UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf
Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Simulación y almacenamiento de programas del Easy8 en Web

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Rafael González Carrizo

Tutor: Antonio Martí Campoy

Curso 2018-2019

Resumen

El objetivo de este proyecto consiste en desarrollar una aplicación web para el almacenamiento y simulación de programas escritos en ensamblador para el computador Easy8.

El computador Easy8 es un computador didáctico, con un procesador de 8 bits, un solo registro de propósito general y un set reducido de instrucciones.

Se espera que este proyecto ayude a los alumnos del Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación de la Universitat Politècnica de València, que aprenden cómo funciona un computador y el lenguaje ensamblador usando el Easy8.

La aplicación web cuenta con un gestor de ficheros para que el usuario pueda organizar sus programas, un editor de código con resaltado de sintaxis y una representación gráfica del emulador.

Palabras clave: aplicación web, simulador, ensamblador, Vue, Laravel, PHP

Abstract

The objective of this project is to develop a web application for the storage and simulation of programs written for the Easy8 computer.

The Easy8 computer is a didactic computer, with an 8-bit processor, a single general-purpose register and a reduced instruction set.

It is expected that this project will help the students of the Bachelor's Degree in Telecommunication Technologies and Services Engineering of the Universitat Politècnica de València, who learn how a computer and the assembly language work using the Easy8.

The web application has a file manager that allows users to organize their programs, a code editor with syntax highlighting and a graphic representation of the emulator.

Keywords: web application, simulator, assembly, Vue, Laravel, PHP

Tabla de contenidos

Índice de figuras

Glosario

- **CSS**: lenguaje para establecer el aspecto visual de una página web.
- **Ensamblador**: lenguaje de programación de bajo nivel.
- **Función *hash***: función determinista que recibe como entrada datos de longitud arbitraria y devuelve datos de una longitud fija, siendo virtualmente imposible obtener los datos de entrada conociendo únicamente la salida.
- **HTML**: lenguaje de marcado para definir el contenido de una página web.
- **HTTP**: protocolo de comunicación sobre el que funciona la World Wide Web.
- **HTTPS**: extensión del protocolo HTTP en el que se cifra la comunicación.
- **JavaScript**: lenguaje de programación interpretado, de alto nivel que permite construir páginas web interactivas.
- **Laravel**: *framework* de PHP.
- **MySQL**: sistema de gestión de bases de datos relacional.
- **PHP**: lenguaje de programación de propósito general que se ejecuta normalmente en el lado del servidor
- **PostgreSQL**: sistema de gestión de bases de datos relacional.
- **Sistema de gestión de bases de datos**: software capaz de gestionar el almacenamiento, la recuperación y manipulación de grandes cantidades de datos.
- **SVG**: formato de imagen vectorial basado en XML.
- **URI**: cadena de caracteres con un formato determinado que identifica, sin ambigüedad, un recurso.
- **VueJS**: *framework* de Vue.
- **XML**: lenguaje de marcado para codificar documentos que pueda ser fácilmente leíble tanto por máquinas como por humanos.

1. Introducción

Conocer el funcionamiento de los computadores y del lenguaje ensamblador es fundamental para muchos ingenieros. Este es el caso de los alumnos del Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación de la Escuela Técnica Superior de Ingeniería de Telecomunicación de la Universitat Politècnica de València. Los alumnos de Fundamentos de Computadores estudian qué es y cómo funciona un procesador y un computador usando el Easy8.

El computador Easy8 trata de hacer que la primera toma de contacto del alumno con estas máquinas presente pocas complicaciones, pues su procesador trabaja con un conjunto muy reducido de instrucciones y registros.

El objetivo de este trabajo consiste en desarrollar un simulador del computador Easy8 para que los alumnos realicen las prácticas y les sirva de apoyo en las clases de teoría.

1.1 Motivación

Este trabajo de fin de grado se ofertó inicialmente por el DISCA como dos trabajos distintos: por un lado, el **desarrollo de un simulador web** del computador Easy8 y, por otro, el **desarrollo de un servidor web** para almacenar sus programas. Ambos sistemas tendrían que estar preparados para trabajar juntos.

Aunque inicialmente escogí el desarrollo del servidor web porque me interesa el desarrollo en el lado del servidor, también me llamaba la atención el primer trabajo porque me gusta el lenguaje ensamblador y nunca había hecho nada parecido. Finalmente, se decidió unir los dos trabajos en uno solo: Simulación y almacenamiento de programas del Easy8 en Web.

Otra de las razones por las que quise escoger este proyecto es porque puede ser usado por los alumnos de Fundamentos de Computadores, es decir, puede tener un impacto real.

1.2 Objetivos

Este trabajo consiste, esencialmente, en el desarrollo de una plataforma web que permita:

1. **Ensamblar y ejecutar** los programas del Easy8 en el navegador del usuario.
2. **Almacenar** en un servidor programas escritos para el Easy8.

La aplicación debe disponer de una interfaz gráfica desde la que los usuarios puedan **gestionar sus ficheros**, pudiendo organizarlos en carpetas y subcarpetas. Además, debe tener un editor de texto para escribir los programas que resalte las palabras clave.

Por último, el simulador **deberá representar los periféricos de manera gráfica**, imitando el aspecto del prototipo físico del Easy8 (ver *Figura 1*), y varias vistas para conocer el estado de los registros, los *flags* y la memoria que permitan al usuario depurar el programa con facilidad.

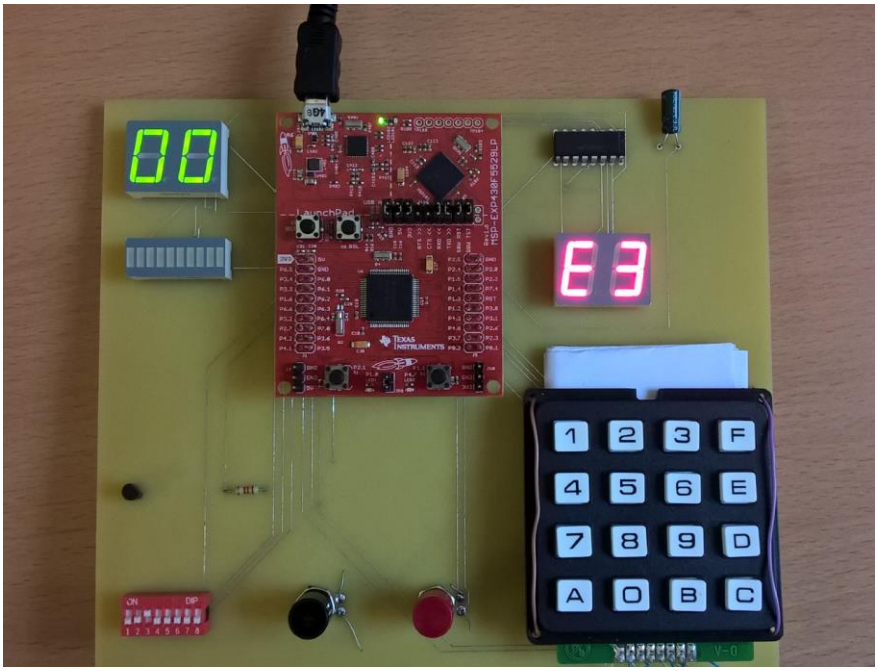

Figura 1: Emulador del Easy8.

Al tratarse de una aplicación web se consiguen varios beneficios:

1. La aplicación será multiplataforma.
2. No necesitará instalación.
3. El usuario tendrá acceso a sus programas independientemente de dónde se encuentre, siempre que tenga un computador con acceso a Internet.
4. Las actualizaciones serán inmediatas.

En la medida de lo posible, se debe reducir el trabajo que tenga que realizar el servidor. Por ejemplo, realizando el ensamblaje y la simulación en el lado del cliente usando JavaScript. Así, el trabajo del servidor consistirá principalmente en autenticar a los usuarios y entregarles sus ficheros de código.

1.3 Metodología

Para desarrollar el proyecto se ha usado la metodología de desarrollo en cascada. Esta metodología divide el proceso en varias fases: toma de requisitos, análisis, diseño, implementación, verificación y mantenimiento (ver *Figura 2*). Es un proceso lineal en el que no se debe pasar a la siguiente etapa hasta que la anterior ha sido validada, aunque en la práctica se producen solapamientos (Jones, 2017).

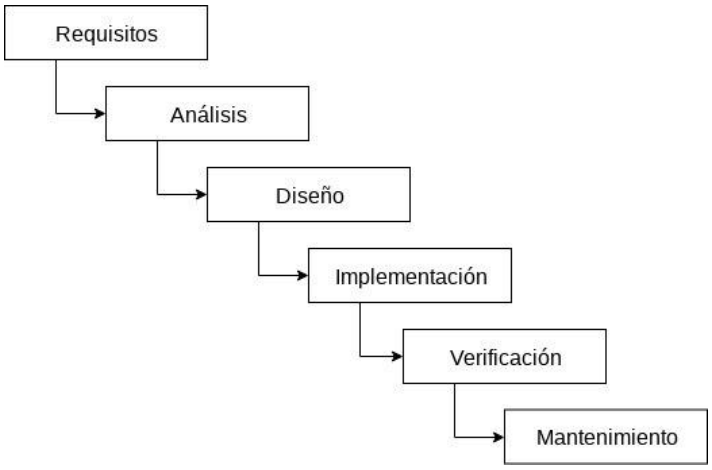

Figura 2: Desarrollo en cascada

Una de las desventajas de esta metodología es que los clientes pueden no conocer exactamente los requisitos hasta que comienzan a ver el software funcionando y modificar los requisitos en fases más tardías puede ser muy costoso.

Sin embargo, en este proyecto los requisitos están bien definidos, así que se ha optado por esta metodología porque es sencilla de entender.

1.4 Estructura

Esta memoria está estructurada en nueve capítulos, siendo el primero de ellos este.

En el capítulo 2, *Estado del arte*, se justificará el desarrollo de este proyecto comparándolo con las alternativas ya existentes.

En el capítulo 3, *El computador Easy8*, se describirá cómo es el computador y el procesador Easy8: cómo es el hardware que lo compone, de qué periféricos dispone y cómo es su arquitectura del juego de instrucciones.

En el capítulo 4, *Análisis*, tomando como referencia el computador Easy8 descrito anteriormente, se concretarán los requisitos que debe cumplir la plataforma web que se va a construir para simularlo.

En el capítulo 5, *Diseño*, se determinará la arquitectura de la aplicación, que afectará de manera significativa en la etapa de implementación.

En el capítulo 6, *Implementación*, se comentarán los aspectos más importantes de la implementación de la plataforma, poniéndose el énfasis en el funcionamiento del ensamblador y la ejecución del código del Easy8.

En el capítulo 7, *Despliegue*, se indicará cómo hacer el despliegue del proyecto en un servicio de computación en la nube llamado Heroku.

En el capítulo 8, *Pruebas*, se comprobará la corrección del proyecto con pruebas de funcionamiento y visualización.

En el capítulo 9, se cerrará la memoria exponiendo las conclusiones sacadas tras el desarrollo de todo el proyecto.

2. Estado del arte

En este capítulo se va a hacer un breve análisis de la herramienta que se usa actualmente para simular el Easy8 y otras herramientas existentes basadas en la web para simular computadores.

2.1 Simulador previo del Easy8

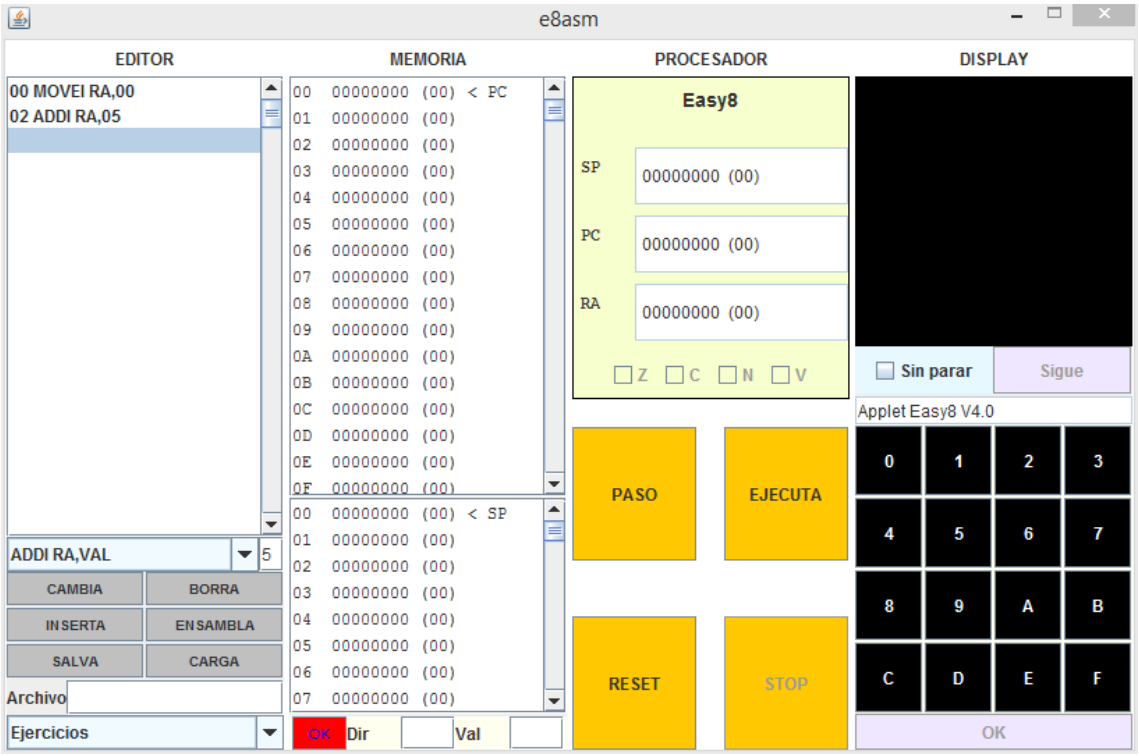

Figura 3: Simulador del Easy8 basado en Java.

El Easy8 ya dispone de un simulador[1], que se puede ver en la *Figura 3*, desarrollado como un applet de Java por el profesor del Departamento de Informática de Sistemas y Computadores, Alberto González Téllez. Los dos principales inconvenientes de la aplicación actual y que se van a mejorar en este trabajo son:

1. **Editor basado en selectores**. El editor del simulador original consiste en una serie de selectores para introducir las instrucciones. Es decir, el programador selecciona la instrucción de entre una serie de opciones en vez de escribirla con su teclado.
   Aunque esta opción puede evitar que se cometan errores de sintaxis, ofrece una experiencia lenta, poco realista y alejada de la experiencia que ofrece cualquier entorno de desarrollo.

[1] https://riunet.upv.es/handle/10251/5177

2. **Pocos periféricos**: sólo dispone de un teclado y un display con dos dígitos de 7 segmentos.
3. **Basado en Java**: es necesario disponer de Java instalado en el equipo para que funcione.

2.2 Simuladores web

En Internet se pueden encontrar varios simuladores de computadores. Algunos de ellos se distribuyen como aplicaciones independientes y otros están basados en la web. Como en este trabajo se va a desarrollar un simulador web, en esta sección se dará una breve descripción de algunos simuladores de este tipo.

2.2.1 Simple 8-bit Assembler Simulator[2]

Se trata de un ensamblador web de 8 bits. A diferencia del que se va a desarrollar en este trabajo, este tiene un juego de instrucciones más complejo y varios registros de propósito general, por lo que podría ser menos interesante para fines didácticos.

El único periférico que posee este simulador es un *display* de caracteres.

2.2.2 Assembly x86 Emulator[3]

Simulador para la arquitectura x86, por lo que la curva de aprendizaje es mucho más pronunciada. Cuenta con varios periféricos: memoria EEPROM, teclado y bombillas.

2.2.3 Yasp[4]

Probablemente Yasp sea el proyecto más similar al Easy8 que se ha encontrado, pues también se trata de un simulador creado con fines didácticos.

Cuenta con 43 instrucciones (73, si se cuentan las sobrecargas), soporte de interrupciones y varios periféricos (tres LED, un potenciómetro y dos botones).

Además, dispone de un gestor de ficheros que funciona con el almacenamiento local del navegador.

2.3 Propuesta

Las razones que llevaron a los profesores de la asignatura a utilizar el Easy8 quedan fuera del alcance de este TFG. Pero una vez elegido dicho computador, es necesario disponer de un simulador para facilitar el aprendizaje.

---

[2] https://schweigi.github.io/assembler-simulator/

[3] https://carlosrafaelgn.com.br/asm86/

[4] http://yasp.me/

En este proyecto se tratará de ofrecer una versión web del Easy8 que ponga solución a los inconvenientes comentados en el apartado *Simulador previo del Easy8*:

1. El usuario podrá escribir sus programas mediante un **editor de texto** que resaltará las palabras clave para facilitar la detección de errores de escritura. Así, se ofrecerá una experiencia más próxima a los entornos de desarrollo profesionales.

2. A los periféricos que ya había se añadirán nuevos: *switches*, un sensor de temperatura y varios LED.

3. Se usará tecnología soportada de manera nativa por los navegadores web para evitar que el usuario tenga que instalar cualquier tipo de plugin o software en su máquina.

Frente a los simuladores comentados en el apartado *Simuladores web*, el simulador que se va a desarrollar aquí ofrecerá una experiencia **más simple** y la capacidad de **almacenar los programas** en el servidor, dos características que resultarán útiles a los alumnos de Fundamentos de Computadores.

3. El computador Easy8

El Easy8 es un computador didáctico que sigue la arquitectura von Neumann, es decir, cuenta con una unidad de procesamiento, una memoria principal, que es compartida por las instrucciones y los datos, y el sistema de entrada y salida.

3.1 Procesador

La palabra del procesador del Easy8 es de 8 bits, lo que significa que sus registros tienen este tamaño y que su memoria y puertos son direccionados con 8 bits.

Los registros con los que cuenta el procesador son:

- Program counter (PC): es el contador de programa, que contiene la dirección de la próxima instrucción que se ejecutará.
- Register accumulator (RA): es el único registro de propósito general del Easy8. Se trata de un acumulador que actúa a la vez como operando y como resultado de la operación en muchas de las instrucciones de la máquina.
- Stack pointer (SP): contiene la dirección en memoria del último dato introducido en la pila.

Todos los registros se inicializan a cero.

Unidad aritmeticológica

La unidad aritmeticológica del Easy8 es muy sencilla. Consiste en un circuito sumador que puede realizar sumas y restas con números representados en complemento a dos.

Además, hay disponibles cuatro bits de estado que se activan para ciertos resultados:

- Flag C: se establece a 1 cuando el resultado de la última instrucción aritmética produce un acarreo en el último bit.
- Flag Z: se pone a 1 cuando el resultado de la operación aritmética es cero.
- Flag N: se establece a 1 cuando el resultado de la operación aritmética es negativo.
- Flag V: valdrá 1 cuando se haya producido un desbordamiento.

3.2 Memoria

La memoria principal del Easy8 está formada por palabras de 8 bits. Cada una de estas palabras tiene asociada una dirección que el procesador podrá usar para manipularlas. Como este solo cuenta con registros de 8 bits, como máximo podrá direccionar $2^8 = 256$, desde la 0 hasta la 255.

3.3 Periféricos

El Easy8 dispone de 256 puertos que permiten la comunicación entre el procesador y los periféricos. Los periféricos de entrada escribirán en el puerto que tengan asignado y el procesador podrá leer de él. De manera similar, para los periféricos de salida, el procesador podrá escribir en sus puertos para enviar órdenes y datos.

Cabe la posibilidad de que un puerto sea de entrada y salida y también es posible que un mismo periférico ocupe varios puertos.

En la *Figura 4* se resumen los periféricos que incorpora actualmente el Easy8, junto con el puerto en el que están conectados.

Periférico	Tipo	Puerto
Teclado hexadecimal	Entrada	0x00
Display de dos dígitos	Salida	0x01
Botón rojo	Entrada	0x02
Sensor de temperatura	Entrada	0x06
Interruptor	Entrada	0x08
Barra de 8 LEDS	Salida	0x09
Botón negro	Entrada	0xAA

Figura 4: Periféricos del Easy8

3.4 Juego de instrucciones

Las instrucciones del Easy8 se codifican en una o dos palabras. La primera representa el código de operación y la segunda, si la hubiese, será un parámetro que puede ser una dirección o un valor inmediato.

A continuación, se muestra el juego completo de instrucciones del Easy8.

3.4.1 Instrucciones codificadas en un byte

Instrucción	Código de operación	Descripción
STOP	0x15	Detiene la ejecución del programa.
INC RA	0x07	Incrementa el valor del registro RA en una unidad.
DEC RA	0x08	Decrementa el valor del registro RA en una unidad.
PUSH RA	0x0F	Guarda el valor de RA en la pila.
POP RA	0x10	Elimina la cima de la pila y guarda su contenido en el registro RA.
RET	0x12	Retorno de llamada a subrutina. La dirección de retorno se almacena en la pila.

Figura 5: Instrucciones codificadas en un byte

3.4.2 Instrucciones codificadas en dos bytes

Instrucción	Código de operación	Parámetro	Descripción
MOVEI RA, VAL	0x00	Valor	Carga en el registro RA el valor VAL.

MOVE RA, DIR	0x01	Dirección	Carga en RA el dato que haya en la posición DIR de la memoria.
MOVE DIR, RA	0x02	Dirección	Guarda en la posición DIR el valor del registro RA.
ADDI RA, VAL	0x03	Valor	Suma al registro RA el valor VAL.
ADD RA, DIR	0x04	Dirección	Suma al registro RA el valor que haya en la posición DIR.
SUBI RA, VAL	0x05	Valor	Como ADDI, pero para la resta.
SUB RA, DIR	0x06	Dirección	Como ADD, pero para la resta.
COMPAREI RA, VAL	0x09	Valor	Compara RA con VAL.
COMPARE RA, DIR	0x0A	Dirección	Compara RA con el valor de DIR.
JUMP DIR	0x0B	Dirección	La ejecución salta a la dirección DIR.
JLESS DIR	0x0C	Dirección	Salta a DIR si, en la última comparación, RA < VAL.
JGREATER DIR	0x0D	Dirección	Salta a DIR si, en la última comparación, RA > VAL.
JEQUAL DIR	0x0E	Dirección	Salta a DIR si, en la última comparación, RA = VAL.
CALL DIR	0x11	Dirección	Salta a DIR guardando el PC en la cima de la pila.
IN PUERTO	0x13	Puerto	Guarda en RA el valor del puerto PUERTO.
OUT PUERTO	0x14	Puerto	Guarda en PUERTO el valor de RA.

Figura 6: Instrucciones codificadas en dos bytes

Cabe destacar que las instrucciones aritméticas (INC, DEC, ADD, ADDI, SUB, SUBI, COMPARE y COMPAREI) actualizan los *flags*. Por ejemplo, si el resultado de una resta da como resultado un cero, el *flag* Z se activaría.

Las instrucciones COMPARE y COMPAREI hacen una resta para realizar la comparación entre los dos operandos. El resultado de la resta no es almacenado en el registro RA pero sí que se actualizan los *flags*. Cuando se ejecuta alguna de las instrucciones de salto condicional (JLESS, JEQUAL, JGREATER), el procesador puede conocer el resultado de la comparación a partir de los valors de los *flags*.

Por ejemplo, si el usuario compara el 5 con el 6, el procesador calculará 5-6=-1. Como el resultado de la operación es negativo (el *flag* N estará activado), significa que el segundo operando es mayor. Si fueran iguales, el resultado de la resta habría sido cero (el *flag* Z estaría activado). Si el primer operando hubiese sido mayor, los *flags* Z y N estarían desactivados, porque el resultado de la resta habría sido positivo.

4. Análisis

4.1 Introducción

4.1.1 Propósito

En este capítulo se establecerán los requisitos del proyecto siguiendo el estándar IEEE 830-1998.

4.1.2 Ámbito del sistema

En este trabajo se va a implementar una aplicación web capaz de simular el computador Easy8, descrito en el capítulo *El computador Easy8,* y de gestionar sus programas. El nombre de la aplicación será Easy8 y se espera que ayude a los alumnos de Fundamentos de Computadores a comprender el funcionamiento de un computador.

4.2 Descripción general

4.2.1 Perspectiva del producto

El Easy8 será una aplicación web. Será un producto independiente que no interactuará con ningún otro sistema informático.

4.2.2 Funciones del producto

La plataforma que se va a construir ofrecerá las siguientes funciones:

- **Gestión y autenticación de usuarios**: la plataforma permitirá a los usuarios registrarse para poder almacenar sus programas.
- **Gestión de ficheros**: se implementará un gestor de archivos que permita al usuario organizar sus programas en carpetas y subcarpetas.
- **Simulación del Easy8**: ensamblado y ejecución de código.
- **Interfaz gráfica del simulador**: la aplicación contará con una representación gráfica del computador Easy8 para que la interacción sea más realista.

4.2.3 Características de los usuarios

Los usuarios de la aplicación serán los alumnos de la asignatura de Fundamentos de Computadores, que, seguramente, no tendrán ningún tipo de experiencia con el lenguaje ensamblador, aunque sí con la web.

4.2.4 Restricciones

La aplicación deberá ser completamente funcional en cualquier versión reciente de los navegadores populares: Firefox, Google Chrome, Opera y Microsoft Edge.

La aplicación se visualizará correctamente en pantallas que, al menos, tengan una resolución de 1366x768. Quedan excluidos los dispositivos móviles.

4.3 Requisitos específicos

4.3.1 Requisitos funcionales

4.3.1.1 Autenticación de usuarios

RF01	Registro de usuarios
Los usuarios tendrán que registrarse para poder hacer uso del sistema. Introducirán datos como su correo electrónico, contraseña, nombre y apellidos. Los usuarios recibirán un correo electrónico para validar su cuenta.	

RF02	Autenticación de usuarios
Los usuarios podrán iniciar sesión en el sistema para poder acceder a sus archivos y al simulador. Se usará el correo electrónico y la contraseña para iniciar sesión.	

RF03	Recuperación de contraseña
Los usuarios podrán iniciar sesión en el sistema para poder acceder a sus archivos y al simulador. Se usará el correo electrónico y la contraseña para iniciar sesión.	

4.3.1.2 Gestión de ficheros

RF04	Creación de carpeta
El usuario podrá crear una carpeta en su sistema de ficheros. Para ello, tendrá que darle un nombre que será único dentro del directorio en el que esté.	

RF05	Creación de fichero de código
El sistema permitirá al usuario crear un fichero de código dentro de alguna de sus carpetas. El nombre será único dentro del directorio en el que esté.	

RF06	Apertura de carpeta
El usuario podrá abrir una carpeta para visualizar su contenido. El sistema garantizará que el usuario no accede a una carpeta que no le pertenece.	

RF07	Apertura de fichero de código
El usuario podrá abrir el fichero de código para poder ver y alterar su contenido.	

El sistema garantizará que el usuario no accede a un fichero que no le pertenece.

4.3.1.3 Simulador

RF08	Ensamblaje del código
El sistema ensamblará el código que haya en el fichero que tenga abierto el usuario. Se dará soporte a todo el juego de instrucciones del Easy8 explicado en el capítulo *El computador Easy8*. Además, se añadirá soporte para etiquetas, de modo que el usuario pueda hacer referencia a direcciones de memoria usando nombres que le sean fáciles de recordar.	

RF09	Coloreado de código
El editor de código coloreará las palabras reservadas para que el usuario pueda detectar más fácilmente los errores de escritura.	

RF10	Manipulación del código
El editor de código permitirá al usuario copiar, cortar y pegar texto	

RF11	Ejecución del código
El simulador ejecutará el código ensamblado. Permitirá al usuario ejecutarlo instrucción por instrucción, lo que facilitará la depuración de los programas, o sin paradas.	

RF12	Visualización de la memoria
El sistema debe ofrecer varias vistas de la memoria. Una de ellas será la vista de programa, otra es la vista de código y la última la de pila. • La **vista de programa** será una vista de la memoria en la que el foco se pondrá sobre la instrucción que se está ejecutando. • La **vista de datos** mantendrá el foco en la última posición de memoria modificada por el programa. • La **vista de pila** mantendrá el foco en la posición del puntero de pila.	

RF13	Visualización de puertos
El sistema ofrecerá una vista del estado de los puertos. Se mantendrá el foco en el último puerto modificado.	

RF14	Visualización de registros
El sistema mostrará el estado de los registros.	

RF15	Visualización de *flags*
El sistema mostrará el estado de los *flags*.	

RF16	Visualización del hardware
El simulador mostrará una representación gráfica del hardware del Easy8 (*displays*, botones, *switches*, sensor de temperatura), tratando de imitar al emulador ya existente, para que el usuario pueda interactuar con él como si fuera un dispositivo físico.	

RF17	Limpieza de memoria y registros
El simulador permitirá al usuario limpiar el contenido de la memoria y los registros.	

4.3.2 Requisitos no funcionales

4.3.2.1 Tiempo rápido de respuesta

- RNF01: la carga inicial de la aplicación durará unos 2 segundos, como máximo.
- RNF02: la apertura de documentos y carpetas debe suceder en menos de 0,5 segundos.
- RNF03: el ensamblaje debe durar menos de 0,5 segundos.

4.3.2.2 Seguridad

- RNF04: el servidor verificará cada petición del usuario para impedir que acceda a recursos que deberían ser inaccesibles para él.
- RNF05: la web podrá ser accesible a través de HTTPS.
- RNF06: cualquier clave de usuario tendrá aplicada una función *hash*.

4.3.2.3 Facilidad de uso

- RNF07: la plataforma no exigirá al usuario la descarga de ningún *plugin* o cualquier tipo de complemento para poder funcionar.

4.3.2.4 Soporte

- RNF08: la aplicación deberá funcionar, al menos, en versiones recientes de los navegadores más importantes: Mozilla Firefox, Google Chrome, Opera y Microsoft Edge.

- RNF09: la aplicación debe visualizarse correctamente en pantallas con una resolución igual o mayor que 1366x768. Los dispositivos móviles quedan excluidos.

- RNF 10: el servidor debe poder ejecutarse en instancias de Heroku y en Debian 9.

- RNF11: se debe poder generar una versión ligera del proyecto que incluya únicamente el módulo de simulación (editor de texto, ensamblaje y ejecución de código), para que se pueda usar la plataforma sin un servidor web.

4.3.2.5 Desarrollo

- RNF12: se usará software libre siempre que se pueda.

- RNF13: la aplicación debe ser fácil de desplegar y de mantener.

- RNF14: el set de instrucciones soportadas por el procesador se debe poder extender fácilmente.

5. Diseño

5.1 Arquitectura de tres capas

La aplicación sigue una arquitectura de tres capas. En este tipo de arquitectura, el sistema se divide en la capa de presentación, la capa de lógica de negocio y la capa de datos. La comunicación entre estas capas es lineal, estando siempre la capa de lógica de negocio como intermediaria entre la de presentación y la de datos.

Esta arquitectura permite que cada una de las capas estuviese en una máquina distinta y, siempre y cuando la interfaz entre cada par de capas fuera la misma, podrían ser actualizadas sin afectar al funcionamiento del resto de capas.

En el caso concreto de este proyecto, la capa de presentación residirá en el navegador del usuario, la capa de negocio estará dividida entre el navegador del usuario y el servidor web y la capa de datos estará únicamente en el servidor (ver *Figura 7*).

El motivo por el que la capa de negocio se ha dividido en dos se explicó en el apartado *Objetivos*: por un lado, la lógica referente a la gestión de ficheros y de usuarios ocurre principalmente en el servidor, mientras que el ensamblado y la ejecución de código del Easy8 se realiza exclusivamente en el cliente para ahorrar recursos al servidor.

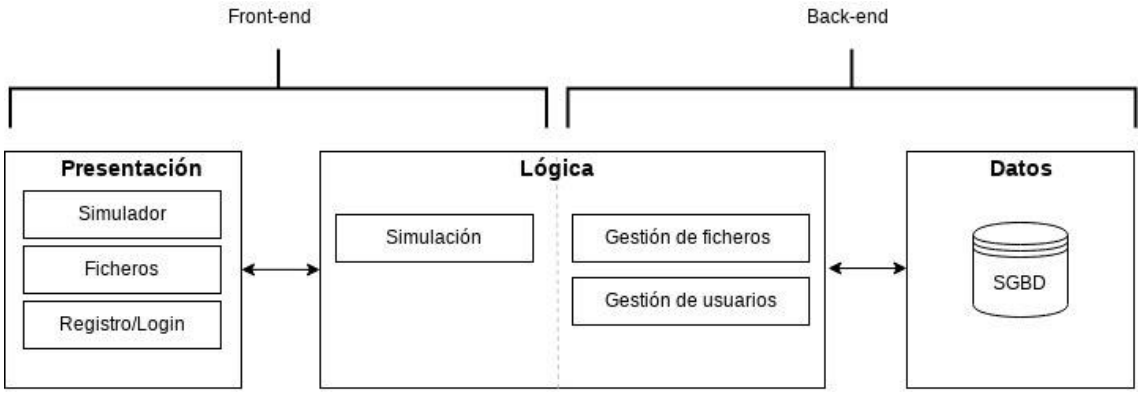

Figura 7: Arquitectura de tres capas del Easy8

5.2 Arquitectura REST

La aplicación, además de poder ser vista desde el punto de vista de las tres capas, puede dividirse en front-end y back-end, entendiéndose el front-end como aquella parte de la aplicación que se ejecuta sobre el navegador web del usuario (correspondería a la interfaz gráfica y la lógica de la simulación) y el back-end, que es la parte que se encuentra en el servidor (lógica de gestión de ficheros, usuarios y la base de datos).

Para hacer posible la comunicación entre el front-end y el back-end, este último ofrecerá una API que será consumida por el front-end.

Esta API utilizará una arquitectura REST o, como mínimo, tratará de cumplir al máximo sus principios.

REST significa Transferencia de Estado Representacional (GeeksforGeeks, s.f.). Es un estilo de arquitectura usado para crear servicios web que se apoya en el protocolo HTTP y aprovecha los verbos de este (GET, POST, PUT, PATCH, DELETE, etc.) para manipular recursos.

Se dice que una API es RESTful cuando implementa la arquitectura REST.

El motivo por el que se ha decidido usar REST ha sido, además de por su sencillez y popularidad, por estar basado en el protocolo HTTP, el mismo que hace posible el funcionamiento de la World Wide Web.

Hay cinco restricciones que deben cumplir un servicio REST para considerarse como tal:

- **Interfaz uniforme** entre el cliente y el servidor:
    - Las API RESTful **se basan en recursos** identificados a través de una URI.
    - Cuando el cliente tiene la representación de un recurso, tiene también la información suficiente para alterarlo.
    - Mensajes autodescriptivos: en los mismos mensajes de petición y respuesta se indica cómo están codificados los datos mandados para que tanto el cliente como el servidor sepan cómo interpretarlos.
    - Hipermedia como motor de estado de la aplicación: junto con la respuesta a cada petición, el servidor debería devolver una serie de links que describen las acciones que se pueden realizar sobre el recurso.
- **Arquitectura cliente-servidor**: gracias a la interfaz uniforme, el cliente no tiene por qué conocer los detalles sobre cómo se almacena la información en el servidor. Del mismo modo, el servidor tampoco tiene razón alguna para preocuparse por la interfaz gráfica o por el estado del usuario.
- **Sin estado**: en cada petición que haga el cliente al servidor se debe enviar toda la información necesaria para que este último pueda ejecutar su trabajo. Esto es así porque el servidor no almacena el contexto en el que se realizan las peticiones.
- **Cacheable**: al igual que ocurre con la web, los clientes pueden guardar en caché la información devuelta por el servidor. Es responsabilidad del servidor indicar, de manera implícita o explícita, si sus respuestas pueden o no ser almacenadas en caché.
- **Sistema por capas**: los clientes no conocen y no se tienen que preocupar de si están haciendo solicitudes al servidor final o a alguna capa intermedia que se haya introducido para mejorar la escalabilidad del sistema.

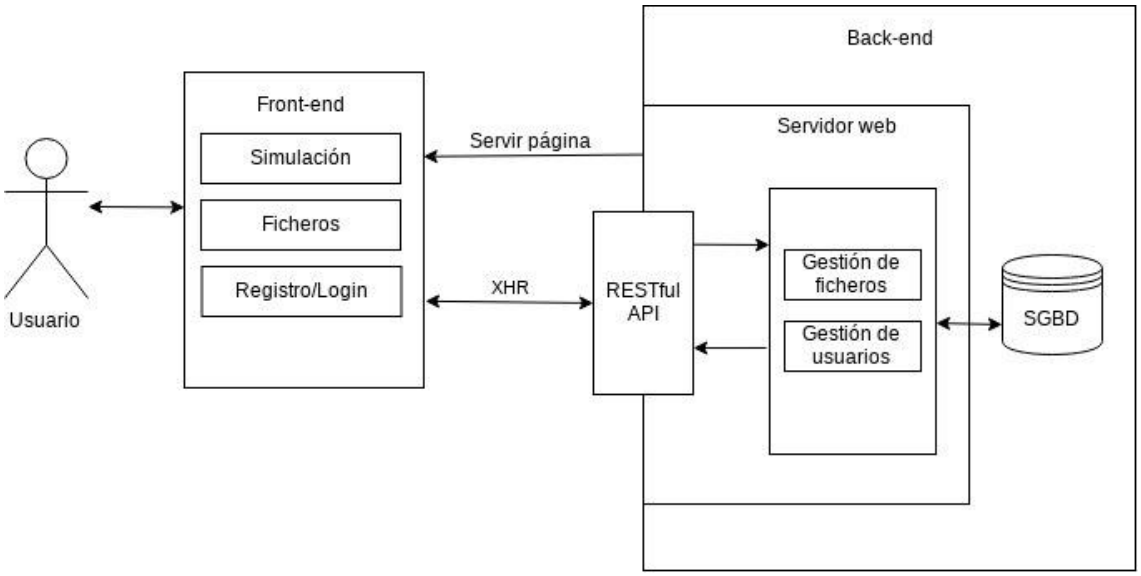

Figura 8: Comunicación a través de la API RESTful

En el diagrama de la *Figura 8* se representan los dos grandes bloques y cómo interactuarán entre ellos. Por una parte, el back-end se encargará de servir la página al usuario. En ese punto, el usuario podrá interactuar con la página que, para obtener y guardar datos, utilizará la interfaz XHR[5] para realizar peticiones HTTP a la API RESTful.

Destacar que los datos que se reciben y envían a través de la API estarán codificados en formato JSON. Como el front-end está escrito principalmente en JavaScript, resulta muy sencillo utilizar este formato: JSON significa "JavaScript Object Notation".

5.3 Base de datos

El sistema debe tener la capacidad de almacenar los datos de los usuarios y sus ficheros. Para hacer esto posible, se va a usar un sistema de gestión de bases de datos relacional. En concreto, la aplicación soportará bases de datos MySQL y PostgreSQL.

El motivo por el que se ha decidido usar un sistema relacional frente a otros tipos ha sido principalmente por su popularidad y por una característica fundamental de este tipo de sistemas: la integridad referencial.

La integridad referencial permite, en el contexto de este proyecto, que, por ejemplo, al eliminar un usuario de la plataforma, también se eliminen automáticamente todos sus ficheros. De manera similar, garantiza que, al eliminar una carpeta, se elimine también todo su contenido.

[5] XHR significa XMLHttpRequest. Es una interfaz para realizar peticiones HTTP desde JavaScript. No está limitada sólo al formato XML, como sugiere su nombre; tiene ese nombre por motivos históricos.

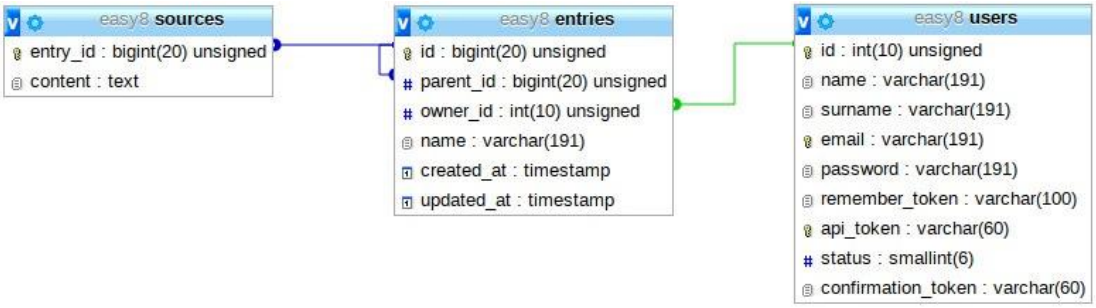

Figura 9: Esquema de la base de datos

En la *Figura 9* se muestran las tres tablas que se han definido en este proyecto junto con las relaciones entre ellas. A continuación, se describe cada una de las tablas:

5.3.1 Tabla *users*

La tabla *users* almacena a cada uno de los usuarios de la plataforma. Estas son sus columnas:

- id: identificador único.
- name: nombre del usuario.
- surname: apellidos del usuario.
- email: correo electrónico que utilizará para iniciar sesión.
- password: contraseña del usuario a la que se ha aplicado una función hash.
- api_token: es una cadena de caracteres aleatoria, secreta y única que sirve para autenticar al usuario.
- remember_token: es usada por el sistema de recuperación de contraseña.
- confirmation_token: es usada por el sistema de confirmación del correo electrónico.
- status: campo booleano que almacena true si el usuario ha validado su cuenta de correo electrónico,

5.3.2 Tabla *entries*

Esta tabla representa una entrada en el sistema de ficheros, que bien puede ser una carpeta o un fichero de código ensamblador.

Cada entrada puede tener una entrada padre, esto permite construir un sistema de ficheros jerárquico (con carpetas y subcarpetas).

Estas son las columnas de la tabla *entries*:

- id: identificador único de esta entrada.
- parent_id: identificador de la entrada padre. Puede ser nulo en caso de que se trate de un directorio raíz.
- owner_id: identificador del usuario propietario de esta entrada.
- name: nombre de la entrada.
- created_at: fecha y hora en la que se creó la entrada.
- updated_at: fecha y hora de la última modificación de la entrada.

5.3.3 Tabla sources

La tabla *sources* contiene el código ensamblador que almacenan los usuarios. Cuando una entrada (tabla *entries*) representa un fichero de código ensamblador, tiene asociada una fila de la tabla *sources*, que contendrá ese código.

Estas son sus columnas:

- entry_id: identificador de la entrada vinculada a este código.
- content: programa ensamblador.

6. Implementación

6.1 Herramientas

En esta sección se explicará el conjunto de herramientas usadas para desarrollar el proyecto. En el diagrama de la *Figura 10* se vuelve a representar la arquitectura de tres capas de la aplicación junto con las principales herramientas y lenguajes usados para implementarlas.

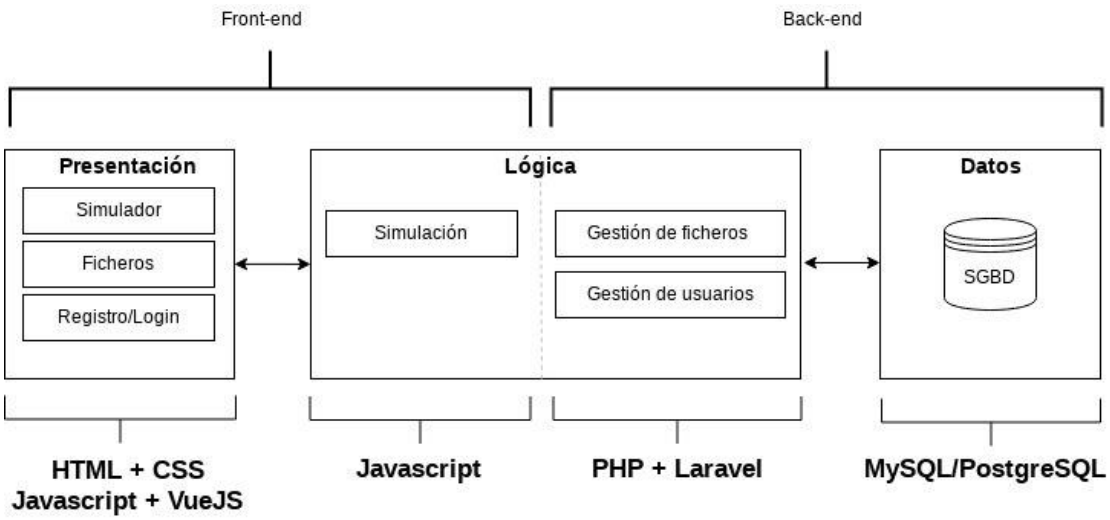

Figura 10: Herramientas principales

6.1.1 Front-end

El front-end está construido con algunas de las tecnologías propias de la web: HTML, CSS y JavaScript. Como el uso que se hace de JavaScript es intensivo (porque el simulador requiere actualizar con mucha frecuencia la interfaz), se ha utilizado un *framework* llamado VueJS (Filipova, 2016) que facilita esta tarea.

VueJS es un *framework* de JavaScript que permite construir aplicaciones web enriquecidas. Entre otras cosas, ofrece una característica llamada *data binding* que permite que la vista se actualice automáticamente con los datos del modelo, lo que evita tener que escribir mucho código.

Además, VueJS permite escribir componentes, que son piezas de código reutilizables compuestas por HTML y CSS para definir el aspecto del componente y código JavaScript para darle comportamiento.

Estas dos características hacen que las aplicaciones escritas con este *framework* sean más fáciles de desarrollar y mantener.

También se ha utilizado una librería de VueJS llamada Vuex[6] y cuya utilidad, esencialmente, reside en permitir mantener el estado global de la aplicación y facilitar la transmisión de datos entre los componentes de Vue.

6.1.2 Back-end

El back-end está hecho con PHP[7] y un *framework* llamado Laravel (Pecoraro, 2015). Laravel provee de un sistema de enrutamiento que permite dirigir las peticiones HTTP del cliente hacia funciones manejadoras que se encargan de dar una respuesta, facilitando que el código esté mejor organizado.

Laravel incluye también un ORM[8] llamado Eloquent que permite manipular la base de datos sin tener que escribir SQL. Eloquent sigue el patrón *active record*, en el que las tablas de la base de datos son representadas mediante clases. Así, para realizar modificaciones en la base de datos se utiliza la programación orientada a objetos, en este caso con PHP, y es el ORM el encargado de transformar esos objetos en registros en la base de datos.

Laravel soporta cuatro sistemas de gestión de bases de datos: MySQL, PostgreSQL, SQLite y SQL Server. SQL Server se descartó por ser software propietario, SQLite por no soportar concurrencia y no poder escalar. Durante el desarrollo se ha usado MySQL por su popularidad y, durante el despliegue, se usará PostgreSQL por motivos de compatibilidad.

6.1.3 Sistema de control de versiones

Para gestionar los cambios hechos en el código de la aplicación, se ha utilizado Git (Chacon & Straub). El repositorio se ha alojado en GitHub para poder sincronizar el proyecto desde varios ordenadores. Además, teniendo el proyecto alojado en GitHub se evitan pérdidas accidentales.

Git también juega un papel importante durante el proceso de despliegue de la aplicación, pues se utiliza para cargar en el servidor de producción todo el código del proyecto.

6.1.4 Servidor

Aunque durante el desarrollo se utiliza un servidor local porque es más cómodo de gestionar, la aplicación se ha desplegado en un servidor para que los usuarios puedan usarlo desde su navegador web sin necesidad de instalar ninguna herramienta de desarrollo.

En concreto, se ha optado por usar en servicio de computación en la nube llamado Heroku que automatiza gran parte del proceso de despliegue. A partir de los ficheros del proyecto, Heroku es capaz de determinar qué software debe estar instalado en el servidor para que este funcione y se encarga de instalarlo. Por ejemplo, tan pronto como se suba

---

[6] https://vuex.vuejs.org/

[7] https://www.php.net/

[8] *Object-relational mapping*

este proyecto, Heroku detectará que necesita un servidor web Apache y PHP y se encargará de instalarlos.

La plataforma también ha sido desplegada con éxito en un servidor privado virtual de la empresa OVH, que disponía de un sistema operativo Debian 9. En este caso, la configuración inicial es más compleja y requiere unos conocimientos básicos para administrar servidores con Linux.

6.2 Estructura de directorios del proyecto

En esta sección se va a explicar la estructura de directorios del proyecto, describiendo, además, algunos de los ficheros más importantes.

Cabe destacar que la estructura general del proyecto viene determinada por Laravel y se puede consultar con más detalle en el apartado correspondiente de la documentación oficial[9].

6.2.1 Carpeta app

La carpeta `app` es la más importante de un proyecto de Laravel porque en ella se encuentran los controladores y modelos.

6.2.1.1 Carpeta Http

6.2.1.1.1 Carpeta Controllers

Los controladores son clases que gestionan las peticiones HTTP que recibe la aplicación. Normalmente, cada tipo de petición HTTP está asociada a una función manejadora que está en una de estas clases.

Lo más habitual es que exista un controlador por cada recurso. Por ejemplo, el `SourceController` contiene una serie de métodos para manipular el recurso "fichero de código fuente" y hay un método por cada una de las operaciones que se puede realizar sobre este tipo de recurso: crear (`store()`), editar (`update()`), listar (`index()`), mostrar (`show()`) y eliminar (`destroy()`).

6.2.1.1.2 Carpeta Requests

Como se ha visto, cuando el cliente realiza una petición HTTP, esta llega a un método manejador de algún controlador. Es importante, como se verá en la sección de *Validación*, comprobar que los datos que se reciben del cliente son válidos.

A menudo, la lógica referente a la validación de la entrada del usuario se incluye en los propios controladores. Sin embargo, cuando esta lógica es muy compleja o es necesario reutilizarla, se implementa en clases externas que se almacenan en la carpeta *Requests*.

6.2.1.2 Carpeta Mail

Laravel incluye herramientas para facilitar el envío de correos electrónicos. En esta carpeta se almacenan los llamados *mailables*, que son clases que representan un tipo de

[9] https://laravel.com/docs/5.8/structure#the-root-directory

correo. Por ejemplo, en el Easy8 se tienen dos tipos: el correo de registro, que permite al usuario validar su correo electrónico; y el correo de restablecimiento de contraseña.

6.2.1.3 Carpeta Policies

Aquí se almacenan las políticas, que son un mecanismo de seguridad que es descrito con más detalle en la sección de *Autorización*.

6.2.1.4 Carpeta Services

En esta carpeta se almacenan los servicios. Un servicio es una clase que ofrece una funcionalidad concreta, evitando que haya mucho código en los controladores.

En este proyecto sólo hay un servicio, `FilesystemService`, que sirve para facilitar la gestión del sistema de ficheros de los usuarios.

6.2.2 Carpeta config

En esta carpeta de almacenan los ficheros de configuración del proyecto. Cada uno de los ficheros se encarga de la configuración de un aspecto concreto. Por ejemplo, en `app.php` tenemos parámetros generales de la aplicación, como su nombre, mientras que en `database.php` se halla la configuración de la base de datos.

6.2.3 Carpeta database

6.2.3.1 Carpeta migrations

Las migraciones son el sistema que tiene Laravel para hacer un seguimiento de la evolución de las tablas y columnas existentes en una base de datos.

Cuando, durante el desarrollo, se desea incluir o modificar alguna tabla, se genera una migración y, programáticamente, se detallan qué modificaciones se van a realizar (por ejemplo: crear una tabla, añadir columnas, definir claves ajenas, etc.).

De esta manera, cuando se sube la aplicación a un entorno de producción se puede reconstruir el estado actual de la base de datos a partir de todas las migraciones existentes.

6.2.3.2 Carpeta seeds

Las seeds son clases que introducen automáticamente datos en la base de datos.

6.2.4 Carpeta resources

En la carpeta `resources` se almacenan ficheros que tienen que ser procesados de alguna manera, como los Single File Components[10] de Vue o las plantillas de Blade[11] de Laravel.

6.2.4.1 Carpeta views

[10] https://vuejs.org/v2/guide/single-file-components.html

[11] https://laravel.com/docs/5.8/blade

En la carpeta *views* se localizan los ficheros de Blade, que es el motor de plantillas de Laravel. Un motor de plantillas permite separar el código PHP (lógica de negocio) del código HTML (presentación).

En este proyecto apenas se ha usado Blade. En su lugar, se ha usado Vue para implementar la parte visual de la página.

6.2.4.2 Carpeta js

6.2.4.2.1 Carpeta components

Aquí se localizan los *Single File Components* de Vue. Por ejemplo, cada pantalla de la aplicación tiene su propio componente: `RegisterComponent`, `LoginComponent`, `SimulatorComponent`, `ExplorerComponent`, etc.

6.2.4.2.2 Carpeta core

En esta carpeta está el núcleo del simulador, del que se habla en profundidad en el apartado *Implementación del simulador*.

6.2.4.2.3 Carpeta store

Aquí se encuentran los ficheros de Vuex[12]. Vuex es una librería para gestionar el estado de la aplicación.

6.2.4.2.4 Fichero app.js

Es el archivo principal de JavaScript, desde el que se realizan tareas de inicialización, importación de dependencias o configuración de rutas.

6.2.5 Carpeta routes

El fichero más relevante en este directorio es `api.php`. Desde aquí se configuran los *endpoints* de la API. Cada uno de ellos está asociado a una función manejadora.

En la *Figura 11* se puede ver, para el recurso `folder`, cómo se asocian una serie de rutas con el correspondiente método manejador de su controlador.

```
Route::get('/folder', 'FolderController@index');
Route::post('/folder', 'FolderController@store');
Route::post('/folder/{id}', 'FolderController@update')->middleware('can:update,entry');
Route::delete('/folder/{entry}', 'FolderController@destroy')->middleware('can:delete,entry');
```

Figura 11: Extracto del fichero api.php

Por ejemplo, cuando el cliente hace una petición HTTP de tipo POST a la ruta `/folder` (segunda línea), se ejecutará la función manejadora `store()` de la clase `FolderController`, que se encarga de crear una nueva carpeta.

[12] https://vuex.vuejs.org/

En el apartado *API del Easy8* se puede consultar la lista completa de *endpoints* que ofrece la API.

6.3 Implementación del simulador

Como se pudo ver en la sección de *Arquitectura de tres capas*, la lógica de la aplicación se debe mantener lo más desacoplada posible de la capa de presentación. Esto ha implicado la división del simulador en dos módulos distintos: por un lado, el núcleo, encargado de ensamblar y ejecutar el código; por otro lado, la interfaz gráfica (editor de texto, botones, vistas de memoria, periféricos, etc).

- El **núcleo**: es un módulo que se encarga únicamente de ensamblar y ejecutar el código sin preocuparse de cómo se representará visualmente el simulador.

  Está escrito en JavaScript puro, es decir, no depende de ninguna librería. El motivo de esto ha sido facilitar su reutilización en otros proyectos. Además, se ha implementado de tal manera que resulte sencillo extender el set de instrucciones soportadas.

- La **interfaz gráfica**: es el conjunto de elementos interactivos que hacen posible que el usuario pueda interactuar fácilmente con el núcleo del simulador, como los botones, las vistas de la memoria, el editor de texto y la representación gráfica de los periféricos.

  La interfaz gráfica del simulador está escrita también en JavaScript, pero con la ayuda del *framework* Vue. Naturalmente, en la arquitectura de tres capas, la interfaz gráfica está en la capa de presentación.

Aunque los programas que se simulan son almacenados en el servidor, tanto el ensamblado como la ejecución del código ocurre únicamente en el lado del cliente. Lo que se pretendía con esto, además de aligerar el trabajo del servidor, era poder generar una versión del proyecto que no requiriese ni siquiera que hubiese servidor.

6.3.1 El núcleo del simulador

Todo el código del núcleo del simulador se puede encontrar en la carpeta `resources/js/core`. Estos son sus ficheros:

- **alu.js**: conjunto de funciones para hacer operaciones con números.
- **assembler.js**: clase que sirve para ensamblar el código del usuario. El ensamblador recibe el código y se encarga de escribir en la memoria el programa ensamblado.
- **instruction-set.js**: set de instrucciones soportadas por el procesador.
- **assembly-rules.js**: set de funciones auxiliares que se usan durante el ensamblado.
- **io.js**: clase que representa los 256 puertos del Easy8.
- **memory.js**: clase representa la memoria del Easy8.
- **registers.js**: clase que representa el conjunto de registros del Easy8.

- **runtime-environment.js**: clase que se encarga de ejecutar el código ensamblado.
- **stack.js**: set de funciones para gestionar la pila del Easy8.

6.3.1.1 El set de instrucciones

El set de instrucciones soportadas por el procesador está contenido en el fichero `resources/js/core/instruction-set.js`. Se trata de un array de objetos en el que cada uno representa una instrucción o pseudoinstrucción.

Este objeto contiene la información suficiente para ensamblar la instrucción y para ejecutarla.

```
[
   /* Más instrucciones */
   {
      mnemonic: 'INC',
      code: 7,
      assembly: function (assembler) {
         assembler.writeByte(this.code);
         return true;
      },
      run: function (memory, registers) {
         registers.incr('RA');
      }
   },
   /* Más instrucciones */
]
```

Figura 12: Ejemplo de instrucción

En el fragmento de código de la *Figura 12: Ejemplo de instrucción* se puede ver la información asociada a la instrucción INC, que incrementa el contenido del registro RA. Estos son los atributos de la instrucción:

- *mnemonic*: es el código nemotécnico de la instrucción.
- *code*: es el código, en formato decimal, de la instrucción.
- *assembly*: esta función debe ensamblar la instrucción. El ensamblador la ejecuta cuando encuentra dicha instrucción en el código. Típicamente la función *assembly* escribe en la memoria el código de la instrucción y, si se trata de una función codificada en dos bytes, tendría que escribir también el parámetro.
- *run*: el simulador llama a esta función cuando toca ejecutar esta instrucción. Recibe por parámetros una instancia de la memoria, los registros y los puertos, así, desde esta función se puede manipular el estado del computador.

En el caso concreto de la instrucción INC, lo único que debe hacer es incrementar el registro RA en una unidad.

Todas las instrucciones deben tener los atributos y funciones mencionados arriba. No obstante, no siempre es necesario que hagan algo. Por ejemplo, la función `run()` no tiene sentido que haga nada para una pseudoinstrucción.

6.3.1.2 Etapa de ensamblado

La primera etapa de la simulación consiste en el ensamblado del código que ha escrito el usuario.

El ensamblador (implementado en el fichero `resources/js/core/assembler.js`) recibe el código y lo parte en líneas. A continuación, de cada línea se extraen los tokens que la forman. Por ejemplo, la instrucción `MOVE RA, 0x02` tendría tres tokens: `MOVE`, `RA` y `0x02`.

Una vez se tienen los tokens de una línea, se consulta en el set de instrucciones la instrucción asociada al mnemónico que se ha obtenido durante la *tokenización* y se ejecuta su función `assembly()`, descrita en el apartado anterior, que se encarga de insertar en la memoria del Easy8 la instrucción ensamblada.

6.3.1.2.1 Resolución de etiquetas

Una etiqueta es un alias de una dirección de memoria. El uso de etiquetas permite al programador hacer referencia a una posición de memoria con un nombre en vez de con un valor numérico. Las etiquetas hacen que el código sea más sencillo de comprender, mantener y depurar.

Por ejemplo, el fragmento de la *Figura 13* incrementa el registro RA continuamente con un bucle:

```
0x00 ADDI RA, 0x01
0x02 JUMP 0x00
```

Figura 13: Resolución de etiquetas. Ejemplo 1.

La primera instrucción incrementa el registro RA en una unidad y la siguiente manda la ejecución al inicio (posición `0x00`), haciendo que se vuelva a ejecutar el ADDI.

Supóngase que el programador desea modificar el código para inicializar el registro RA con algún valor (p. ej.: 5) tendría que hacer algo como en la *Figura 14*:

```
0x00 MOVEI RA, 0x05
0x02 ADDI RA, 0x01
0x04 JUMP 0x02
```

Figura 14: Resolución de etiquetas. Ejemplo 2.

Nótese que, además de introducir una instrucción al principio del programa para iniciar

el registro RA a 5, se ha tenido que actualizar el parámetro de la instrucción JUMP con la nueva dirección de la instrucción ADDI, puesto que se quiere que el salto sea hacia el ADDI (dirección `0x02`) y no hacia el MOVEI (dirección `0x00`).

Este estilo de programación puede introducir muchos errores, bien porque el programador se equivoca al escribir una dirección o bien porque olvida que tiene que actualizar una. El problema empeora con programas más largos.

La finalidad de añadir soporte para etiquetas es que el usuario pueda escribir el programa como se observa en la *Figura 15*:

```
0x00 ADDI RA, 0x05
0x02 @loop: ADDI RA, 0x01
0x04 JUMP @loop
```

Figura 15: Resolución de etiquetas. Ejemplo 3.

Como se ve en la *Figura 15*, con esta nueva notación, el usuario puede crear una etiqueta con un nombre que le sea fácil de recordar (segunda línea) para hacer referencia a ella desde otros lugares (tercera línea). Así, si el usuario introdujera nuevas instrucciones al inicio del programa, no tendría que hacer ninguna modificación al bucle para que tuviera el mismo comportamiento.

La gestión de las etiquetas es un proceso que se realiza durante el ensamblado, no durante la ejecución. Cuando el ensamblador se encuentra una nueva etiqueta, anota en una tabla a qué dirección hace referencia. En el código de la *Figura 15*, por ejemplo, el ensamblador almacenaría que la etiqueta `loop` hace referencia a la dirección `0x02`.

El proceso mediante el cual se reemplazan las etiquetas por las direcciones reales (`JUMP @loop` → `JUMP 0x02`) es un poco más sensible porque en el momento de encontrar una referencia a una etiqueta todavía podría no conocerse a qué dirección está asociada. Este problema se puede ver en el código de la *Figura 16*:

```
ADDI RA, 0x05
# más código
JUMP @salir # Se usa una etiqueta que todavía se desconoce.
# más código
@salir: STOP
```

Figura 16: Resolución de etiquetas. Ejemplo 4.

Al momento de ensamblar la instrucción `JUMP @salir`, el ensamblador todavía no conoce a qué dirección hace referencia la etiqueta “salir”. Es por esto por lo que el ensamblador del Easy8 realiza dos pasadas para ensamblar un programa:

- En la **primera pasada** ensambla las instrucciones y, si alguna hace referencia a una etiqueta, almacena un cero de manera temporal. Además, durante esta pasada asocia las etiquetas a las direcciones de memoria. De modo que, al

terminar la primera pasada el ensamblador ya conoce a qué dirección apunta cada etiqueta.

- En la **segunda pasada** es cuando se reemplazan todos esos ceros por el valor que le corresponde.

6.3.1.3 Etapa de ejecución

La ejecución del código ocurre en la clase `RuntimeEnvironment` (fichero `resources/js/core/runtime-environment.js`). Esta clase aúna todos los elementos del computador Easy8: los registros, la memoria y los puertos.

El simulador comienza leyendo la memoria, que, en este punto, ya debería tener el programa correctamente ensamblado. El registro PC es el que marca qué posición de la memoria se lee en un momento dado.

Cuando se lee el código de instrucción, se revisa, en el set de instrucciones, a qué instrucción le corresponde este y se ejecuta el método `run()` que se describió en el apartado *El set de instrucciones*.

Una posibilidad que se llegó a considerar (y a poner a prueba) para ejecutar las instrucciones, era ejecutar una después de otra hasta que el programa terminara. Esto era un problema porque si la ejecución del programa tomaba mucho tiempo, el simulador bloqueaba el navegador y, por tanto, el usuario dejaba de poder interactuar con cualquier elemento de la aplicación. La única manera que tenía de terminar con la ejecución de su programa era cerrar la pestaña.

Una posible alternativa habría sido ejecutar la simulación en un hilo independiente del de la interfaz, así, el hilo principal habría podido seguir gestionando los eventos de la interfaz mientras un hilo secundario se encargaba de la simulación. Sin embargo, esto no es posible porque JavaScript es un lenguaje monohilo.

Un tercer enfoque para resolver este problema pasaba por usar una función de JavaScript llamada `setInterval()` que permite ejecutar otra función en intervalos de tiempo regulares. Lo que se hace, es solicitar a `setInterval()` que ejecute una instrucción cada 50 milisegundos. Así, se consigue que, durante el resto del tiempo, el hilo de JavaScript esté libre y el usuario pueda interactuar con la web, creando una falsa sensación de paralelismo.

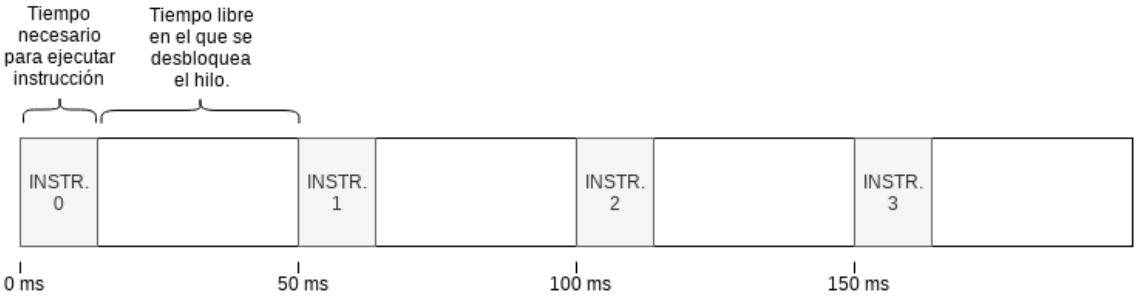

Figura 17: Diagrama temporal de ejecución de instrucciones

En el diagrama de la *Figura 17* se puede ver cómo sucede la ejecución de cuatro instrucciones. Como las instrucciones tardan muy poco tiempo en ejecutarse y sólo se

ejecuta una cada 50ms, tras cada ejecución hay un instante de tiempo en el que el hilo de JavaScript se queda ocioso, dando la oportunidad al hilo para gestionar otras tareas (por ejemplo, los eventos del usuario).

Esta última técnica es la que se ha implantado con éxito en el simulador.

6.3.2 La interfaz gráfica del simulador

Como ya se ha comentado, la aplicación debe ofrecer al usuario una representación gráfica del emulador del Easy8 para que pueda interactuar con él como si se tratara de un dispositivo físico real.

Para implementar esta parte, se dibujó el emulador usando un programa de dibujo vectorial que es capaz de exportar al formato SVG. El resultado se ve en la *Figura 18*.

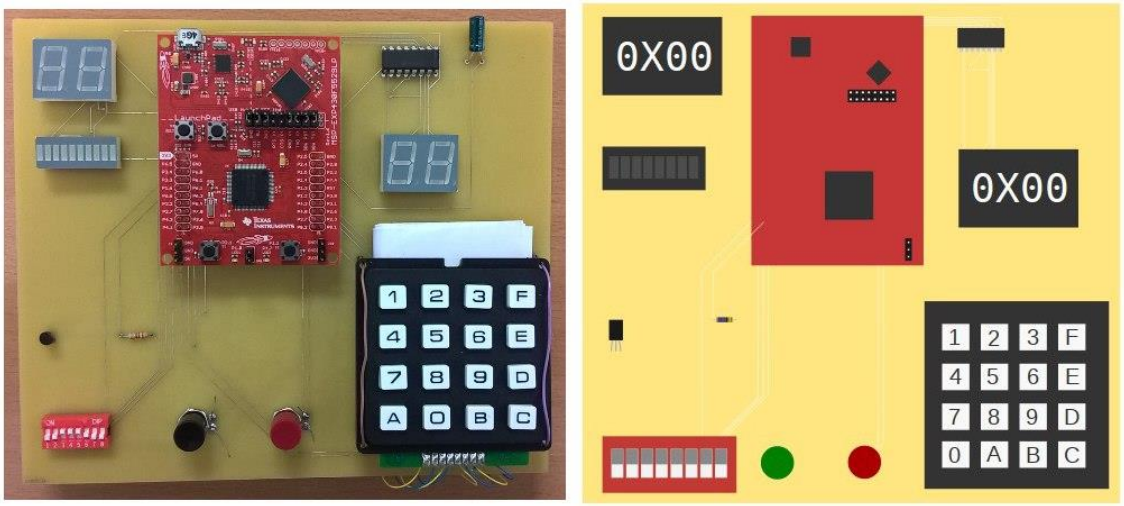

Figura 18: Emulador y representación de este

La ventaja del formato SVG es que se basa en XML y actualmente los navegadores web son capaces de interpretar ese XML si se introduce en la etiqueta `<svg>` que incorpora HTML5. Esto permite tratar cualquier elemento del dibujo como si se tratara de un elemento más de la página web y, por tanto, se le pueden añadir *listeners* para poder ejecutar código cuando el usuario realiza alguna acción sobre alguno de los elementos del dibujo.

El código de la interfaz gráfica del simulador, incluido el SVG, se encuentra en un componente de Vue: `resources/js/components/SimulatorComponent.vue`, que se puede ver en la *Figura 19*.

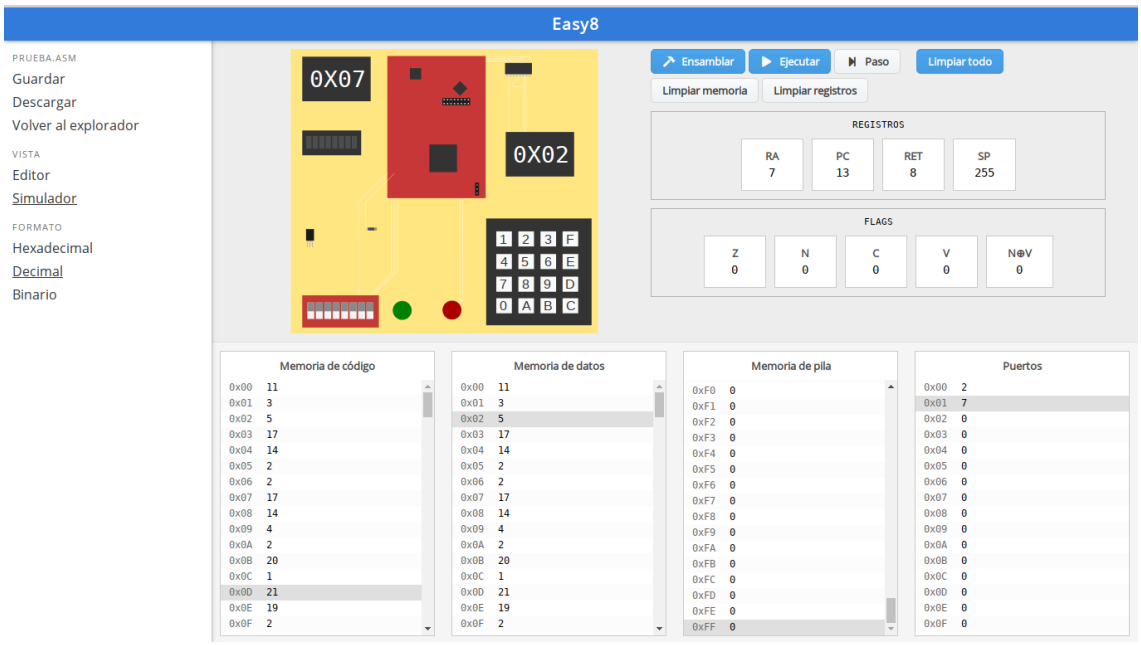

Figura 19: Representación del emulador integrado dentro de la plataforma

Es en este componente donde se crea la instancia del ensamblador y del entorno de ejecución (clases `Assembler` y `RuntimeEnvironment`) descritos en el apartado anterior. El núcleo del simulador trabaja con gran autonomía, tan sólo hay que indicarle el código a ensamblar y ejecutar. El núcleo desconoce completamente la representación gráfica del emulador, lo único que conoce es el contenido de los puertos. Es desde `SimulatorComponent.vue` desde donde se modifica el contenido de esos puertos en función de las acciones que el usuario ha realizado.

Por ejemplo, en `SimulatorComponent.vue` hay un método llamado `redButtonPressed()` que es ejecutado cuando el usuario ha hecho clic en el botón rojo. El núcleo del simulador no detecta directamente el clic del usuario porque el núcleo es completamente ajeno a la interfaz y sus eventos. Lo que ocurre desde el método `redButtonPressed()` es que se escribe un 1 en el puerto asociado al botón rojo, como se puede observar en la *Figura 20*.

```
redButtonPressed() {
    this.runtimeEnvironment.getIo()
            .writePort(IODevices.K_BUTTON, 1);
}
```

Figura 20: Implementación de redButtonPressed()

Para los periféricos de salida ocurre algo similar: cuando el entorno de ejecución realiza algún cambio sobre un periférico de este tipo, no trata de realizar ninguna modificación sobre la interfaz, sino que es esta la que nota que ha habido un cambio sobre un puerto y reacciona de la manera apropiada.

Esta separación entre el núcleo del simulador y los periféricos pretende ser una representación fiel de la realidad, donde el procesador sólo conoce los puertos de entrada y salida y no los detalles de los periféricos conectados a ellos. Además, esto permite

fácilmente añadir, eliminar o modificar los periféricos simulados sin necesidad de modificar el núcleo de la simulación.

6.4 Implementación del back-end

6.4.1 API del Easy8

Como se explicó en el capítulo de *Diseño*, el back-end necesita algún mecanismo de comunicación con el front-end. En este proyecto se decidió que el back-end implementara una API tratando se seguir la arquitectura REST.

Las API RESTful se basan en recursos, así que el primer paso para implementar una pasa por definir qué recursos va a tener que ofrecer el back-end del Easy8:

- Usuarios (recurso *user*).
- Carpetas (recurso *folder*).
- Ficheros de código ensamblador (recurso *source*).
- Recuperación de contraseña (recurso *password-reset*). Este recurso representa el proceso de recuperación de contraseña.

El siguiente paso es establecer qué tipo de acciones se van a poder ejecutar sobre cada uno de los recursos:

- Usuarios: crear, obtener.
- Recuperación de contraseña: crear, actualizar.
- Carpetas: listar, crear, modificar y eliminar.
- Ficheros de código ensamblador: listar, obtener, crear, modificar y eliminar.

A partir de los recursos y las acciones, se obtienen los *endpoints* de la API, que aparecen en la *Figura 21*.

Verbo HTTP	**Endpoint**	**Descripción**
POST	/login	Obtiene un usuario a partir de su email y contraseña.
GET	/user/{user_id}	Obtiene un usuario.
POST	/user	Crea un usuario nuevo.
POST	/user/{user_id}/confirm	Confirma el correo electrónico de un usuario.
POST	/password-reset	Solicita la recuperación de la contraseña.
POST	/password-reset/{token}	Ejecuta la recuperación de contraseña a partir de un token que el usuario recibe por correo electrónico.
GET	/folder	Obtiene la lista de carpetas.
POST	/folder	Crea una nueva carpeta.
POST	/folder/{folder_id}	Edita los atributos de una carpeta existente.
DELETE	/folder/{folder_id}	Elimina una carpeta y todo su contenido.
GET	/source	Obtiene la lista de s.
GET	/source/{source_id}	Obtiene un fichero concreto.
POST	/source	Crea un fichero de código nuevo.

POST	/source/{source_id}	Edita los atributos o contenido de un fichero de código existente.
DELETE	/source/{source_id}	Elimina un fichero de código.

Figura 21: Endpoints de la API

La API del Easy8 es una simplificación de la arquitectura REST, pues no cumple con todas sus restricciones:

- Por comodidad, se ha añadido el *endpoint* `/login`, que representa una acción, no un recurso.
- Otro motivo es que no se hace ningún uso de hipermedia como motor de estado de la aplicación.

6.4.2 Seguridad

En el siguiente apartado se detallarán los distintos mecanismos de seguridad que se implementan en el back-end.

6.4.2.1 Autenticación

Casi todas las peticiones que se realizan desde el front-end hacia el back-end requieren que el usuario esté autenticado para comprobar que, por ejemplo, no está tratando de modificar un fichero de código que no es suyo.

Como se mencionó en el apartado *Tabla users,* cada usuario del sistema tiene asignado un API token, que es una cadena de caracteres aleatoria, única y secreta que le va a identificar de manera inequívoca.

El front-end, en cada petición que realiza, adjunta ese API token para que llegue al servidor. El servidor lo revisará y podrá identificar a qué usuario pertenece ese token y, por tanto, qué usuario está realizando la petición.

Una alternativa para autenticar al usuario podría haber consistido en reemplazar ese API token por el par *(email, contraseña)*. El email serviría para identificar al usuario y la contraseña se utilizaría para verificar que es realmente el dueño de la cuenta.

Naturalmente, adjuntar el correo y la contraseña en cada petición es una opción menos segura que mandar el API token. Además, el API token puede ser renovado periódicamente sin que el usuario tenga que intervenir. Eso no ocurre con el correo electrónico y la contraseña.

6.4.2.2 Autorización

Gracias a la autenticación explicada en el apartado anterior, podemos conocer qué usuario está realizando una petición. Muy a menudo habrá que comprobar si ese usuario tiene acceso al recurso que quiere manipular.

Por ejemplo, podríamos estar recibiendo una petición de un usuario para crear un fichero en una carpeta determinada. Es importante que el back-end verifique que el usuario que nos manda la petición es el dueño de la carpeta contenedora y, si no lo es, impedir la acción y devolver un mensaje de error.

Para realizar esto, Laravel incluye un mecanismo llamado “políticas” que permite organizar la lógica referente a la autorización.

Esencialmente, una política es una clase que está asociada a un modelo y define unos métodos para cada una de las acciones que se pueden realizar sobre este, como se aprecia en el esquema de la *Figura 22*. Estos métodos son los que realizan la comprobación que permite o bloquea la acción que el usuario intenta ejecutar sobre el modelo.

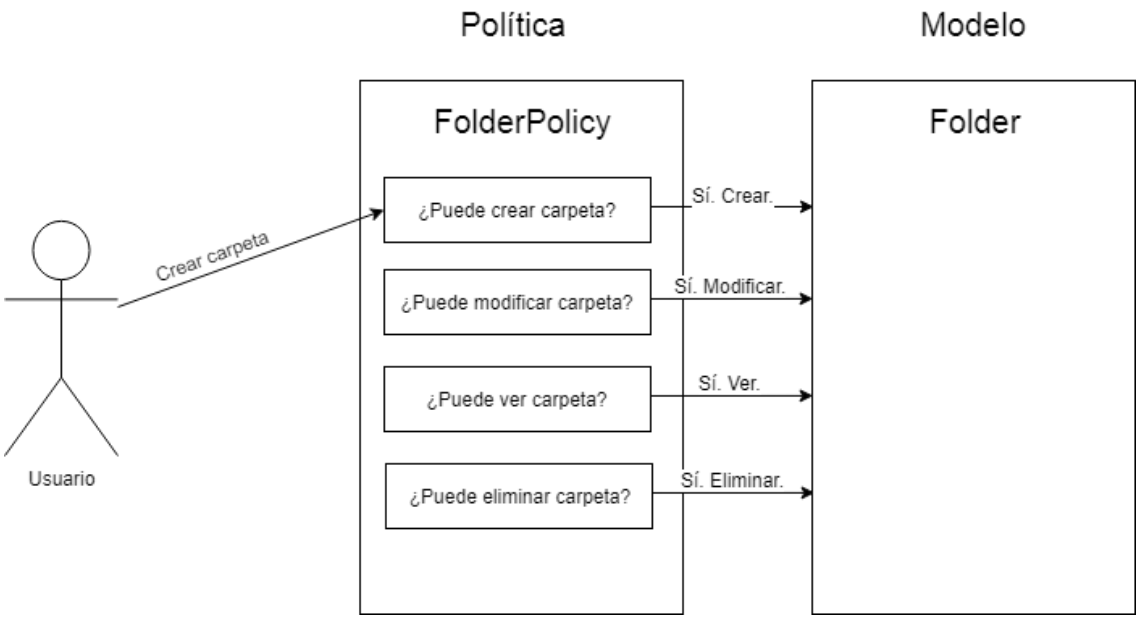

Figura 22: Esquema de autorización a través de políticas

6.4.2.3 Validación

Además de autenticar al usuario y comprobar que los recursos a los que accede son realmente suyos, es muy importante comprobar que los datos que manda son válidos.

Para ello, Laravel incluye un sistema que permite, de una manera muy breve, establecer qué reglas debe cumplir cada uno de los datos que se reciben del usuario.

```
$request->validate([
            'name' => 'required|max:255',
            'surname' => 'required|max: 255',
            'email' => 'required|email|unique:users',
            'password' => 'required|confirmed|min:5'
]);
```

Figura 23: Ejemplo de validación

En el fragmento de código de la *Figura 23* se muestra, para cada uno de los posibles parámetros que puede mandar el usuario cuando se va a registrar, una lista de reglas que se deben cumplir. Por ejemplo, se indica que el campo email es obligatorio (`required`), debe tener el formato de un correo válido (`email`) y no debe estar ya registrado en el sistema (`unique:users`).

7. Despliegue

En este capítulo se va a describir el despliegue de la aplicación en un servicio de computación en la nube llamado Heroku[13], que automatiza parte del proceso y facilita la configuración del proyecto evitando, en buena medida, que haya que tener conocimientos de administración de servidores.

Para poder realizar este proceso, es necesario tener una cuenta en la plataforma mencionada. Asimismo, se asume que se tiene el repositorio de Git del proyecto[14] y que se ha instalado la herramienta Heroku CLI[15].

7.1 Crear la aplicación de Heroku

Habiendo iniciado sesión en Heroku, hay que hacer clic en *Create new app* (ver *Figura 24*). A continuación, habrá que darle un nombre cualquiera a la aplicación y seleccionar la región geográfica donde se situará el servidor.

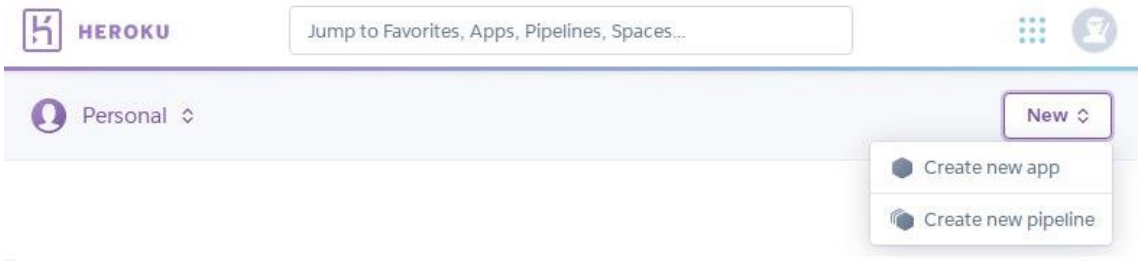

Figura 24: Creación de aplicación en Heroku

7.2 Configurar el repositorio de Git

A continuación, usando la consola, habrá que situarse en el repositorio Git del proyecto y ejecutar el comando `heroku git:remote -a **easy8**`, que vinculará el repositorio local con repositorio remoto de Heroku. El texto en negrita tendrá que ser reemplazado por el nombre que se le haya dado a la aplicación en el paso anterior.

7.3 Cargar el proyecto

[13] https://heroku.com

[14] https://github.com/rafaelgc/easy8

[15] https://devcenter.heroku.com/articles/heroku-cli#download-and-install

Ejecutando el comando `git push heroku master`, se cargará el proyecto en el servidor de Heroku. Heroku se encargará de instalar las dependencias necesarias para que el proyecto funcione, como se puede ver en la *Figura 25*

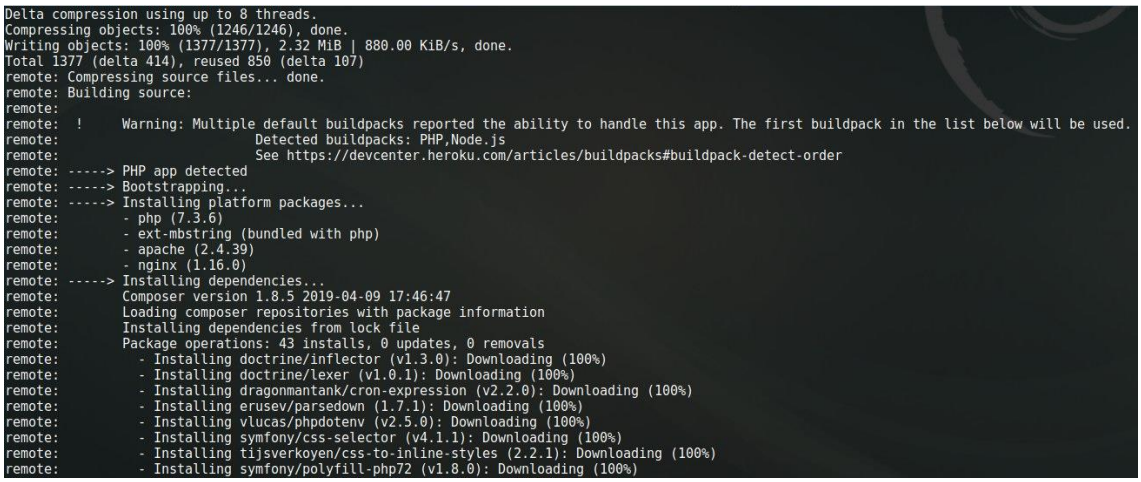

Figura 25: Carga del proyecto en Heroku usando Git.

7.4 Instalar el complemento de PostgreSQL

El servidor de Heroku no dispone de ningún sistema de gestión de bases de datos instalado. Si se necesita uno, como es el caso, es necesario añadir un complemento a la aplicación de Heroku.

Desde la web de Heroku, en la página de la aplicación, habrá que ir a la pestaña *Resources* y buscar e instalar un complemento llamado Heroku Postgres (ver *Figura 26*).

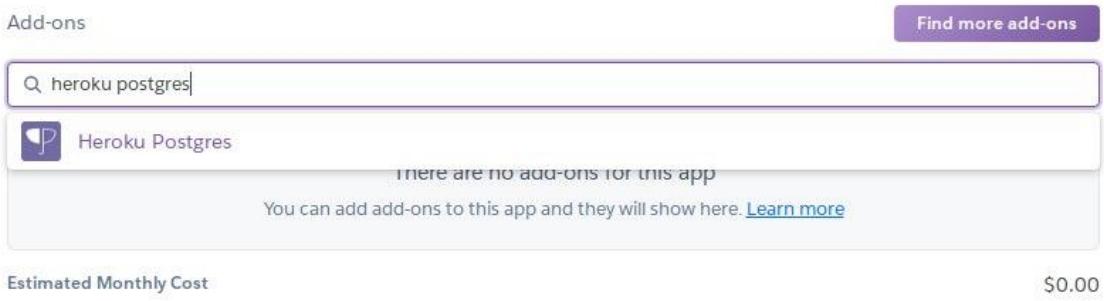

Figura 26: Añadir el complemento Heroku Postgres a la aplicación.

7.5 Generar clave de cifrado

Laravel necesita una clave, que puede establecer el usuario, para cifrar las *cookies*. Para generar esta clave, habrá que conectarse al servidor de Heroku.

Estando en la carpeta del proyecto, hay que ejecutar el comando `heroku run bash`. Una vez se haya establecido la conexión, se debe ejecutar `php artisan key:generate -`

`show`, que devolverá una cadena de caracteres que habrá que guardar para usarla en el siguiente paso (ver *Figura 27*).

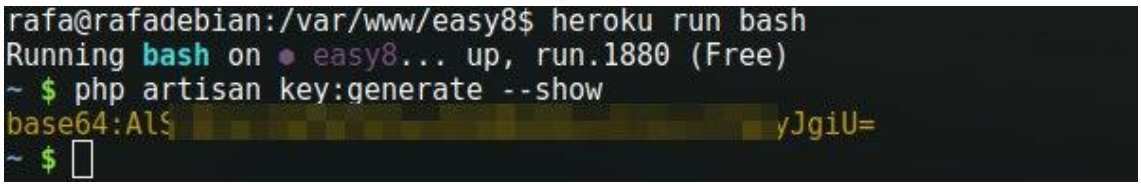

Figura 27: Generar clave de cifrado

7.6 Configurar el proyecto

Laravel permite establecer configuraciones a través de una serie de pares clave-valor. Por ejemplo, con la clave `DB_HOST` se puede indicar en qué máquina se encuentra la base de datos.

Esta serie de pares clave-valor se puede establecer desde varios lugares. En primer lugar, Laravel buscará el par entre las variables del entorno del sistema operativo. Si no las encontrara en ese lugar, buscaría en el fichero de configuración `.env`, que se localiza en la raíz del proyecto. Si no se encontrara en ninguno de esos dos lugares, se recurriría a los ficheros PHP de configuración del proyecto, localizados en la carpeta `config`.

Los dos primeros métodos son los más cómodos de utilizar. Como Heroku permite establecer las variables del sistema desde la web, se utilizará este método.

Estando en la página de la aplicación de Heroku, hay que acceder a la pestaña *Settings*. Allí se hará clic sobre *Reveal Config Vars*.

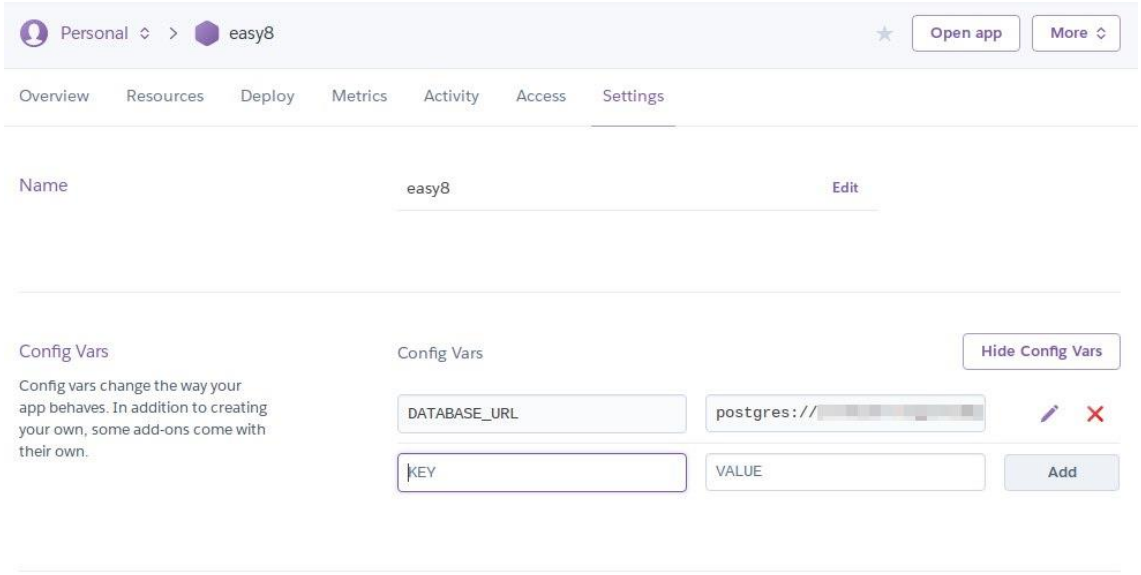

Figura 28: Configuración de variables

Al haber instalado el complemento Heroku Postgres, se habrá introducido automáticamente una clave llamada `DATABASE_URL`, como se ve en la *Figura 28*, que

servirá a la plataforma Easy8 para conocer los detalles de conexión a la base de datos. Será necesario introducir una clave llamada `DB_CONNECTION` cuyo valor sea `pgsql`.

A continuación, se configurarán otros parámetros generales de la aplicación, como su nombre o URL. En la *Figura 29* aparecen los parámetros que hay que añadir.

Clave	Valor
`APP_NAME`	`Easy8`
`APP_ENV`	`production`
`APP_KEY`	La clave generada en *Generar clave de cifrado*.
`APP_DEBUG`	En producción se pone normalmente a `false`, aunque se puede poner a `true` para obtener mensajes de depuración más descriptivos.
`APP_URL`	La URL de la aplicación. La puedes ver en la sección *Domains* de la pestaña *Settings*.

Figura 29: Configuración general

Lo único que queda por configurar es el sistema de envío de correos electrónicos. El envío de correos sirve para que Easy8 envíe un correo a los usuarios para validar su cuenta.

Es posible deshabilitar esta característica estableciendo la clave `MAIL_ENABLED` a `false`. Si es así, todas las cuentas se considerarán validadas tan pronto como sean registradas y los usuarios no recibirán correo electrónico alguno.

Si se desea habilitar la validación, será necesario poner `MAIL_ENABLED` a `true`. En ese caso, habrá que configurar algún servicio de correo electrónico. En este caso, se ha usado Mailgun [16] por su sencillez.

Mailgun requiere que el usuario disponga de algún dominio desde el que se enviarán los correos. Una vez se tenga configurado uno, Mailgun da una serie de claves que tendremos que añadir a la configuración de Heroku.

En la *Figura 30* aparecen los atributos que hay que añadir:

Clave	Valor
`MAIL_ENABLED`	`true` o `false`. Si es `false`, no es necesario indicar los atributos de abajo.
`MAIL_DRIVER`	`mailgun`
`MAILGUN_DOMAIN`	Dominio que se ha establecido en la página de Mailgun para enviar los correos.
`MAILGUN_SECRET`	Clave secreta.
`MAILGUN_ENDPOINT`	URL base de la API de Mailgun.

[16] https://mailgun.com

`MAIL_FROM_ADDRESS`	Dirección de correo desde la que se envían los emails. Por ejemplo, `contact@tudominio.xyz`.

Figura 30: Configuración del sistema de envío de correos electrónicos

La *Figura 31* es una captura de la página de Mailgun con los todos los datos necesarios marcados.

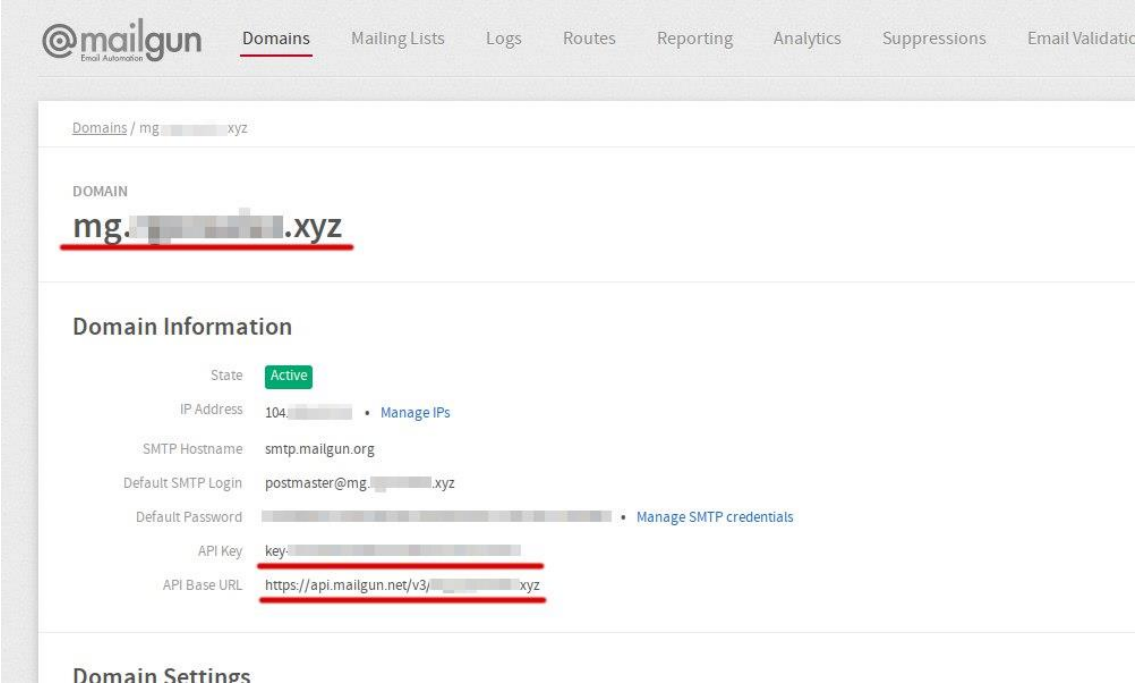

Figura 31: Datos de Mailgun

7.7 Migración de la base de datos

Para insertar en la base de datos las tablas necesarias para que funcione la aplicación habrá que poner en marcha el sistema de migraciones de Laravel, que se encargará de reconstruir las tablas a partir de los ficheros de migración.

Para iniciar la migración, es necesario conectarse de nuevo al servidor de Heroku ejecutando el comando `heroku run bash`. Estando ya dentro del servidor, ejecutaremos `php artisan migrate`. El resultado de la migración será parecido al que se ve en la *Figura 32*.

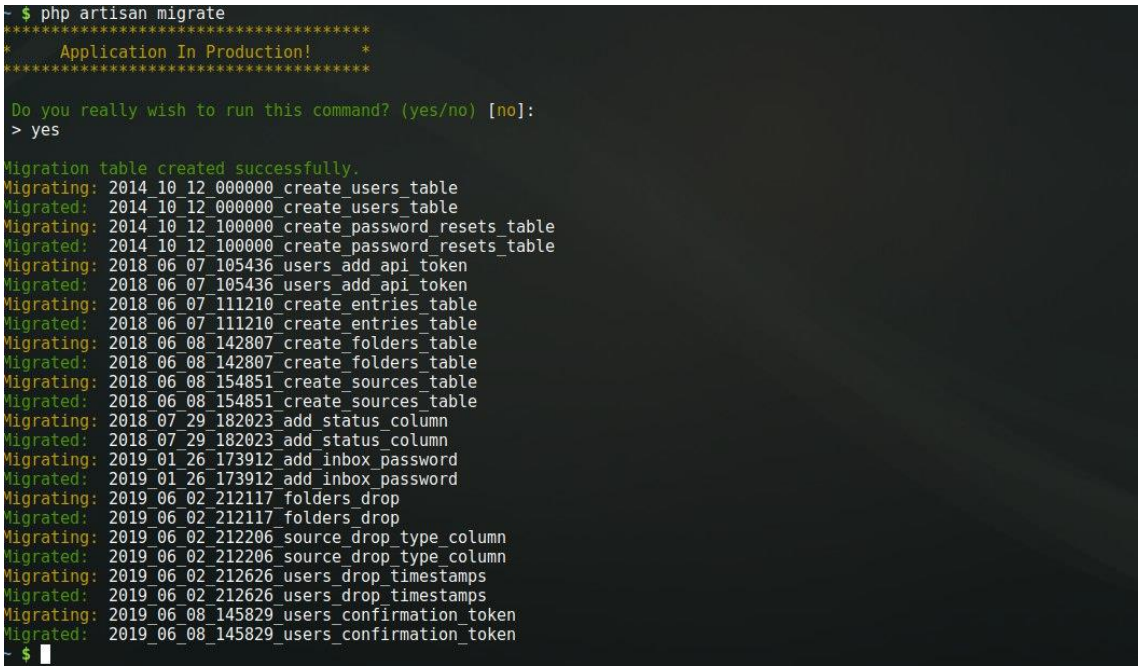

Figura 32: Migración de la base de datos

En este punto, la aplicación ya debería estar funcionado. Para ponerla a prueba, se puede acceder a ella a través de la URL que proporciona Heroku (ver *Figura 33*).

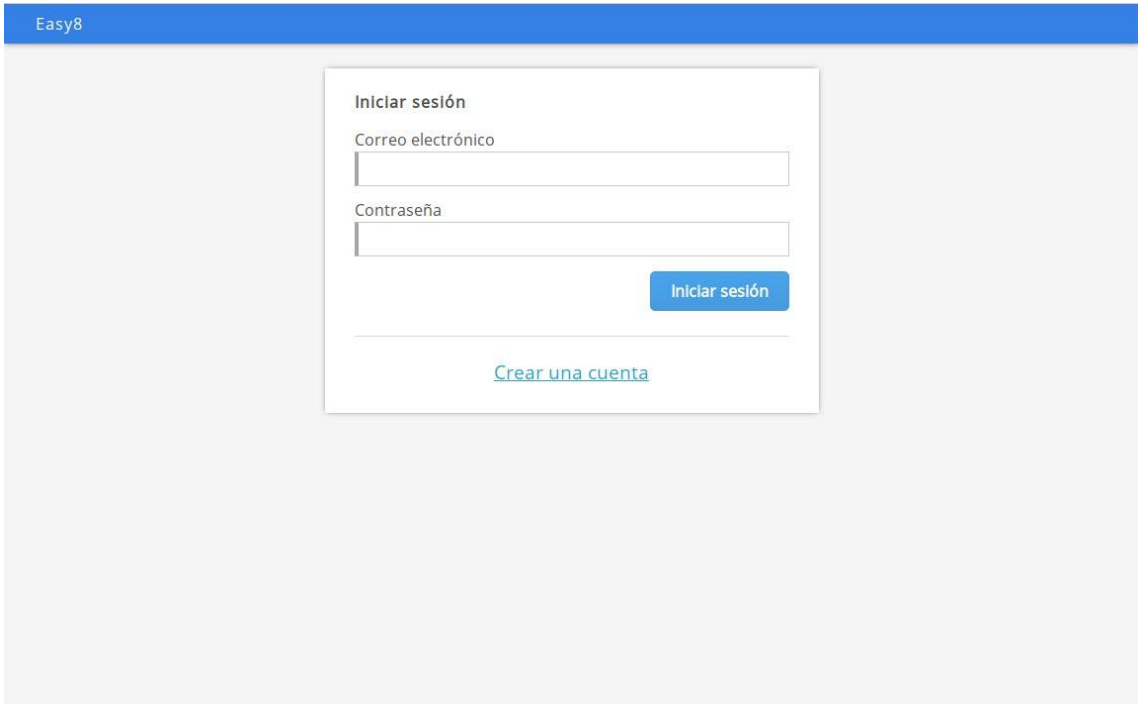

Figura 33: Aplicación en marcha

8. Pruebas

En este capítulo se realizarán distintas pruebas para verificar el correcto funcionamiento de la plataforma.

8.1 Pruebas de uso

A continuación, se va a hacer un recorrido completo por toda la aplicación, desde el registro del usuario hasta la simulación de un programa, para confirmar que el comportamiento general es correcto.

8.1.1 Registro

Para registrarse, el usuario debe introducir algunos datos personales (ver *Figura 34*).

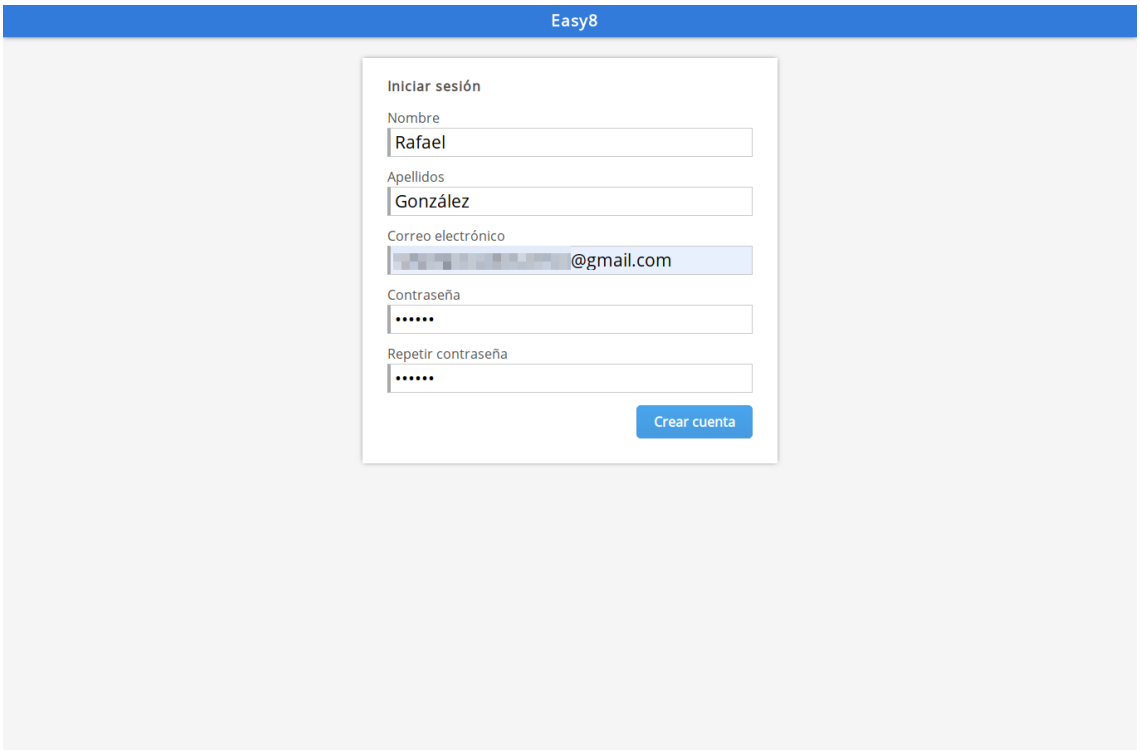

Figura 34: Pantalla de registro

Cuando hace clic en Crear cuenta, se recibirá un correo electrónico con un enlace para confirmar el registro (ver *Figura 35*).

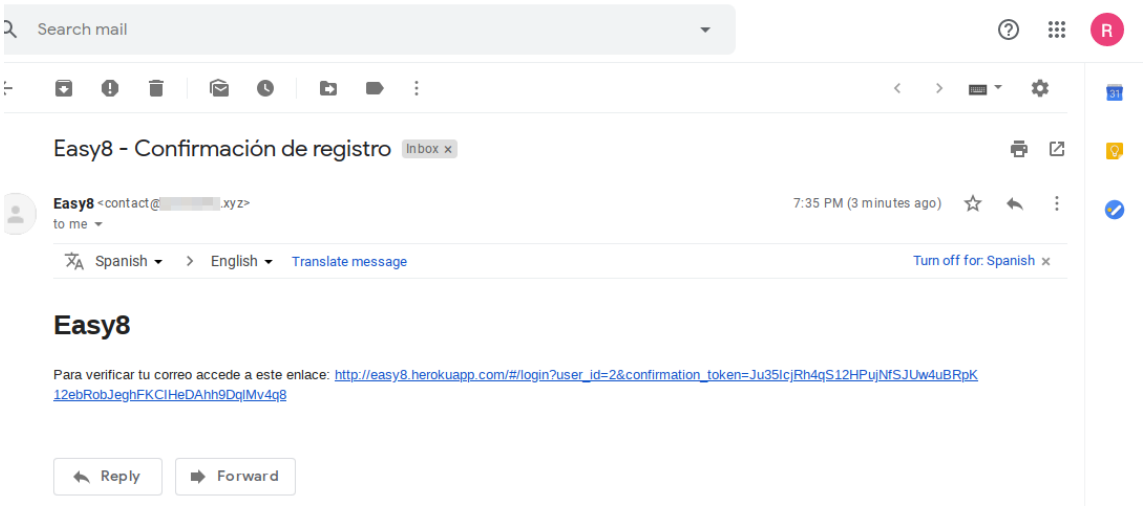

Figura 35: Correo electrónico de confirmación

Una vez haya visitado el enlace, el usuario podrá iniciar sesión con normalidad.

8.1.2 Iniciar sesión

El usuario inicia sesión introduciendo su correo electrónico y contraseña (ver *Figura 36*). Si las credenciales son correctas, el usuario es mandado al explorador de ficheros.

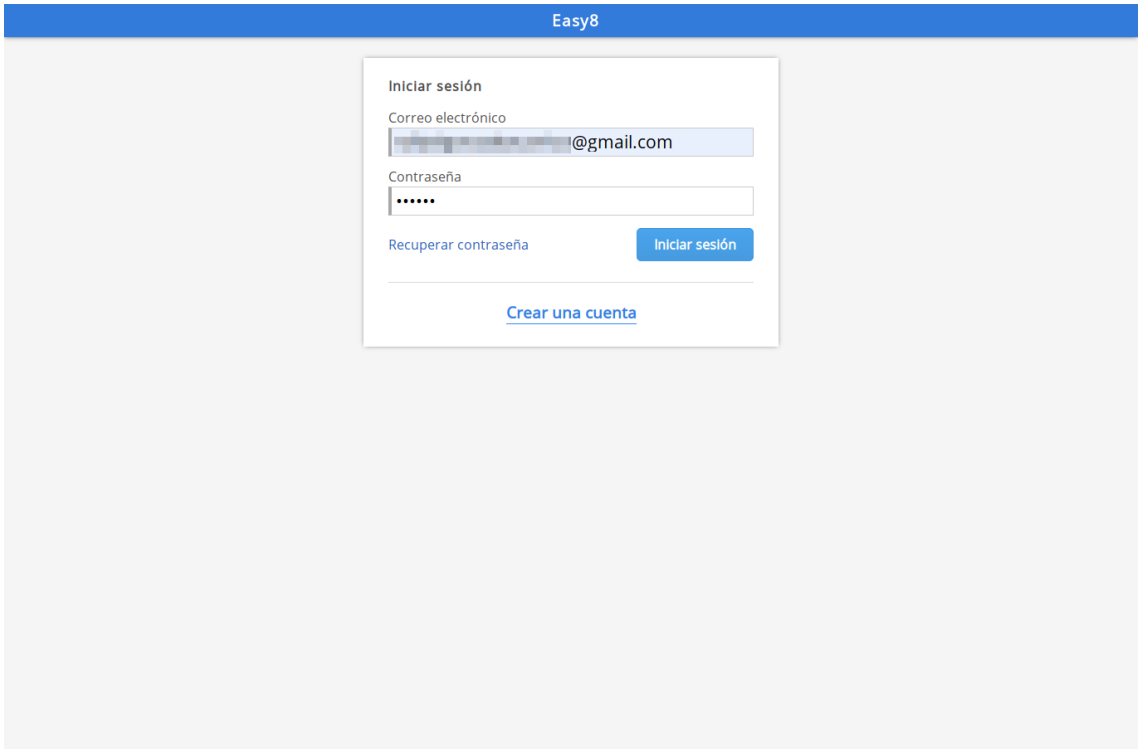

Figura 36: Inicio de sesión

8.1.3 Explorador de ficheros

En el explorador de ficheros (ver *Figura 37*), que inicialmente estará vacío, el usuario podrá crear carpetas y ficheros de código, como se puede ver en la *Figura 38*.

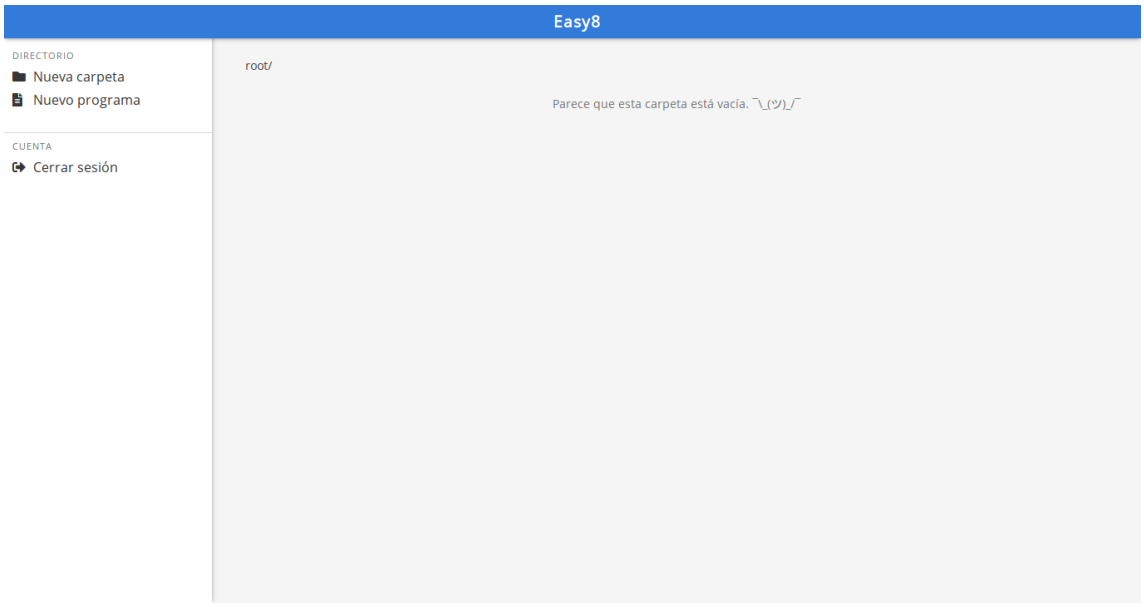

Figura 37: Explorador de ficheros vacío

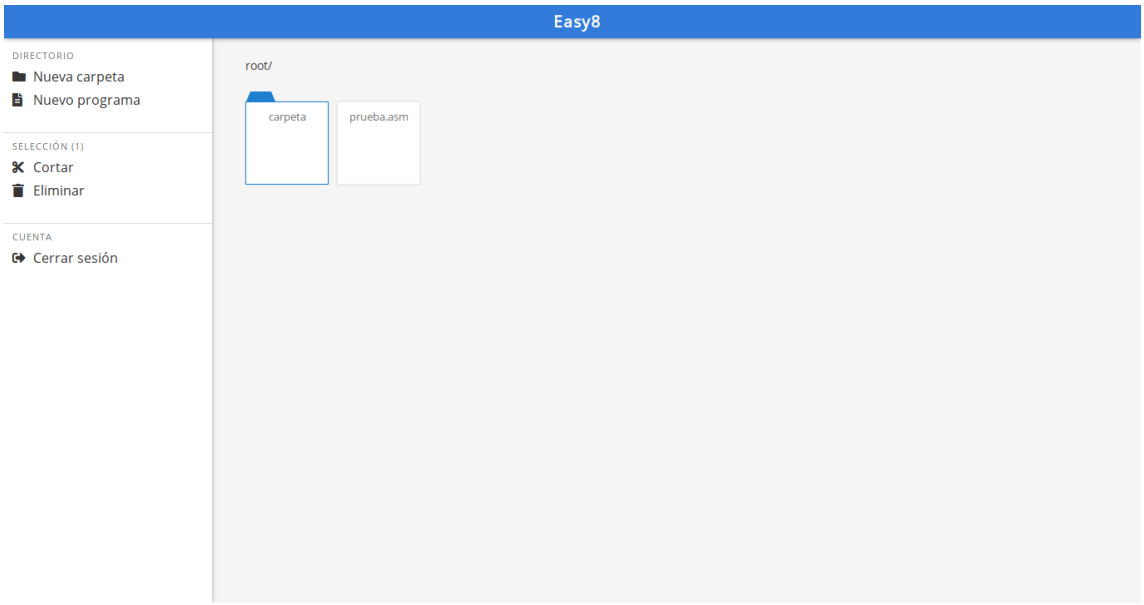

Figura 38: Explorador de ficheros con una carpeta y un fichero de código

8.1.4 Simulación

Haciendo doble clic, en el explorador de ficheros, sobre el fichero de código, se abrirá el editor y simulador.

Para probar el ensamblaje y la ejecución, se ha escrito un programa que lee dos números por teclado y muestra en el *display* el resultado de su suma (ver *Figura 39*).

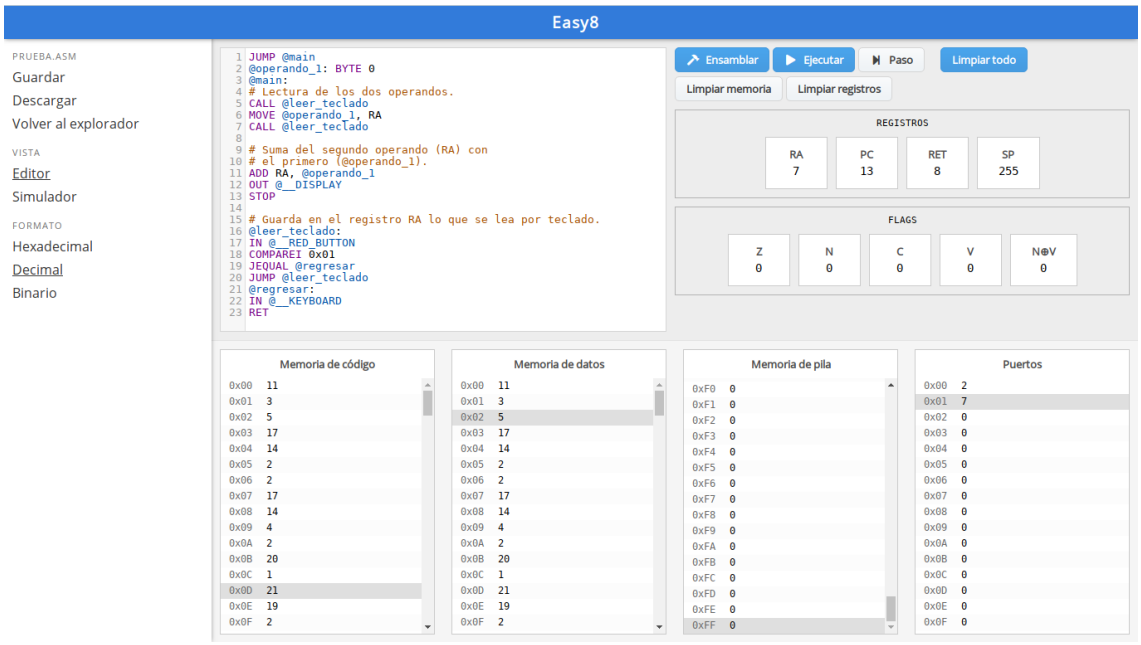

Figura 39: Edición de un programa

Tras la ejecución, en la que se ha introducido un cinco y un dos, el resultado que muestra el display (arriba a la izquierda) es siete (ver *Figura 40*).

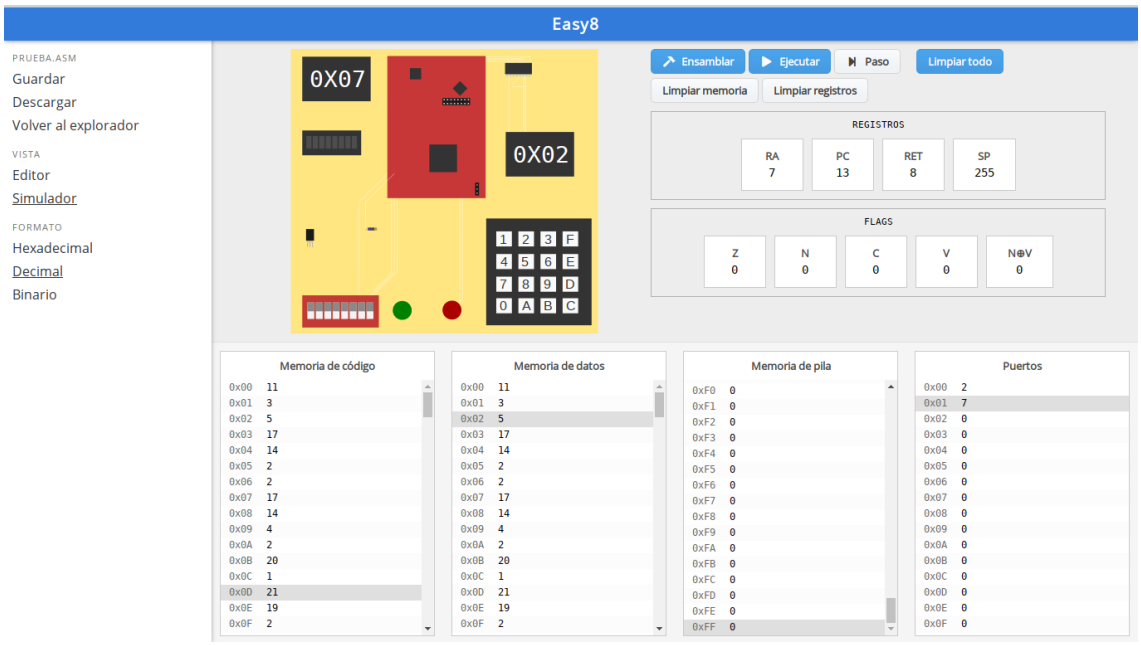

Figura 40: Resultado de la ejecución de un programa

8.2 Validación de Lighthouse

Lighthouse es una herramienta automatizada para evaluar el rendimiento, la calidad, la accesibilidad y corrección de aplicaciones web. Viene incluida entre las herramientas de desarrolladores de Google Chrome.

Lighthouse divide su evaluación en cinco apartados:

- **Rendimiento**: tiene en cuenta factores como el tiempo de carga de la página, el tiempo necesario para renderizarse y el uso de JavaScript.
- **Accesibilidad**: comprueba el buen uso del contraste y la correcta utilización de etiquetas y atributos de HTML, entre otras características.
- **Buenas prácticas**: estudia el uso de HTTPS, el uso de librerías de JavaScript con vulnerabilidades o tecnologías obsoletas.
- **SEO**: analiza el correcto uso de títulos, de la existencia de un fichero robots.txt válido o la necesidad de instalar *plugins*.

A cada uno de estos apartados, Lighthouse les asigna una puntuación del 0 al 100. A continuación, se presentan los resultados de las distintas pantallas del Easy8.

8.2.1 Autenticación

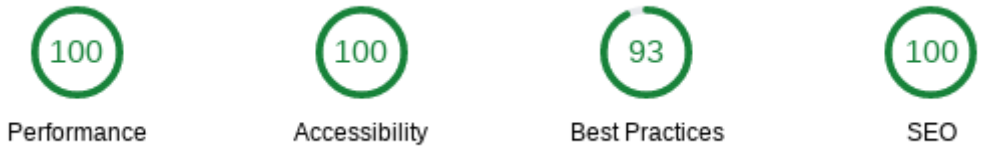

Figura 41: Evaluación de la pantalla de autenticación

El motivo por el que no se ha alcanzado el 100 en *Best practices* ha sido por no estar usando el protocolo HTTP/2 para cargar todos los recursos. Esto no se ha podido lograr porque Heroku no soporta HTTP/2. Este mismo problema se aplica en las siguientes pantallas.

8.2.2 Registro

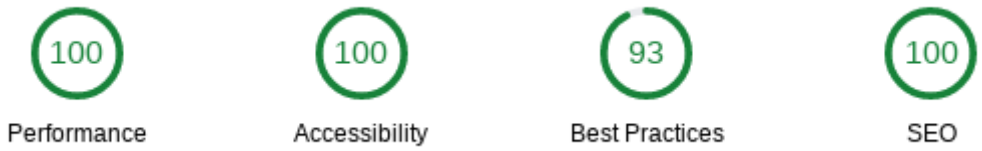

Figura 42: Evaluación de la pantalla de registro

8.2.3 Recuperación de contraseña

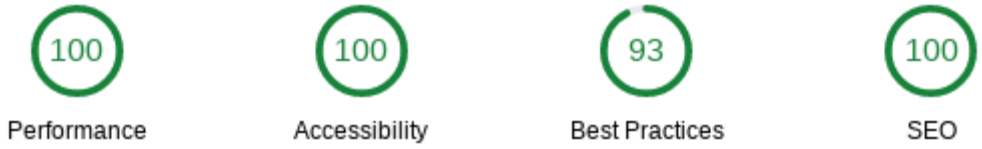

Figura 43: Evaluación de la pantalla de recuperación de contraseña

8.2.4 Explorador de ficheros

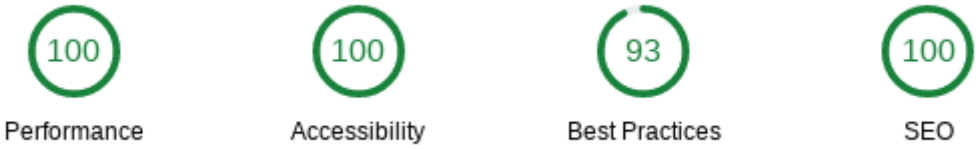

Figura 44: Evaluación del explorador de ficheros

8.2.5 Simulador

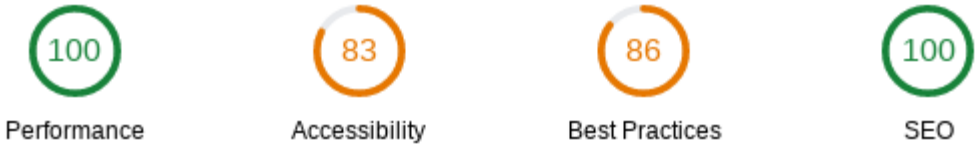

Figura 45: Evaluación del simulador

La evaluación de accesibilidad es más baja por dos motivos:

- **Bajo contraste que hay entre las direcciones de la memoria y el fondo**: se ha decidido ignorar esta advertencia porque reduciendo el contraste de la dirección se logra destacar más el valor del contenido de la memoria, que es lo más importante.
- **No etiquetar el área de texto**: normalmente, cualquier campo de texto tiene asociada una etiqueta que indica al usuario qué debe escribir allí. En el caso del simulador, tenemos el área de texto en la que se escribe el código que se ha decidido no etiquetar porque la etiqueta no aportaría ninguna aclaración al usuario.

La evaluación de *Best practices* ha sido peor que las anteriores por un uso poco apropiado que hace CodeMirror (la herramienta que se usa para implementar el editor de texto con coloreado de sintaxis) de los *listeners* de JavaScript.

8.3 Pruebas de visualización

A continuación, se mostrarán unas capturas de la pantalla del simulador en distintas resoluciones para probar que se ha cumplido con el requisito referente a la resolución mínima que se debe soportar.

Se ha hecho la prueba con la pantalla del simulador porque es la más compleja de todas (ver *Figura 46*, *Figura 47* y *Figura 48*).

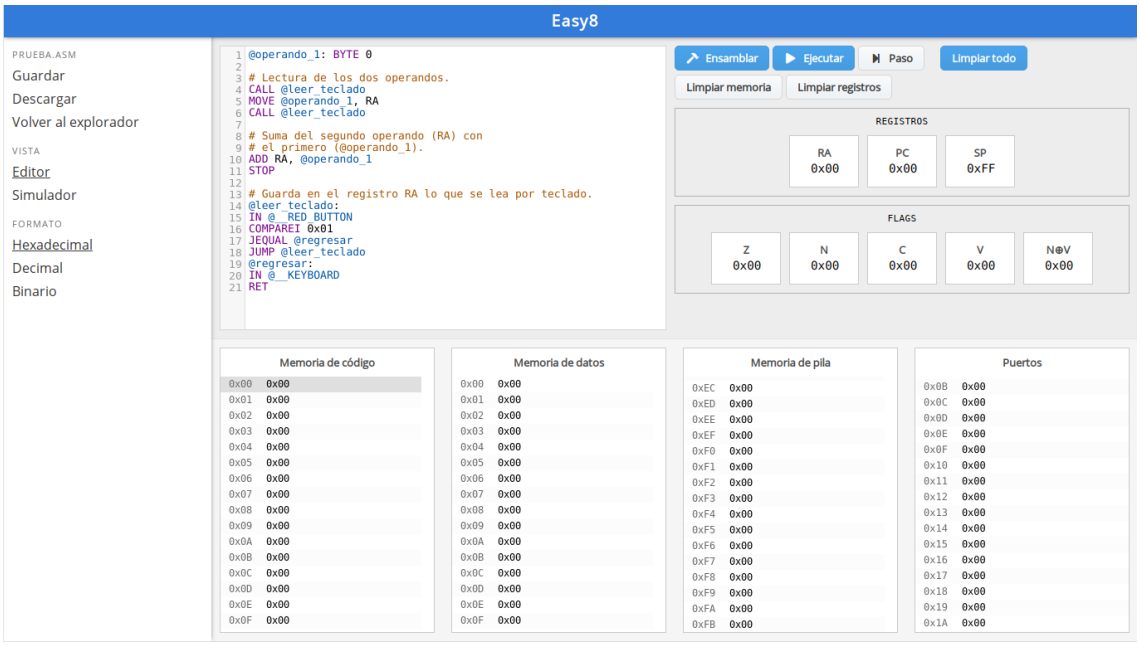

Figura 46: Resolución 1366x768

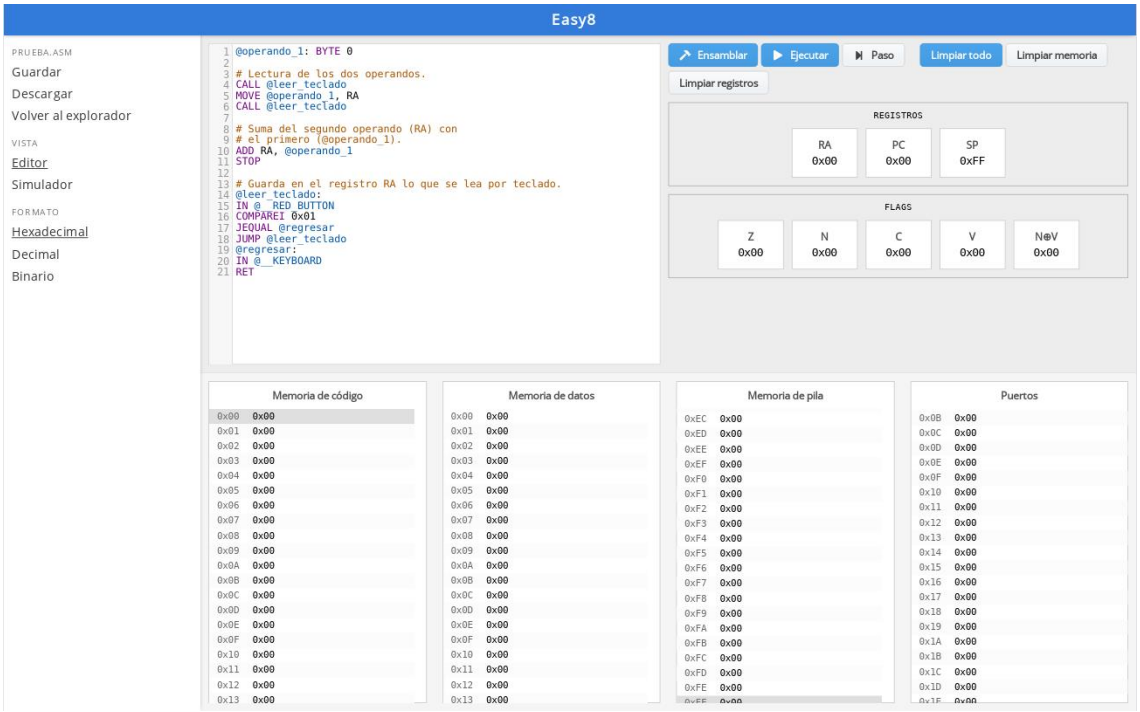

Figura 47: Resolución 1440x900

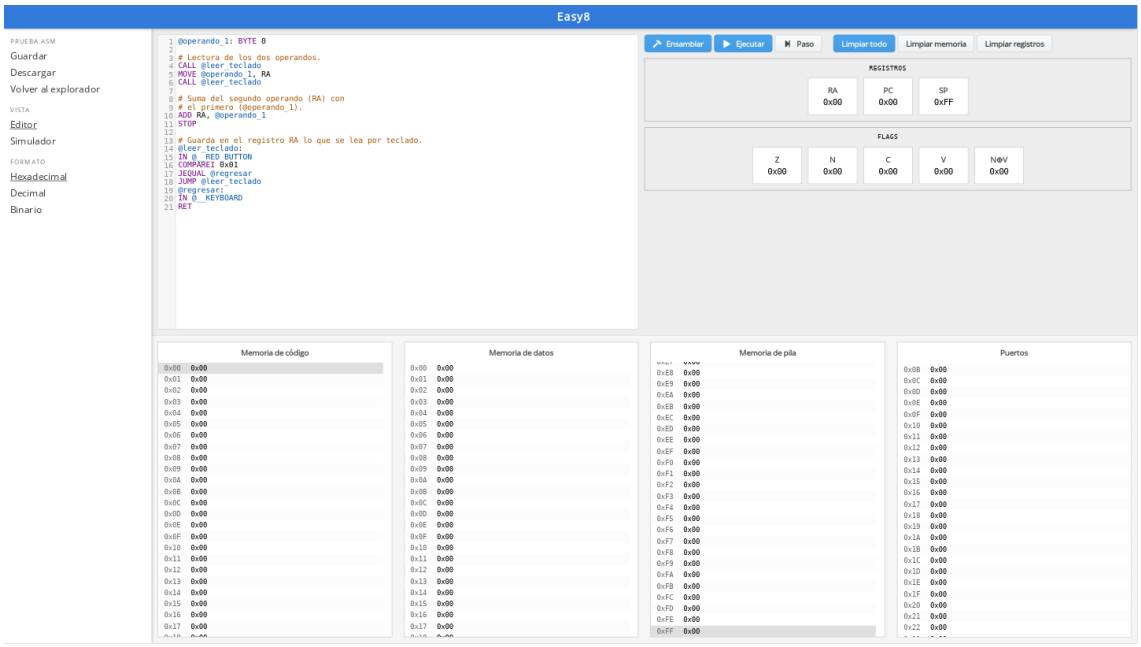

Figura 48: Resolución 1920x1080

9. Conclusiones

La finalidad de este trabajo era desarrollar una plataforma web en la que los usuarios pudieran simular y gestionar sus programas en lenguaje ensamblador para el computador Easy8. Los usuarios objetivo son los alumnos de Fundamentos de Computadores del Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación, que tendrán un primer acercamiento a los computadores y al lenguaje ensamblador usando el Easy8, por lo que había que desarrollar un sistema sencillo que mejorara la versión Java ya existente del simulador.

En este punto, se puede decir que se han cumplido satisfactoriamente todos los requisitos establecidos en la etapa de análisis, como se ha podido comprobar en el capítulo de *Pruebas*.

Aunque no es la primera vez que trabajo con Laravel y Vue (las dos herramientas más importantes que he usado para desarrollar la aplicación), desarrollar esta plataforma me ha ayudado a comprenderlos mejor.

Sí que ha sido una novedad para mí hacer un simulador de un computador. Aunque la arquitectura del Easy8 es muy sencilla, el mayor reto de este trabajo ha sido escribir un software capaz de ensamblar y ejecutar su código. Ha sido, sin embargo, la parte más satisfactoria de todo el proceso.

Por último, listaré algunas de las asignaturas del Grado que me han ayudado:

- En **Fundamentos de Computadores** adquirí los conocimientos sobre computadores y lenguaje ensamblador sin los que habría sido mucho más complicado realizar este trabajo.
- **Estructuras de datos y algoritmos** me ha servido, sobre todo, para hacer más eficiente el proceso de ensamblaje. Usar la estructura adecuada ayuda, además, a tener un código más breve y fácil de leer.
- Creo que uno de los errores más fáciles de cometer a la hora de diseñar una interfaz es hacerla para ti y no para el usuario de tu aplicación. La asignatura **Interfaces persona computador** ayuda a pensar en el usuario.
- En **Redes de computadores** tuve la oportunidad de conocer mejor el protocolo HTTP, que ha sido clave para la comunicación en este proyecto.
- **Ingeniería del software** ha sido imprescindible durante todo el proyecto, pero especialmente en la etapa de *Análisis*.
- **Bases de datos y sistemas de información** me ha servido para diseñar correctamente la base de datos de la aplicación.
- **Lenguajes de programación y procesadores de lenguajes** me ha ayudado a razonar sobre cómo procesar el código ensamblador que escribe el usuario.

10. Bibliografía

Chacon, S., & Straub, B. (s.f.). *Pro Git*. Nueva York: Apress.

Filipova, O. (2016). *Learning Vue.js 2*. Birmingham: Packt.

GeeksforGeeks. (s.f.). Obtenido de REST API Architectural Constraints: https://www.geeksforgeeks.org/rest-api-architectural-constraints/

Jones, C. (2017). Waterfall Development. En C. Jones, *Software methodologies: A quantitative guide* (págs. 479-480). Boca Ratón: CRC Press.

Pecoraro, C. J. (2015). *Mastering Laravel*. Birmingham: Packt.