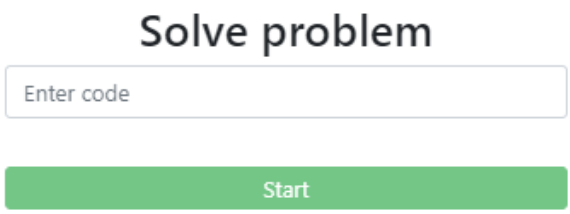

Figura 35 - Página de introducir código de problema

El fichero *problemController.js* contiene las funciones relativas a la parte visual de los bloques. Este fichero lo comparten alumno y profesor, por lo que no entraremos en detalle en las partes del profesor. Vamos a explicar las partes más destacables de este fichero. Al principio contiene una serie de variables que utilizará para mostrar los

bloques, entre las que destacan *templates* (figura 36, líneas 12-34) y *allowedTypes* (figura 36, líneas 35-43). *Templates* contiene una plantilla vacía de todos los tipos de bloques. De aquí se coge la plantilla base cuando añadimos un bloque, pero también se utiliza para mostrar la columna de la izquierda de la figura 37 (que sirve para añadir bloques). La estructura JSON de los bloques contiene a veces unos corchetes vacíos, lo que significa que ahí podrá haber bloques anidados. *AllowedTypes* se utiliza para definir qué tipos de bloques se permiten dentro de cada bloque concreto, según sus tipos.

```
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      ];
35      $scope.allowedTypesMain = ['declaration', 'assignment', 'conditional', 'loopfor', 'loopwhile', 'custom'];
36      $scope.allowedTypesInputs = ['inputboolean', 'inputvariable', 'inputstring', 'inputnumeric', 'comparation', 'negation', 'custom'];
37      $scope.allowedTypesNegation = ['inputboolean', 'inputvariable', 'comparation', 'custom'];
38      $scope.allowedTypesConditions = ['inputboolean', 'inputvariable', 'comparation', 'negation', 'custom'];
39      $scope.allowedTypesCustom =  {  'int':['inputnumeric', 'inputvariable'],
40                                      'double':['inputnumeric', 'inputvariable', "custom"],
41                                      'string':['inputstring', 'inputvariable'],
42                                      'boolean':['inputboolean', 'inputvariable'],
43                                      'var':['inputboolean', 'inputvariable', 'inputstring', 'inputnumeric']};
44
```

Figura 36 - Fichero problemController.js

Figura 37 - Página de resolver problema con bloques

Cada vez que se añade, modifica o elimina un bloque de tipo declaration se llama a la función updateStudentVariables (figura 38, línea 158) que se encarga de actualizar el listado de variables de la aplicación, para que se puedan utilizar como opciones de desplegables en los bloques de tipo assignment y de tipo variable.

```
158     $scope.updateStudentVariables = function () {
159         $scope.variables = [];
160         $scope.updateVariablesRec($scope.solution.studentSolution);
161     }
162
163     $scope.updateVariablesRec = function (blocks) {
164         angular.forEach(blocks, function(value, key) {
165             if(value.type == 'declaration' && value.variable !== '') {
166                 $scope.variables.push(value.variable);
167             }
168             if(value.thens !== undefined && value.thens.length > 0) {
169                 $scope.updateVariablesRec(value.thens);
170                 $scope.updateVariablesRec(value.elses);
171             }
172             if(value.body !== undefined && value.body.length > 0) {
173                 $scope.updateVariablesRec(value.body);
174             }
175         });
176     }
```

Figura 38 - Fichero problemController.js

Para mostrar la vista de los bloques (figura 37), primero se obtiene el id de la solución de la URL actual y mediante una petición GET usando el id, obtenemos los datos de la solución. Luego, de la id de la solución, obtenemos los datos del enunciado del problema y después, con el id del creador del problema, los bloques personalizados del profesor correspondiente. Este proceso se ve en la figura 39.

```
46      if(angular.equals($rootScope.userRole, "teacher")){ ...
97      }else if(angular.equals($rootScope.userRole, "student")){
98          $http.get("http://localhost:8080/solutions/"+currentId).then(
99              function successCallback(response) {
100                 $scope.solution = response.data;
101                 $http.get("http://localhost:8080/problems/"+response.data.problemId).then(
102                     function successCallback(responseProblem) {
103                         $scope.problem = responseProblem.data;
104                         $scope.updateStudentVariables();
105                         //Cargar los custom blocks como estudiante
106                         $http.get("http://localhost:8080/custom-blocks/byTeacher/"+$scope.problem.teacherId).then(
107                             function successCallback(response) {
108                                 $scope.templates[$scope.templates.length-1].blocks = response.data;
109                             },
```

Figura 39 - Fichero problemController.js

Por último, vamos a explicar la estructura del fichero *solution-edit.phtml*, ya que es la vista más compleja de la aplicación. Visualmente se corresponde con la figura 37. Se divide en 4 partes:

- Líneas 2-38: columna de la izquierda. Se muestra la lista de tipos de bloques y la papelera para eliminarlos.
- Líneas 40-71: panel principal. Se muestran los datos del problema, los botones de guardado e info y los bloques ya existentes en la solución, si los hubiera.
- Líneas 73-112: plantilla de estructura de control switch que decide, en función del tipo del bloque, qué plantilla le pertenece.
- Líneas 114-483: plantillas para los distintos tipos de bloques.

La librería que hemos utilizado para el funcionamiento de arrastrar bloques tiene atributos propios que permiten definir cómo queremos que funcione el movimiento de los bloques. Para explicar las más comunes, hemos cogido como ejemplo el panel principal de la vista de solucionar problema, figura 40.

- *dnd-list*: listado de bloques total, del que se van modificando bloques según se muevan.
- *dnd-allowed-types*: listado de los tipos de bloques permitidos en el panel principal. Por ejemplo, no se puede insertar aquí un bloque de tipo boolean.
- *dnd-draggable*: objeto que será movido.
- *dnd-type*: tipo del bloque. Por ejemplo, puede usarse para ver si su tipo es permitido en la lista a la que lo desplacemos.
- *dnd-moved*: qué ocurre con el objeto en esta lista cuando se mueve a otra. En este ejemplo se elimina, para producir el efecto de moverse de un sitio a otro.

```
52              <div class="card-body scroll">
53                  <ul dnd-list="solution.studentSolution"
54                  dnd-allowed-types="allowedTypesMain"
55                  class="main-dropzone">
56                      <li ng-repeat="block in solution.studentSolution"
57                      dnd-draggable="block"
58                      dnd-type="block.type"
59                      dnd-moved="solution.studentSolution.splice($index, 1)"
60                      class="background-{{block.type}}"
61                      >
62                          <div ng-include="'blocks'"></div>
63                      </li>
64                      <li class="dndPlaceholder my-auto">
65                          Drop here
66                      </li>
67                  </ul>
68              </div>
```

Figura 40 - Fichero solution-edit.html

5. Pruebas

Para comprobar el funcionamiento correcto de la aplicación, hemos realizado una prueba por cada caso de uso, siguiendo tanto el escenario principal como el alternativo. Durante el proceso no hemos encontrado errores, lo cual es de esperar pues durante la implementación íbamos comprobando todo lo que hacíamos. Por lo tanto, el proceso de pruebas ha sido satisfactorio. De haber contado con más tiempo, habríamos dado a utilizar la herramienta a alguien ajeno al proyecto para ver cómo interactúa con la aplicación, pero no ha sido posible.

Por otro lado, para comprobar el uso de la herramienta en un caso real, hemos cogido de ejemplo un problema de MATLAB y lo hemos solucionado utilizando nuestra aplicación. El ejercicio que hemos tomado como ejemplo es el de la figura 41 y su correspondiente solución en MATLAB es la figura 42.

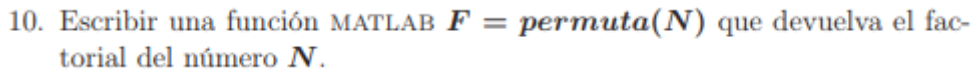
10. Escribir una función MATLAB $\boldsymbol{F = permuta(N)}$ que devuelva el factorial del número $\boldsymbol{N}$.

Figura 41 - Ejemplo ejercicio MATLAB: enunciado

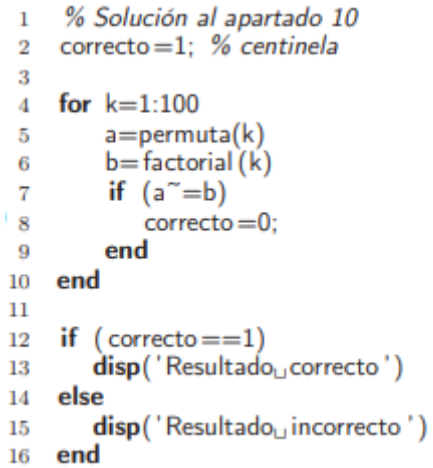
```
1  % Solución al apartado 10
2  correcto=1; % centinela
3
4  for k=1:100
5      a=permuta(k)
6      b=factorial(k)
7      if (a~=b)
8          correcto=0;
9      end
10 end
11
12 if (correcto==1)
13    disp('Resultado correcto')
14 else
15    disp('Resultado incorrecto')
16 end
```

Figura 42 - Ejemplo ejercicio MATLAB: solución original

La figura 43 representa el ejercicio de la figura 41 resuelto con la aplicación que hemos desarrollado. La principal diferencia es que en nuestra aplicación es necesario declarar las variables para poder utilizarlas, el resto es bastante similar. Los bloques rosas se corresponden con los bloques personalizados que ha creado el profesor para que el alumno pueda resolver los ejercicios propuestos.

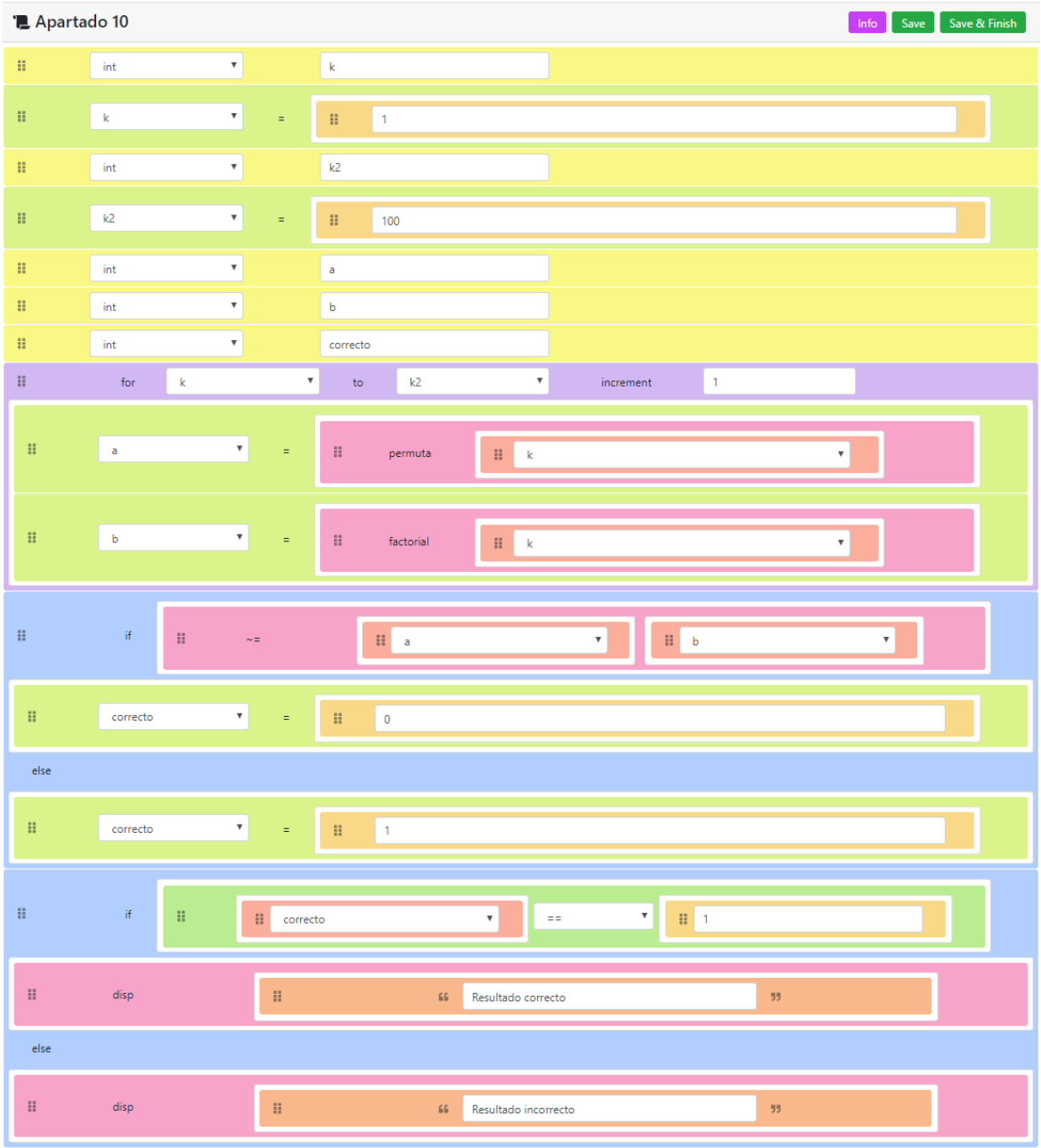

Figura 43 - Ejemplo ejercicio MATLAB: solución aplicación

6. Conclusiones y trabajos futuros

6.1. Dificultades encontradas

Durante la realización de este proyecto han surgido algunas dificultades que han ralentizado su progreso. Las principales son:

- Aprender **tecnologías nuevas**. La mayoría de las tecnologías utilizadas eran nuevas para ambos miembros del grupo y ha sido un reto tanto aprender a utilizarlas como instalarlas e integrarlas para que se conecten todas entre sí.
- Utilizar la librería **Drag & Drop Lists**. Inicialmente esta librería ofrece bloques que se pueden arrastrar y contienen únicamente un campo de texto. Nosotros la hemos adaptado para que depende de lo que necesitemos en cada tipo de bloque, éste tenga título o no, uno o varios campos de texto, desplegables, etc. o incluso para que puedan meterse bloques dentro de bloques. Sin tener esta librería de base el proyecto se habría complicado notablemente, pero aun así tuvimos que personalizarla bastante para que se adaptara al proyecto. Todo esto sin apenas conocer AngularJS.
- Diseñar el **diagrama de clases**. La mayor parte de esta tarea fue sencilla, sin embargo, nos costó diseñar la parte relativa a los bloques. El objetivo era encontrar un diagrama lo más simple posible pero que a su vez abarcase todos los distintos tipos de bloques con sus respectivas estructuras de datos. Como ya hemos mencionado previamente, incluso cambiamos de modelo de base de datos relacional a no relacional en el proceso.
- Establecer **rutas** de la aplicación. Por defecto, AngularJS mantiene la URL base de la aplicación todo el tiempo y en función de la vista que está mostrando añade al final de ésta una almohadilla seguida del identificador correspondiente a la vista (ej: /aplicacion#solucionarProblema). Este método no añade, por ejemplo, el id del ejercicio que estamos solucionando, así que al recargar la página la aplicación no sería capaz de recordar en qué problema estábamos. Entonces necesitábamos URLs absolutas en todo momento por dos motivos principales: para saber si la URL estaba permitida para el rol de ese usuario (ej.: /alumno/solucionarProblema/) y para poder acceder por URL a todas las partes de la aplicación (ej.: /alumno/solucionarProblema/123). La manera de resolver este problema fue usando Node.js y configurando el fichero server.js.

- Introducir **bloques de tipo array**. Estudiamos esta idea, pero decidimos no implementarla ya que aumentaba considerablemente la complejidad del diseño de la base de datos y de la aplicación. Además, al introducir arrays también habría que introducir funciones que permitan trabajar con ellos: merge, push, pop, sort, slice, etc.
- Crear el **algoritmo de corrección**. Aunque esta parte del proyecto fue implementada por el otro miembro del grupo, la base la pensamos entre los dos. Creemos que es una de las partes más complejas y no queríamos que el peso total recayera sobre una persona. Es complicado comparar dos estructuras compuestas similares y obtener una nota de esta comparación. La solución que encontramos más sencilla fue ir quitando de la solución del alumno los bloques coincidentes con la solución del profesor. De esta forma los bloques que hemos podido quitar están bien (y suman nota) y los que no, sobran (y restan un poco de nota). Los que faltan simplemente no sumarán nota. Estamos contentos con el resultado, aunque sabemos que no es una solución perfecta. Sin embargo, sí es una buena solución teniendo en cuenta que teníamos tiempo limitado.

6.2. Posibles mejoras

En este apartado vamos a explicar posibles mejoras para la aplicación que no hemos desarrollado, ya sea por falta de tiempo o porque directamente no entraban en la planificación del proyecto, pero aun así creemos que serían interesantes.

Mejoras relativas a la aplicación en general:

- Mejorar la **seguridad** de la aplicación. Actualmente las contraseñas no se guardan encriptadas, lo cual es un requisito fundamental para cualquier aplicación en producción. Sin embargo, decidimos invertir el tiempo en otras funcionalidades de la aplicación. Adicionalmente, sería necesario securizar las URLs tanto de la aplicación en AngularJS como de la API REST, para limitar las peticiones HTTP y que solo respondan si el usuario es el correspondiente.
- Crear un rol de usuario **administrador** con su correspondiente panel que le permita crear usuarios de cualquier tipo. Actualmente hay que crearlos manualmente desde la base de datos.

- Hacer que la interfaz sea completamente **adaptativo** (*responsive***)**. La versión actual es bastante adaptativo puesto que hemos usado Bootstrap, que está orientado al diseño adaptativo. Sin embargo, no hemos probado la aplicación en otros dispositivos con variables resoluciones y probablemente necesite ajustes antes de ser del todo funcional en éstos.
- Introducir otros **tipos de bloques** de estructura más compleja como vectores, arrays, matrices, etc. Como ya hemos mencionado anteriormente, la complejidad de esto se excedía del proyecto.

Mejoras relativas al alumno:

- Modificar los problemas para tengan un **límite de tiempo** y puedan configurarse como “práctica” o “examen”. Si está configurado como examen, una vez entregado, no podrá volver a editarse y una vez pasado el límite de tiempo se guardará y se cerrará. Si está configurado como práctica, el límite de tiempo le ayudará a orientarse y a practicar de cara al examen.
- Mostrar la **nota provisional automática** al enviar un examen. Para ello, podemos utilizar el mismo algoritmo que le sugiere nota al profesor cuando está corrigiendo. Antes de introducir esta mejora, sería conveniente pulir el algoritmo de corrección.

6.3. Conclusión

En líneas generales, se ha conseguido cumplir con todos los objetivos iniciales de manera satisfactoria. Aunque en un principio nos costaba avanzar porque apenas conocíamos algunas de las tecnologías con las que hemos trabajado, éstas eran afines con los objetivos del proyecto y una vez sentadas las bases, hemos ido avanzando a buen ritmo.

El realizar este proyecto en grupo ha sido todo un acierto pues gracias a eso la aplicación es más completa y tiene más funcionalidades, además de que ha suavizado los impedimentos que supone aprender a trabajar con tecnologías nuevas.

Referencias

- Documentación de AngularJS
  https://docs.angularjs.org/api
- Tutorial de AngularJS
  https://www.w3schools.com/angular/
- Librería Drag & Drop Lists
  http://marceljuenemann.github.io/angular-drag-and-drop-lists/demo/#/types
- Documentación de Bootstrap
  https://getbootstrap.com/docs/4.0/getting-started/introduction/
- Extensión para Chrome Advanced REST Client
  https://chrome.google.com/webstore/detail/advanced-rest-client/hgmloofddffdnphfgcellkdfbfbjeloo/related
- Node.js
  https://nodejs.org/es/
- Spring Tools Suite
  https://spring.io/tools3/sts/all
- Tutorial de Spring Tools con MongoDB
  https://www.codementor.io/gtommee97/rest-api-java-spring-boot-and-mongodb-j7nluip8d
- MongoDB
  https://www.mongodb.com/
- Pencil Project
  https://pencil.evolus.vn/