



UNIVERSITAT  
POLITÈCNICA  
DE VALÈNCIA



Escola Tècnica  
Superior d'Enginyeria  
Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica  
Universitat Politècnica de València

# **Aplicación Web de bases de datos usando el Framework Ruby on Rails**

**TRABAJO FIN DE GRADO**

Grado en Ingeniería Informática

*Autor:* Daniel Arastey Aroca

*Tutor:* José Vicente Busquets Mataix

Curso 2017-2018



# Resum

Aquest treball està centrat en la creació d'una plataforma autoallotjada de *blogs* personals. Aquesta plataforma permet que el seu usuari comparteixi públicament continguts de tot tipus, i si així ho desitja, que els seus lectors puguin compartir les seues opinions a la secció de comentaris de cada entrada. Aquest projecte constitueix un senzill exemple de com construir una aplicació mitjançant el *framework* Ruby on Rails, a més de mostrar com es realitzen les operacions bàsiques sobre bases de dades relacionals amb aquesta ferramenta.

**Paraules clau:** Ruby, Rails, bases de dades, desenvolupament web, MVC

---

# Resumen

Este trabajo está centrado en la creación de una plataforma autoalojada de *blogs* personales. Esta plataforma permite que su usuario comparta de forma pública contenidos de todo tipo, y si así lo desea, que sus lectores puedan mostrar sus opiniones en la sección de comentarios de cada entrada. Este proyecto constituye un sencillo ejemplo de cómo construir una aplicación mediante el *framework* Ruby on Rails, además de mostrar cómo se realizan las operaciones básicas sobre bases de datos relacionales con dicha herramienta.

**Palabras clave:** Ruby, Rails, bases de datos, desarrollo web, MVC

---

# Abstract

This thesis is focused on the creation of a self-hosted personal blogging platform. This platform allows its user to share publicly contents of any kind, and if they want to, its readers will be able to show their opinions in the comments section of every post. This project represents a simple example of how to build an application using the Ruby on Rails Framework, as well as show how basic operations on relational databases are performed with said tool.

**Key words:** Ruby, Rails, databases, web development, MVC

---



# Índice general

---

<b>Índice general</b>	<b>V</b>
<b>Índice de figuras</b>	<b>IX</b>
<b>1 Introducción</b>	<b>1</b>
1.1 Motivación . . . . .	1
1.2 Objetivos . . . . .	1
1.3 Estructura de la memoria . . . . .	2
1.3.1 Nota acerca de las imágenes . . . . .	2
1.4 Agradecimientos . . . . .	2
<b>2 El lenguaje de programación Ruby</b>	<b>3</b>
2.1 Características . . . . .	3
2.2 <i>Software</i> en Ruby relevante . . . . .	4
2.3 Las gemas de Ruby . . . . .	4
<b>3 El <i>framework</i> Ruby on Rails</b>	<b>5</b>
3.1 ¿Qué es un <i>framework</i> ? . . . . .	5
3.2 Ruby on Rails . . . . .	5
3.2.1 Resumen técnico . . . . .	5
3.3 El patrón Modelo-Vista-Controlador . . . . .	6
3.3.1 Modelo . . . . .	7
3.3.2 Vista . . . . .	7
3.3.3 Controlador . . . . .	7
3.4 Rails y el patrón MVC . . . . .	8
3.4.1 La clase ActiveRecord . . . . .	8
3.4.2 La clase ActionController . . . . .	8
3.4.3 La clase ActionView . . . . .	8
<b>4 Base de datos: MariaDB</b>	<b>9</b>
4.1 Definición . . . . .	9
4.2 Características . . . . .	10
<b>5 Entorno de desarrollo</b>	<b>11</b>
5.1 El IDE: Atom . . . . .	11
5.2 El sistema de control de versiones: Git . . . . .	12
5.2.1 Acerca de Git . . . . .	12
5.3 Las hojas de estilo: Bootstrap . . . . .	14
5.4 Presentación de texto: Markdown . . . . .	15
5.5 Comentarios: Disqus . . . . .	15
5.6 Correo: SendGrid . . . . .	16
<b>6 La aplicación, análisis</b>	<b>17</b>
6.1 Análisis de requisitos . . . . .	17
6.2 Casos de uso . . . . .	17

6.3	Modelado de la aplicación	19
<b>7</b>	<b>La aplicación, desarrollo</b>	<b>21</b>
7.1	Instalación de las herramientas básicas e inicio	21
7.1.1	Instalación	21
7.1.2	Inicio del proyecto	22
7.1.3	Configuración de la base de datos	23
7.2	Definición del modelo inicial	24
7.2.1	Introducción de datos de prueba	26
7.3	Creación de controladores	26
7.4	Rutas	28
7.5	Creación de vistas	29
7.5.1	La gema Summarize	30
7.5.2	El índice de <i>posts</i>	30
7.5.3	Vistas adicionales para <i>posts</i>	31
7.6	Gestión de usuarios	31
7.7	Mailer y reinicio de contraseñas	34
7.7.1	Mailer	34
7.7.2	Reinicio de contraseñas	35
7.8	Categorías	37
7.8.1	Controlador	37
7.8.2	Vistas	38
7.9	Aplicación de estilos con Bootstrap	38
7.10	Presentación de contenido	41
7.11	Comentarios	42
7.12	Configuración	43
7.13	Feed RSS	44
<b>8</b>	<b>La aplicación, despliegue</b>	<b>47</b>
8.1	Elección del servidor e instalación inicial	47
8.2	Instalación de dependencias	48
8.2.1	Ruby y Rails	48
8.2.2	MariaDB	48
8.2.3	Otras dependencias	49
8.3	Instalación de la aplicación	49
8.4	Despliegue público	50
8.4.1	Configuración de arranque automático	51
8.4.2	Instalación y configuración de NGINX	51
8.4.3	Cifrado con Let's Encrypt	53
<b>9</b>	<b>Conclusiones</b>	<b>55</b>
9.1	Futuras mejoras	55
	<b>Bibliografía</b>	<b>57</b>
<hr/>		
	Apéndice	
<b>A</b>	<b>Código fuente</b>	<b>59</b>
A.1	app/controllers	59
A.1.1	admin_controller.rb	59
A.1.2	application_controller.rb	59
A.1.3	categories_controller.rb	59
A.1.4	password_resets_controller.rb	60

---

A.1.5	posts_controller.rb	61
A.1.6	sessions_controller	63
A.1.7	users_controller.rb	63
A.2	app/mailers	64
A.2.1	application_mailer.rb	64
A.2.2	user_mailer.rb	65
A.3	app/models	65
A.3.1	category.rb	65
A.3.2	post.rb	65
A.3.3	user.rb	65
A.4	app/views	66
A.5	config	66
A.5.1	initializers/application_controller_renderer.rb	66
A.5.2	initializers/junablog.erb	66
A.5.3	initializers/mail.erb	66
A.5.4	database.yml	66
A.5.5	junablog.yml	67
A.5.6	routes.rb	67
A.6	db	68
A.6.1	seeds.rb	68
A.7	Raíz del proyecto	68
A.7.1	Gemfile	68





# Índice de figuras

---

3.1	Diagrama de la arquitectura MVC. . . . .	7
5.1	Pantalla de inicio de sesión de la aplicación. . . . .	14
6.1	Casos de uso de usuarios no autenticados. . . . .	18
6.2	Casos de uso de usuarios autenticados. . . . .	18
6.3	Base de datos de la aplicación. . . . .	19
7.1	Instalación de la gema Rails. . . . .	22
7.2	Aplicación «Hola Mundo!» generada automáticamente por Rails. . . . .	23
7.3	Salida del comando de creación de bases de datos . . . . .	24
7.4	Salida de la aplicación de generación de modelos. . . . .	24
7.5	Contenido de fichero de migración. . . . .	25
7.6	Contenido de «db/schema.rb» tras migración. . . . .	25
7.7	Contenido de «models/category.rb» . . . . .	25
7.8	Contenido de «models/post.rb» . . . . .	25
7.9	Contenido de «db/seeds.rb» . . . . .	26
7.10	Contenido de la tabla «categories» después de cargar las semillas. . . . .	26
7.11	Ficheros creados por «rails generate controller» . . . . .	27
7.12	Método «index» del controlador Posts . . . . .	27
7.13	Rutas para el controlador Posts . . . . .	28
7.14	Rake mostrando las rutas actuales . . . . .	29
7.15	Contenido de la vista por defecto «application.html.erb». . . . .	29
7.16	Contenido de la vista por «posts/index.html.erb». . . . .	30
7.17	Contenido de la vista «posts/edit.html.erb». . . . .	31
7.18	Rutas para los controladores «Sessions» y «Admin». . . . .	32
7.19	Métodos para autorización en «application_controller.rb». . . . .	32
7.20	Gestión de sesiones en «users_controller.rb». . . . .	33
7.21	Borrado de usuario con cierre de sesión «users_controller.rb». . . . .	34
7.22	Inicializador de configuración de conexión al servidor de correo. . . . .	35
7.23	Correo electrónico enviado por el <i>mailer</i> mediante SendGrid. . . . .	35
7.24	Rutas autogeneradas para el controlador «password_resets». . . . .	36
7.25	Código de «app/views/layouts/application.html.erb» tras incluir Bootstrap. . . . .	39
7.26	Esqueleto de vista siguiendo la distribución en rejilla mediante Boots- trap. . . . .	40
7.27	Índice de <i>posts</i> con distribución en rejilla basada en Bootstrap. . . . .	40
7.28	Generador personalizado de HTML mediante Redcarpet y método para su uso. . . . .	41
7.29	Entrada del <i>blog</i> con sección de comentarios basada en Disqus. . . . .	42
7.30	Contenido del fichero «config/junablog.yml». . . . .	43

7.31	Método «feed» en el controlador de <i>posts</i> . . . . .	44
7.32	Generador de XML para el <i>feed</i> RSS. . . . .	44
7.33	<i>Feed</i> RSS en el lector integrado de Firefox. . . . .	45
8.1	Fichero «config/unicorn.rb». . . . .	50
8.2	<i>Script</i> de autoarranque de Unicorn. . . . .	51
8.3	Configuración de NGINX. . . . .	52
8.4	Junablog con certificado válido emitido por Let's Encrypt. . . . .	54

---

# CAPÍTULO 1

## Introducción

---

A continuación se ofrece una breve introducción acerca del presente trabajo. Los aspectos que se mencionan aquí serán explicados con más detalle en los próximos capítulos.

### 1.1 Motivación

---

Dada la situación actual de la computación personal, que se está trasladando cada vez más a la red, en lugar de ocurrir en nuestros equipos, es una buena idea aprender a desarrollar aplicaciones usando herramientas dedicadas a la programación de *software* basado en web, dado que existe mercado para este tipo de trabajos para mucho tiempo.

Además de por dicho motivo, la necesidad personal del autor de tener un *blog* personal, ha propiciado que se optara por dicha aplicación, a pesar del gran abanico de opciones disponibles para ese fin.

### 1.2 Objetivos

---

El objetivo principal de este proyecto es la implementación de una plataforma de *blogs* personales autoalojada, mediante el uso del *framework* Ruby on Rails. Dado que este tipo de aplicaciones, aúnan las operaciones básicas sobre bases de datos, y tienen elementos bien definidos mostrando los distintos comportamientos de dichas operaciones.

Como objetivo adicional, se incluye el despliegue de la aplicación en un servidor accesible públicamente desde internet. Y la integración de esta con servicios de terceros, como Disqus (para gestión de comentarios) o la red de distribución de contenidos de Bootstrap para el alojamiento de las hojas de estilo. Además, la aplicación debe ser capaz de enviar correo electrónico a los usuarios para el restablecimiento de contraseñas, usando una combinación de código propio y un servicio externo, para garantizar el funcionamiento independientemente de la disponibilidad de un MTA<sup>1</sup> en el sistema.

---

<sup>1</sup>*Mail Transport Agent*, aplicación reponsable de la gestión del correo electrónico en el servidor.

## 1.3 Estructura de la memoria

---

Esta memoria está dividida en múltiples capítulos, en los que se cubren distintos aspectos acerca del trabajo realizado, incluyendo descripciones e información relevante sobre las tecnologías y herramientas utilizadas, así como del trabajo en sí mismo.

En primer lugar se describe la instrumentación del trabajo, empezando por Ruby, para a continuación explicar las características del *framework* Ruby on Rails y su relación con la arquitectura Modelo-Vista-Controlador. A continuación, se describe la base de datos utilizada como almacén de información en esta aplicación. Acto seguido se presentan las diversas herramientas y servicios de terceros utilizados, tanto para el desarrollo, como para aportar funcionalidad añadida al *blog*. Al terminar esta serie de secciones, se explica el desarrollo del propio trabajo, y las conclusiones extraídas de este.

### 1.3.1. Nota acerca de las imágenes

Todas las imágenes utilizadas son propias, extraídas de fuentes de dominio público o con licencias que permiten su reutilización. Se acreditarán aquellas imágenes que no sean propias, independientemente de su origen o licencia.

## 1.4 Agradecimientos

---

Quisiera dar mis más sinceros agradecimientos a todos aquellos que en algún momento se han puesto pesados diciéndome que acabe de una vez, ya sean familiares, amigos o conocidos. También a los creadores y la comunidad de Wikipedia y Stackoverflow y a las decenas de bandas que me han mantenido despierto cuando debería estar durmiendo.

A Michael Hartl, autor del fantástico libro «Ruby on Rails Tutorial», por crear el contenido, y por publicarlo de manera gratuita.

*Kaikki ystäväni Metropolia AMK:ssa, kiitos paljon.*

*Al mijn vrienden en collega's bij Kahuna, hartelijk bedankt.*

---

## CAPÍTULO 2

# El lenguaje de programación Ruby

---

Ruby es un lenguaje de programación orientado a objetos e interpretado, lanzado al público bajo una licencia de *software* libre en 1995 [1]. Desde su lanzamiento, ha ido ganando popularidad, causando que disponga de librerías para desarrollar desde el más sencillo *script* hasta aplicaciones más complejas, tanto para consola, como gráficas o de servidor en red.

### 2.1 Características

---

Algunas de las características principales de Ruby son [2]

- Orientación a objetos.
- Flexibilidad. Ruby permite alterar libremente partes esenciales.
- Uso de variables sin declaración previa y con definición de alcance (global, local o de instancia)
- Sintaxis sencilla, similar a la de Python.
- Compatibilidad con expresiones regulares de PERL (PCRE, *Perl Compatible Regular Expressions*).
- Multiplataforma.
- Manejo de excepciones para facilitar la gestión de errores en tiempo de ejecución.
- Gestión de hilos independiente del Sistema Operativo subyacente.

Estas son solo algunas de las propiedades del lenguaje, que lo han convertido en la opción preferida para muchos desarrolladores, siendo Ruby el que ocupa el décimo puesto en el índice TIOBE [3], que presenta los lenguajes más populares del momento.

## 2.2 *Software* en Ruby relevante

---

A continuación se listan unos pocos de los numerosos programas de uso extendido que usan este lenguaje:

- **Brew/Homebrew:** Homebrew es un gestor de paquetes similar a apt-get (en distribuciones GNU/Linux basadas en Debian) para macOS X, que simplifica la instalación de programas desde el terminal.
- **Metasploit Framework:** Uno de los principales programas utilizados en auditorías de seguridad de sistemas y redes. Permite el reconocimiento de estos en busca de vulnerabilidades para su posterior explotación.
- **Puppet:** Aplicación para la automatización de configuraciones de servidores UNIX. Utilizada de forma muy extensa para la gestión de infraestructuras de tecnologías de la información.
- **Vagrant:** Herramienta para la creación, configuración y gestión de máquinas virtuales. Popular para el despliegue de máquinas en plataformas en la nube como DigitalOcean o Amazon EC2.

## 2.3 Las gemas de Ruby

---

Como ocurre con la mayoría de lenguajes, a menudo es necesario utilizar código de terceros para ampliar la funcionalidad de Ruby, ya sea para permitir el uso de APIs (siglas en inglés para interfaz de programación de aplicaciones), como para añadir funciones útiles (tales como la posibilidad de conectar a bases de datos o permitir crear interfaces gráficas para nuestros programas en Ruby).

Estos añadidos al lenguaje, se conocen en Ruby como gemas. Las gemas no solo aportan extensiones del lenguaje a modo de librerías, sino que también se presentan habitualmente como ejecutables con funcionalidad propia, y que pueden ser usados fuera de *scripts*, es decir, que no es necesario escribir un programa para hacer uso de algunas gemas.

Las gemas se distribuyen a través de internet mediante un repositorio de acceso público conocido como Rubygems[4]. Es posible ver más información sobre las distintas gemas desde la web de Rubygems. No es necesario utilizar la web para descargar y utilizar una gema en nuestro proyecto, es posible descargarlas desde línea de comandos utilizando la siguiente orden, incluida con la distribución oficial de Ruby:

```
gem install $NOMBRE_GEMA -v $VERSION
```

El uso del parámetro -v permite descargar versiones antiguas de gemas concretas, en caso de que sea necesario para satisfacer dependencias o por incompatibilidad con la edición de Ruby que se esté usando.

---

## CAPÍTULO 3

# El *framework* Ruby on Rails

---

El *framework* Ruby on Rails, está formado por una serie de gemas y ejecutables, combinados para el desarrollo de aplicaciones web dinámicas utilizando el lenguaje de programación Ruby.

### 3.1 ¿Qué es un *framework*?

---

Un *framework*, es un conjunto de herramientas *software* combinadas de forma que simplifican o automatizan tareas que de otro modo serían demasiado complejas o tediosas para sus usuarios, dejando la parte más complicada lista para ser utilizada. Además de Rails, del que se hablará a continuación, algunos ejemplos son: Bootstrap, que permite aplicar plantillas vistosas y adaptativas a sitios web sin apenas necesidad de alterar el código CSS, o Laravel, destinado al desarrollo de aplicaciones web con PHP siguiendo el patrón Modelo-Vista-Controlador, del que se hablará más adelante.

### 3.2 Ruby on Rails

---

Ruby on Rails es un *framework* para desarrollo de aplicaciones web y APIs REST basado en la arquitectura Modelo-Vista-Controlador. Su desarrollo fue iniciado en 2004, y a finales de 2005 vio la luz la primera versión estable [5], en una época en la que se estaba generalizando el acceso a internet y la computación personal empezaba a trasladarse cada vez más a la red.

Ruby on Rails se distribuye al público bajo la licencia MIT, que permite su libre uso, modificación y distribución. [6] Esta licencia permite el uso de las herramientas de Rails en productos comerciales. Esto ha permitido que Rails haya sido la solución elegida para el desarrollo de plataformas populares como Airbnb, Kickstarter o Github.

#### 3.2.1. Resumen técnico

Rails se ofrece de forma estándar como una gema de Ruby para su sencilla instalación mediante el comando *gem*, y aunque es posible instalarlo desde los

repositorios de paquetes de las principales distribuciones GNU/Linux, esta práctica no es recomendable, ya que existe la posibilidad de que Rails o alguna de sus dependencias no sea completamente compatible con la versión de Ruby instalada en el sistema, pudiendo causar un mal funcionamiento de la aplicación.

A no ser que se especifique lo contrario, el *framework* incluye por defecto un servidor web básico (llamado Puma) y una librería que contiene el controlador de la base de datos SQLite. Si bien estas herramientas proporcionan la funcionalidad suficiente para un entorno de pruebas, estas suelen tener serias limitaciones para soportar las altas cargas causadas por el tráfico que puede llegar a recibir una aplicación disponible en internet, por lo que es recomendable cambiarlas por otras más aptas para producción. Es posible editar la configuración del proyecto para definir entornos de desarrollo y producción, con gemas específicas para cada uno, permitiendo por ejemplo usar SQLite en desarrollo y PostgreSQL en producción.

La arquitectura completa de un proyecto en Rails está pensada para utilizar únicamente Ruby, eliminando la necesidad de otros *frameworks* de mezclar múltiples lenguajes, dependiendo de si son necesarios para control, acceso a datos, o para interfaz. Para estos dos últimos, Rails ofrece ActiveRecord y ERB, respectivamente. ActiveRecord es la capa responsable de representar la lógica y datos del sistema, y ERB es una herramienta que permite incrustar código Ruby en documentos HTML y programar directamente sobre la vista, de una forma similar a PHP, sin escribir código que la genere desde cero.

Rails, además, hace uso de Rspec, una utilidad que permite la realización de pruebas unitarias directamente desde la consola, sin tener que lanzar la aplicación completa, facilitando así los procesos de desarrollo basado en pruebas.

A todo esto, cabe añadir, que al estar completamente basado en Ruby, que es un lenguaje interpretado, permite lanzar la aplicación de manera rápida, al no necesitar volver a compilar nada tras hacer cambios a su código.

### 3.3 El patrón Modelo-Vista-Controlador

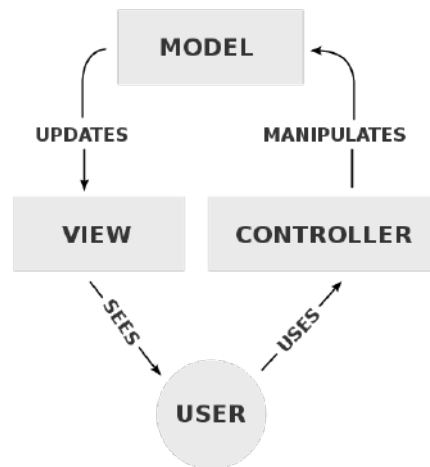
---

El Patrón Modelo-Vista-Controlador (MVC) es una arquitectura de *software* que separa la lógica y los datos de su representación y las comunicaciones. Esta arquitectura está diseñada para facilitar el mantenimiento de las aplicaciones, la reutilización de código y la separación de tareas. Los elementos de MVC cumplen las siguientes funciones:

- **Modelo:** Define los datos del sistema, gestionando el acceso a estos y su transformación.
- **Vista:** Presenta la información al usuario de la aplicación. Es posible definir múltiples vistas para la misma información.
- **Controlador:** Recibe entradas y las convierte a comandos para el modelo o la vista. Habitualmente, la entrada del usuario se transmite al controlador mediante elementos del interfaz gráfico, perteneciente a la vista.



A continuación se explican los diferentes elementos con más detalle.



**Figura 3.1:** Diagrama de la arquitectura MVC. Liberada al dominio público por el usuario de Wikimedia Commons RegisFrey.

### 3.3.1. Modelo

Este componente de MVC es responsable del acceso a datos. En él se definen los diferentes mecanismos necesarios para el acceso y modificación de la información que maneja la aplicación. Dicha información puede estar almacenada de múltiples maneras, en el caso de Rails, se hace uso de una base de datos para ese fin. Por este motivo, el modelo es el módulo de MVC en el que se deberán gestionar las consultas a la base de datos necesarias para cada una de las operaciones CRUD (*Create, Read, Update and Delete*; Crear, Leer, Actualizar y Eliminar).

### 3.3.2. Vista

Esta capa se encarga de presentar a los usuarios la información usando los elementos de interfaz necesarios para dicho fin. Para Frameworks como Rails, Symfony o Django, la vista se suele presentar mediante HTML, aunque también es posible presentarla de otras maneras diferentes (como por ejemplo XML o JSON para APIs REST que presentan la información en un formato que ha de ser consumido por otra aplicación).

### 3.3.3. Controlador

El controlador actúa como intermediario entre los otros dos elementos. Solicita información al modelo y la transmite a la vista, y recibe información del usuario (mediante elementos de interfaz presentados por la vista) que posteriormente pasará al modelo. Para ese fin, el controlador es el módulo donde se alojan los manejadores de eventos, que reaccionan a las distintas acciones, según sea necesario.

## 3.4 Rails y el patrón MVC

---

Rails ofrece una serie de clases y herramientas dedicadas exclusivamente a la implementación de una arquitectura MVC.

### 3.4.1. La clase ActiveRecord

Este módulo, que cumple el papel de Modelo en MVC, está diseñado para manejar todas las tareas de la aplicación relacionadas con la base de datos. Esto incluye la conexión, y la recuperación y almacenamiento de información desde la base de datos.

Este componente se basa en la abstracción de la base de datos, que permite que el desarrollador pueda realizar todas las tareas necesarias en la base de datos (incluyendo la creación de tablas) sin tener que utilizar el lenguaje de consultas del SGBD en cuestión. Para esto incluye la herramienta de migraciones.

### 3.4.2. La clase ActionController

El controlador gestiona la lógica del programa, actuando como unión entre el navegador, la capa de presentación y los datos de la aplicación. Se encarga de recibir peticiones del navegador y decidir cómo gestionarlas, además de recuperar información del modelo y transmitirla a la vista.

Esta clase, además proporciona otras funcionalidades, como la autenticación HTTP, el manejo de Cookies y la gestión de excepciones. También aporta herramientas para mejorar la seguridad de la aplicación como son el forzado de SSL y la prevención de falsificación de peticiones.

### 3.4.3. La clase ActionView

Al igual que ActionController, esta clase forma parte de la librería ActionPack. El módulo ActionView, está pensado para incluir únicamente la lógica de presentación. Es decir, no debería incluir lógica de aplicación ni acceder directamente a la base de datos bajo ningún concepto.

Es posible que una vista incluya únicamente HTML, sin usar Ruby, pero es altamente probable que sea necesario generar la vista en base a los resultados de una operación realizada por el controlador. Para ello, esta clase hace uso de ERB<sup>1</sup>, un formato de plantillas que permite incluir código Ruby en medio de ficheros HTML. Estos ficheros (que reciben la extensión .html.erb) son procesados por la clase ActionView, sustituyendo el código Ruby incrustado por el resultado de su ejecución. Un ejemplo de HTML con ERB embebido es el siguiente:

```
<p><%= 'Hello World!' %></p>
```

El código situado entre « % » sería evaluado por Ruby, y sería situado en el HTML resultante entre las etiquetas « <p> ».

---

<sup>1</sup>Embedded Ruby

---

## CAPÍTULO 4

# Base de datos: MariaDB

---

A pesar de que Rails incluye por defecto la base de datos SQLite y las gemas necesarias para su uso en el momento de crear un nuevo proyecto, se ha decidido optar por MariaDB por ser considerada más apta para su uso en producción.

### 4.1 Definición

---

MariaDB es un sistema de gestión de bases de datos relacionales (SGBDR) lanzado como *fork* del proyecto MySQL en 2009, motivado por la incertidumbre del futuro de este tras ser adquirido por Oracle Corporation.[7] Al tratarse de un proyecto basado en una copia exacta del código fuente de MySQL, MariaDB puede utilizarse como reemplazo directo de este, utilizando las mismas librerías y herramientas para el acceso y uso del SGBDR.

Al igual que MySQL, MariaDB permite la ejecución de instrucciones en paralelo y de forma distribuida para múltiples usuarios al mismo tiempo de una manera muy robusta. Esto permite que sea una excelente opción como SGBDR tanto para aplicaciones que se ejecuten en un equipo, como en red (tanto red local como internet).

Este sistema utiliza el lenguaje SQL<sup>1</sup>, considerado estándar para el uso de base de datos. Dicho lenguaje puede ser utilizado para todas las tareas de gestión de la base de datos: desde crear, consultar, alterar y borrar registros, hasta gestionar usuarios, privilegios y otras tareas de administración de la base de datos.

Dado que es un reemplazo directo de MySQL que no requiere cambios fuera del SGBDR para su uso, ha sustituido a este como sistema de bases de datos para numerosas entidades de gran relevancia en internet, como Google[8] o Wikipedia[9]. Además, debido a los cambios de licencia que podía conllevar la compra de MySQL por Oracle, algunas de las distribuciones más importantes de GNU/Linux han retirado MySQL de sus repositorios en favor de MariaDB. Entre dichas distribuciones se encuentran algunas como Debian [10], Arch Linux[11] o Red Hat Enterprise Linux.[12]

MariaDB se distribuye bajo la licencia de *software* libre GNU GPL versión 2.[13]

---

<sup>1</sup>Siglas en inglés de *Structured Query Language*, Lenguaje de Consulta Estructurado

## 4.2 Características

---

Estas son algunas de las características más importantes de MariaDB:

- Multiplataforma: Disponible para Windows, GNU/Linux y BSD.
- Multiusuario. El SGBDR dispone de funciones para el control de acceso basado en roles (RBAC). [14]
- Disponibilidad de APIs para gran cantidad de lenguajes de programación, propiciada por la disponibilidad de estas para su uso con MySQL.
- Funcionamiento multi-hilo, permitiendo el uso de múltiples CPU.
- Procedimientos almacenados, similares a los bloques PL/SQL de Oracle. Estos permiten ejecutar código al cumplirse la condición de disparo de un *trigger*.
- Creación de vistas, que permiten limitar qué datos se presentan a los usuarios del SGBDR y de qué forma.
- Cacheado de consultas, permitiendo devolver rápidamente resultados de consultas habituales. [15]
- Esquema de información, que representa la estructura y configuración de la base de datos de forma accesible mediante consultas SQL.
- Posibilidad de configurar instancias replicadas en múltiples host para alta disponibilidad y tolerancia a fallos. [16]
- Soporte de protocolos SSL y TLS para el cifrado punto a punto de las comunicaciones. (Deshabilitado por defecto [17] )

Esta serie de características, heredadas de MySQL y posteriormente ampliadas, convierten a MariaDB en una aplicación idónea en despliegues de todo tipo de tamaños que requieran el uso de bases de datos relacionales.

---

## CAPÍTULO 5

# Entorno de desarrollo

---

En este capítulo se describen las herramientas utilizadas para el desarrollo de este trabajo, más allá de Ruby/Rails y MariaDB.

### 5.1 El IDE: Atom

---

Para el desarrollo de las partes basadas en código, se ha elegido el editor de textos Atom<sup>1</sup>, desarrollado por Github. Aunque es posible utilizar IDEs<sup>2</sup> dedicados para Ruby como Rubymine, entornos en línea compatibles con Ruby como Cloud9 IDE (ahora parte de Amazon Web Services), o IDEs genéricos como Netbeans con añadidos para incluir compatibilidad con Ruby/Rails, se ha optado por usar un editor más sencillo, dado que no se van a aprovechar todas las funcionalidades que estos entornos ofrecen.

Atom está pensado para ser usado como editor de código, por lo que presenta una serie de características que lo hacen idóneo para su uso en desarrollo de aplicaciones:

- Es un editor multiplataforma, programado usando Electron, que lo hace compatible con Windows, GNU/Linux y OS X.
- Resaltado de sintaxis multilenguaje: incluyendo por defecto C, C++, C#, CSS, Go, HTML, Javascript, PHP, Python, Ruby/Rails y muchos más. Además de existir paquetes para añadir resaltado para otros lenguajes de programación.
- Navegador de ficheros integrado en el editor, permitiendo la carga rápida de ficheros de un proyecto.
- Gestor de paquetes integrado, utilizado para la búsqueda, instalación y desarrollo de *plug-ins*, o añadidos para proporcionar funcionalidad adicional.
- Edición simultánea de múltiples ficheros. El editor se separa en paneles para permitir editar múltiples ficheros, o consultar uno mientras se edita otro.

---

<sup>1</sup><https://atom.io/>

<sup>2</sup>*Integrated Development Environment*, Entorno Integrado de Desarrollo

Funcionalidad útil para mostrar la documentación de una clase en un panel al mismo tiempo que en el otro panel se escribe código que usa dicha clase.

- Autocompletado de código. Esta función permite completar de forma automática nombres de variables, funciones, o estructuras básicas del código (como bucles, condicionales o prototipos de funciones).
- Búsqueda y reemplazamiento avanzado. Atom es capaz de buscar y reemplazar no solo en el fichero en el que se está trabajando en ese momento, sino también en todos los ficheros del proyecto, sin abrirlos previamente.

Además de estas características, Atom ofrece integración con Git para control de versiones, siendo posible usar otros sistemas como Subversion o Mercurial mediante la instalación de paquetes adicionales.

## 5.2 El sistema de control de versiones: Git

---

Para el desarrollo de aplicaciones es necesario llevar un control de los cambios que se realizan en el código, con el propósito de poder revertir cambios que hayan podido introducir un fallo en el producto, así como identificar en qué momento y por qué usuarios se han realizado estos cambios. Aunque es posible realizar esto de forma manual (mediante copias manuales, ficheros diferenciales y documentos con el histórico), o automatizando partes mediante herramientas como Rsync o Dropbox, es recomendable utilizar un sistema de control de versiones, que unifica en una sola herramienta todas las tareas y gestiona los conflictos que puedan surgir al usar un proceso manual en un entorno con múltiples programadores.

Existen numerosos sistemas de control de versiones: Subversion, CVS, Mercurial, Bazaar, etcétera. Para el desarrollo de esta aplicación, se ha escogido Git, por su simplicidad de uso e integración con el editor de texto utilizado. Además, se usa este sistema para la redacción de esta memoria, utilizando la herramienta a modo de sistema de copia de seguridad remota y como método de sincronización del documento en los múltiples equipos en que se ha trabajado.

En internet se ofrece un gran abanico de opciones para alojar repositorios Git, el más extendido es Github<sup>3</sup>, aunque para este proyecto se ha optado por usar el servicio alojado por Atlassian, Bitbucket<sup>4</sup>, ya que este ofrece de forma gratuita un número ilimitado de repositorios privados (que es una necesidad temporal de este trabajo mientras está en desarrollo), a diferencia de Github, donde es necesario usar una cuenta de pago para dicho fin.

### 5.2.1. Acerca de Git

Git es un sistema de control de versiones distribuido que nace en 2005 a partir de la necesidad de Linus Torvalds y el resto de colaboradores del *kernel* del sistema operativo Linux de abandonar BitKeeper, tras haber cambiado estos últimos

---

<sup>3</sup><https://github.com/>

<sup>4</sup><https://bitbucket.org/>

la licencia de uso de su sistema. [18] En aquel momento, los sistemas de control de versiones libres disponibles no satisfacían las necesidades de Linus, por lo que inició el desarrollo de un sistema nuevo, que funcionara de forma similar a Bit-Keeper y que incluyera salvaguardas contra la corrupción de datos, accidental o maliciosa.[19]

A lo largo de los años, Git ha recibido actualizaciones que han introducido funcionalidades haciéndolo muy interesante para desarrollos de todo tipo. Prueba de esto, es que Git se ha convertido en el sistema de control de versiones elegido en la mayoría de las empresas de internet más importantes, como Facebook o Google, y para gestionar el desarrollo de proyectos de gran tamaño, como Ruby on Rails o KDE.

Algunas de las características que lo diferencian de sistemas clásicos como CVS o SVN son:

- Soporte para desarrollo no-lineal, es decir, permite duplicar el código y separarlo de la rama principal desde el propio repositorio, independizando los cambios que se hagan entre ramas, hasta el momento de unirlos (*merge*).
- Cada copia del repositorio recibe el historial de cambios completo, pudiendo revertir a una versión anterior incluso sin disponer una copia del repositorio en el momento de lanzar dicha versión. Esto es útil para crear ramas o *forks* en base al estado deseado del proyecto.
- Uso de protocolos estándar como HTTP, FTP o SSH para realizar conexiones, evitando tener que implementar protocolos propios y haciendo que la mayoría de cortafuegos permitan su uso sin cambios.
- Autenticación de la historia. El histórico de cambios se utiliza para definir el identificador del estado actual, haciendo que cualquier intento de sustituir el contenido histórico (con fines maliciosos o no) sea fácilmente detectable.
- Soporte para compresión al vuelo, ahorrando ancho de banda para servidores y evitando alcanzar cuotas máximas de uso de datos para los usuarios.

Git se ofrece como herramienta de consola de comandos, y está disponible en los repositorios de las principales distribuciones del sistema operativo GNU/Linux, viniendo preinstalado por defecto en algunas de ellas. Para Windows, se proporciona una instalación de MinGW<sup>5</sup>, que contiene la consola de comandos Bash de GNU y permite el uso de Git desde esta. Existen además herramientas para trabajar con repositorios Git mediante interfaz gráfico, como gitk (que se proporciona con el instalador de Git para Windows) o el cliente de escritorio de Github<sup>6</sup>.

---

<sup>5</sup>Minimalist GNU for Windows <http://www.mingw.org/>

<sup>6</sup><https://desktop.github.com/>



## 5.3 Las hojas de estilo: Bootstrap

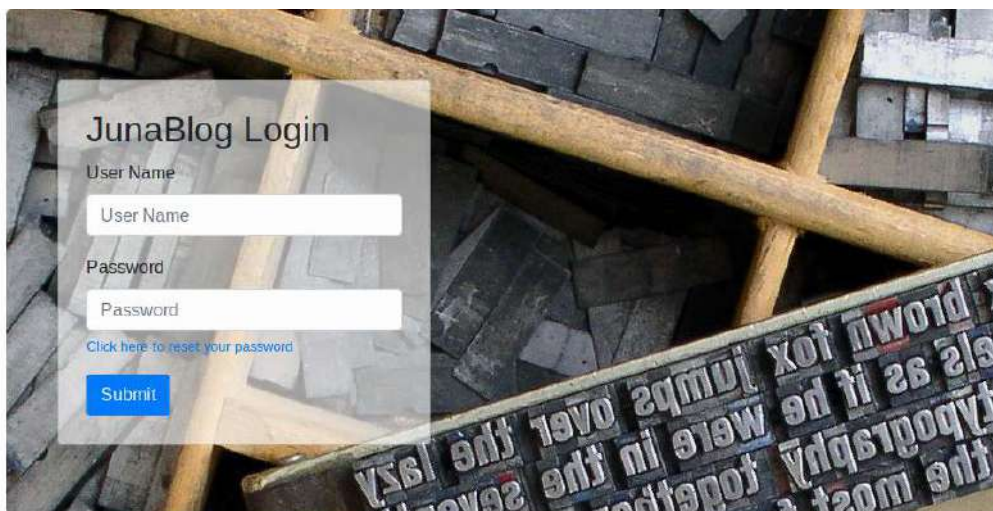
Cualquier aplicación web moderna, hace uso de hojas de estilo en cascada (CSS) para definir el aspecto visual que debe tener. Desde los colores del texto, hasta la posición de los distintos elementos, pasando por efectos y animaciones, todo lo relacionado con el aspecto visual se gestiona desde estas.

Dado que el apartado gráfico no forma parte del alcance de la aplicación, se ha decidido usar el *framework* Bootstrap para obtener un resultado vistoso, con poco esfuerzo. Bootstrap<sup>7</sup> está formado por un conjunto de herramientas creadas por Twitter inicialmente para ser utilizado en su web, pero que posteriormente fue lanzado al público para su uso sin restricciones.

Bootstrap contiene el código CSS y Javascript necesario para definir el diseño de una página de una forma sencilla, pero personalizable mediante el uso de temas o sobrescribiendo propiedades un un CSS personal. Utilizando las propiedades de Bootstrap, es posible incluir elementos de interfaz<sup>8</sup> (menús, botones, formularios, etcétera) generalmente más vistosos que los que proporciona el propio navegador, así como gestionar atributos como posición de bloques de la página o la alineación del texto.

Además de facilitar el diseño de la página, el uso de Bootstrap genera páginas adaptativas (*responsive*) sin necesidad de introducir demasiados cambios en el código de estas, pudiendo crear de una forma sencilla aplicaciones que se muestren correctamente tanto en escritorio como en dispositivos móviles, reduciendo la carga de trabajo de los diseñadores.

Bootstrap se puede incluir en un proyecto cargando los ficheros de su distribución, mediante una gema de Ruby, o mediante el uso de su CDN<sup>9</sup>, que es la opción por la que se ha optado.



**Figura 5.1:** Pantalla de inicio de sesión de la aplicación, diseñada usando Bootstrap.

<sup>7</sup><https://getbootstrap.com/>

<sup>8</sup><https://getbootstrap.com/docs/4.0/getting-started/introduction/>

<sup>9</sup>Content Delivery Network, red de entrega de contenidos.



---

## 5.4 Presentación de texto: Markdown

---

La mayoría de plataformas de blogs incluyen herramientas para presentar el texto con formato, así como incluir imágenes. Sin embargo, el desarrollo de una galería o la inclusión de un editor de texto WYSIWYG<sup>10</sup> no entran en el alcance de este trabajo. No obstante, se ha decidido integrar el uso de Markdown para poder dar formato al contenido de los *posts*, incluyendo imágenes desde servicios externos, dada su facilidad de implementación, y su uso extendido (se usa Markdown en sitios web como Reddit o Github, por ejemplo) que supone la existencia de una gran cantidad de documentación.

Markdown fue creado en 2004 con la finalidad de permitir que gente sin conocimientos de HTML pudiera redactar fácilmente textos para la web con estilos y que siguiera siendo legible en su formato original. [20]

Dado que es necesario adaptar ciertas partes del código HTML generado a partir de la entrada proporcionada en formato Markdown, para que se adapten a las hojas de estilo que se van a utilizar, se ha elegido utilizar la gema Redcarpet, que permite modificar su generador de HTML de forma sencilla, sin tener que alterar el código base de esta librería.[21]

---

## 5.5 Comentarios: Disqus

---

Es bastante habitual que las entradas de un blog incluyan una sección con comentarios de los lectores. Sin embargo, la creación desde cero de esta sección, incluyendo todas las funcionalidades que se esperan actualmente de ella, puede suponer un gran trabajo, no solo en el desarrollo, sino en el mantenimiento. Por ello se ha decidido usar el servicio de terceros Disqus<sup>11</sup>, que libera de esta carga a los responsables de la aplicación.

Disqus incluye todas las funcionalidades necesarias para la gestión de comentarios en un blog: Identificación de usuarios usando multitud de servicios (Disqus, Facebook, Google, Twitter, etcétera), agrupar comentarios en conversaciones y funciones de moderación para los gestores del blog. Muchos blogs han migrado sus comentarios a Disqus, en algunos casos, sustituyendo la funcionalidad ofrecida por el proveedor, como ocurre con muchos blogs alojados en Tumblr.

Empezar a utilizar Disqus es bastante sencillo, basta con registrarse de forma gratuita, y el servicio ofrece una clave de uso y el código Javascript que se debe incluir en la página para cargar los comentarios. Es posible utilizar Disqus en una aplicación Rails de forma sencilla, mediante la gema «disqus».

---

<sup>10</sup>What You See Is What You Get, lo que ves es lo que obtienes.

<sup>11</sup><https://disqus.com/>

## 5.6 Correo: SendGrid

---

La aplicación debe ser capaz de enviar correo para permitir a los usuarios reiniciar su contraseña en caso de que sea olvidada.

Es posible utilizar un MTA instalado en el servidor donde se ejecuta la aplicación, o habilitar el acceso a aplicaciones menos seguras en proveedores gratuitos como GMail. Sin embargo, esto puede causar que en ocasiones los correos enviados desde la aplicación vayan a la carpeta de correo no deseado, donde es posible que el usuario no los vea. Por ese motivo se ha decidido usar Sendgrid, que ofrece hasta 100 envíos al día de forma gratuita, y sus correos incluyen las cabeceras SPF<sup>12</sup> y DKIM<sup>13</sup> necesarias, evitando en la mayoría de casos que sus envíos se marquen como no deseados.

---

<sup>12</sup>*Sender Policy Framework*, Convenio de Remitentes

<sup>13</sup>*Domain Keys Identified Mail*, Correo Identificado con Claves de Dominio

---

## CAPÍTULO 6

# La aplicación, análisis

---

### 6.1 Análisis de requisitos

---

La aplicación a implementar se basa en una página de publicación de contenidos, centrada en la temática o temáticas del autor, y donde cualquier usuario de Internet puede acceder a leer dichos contenidos y debatir sobre ellos en la sección de comentarios. La aplicación dispondrá de uno o varios usuarios que se encargarán de su gestión, y la publicación de contenidos.

- Cualquier usuario puede acceder a las publicaciones, y participar en las discusiones, salvo que por su comportamiento se le prohíba el uso de la plataforma de comentarios.
- Todas las tareas de gestión se realizarán por un usuario autenticado. Esto incluye creación, modificación y eliminación de *posts*, usuarios y categorías.
- Los usuarios autenticados tendrán privilegios de administración sobre la aplicación, pudiendo gestionar todo el contenido, independientemente del usuario que lo haya creado.

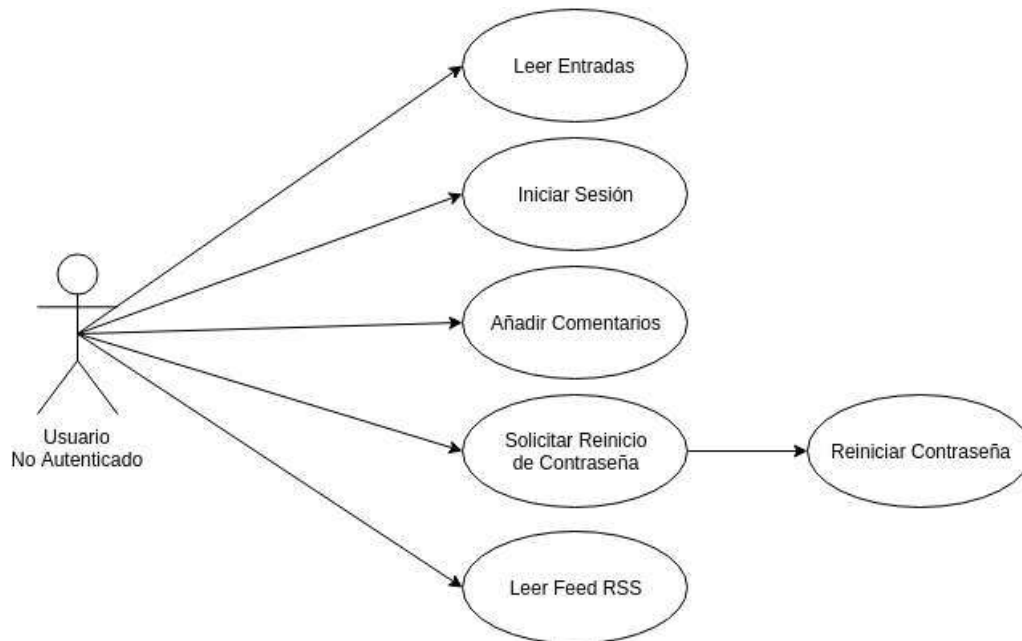
Tanto el contenido, como la información necesaria para su gestión, se almacenan en una base de datos relacional. El control sobre esta se cede parcialmente a la aplicación, siendo posible el acceso manual al SGBD para realizar tareas de mantenimiento, así como de administración del contenido.

Si un usuario que disponga de cuenta en la plataforma olvida su contraseña, puede solicitar que esta sea restablecida, mediante el envío de un correo electrónico con un enlace que le permitirá crear una clave nueva para su cuenta.

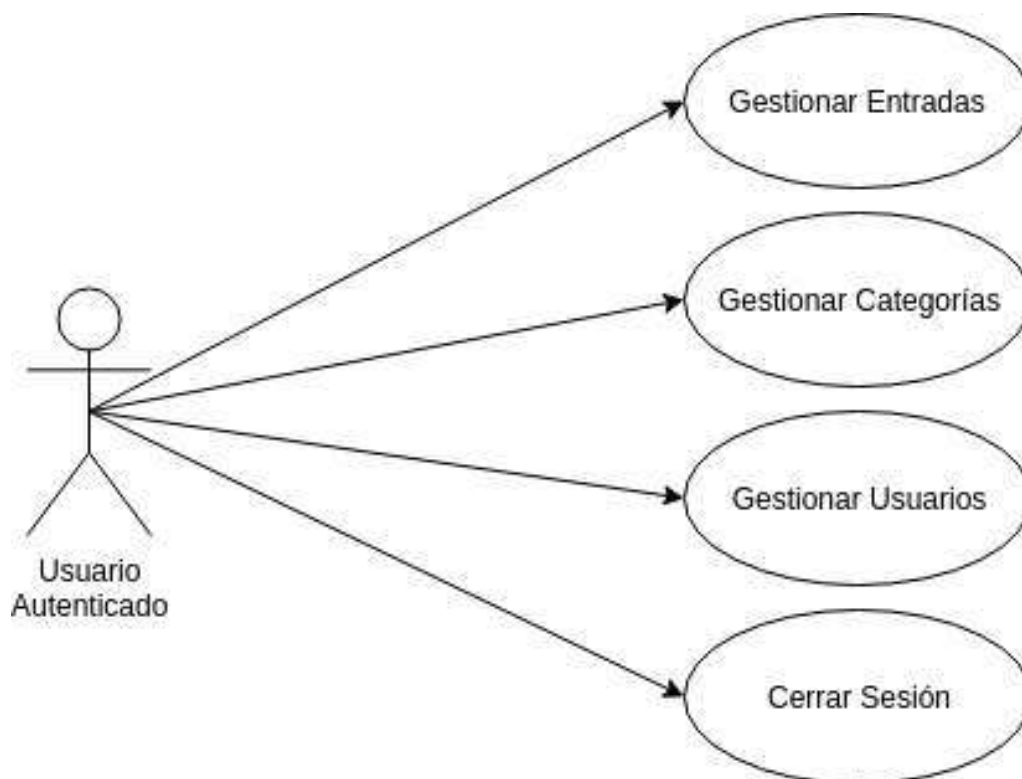
### 6.2 Casos de uso

---

De los requisitos expuestos en el apartado anterior se extraen los siguientes casos de uso



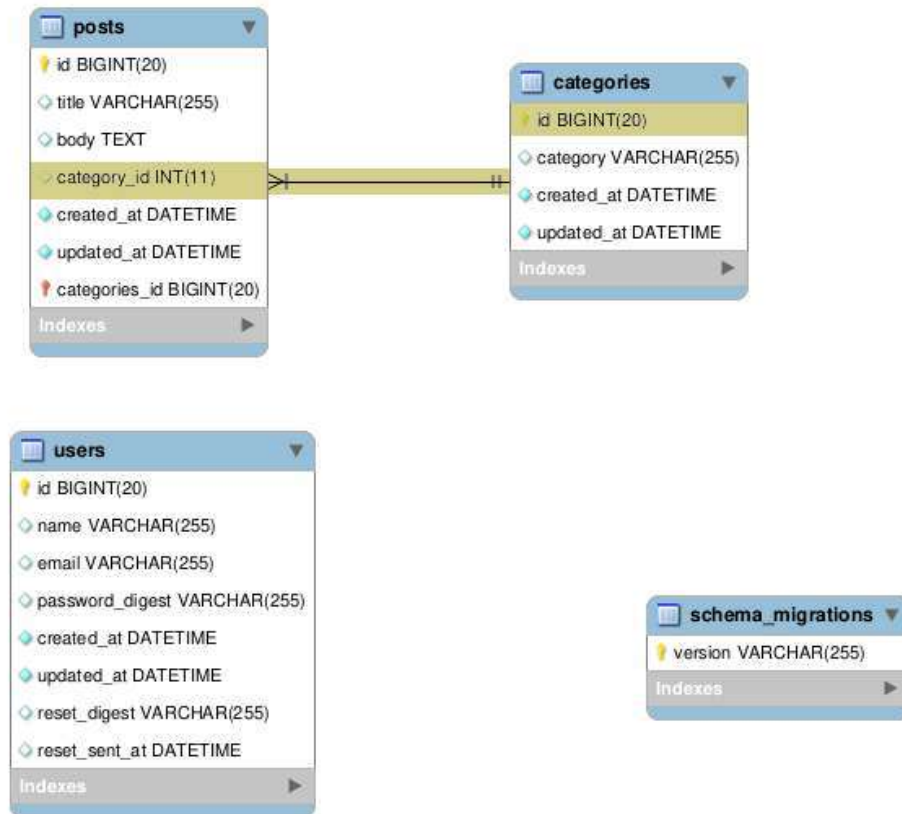
**Figura 6.1:** Casos de uso de usuarios no autenticados.



**Figura 6.2:** Casos de uso de usuarios autenticados.

## 6.3 Modelado de la aplicación

Conforme al análisis de requisitos, el modelado de la base de datos queda de la siguiente manera.



**Figura 6.3:** Base de datos de la aplicación.



---

# CAPÍTULO 7

## La aplicación, desarrollo

---

El presente capítulo describe el procedimiento realizado para la instalación de las herramientas necesarias, así como describe los pasos seguidos a la hora de desarrollar la propia aplicación.

### 7.1 Instalación de las herramientas básicas e inicio

---

En esta sección se describe el procedimiento seguido para instalar las herramientas básicas necesarias para iniciar el desarrollo: Ruby, Rails y MariaDB. Además se muestran los comandos a ejecutar para crear una estructura básica del proyecto.

#### 7.1.1. Instalación

**Nota importante:** El entorno de desarrollo está construido sobre una instalación de Manjaro GNU/Linux, distribución basada en Arch. Por tanto, los comandos que se muestren para instalar paquetes desde los repositorios, deberán ser adaptados a la distribución usada por el lector (*apt* en Debian/Ubuntu, *yum* en Red Hat/Centos, etcétera). No se provee método de instalación para Microsoft Windows u OS X en esta memoria.

En primer lugar instalamos Ruby y la base de datos, MariaDB. No instalamos rails mediante el repositorio, ya que como se menciona en la sección 3.2.1, se instalará mediante gemas de Ruby. Para ello usaremos el siguiente comando:

```
sudo pacman -S ruby mariadb nodejs
```

Tras introducir la contraseña del usuario y aceptar las advertencias de Pacman, los tres paquetes estarán instalados y listos para ser utilizados. La instalación de Node.js se lleva a cabo ya que es uno de los motores de Javascript compatibles con Ruby<sup>1</sup>, necesarios para hacer funcionar el proyecto. Se ha elegido Node.js ya que es posible utilizarlo con otras aplicaciones, y no está limitado a su uso desde Ruby.

---

<sup>1</sup><https://github.com/rails/execjs>

Tras esto, es necesario ejecutar las siguientes órdenes para la configuración inicial de la base de datos.

```
sudo mysql_install_db --user=mysql --basedir=/usr --datadir=/var/lib/mysql
systemctl start mariadb
sudo mysql_secure_installation
```

Estos comandos definen la localización de los ficheros de datos, el usuario del sistema operativo bajo el cual se ejecuta el servicio de MariaDB, se inicia este último y se inicia un asistente para configurar de manera segura la base de datos recién instalada (establece una contraseña para root y prohíbe el acceso remoto con dicha cuenta, elimina usuarios anónimos, y borra las bases de datos de prueba).

A continuación, se debe instalar rails mediante el comando gem, de la forma que se menciona en la sección 2.3. En este caso se omite el parámetro «-v», instalando la versión más reciente.



```
[foleranser@maes ~]$ gem install rails
Fetching: rails-5.1.5.gem (100%)
Successfully installed rails-5.1.5
Parsing documentation for rails-5.1.5
Installing ri documentation for rails-5.1.5
Done installing documentation for rails after 0 seconds
1 gem installed
[foleranser@maes ~]$ rails --version
Rails 5.1.5
[foleranser@maes ~]$
```

Figura 7.1: Instalación de la gema Rails y verificación de su versión.

### 7.1.2. Inicio del proyecto

Después de instalar Rails, se puede crear el nuevo proyecto en el directorio de desarrollo. Para ello basta ejecutar el siguiente comando, que creará la estructura de directorios, instalará las dependencias necesarias y realizará una configuración por defecto básica para empezar a funcionar.

```
rails new junablog -d mysql
```

El parámetro -d permite elegir la base de datos que se va a utilizar, e instala la gema necesaria para su uso. Dado que esta aplicación no hace uso de las extensiones exclusivas de MariaDB, se utiliza la gema «mysql», que es compatible con dicho SGBDR y permitiría la migración de la aplicación a un servidor donde mysql esté disponible pero no MariaDB. Tras ejecutar esto, dispondremos de una aplicación «Hola Mundo!» funcional, que podemos lanzar mediante la siguiente orden desde el directorio del proyecto:

```
rails s
```

El parámetro «s» es una abreviatura para *server*, y se usa para iniciar el servidor HTTP Puma, proporcionado por defecto por Rails. Si no se cambia manualmente esta configuración, Puma aceptará conexiones entrantes a través del puerto



3000. Si accedemos a este puerto desde el navegador, obtendremos la imagen que se muestra a continuación en la figura 7.2.



**Figura 7.2:** Aplicación «Hola Mundo!» generada automáticamente por Rails.

### 7.1.3. Configuración de la base de datos

Por defecto, Rails utiliza SQLite como su base de datos, y no es necesario hacer ningún cambio en la configuración de la aplicación ni en la del SGBDR para poder funcionar. En cambio, para MariaDB, es necesario hacer unos pequeños cambios en ambas.

Dado que el uso de la cuenta «root» no es recomendable, por motivos de seguridad, se va a crear un usuario cuyos privilegios estarán limitados a las bases de datos necesarias para junablog. Para esto, se deben ejecutar las siguientes instrucciones SQL en la consola de MariaDB:

```
CREATE USER junablog IDENTIFIED BY 'junablog';  
GRANT ALL PRIVILEGES ON 'junablog_%'.* TO 'junablog'@'%';
```

La primera instrucción crea el usuario «junablog» con contraseña «junablog»<sup>2</sup>. La siguiente asignará a dicho usuario permisos sobre todas las bases de datos cuyo nombre empiece por «junablog\_», existan o no, esto será suficiente, ya que Rails usará dicho prefijo en todas sus bases de datos.

Acto seguido se debe editar fichero «config/database.yml» para incluir el usuario y la contraseña del SGBDR en todas las entradas de contraseña, y posteriormente se ejecuta el siguiente comando:

```
rake db:create:all
```

Esto iniciará sesión en el MariaDB con el usuario que se ha configurado, y creará todas las bases de datos (desarrollo, pruebas y producción).

---

<sup>2</sup>Es recomendable no utilizar el nombre de usuario como contraseña, especialmente en entornos de producción, pero por simplicidad en el desarrollo, se ha utilizado esta opción.

```
[daniel@Yukihira junablog]$ rake db:create:all
Created database 'junablog_development'
Created database 'junablog_test'
Created database 'junablog_production'
[daniel@Yukihira junablog]$
```

Figura 7.3: Salida del comando de creación de bases de datos

No es necesario crear manualmente las tablas que se van a utilizar, ya que se definirán los datos en el modelo, y posteriormente se usará la herramienta «rake», que lo hará por nosotros.

## 7.2 Definición del modelo inicial

Antes de introducir inicio de sesión a la aplicación, es necesario implementar la funcionalidad más básica, la publicación de *posts*. Para ello, de momento solo necesitaremos definir los modelos de *posts* y categorías.

Para este fin, Rails nos proporciona una herramienta de línea de comandos, que creará el modelo por nosotros, y nos permitirá crear las relaciones entre modelos antes de introducirlos en la base de datos. Para generar los dos modelos y los ficheros necesarios usamos las siguientes instrucciones en la consola.

```
rails generate model Post title:string body:text category_id:integer
rails generate model Category category:string
```

```
[folaransen@maas junablog]$ rails generate model Post title:string body:text cat_id:integer
Running via Spring preloader in process 8812
  invoke  active_record
  create  db/migrate/20180227165744_create_posts.rb
  create  app/models/post.rb
  invoke  test_unit
  create  test/models/post_test.rb
  create  test/fixtures/posts.yml
[folaransen@maas junablog]$ rails generate model Category category:string
Running via Spring preloader in process 8825
  invoke  active_record
  create  db/migrate/20180227165833_create_categories.rb
  create  app/models/category.rb
  invoke  test_unit
  create  test/models/category_test.rb
  create  test/fixtures/categories.yml
[folaransen@maas junablog]$
```

Figura 7.4: Salida de la aplicación de generación de modelos.

Estos comandos generan una serie de ficheros que contienen la definición del modelo, el código fuente base para las pruebas del modelo, y el fichero de migración. Una migración en Rails define los cambios a aplicar sobre la base de datos, pero sin la necesidad de utilizar sentencias SQL. La figura 7.5 presenta el contenido de uno de los ficheros de migración:

Vemos que se crea una tabla llamada «posts», en la cual se introducen los parámetros que hemos pasado a Rails a través del comando «generate», y además se observa que Rails ha introducido un campo nuevo («timestamps»), que crea columnas en la tabla para comprobar cuándo se introdujo y cuándo se modificó por última vez un registro. Con el comando «rake» ejecutamos la migración:

```
rake db:migrate
```

```
1 class CreatePosts < ActiveRecord::Migration[5.1]
2   def change
3     create_table :posts do |t|
4       t.string :title
5       t.text :body
6       t.integer :category_id
7
8       t.timestamps
9     end
10  end
11 end
```

**Figura 7.5:** Contenido de fichero de migración.

```
ActiveRecord::Schema.define(version: 20180227165833) do

  create_table "categories", force: :cascade, options: "ENGINE=InnoDB DEFAULT CHARSET=utf8" do |t|
    t.string "category"
    t.datetime "created_at", null: false
    t.datetime "updated_at", null: false
  end

  create_table "posts", force: :cascade, options: "ENGINE=InnoDB DEFAULT CHARSET=utf8" do |t|
    t.string "title"
    t.text "body"
    t.integer "cat_id"
    t.datetime "created_at", null: false
    t.datetime "updated_at", null: false
  end

end
```

**Figura 7.6:** Contenido de «db/schema.rb» tras ejecutar la migración.

Tras ejecutar «rake», dispondremos de las tablas correspondientes en nuestra base de datos, y se creará el fichero «db/schema.rb», con el contenido que se puede ver en la figura 7.6.

Finalmente introducimos las relaciones y restricciones en los modelos recién creados tal y como vemos en las figuras 7.7 y 7.8

```
1 class Category < ApplicationRecord
2   has_many :posts
3 end
```

**Figura 7.7:** Contenido de «models/category.rb»

```
1 class Post < ApplicationRecord
2   belongs_to :category
3   validates :title, :body, :category_id, presence: true
4 end
```

**Figura 7.8:** Contenido de «models/post.rb»

### 7.2.1. Introducción de datos de prueba

Ruby on rails provee un mecanismo para introducir datos de prueba en el almacén, de modo que no sea necesario introducirlos a mano utilizando SQL. Esto se conoce como *seeds*, o semillas. Introducimos los cambios que se muestran en la figura 7.9, y mediante la herramienta «rake», estos serán procesados e insertados. Por el momento, solo es necesario introducir semillas en la tabla de categorías.

```
1 Category.create(category: 'Personal')
2 Category.create(category: 'CTF Write-ups')
3 Category.create(category: 'Security')
```

Figura 7.9: Contenido de «db/seeds.rb»

Tras introducir dichos cambios, se debe ejecutar la siguiente orden desde el directorio del proyecto, que los trasladará a la base de datos, tal y como se puede comprobar en la figura 7.10.

```
rake db:seed
```

```
MariaDB [junablog_development]> SELECT * from categories;
+----+-----+-----+-----+
| id | category | created_at | updated_at |
+----+-----+-----+-----+
| 1 | Personal | 2018-02-27 17:51:50 | 2018-02-27 17:51:50 |
| 2 | CTF Write-ups | 2018-02-27 17:51:50 | 2018-02-27 17:51:50 |
| 3 | Security | 2018-02-27 17:51:50 | 2018-02-27 17:51:50 |
+----+-----+-----+-----+
3 rows in set (0.00 sec)

MariaDB [junablog_development]>
```

Figura 7.10: Contenido de la tabla «categories» después de cargar las semillas.

## 7.3 Creación de controladores

Tras definir el modelo de datos para las clases Post y Category. Es necesario aportarles funcionalidad. Tal y como se explicó en el apartado 3.3.3, es en el controlador del esquema MVC donde se sitúa la lógica de la aplicación. Al igual, que ocurre con el modelo, Ruby on Rails permite la sencilla creación del esqueleto de los controladores mediante su herramienta de línea de comandos:

```
rails generate controller Posts
```

.

Tras invocar esta instrucción, Rails creará los ficheros necesarios, tal y como se muestra en la figura 7.11, además, creará también el directorio donde se incluirán las vistas asociadas al controlador generado.

```
[foleranser@maes junablog]$ rails generate controller Posts
Running via Spring preloader in process 11381
  create  app/controllers/posts_controller.rb
  invoke  erb
  create  app/views/posts
  invoke  test_unit
  create  test/controllers/posts_controller_test.rb
  invoke  helper
  create  app/helpers/posts_helper.rb
  invoke  test_unit
  invoke  assets
  invoke  coffee
  create  app/assets/javascripts/posts.coffee
  invoke  scss
  create  app/assets/stylesheets/posts.scss
[foleranser@maes junablog]$
```

Figura 7.11: Ficheros creados por «rails generate controller»

Tras tener una base sobre la que construir el controlador, es necesario definir todas las acciones que se utilizarán para este, y que deben satisfacer todas las operaciones CRUD<sup>3</sup>.

- Una acción «index», que listará todos los *posts*.
- La acción «mostrar», que presenta una única entrada del *blog*.
- Acciones «nuevo» y «crear» que permiten redactar y publicar nuevos *posts*, respectivamente.
- Acciones «editar» y «actualizar» que permiten modificar una entrada y publicar los cambios.
- La acción «eliminar».

Estas acciones se incluyen en el fichero «app/controllers/posts\_controller.rb» y para definirlas, solo es necesario crear un método para cada una dentro de la clase del controlador. Más adelante, se podrán utilizar estos métodos como acciones al interactuar con la página. Dada la simplicidad de Rails en el acceso a datos, los métodos pueden ser realmente simples, sobretodo si no deben realizar transformaciones sobre la información. Esto se observa en el método «index», que se muestra en la figura 7.12.

```
1 def index
2   @posts = Post.all
3 end
```

Figura 7.12: Método «index» del controlador Posts

Es posible que varios métodos necesiten realizar la misma acción para cumplir su función. Por ejemplo, los métodos «show» y «destroy»<sup>4</sup>, necesitan buscar el *post* sobre el que realizar su acción. Para ello, Rails permite definir acciones previas

<sup>3</sup>Create, Read, Update and Delete; Crear, leer, actualizar y borrar, las cuatro funciones básicas de la capa de persistencia

<sup>4</sup>«mostrar» y «destruir»

(*before actions*), que contienen un método al que llamar antes de realizar la tarea propia, y opcionalmente, una lista de los métodos que deben invocar la acción previa (si no se provee una lista, por defecto la acción previa se aplicará a todos los métodos):

```
before_action :find_post, only[:show, :edit, :update, :destroy]
```

Este código llamaría al procedimiento «find\_post» antes de ejecutar las acciones «show», «edit», «update» o «destroy». Es posible, además que un método solo necesite realizar su acción previa, pareciendo en el código que el método está vacío, cuando en realidad sí que lleva a cabo una tarea. Es el caso de «show», que solo necesita recuperar un post, y que será la vista quien haga el resto, al tratarse de una acción que solo muestra información, y no la transforma.

## 7.4 Rutas

La creación de un controlador no es suficiente para que el servidor pueda dirigir la petición de un usuario al lugar correcto. Es necesario definir las rutas, donde se especifica cada petición (incluyendo método HTTP y URL) y el método al que se debe asociar. Para ello, en el fichero «config/routes.rb» se incluyen estas definiciones. Se muestra las rutas insertadas en la figura 7.13 y la interpretación que hace Rails de dicha configuración en la figura 7.14

```
1 Rails.application.routes.draw do
2   # For details on the DSL available within this file, see http://
3   # guides.rubyonrails.org/routing.html
4   root 'posts#index'
5   resources :posts
6   end
```

**Figura 7.13:** Rutas para el controlador Posts

Utilizando la instrucción «resources :posts» rails crea por defecto las rutas necesarias para satisfacer todas las operaciones CRUD.

Además, se ha incluido una ruta de forma manual para dirigir el índice a la lista de *posts*, siguiendo la siguiente sintaxis:

```
[MÉTODO HTTP] '/ruta/al/recurso/:parametro', to: 'controlador#método'
```

```
[foleanser@maes junablog]$ rake routes
  Prefix Verb  URI Pattern                      Controller#Action
   root  GET    /                               posts#index
  posts  GET    /posts(.:format)                 posts#index
         POST   /posts(.:format)                 posts#create
  new_post GET    /posts/new(.:format)             posts#new
  edit_post GET    /posts/:id/edit(.:format)        posts#edit
   post  GET    /posts/:id(.:format)             posts#show
         PATCH  /posts/:id(.:format)             posts#update
         PUT    /posts/:id(.:format)             posts#update
         DELETE /posts/:id(.:format)             posts#destroy
[foleanser@maes junablog]$
```

Figura 7.14: Rake mostrando las rutas actuales

## 7.5 Creación de vistas

El último elemento que quedaría por definir para completar las partes del patrón MVC, son las vistas. Un proyecto de Ruby on Rails contendrá al menos una vista, definida en el fichero «app/views/layouts/application.html.erb». Al igual que el resto de vistas, esta usará el formato «.html.erb», que permitirá introducir instrucciones en Ruby en el interior de código HTML. Esta vista por defecto contiene un esqueleto predefinido que usarán el resto de vistas, de modo que no es necesario crear todo el HTML en el resto. Para ello, se incluye la siguiente instrucción, que especifica dónde se debe incluir el código del resto de vistas:

```
<%= yield %>
```

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Junablog</title>
5     <%= csrf_meta_tags %>
6
7     <%= stylesheet_link_tag 'application', media: 'all', 'data-
8       turbolinks-track': 'reload' %>
9     <%= javascript_include_tag 'application', 'data-turbolinks-track':
10       'reload' %>
11   </head>
12
13   <body>
14     <%= yield %>
15   </body>
16 </html>
```

Figura 7.15: Contenido de la vista por defecto «application.html.erb».

Aprovechando este recurso, es posible incluir hojas de estilo y código javascript que vaya a ser común a todas las páginas, y se limita la necesidad de desarrollo de cada vista a solamente su parte única.

### 7.5.1. La gema Summarize

Dado que una entrada del *blog* puede ser arbitrariamente larga, puede ser un problema para la legibilidad y para el consumo de ancho de banda el mostrar los *posts* completos en el índice de la página, donde estarán disponibles todas las entradas publicadas. Para solucionar estos problemas, existen múltiples soluciones. Las más habituales en este tipo de aplicaciones consisten en mostrar únicamente los títulos de las entradas, a modo de enlaces, o mostrar el título y una entradilla o resumen. Para Junablog se ha optado por esta última.

Para no añadir complejidad al desarrollo, en lugar de incluir a mano una entradilla, esta se genera automáticamente mediante Summarize<sup>5</sup>. Esta gema es capaz de resumir automáticamente textos para reducirlos a la extensión deseada. Esta gema dispone de reglas para múltiples lenguajes, pero debido a que el contenido va a ser creado en inglés, este parámetro no se incluirá.

Para su instalación, basta con incluir la gema en el fichero «Gemfile» (siguiendo la sintaxis que se menciona en este), y utilizarla es tan simple como usar la siguiente instrucción.

```
"cadena a resumir".summarize(:ratio => 25)
```

Esto resumiría el texto dejando el resultado en un 25 % de su extensión original. En los próximos apartados se verá su uso.

### 7.5.2. El índice de *posts*

Dado que Rails ha creado una base, solo es necesario incluir el código correspondiente a esta vista en concreto en su fichero. El *framework* incluye objetos que permiten insertar elementos del interfaz ya que generan el código HTML en base a los parámetros que se proporcionen. Tal y como se muestra en la figura 7.16

```
1 <% @posts.each do |post| %>
2   <h1>%= link_to "post.title", post %> </h1>
3   <p>%= post.body.summarize(:ratio => 25) %> </p>
4 <% end %>
```

Figura 7.16: Contenido de la vista por «posts/index.html.erb».

El símbolo «@» que se puede ver en la primera línea indica que se está accediendo a una variable de instancia (es decir, única para un objeto en concreto), en este caso, se refiere a un vector de objetos con la clase Post, que ya fue creado en el método «index» de su controlador. Crear la variable de este modo, permite pasar información desde el controlador hacia la vista.

Ahora mismo, si se inicia el servidor y se apunta el navegador a la aplicación, nos aparecerá una página vacía. Este comportamiento es normal, dado que aún no se ha añadido ningún *post*. A continuación se crearán las vistas que permitirán realizar el resto de acciones, incluyendo la creación de entradas.

<sup>5</sup><https://www.mobomo.com/2010/12/summarize-a-ruby-c-binding-for-open-text-summarizer/>



### 7.5.3. Vistas adicionales para *posts*

Tras crear la primera vista, es necesario crear varias más para cubrir todos los métodos públicos del controlador. Se crearán las siguientes vistas:

- **edit.html.erb**: Incluirá el editor del *post*, así como un enlace para invocar el método «destroy»
- **new.html.erb**: Casi idéntico a la vista anterior, pero utilizado para crear una nueva entrada.
- **show.html.erb**: Mostrará un único *post*.

Dado que las vistas para editar y crear una nueva entrada son casi idénticas, es posible utilizar una funcionalidad de Rails diseñada para estos casos: las vistas parciales. Las vistas parciales funcionan de una forma parecida a la vista «app/views/layouts/application.html.erb», como se ha explicado al principio de la sección 7.5. Basta con crear un fichero separado con el contenido de la vista parcial (cuyo nombre deberá empezar por \_), y en aquellas donde se quiera cargar, se utilizará la instrucción «render», como se puede observar en la figura 7.17.

```
1 <h1>Edit Post</h1>
2 <%= render "form" %>
3 <%= link_to "Back", root_path %>
4 <%= link_to "Delete", post_path, method: :delete, data: { confirm: "
  This will delete the post permanently, please confirm your action."
} %>
```

Figura 7.17: Contenido de la vista «posts/edit.html.erb».

Esto cargará el contenido de la vista parcial, y después mostrará un enlace para volver atrás, y otro para eliminar la entrada.

## 7.6 Gestión de usuarios

La aplicación debe ser capaz de realizar una gestión básica de usuarios, dado que no se debería permitir que cualquiera pueda acceder al panel de administración ni crear entradas. Para ello, se protege las secciones necesarias con usuario y contraseña. Para llevar esto a cabo, será necesario instalar la gema Bcrypt (del mismo modo que se instaló Summarize en el apartado 7.5.1), que se encargará de realizar el cifrado de las contraseñas antes de guardarlas en la base de datos.

Para crear el modelo usuario, esta vez vamos a usar la herramienta de *scaffolding* que ofrece Rails. Esta herramienta crea de forma automática el modelo, la vista y el controlador para una clase, ya poblados con el código básico para operaciones CRUD. Si bien esta funcionalidad puede ser útil, no siempre es adecuada para ciertos proyectos, aunque en este caso, es suficiente. El *scaffolding* se genera con los siguientes comandos:

```
rails generate scaffold User name:string password:digest
rake db:migrate
```

Además se deberá generar un controlador de sesiones y otro para el acceso de administración, en este caso no se mantienen las sesiones en base de datos, por lo que ambos controladores se crearán tal cual, sin modelo.

```
rails generate controller Sessions new create destroy
rake generate controller Admin index.
```

El *scaffolding* habrá creado automáticamente las rutas para la gestión de usuarios. Es necesario incluir manualmente rutas para los controladores «Sessions» y «Admin». Las rutas añadidas se muestran a continuación, en la figura 7.18.

```
1  get '/admin',    to: 'admin#index'
2  get '/login',    to: 'sessions#new'
3  post '/login',   to: 'sessions#create'
4  get '/logout',   to: 'sessions#destroy'
```

**Figura 7.18:** Rutas para los controladores «Sessions» y «Admin».

Para evitar que varios usuarios puedan compartir nombre o dirección de correo (importante dado que más adelante se va a incluir un mecanismo de reinicio de contraseñas olvidadas), y para asegurar que el usuario tiene una contraseña generada de forma segura mediante Bcrypt, se incluyen las siguientes líneas en el fichero «models/user.rb», que ha sido generado por el *scaffolding*.

```
has_secure_password
validates :name, uniqueness: true
validates :email, uniqueness: true
```

Editando las plantillas «.html.erb» correspondientes, se debe generar un formulario de inicio de sesión, y una página de administración. Para facilitar esta tarea, es de incluir el código de la figura 7.19 en «application\_controller.rb», permitiendo que se pueda utilizar en toda la aplicación.

```
1  def current_user
2    @current_user ||= User.find(session[:user_id]) if session[:user_id]
3  end
4  helper_method :current_user
5
6  def authorize
7    redirect_to '/login' unless (current_user or User.all().length ==
8    0)
9  end
```

**Figura 7.19:** Métodos para autorización en «application\_controller.rb».

La comprobación «User.all().length ==0» se ha incluido para no tener que generar un usuario mediante semillas de la base de datos, permitiendo de esa forma

acceder al panel de administración y crear un usuario inicial tras la primera instalación de la aplicación. Tras la creación del primer usuario, la comprobación fallará, y la autorización se basará en la presencia de una sesión con un usuario autenticado. Dado que todos los controladores heredan de la clase «Application-Controller», el método «authorize» está disponible para todas las clases, y eso nos permite utilizarlo como acción previa a métodos que necesiten autorización previa. Por ejemplo, en el controlador de *posts*, nos interesaría prohibir el acceso a las acciones de crear o editar entradas, guardar cambios y eliminar:

```
before_action :authorize, only: [:new, :create, :edit, :update, :destroy]
```

La figura 7.20 muestra el contenido del controlador de Sesiones, que se encarga de autenticar al usuario y redirigir al lugar correspondiente.

```
1  def new
2  end
3
4  def create
5    user = User.find_by_name(params[:name])
6    if user && user.authenticate(params[:password])
7      session[:user_id] = user.id
8      redirect_to '/admin'
9    else
10     redirect_to '/login'
11   end
12 end
13
14 def destroy
15   session[:user_id] = nil
16   redirect_to '/login'
17 end
```

Figura 7.20: Gestión de sesiones en «users\_controller.rb».

Tal y como se ha construido la aplicación, si se elimina un usuario mientras este tiene una sesión iniciada, puede causar errores al acceder a secciones donde haya elementos protegidos. Para ello se incluye el código de la figura 7.21 en el método «destroy» del controlador de usuarios. Este código verifica si el usuario que se va a eliminar («@user») corresponde con el usuario autenticado («@current\_user»), y en ese caso elimina su sesión, así como la variable «@current\_user». De este modo, la navegación vuelve a funcionar normalmente.

```
1  def destroy
2  if (current_user)
3    if (@current_user == @user)
4      session[:user_id] = nil
5      @current_user = nil
6    end
7  end
8  @user.destroy
9  respond_to do |format|
10    format.html { redirect_to users_url, notice: 'User was
11      successfully destroyed.' }
12    format.json { head :no_content }
13  end
end
```

Figura 7.21: Borrado de usuario con cierre de sesión «users\_controller.rb».

## 7.7 Mailer y reinicio de contraseñas

Dado que la aplicación tiene secciones protegidas por usuario y contraseña, existe la posibilidad de que algún usuario olvide o pierda esta última. Por ello, es necesario incluir un mecanismo que permita generar una nueva, de este modo, se elimina la necesidad de utilizar la consola de Rails o el acceso directo a la base de datos para retomar control sobre la cuenta.

### 7.7.1. Mailer

El mecanismo más extendido para recuperación de contraseñas es el envío de un correo con un enlace que permita verificar que el usuario ha solicitado el reinicio. Rails ofrece una herramienta para gestionar envíos de correo, con plantillas generadas dinámicamente. En este trabajo se usa esta herramienta y se configura para utilizar el servicio de envío de correos de SendGrid, para evitar que los *emails* que se envíen terminen en la bandeja de correo no deseado.

Para utilizar SendGrid, basta con registrarse en su web, y solicitar una clave para su API. Por motivos de seguridad, solo es mostrada una vez se genera y no es posible recuperarla (siendo necesario crear una nueva si se pierde la original), por lo que es recomendable guardarla en un lugar seguro.

En primer lugar, generaremos un *mailer* desde la consola de comandos, situados en la raíz del proyecto Rails.

```
rails generate mailer UserMailer password_reset
```

Esto generará los ficheros necesarios donde se deberá configurar el envío.

- «app/mailers/application\_mailer.rb» y «app/mailers/user\_mailer.rb»: Contienen la configuración básica (dirección origen y plantilla a utilizar), así como los métodos que gestionan la creación del correo.

- **Directorio «app/views/user\_mailer»:** Contiene las plantillas en formato HTML y texto claro a utilizar para generar el cuerpo del mensaje.
- **Directorio «app/views/layouts»:** Al igual que se mencionaba en el apartado 7.5, en este directorio se incluyen partes de las plantillas del correo para todos los distintos envíos que hagan los *mailers* de que disponga la aplicación.

Además de rellenar dichos ficheros con el código correspondiente, se ha creado un inicializador («config/initializer/mail.rb») que contiene la información de conexión al servidor de correo, y que se cargará en cada inicio del proyecto. En la figura 7.22 se muestra el código de este inicializador (modificado para no incluir información privada), ya configurado con los detalles que proporciona SendGrid.

```
1 ActionMailer::Base.smtp_settings = {  
2   :address      => 'smtp.sendgrid.net',  
3   :port         => '587',  
4   :authentication => :plain,  
5   :user_name     => 'apikey',  
6   :password      => 'OCULTO - CLAVE DE API',  
7   :enable_starttls_auto => true  
8 }
```

**Figura 7.22:** Inicializador de configuración de conexión al servidor de correo.



**Figura 7.23:** Correo electrónico enviado por el *mailer* mediante SendGrid.

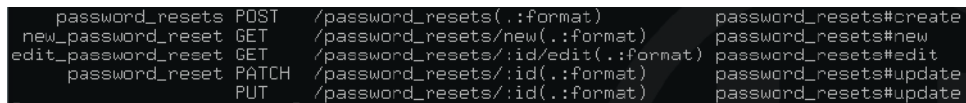
### 7.7.2. Reinicio de contraseñas

Por simplicidad en el desarrollo, solo se habían incluido los datos básicos en el modelo de usuario (nombre, correo y contraseña). Como es necesario comprobar si un usuario ha solicitado un reinicio de contraseña, se ha de modificar el modelo, para incluir una clave de reinicio, y su fecha de generación, que se utilizará para verificar su caducidad. Además habrá que generar un controlador que se encargue de todas las gestiones. Para ello, basta con ejecutar los siguientes comandos, que generarán una migración y la incluirán en la base de datos:

```
rails g migration add_reset_to_users reset_digest:string \  
reset_sent_at:datetime  
rake db:migrate  
rails g controller password_resets new edit
```

Al igual que se ha hecho anteriormente, permitiremos que Rails genere las rutas de forma automática mediante la siguiente línea del fichero «config/routes.rb»:

```
resources :password_resets,      only: [:new, :create, :edit, :update]
```



```
password_resets POST /password_resets(.:format) password_resets#create
new_password_reset GET /password_resets/new(.:format) password_resets#new
edit_password_reset GET /password_resets/:id/edit(.:format) password_resets#edit
password_reset PATCH /password_resets/:id(.:format) password_resets#update
password_reset PUT /password_resets/:id(.:format) password_resets#update
```

**Figura 7.24:** Rutas autogeneradas para el controlador «password\_resets».

Del mismo modo que se hizo en apartados anteriores, construiremos las vistas que correspondan. En este caso se han creado dos vistas:

- **«app/views/password\_resets/new.html.erb»:** Esta vista simplemente contiene un formulario con un campo único en el cual el usuario introducirá su nombre o su correo electrónico, iniciando el proceso de recuperación.
- **«app/views/password\_resets/edit.html.erb»:** Previo a la carga de esta vista, se valida la clave de restablecimiento de contraseña, y se comprueba si esta sigue vigente. En caso de que la solicitud sea válida, al usuario se le presentará con un formulario en el que puede introducir una nueva contraseña.

Tras la creación de las vistas, se incluyen una serie de métodos adicionales en el modelo de Usuario de modo que sean accesibles desde los objetos de la clase User que utiliza el controlador «password\_resets». La funcionalidad de los nuevos métodos se explica a continuación:

- **«password\_reset\_expired?»:** Verifica si la fecha de generación de la clave de reinicio tiene más de dos horas de antigüedad. Esto se usará para verificar la vigencia de la clave.
- **«create\_reset\_digest»:** Genera una clave de reinicio de contraseña, y la guarda en la base de datos junto a la fecha de creación de esta primera. Al igual que para el almacenaje de las contraseñas, este método usa BCrypt para generar los *tokens*.
- **«send\_password\_reset\_email»:** Invoca al *mailer*, y solicita el envío inmediato del correo de restablecimiento de contraseña.

Finalmente, se ha incluido el código necesario en los métodos «create», «edit», «update» y «check\_expiration» del controlador «password\_resets». Seguidamente se presenta la funcionalidad de cada uno:

- **«create»:** Comprueba mediante nombre y/o correo electrónico la existencia de un usuario en la base de datos. En caso de que exista, se solicita la generación de la clave de restablecimiento de contraseña y su envío por *email*. Tanto si existe el usuario o no, se muestra un mensaje genérico, limitando la posibilidad de que un atacante pueda generar una lista de usuarios válidos mediante fuerza bruta.

- **«edit»**: Este método obtiene un usuario mediante su correo electrónico, que se incluye en la URL de reinicio de contraseña, y lo guarda en la variable «@user», que será utilizada posteriormente por el método «update», invocado por el formulario de nueva contraseña.
- **«update»**: Verifica la correctitud de los datos introducidos en el formulario y solicita al modelo la actualización del usuario con la nueva contraseña. En caso de fallo, volverá a cargar el formulario.
- **«check\_expiration»**: Solicita al modelo de Usuarios la comprobación de caducidad de la clave de reinicio, en caso de que esta última ya no sea válida, cargará el formulario para solicitar un nuevo restablecimiento.

## 7.8 Categorías

---

La inmensa mayoría de sistemas de publicación de *blogs* incluyen funcionalizar para organizar sus entradas en categorías, y así poder ver únicamente las que nos resulten interesantes. Para ello, se ha desarrollado un módulo básico que permite su gestión. De un modo similar a como se hizo con el modelo de *posts*, se han creado las estructuras necesarias. Dado que el proceso de desarrollo de este módulo es idéntico al de *posts*, no se va a describir el procedimiento. En su lugar, se enumeran los distintos elementos y su función.

### 7.8.1. Controlador

A continuación se listan los distintos métodos de este controlador, describiendo brevemente su funcionalidad

- **«show»**: Obtiene una lista de todos los *post* de la categoría seleccionada, y los presenta de una forma idéntica al índice de la aplicación, es decir, mostrando los títulos de los *post* y una entradilla generada automáticamente con la gema Summarize.
- **«new»**: Carga la vista con el formulario para crear una nueva categoría.
- **«create»**: Procesa la entrada del usuario en el formulario, y crea una categoría con el nombre seleccionado.
- **«edit»**: Carga la vista con el formulario para modificar una categoría existente.
- **«update»**: Modifica el nombre de la categoría obtenida por el método «edit» con la información proporcionada por el usuario.
- **«destroy»**: Elimina por completo la categoría de la base de datos.

### 7.8.2. Vistas

Se han definido tres vistas completas y una vista parcial (tal y como se hizo para el formulario de creación de *posts* en el apartado 7.5.3).

En primer lugar, la vista «show.html.erb», que muestra todas las entradas de una categoría, tal y como se presentan en el índice de la aplicación. Se incluye un resumen generado automáticamente y un enlace para acceder al *post* completo.

A continuación, las vistas «new.html.erb» y «edit.html.erb», para crear nuevas categorías o renombrarlas, respectivamente. Dado que son prácticamente idénticas (usan el mismo formulario), se ha creado una vista parcial «\_form.html.erb» que contiene el formulario que usarán las anteriores vistas.

## 7.9 Aplicación de estilos con Bootstrap

---

Tal y como se mencionaba en el apartado 5.3, en esta aplicación se ha utilizado el *framework* Bootstrap, creado por Twitter para aplicar estilos a la página de forma sencilla.

Aunque existe una gema para usar Bootstrap desde Rails, se ha optado por usar la versión que se distribuye desde su CDN, que se ofrece de forma gratuita para que desarrolladores web puedan hacer uso de esta herramienta, así como disponer de su versión más reciente sin tener que reemplazar ficheros en sus servidores de forma manual.

Empezar a utilizar bootstrap de este modo es muy simple, basta con editar el fichero «app/views/layouts/application.html.erb», que utilizan todas las vistas de la aplicación (a no ser que se especifique de forma manual, que una vista deba ignorar este fichero por completo). En la web del CDN de Bootstrap<sup>6</sup> se proporciona tanto las URL de los recursos, como el código HTML que se debe incluir, ya preparado para copiar y pegar. La figura 7.25 muestra el código completo del fichero «application.html.erb» tras incluir el código para la carga de Bootstrap.

---

<sup>6</sup><https://www.bootstrapcdn.com/>



```

1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Junablog</title>
5     <%= csrf_meta_tags %>
6     <%= stylesheet_link_tag 'application', media: 'all', 'data-
      turbolinks-track': 'reload' %>
7     <%= javascript_include_tag 'application', 'data-turbolinks-track':
      'reload' %>
8     <link href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/
      bootstrap.min.css" rel="stylesheet" integrity="sha384-
      Gn5384xqQ1aoWXA+058RXPxPg6fy4IWvTNh0E263XmFcJlSAwiGgFAW/
      dAiS6JXm" crossorigin="anonymous">
9   </head>
10
11   <body>
12     <%= yield %>
13
14     <!-- SCRIPTS DE JQUERY, POPPER y BOOTSTRAP, necesarios para el
      funcionamiento de la plantilla -->
15     <script src="https://code.jquery.com/jquery-3.2.1.slim.min.js"
      integrity="sha384-KJ3o2DKtIkVYIK3UENzmM7KCKRr/rE9/
      Qpg6aAZGJwFDMVNA/GpGFF93hXpG5KkN" crossorigin="anonymous"></
      script>
16     <script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js
      /1.12.9/umd/popper.min.js" integrity="sha384-ApNbgh9B+
      Y1QKtv3Rn7W3mgPxhU9K/ScQsAP7hUibX39j7fakFPskvXusvfa0b4Q"
      crossorigin="anonymous"></script>
17     <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/js/
      bootstrap.min.js" integrity="sha384-JZR6Spejh4U02d8jOt6vLEHfe/
      JQGiRRSQQxSfFWpi1MquVdAyjUar5+76PVCmYl" crossorigin="anonymous"
      ></script>
18   </body>
19 </html>

```

**Figura 7.25:** Código de «app/views/layouts/application.html.erb» tras incluir Bootstrap.

Una vez incluido el código necesario, ya es posible utilizar los componentes de interfaz que ofrece Bootstrap<sup>7</sup>, la aplicación de estilos se basa principalmente en añadir clases CSS al atributo «class» del elemento HTML en cuestión. Este atributo se puede modificar tanto directamente desde el HTML, como desde el código ruby empotrado como en el siguiente ejemplo:

```
<%= submit_tag "Submit", class: "btn btn-primary" %>
```

Bootstrap ofrece una implementación de la distribución en rejilla o *grid layout*, basada en un total de doce columnas predefinidas, y filas que serán definidas por el desarrollador, normalmente en base al contenido. Estas doce columnas no se aplican únicamente a la distribución general, sino que cada elemento que pueda mostrar contenido, se dividirá a su vez en otras doce columnas. Esto otorga una gran flexibilidad a la hora de situar las distintas piezas que forman la página. Esto

<sup>7</sup><https://getbootstrap.com/docs/4.0/components/>

se ha aprovechado en este trabajo para incluir una cabecera en la parte superior, y una barra lateral situada a la derecha del contenido, y todo esto centrado, basando los márgenes en la separación por columnas. Se muestra una versión simplificada en la figura 7.26.

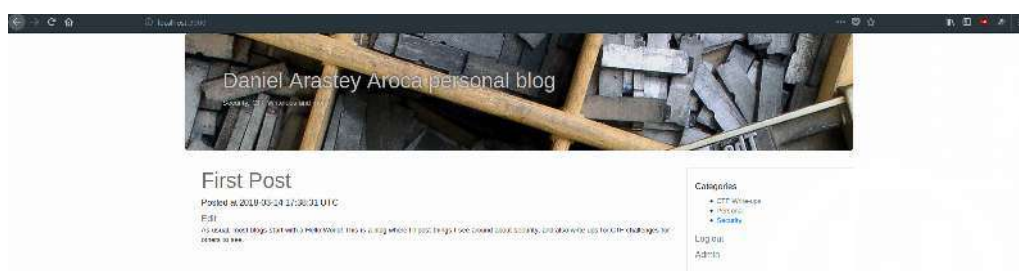
```

1 <div class="container col-md-8 offset-md-2">
2   <!--TITULO (JUMBOTRON)-->
3   <div class="jumbotron juna-header">
4     <div class="container">
5       <!--TITULO-->
6     </div>
7   </div>
8   <!--FIN DEL JUMBOTRON -->
9 </div>
10 <div class="row col-md-8 offset-md-2">
11
12   <div class="container col-9">
13     <!--CONTENIDO-->
14   </div>
15
16 <!--BARRA LATERAL-->
17 <div class="card col-3 ">
18   <!--CONTENIDO DE LA BARRA LATERAL-->
19 </div>
20 <!--FIN DE LA BARRA LATERAL-->
21 </div>
22 </div>

```

**Figura 7.26:** Esqueleto de vista siguiendo la distribución en rejilla mediante Bootstrap.

El uso de las clases «col-...» establece el número de columnas que ocupa cada elemento en su elemento padre. Por ejemplo, la fila tras el *jumbotron* está definida por ocho columnas («col-md-8») y una separación de dos columnas por cada lado («offset-md-2»). Dentro de este bloque de ocho columnas, por tanto, se pueden definir hasta doce nuevas columnas, cuya anchura total ocuparía lo mismo que el bloque padre (ocho columnas sobre el general). En este caso, dentro de dicho bloque se ha definido la sección de contenido (ocupando nueve columnas) y la barra lateral (ocupando tres columnas). En los bloques correspondientes se ha ido incluyendo el código Ruby en línea de las versiones iniciales de cada vista, lo cual hace que la transición de la versión sin estilos a la versión con Bootstrap sea realmente simple. La figura 7.27 muestra cómo queda una página en la que se haya utilizado este esqueleto.



**Figura 7.27:** Índice de *posts* con distribución en rejilla basada en Bootstrap.

Aprovechando el esqueleto mostrado en la figura 7.26 se han diseñado todas las vistas, a excepción del inicio de sesión, que utiliza el diseño mostrado en la figura 5.1, y que se ha reutilizado para el formulario de recuperación de contraseñas.

## 7.10 Presentación de contenido

Para disponer de la capacidad de generar contenido usando texto con formato, se ha decidido usar el lenguaje Markdown, como ya se menciona en la sección 5.4. En Rubygems se encuentran disponibles numerosas librerías en forma de gema que permiten el procesado de Markdown y su presentación en formato HTML. De entre todas las opciones, se ha elegido Redcarpet, dado que permite modificar el código que genera, y que incluye funcionalidad para generar un texto sin estilo, eliminando el marcado utilizado por Markdown.

La instalación es muy sencilla, siguiendo el procedimiento estándar (incluir la gema en el fichero «Gemfile» y ejecutando «bundle install») se dispone de todas las herramientas de Redcarpet a lo largo del proyecto.

Dado que las imágenes que se incluyan deben tener un atributo «class» concreto para que se muestren correctamente en Bootstrap, sin romper el diseño, se ha de crear un procesador de Markdown personalizado. Por defecto, al crearlo se heredan los generadores predefinidos de todos los elementos, por lo que es necesario incluir únicamente aquellos elementos que sea necesario alterar, en este caso solo se han de modificar las imágenes. Dado que el código es sencillo y que solo se va a usar en el controlador de *posts*, la nueva clase se ha incluido en su interior, en lugar de crear un fichero dedicado. La figura 7.28 contiene el código para crear el nuevo generador de HTML, así como el método para su uso, establecido como operación previa a la acción «show».

```
1 class PostsController < ApplicationController
2
3
4   class CustomRender < Redcarpet::Render::HTML
5     def image(link, title, content)
6       %()
7     end
8   end
9
10  ...
11  before_action :initialize_md_renderer, only: [:show]
12  ...
13  private
14  ...
15  def initialize_md_renderer
16    @markdownRenderer = Redcarpet::Markdown.new(CustomRender,
17      autolink: true, tables: true)
18  end
```

**Figura 7.28:** Generador personalizado de HTML mediante Redcarpet y método para su uso.

A continuación se inserta el siguiente código en la vista «app/views/posts/show.html.erb» en el lugar donde deba mostrarse el contenido, y utilizará nuestro generador personalizado, mostrando las imágenes con la clase «img-fluid»:

```
<%= raw @markdownRenderer.render(@post.body) %>
```

## 7.11 Comentarios

Ahora que se dispone de las vistas definitivas de la aplicación, se puede incluir la sección de comentarios, y ajustar la plantilla de modo que esta no se rompa.

En lugar de realizar una implementación propia de comentarios, se ha optado por usar Disqus, dado que su implementación es muy sencilla, y ofrece capacidades avanzadas de moderación desde su panel de control.

En primer lugar es necesario registrarse en la web de Disqus<sup>8</sup>. Tras el registro y la verificación del correo electrónico. Se nos proporciona una serie de opciones para incluir su sistema de gestión de comentarios en distintas plataformas de blogs y publicación web (Wordpress, Tumblr, Blogger, Drupal). Dado que se trata de un proyecto personalizado, no compatible con ninguna de dichas plataformas, utilizamos la opción *universal code*. Esto nos proporciona un código HTML que debemos incluir en la vista correspondiente.

No obstante, por no llenar la vista con un bloque de código que puede hacer más difícil su mantenimiento, se ha incluido dicho código en la vista parcial «app/views/posts/\_disqus». De este modo, en la vista de posts, basta con incluir el siguiente código en el lugar adecuado:

```
<%= render "disqus" %>
```

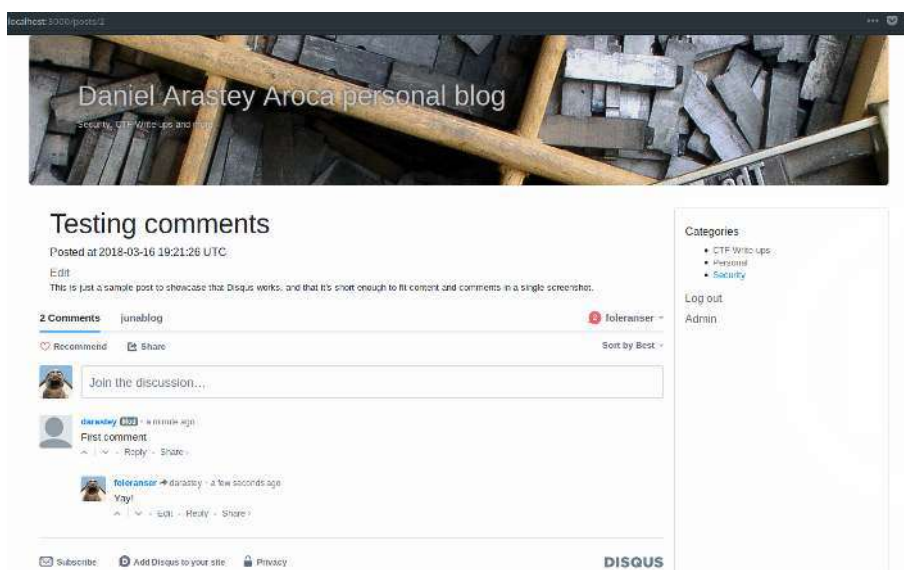


Figura 7.29: Entrada del *blog* con sección de comentarios basada en Disqus.

<sup>8</sup><https://disqus.com>

## 7.12 Configuración

Hasta ahora, todas las variables que pudieran necesitar ser alteradas para futuras instalaciones de Junablog (nombre del blog, servidor y credenciales de correo, etcétera) estaban incluidas directamente en el código de cada uno de los módulos. Esto dificulta el mantenimiento, y vuelve tediosa la realización de cambios que afecten a varias vistas (por ejemplo el nombre y la descripción del blog se muestra en todas las vistas excepto el inicio de sesión y el restablecimiento de contraseñas.

Para ello se pretende extraer esta información a un fichero de configuración en formato YAML (el formato estándar utilizado por aplicaciones Ruby) que será cargado en el arranque de la aplicación.

En este caso se ha creado un fichero YAML («config/junablog.yml») con dos secciones: «blog» y «mailer», que incluyen configuraciones sobre el propio *blog* (título y descripción) y de la conexión al servicio de correo, respectivamente. Esto se muestra a continuación en la figura 7.30.

```
1 blog:
2   name: "Daniel Arastey Aroca personal blog"
3   tagline: "Security , CTF Write-ups and more"
4   host: "http://localhost:3000"
5   owner: "Daniel Arastey Aroca"
6
7 mailer:
8   host: "smtp.sendgrid.net"
9   port: 587
10  user_name: "apikey"
11  password: "OCULTO – CLAVE DE API"
12  link_host: "localhost:3000"
```

**Figura 7.30:** Contenido del fichero «config/junablog.yml».

A continuación se ha creado un fichero en la carpeta «initializers». Cualquier fichero Ruby que se incluya en este directorio se cargará en el arranque de la aplicación. El fichero contiene la directiva para cargar el fichero YAML, y guardar las opciones en una variable que estará disponible en toda la aplicación:

```
JUNABLOG_CONFIG = YAML.load_file("#{Rails.root.to_s}/config/junablog.yml")
```

Esta instrucción permite el acceso a las opciones mediante la sintaxis «JUNABLOG\_CONFIG ["SECCIÓN"] ["OPCIÓN"]». El código que implementaba estas opciones ha sido sustituido en todo el proyecto por sentencias con este formato, como la siguiente, presente en todas las vistas que muestran la cabecera del *blog*:

```
<div class="container">
  <h1 class="blogTitle"><%= JUNABLOG_CONFIG["blog"]["name"] %></h1>
  <p class="blogTitle"><%= JUNABLOG_CONFIG["blog"]["tagline"] %></p>
</div>
```

## 7.13 Feed RSS

Otra funcionalidad típica de la mayoría de plataformas de publicación de *blogs*, son los *feeds* RSS, que se utilizan en lectores de noticias para suscribirse a las publicaciones de una página, y que de forma automática se pueda obtener sus últimas actualizaciones. Este sistema consiste en ficheros XML que contienen enlaces a las entradas del *blog*, así como resúmenes o el cuerpo del *post* completo.

Rails ofrece la funcionalidad necesaria para implementar este método de publicación por defecto, sin necesidad de instalar gemas adicionales.

En primer lugar, se debe incluir el código de la figura 7.31 en el controlador «posts\_controller.rb». Se ha incluido el código de Redcarpet para eliminar las etiquetas de Markdown en la entrada que se ofrece en el *feed*.

```

1 def feed
2   @posts = Post.all
3   @stripdown = Redcarpet::Markdown.new(Redcarpet::Render::StripDown,
4     :space_after_headers => true)
5   respond_to do |format|
6     format.rss { render :layout => false }
7   end
8 end

```

Figura 7.31: Método «feed» en el controlador de *posts*.

A continuación, se generaría la vista «app/views/posts/feed.rssbuilder», con el código que muestra la figura 7.32.

```

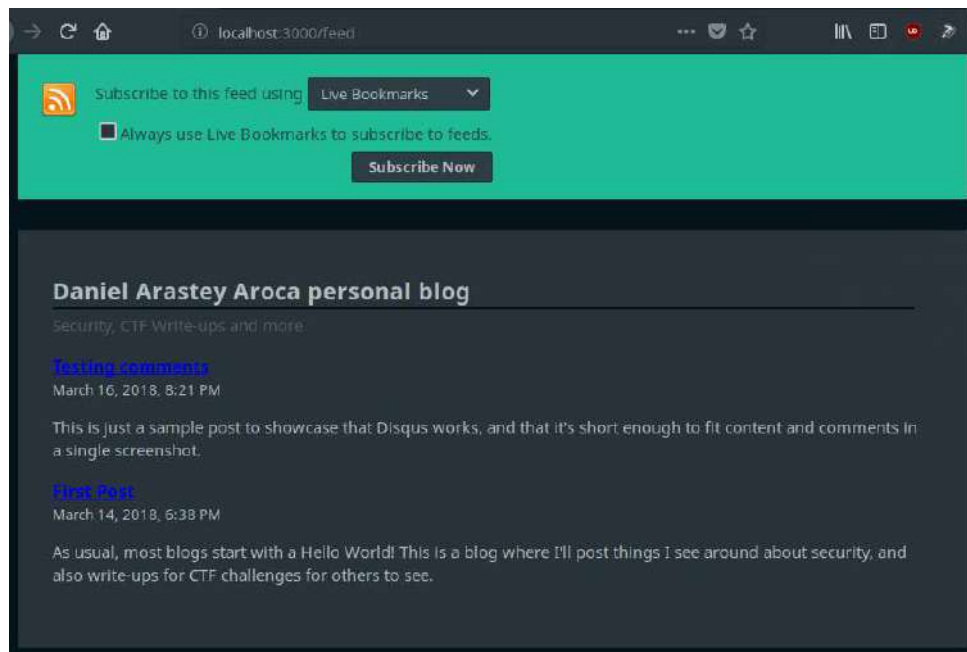
1 #encoding: UTF-8
2 xml.instruct! :xml, :version => "1.0"
3 xml.rss :version => "2.0" do
4   xml.channel do
5     xml.title JUNABLOG_CONFIG["blog"]["name"]
6     xml.author JUNABLOG_CONFIG["blog"]["owner"]
7     xml.description JUNABLOG_CONFIG["blog"]["tagline"]
8     xml.link JUNABLOG_CONFIG["blog"]["host"]
9     xml.language "en"
10    for post in @posts.order(created_at: :desc)
11      xml.item do
12        xml.title post.title
13        xml.author JUNABLOG_CONFIG["blog"]["owner"]
14        xml.pubDate post.created_at.to_s(:rfc822)
15        xml.link JUNABLOG_CONFIG["blog"]["host"]+"/posts/"+post.id.to_s
16        xml.guid post.id
17        text = @stripdown.render(post.body).summarize(:ratio => 25)
18        xml.description "<p>" + @stripdown.render(post.body).summarize
19          (:ratio => 25) + "</p>"
20      end
21    end
22 end

```

Figura 7.32: Generador de XML para el *feed* RSS.

Finalmente, basta con añadir una entrada en el fichero de rutas, y si se desea, un enlace al *feed* en las vistas que muestren contenido.

```
get '/feed', to: 'posts#feed', defaults: { format: 'rss' }
```



**Figura 7.33:** *Feed* RSS en el lector integrado de Firefox.





---

## CAPÍTULO 8

# La aplicación, despliegue

---

En este capítulo se explica los pasos seguidos para hacer que la aplicación esté disponible desde un servidor accesible públicamente en Internet. A diferencia del desarrollo de la aplicación, que se ha llevado a cabo usando una distribución basada en Arch Linux, el despliegue se ha realizado en un servidor basado en Ubuntu Server, lo cual altera ligeramente el proceso de instalación.

### 8.1 Elección del servidor e instalación inicial

---

En primer lugar se ha comparado ofertas de servidores con acceso SSH. Es posible utilizar proveedores de computación en la nube como Amazon Web Services, Microsoft Azure o Heroku. Sin embargo, se ha optado por un servidor privado virtual (VPS). De entre los múltiples proveedores, finalmente se ha optado por Digital Ocean, ya que disponen de una red de reconocida calidad, y la relación características/precio es aceptable para la embergadura de la aplicación y la finalidad de este despliegue.

La configuración elegida dispone de 1 GB de RAM, un núcleo de procesador (frecuencia desconocida), 25 GB de almacenamiento en disco SSD y 1 TB de transferencia mensual, A un precio de 5 euros al mes.

Tras cargar saldo en la cuenta del proveedor, se solicita el despliegue de un servidor con dichas características, escogiendo Ubuntu Server 17.10 como sistema operativo, y estableciendo Londres como localización de la máquina. Una vez seleccionadas sus opciones y establecido un nombre para la máquina, se nos presenta su dirección IP en el panel de control, y se recibe un correo electrónico con la contraseña del usuario root.

Al iniciar sesión por primera vez, deberemos cambiar la contraseña del usuario root. Acto seguido, se debe crear un nuevo usuario (sobre el que se ejecutará la aplicación) y añadirlo al grupo sudo. Esto permitirá que el usuario pueda realizar tareas de administración, pero que el *software* que este ejecute funcione con privilegios reducidos.

```
adduser junablog
usermod -a -G sudo junablog
```

## 8.2 Instalación de dependencias

---

Dadas las diferencias en funcionamiento y contenido de los gestores de paquetes entre la distribución donde se ha llevado a cabo el desarrollo y donde se realiza el despliegue de la aplicación, hay tareas del proceso de instalación que varían enormemente. A continuación se explica cómo se instalan todas las dependencias de la aplicación, desde la versión correcta de Ruby hasta las librerías necesarias para el funcionamiento de las distintas gemas.

En primer lugar es necesario instalar las herramientas básicas de desarrollo. Ubuntu proporciona un metapaquete que las instala por nosotros.

```
sudo apt-get install build-essential
```

**Nota:** Salvo que se indique lo contrario, todos los comandos que se ejecuten en esta sección se lanzan desde el usuario «junablog».

### 8.2.1. Ruby y Rails

La versión de Ruby que se encuentra disponible en los repositorios de Ubuntu 17.10 es la 2.3<sup>1</sup>, que puede poseer incompatibilidades con la versión utilizada durante el desarrollo, 2.5. Por tanto, no es posible utilizar el gestor de paquetes para instalar el lenguaje base. No obstante, para simplificar la instalación existe un *script* de uso muy extendido para este fin, llamado RVM<sup>2</sup>, utilizado para gestionar distintas versiones de Ruby en un mismo sistema.

Basta con ejecutar tres instrucciones en la línea de comandos para instalar RVM y Ruby:

```
gpg --keyserver hkp://keys.gnupg.net --recv-keys \
  409B6B1796C275462A1703113804BB82D39DC0E3 \
  7D2BAF1CF37B13E2069D6956105BD0E739499BDB
curl -sSL https://get.rvm.io | bash -s stable
rvm ruby 2.5.0
```

Al igual que se hizo durante el desarrollo, Rails se instala usando el comando «gem» para evitar problemas de compatibilidad que pueda causar la versión presente en los repositorios.

```
gem install rails
```

### 8.2.2. MariaDB

Aunque la versión disponible en el repositorio de Ubuntu coincide con la del repositorio de Arch Linux, se ha optado por usar el repositorio propio de MariaDB, para instalar la versión estable más reciente y evitar cualquier fallo que pudiera existir en la anterior versión.

---

<sup>1</sup><https://packages.ubuntu.com/artful/ruby/ruby2.3>

<sup>2</sup>Ruby Version Manager

En la web de MariaDB existe una sección que explica cómo añadir sus repositorios a la distribución que estemos utilizando, que nos permite elegir versión y localización del servidor de descarga. Basta con seguir los pasos que se muestran en la web para tener una instalación funcional del SGBD.

```
sudo apt-get install software-properties-common
sudo apt-key adv --recv-keys \
  --keyserver hkp://keyserver.ubuntu.com:80 0xF1656F24C74CD1D8
sudo add-apt-repository 'deb [arch=amd64,i386] \
  http://tedeco.fi.upm.es/mirror/mariadb/repo/10.2/ubuntu artful main'
sudo apt-get update
sudo apt-get install mariadb-server mariadb-client mariadb-common
```

Durante la instalación se nos pedirá que establezcamos una nueva contraseña para el usuario root.

Arrancamos el servicio de MariaDB e iniciamos el cliente, para ejecutar los comandos mostrados en la sección 7.1.3, que crearán un usuario con privilegios limitados en el SGBD. Dado que esta instalación es pública, se recomienda utilizar una contraseña segura para este usuario.

### 8.2.3. Otras dependencias

Existen múltiples librerías de las que debemos disponer para el correcto funcionamiento de las distintas gemas de Ruby que se instalan por nuestra aplicación. Basta con utilizar «apt-get» e instalar las versiones de dichas librerías presentes en el repositorio.

```
sudo apt-get install libmysqlclient-dev libglib2.0-dev libxml2-dev
```

## 8.3 Instalación de la aplicación

---

Dado que durante el desarrollo se ha utilizado git para el control de versiones, es posible utilizarlo para realizar el despliegue de una instalación funcional de Junablog. Para ello basta con utilizar «git clone [URL DEL REPOSITORIO]<sup>3</sup>», desde el directorio *home* del usuario junablog.

Ejecutando el comando que se muestra a continuación se instalan todas las gemas de las que depende la aplicación:

```
bundle install
```

Seguidamente, con el editor de texto que elijamos, modificaremos el fichero «config/database.yml» para incluir las credenciales del SGBD, y acto seguido utilizaremos la herramienta «rake» para crear la base de datos.

---

<sup>3</sup>El repositorio no se ha hecho público a la fecha de redacción de la memoria

```
RAILS_ENV=production rake db:create
RAILS_ENV=production rake db:migrate
```

Tras ejecutar dichos comandos, sería posible lanzar la aplicación, que estaría funcionando por defecto en el puerto 3000 de nuestro servidor.

```
rails s -e production
```

No obstante, detendremos el servidor, dado que nos interesa que la aplicación sea accesible sin especificar puertos en el navegador. Para ello se seguirán los pasos de la siguiente sección.

## 8.4 Despliegue público

Para realizar el despliegue de la aplicación se ha seguido el manual ofrecido por Digital Ocean en su página web, que utiliza NGINX como *proxy*, recibiendo las peticiones y redirigiéndolas a la aplicación. Aunque está enfocado a la versión 14.04 de Ubuntu, es válido para la última versión, y se puede adaptar fácilmente a otras distribuciones.[22] El manual explica la creación de una aplicación básica, estos pasos se pueden omitir y adaptar los que sean relevantes para nuestro proyecto.

En primer lugar, se añadirá la gema «unicorn» al fichero «Gemfile». Unicorn es el servidor web que sustituirá a Puma. Ejecutaremos «bundle install» para instalarlo junto a sus dependencias.

Con el editor de texto de nuestra elección, crearemos el fichero «config/unicorn.rb», con el contenido de la figura 8.1.

```
1 # set path to application
2 app_dir = File.expand_path("../..", __FILE__)
3 shared_dir = "#{app_dir}/shared"
4 working_directory app_dir
5
6
7 # Set unicorn options
8 worker_processes 2
9 preload_app true
10 timeout 30
11
12 # Set up socket location
13 listen "#{shared_dir}/sockets/unicorn.sock", :backlog => 64
14
15 # Logging
16 stderr_path "#{shared_dir}/log/unicorn.stderr.log"
17 stdout_path "#{shared_dir}/log/unicorn.stdout.log"
18
19 # Set master PID location
20 pid "#{shared_dir}/pids/unicorn.pid"
```

Figura 8.1: Fichero «config/unicorn.rb».

Esto hará que al arrancar Unicorn, este no escuche en el puerto 3000, sino que se creará un fichero de *socket* UNIX que se utilizará internamente en el servidor para la comunicación entre NGINX y Unicorn. Para que esto funcione, se deben crear los directorios que se mencionan en el fichero:

```
mkdir -p shared/pids shared/sockets shared/log
```

### 8.4.1. Configuración de arranque automático

Una vez se ha sustituido Puma por Unicorn, y se ha configurado este último para su correcto funcionamiento, se crea la configuración necesaria para que el servidor arranque al iniciar el sistema, de modo que ante un reinicio o caída del servidor, la aplicación vuelva a funcionar automáticamente.

Como administrador, editamos el fichero «/etc/init.d/unicorn\_junablog» y lo rellenamos con el código del manual[22] modificando de manera acorde a nuestra aplicación 8.2.

```
1 #!/bin/sh
2 ...
3 ### END INIT INFO
4 set -e
5 USER="junablog"
6 APP_NAME="junablog"
7 APP_ROOT="/home/$USER/$APP_NAME"
8 ENV="production"
9 PATH="/home/$USER/.rbenv/shims:/home/$USER/.rbenv/bin:$PATH"
10 ...
```

**Figura 8.2:** Script de autoarranque de Unicorn.

Después, se ajustarán los permisos, se configurará para el arranque automático y se iniciará el servicio.

```
sudo chmod 755 /etc/init.d/unicorn_junablog
sudo update-rc.d unicorn_junablog defaults
sudo service unicorn_apname start
```

### 8.4.2. Instalación y configuración de NGINX

NGINX es un servidor web, similar a Apache, con un bajo consumo de recursos pero alta eficiencia. Estas características, junto a la simplicidad de su configuración, lo hacen idóneo para utilizarlo como *proxy*, es decir: NGINX recibe las peticiones, y las transmite a la aplicación. De este modo, únicamente se exponen al exterior los puertos habituales para web, 80 y 443.

**Nota:** Para la realización de esta sección y la siguiente, se ha creado un subdominio (blog.arastey.me) en un registrador de nombres de dominio, antes de llevar a cabo el proceso aquí descrito, se ha configurado una entrada tipo A apuntando a la dirección IP del servidor.

La instalación de NGINX se ha realizado mediante el repositorio de Ubuntu:

```
sudo apt-get install nginx
```

La configuración de NGINX necesaria consiste en editar el fichero «/etc/nginx/sites-available/default», tal y como se muestra en la figura 8.3.

```
1 upstream app {
2     # Path to Unicorn SOCK file , as defined previously
3     server unix:/home/junablog/junablog/shared/sockets/unicorn.sock
4         fail_timeout=0;
5 }
6 server {
7     listen 80;
8     server_name blog.arastey.me;
9
10    root /home/junablog/junablog/public;
11
12    try_files $uri/index.html $uri @app;
13
14    location @app {
15        proxy_pass http://app;
16        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
17        proxy_set_header Host $http_host;
18        proxy_redirect off;
19    }
20
21    error_page 500 502 503 504 /500.html;
22    client_max_body_size 4G;
23    keepalive_timeout 10;
24 }
```

**Figura 8.3:** Configuración de NGINX.

Tras realizar esta configuración, actualizaremos el fichero `junablog.yml` de nuestro proyecto, para reflejar el cambio en el nombre de *host*. Además, modificaremos la configuración del entorno de producción («`config/environments/production.rb`») para permitir servir contenido estático

```
config.public_file_server.enabled = true
```

Y precompilaremos los recursos (imágenes y CSS) para que se ofrezcan como contenido estático:

```
RAILS_ENV rake assets:precompile
```

Al reiniciar el servicio de Unicorn y arrancar el de NGINX, la aplicación estará disponible en <http://blog.arastey.me>.

```
sudo /etc/init.d/unicorn_junablog restart
sudo systemctl start nginx
```

### 8.4.3. Cifrado con Let's Encrypt

La inmensa mayoría de páginas web se ofrecen tanto usando HTTP como usando HTTPS. Dado que se ha utilizado NGINX para presentar la aplicación al exterior, es posible configurar de una manera muy simple el uso de HTTPS, sin modificar la configuración de la aplicación.

Para implementar HTTPS es necesario generar certificados que contengan las claves necesarias para establecer una comunicación cifrada entre el cliente y el servidor. Existen múltiples entidades que emiten estos certificados (Thawte, Verisign, Digicert...), pero su servicio es de pago, ya que incluye soporte y verificación de identidad. Dado que para este proyecto únicamente interesa el cifrado del tráfico, se ha escogido Let's Encrypt como autoridad de certificación.

Let's Encrypt es una autoridad de certificación fundada por miembros de Mozilla, Electronic Frontier Foundation, Cisco Systems, Akamai Technologies y la Universidad de Michigan. Su actividad consiste en ofrecer certificados de forma gratuita, alcanzando un total de cien millones de certificados emitidos en 2017 [23].

Al igual que en el apartado anterior, el proveedor del servidor, Digital Ocean, ofrece una guía de configuración de NGINX con Let's Encrypt. [24]

Primeramente, es necesario añadir el repositorio de «certbot», la aplicación de Let's Encrypt para obtener los certificados, e instalarlo en el sistema.

```
sudo add-apt-repository ppa:certbot/certbot
sudo apt-get update
sudo apt-get install python-certbot-nginx
```

A continuación, se debe verificar que el fichero «/etc/nginx/sites-available/default.conf» contiene el dominio que se va a utilizar en la línea «server\_name», en caso de que no esté configurada correctamente, o utilice todavía el nombre «localhost» basta con ajustarla según nuestro nombre de *host*.

```
server_name blog.arastey.me;
```

Una vez ajustado, podemos reiniciar el servicio de NGINX y solicitar el certificado mediante «certbot».

```
sudo certbot --nginx -d blog.arastey.me
```

Esto iniciará un proceso de validación, en el cual se nos solicitará nuestro correo electrónico, al que enviarán avisos cuando se acerque la fecha de caducidad del certificado, para que lo podamos renovar. Es posible que durante el proceso de validación, se solicite incluir un fichero en el servidor con un contenido específico. En dicho caso, siguiendo las instrucciones específicas para NGINX situaremos el fichero en la ruta correspondiente.

Una vez finalizada la validación, podremos acceder a nuestro servidor usando HTTPS, y se podrá comprobar que el certificado es válido.

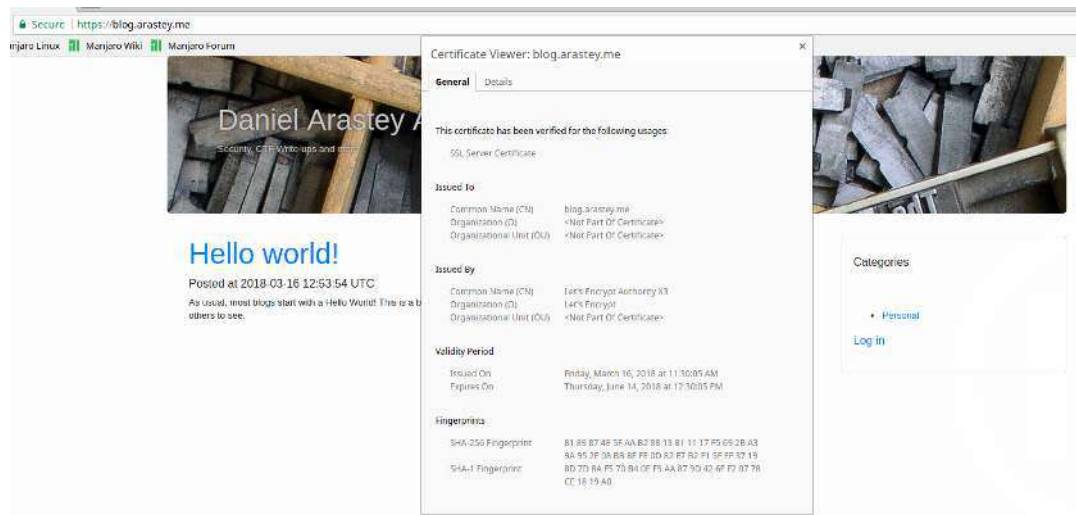


Figura 8.4: Junablog con certificado válido emitido por Let's Encrypt.



---

## CAPÍTULO 9

# Conclusiones

---

Basado en lo que se ha visto durante el desarrollo de este trabajo, se ha podido demostrar la facilidad de desarrollo de aplicaciones basadas en el patrón Modelo Vista Controlador mediante Ruby on Rails. Y aunque no se ha realizado ninguna transformación de datos compleja, la metodología de desarrollo de Rails, deja entrever que incluso en dicho caso es una buena herramienta.

Además del propio desarrollo, ha sido posible comprobar la sencillez con la que se puede integrar servicios y funcionalidad de terceros, sea mediante el uso de gemas (Summarize, Redcarpet) como mediante implementación directa (*mailer*, Disqus). Esto es debido a la gran flexibilidad que ofrece Ruby.

Aunque no sea apto para todas las situaciones, Ruby demuestra por qué es uno de los lenguajes de programación más populares tanto por sí mismo como combinado con Rails, dadas las ventajas que se han mencionado y que se han podido descubrir mediante el desarrollo de este trabajo.

A todo esto, cabe añadir la gran ayuda que ha representado el uso de Git como control de versiones, que ha permitido alternar entre múltiples equipos informáticos para el desarrollo, manteniendo el código sincronizado entre ellos. Además, ha permitido deshacer cambios que han introducido fallos en la aplicación, volviendo a una versión anterior ejecutando un único comando.

A modo de valoración personal, este trabajo ha servido para obtener una «vista previa» del trabajo de un desarrollador web *full-stack*. Se ha aprendido cómo es el desarrollo tanto de *frontend* como de *backend*, así como los mecanismos que un *framework* como Rails ofrece para unirlos y las sinergias entre ellos. Si bien el desarrollo web no es el campo de trabajo al que el autor está enfocado, el desarrollo de esta aplicación hace que conozca lo básico para redirigir su carrera en ese sentido, si llegara a ser necesario.

### 9.1 Futuras mejoras

---

A pesar del trabajo realizado, existen numerosas mejoras que mejorarían enormemente la aplicación, y que en un futuro se irán añadiendo, conforme sean necesarias para cubrir los requisitos del autor. A continuación se listan algunas de las posibles mejoras a implementar:

- **Páginas estáticas:** La inmensa mayoría de *blogs* contienen páginas estáticas con información sobre la propia bitácora, los autores o formas de contacto. Su implementación sería similar a los *posts*.
- **Editor mejorado y subida de imágenes:** Una gran mejora sería la integración de un editor de texto visual, y la inclusión de un sistema de subida de imágenes, para poder alojar estos objetos en nuestro servidor, en lugar de depender de sitios de terceros.
- **Mejor validación de los datos:** Para aumentar la seguridad de la aplicación, comprobando que el formato de los datos que el usuario introduzca sea válido, y que no contenga vectores de ataque.
- **Asignación de roles a usuarios:** Ahora mismo, todos los usuarios de la plataforma disponen de permisos de administración. Sería conveniente implementar RBAC<sup>1</sup>, de modo que se puedan asignar permisos concretos a cada usuario en base a la función que deban cumplir.
- **Mejorar visionado desde dispositivos móviles:** A pesar de que Bootstrap genera una página adaptativa, las vistas que se han creado no han recibido optimizaciones específicas para ser mostradas en móviles o tabletas, por lo que actualmente el diseño solo se ve correctamente en un navegador de escritorio.
- **Mayor separación de código y opciones:** Implementar un mayor uso de variables a cargar desde un fichero YAML, haciendo más sencilla la configuración de la aplicación al instalarla en otro servidor, y evitando la presencia de información privada en el código que se publique.

---

<sup>1</sup>Role Based Access Control, Control de Accesos basado en Roles.

# Bibliografía

---

- [1] Licencia BSDL aplicada al lenguaje de programación Ruby. Consultado en 20 de febrero de 2018. <https://www.ruby-lang.org/en/about/license.txt>.
- [2] Acerca de Ruby. Consultado en 20 de febrero de 2018. <https://www.ruby-lang.org/es/about/>.
- [3] Índice TIOBE de lenguajes de programación. Consultado en 20 de febrero de 2018. <https://www.tiobe.com/tiobe-index/>.
- [4] Sitio web de Rubygems. Consultado en 20 de febrero de 2018. <https://rubygems.org/>.
- [5] Rails 1.0: Party like it's one oh oh! Consultado en 22 de febrero de 2018. <http://weblog.rubyonrails.org/2005/12/13/rails-1-0-party-like-its-one-oh-oh/>.
- [6] The MIT License. Consultado en 22 de febrero de 2018. <https://opensource.org/licenses/MIT> Enlazado en. <http://rubyonrails.org/>
- [7] Dead database walking: MySQL's creator on why the future belongs to MariaDB. Consultado en 22 de febrero de 2018. [https://www.computerworld.com.au/article/457551/dead\\_database\\_walking\\_mysql\\_creator\\_why\\_future\\_belongs\\_mariadb/](https://www.computerworld.com.au/article/457551/dead_database_walking_mysql_creator_why_future_belongs_mariadb/)
- [8] Google swaps out MySQL, moves to MariaDB. Consultado en 22 de febrero de 2018. [https://www.theregister.co.uk/2013/09/12/google\\_mariadb\\_mysql\\_migration/](https://www.theregister.co.uk/2013/09/12/google_mariadb_mysql_migration/)
- [9] Wikipedia Adopts MariaDB. Consultado en 22 de febrero de 2018. <https://blog.wikimedia.org/2013/04/22/wikipedia-adopts-mariadb/>
- [10] MariaDB Server Default in Debian 9. Consultado en 22 de febrero de 2018. <https://mariadb.com/resources/blog/mariadb-server-default-debian-9>
- [11] MariaDB replaces MySQL in repositories. Consultado en 22 de febrero de 2018. <https://www.archlinux.org/news/mariadb-replaces-mysql-in-repositories/>
- [12] Red Hat ditches MySQL, switches to MariaDB. Consultado en 22 de febrero de 2018. <https://www.itwire.com/business-it-news/open-source/60292-red-hat-ditches-mysql-switches-to-mariadb>

- [13] MariaDB server license. Consultado en 22 de febrero de 2018. <https://mariadb.com/kb/en/library/mariadb-license/>
- [14] Roles Overview. Bundling privileges together. Consultado en 22 de febrero de 2018. [https://mariadb.com/kb/en/library/roles\\_overview/](https://mariadb.com/kb/en/library/roles_overview/)
- [15] Query Cache. Consultado en 22 de febrero de 2018. <https://mariadb.com/kb/en/library/query-cache/>
- [16] High Availability with Multi-Source Replication in MariaDB Server. Consultado en 22 de febrero de 2018. <https://mariadb.com/resources/blog/high-availability-multi-source-replication-mariadb-100>
- [17] Secure Connections Overview. Consultado en 22 de febrero de 2018. <https://mariadb.com/kb/en/library/secure-connections-overview/>
- [18] Getting Started - A Short History of Git. Consultado en 23 de febrero de 2018. <https://git-scm.com/book/en/v2/Getting-Started-A-Short-History-of-Git>
- [19] Lista de correo de git - Re: fatal: serious inflate inconsistency. Consultado en 23 de febrero de 2018. <https://marc.info/?l=git&m=118143549107708>
- [20] Daring Fireball - Markdown (Versión archivada en 2004). Consultado en 25 de febrero de 2018. <https://web.archive.org/web/20040402182332/http://daringfireball.net/projects/markdown/>
- [21] Redcarpet is written with sugar, spice and everything nice. Guía de uso de la gema Redcarpet para Ruby. Consultado en 25 de febrero de 2018. <http://www.rubydoc.info/gems/redcarpet/3.4.0>
- [22] How To Deploy a Rails App with Unicorn and Nginx on Ubuntu 14.04. Consultado en 16 de marzo de 2018. <https://www.digitalocean.com/community/tutorials/how-to-deploy-a-rails-app-with-unicorn-and-nginx-on-ubuntu-14-04>
- [23] Milestone: 100 Million Certificates Issued. Consultado en 16 de marzo de 2018. <https://letsencrypt.org/2017/06/28/hundred-million-certs.html>
- [24] How To Secure Nginx with Let's Encrypt on Ubuntu 16.04. Consultado en 16 de marzo de 2018. <https://www.digitalocean.com/community/tutorials/how-to-secure-nginx-with-let-s-encrypt-on-ubuntu-16-04>

---

# APÉNDICE A

## Código fuente

---

En este apéndice se incluye el código fuente de la aplicación. Solo se incluye aquellos ficheros que se hayan creado/modificado.

### A.1 app/controllers

---

#### A.1.1. admin\_controller.rb

```
1 class AdminController < ApplicationController
2   def index
3     @users=User.all
4     redirect_to '/login' unless (current_user or @users.length == 0 )
5     @posts=Post.all
6     @categories=Category.all
7   end
8 end
```

#### A.1.2. application\_controller.rb

```
1 class ApplicationController < ActionController::Base
2   protect_from_forgery with: :exception
3
4   def current_user
5     @current_user ||= User.find(session[:user_id]) if session[:user_id]
6   end
7   helper_method :current_user
8
9   def authorize
10    redirect_to '/login' unless (current_user or User.all().length ==
11      0)
12  end
13 end
```

#### A.1.3. categories\_controller.rb

```
1 class CategoriesController < ApplicationController
2   before_action :authorize, only: [:new, :create, :edit, :update, :
    destroy]
3   before_action :find_category, only: [:show, :edit, :update, :destroy]
4
5   def show
6     @categoryPosts = @category.posts
7     @categories = Category.all
8     @stripdown = Redcarpet::Markdown.new(Redcarpet::Render::StripDown,
        \
9       :space_after_headers => true)
10  end
11
12  def new
13  end
14
15  def create
16    @category=Category.new(:category => params[:name])
17    if @category.save!
18      redirect_to '/admin'
19    else
20      render 'new'
21    end
22  end
23
24  def edit
25  end
26
27  def update
28    name = category_params[:name]
29    @category.update(:category => name)
30    redirect_to '/admin'
31  end
32
33  def destroy
34    @category.destroy
35    redirect_to '/admin'
36  end
37
38  private
39    def find_category
40      @category = Category.find(params[:id])
41    end
42
43    def category_params
44      params.require(:category)
45    end
46  end
47 end
```

#### A.1.4. password\_resets\_controller.rb

```
1 class PasswordResetsController < ApplicationController
2   before_action :check_expiration, only: [:edit, :update]
3
4   def new
```

```

5  end
6
7  def create
8
9      @user = User.find_by(email: params[:password_reset][:mailOrUser])
10     if @user
11         @user.create_reset_digest
12         @user.send_password_reset_email
13     else
14         @user = User.find_by_name(params[:password_reset][:mailOrUser])
15         if @user
16             @user.create_reset_digest
17             @user.send_password_reset_email
18         end
19     end
20     flash[:info] = "Email sent with password reset instructions"
21     redirect_to root_url
22 end
23
24 def update
25     if params[:user][:password].empty?                # Case (3)
26         @user.errors.add(:password, "can't be empty")
27         render 'edit'
28     elsif @user.update_attributes(user_params)        # Case (4)
29         #log_in @user
30         flash[:success] = "Password has been reset."
31         redirect_to login_path
32     else
33         render 'edit'                                # Case (2)
34     end
35 end
36
37 def edit
38     @user = User.find_by(email: params[:email])
39 end
40
41 private
42
43 def user_params
44     params.require(:user).permit(:password, :password_confirmation)
45 end
46
47 def check_expiration
48     @user = User.find_by(email: params[:email])
49     if @user.password_reset_expired?
50         flash[:danger] = "Password reset has expired."
51         redirect_to new_password_reset_url
52     end
53 end
54 end

```

### A.1.5. posts\_controller.rb

```

1  class PostsController < ApplicationController
2
3
4      class CustomRender < Redcarpet::Render::HTML

```

```

5   def image(link, title, content)
6     %()
7   end
8 end
9
10  before_action :authorize, only: [:new, :create, :edit, :update, :
    destroy]
11  before_action :find_post, only: [:show, :edit, :update, :destroy]
12  before_action :initialize_md_renderer, only: [:show]
13
14  def index
15    @posts = Post.all
16    @categories = Category.all
17    @stripdown = Redcarpet::Markdown.new(Redcarpet::Render::StripDown,
    :space_after_headers => true)
18  end
19
20  def show
21    @categories = Category.all
22  end
23
24  def new
25    @post=Post.new
26    if (Category.all().length == 0)
27      Category.new(:category => 'Other').save
28    end
29  end
30
31  def create
32    @post=Post.new(post_params)
33    if @post.save!
34      redirect_to @post
35    else
36      render 'new'
37    end
38  end
39
40  def edit
41  end
42
43  def update
44    if @post.update(post_params)
45      redirect_to @post
46    else
47      render 'edit'
48    end
49  end
50
51  def destroy
52    @post.destroy
53    redirect_to root_path
54  end
55
56  def feed
57    @posts = Post.all
58    @stripdown = Redcarpet::Markdown.new(Redcarpet::Render::StripDown,
    :space_after_headers => true)
59    respond_to do |format|
60      format.rss { render :layout => false }

```



```
61     end
62   end
63
64   private
65
66   def post_params
67     params.require(:post).permit(:title, :body, :category_id)
68   end
69
70   def find_post
71     @post = Post.find(params[:id])
72   end
73
74   def initialize_md_renderer
75     @markdownRenderer = Redcarpet::Markdown.new(CustomRender, \
76       autolink: true, tables: true)
77   end
78
79   end
```

### A.1.6. sessions\_controller

```
1 class SessionsController < ApplicationController
2   def new
3   end
4
5   def create
6     user = User.find_by_name(params[:name])
7     if user && user.authenticate(params[:password])
8       session[:user_id] = user.id
9       redirect_to '/admin'
10    else
11      redirect_to '/login'
12    end
13  end
14
15  def destroy
16    session[:user_id] = nil
17    redirect_to '/login'
18  end
19 end
```

### A.1.7. users\_controller.rb

```
1 class UsersController < ApplicationController
2
3   before_action :authorize
4   before_action :set_user, only: [:edit, :update, :destroy]
5
6   def new
7     @user = User.new
8   end
9
10  def edit
```

```
11 end
12
13 def create
14   @user = User.new(user_params)
15   if @user.save
16     redirect_to admin_path, notice: 'User was successfully created.'
17   else
18     render :new
19   end
20 end
21
22 def update
23   if @user.update(user_params)
24     redirect_to admin_path, notice: 'User was successfully updated.'
25   else
26     render :edit
27   end
28 end
29
30
31 def destroy
32   if (current_user)
33     if (@current_user == @user)
34       session[:user_id] = nil
35       @current_user = nil
36     end
37   end
38   @user.destroy
39   redirect_to admin_path, notice: 'User was successfully destroyed.'
40 end
41
42 private
43
44 def set_user
45   @user = User.find(params[:id])
46 end
47
48 def user_params
49   params.require(:user).permit(:name, :email, \
50     :password, :password_confirmation)
51 end
52
53 end
```

## A.2 app/mailers

### A.2.1. application\_mailer.rb

```
1
2 class ApplicationMailer < ActionMailer::Base
3   default from: 'tfg.junablog@gmail.com'
4   layout 'mailer'
5 end
```

### A.2.2. user\_mailer.rb

```
1 class UserMailer < ApplicationMailer
2   def password_reset(user)
3     @user = user
4     mail to: user.email, subject: "Password reset"
5   end
6 end
```

## A.3 app/models

### A.3.1. category.rb

```
1 class Category < ApplicationRecord
2   has_many :posts
3 end
```

### A.3.2. post.rb

```
1 class Post < ApplicationRecord
2   belongs_to :category
3   validates :title, :body, presence: true
4 end
```

### A.3.3. user.rb

```
1 class User < ApplicationRecord
2   attr_accessor :reset_token
3   has_secure_password
4   validates :name, uniqueness: true
5   validates :email, uniqueness: true
6
7   def send_password_reset_email
8     UserMailer.password_reset(self).deliver_now
9   end
10
11   def create_reset_digest
12     self.reset_token = SecureRandom.urlsafe_base64
13     cost = ActiveSupport::SecurePassword.min_cost ? BCrypt::Engine::
14       MIN_COST : BCrypt::Engine.cost
15     update_attribute(:reset_digest, BCrypt::Password.create(
16       reset_token, cost: cost))
17     update_attribute(:reset_sent_at, Time.zone.now)
18   end
19   def password_reset_expired?
20     reset_sent_at < 2.hours.ago
21   end
22 end
```

## A.4 app/views

Dada la extensión del código y la similaridad entre vistas, se ha publicado el código de estas en el repositorio TODO.

## A.5 config

### A.5.1. initializers/application\_controller\_renderer.rb

```
1 require 'redcarpet/render_strip'
```

### A.5.2. initializers/junablog.erb

```
1 JUNABLOG_CONFIG = YAML.load_file("#{Rails.root.to_s}/config/junablog.
  yml")
```

### A.5.3. initializers/mail.erb

```
1 ActionMailer::Base.smtp_settings = {
2   :address      => JUNABLOG_CONFIG["mailer"]["host"],
3   :port         => JUNABLOG_CONFIG["mailer"]["port"],
4   :authentication => :plain,
5   :user_name    => JUNABLOG_CONFIG["mailer"]["user_name"],
6   :password     => JUNABLOG_CONFIG["mailer"]["password"],
7   :enable_starttls_auto => true
8 }
```

### A.5.4. database.yml

```
1 default: &default
2   adapter: mysql2
3   encoding: utf8
4   pool: <%= ENV.fetch("RAILS_MAX_THREADS") { 5 } %>
5   username: junablog
6   password: EDITADO - CLAVE
7   # socket: /var/run/mysqld/mysqld.sock
8
9 development:
10  <<: *default
11  database: junablog_development
12
13 # Warning: The database defined as "test" will be erased and
14 # re-generated from your development database when you run "rake".
15 # Do not set this db to the same as development or production.
16 test:
17  <<: *default
18  database: junablog_test
19
```

```
20 # As with config/secrets.yml, you never want to store sensitive
    information,
21 # like your database password, in your source code. If your source code
    is
22 # ever seen by anyone, they now have access to your database.
23 #
24 # Instead, provide the password as a unix environment variable when you
    boot
25 # the app. Read http://guides.rubyonrails.org/configuring.html#
    configuring-a-database
26 # for a full rundown on how to provide these environment variables in a
    production deployment.
27 #
28 #
29 # On Heroku and other platform providers, you may have a full
    connection URL
30 # available as an environment variable. For example:
31 #
32 #   DATABASE_URL="mysql2://myuser:mypass@localhost/somedatabase"
33 #
34 # You can use this database configuration with:
35 #
36 #   production:
37 #     url: <%= ENV['DATABASE_URL'] %>
38 #
39 production:
40   <<: *default
41   database: junablog_production
```

### A.5.5. junablog.yml

```
1 blog:
2   name: "Daniel Arastey Aroca personal blog"
3   tagline: "Security, CTF Write-ups and more"
4   host: "http://localhost:3000"
5   owner: "Daniel Arastey Aroca"
6
7 mailer:
8   host: "smtp.sendgrid.net"
9   port: 587
10  user_name: "apikey"
11  password: "EDITADO - CLAVE DE API"
12  link_host: "localhost:3000"
```

### A.5.6. routes.rb

```
1 Rails.application.routes.draw do
2
3   root 'posts#index'
4
5   get '/admin', to: 'admin#index'
6
7   get '/login', to: 'sessions#new'
8   post '/login', to: 'sessions#create'
9   get '/logout', to: 'sessions#destroy'
```

```
10 get '/feed', to: 'posts#feed', defaults: { format: 'rss' }
11
12
13
14 resources :users, only: [:new, :create, :edit, :update, :destroy]
15 # For details on the DSL available within this file, see http://
    guides.rubyonrails.org/routing.html
16
17 resources :posts
18 resources :password_resets, only: [:new, :create, :edit, :update]
19 resources :categories, only: [:new, :edit, :update, :show, :destroy]
20 post '/categories/new', to: 'categories#create'
21
22 end
```

## A.6 db

---

### A.6.1. seeds.rb

```
1 Category.create(category: 'Personal')
2 Category.create(category: 'CTF Write-ups')
3 Category.create(category: 'Security')
```

## A.7 Raíz del proyecto

---

### A.7.1. Gemfile

```
1 source 'https://rubygems.org'
2
3 git_source(:github) do |repo_name|
4   repo_name = "#{repo_name}/#{repo_name}" unless repo_name.include?("/")
5   "https://github.com/#{repo_name}.git"
6 end
7
8
9 # Bundle edge Rails instead: gem 'rails', github: 'rails/rails'
10 gem 'rails', '~> 5.1.5'
11 # Use mysql as the database for Active Record
12 gem 'mysql2', '>= 0.3.18', '< 0.5'
13 # Use Puma as the app server
14 gem 'puma', '~> 3.7'
15 # Use SCSS for stylesheets
16 gem 'sass-rails', '~> 5.0'
17 # Use Uglifier as compressor for JavaScript assets
18 gem 'uglifier', '>= 1.3.0'
19 # See https://github.com/rails/execjs#readme for more supported
    runtimes
20 # gem 'therubyracer', platforms: :ruby
21
22 # Use CoffeeScript for .coffee assets and views
```

```
23 gem 'coffee-rails', '~> 4.2'
24 # Turbolinks makes navigating your web application faster. Read more:
   # https://github.com/turbolinks/turbolinks
25 gem 'turbolinks', '~> 5'
26 # Build JSON APIs with ease. Read more: https://github.com/rails/
   # jbuilder
27 gem 'jbuilder', '~> 2.5'
28 # Use Redis adapter to run Action Cable in production
29 # gem 'redis', '~> 4.0'
30 # Use ActiveModel has_secure_password
31 # gem 'bcrypt', '~> 3.1.7'
32
33 # Use Capistrano for deployment
34 # gem 'capistrano-rails', group: :development
35 gem 'summarize'
36 gem 'bcrypt'
37 gem 'config'
38 gem 'redcarpet'
39 gem 'htmlentities'
40
41
42 group :development, :test do
43   # Call 'byebug' anywhere in the code to stop execution and get a
   # debugger console
44   gem 'byebug', platforms: [:mri, :mingw, :x64_mingw]
45   # Adds support for Capybara system testing and selenium driver
46   gem 'capybara', '~> 2.13'
47   gem 'selenium-webdriver'
48 end
49
50 group :development do
51   # Access an IRB console on exception pages or by using <%= console %>
   # anywhere in the code.
52   gem 'web-console', '>= 3.3.0'
53   gem 'listen', '>= 3.0.5', '< 3.2'
54   # Spring speeds up development by keeping your application running in
   # the background. Read more: https://github.com/rails/spring
55   gem 'spring'
56   gem 'spring-watcher-listen', '~> 2.0.0'
57 end
58
59 # Windows does not include zoneinfo files, so bundle the tzinfo-data
   # gem
60 gem 'tzinfo-data', platforms: [:mingw, :mswin, :x64_mingw, :jruby]
```