



UNIVERSITAT  
POLITÈCNICA  
DE VALÈNCIA



Escola Tècnica  
Superior d'Enginyeria  
Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica  
Universitat Politècnica de València

# **Implementación de un sitio web para un concurso de programación paralela**

**TRABAJO FIN DE GRADO**

Grado en Ingeniería Informática

*Autor:* Nicolás Fernando Martini

*Tutor:* Pedro Alonso Jordá

Curso 2019-2020



# Dedicatoria

---

...



# Agradecimientos

---

*A mi tutor Pedro Alonso Jordá por su apoyo y la oportunidad de hacer este proyecto con la esperanza que produzca un impacto positivo en la enseñanza en los cursos venideros.*

*A la Escola Tècnica Superior d'Enginyeria Informàtica y todos sus profesores, gracias a ellos he podido crecer personal y profesionalmente.*

*Al Ministerio de Educación y Gobierno de España, gracias a sus ayudas he podido acceder los estudios de grado.*

## Resum

...  
...  
...

**Paraules clau:** ?, ?, ?, ?

---

## Resumen

El presente trabajo aborda la implementación de un sitio web para concursos de programación paralela, con el añadido de la gamificación o ludificación, una técnica de aprendizaje que busca recompensar al usuario y aumentar su motivación al sumar elementos y dinámicas propias de los juegos para así ofrecer una experiencia enriquecedora y positiva.

Esta es una tarea compleja, por un lado llevar el proceso de envío de código a un entorno web y la interacción que tendrá con el cluster kahan del DSIC. Y por el otro, transformar las actividades de laboratorio de las asignaturas CPA y LPP con nuevas mecánicas que produzcan al estudiante buscar mejorar sus resultados inclusive después de llegar a una resolución correcta a los ejercicios planteados.

Este proyecto ha sido realizado con el apoyo del DSIC de la UPV con la finalidad de complementar otras herramientas utilizadas hoy en día en la enseñanza.

**Palabras clave:** programación paralela, concurso de programación, gamificación, ludificación

---

## Abstract

...  
...  
...

**Key words:** parallel programming, programming competition, gamification

---

# Índice general

Índice general	VII
Índice de figuras	IX
Índice de tablas	IX
<b>1 Introducción</b>	<b>1</b>
1.1 Motivación . . . . .	1
1.2 Objetivo . . . . .	2
1.3 Impacto esperado . . . . .	2
1.4 Estructura de la memoria . . . . .	3
1.5 Convenciones . . . . .	3
<b>2 Estado del arte</b>	<b>5</b>
2.1 Crítica al estado del arte . . . . .	5
<b>3 Análisis del problema</b>	<b>7</b>
3.1 Actores . . . . .	7
3.2 Historias de usuario . . . . .	8
3.3 Requisitos funcionales . . . . .	9
3.4 Requisitos no funcionales . . . . .	18
3.5 Bocetos . . . . .	19
3.6 Clusters de computadoras . . . . .	21
3.7 Riesgos . . . . .	22
3.8 Análisis del marco legal y ético . . . . .	22
3.9 Plan de Trabajo . . . . .	22
<b>4 Diseño de la solución</b>	<b>25</b>
4.1 Interacción del Sistema . . . . .	25
4.2 Tecnología utilizada . . . . .	26
4.3 Arquitectura del Sistema . . . . .	27
4.4 Modelo de datos . . . . .	28
4.5 Diseño de la Aplicación en Flask . . . . .	29
4.6 Sistema de Colas interno . . . . .	31
<b>5 Desarrollo de la solución</b>	<b>33</b>
5.1 Flask <i>boilerplate</i> . . . . .	33
5.2 Paquetes de Python . . . . .	33
5.2.1 Flask . . . . .	34
5.2.2 Sistemas de Colas . . . . .	34
5.2.3 Conexión con sistemas externos . . . . .	35
5.2.4 Datos de ejemplo . . . . .	35
5.2.5 Otros . . . . .	35
5.3 Envío de soluciones a Tareas . . . . .	35
5.4 Ejecución en una Cola interna . . . . .	37
5.5 Asignación de Insignias . . . . .	41
5.6 Entregables . . . . .	42
<b>6 Implantación</b>	<b>45</b>

6.1	Docker y Kubernetes . . . . .	46
6.2	Despliegue en la Nube con Kubernetes . . . . .	47
6.2.1	Pasos de un despliegue en GCE . . . . .	47
6.2.2	Costos asociados . . . . .	49
<b>7</b>	<b>Pruebas</b>	<b>53</b>
<b>8</b>	<b>Mantenimiento</b>	<b>55</b>
8.1	Entorno local de desarrollo . . . . .	55
8.2	Depuración de errores . . . . .	57
8.3	Calidad de código . . . . .	57
8.4	Gestión de versiones . . . . .	58
<b>9</b>	<b>Extensibilidad</b>	<b>59</b>
9.1	Otros sistemas de Gestión de Tareas . . . . .	59
9.2	Localización . . . . .	61
<b>10</b>	<b>Conclusiones</b>	<b>63</b>
10.1	Relación del trabajo desarrollado con los estudios cursados . . . . .	63
10.2	C . . . . .	63
<b>11</b>	<b>Trabajos futuros</b>	<b>65</b>
	<b>Bibliografía</b>	<b>67</b>
<hr/>		
	Apéndice	
<b>A</b>	<b>Configuración del sistema</b>	<b>69</b>
A.1	Inicialización . . . . .	69
A.2	Parámetros . . . . .	70



## Índice de figuras

---

3.1	Boceto Tarea . . . . .	19
3.2	Boceto Página de inicio . . . . .	20
3.3	Boceto Perfil . . . . .	20
3.4	Diagrama del cluster kahan . . . . .	21
4.1	Modelado Interacción entre elementos del sistema . . . . .	26
4.2	Arquitectura del Sistema . . . . .	27
4.3	Diagrama UML de la Base de Datos . . . . .	30
5.1	AdminLTE: plantilla de Panel de Control . . . . .	34
5.2	Diagrama de flujo de una petición web para un Envío . . . . .	37
6.1	Diferencias entre <i>Docker</i> y máquina virtual . . . . .	45
6.2	nuevo proyecto en GCE . . . . .	48
6.3	Calculadora de precios en GCE . . . . .	50
6.4	Estimación de costos en GCE . . . . .	51
8.1	Inicio de sesión en Pizarra . . . . .	56
9.1	Selector de Idiomas en Pizarra . . . . .	62

## Índice de tablas

---

3.1	RF-1 Registro de usuarios . . . . .	9
3.2	RF-2 Ingreso de usuarios . . . . .	10
3.3	RF-3 Equipos . . . . .	10
3.4	RF-4 Página de Inicio . . . . .	10
3.5	RF-5 Mis Envíos . . . . .	11
3.6	RF-6 Visualización de Envío . . . . .	11
3.7	RF-7 Visualización de Tareas . . . . .	12
3.8	RF-8 Tarea . . . . .	12
3.9	RF-9 Envío . . . . .	13
3.10	RF-10 Tabla de Posiciones . . . . .	13
3.11	RF-11 Perfil . . . . .	14
3.12	RF-12 Perfil . . . . .	14
3.13	RF-13 Configuración del Sistema . . . . .	14
3.14	RF-14 Administración de Envíos . . . . .	15
3.15	RF-15 Administración de Grupos . . . . .	15
3.16	RF-16 Administración de colas internas . . . . .	15

3.17 RF-17 Administración de Alumnos . . . . .	16
3.18 RF-18 Administración de Tareas . . . . .	16
3.19 RF-19 Creación y edición de Tarea . . . . .	17
3.20 RF-20 FAQ . . . . .	17
3.21 RNF-1 Usabilidad . . . . .	18
3.22 RNF-2 Implantación . . . . .	18
3.23 RNF-3 Configuración . . . . .	18
3.24 RNF-4 Localización . . . . .	18
3.25 RNF-5 Almacenamiento . . . . .	18
3.26 RNF-6 Comunicación . . . . .	19

---

# CAPÍTULO 1

## Introducción

---

En este proyecto se trabajarán dos conceptos importantes, la programación paralela y la *gamificación*.

La programación paralela es una rama importante de la computación donde se busca partir problemas de gran magnitud en pedazos más pequeños donde cada partición es ejecutada de forma simultánea por diferentes recursos computacionales de forma coordinada. Este tipo de ejecución tiene varias ventajas como la posibilidad de obtener soluciones a problemas que no pueden ser resueltos en un tiempo razonable, que tienen un orden y complejidad mayor. La programación paralela también trae sus desventajas como una mayor complejidad al escribir los programas, evitar condiciones de carrera y la sincronización entre los recursos.

La gamificación es una técnica o método de aprendizaje que busca potenciar la motivación, esfuerzo e implicación utilizando elementos y estructuras propias de juegos como destacar en posiciones de clasificación, la competencia entre equipos y la obtención de premios. Todo esto para crear una experiencia positiva y dinámica que implique a los usuarios y produzca una asimilación natural de los contenidos.

El objetivo es crear una herramienta de software (aplicación web) que acerque estos “dos mundos” y crear a una experiencia que fomente la colaboración, incentive al alumnado en mejorar sus resultados y afianzar conceptos desde el marco de la competición.

Con la implementación de un sitio web para un concurso de programación paralela y el uso de la gamificación se espera lograr una mejora en la participación de los estudiantes y así llegar a transformar las actividades de laboratorio de Lenguajes y Entornos de la Programación Paralela (LPP), asignatura de la rama de Computación, como también la de Computación Paralela (CPA), asignatura obligatoria del tercer bloque; ambas del Grado de Ingeniería Informática (GII) de la Escola Tècnica Superior d'Enginyeria Informàtica (ETSINF).

### 1.1 Motivación

---

Desde el inicio de mis estudios en el GII me han interesado los concursos de programación, ya sean los promovidos desde la propia ETSINF así como también los disponibles en plataformas online como UVa<sup>1</sup>, HackerRank<sup>2</sup>, CheckIO<sup>3</sup> y Project Euler<sup>4</sup>. Estas competiciones ayudan a promover el interés en diferentes conceptos fundamentales que

---

<sup>1</sup>UVa Online Judge website: <https://onlinejudge.org/>.

<sup>2</sup>HackerRank website: <https://www.hackerrank.com/>.

<sup>3</sup>CheckIO website: <https://checkio.org/>.

<sup>4</sup>Project Euler website: <https://projecteuler.net/>.

debe conocer cualquier profesional de las Ciencias de la Computación, como son las estructuras de datos y la algoritmia, con el añadido de un marco lúdico con elementos de lo que llamamos previamente gamificación.

Normalmente los concursos o competiciones de programación sólo buscan la resolución de una serie de problemas para un conjunto finito de entradas sin importar los tiempos de ejecución o su eficiencia. Por eso al ver la publicación de la propuesta del Trabajo Final de Grado (TFG) donde se añade la programación paralela me ha despertado el interés ya que también se premiará la eficacia del código, esto es, obtener una solución correcta en el menor tiempo posible y además, el añadido de tener que trabajar con un elemento externo como lo es el *cluster* kahan del Departamento de Sistemas Informáticos y Computación (DSIC).

Crear esta aplicación que podrán utilizar los alumnos en cursos venideros es mi principal motivación y forma de devolver a la Universidad las herramientas y formación recibida para lograr convertirme en un profesional de la Informática.

## 1.2 Objetivo

---

Desde mi punto de vista no existe un único objetivo al realizar un TFG, mas allá de que el principal de este sea la implementación de un sitio web para concursos de programación paralela existen otros puntos importantes a considerar; la documentación del proceso realizado, los conocimientos aplicados, lo que se ha aprendido y qué aportes puede brindar a la comunidad científica.

Considero de suma importancia poner de manifiesto los siguientes objetivos secundarios: 1) el seguimiento de un proceso formal para el análisis, diseño, implantación y mantenimiento del software, siendo conceptos previamente vistos en la rama de Ingeniería de Software y esenciales en cualquier proyecto ingenieril; 2) ofrecer una herramienta *open source* abierta a la colaboración, de uso libre y a disposición de otros centros de estudio; 3) explorar nuevas herramientas utilizadas actualmente en el mercado ampliando así mis capacidades como profesional de la Informática.

## 1.3 Impacto esperado

---

La creación de este producto logra una serie de mejoras con respecto a las alternativas actuales de software para competiciones de programación paralela. Como principal característica se añade la posibilidad de despliegue de la aplicación en un servicio en la nube (*cloud*) y el uso de tecnologías actuales en el ámbito del desarrollo de software como lo son *Docker*<sup>5</sup> y *Kubernetes*<sup>6</sup>.

Otro beneficio añadido es la simple implantación por parte del Administrador ofreciendo una interfaz web de configuración y mantenimiento amigable donde en la competencia son inexistentes o muy pobres. Así como también al Alumno se le simplifica la interacción con la aplicación utilizando peticiones web estándar haciendo uso de una API sin la necesidad de software adicional.

En su conjunto la solución completa que se detalla en la memoria, de fácil extensibilidad, localizada en distintos idiomas y abierta mejoras por parte de la comunidad de desarrollo da pie a otros centros de estudio a considerarla como un herramienta adicional a la enseñanza.

---

<sup>5</sup>Sitio web <https://www.docker.com/>

<sup>6</sup>Sitio web Kubernetes: <https://kubernetes.io/>

---

## 1.4 Estructura de la memoria

---

La memoria recoge el proceso de “desarrollo del software” o “ciclo de vida del software” con bloques adicionales que forman parte de un TFG como lo son el Estado del arte o las Conclusiones. Está dividida en los siguientes capítulos, cada uno con sus objetivos.

- **Estado del arte:** análisis de las alternativas que existen al trabajo planteado.
- **Análisis del problema:** recopilación de los requerimientos para así llegar a una propuesta que abarque las necesidades actuales de la asignatura LPP.
- **Diseño de la solución:** descripción de los elementos que forman parte de la solución y el proceso que se ha seguido para la elección de las herramientas utilizadas.
- **Desarrollo de la solución:** enfoque que se ha seguido para el desarrollo del software con sus entregables.
- **Implantación:** enumeración de los pasos a seguir para el despliegue de la solución en una plataforma cloud.
- **Mantenimiento:** explicación de cómo configurar un entorno de desarrollo local, depurar errores y mantener una calidad de código aceptable.
- **Extensibilidad:** disertación acerca de las formas en las cuales se pueden añadir funcionalidades nuevas.
- **Conclusiones:** reflexiones finales sobre el TFG y los objetivos alcanzados.
- **Trabajos futuros:** propuesta de puntos interesantes a tratar en próximas iteraciones o nuevas versiones del software.

---

## 1.5 Convenciones

---

En esta memoria se han seguido una serie de convenciones listadas a continuación:

- las palabras en inglés que aparecen entre paréntesis a continuación de un término hacen referencia a al objeto en *python* que lo representa en el código fuente.
- el código fuente de la aplicación incluido en la memoria tiene en su etiqueta la ubicación del fichero al que hace referencia.
- las palabras en inglés están remarcadas en cursiva.
- los ficheros del código fuente nombrados en párrafos están remarcados en negrita.



---

---

## CAPÍTULO 2

# Estado del arte

---

### 2.1 Crítica al estado del arte

---





---

## CAPÍTULO 3

# Análisis del problema

---

Como en todo proyecto de desarrollo de software debemos escoger un modelo o proceso que garantice el éxito y la calidad del producto final. Siguiendo un modelo tradicional el primer paso es la documentación de requisitos [1] en el cual se realiza un análisis que involucra a los usuarios (Actores) que utilizarán el sistema, cada uno con sus necesidades (Requisitos). De este estudio se llegará a una especificación funcional del software antes de comenzar a escribir siquiera una línea de código. Una vez terminada esta etapa se irán sucediendo las siguientes en próximos capítulos dado el orden habitual de un ciclo de vida clásico [2] o en “cascada”: toma de requisitos, diseño, implementación, pruebas, implantación y mantenimiento.

### 3.1 Actores

---

El sistema contará con dos perfiles de usuarios que llamaremos actores, estos son el Alumno y el Administrador. También veremos que en la memoria se hace mención al “usuario” (*User*), esto se hace para referirnos a funcionalidades que se aplican tanto a Alumnos como Administradores.

- **Alumno:** el actor principal del sistema. Enviará soluciones a los problemas propuestos y tratará de mejorar sus resultados para subir su ranking en las tablas de posiciones (*Leaderboard*). Su trabajo será recompensado en forma de puntos e insignias, que son trofeos o marcas obtenidas después de lograr cierto objetivo. Un alumno pertenecerá a un Grupo (*Group*), que será el curso en el cual está matriculado, y también podrá formar un Equipo (*Team*) con sus compañeros de clase para resolver problemas en conjunto.
- **Administrador:** el actor que se encarga de poner el sistema en marcha. Tendrá que tener conocimientos sólidos en programación paralela para poder crear Tareas que varíen en dificultad y motive al alumnado a mejorar sus resultados. Normalmente será el profesor de la asignatura LPP o CPA.

Antes de comenzar a listar las **Historias de usuario** hay dos conceptos importantes a definir que son los de Tarea (*Assignment*) y Envío (*Request*). La Tarea este es un problema o ejercicio a resolver en la aplicación con una serie de entradas y una salida (resultado) que el Alumno desconoce; la Tarea es creada por el Administrador. El Envío es una petición web de un *usuario* hacia la aplicación web con el código fuente a compilar y ejecutar que tratará de dar solución a una Tarea específica.

Adicionalmente, cuando en la memoria se haga mención a la palabra *job*, no nos referiremos a una Tarea sino al “trabajo” que será procesado en una cola (*queue*) interna o del *cluster* (kahan en nuestro caso).

## 3.2 Historias de usuario

---

El primer paso es realizar una entrevista con los actores involucrados para describir sus necesidades en un lenguaje sencillo que más adelante darán lugar a los requisitos funcionales (RF) y no funcionales de la aplicación (RNF). A esto se le llama **Historias de usuario**.

Para este caso en particular ambos actores fueron interpretados por el tutor y el autor del TFG intercambiando roles al comienzo de la elaboración de la memoria y el resultado es el siguiente:

- Como Alumno, quiero ingresar al sistema utilizando mis credenciales personales.
- Como Alumno, quiero cambiar mi contraseña.
- Como Alumno, quiero poder registrarme en el sistema.
- Como Alumno, quiero ver un listado de los integrantes de mi Grupo.
- Como Alumno, quiero ver un listado de mis Tareas a resolver.
- Como Alumno, quiero ver el listado de posiciones de mi Grupo.
- Como Alumno, quiero ver el listado de posiciones de una Tarea.
- Como Alumno, quiero poder hacer un Envío a una de mis Tareas asignadas.
- Como Alumno, quiero formar parte de un equipo con otros Alumnos de mi Grupo.
- Como Alumno, quiero ver las Insignias que he obtenido.
- Como Alumno, quiero acceder a mi perfil para ver toda mi información.
- Como Alumno, quiero visualizar los resultados de los Envíos.
- Como Alumno, quiero ver un resumen de mi actividad.
- Como Alumno, quiero ver la cantidad de puntos que tengo.
- Como Administrador, quiero crear Grupos.
- Como Administrador, quiero borrar Grupos.
- Como Administrador, quiero asignar Alumnos a Grupos.
- Como Administrador, quiero crear Alumnos subiendo un fichero.
- Como Administrador, quiero borrar Alumnos.
- Como Administrador, quiero restablecer la contraseña de un Alumno.
- Como Administrador, quiero crear Tareas y asignarlas a uno o más Grupos.
- Como Administrador, quiero crear Insignias y asignarlas a una o más Tareas.

- Como Administrador, quiero asignar Insignias a una o más Tareas.
- Como Administrador, quiero editar Tareas.
- Como Administrador, quiero editar Insignias.
- Como Administrador, quiero poder configurar la aplicación.
- Como Administrador, quiero desplegar la aplicación en una solución *cloud*.

### 3.3 Requisitos funcionales

Una vez identificadas las **Historias de Usuario** más importantes el próximo paso es convertirlas en requisitos funcionales formales, específicos y consistentes. Esto evitará ambigüedades y dejará por sentado cómo se comportará el sistema. Aquí se describen todos los requisitos del aplicativo siguiendo una plantilla adaptada de varios modelos consultados en Internet [7]. A la plantilla creada se le ha añadido un identificador único para dar un seguimiento durante la etapa de desarrollo, pruebas y entrega final; la misma tiene los siguientes apartados:

- **Nombre:** título del requerimiento funcional.
- **Descripción:** objetivo que debe cumplir el requisito.
- **Prioridad:** nivel de importancia.
  - Muy Alta.
  - Alta.
  - Media.
  - Baja.
- **Criterio de aceptación:** condiciones que se deben cumplir o satisfacer para que el requisito se considere realizado.

#### RF-1

Nombre	Registro de usuarios
Descripción	El sistema permitirá el registro de usuarios mediante dirección de correo electrónico
Prioridad	Media
Criterio de aceptación	<ul style="list-style-type: none"><li>• no puede existir más de un usuario con el mismo correo electrónico</li><li>• se generará un usuario único con el alias del correo electrónico</li><li>• se podrá deshabilitar el registro de usuarios mediante configuración del sistema</li><li>• se podrá limitar el registro de usuarios a correos electrónicos que pertenezcan a dominios específicos (eg: @upv.es, @inf.upv.es)</li></ul>

Tabla 3.1: RF-1 Registro de usuarios

**RF-2**

Nombre	Ingreso de usuarios
Descripción	El sistema permitirá el ingreso de usuarios mediante dirección de correo electrónico o nombre de usuario
Prioridad	Muy Alta
Criterio de aceptación	solo se podrá ingresar al sistema si la cuenta está activa

**Tabla 3.2:** RF-2 Ingreso de usuarios**RF-3**

Nombre	Equipos
Descripción	Los Alumnos podrán formar parte de un Equipo con otros compañeros de Grupo
Prioridad	Baja
Criterio de aceptación	<ul style="list-style-type: none"> <li>• solo se podrá formar parte de un único Equipo en un momento determinado de tiempo</li> <li>• si un Equipo no tiene integrantes se eliminará del sistema</li> <li>• los Equipos contarán con una URL única que permitirá a otros Alumnos unirse a los mismos</li> <li>• un Alumno no puede unirse un Equipo que no forme parte de su Grupo o que haya llegado al máximo de integrantes</li> <li>• los Equipos tendrán un máximo de integrantes que podrá ser configurado por el Administrador</li> <li>• un Equipo se creará desde el perfil de un Alumno</li> </ul>

**Tabla 3.3:** RF-3 Equipos**RF-4**

Nombre	Página de Inicio
Descripción	Los Alumnos al ingresar verán un resumen agregado de su estado general en la aplicación
Prioridad	Alta
Criterio de aceptación	<p>La Página de Inicio (<i>Dashboard</i>) del Alumno deberá mostrar la siguiente información:</p> <ul style="list-style-type: none"> <li>• resumen de los últimos envíos</li> <li>• extracto calculado de cantidad de envíos, tareas, <i>quota</i> disponible (porcentaje de tiempo restante de ejecución) y puntaje</li> <li>• la última insignia obtenida, en caso que no tuviere, alentarlos a completar una tarea para conseguir su primera</li> <li>• los integrantes del equipo, en caso que no tuviere, alentarlos a crear uno nuevo o unirse a uno existente</li> </ul>

**Tabla 3.4:** RF-4 Página de Inicio

**RF-5**

Nombre	Mis Envíos
Descripción	Los <i>usuarios</i> , tanto Administradores como Alumnos, podrán ver un resumen de sus Envíos (peticiones web que incluyen el código fuente a ejecutar)
Prioridad	Muy Alta
Criterio de aceptación	<p>El resumen de Envíos se visualizará en forma de tabla, se podrá filtrar por la información disponible y deberá mostrar las siguientes columnas:</p> <ul style="list-style-type: none"><li>• ID único en el sistema</li><li>• Fecha y Hora de envío</li><li>• Tarea a la que corresponde</li><li>• Estado del envío</li><li>• Puntaje obtenido</li><li>• Tiempo de ejecución</li></ul>

**Tabla 3.5:** RF-5 Mis Envíos**RF-6**

Nombre	Visualización de Envío
Descripción	Al realizar un Envío, un <i>usuario</i> podrá navegar a una URL única para ver toda la información disponible y hacer el seguimiento de su estado
Prioridad	Alta
Criterio de aceptación	<p>Se deben mostrar los siguientes apartados e información y cumplir con las siguientes restricciones y funcionalidades:</p> <ul style="list-style-type: none"><li>• código fuente</li><li>• análisis estático del código fuente enviado</li><li>• resultado del código ejecutado (<i>output</i> de la consola o terminal)</li><li>• tiempo de ejecución y status</li><li>• cada Envío tiene un ID único que no puede ser repetido en el sistema</li><li>• un Envío solo puede ser visualizado por el autor o un <i>usuario</i> con Rol de Administrador</li></ul>

**Tabla 3.6:** RF-6 Visualización de Envío

**RF-7**

Nombre	Visualización de Tareas
Descripción	Todos los <i>usuarios</i> podrán acceder a un apartado donde se muestre un listado en forma de resumen de las Tareas disponibles
Prioridad	Alta
Criterio de aceptación	Se deben mostrar los siguientes apartados e información: <ul style="list-style-type: none"> <li>• si el usuario es un Alumno se deben mostrar las tareas asignadas a su grupo</li> <li>• si el usuario es un Administrador se deben mostrar todas las tareas en el sistema</li> <li>• por cada Tarea debe haber un resumen, insignias y últimos envíos</li> <li>• no se deben mostrar las Insignias secretas</li> </ul>

**Tabla 3.7:** RF-7 Visualización de Tareas**RF-8**

Nombre	Tarea
Descripción	Visualización de Tarea a completar con su descripción completa
Prioridad	Alta
Criterio de aceptación	Se deben mostrar los siguientes apartados e información: <ul style="list-style-type: none"> <li>• descripción completa</li> <li>• Insignias a obtener</li> <li>• Insignias secretas que el usuario ha obtenido</li> <li>• status de la Tarea: abre pronto, abierto, cierra pronto, cerrada</li> <li>• si el <i>usuario</i> ha realizado un Envío que ha finalizado una ejecución satisfactoria se le debe informar en un mensaje</li> <li>• enlace para hacer un envío con sus credenciales</li> <li>• enlace a la tabla de posiciones</li> </ul>

**Tabla 3.8:** RF-8 Tarea

**RF-9**

Nombre	Envío
Descripción	API disponible a los <i>usuarios</i> para las peticiones web que contienen código fuente
Prioridad	Muy Alta
Criterio de aceptación	<p>El código enviado debe compilarse, ejecutarse y su resultado verificado con la Tarea que tiene asignada. La API creada para el Envío de código fuente debe cumplir con los siguientes apartados:</p> <ul style="list-style-type: none"> <li>• las peticiones deben ser seguras con la utilización de un token único por usuario</li> <li>• la URL de envío debe ser única por Tarea</li> <li>• si el usuario es un Alumno se debe limitar la cantidad de peticiones que puede hacer en una franja de tiempo determinada (segundos)</li> <li>• si el usuario es un Alumno debe verificarse que la Tarea esté asignada a su Grupo y se encuentra abierta para recibir peticiones</li> <li>• si el usuario es un Administrador no debe existir ningún tipo de limitación en las peticiones web</li> <li>• dependiendo el resultado de la ejecución de un Envío se le deben asignar los puntos correspondientes</li> <li>• las respuestas deben ser en formato JSON<sup>1</sup></li> </ul>

**Tabla 3.9:** RF-9 Envío**RF-10**

Nombre	Tabla de Posiciones
Descripción	Página en la cual se muestran las posiciones por Tarea y Grupo en formato de tabla
Prioridad	Alta
Criterio de aceptación	<ul style="list-style-type: none"> <li>• los Alumnos solo pueden visualizar las tablas de posiciones de sus Tareas asignadas y el Grupo al que pertenecen</li> <li>• los Administradores pueden filtrar las tablas de posiciones por Grupos</li> <li>• la posición de un Envío en una Tarea está dada por el tiempo de ejecución, a menor tiempo de ejecución, mejor posicionada</li> <li>• la posición de un Alumno en una tabla de posición de Grupo está dada por la cantidad de puntos que lleva acumulado, mientras más puntos, mejor posición</li> </ul>

**Tabla 3.10:** RF-10 Tabla de Posiciones

**RF-11**

Nombre	Perfil
Descripción	Un <i>usuario</i> del sistema tiene su perfil donde puede ver un resumen de su actividad, cambiar su contraseña, ver al Grupo al que pertenece, cambiar el token de Envío de Tareas y su puntaje acumulado
Prioridad	Media
Criterio de aceptación	Se debe mostrar la siguiente información: <ul style="list-style-type: none"> <li>• apartado con un resumen de actividad con los Envíos realizados y las Insignias obtenidas</li> <li>• apartado para cambiar la contraseña</li> <li>• apartado para generar un nuevo token</li> <li>• apartado para ver los integrantes de su Grupo</li> <li>• apartado de Insignias obtenidas</li> <li>• un resumen de los puntos acumulados y cantidad de Envíos realizados</li> <li>• equipo al que pertenece con sus integrantes</li> </ul>

**Tabla 3.11:** RF-11 Perfil**RF-12**

Nombre	Insignias
Descripción	Trofeo o marca que se le asigna a un Alumno cuando se cumple una condición al procesarse un Envío
Prioridad	Media
Criterio de aceptación	Una Insignia tiene las siguientes restricciones <ul style="list-style-type: none"> <li>• se debe asignar de forma dinámica a través de una regla al terminarse el proceso de un envío.</li> <li>• un Alumno puede obtener una Insignia específica una única vez.</li> <li>• una ejecución fallida (TIMEWALL o ERROR) pueden ser condiciones para asignarlas.</li> </ul>

**Tabla 3.12:** RF-12 Perfil**RF-13**

Nombre	Configuración del Sistema
Descripción	Apartado con el resumen de todos los parámetros de despliegue de la aplicación
Prioridad	Baja
Criterio de aceptación	Se debe mostrar en forma de tabla todos los parámetros de despliegue de la aplicación en formato clave:valor. Esta página solo puede ser accedida por un Administrador

**Tabla 3.13:** RF-13 Configuración del Sistema



**RF-14**

Nombre	Administración de Envíos
Descripción	Página con un listado de todos los Envíos que han sido procesados por el sistema
Prioridad	Alta
Criterio de aceptación	<p>Solo los Administradores pueden acceder a esta página. El listado de Envíos debe contener la siguiente información</p> <ul style="list-style-type: none"> <li>• ID único del sistema con su URL única</li> <li>• fecha y hora de envío</li> <li>• <i>usuario</i> que ha realizado el Envío con su correo y grupo al que pertenece si es un Alumno</li> <li>• Tarea a la cual pertenece el Envío</li> <li>• <i>status</i> del Envío (en espera, encolado, en ejecución, error, etc..)</li> <li>• puntos que se han asignado al Envío</li> <li>• tiempo de ejecución</li> </ul>

**Tabla 3.14:** RF-14 Administración de Envíos**RF-15**

Nombre	Administración de Grupos
Descripción	Página con un listado de los Grupos de Alumnos
Prioridad	Media
Criterio de aceptación	<p>Se debe poder crear nuevos Grupos con un código único y descripción, además de poder eliminarlos. Solo los Administradores pueden acceder a esta página. El listado debe contener la siguiente información</p> <ul style="list-style-type: none"> <li>• código único</li> <li>• descripción</li> <li>• cantidad de Alumnos asignados al Grupo</li> <li>• listado de Tareas asignadas al Grupo</li> </ul>

**Tabla 3.15:** RF-15 Administración de Grupos**RF-16**

Nombre	Administración de colas internas
Descripción	Página con información del estado de las colas internas de la aplicación
Prioridad	Alta
Criterio de aceptación	<p>Solo los Administradores pueden acceder a esta página. Debe mostrarse una resumen con la siguiente información y funcionalidades:</p> <ul style="list-style-type: none"> <li>• colas internas disponibles y su estado</li> <li>• información sobre las <i>jobs</i> en las colas.</li> </ul>

**Tabla 3.16:** RF-16 Administración de colas internas

**RF-17**

Nombre	Administración de Alumnos
Descripción	página con un listado de todos los Alumnos del sistema
Prioridad	Alta
Criterio de aceptación	<p>Solo los Administradores pueden acceder a esta página. Se debe poder borrar Alumnos del sistema y regenerar sus contraseñas. El listado de Alumnos debe contener la siguiente información:</p> <ul style="list-style-type: none"> <li>• nombre de usuario, correo y nombre</li> <li>• grupo al que pertenece</li> <li>• cuota disponible</li> <li>• puntos</li> </ul>

**Tabla 3.17:** RF-17 Administración de Alumnos**RF-18**

Nombre	Administración de Tareas
Descripción	Página con listado de todas las Tareas disponibles en el sistema en formato de tabla
Prioridad	Alta
Criterio de aceptación	<p>Solo los Administradores pueden acceder a esta página. Posibilidad de crear una nueva Tarea. La información que debe mostrar la tabla es la siguiente:</p> <ul style="list-style-type: none"> <li>• ID único</li> <li>• status de la Tarea: abierta o cerrada</li> <li>• nombre</li> <li>• título</li> <li>• cantidad de Envíos</li> <li>• fecha de inicio y fin</li> <li>• Grupos a la cual está asignada</li> <li>• URL para la edición</li> </ul>

**Tabla 3.18:** RF-18 Administración de Tareas

**RF-19**

Nombre	Creación y edición de Tarea
Descripción	Página donde se puede crear una nueva Tarea o editar una ya existente en el sistema con un editor HTML para la descripción de la misma
Prioridad	Muy Alta
Criterio de aceptación	<p>Solo los Administradores pueden acceder a esta página.  Se puede eliminar una Tarea desde la edición.  Se debe poder insertar código fuente para una mejor visualización en la descripción.  Se pueden añadir ficheros adjuntos que serán copiados al cluster kahan junto al código fuente a ejecutar de un Envío.  Los campos disponibles en la creación o edición de una Tarea son los siguientes:</p> <ul style="list-style-type: none"> <li>• nombre único</li> <li>• título</li> <li>• fecha de inicio y fin</li> <li>• puntos a asignar en caso de un resultado satisfactorio</li> <li>• editor texto para la descripción de la Tarea en formato HTML</li> <li>• resultado esperado de la ejecución</li> <li>• listado de Grupos a asignar la tarea</li> <li>• listado de Insignias que se pueden obtener en la Tarea</li> <li>• ficheros adjuntos</li> </ul>

**Tabla 3.19:** RF-19 Creación y edición de Tarea**RF-20**

Nombre	FAQ
Descripción	Página con preguntas frecuentes que se pueden hacer los usuarios del sistema y sus respuestas
Prioridad	Baja
Criterio de aceptación	<ul style="list-style-type: none"> <li>• debe estar localizado en al menos 2 idiomas</li> <li>• no debe contener más de 10 preguntas</li> </ul>

**Tabla 3.20:** RF-20 FAQ

### 3.4 Requisitos no funcionales

Los requisitos no funcionales son criterios que se deben cumplir para juzgar la correcta operación del sistema; en contraste con los requisitos funcionales no definen comportamientos específicos.

Para las siguientes tablas también se ha creado una plantilla estándar similar a la de requisitos funcionales pero con menos atributos y la definición de los mismos es la misma de las tablas vistas anteriormente.

#### RNF-1

Nombre	Usabilidad
Descripción	El sitio web deberá tener una interfaz sencilla y fácil de utilizar.
Prioridad	Muy Alta

**Tabla 3.21:** RNF-1 Usabilidad

#### RNF-2

Nombre	Implantación
Descripción	El aplicativo debe poder implantarse de forma automatizada
Prioridad	Media

**Tabla 3.22:** RNF-2 Implantación

#### RNF-3

Nombre	Configuración
Descripción	El aplicativo debe permitir configurar los parámetros de despliegue y uso
Prioridad	Alta

**Tabla 3.23:** RNF-3 Configuración

#### RNF-4

Nombre	Localización
Descripción	El aplicativo debe soportar la localización en diferentes idiomas
Prioridad	Media

**Tabla 3.24:** RNF-4 Localización

#### RNF-5

Nombre	Almacenamiento
Descripción	Toda la información de la aplicación debe ser persistida en una Base de datos (BD) y el código fuente almacenado en un sistema de ficheros
Prioridad	Muy alta

**Tabla 3.25:** RNF-5 Almacenamiento

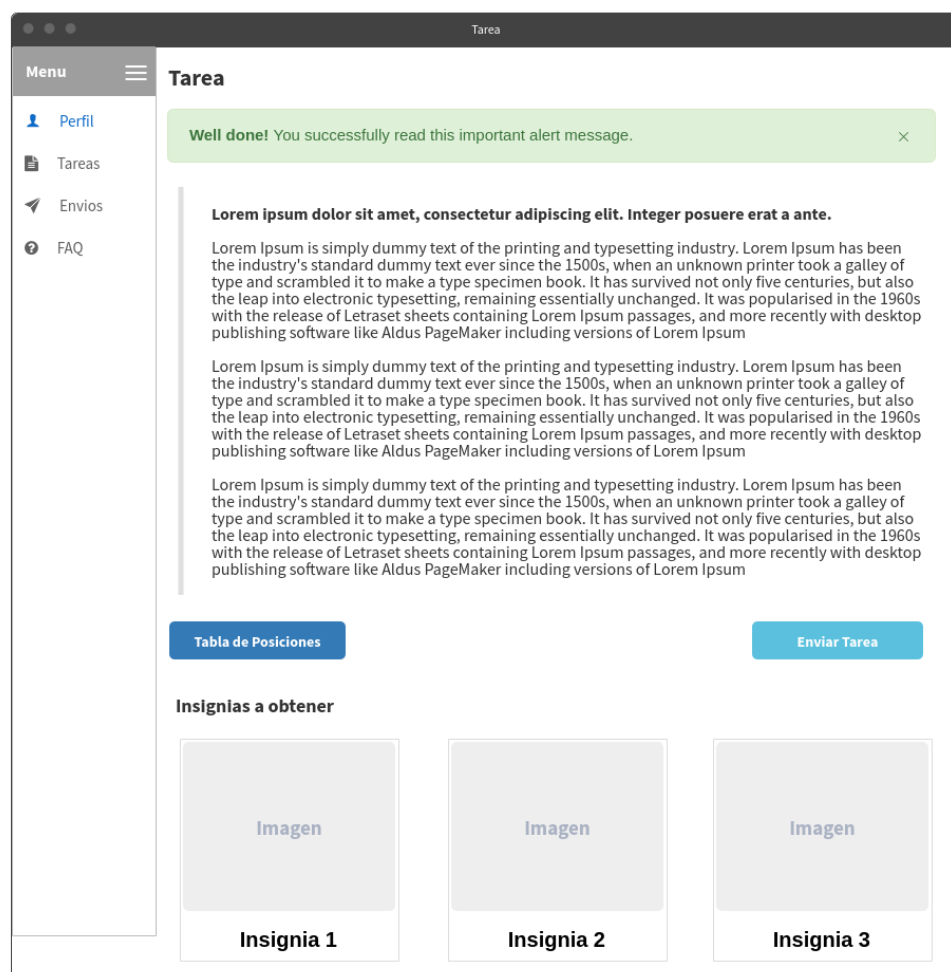
**RNF-6**

Nombre	Comunicación
Descripción	Todas las comunicaciones con sistemas externos deben realizarse con protocolos seguros.
Prioridad	Muy alta

**Tabla 3.26:** RNF-6 Comunicación**3.5 Bocetos**

Como parte del análisis del problema se han realizado bocetos de la interfaz web de algunos de los requisitos funcionales que tienen una gran cantidad de información a mostrar. Estos bocetos se llaman *wireframes* y representan de forma sencilla la estructura de una página web. No se espera que muestren el diseño final o que sean perfectos estéticamente ya que su cometido es aportar claridad, detectar problemas de usabilidad y crear una conexión entre los requisitos e información con la interfaz.

Los bocetos corresponden a los requisitos funcionales Página de inicio (RF-4) para la Figura 3.2, Tarea (RF-8) para la Figura 3.1 y Perfil (RF-11) para la Figura 3.3.

**Figura 3.1:** Boceto Tarea (RF-8)

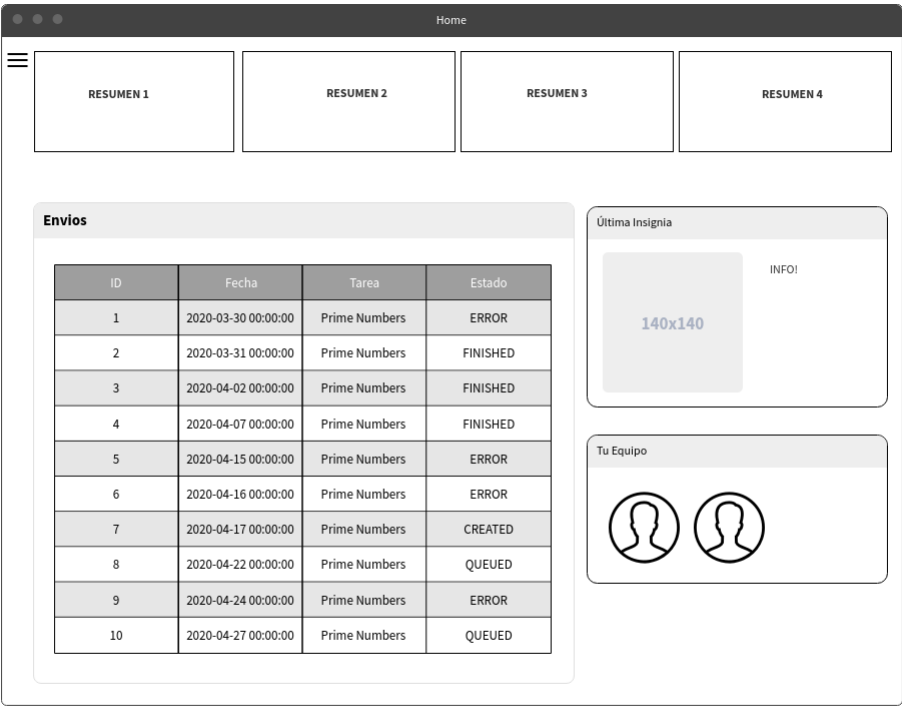


Figura 3.2: Boceto Página de inicio (RF-4)

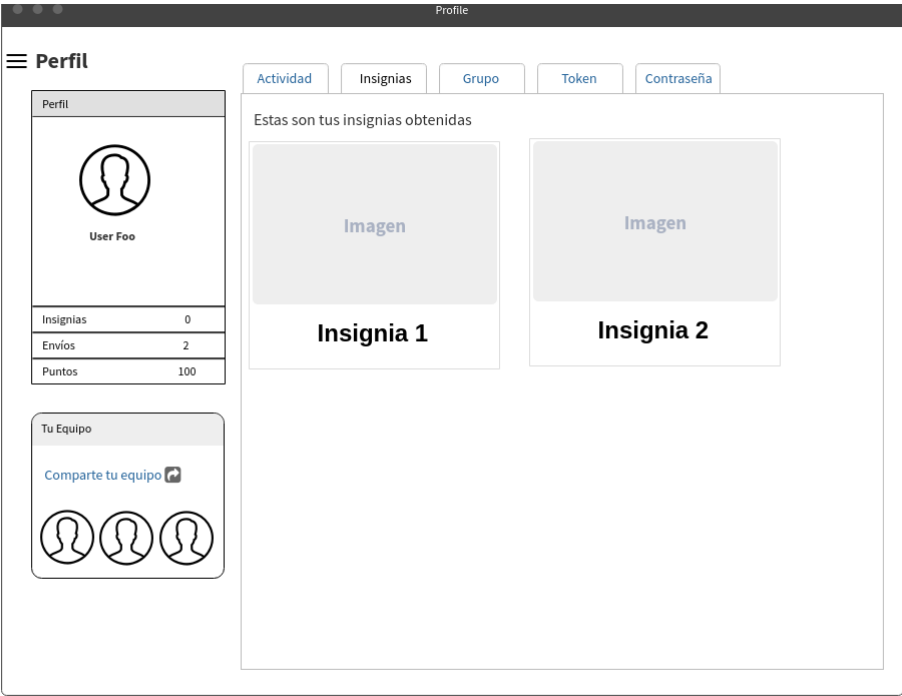


Figura 3.3: Boceto Perfil (RF-11)

## 3.6 Clusters de computadoras

El objeto de la aplicación es el de ejecutar trabajos (*jobs*) en un *cluster* de computadoras. Estas configuraciones están formadas por nodos de computación unidos entre sí mediante redes de alta velocidad. Su programación está basada en el paradigma conocido como *programación en memoria distribuida* donde todos los nodos ejecutan el mismo código, particularizado por un identificador de proceso (valor de 0 a  $p-1$ , siendo  $p$  el número de nodos) y se intercambian datos entre sí mediante paso explícito de mensajes. La programación de este tipo de configuraciones suele realizarse utilizando MPI (Message Passing Interface), una librería que permite la programación de estos equipos siguiendo el paradigma mencionado. Es usual que estos equipos dispongan de un nodo principal llamado *Frontend*, al cuál se conectan los usuarios mediante una conexión ssh para desarrollar código y lanzar el programa que se ejecutará en los nodos de cómputo.

En este trabajo utilizaremos un *cluster* de computadores particular, como ejemplo de uso y representante de este tipo de configuraciones, denominado kahan. El *cluster* kahan es un *cluster* de computación perteneciente a la UPV [8] y será el encargado de procesar el código que envíen los alumnos a la aplicación web. Consta de un nodo cabecera (*Frontend*) a la que un usuario se puede conectar mediante escritorio remoto o consola y así enviar el trabajo a procesar en alguna de sus colas. El nodo *Frontend* es el encargado de comunicarse con el *cluster* de 6 nodos y procesar los envíos.

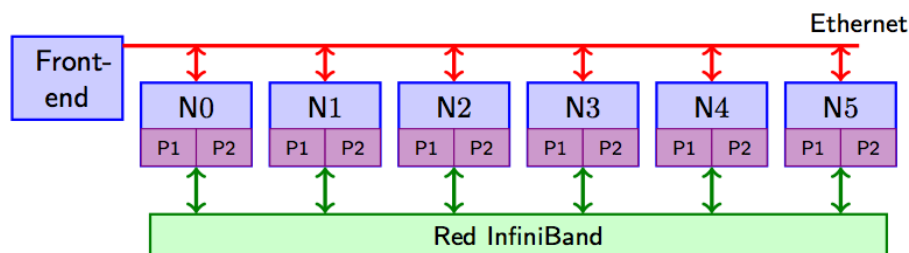


Figura 3.4: Diagrama del cluster kahan

En las prácticas de las asignaturas de LPP y CPA la ejecución de los pasos de conexión al *Frontend*, envío de código a una cola, revisión de estado de la ejecución y descarga de resultados es un trabajo manual que inclusive llega a ser tedioso ya que los tiempos en los cuales será procesada nuestra petición varía según el uso y estado actual de las colas en kahan. Por esto, uno de los puntos más importantes que vale la pena remarcar dada su complejidad será lograr que la aplicación se encargue de la comunicación con el *cluster* realizando el proceso completo de forma automatizada, añadiendo también la verificación de resultados y así ofrecer una mejor experiencia de usuario.

Además, el proceso de interacción con el *cluster* kahan debe ser asíncrono y no bloquear el funcionamiento de la aplicación web y la mejor forma de lograr esto es disponer de un *Administrador de Tareas*, que en la práctica es un sistema de colas donde se envían “trabajos” (*jobs*) a procesarse en un determinado momento de tiempo.

El trabajo, no obstante, se ha desarrollado con la idea puesta en su extensibilidad a cualquier otro *cluster*. De hecho, el *cluster* kahan va a ser sustituido este curso que viene por *kahan2*, que será más avanzado y ligeramente diferente del actual. El trabajo realizado aquí, por tanto, está preparado para ser migrado al nuevo *cluster* con mínimas modificaciones.

### 3.7 Riesgos

---

Un análisis de riesgo también forma parte del estudio del problema ya que se debe evaluar si el resultado esperado es realizable en tiempo y forma. En el listado de requisitos anterior se pueden identificar tres puntos críticos ya que tienen una complejidad alta.

En primer lugar, la interacción con el *cluster* kahan puede ser dificultosa. Por un lado, porque es posible que no se disponga del software adecuado en el *Frontend* para la instalación de la aplicación o porque los administradores del sistema no tengan la capacitación para instalar dicho software. Por otro lado, porque la solución a lo anterior significa el acceso de la aplicación desde un sistema externo. Además, hay que sumarle la posibilidad de que la aplicación se despliegue en una solución *cloud*, siendo el autor de la memoria un usuario novel de esos sistemas. Y por último la verificación de resultados, asignación de Insignias y tablas de posiciones (*Leaderboards*) que pueden ir cambiando en el tiempo.

Para mitigar riesgos se ha seguido un plan de acción con el apoyo del Tutor. En el primer caso se han hecho pruebas utilizando los protocolos SSH y Secure Copy (SCP) para replicar los pasos manuales que haría un Alumno en una práctica de una asignatura como LPP o CPA y verificar que pueden ser automatizados. El Autor por su parte ha participado de un curso online [9] sobre la solución *cloud* escogida para mejorar su base de conocimiento; y, por último, se han analizado las librerías disponibles del *software* a utilizar para simplificar las tareas complejas expuestas previamente.

Una vez realizadas estas tareas y viendo que el riesgo remanente es bajo se ha optado por continuar con el calendario de entrega del TFG en la fecha acordada.

### 3.8 Análisis del marco legal y ético

---

Como este es un proyecto de código abierto se ha escogido la Licencia MIT<sup>2</sup> ya que es una de las más permisivas para fomentar la reutilización y colaboración. Este tipo de licencia permite el libre uso de forma privada o comercial, la modificación y distribución del código sin ningún tipo de garantía donde solo se pide que se mantengan las menciones a la licencia y derechos de autor.

Además, todos los elementos utilizados en el producto final que no son creación del Autor de este TFG, como librerías *javascript* e imágenes, son de libre uso y tienen su atribución correspondiente.

### 3.9 Plan de Trabajo

---

Después de haber revisado el listado de todos los requerimientos funcionales se ha creado un resumen agrupado a alto nivel para dar un seguimiento sin la necesidad de una herramienta externa. El listado se ha subido como parte del fichero **README.md** del repositorio de código y permite tanto al Tutor como al Autor del TFG ver el estado del desarrollo y cuáles son los tópicos a abordar en el paso siguiente.

Con el uso del listado y una reunión semanal vía *Skype* se ha seguido el siguiente protocolo: 1) revisión de las funcionalidades implementadas mediante una *demo*; 2) ronda de preguntas y dudas a resolver; 3) acuerdo entre partes de las próximas funcionalidades a implementar para la próxima semana.

---

<sup>2</sup>Licencia MIT: <https://opensource.org/licenses/MIT>.



Debido a ciertas dificultades presentadas en el proceso de desarrollo y el esfuerzo que podía realizar el Autor (tiempo libre disponible) debido a la carga de su trabajo actual se ha ido pivotando entre las prioridades cada semana. Desde un punto de vista de proceso se puede asumir que se ha seguido un Desarrollo Ágil, similar al de *Scrum*, seleccionando un listado de tareas a abordar en un *sprint* (iteración para entregar “valor”) con una duración de una semana.

*Scrum*, como metodología de Desarrollo Ágil, se centra en tener un *software* en funcionamiento antes que seguir un proceso rígido de desarrollo; premiando así la entrega constante y la posibilidad de adecuarse al cambio.



---

## CAPÍTULO 4

# Diseño de la solución

---

Una vez analizados los requisitos funcionales y no funcionales el próximo paso consiste en plantear el diseño de la aplicación. Aunque la finalidad de este trabajo es la creación de una aplicación web existen elementos adicionales que deben considerarse para cumplir con el objetivo ya que, aunque el punto de entrada de interacción del sistema será una página web, toda la información se tendrá que alojar en una Base de datos (BD) y los Envíos que contienen ficheros con código fuente para la resolución de Tareas deberán guardarse en un Sistema de archivos de red (NFS). Estos elementos junto a otros más forman parte de lo que será la **Arquitectura de la solución**. Pero antes de llegar a esto hay que comprender cómo será la **Interacción del Sistema** y decidir el **Software** a utilizar que lo haga posible.

Además, para alojar y recuperar información de una BD debemos crear unos modelos que representen a todos los objetos que interactúan dentro del sistema, lo que se conoce como **Modelo de Datos**. Por último, la interacción con el *cluster* kahan es tan importante que tendremos que incluir un *Administrador de Tareas* para el procesamiento de *jobs* de forma asíncrona como un **Sistema de Colas interno**.

### 4.1 Interacción del Sistema

---

En la introducción del capítulo se han visto algunos elementos de la aplicación y es importante realizar un bosquejo de cómo será la interacción entre ellos para modelar la solución de forma correcta. Haciendo un repaso podemos encontrar los siguientes:

- La aplicación web con la que interactúan los usuarios.
- La Base de datos (BD) donde se guarda la información.
- Un Sistema de archivos de red (NFS) para alojar los ficheros con código fuente de los Envíos.
- Un *Administrador de Tareas* basado en un sistema de colas.
- El *cluster* kahan del DSIC.

Una interacción habitual sería la que se muestra en la Figura 4.1, donde el *Administrador de Tareas* es el encargado de la comunicación con el sistema de colas de kahan para actualizar el *status* del Envío. A partir de aquí ya queda clara la importancia de una separación de roles entre lo que será el *frontend* o punto de interacción de los usuarios con la aplicación y el procesamiento de Envíos. Esto tendrá vital importancia en la elección del software a utilizar.

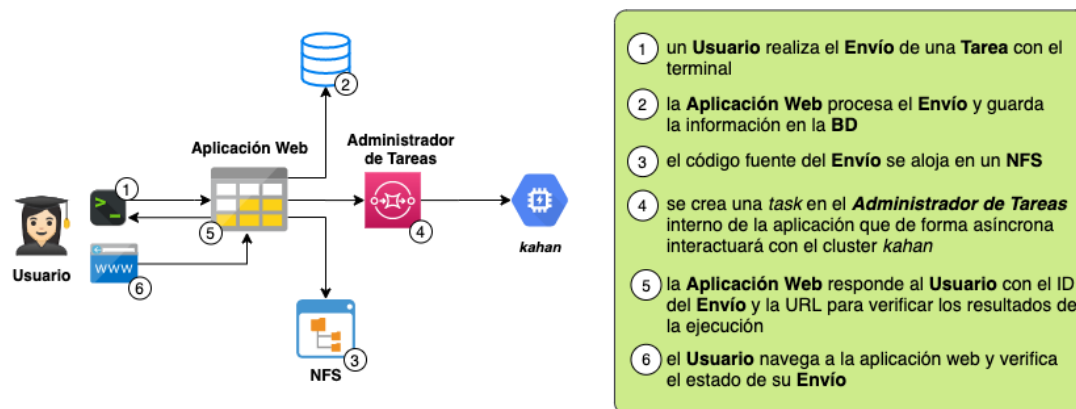


Figura 4.1: Interacción entre elementos del sistema.

## 4.2 Tecnología utilizada

Como hemos visto en el apartado **Estado del Arte** la única herramienta similar a la aplicación web a desarrollar es *Tablón*, no sin sus muchas limitaciones y problemas que ya hemos expuesto previamente. Sin embargo, *Tablón* es un buen punto de partida porque ya ha sido utilizado previamente por el Tutor en la asignatura de LPP, y algunos de los requisitos funcionales, como la Tabla de Posiciones (RF-10), ya existen y pueden servir como inspiración.

Dadas las limitaciones que tiene *Tablón*, una de las aspiraciones en la creación de esta aplicación web para concursos de programación paralela es que sea su sucesor (o reemplazo) a futuro. Por esto mismo he decidido llamar a la aplicación web *Pizarra*, en honor a *Tablón*, y de aquí en adelante, cuando se mencione *Pizarra*, se hará referencia a la aplicación web y otros elementos adicionales necesarios para su funcionamiento.

El lenguaje de programación a utilizar será *python*, ya conocido por el Autor y con una extensa cantidad de Paquetes, que son librerías que añaden funcionalidades extra y que serán de gran utilidad para acelerar los tiempos de desarrollo.

Dado que ya hemos abordado la **Interacción del Sistema** a alto nivel solo resta escoger el **Software** que dará vida a la aplicación web y al *Administrador de Tareas*. Debido a una experiencia previa en otra asignatura de estudios de grado, he decido optar por *Flask*<sup>1</sup>, un micro-framework de *python* con gran acogida por la comunidad de desarrollo que incluye un sistema de plantillas para la generación de páginas web llamado Jinja<sup>2</sup>.

En cuanto al *Administrador de Tareas*, existen diferentes Paquetes en *python* para crear un **Sistema de Colas interno**. En este caso se ha optado por *RQ*<sup>3</sup> (Redis Queue), que utiliza pocos recursos y permite que los “trabajos” o *jobs* se procesen en determinado tiempo futuro, funcionalidad que será útil para revisar los resultados de ejecuciones en kahan. *RQ* permitirá desacoplar las Colas del aplicativo web al utilizar un servicio de *Redis* como repositorio de *jobs*, además de disponer de un panel de control (*dashboard*) que puede ser embebido en cualquier aplicación web.

<sup>1</sup>Sitio web Flask: <https://flask.palletsprojects.com/en/1.1.x/>.

<sup>2</sup>Sitio web Jinja: <https://jinja.palletsprojects.com/en/2.11.x/>

<sup>3</sup>Sitio web RQ: <https://python-rq.org/>.

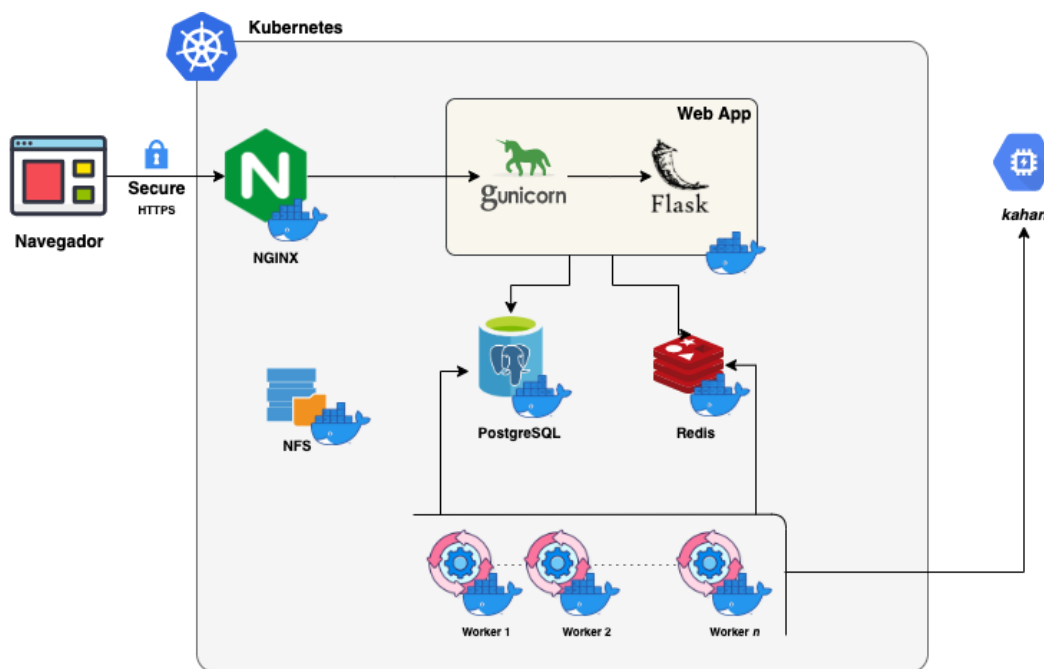


Figura 4.2: Arquitectura del Sistema.

### 4.3 Arquitectura del Sistema

Para el despliegue de aplicaciones basadas en *Flask* existe una lista de recomendaciones que debemos seguir [10]. En primer lugar, *Flask* no está optimizado para aceptar peticiones web directamente de un navegador ya que utiliza la Web Server Gateway Interface (WSGI) para mostrar el contenido web de sus aplicaciones, siendo WSGI un estándar de *python* que permite a cualquier aplicación comunicarse a través del protocolo HTTP, es decir, Internet.

Además, la ejecución sin más de un *script* en *python* no es una buena práctica ya que una excepción del sistema u otras situaciones anormales que puedan producirse puede abortar el proceso. Para estos casos existe Gunicorn<sup>4</sup>, que funciona como un servidor de aplicaciones y se encarga de ser el intermediario entre una petición web y *Flask*, además de monitorear el servicio y reiniciarlo en caso de problemas.

Otra buena práctica al realizar un despliegue es poner nuestra aplicación detrás de un proxy reverso, esto es, un Servidor Web que se encargue de administrar las conexiones y servir los certificados digitales y ficheros estáticos. Este tipo de arquitectura brinda una mejora en el rendimiento global del sistema y una separación de roles donde cada capa o *layer* tiene una tarea específica. Para la arquitectura actual se ha escogido NGix<sup>5</sup>, uno de los servidores webs más utilizados en Internet.

Ya se ha mencionado que el Paquete *RQ* utiliza un servicio de *Redis* para el **Sistema de Colas interno**, pero además brinda una funcionalidad extra llamada *Worker* o “trabajador”: un proceso en *python* que se ejecuta como un proceso en segundo plano con el único objetivo de procesar los *jobs* de una Cola. El uso de estos *Workers* no tiene límite y permite un uso “elástico” de los recursos, aumentando y disminuyendo su cantidad dependiendo la cantidad de *jobs* a procesar.

<sup>4</sup>Sitio Web Gunicorn: <https://gunicorn.org/>.

<sup>5</sup>Sitio web Ngix: <https://www.nginx.com/>.

Existen tres elementos de la arquitectura que necesitan de almacenamiento: 1) la aplicación web, para guardar los ficheros adjuntos de las Tareas y el código fuente de los Envíos, 2) los *Workers*, que deben poder acceder al código fuente para transferirlo a kahan, y 3) la Base de datos, para almacenar las tablas.

Por último, todos los elementos estarán desplegados en forma de *contenedores* Docker, un sistema que empaqueta el *software* con todo lo necesario para que se ejecute y que es explicado en detalle en el Capítulo 6 dedicado a la **Implantación**.

## 4.4 Modelo de datos

Una de las formas habituales de trabajar con BD en *python* es utilizar un *Object-Relational mapping* (ORM) que transforma clases que representan una entidad o modelo a una estructura de datos relacional. La librería a utilizar será *SQLAlchemy*, un ORM que soporta varios motores de BD, entre otros SQLite, MySQL, PostgreSQL, Oracle o MS SQL. En este caso se ha optado por SQLite para el entorno de desarrollo local (se crea el fichero *database.db* en el directorio raíz del aplicativo), y PostgreSQL, para un entorno productivo.

SQLite porque no necesita tener iniciado un proceso en la máquina ya que se escribe directamente a disco con un conector (*driver*) y es ultra liviano en el uso de recursos ya sean memoria y espacio en disco. PostgreSQL tiene todas las características de un motor de BD robusto al nivel de otras soluciones comerciales, ofrece una imagen de *Docker* y no es necesario implementar un modelo de replicación. Otra añadido es que ambas herramientas son de libre uso y tienen soporte continuo de la comunidad.

Dados los requerimientos funcionales, se han identificado y creado las siguientes clases de *python* que, mediante el uso de *SQLAlchemy*, podrán ser persistentes, consultadas y actualizadas en la Base de datos (BD).

- **User:** *usuario* del sistema.
- **Role:** roles del sistema, en este caso Alumno y Administrador.
- **Team:** Alumno que forman un Equipo.
- **ClassGroup:** Grupo de Alumnos que representan una clase en particular.
- **Badge:** Insignia obtenida al completarse objetivos.
- **Request:** Envío de código fuente de un usuario a una Tarea.
- **Assignment:** Tarea a resolver, puede estar asignada a uno o más grupos.
- **LeaderBoard:** marcador de posiciones para una Tarea y Grupo específico.
- **Attachment:** fichero adjunto añadido a una Tarea.
- **Job:** trabajo a procesar, utilizada por la Cola interna de *Pizarra*.

Los modelos expuestos previamente se pueden encontrar en el fichero **models.py** siguiendo el estándar de *Flask* y se expresan como clases que extienden de la clase *db.Model* del paquete de *SQLAlchemy*. Esto permite la serialización del objeto y su persistencia en la Base de datos (BD).

Se muestra a continuación, como ejemplo, la representación de un Equipo (*Team*):

```
...  
class Team(db.Model):  
    """  
    Represents a Team of students in the database  
    A Student can be a part of only one team  
    A Team can have many students  
    """  
    __tablename__ = 'team'  
    id = Column(Integer, primary_key=True)  
    name = Column(String, unique=True)  
    key = Column(String)  
    members = relationship('User', back_populates='team')  
    ...
```

models.py

En el código expuesto se pueden identificar los siguientes elementos:

- la clase *Team* extendiendo la clase *db.Model*,
- una descripción de la clase y su propósito enmarcada en comentarios,
- el nombre de la tabla para persistir el objeto una vez serializado,
- las diferentes columnas de la tabla con sus atributos (si es clave primaria, clave única y el tipo de datos), y
- la relación entre *Team* y *User* (un Equipo tiene integrantes y un Alumno es parte de un Equipo).

Una vez definidos todas las clases con sus relaciones, podemos generar un diagrama de la BD. Esto facilita la comprensión y ofrece una mejor visión general. Utilizando una herramienta ya incluida en el IDE de desarrollo *PyCharm*<sup>6</sup> se obtiene la Figura 4.3.

Al ser las tablas de la BD objetos serializados de *python*, podemos asumir que la figura generada es la misma representación de un diagrama de clases en Lenguaje Unificado de Modelado (UML): una estructura que muestra las relaciones de las clases dentro del sistema, sus atributos y métodos u operaciones.

## 4.5 Diseño de la Aplicación en Flask

Al crear una nueva aplicación en *Flask*, existen ciertos patrones o mejores prácticas a seguir. Entre ellas se encuentra la utilización de uno de los patrones de diseño más comunes en la ingeniería de software, Modelo Vista Controlador (MVC) [3]. Con este patrón se crea una separación de capas entre lo que es el Modelo, que contiene la representación de los datos y su persistencia, la Vista, que es la interfaz de usuario, y el Controlador, como intermediario entre el Modelo y la Vista.

Una aplicación estándar en *Flask* incluye los siguientes ficheros:

- **models.py**: representación del modelo de datos (Modelo).

<sup>6</sup>Sitio web Pycharm: <https://www.jetbrains.com/pycharm/>.

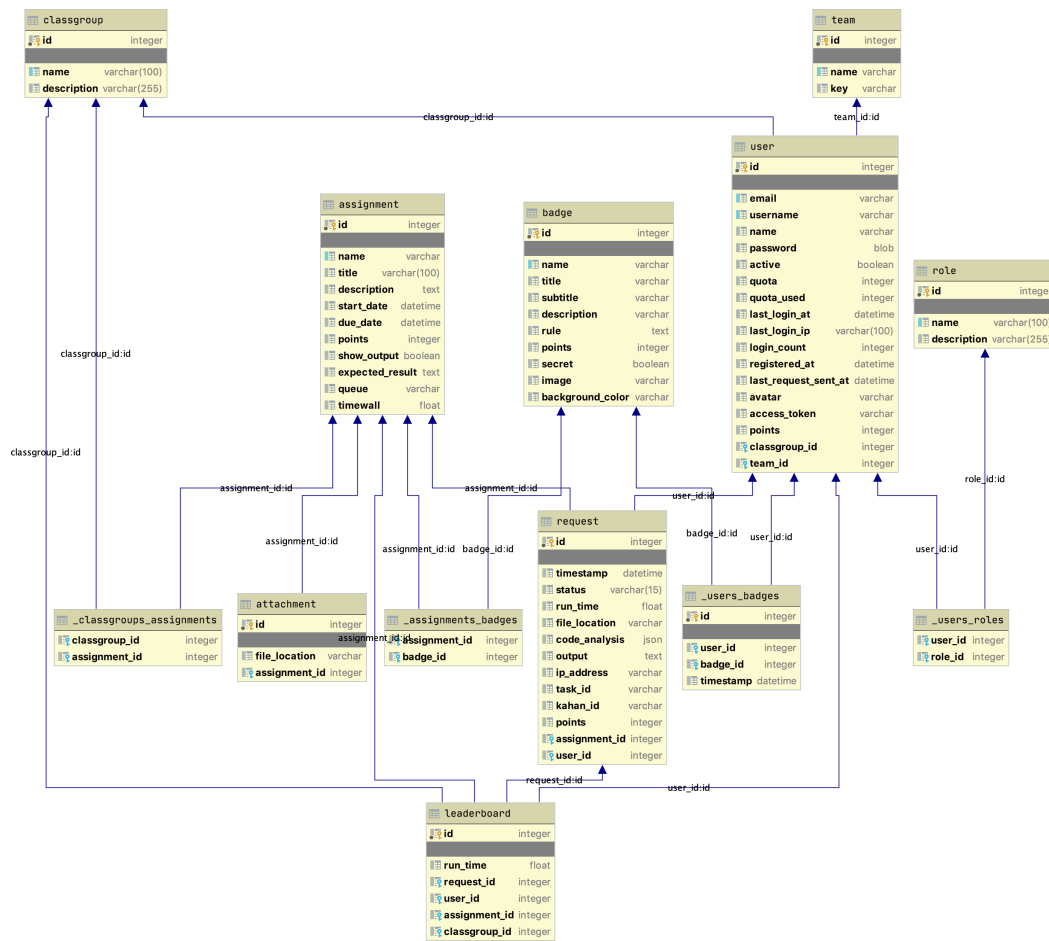


Figura 4.3: Diagrama UML de la Base de Datos.

- **templates:** carpeta con las plantillas HTML para la generación de páginas web (Vista).
- **routes.py:** rutas de acceso a la aplicación (Controlador).
- **\_\_init\_\_.py:** inicialización de la aplicación y librerías.
- **forms.py:** formularios y validación.

Además, otra buena práctica consiste en separar las aplicaciones en módulos para añadir extensibilidad mediante reglas, nuevas vistas, separación de roles y otros beneficios añadidos mediante el uso de *blueprints* [4], una funcionalidad incluida en *Flask*.

De esta forma se ha optado por crear los siguientes módulos o *blueprints*, todos estos dentro del directorio **app**:

- **account:** apartado de “mi cuenta” de un usuario.
- **admin:** panel de control del administrador, solo accesible para usuarios con ese rol.
- **base:** contiene la definición de Modelos y plantillas base.
- **data:** importación de datos de ejemplo.
- **home:** dashboard de usuario.



Si se lista el directorio **app** se puede ver cómo cada módulo contiene una estructura de datos similar, siguiendo el estándar descrito previamente.

```
$ tree -L 2 app/  
app/  
|--- __init__.py  
|--- account  
|       |--- __init__.py  
|       |--- forms.py  
|       |--- routes.py  
|       +--- templates  
|--- admin  
|       |--- __init__.py  
|       |--- forms.py  
|       |--- routes.py  
|       +--- templates  
|--- base  
|       |--- __init__.py  
|       |--- forms.py  
|       |--- models.py  
...
```

## 4.6 Sistema de Colas interno

Como se vio previamente, *Pizarra* tiene un sistema de Colas interno. Esto se debe a varios motivos de peso. En primer lugar, se quiere establecer una separación de roles entre la aplicación web con la que los usuarios interactúan y la ejecución de código fuente. *Pizarra* debe estar preparada para encolar tareas en el cluster kahan del DSIC, pero otra de las posibilidades es la disponibilidad de poder hacerlo de forma local para cualquier otra persona que quiera hacer uso del aplicativo. *Tablón*, el software en el cuál nos hemos inspirado, también ofrece ejecución local, pero con ciertas limitaciones, siendo una de ellas la obligación de reiniciar la aplicación cada 60 minutos [12]:

*«El servidor tablón está configurado para reiniciarse cada cierto tiempo. De está forma se eliminan trabajos erróneos y se eliminan las peticiones http no completadas. El tiempo que resta hasta el próximo reinicio se muestra en la página principal».*

Además, una ejecución local desde el mismo aplicativo obliga a brindarle los recursos necesarios, ya sean procesadores (CPUs) o memoria (RAM). Esto no es muy eficiente, ya que las aplicaciones en *python*, y especialmente en *Flask*, consumen muy pocos recursos. Por eso mismo, este aplicativo se puede ejecutar en dos modos diferentes:

- **Modo Pizarra:** aplicación web con la que interactúan los usuarios.
- **Modo Worker:** un proceso que se encarga de revisar en las Colas internas si hay *jobs* a procesar.

La elección por la cual se han escogido dos modos diferentes de ejecución de la aplicación en contraposición a la creación de diferentes *scripts* es para poder hacer uso de la misma definición de contenedor en *Docker*, una estrategia habitual al utilizar esta solución [11].

Por último, otro beneficio que brinda esta separación de roles consiste en que se pueden ejecutar un número indefinido de *Workers*, donde cada uno de ellos puede tener diferentes colas configuradas y unos recursos disponibles exclusivos.

---

## CAPÍTULO 5

# Desarrollo de la solución

---

En este capítulo presentamos un resumen de cómo se ha pasado del diseño de la aplicación al producto final dando un repaso de cómo se ha abordado el desarrollo de los aspectos más relevantes como el Envío de Tareas, el motor de reglas y el autómata creado para el proceso de tareas en diferentes tipos de Colas. Esto no podría ser posible sin describir el listado completo de librerías necesarias y el sistema de plantillas que se utiliza para generar el contenido web. Por último, se presenta un resumen de los entregables que se fueron generando para dar vida al producto final.

### 5.1 Flask *boilerplate*

---

La usabilidad es uno de los aspectos más importantes de una aplicación web y, en este caso, no se puede dejar de lado. Un *usuario* de *Pizarra* esperará una interfaz intuitiva y fácil de utilizar que le facilite el proceso de envío de soluciones (*Requests*) a los problemas propuestos (*Assignments*). No queremos que la aplicación se interponga entre el proceso de aprendizaje y la satisfacción de participar y obtener logros (*Badges*). Una gran cantidad de aplicaciones utilizan las mismas librerías para maquetar los sitios web. Entre estas librerías, la más importante es *Bootstrap*<sup>1</sup> y, además, alrededor de estas librerías existen *Dashboards* o Paneles de Control de código abierto para utilizar como punto de partida en nuevas aplicaciones.

Este conjunto de librerías y plantillas HTML tienen el nombre de *boilerplate* y aceleran el desarrollo de una aplicación web tanto para usuarios noveles como también para usuarios expertos, ya que solo deben preocuparse de reutilizar componentes ya predefinidos.

Después de una búsqueda y pruebas de varias alternativas se ha optado por *AdminLTE*<sup>2</sup> (Figura 5.1) dada su variedad de plantillas disponibles, uso de otras librerías con amplia documentación y un aspecto visual amigable y moderno.

### 5.2 Paquetes de Python

---

Para dar funcionalidades adicionales en *python*, normalmente se utilizan librerías o Paquetes externos que simplifican y aceleran el desarrollo de la aplicación. A continuación se presenta un repaso de las librerías que son necesarias para *Pizarra*. Este listado se encuentra disponible en el fichero **requirements.txt**, un estándar utilizado por desarro-

---

<sup>1</sup>Sitio web Bootstrap: <https://getbootstrap.com/>.

<sup>2</sup>Sitio web AdminLTE: <https://adminlte.io/>.

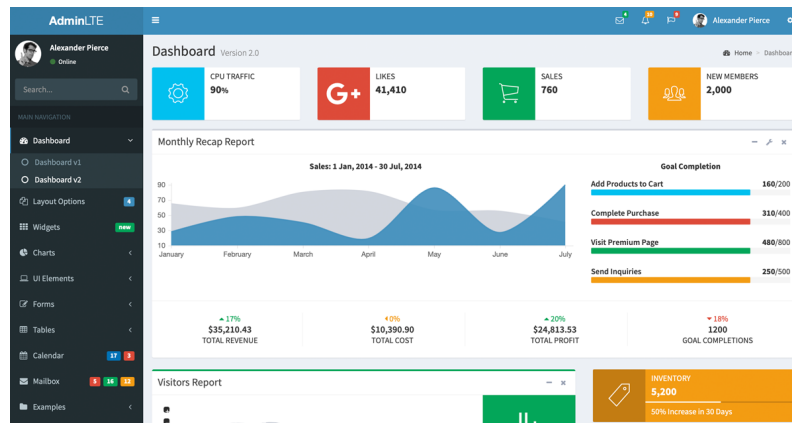


Figura 5.1: AdminLTE: plantilla de Panel de Control.

lladores y proveedores de servicios de *hosting* [5] para su fácil instalación a posteriori por administradores de paquetes de *python*, como *pip*<sup>3</sup>.

Un sistema de administración de paquetes es un *software* que se encarga de resolver las referencias en un repositorio central, de su descarga y de su instalación de forma segura.

### 5.2.1. Flask

Este es el framework base de la aplicación, junto con otros Paquetes adicionales que dan soporte a ciertas funcionalidades no incluidas en la librería base:

- **flask**: framework base.
- **flask\_login**: funcionalidades extras para el manejo de sesiones de *usuario*.
- **flask\_security**: uso de roles en el sistema.
- **flask\_wtf**: validación de formularios web.
- **flask\_sqlalchemy**: soporte a SQLAlchemy para el mapeo objeto-relacional a diferentes BD.
- **flask\_migrate**: migración de base de datos basadas en SQLAlchemy.
- **flask-babel**: localización de la aplicación en diferentes idiomas.
- **email-validator**: validación de correos electrónicos para la librería wtf.

### 5.2.2. Sistemas de Colas

Paquetes para el funcionamiento del Sistema de Colas interno y conectividad con *Redis*:

- **rq**: librería de Redis Queue (*RQ*).
- **rq-dashboard**: visualización web del estado de *queues* y *jobs*.
- **redis**: soporte y conectividad al servicio de *Redis*.

<sup>3</sup>Sitio web pip: <https://pip.pypa.io/en/stable/>.

### 5.2.3. Conexión con sistemas externos

Para poder utilizar las Colas de kahan es necesario poder conectarnos vía protocolos seguros, ya sea para ejecutar comandos o para copiar ficheros, para cada uno de ellos son necesarios:

- **paramiko**: soporte de protocolo Secure Shell (SSH)<sup>4</sup>.
- **scp**: soporte de protocolo Secure Copy (SCP)<sup>5</sup>.

### 5.2.4. Datos de ejemplo

Para facilitar el desarrollo y su posterior mantenimiento se debe poder disponer de una forma fácil y estandarizada de carga de datos de ejemplo. Para esta funcionalidad se ha optado por ficheros en formato *yaml*. Los Paquetes que añaden este soporte son:

- **pyyaml**: analizador de ficheros *yaml*.
- **sqa-yaml-fixtures**: definición de datos via *yaml*, transformación a objetos e importación a la BD con *SQLAlchemy*.

### 5.2.5. Otros

Además, hay algunas funcionalidad específicas que requieren el uso de las siguientes librerías:

- **gunicorn**: ejecución y monitoreo de la aplicación.
- **lizard**: analizador de código estático.
- **rule\_engine**: evaluador de reglas con expresiones lógicas, utilizado para asignar Insignias.

## 5.3 Envío de soluciones a Tareas

---

Cuando un Alumno realiza un Envío, esto es, una petición web que incluye el código fuente a compilar y ejecutar como propuesta de solución a una Tarea, la aplicación debe procesar la petición. El Envío pasa por una serie de verificaciones que se describen a continuación, algunas de las cuáles no se realizan para un usuario con rol de Administrador del sistema:

1. Se autentifica al usuario.
2. Se busca la Tarea en la BD.
3. La Tarea debe estar abierta a recibir Envíos.
4. El Alumno pertenece a un Grupo que la tenga asignada.
5. El Alumno no ha enviado peticiones muy rápido.

---

<sup>4</sup>Sitio web SSH: <https://tools.ietf.org/html/rfc4253>

<sup>5</sup>Sitio web SCP: <https://datatracker.ietf.org/doc/draft-evans-v2-scp/>

6. El Alumno tiene Quota (tiempo disponible de ejecución) para procesar el Envío.
7. El Envío contiene un fichero a procesar.
8. El fichero a procesar con el código fuente no contiene código malicioso.

Si alguno de los pasos anteriores no pasa uno de los puntos de verificación se devuelve un mensaje de error con el código HTTP correspondiente. En el caso de que todas los chequeos sean satisfactorios se crea un objeto de tipo *Request* en la BD y otro de tipo *job* para la Cola interna en *Redis*.

El Envío de una Tarea se realiza desde un terminal de comandos u otro cliente que permita realizar peticiones web como *Postman*<sup>6</sup>, y está construida para devolver respuestas en formato JSON, que es un estándar utilizado en muchas aplicaciones web cuando se diseña una API. Un Envío contiene como parámetros el usuario, el token de acceso, un fichero con el código fuente y la URL de la Tarea. Al procesarse la petición se recibe la respuesta.

En el siguiente código se muestran tres ejemplos: 1) una autenticación fallida, 2) un Envío satisfactorio, y 3) un Envío no procesado porque el usuario debe esperar un tiempo.

```
$ curl -k -X POST -F 'file=@primos.c' http://foo:bar@127.0.0.1:5000/assignments/primos
{
  "code": 401,
  "message": "Authentication failed"
}
$ curl -k -X POST -F 'file=@primos.c' http://foo:gB8HSgWhfg@127.0.0.1:5000/assignments/primos
{
  "code": 201,
  "message": "Request created, please navigate to http://127.0.0.1:5000/requests/13 to check the results"
}
$ curl -k -X POST -F 'file=@primos.c' http://foo:gB8HSgWhfg@127.0.0.1:5000/assignments/primos
{
  "code": 403,
  "message": "You are sending Requests too fast. Time between Requests is 60 seconds"
}
```

El diagrama de flujo de una petición web para un Envío es el que se ve en la Figura 5.2.

---

<sup>6</sup>Sitio web Postman: <https://www.postman.com/>.

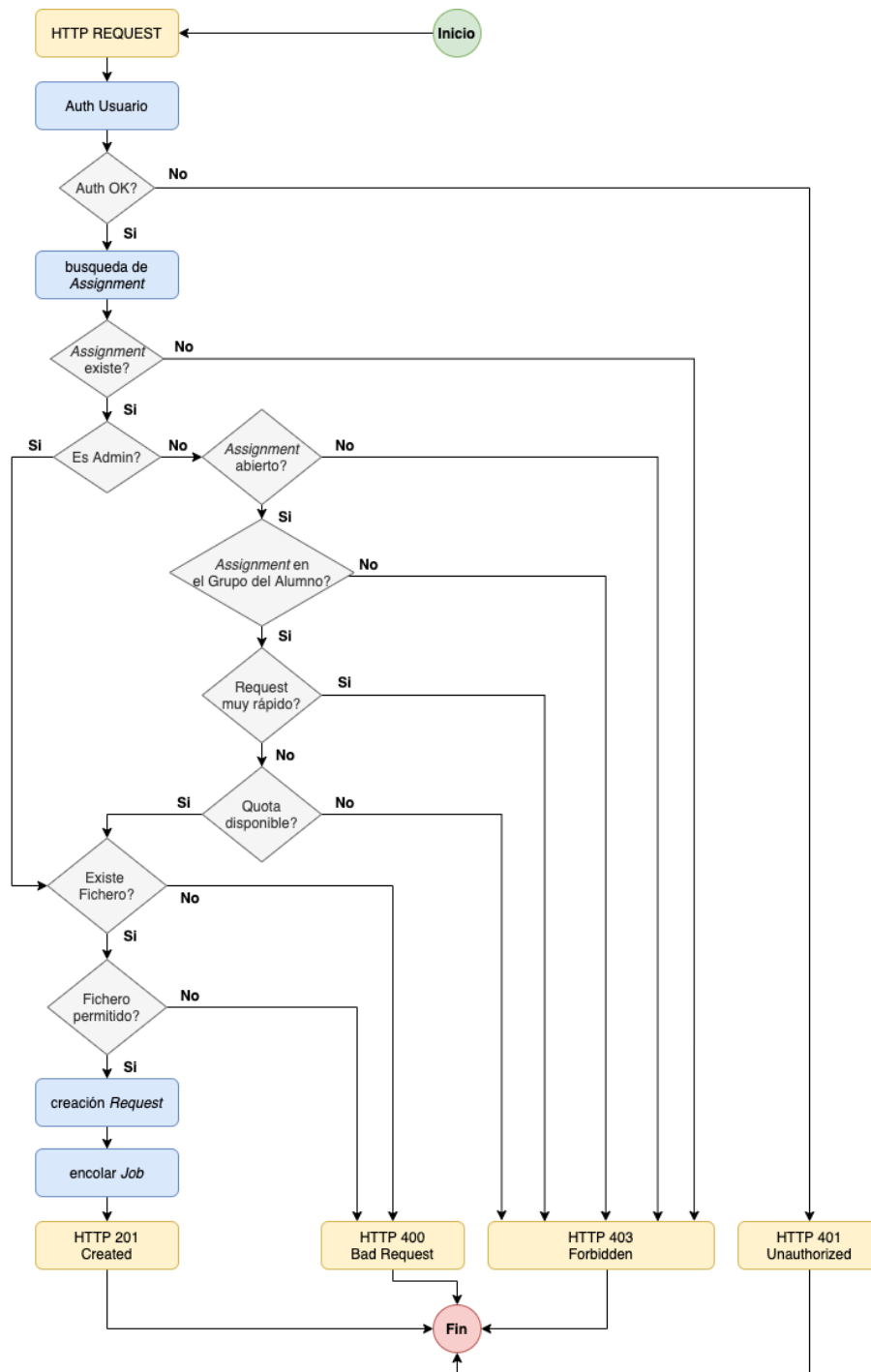


Figura 5.2: Diagrama de flujo de una petición web para un Envío.

## 5.4 Ejecución en una Cola interna

Previamente se ha hablado de que el Paquete *RQ* permite simular un *Administrador de Tareas* con el uso de una Cola interna de “trabajos” o *jobs* ejecutados por unos *Workers*, aunque ha faltado explicar el cómo. Primero, vamos a presentar las dos clases que proporciona *RQ*; 1) *Worker* y 2) *Queue*, y cómo son utilizadas por *Pizarra*.

En el siguiente ejemplo podemos ver la creación de un “trabajador” o *Worker* al iniciar *Pizarra* en este modo.

```
...
redis_url = app.config["RQ_DASHBOARD_REDIS_URL"]
redis_connection = redis.from_url(redis_url)
with Connection(redis_connection):
    worker = Worker(app.config['QUEUES'])
    worker.work(with_scheduler=True)
...
```

app/\_\_init.py\_\_

El extracto de código del fichero **app/\_\_init.py\_\_** realiza el siguiente trabajo:

1. Lee de la configuración de *Pizarra* la URL del servicio de *Redis*.
2. Crea una conexión con el servicio de *Redis*.
3. Crea un Objeto de tipo *Worker* con las Colas disponibles.
4. Pone el *Worker* a “trabajar” revisando el servicio de *Redis* periódicamente para ver si hay *jobs* a ejecutar.

El próximo paso es ver cómo se crean los *jobs* que, en el caso de nuestra aplicación, serán las *Requests* o Envíos de los *usuarios* a procesar por los *Workers*. Para esta funcionalidad tenemos dos métodos en el fichero **app/base/models\_jobs.py**

```
...
def process_job(job):
    """
    process a given job
    """
    job.process()

def create_job(user_request) -> str:
    """
    creates and enqueues a job from pizarra to be executed by a
    worker and returns job id
    """
    with Connection(redis.from_url(current_app.config["
        RQ_DASHBOARD_REDIS_URL"])):
        q = Queue(name=user_request.assignment.queue)
        rq_job = q.enqueue(process_job, LocalJob(user_request))
        return rq_job.get_id()
...
```

app/base/models\_jobs.py

El método *create\_job* recibe como parámetro un *Request*, donde su definición no es relevante, salvo que se utilizará para el *job* a procesar y, además, tiene el nombre de la *Queue* interna donde debe encolarse. Realiza lo siguiente:

1. Abre una conexión con el servicio de *Redis*.
2. Crea un objeto de tipo *Queue* con el nombre de la Cola a utilizar.



3. crea un *job* utilizando el método *process\_job*, que recibe como parámetro una clase genérica que debe tener implementado el método *process*. Ese *job* se inserta en la *Queue*.
4. Devuelve el ID del *job* para poder asignarlo a la *Request*.

Por último, veo muy importante explicar la creación del siguiente autómeta, ya que es la forma en la cual se ha podido resolver la ejecución en una Cola Local de kahan así como en cualquier otra que se quiera utilizar en un futuro. El código presentado de la clase *LocalJob* es una simplificación de la clase real de *Pizarra*, donde se presentan los elementos básicos para dar el contexto de su funcionamiento. Para facilitar la lectura se ha dividido el código fuente en tres recuadros, asumiendo que forman un solo código.

```
class StepResult(enum.Enum):
    START = 0
    OK = 1
    NOK = 2
    WAIT = 3
    END = 4

class LocalJob:
    def __init__(self, user_request):
        ...
        self.user_request = user_request
        self.task_process = {
            RequestStatus.CREATED: {
                'f': self.start,
                'steps': {
                    StepResult.OK: RequestStatus.VERIFYING,
                }
            },
            RequestStatus.VERIFYING: {
                'f': self.verify,
                'steps': {
                    StepResult.OK: RequestStatus.COMPILE,
                    StepResult.NOK: RequestStatus.ERROR
                }
            }
        }
        ...
    }
```

app/base/models\_jobs.py

Este autómeta tiene una serie de etiquetas que ayudarán a saber a qué estado se debe pasar a continuación (*StepResult*). Al crear un objeto del tipo *LocalJob* en esta versión simplificada se puede ver cómo se asigna el *Request* y, además, existe la definición del autómeta como un mapa, donde para una etiqueta se asigna un método a ejecutar y un mapa de transiciones que depende del resultado de la ejecución del paso.

```
def get_task_step(self):
    return self.task_process[self.user_request.status]

def is_job_completed(self):
```

```

return self.user_request.status in [RequestStatus.CANCELED
    , RequestStatus.ERROR, RequestStatus.TIMEWALL,
    RequestStatus.FINISHED, RequestStatus.UNHANDLED_ERROR]

```

app/base/models\_jobs.py

La función *get\_task\_step* devuelve el método para el paso en el que se encuentra el autómata y el atributo dinámico *is\_job\_completed* verifica si el proceso ha terminado.

```

def process(self):
    """
    process the task request step by step until the task is
    enqueued again or it finishes
    """
    step_result = StepResult.OK
    while step_result != StepResult.END:
        step = self.get_task_step()
        f = step['f']
        step_result = f()

        if step_result == StepResult.WAIT:
            requeue_job(self)
            break

        self.update_request(step['steps'][step_result])

def start(self):
    return StepResult.OK

def verify(self):
    return StepResult.OK if (
self.__not_contains_malicious_content() and self.
__static_code_analysis()) else StepResult.NOK

```

app/base/models\_jobs.py

En el tercer bloque se encuentra la función *process* que, como explicamos previamente, es el método que ejecuta el *Worker* al obtener un *job* de una Cola. Obtiene el estado actual de la *Request* y lleva a cabo el siguiente proceso hasta terminar:

1. Obtener la función.
2. Ejecutar la función y asignar el resultado al estado actual.
3. Si el resultado es *WAIT*, vuelve a encolar la tarea para ejecutarla en un futuro. Esto se realiza para Colas como la de kahan, donde debemos revisar el estado de una ejecución cada cierto periodo de tiempo.
4. Se actualiza el estado de la *Request*, donde el nuevo estado se obtiene del mapa *steps* del resultado del paso.

Como se puede ver en el método de ejemplo *start* se devolverá un OK y el nuevo estado de la *Request* será *VERIFYING*. En el próximo paso ejecutará el método *verify* que

llama a dos métodos adicionales que verifican que no haya código malicioso y realiza el análisis del código estático; y así hasta terminar la ejecución.

Gracias a este simple pero efectivo autómatas se podrá extender la aplicación para que soporte otros tipos de Cola, tal como se verá más adelante en el Capítulo 9 dedicado a la Extensibilidad del software desarrollado.

## 5.5 Asignación de Insignias

Cuando se llegó a la etapa de desarrollo en la cuál se tenían que crear las Insignias (*Badges*) se encontró un problema de gran dificultad. ¿Cómo permitir que un Administrador, desde su panel web, pueda hacer que una Insignia se aplique cuando suceda algo en particular? Las Insignias no solo se obtienen al completar una Tarea de forma correcta, sino que también pueden obtenerse en otros casos como, por ejemplo, que el Envío se encuentre en los primeros tres puestos de la tabla de posiciones (*leaderboard*).

La primera opción, que posteriormente fue descartada, fue crear una clase en *python* por cada tipo de Insignia que se quería asignar a un Alumno. Esto quitaba dinamismo y forzaba al Administrador a modificar el código fuente de la aplicación, algo que quería evitarse ya que impone una barrera de entrada a cualquier persona que quiera hacer uso de *Pizarra* en un futuro.

Después de investigar otras posibilidades y realizar una serie de pruebas se optó por añadir el Paquete *rule\_engine* que permite la creación de objetos de tipo "Rule". Estos objetos pueden ser evaluados con expresiones lógicas que son escritas como cadenas de texto y pueden guardarse en una columna de la BD como atributo de la clase *Badge*. *rule\_engine* no es la única librería que añade este tipo de funcionalidad en *python* pero su simplicidad y el soporte de *Data objects* [13], que es la representación de una clase en formato JSON, fueron dos razones por las cuáles se optó por esta solución.

Con la posibilidad de escribir reglas como cadenas de texto y sabiendo que un *job* tiene los atributos *user*, *request*, *assignment* y *leaderboard* como diccionarios de *python*, podemos crear *Badges* como las que se encuentran en los datos de ejemplos del fichero **sample.yml**.

```
- Badge :  
  - name: "beat_the_machine"  
    description: "Earn when your program runs below a certain  
                threshold"  
    rule: "request.run_time < 10"  
  - name: "leader"  
    description: "Earn when your Request is in the first  
                positions of the leaderboard"  
    rule: "leaderboard.position <= 3"
```

app/data/sample.py

En el ejemplo mostrado en **sample.yml** podemos observar la creación de dos Insignias que se asignarán a un Alumno cuando la ejecución de su programa tarde menos de 10 segundos, y también otro caso donde la posición de la ejecución quede entre las tres mejores. La descripción de los atributos es:

- **name:** ID único de la Insignia.

- **description:** descripción que se mostrará a los *usuarios* del sistema.
- **rule:** regla que se comprobará al terminar la ejecución.

El sistema de reglas expuesto dará pie a la creación de algunas Insignias especiales como, por ejemplo, una primera ejecución correcta en el *cluster*, haber agotado el tiempo de ejecución al recibir un TIMEWALL, que el ID del *Request* sea un número especial como el 1000 y otras.

## 5.6 Entregables

---

Dado que la creación de la aplicación ha sido elaborada dentro de un marco de Desarrollo Ágil, cada entregable ha presentado una serie de nuevas funcionalidades hasta llegar al producto final. Mediante el siguiente resumen de las versiones se puede ver cómo, partiendo de unos cimientos, ha ido evolucionando el *software* hasta su estado actual.

- **Versión 0.1**
  - **fecha entrega:** 8 de Abril.
  - **resumen:** registro e ingreso de *usuarios* con visualización de página de inicio.
  - **requisitos funcionales:** RF-1, RF-2 y RF-4.
- **Version 0.2**
  - **fecha entrega:** 22 de Abril.
  - **resumen:** envío de código, análisis estático y procesamiento. Visualización de Tareas. Creación de equipos.
  - **requisitos funcionales:** RF-3, RF-7, RF-8 y RF-9.
- **Version 0.3**
  - **fecha entrega:** 29 de Abril.
  - **resumen:** asignación de Insignias y visualización de Envíos.
  - **requisitos funcionales:** RF-5, RF-6 y RF-12.
- **Version 0.4**
  - **fecha entrega:** 7 de Mayo.
  - **resumen:** panel de administración.
  - **requisitos funcionales:** RF-13, RF-14, RF-15, RF-16, RF-17, RF-18.
- **Version 0.5**
  - **fecha entrega:** 13 de Mayo.
  - **resumen:** Equipos. Creación y edición de Tareas. Ejecución en kahan.
  - **requisitos funcionales:** RF-3 y RF-19.
- **Version 0.6**
  - **fecha entrega:** 20 de Mayo.
  - **resumen:** FAQ y Perfil.
  - **requisitos funcionales:** RF-11 y RF-20.

**■ Version 1.0**

- **fecha entrega:** 27 de Mayo.
- **resumen:** versión final con correcciones menores y despliegue en la nube.



---

## CAPÍTULO 6

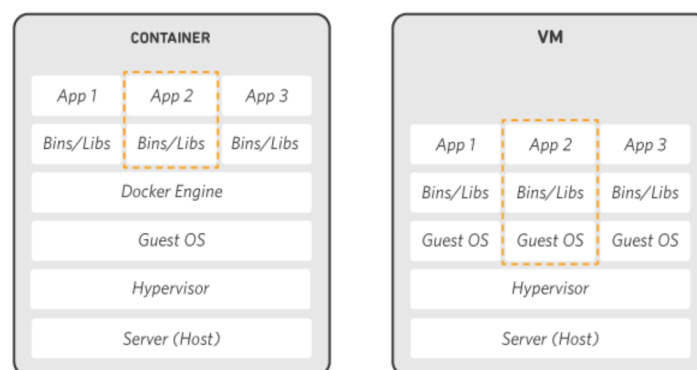
# Implantación

---

*Pizarra* es una herramienta que necesita de otros aplicativos para su funcionamiento: una Base de datos (BD), *Redis* para el sistema de Colas interno de envío de Tareas, un NFS para compartir recursos y *Nginx* para actuar como servidor web y redirigir las peticiones.

En el pasado, la implantación hubiera requerido la intervención del equipo de Administración de Sistemas para instalar el SO de cada máquina, crear usuarios, asignar permisos, abrir puertos y otro sinfín de tareas. Hoy en día es diferente. Para evitar tener que realizar todos estos pasos y acelerar la puesta en producción el proyecto incluye la posibilidad de que todos los elementos de la arquitectura se desplieguen en Contenedores. Estos Contenedores están creados con la tecnología *Docker* y son paquetes que incluyen todo lo necesario, ya sean librerías u otras herramientas del sistema necesarias para que el software se ejecute.

*Docker* ofrece una funcionalidad similar a la de una Máquina Virtual (VM), siendo algo bastante diferente. En primer lugar no es necesario asignar recursos físicos a cada Contenedor ya que es el mismo *Docker Engine* (servicio) el que se encarga de asignarlos de forma dinámica (Figura 6.1) y esto hace que se aproveche mejor el *hardware*; además tienen una huella en espacio en disco mucho menor que el tradicional SO con todos las librerías y servicios que vienen preinstalados ya que solo contienen lo esencial para su funcionamiento. Como ejemplo, el Contenedor de *Redis* que utilizamos ocupa 29.8 MB cuando una VM con el mismo servicio pueden llegar a ser varios Gigabytes.



**Figura 6.1:** Diferencias entre *Docker* y máquina virtual

## 6.1 Docker y Kubernetes

Para nuestro entorno de desarrollo local utilizamos Contenedores públicos como el de *PostgreSQL* que, con un fichero de configuración, permite tener el servicio ejecutándose en cuestión de minutos. A continuación, se muestra un extracto del fichero **docker-compose.yml** del repositorio:

```
version: "3.1"
services:
  postgres:
    image: postgres: 9.6-alpine
    container_name: postgres
    volumes:
      - /data/pizarra/postgres:/var/lib/postgresql/data
    ports:
      - 5432: 5432
    environment:
      - POSTGRES_USER=pizarra
      - POSTGRES_PASSWORD=pizarra
    restart: unless-stopped
...
```

docker-compose.yml

Los parámetros tienen el siguiente uso:

- **version:** versión de la API de *Docker*.
- **services:** listado de servicios a desplegar.
- **image:** Contenedor oficial de *PostgreSQL*<sup>1</sup> sobre Linux Alpine.
- **container\_name:** nombre que se le dará al Contenedor al iniciarse.
- **volumes:** mapeo de directorios de nuestro entorno de desarrollo local al Contenedor. Esto se hace para evitar perder la información de la BD ya que los Contenedores son stateless (no mantienen un estado y al reiniciarse o eliminarse se pierde toda la información).
- **ports:** mapeo de puertos del SO a los Contenedores, esto permite acceder al servicio de *PostgreSQL* por un puerto local.
- **environment:** variables de entorno del SO, en este caso se define un usuario y una contraseña para acceder a la BD.
- **restart:** con el parámetro unless-stopped nuestro Contenedor se reiniciará de forma automática en caso de errores, a menos que se envíe un mensaje de finalización de ejecución.

Con los parámetros previos y siguiendo la documentación de variables de entorno disponibles, al iniciarse el Contenedor se crearán los schemas y directorios necesarios en caso de que no existan y un usuario con privilegios con el nombre y contraseña proporcionados.

<sup>1</sup>Postgres Official Docker Image: [https://hub.docker.com/\\_/postgres](https://hub.docker.com/_/postgres).



Para un entorno de desarrollo local y/o de pruebas *Docker* puede ser una herramienta que cumpla con nuestras necesidades, pero para un entorno de producción es necesario contar con un servicio que realice la implantación de forma automatizada, que sea eficiente en el manejo de recursos y resiliente en caso de problemas. Para esta tarea utilizaremos *Kubernetes*, un *orquestador* de Contenedores que permite auto escalado de aplicaciones, automatización de despliegues y configuración de forma declarativa.

## 6.2 Despliegue en la Nube con Kubernetes

Existen ciertas limitaciones en *cluster* kahan que han impedido la instalación del servicio de *Kubernetes* (K8s). Este era uno de los riesgos contemplados en el trabajo. Para resolver este problema se ha optado por un despliegue de *Pizarra* en la nube.

Existen varios proveedores de K8s en Internet (puede consultarse el listado de todos los *partners* en el sitio oficial de *Kubernetes*<sup>2</sup>) siendo los más importantes: Google Cloud Engine (GCE), Amazon Web Services (AWS) y Microsoft Azure. Se ha optado por desplegar la aplicación en GCE, ya que dispone de buena documentación y, además, \$300 de crédito promocional, que es más que suficiente para tener el aplicativo funcionando durante meses.

### 6.2.1. Pasos de un despliegue en GCE

En primer lugar debemos crear una cuenta en Google o utilizar una existente que nunca ha activado GCE de forma previa ya que el crédito promocional de bienvenida tiene una caducidad de 12 meses desde la activación.

Con nuestra cuenta accedemos a la Consola Web de GCE y creamos nuestro proyecto, en este caso lo llamaremos “Pizarra” (Figura 6.2). Nuestro aplicativo utiliza *Kubernetes Engine* y el *Container Registry* que son visibles desde el menú lateral. Entraremos a cada uno de ellos para habilitar su API.

Una vez creado el proyecto y habilitadas las API’s hay que descargar<sup>3</sup> e inicializar<sup>4</sup> Google Cloud SDK para hacer uso de la consola de comandos de forma local.

Todos los pasos descritos a continuación se ejecutan de forma automática con un script<sup>5</sup> incluido en el repositorio de código. Lo único que se necesita de forma previa es la configuración de algunos parámetros de despliegue como variables de entorno del SO donde se ejecuta. Las variables de entorno necesarias son el ID del proyecto en GCE y las credenciales de la BD que serán guardados como un secreto en *Kubernetes*. Podemos exportarlas antes de lanzar nuestro script ejecutando los siguientes comandos en la consola:

```
$ export PROJECT_ID=pizarra-id
$ export DB_USERNAME=pizarra
$ export DB_PASSWORD=pizarra
```

Estas son las líneas que ejecuta el *script*, donde todos los ficheros que forman parte de cada comando se encuentran en el repositorio<sup>6</sup>:

<sup>2</sup>Kubernetes - Partners: <https://kubernetes.io/es/partners>.

<sup>3</sup>Cómo instalar el SDK de Google Cloud: <https://cloud.google.com/sdk/install?hl=es-419>.

<sup>4</sup>Cómo inicializar el SDK de Cloud: <https://cloud.google.com/sdk/docs/initializing?hl=es-419>.

<sup>5</sup>Script Despliegue GCE: <https://github.com/nimar3/pizarra/blob/master/gce/commands.sh>.

<sup>6</sup>Ficheros de despliegue en GCE: <https://github.com/nimar3/pizarra/tree/master/gce>.

The figure consists of two screenshots of the Google Cloud Platform (GCP) interface. The top screenshot shows the 'Dashboard' page. At the top, there is a blue header with the 'Google Cloud Platform' logo and navigation icons. Below the header, the word 'Dashboard' is displayed. A message box states 'To view this page, select a project.' with an information icon on the left and a red-bordered button labeled 'CREATE PROJECT' on the right. The bottom screenshot shows the 'New Project' page. It has the same blue header. Below it, the word 'New Project' is displayed. There is a form with a 'Project name \*' field containing 'pizarra' and a help icon. Below this, it says 'Project ID: pizarra-279707. It cannot be changed later.' with an 'EDIT' link. There is a 'Location \*' field with a dropdown menu showing 'No organization' and a 'BROWSE' button. At the bottom, there are 'CREATE' and 'CANCEL' buttons.

**Figura 6.2:** Creación de un nuevo proyecto en Google Cloud Engine.

1. Establecimiento del proyecto y zona geográfica donde desplegaremos el aplicativo. Por cercanía geográfica se ha escogido el centro de datos de Países Bajos.

```
$ gcloud config set project ${PROJECT_ID}
$ gcloud config set compute/zone europe-west4
```

2. Credenciales para la BD para ser obtenidas a posterior por los contenedores de *Kubernetes*.

```
$ kubectl create secret generic pizarra-credentials \
--from-literal db_username=${DB_USERNAME} \
--from-literal db_password=${DB_PASSWORD}
```

3. Subida de Contenedores de *Nginx* y *Pizarra* al Registro de GCE. Los otros Contenedores utilizarán imágenes públicas.

```
$ docker push eu.gcr.io/${PROJECT_ID}/pizarra
$ docker push eu.gcr.io/${PROJECT_ID}/nginx
```

4. Creación de en *Kubernetes* con 2 nodos. Una vez creado se descargan las credenciales para trabajar con él.

```
$ gcloud container clusters create pizarra --num-nodes=2
$ gcloud container clusters get-credentials pizarra
```

5. Creación de Almacenamiento de 10 GiB para la persistencia de datos que será utilizado por el NFS. Replicado en varias zonas ya que el almacenamiento no es parte de *Kubernetes Engine*.

```
$ kubectl apply -f storage.yaml
```

6. Creación del servicio de Sistema de archivos de red (NFS).

```
$ kubectl apply -f nfs.yaml
```

7. Creación del servicio de Redis para el sistema de colas local.

```
$ kubectl apply -f redis.yaml
```

8. Creación del servicio de Base de datos.

```
$ kubectl apply -f postgres.yaml
```

9. Despliegue de *Pizarra* y 1 *Worker* que se encargará de encolar las las Tareas en *kahan*.

```
$ kubectl apply -f pizarra.yaml
```

10. Despliegue del servicio de *NGinx* con una IP pública para el acceso de usuarios por Internet.

```
$ kubectl apply -f nginx.yaml
```

Una vez completados todos los pasos se puede obtener la IP pública de *Pizarra* ejecutando el siguiente comando:

```
$ kubectl get services
```

Con esto se ha completado el despliegue y ya podemos navegar a *Pizarra* para comenzar a *interactuar* con el aplicativo.

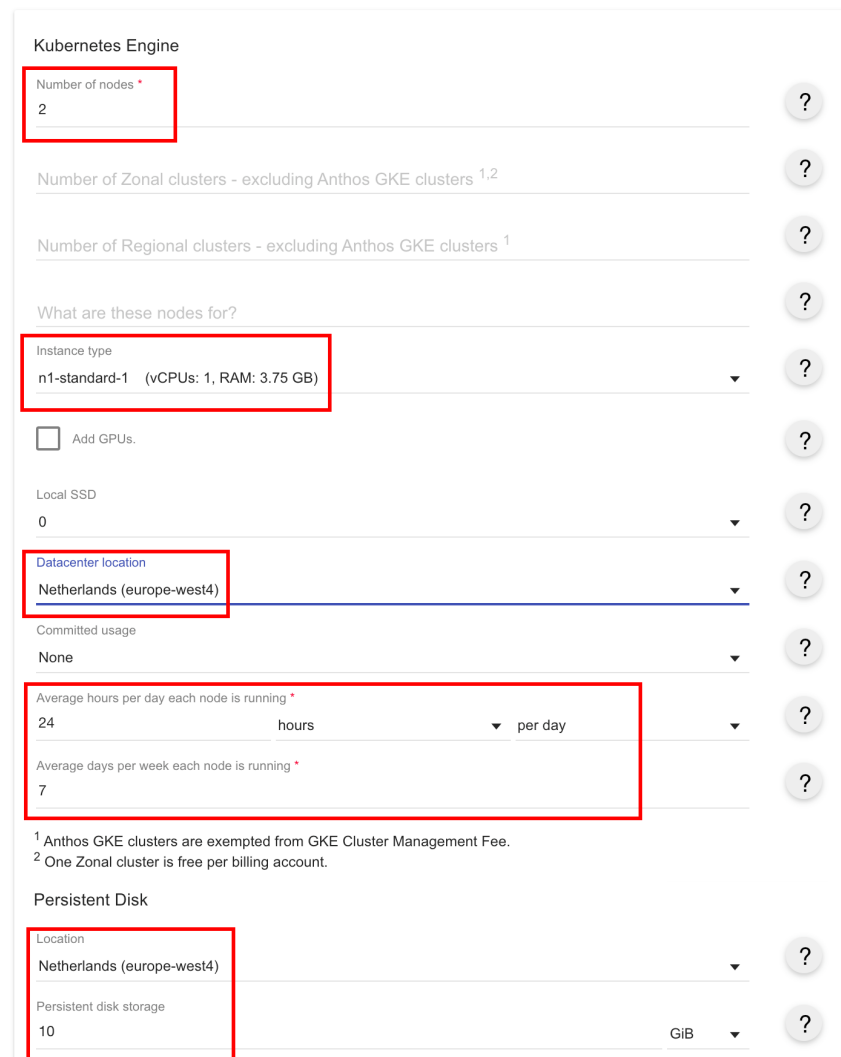
### 6.2.2. Costos asociados

GCE cuenta con una herramienta<sup>7</sup> para estimar el gasto mensual en el que podemos incurrir con la utilización de los recursos contratados. Se ha realizado una estimación con los siguientes parámetros (Figura 6.3):

- Kubernetes:
  - Número de nodos: 2.
  - Tipo de Instancia: *n1-standard-n1* (vCPUs: 1, RAM: 3.75 GB).
  - Centro de Datos: Países Bajos.
  - Uso estipulado: 24 horas al día los 7 días a la semana.
- Persistencia:
  - Tamaño disco: 10 GiB.
  - Centro de Datos: Países Bajos.

El número de nodos es 2 para tener servicio en caso de que haya una caída de uno de los nodos del *cluster*, el centro de datos se ha escogido Países Bajos por cercanía geográfica y una instancia de tipo *n1-standard-n1* que ofrece los recursos mínimos necesarios para los contenedores a lanzar. El disco es compartido entre la BD y *Pizarra* con suficiente margen para evitar problemas de espacio.

<sup>7</sup>Calculadora de precios de Google: <https://cloud.google.com/products/calculator/>.



**Kubernetes Engine**

Number of nodes \*  
2

Number of Zonal clusters - excluding Anthos GKE clusters <sup>1,2</sup>

Number of Regional clusters - excluding Anthos GKE clusters <sup>1</sup>

What are these nodes for?

Instance type  
n1-standard-1 (vCPUs: 1, RAM: 3.75 GB)

☐ Add GPUs.

Local SSD  
0

Datacenter location  
Netherlands (europe-west4)

Committed usage  
None

Average hours per day each node is running \*  
24 hours per day

Average days per week each node is running \*  
7

<sup>1</sup> Anthos GKE clusters are exempted from GKE Cluster Management Fee.  
<sup>2</sup> One Zonal cluster is free per billing account.

**Persistent Disk**

Location  
Netherlands (europe-west4)

Persistent disk storage  
10 GiB

**Figura 6.3:** Configuración de la calculadora de precios para el despliegue en Kubernetes de Pizarra.

Utilizando la calculadora da un costo aproximado de €49.00 mensuales. Como dato adicional, se puede variar de Centro de Datos y otros parámetros como el uso estipulado para reducir los costos aún más. Podemos guardar la estimación o enviárnosla por correo electrónico (Figura 6.4).

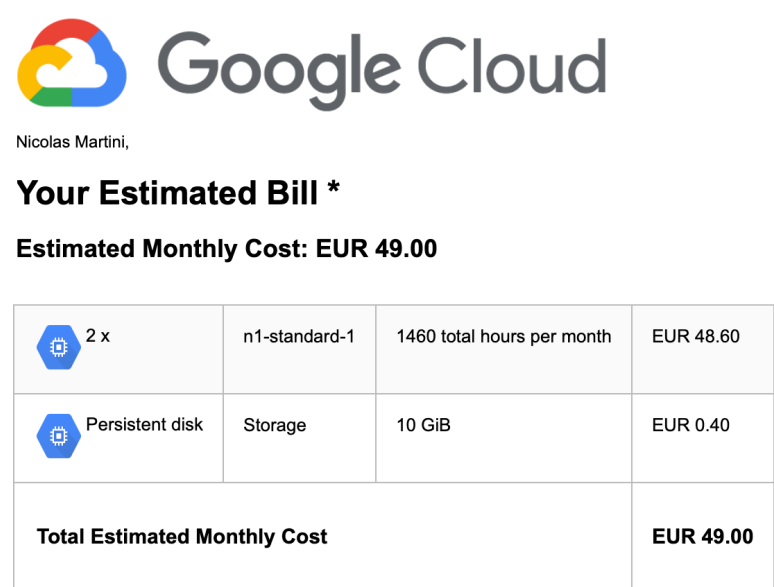


Figura 6.4: Ejemplo de correo de GCE con una estimación de los costos asociados por mes.



---

---

## CAPÍTULO 7

# Pruebas

---





---

## CAPÍTULO 8

# Mantenimiento

---

El desarrollo de *software* es un proceso evolutivo y requiere de un mantenimiento continuo para asegurar el correcto funcionamiento, como así también la posibilidad de añadir mejoras y nuevas funcionalidades. Para poder asegurar este proceso tienen que cumplirse ciertas pautas como el acceso al código fuente, en este caso disponible en un repositorio de código abierto en Internet, una guía de como configurar un entorno de desarrollo local, el proceso a seguir para mantener un estándar de calidad en el código fuente y el manejo de versiones. Los siguientes apartados abordan todos estos puntos para cumplir con el proceso de mantenimiento de *software*.

### 8.1 Entorno local de desarrollo

---

Para configurar un entorno de desarrollo local se debe cumplir los siguientes requisitos.

- Tener instalado *python3*<sup>1</sup> y *Docker*<sup>2</sup> en nuestra máquina
- Clonado el repositorio de *Pizarra* realizando los siguientes pasos:

1. Creación de directorio para el proyecto

```
$ mkdir pizarra
$ cd pizarra/
```

2. Inicialización de un nuevo repositorio y añadida la dirección remota

```
$ git init .
Initialized empty Git repository in /Users/nmartini
/pizarra/.git/
$ git remote add upstream git@github.com:nimar3/
pizarra.git
```

3. Descarga del código fuente

```
$ git fetch upstream
remote: Enumerating objects: 589, done.
remote: Counting objects: 100% (589/589), done.
remote: Compressing objects: 100% (364/364), done.
```

---

<sup>1</sup>Descarga oficial de Python: <https://www.python.org/downloads/>.

<sup>2</sup>Descarga oficial de Docker: <https://docs.docker.com/get-docker/>.

```
remote: Total 6891 (delta 396), reused 393 (delta
221), pack-reused 6302
Receiving objects: 100% (6891/6891), 19.14 MiB |
675.00 KiB/s, done.
Resolving deltas: 100% (1748/1748), done.
```

4. *checkout* a la rama donde se va a trabajar, en este caso *master*

```
$ git checkout master
Checking out files: 100% (4858/4858), done.
Branch 'master' set up to track remote branch '
master' from 'upstream'.
Already on 'master'
```

- Creación de un entorno virtual e instalación de todos los Paquetes de *python* necesarios.

```
$ python3 -m venv env
$ source env/bin/activate
(env) $ pip3 install -r requirements_dev.txt
```

Si hemos realizado cambios en el código fuente se debe ejecutar el siguiente comando para generar una nueva versión de nuestro Contenedor de *Docker* de *Pizarra*:

```
$ docker build -t pizarra .
```

Por último, para iniciar *Pizarra* con todos los componentes introducimos el siguiente comando en la raíz del repositorio. Todos los componentes deben mostrar que se han creado correctamente con el mensaje *done*.

```
(env) $ docker-compose up
Creating network "pizarra_default" with the default driver
Creating redis ... done
Creating postgres ... done
Creating pizarra ... done
Creating worker ... done
Creating nginx ... done
```

Se puede acceder a la aplicación web navegando a la URL <https://127.0.0.1/> y se redirigirá automáticamente al inicio de sesión (Figura 8.1)

En algunos casos sólo es necesario la ejecución del aplicativo web con una BD local en SQLite para hacer pruebas sobre cambios que no afectan otros componentes. Con este tipo de ejecución solo se podrá acceder a la aplicación web sin ningún *Worker* o el servicio de *Redis*, en resumen, se está ejecutando solamente el *script* de la aplicación en *Flask* y será útil para visualizar modificaciones a nivel visual.

Esto se puede hacer utilizando el siguiente comando y *Pizarra* estará disponible en la URL <http://127.0.0.1:5000/> (el puerto por defecto de *Flask* es el 5000).

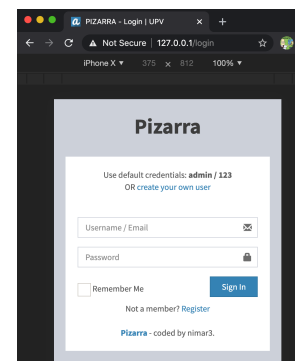


Figura 8.1: Inicio de sesión en Pizarra.

```
(env) $ python3 run.py
* Serving Flask app "app" (lazy loading)
* Running on http://127.0.0.1:5000/ (Press
  CTRL+C to quit)
```

## 8.2 Depuración de errores

Como todo software, *Pizarra* no estará exento de errores o *bugs* que no se han identificado en la etapa de desarrollo. Para aplacar estos problemas *python* ofrece la ejecución de *scripts* en modo *Debug*, lo que nos permite poner puntos de ruptura y hacer una ejecución “paso a paso” del código para identificar las causas de errores.

En el entorno de desarrollo local el modo *Debug* siempre está habilitado pero si queremos habilitarlo cuando ejecutamos el aplicativo en modo *Production* debemos especificarlo explícitamente en las variables de entorno del SO, en este caso y para más información revisar el Apéndice A: Configuración del Sistema.

## 8.3 Calidad de código

Crear código de calidad es una tarea compleja y si en un futuro se espera colaboración externa en *Pizarra* se debe seguir una serie de pautas para que todos puedan trabajar sin tener que descifrar lo que se ha hecho. En estos casos lo que se hace es seguir unas guías de estilo que estipulan ciertas reglas de cómo debe escribirse el código fuente y hay herramientas que ayudan a identificar problemas como Paquete *flake8*<sup>3</sup>.

En el directorio raíz del repositorio de código se ejecuta el siguiente comando y podemos ver un listado de reglas que no se cumplen y debemos solucionar:

```
$ flake8 app/
app/__init__.py:8:1: F401 'logging.DEBUG' imported but unused
app/__init__.py:71:5: E722 do not use bare 'except'
app/home/routes.py:59:101: E501 line too long (106 > 100
  characters)
app/home/routes.py:210:5: E722 do not use bare 'except'
app/home/routes.py:216:101: E501 line too long (108 > 100
  characters)
app/admin/forms.py:32:28: N805 first argument of a method
  should be named 'self'
```

Adicionalmente, la configuración de *flake8* se encuentra en el fichero **setup.tfg** en la raíz del repositorio en caso que se desee modificar la forma en la cual se ejecuta.

```
[flake8]
max-line-length=100
ignore=E402,E266
exclude=./migrations
```

setup.tfg

<sup>3</sup>Sitio Web de Flake8: <https://flake8.pycqa.org/en/latest/>

---

## 8.4 Gestión de versiones

---

Como se ha visto anteriormente desde el inicio del desarrollo de la aplicación se ha utilizado como herramienta de gestión de versiones *Git*<sup>4</sup> para el seguimiento de los cambios y entregables. La rama principal es “master” y solo para requisitos de gran impacto se han creado nuevas que finalmente han sido fusionadas a “master” una vez validadas. Cada vez que se producía el fin de un *sprint* con su bloque de tareas se creaba una nueva “etiqueta” o *tag* marcando la versión; comenzando desde la 0.1 llegando al final de la entrega del TFG a la 1.0.

Este ha sido un trabajo de una sola persona y se ha seguido el método de *Feature Branch Workflow* [14] donde se tiene una rama principal y nuevas funcionalidades o *features* divergen de esta para luego volver a fusionarse. Ya que se espera recibir colaboraciones a futuro se deberá acomodar a un proceso que involucre a un equipo de trabajo y ramas en diferentes estados como *dev*, *release* y *master* como en el método *Gitflow Workflow*.

---

<sup>4</sup>Sitio web Git: <https://git-scm.com/>.

---

## CAPÍTULO 9

# Extensibilidad

---

Uno de los puntos más importantes es permitir la colaboración y que *Pizarra* se pueda utilizar otros sistemas de tareas que no sean una Cola local o la de kahan (TORQUE<sup>1</sup>); ya sabiendo que en el próximo curso académico kahan será reemplazado por *kahan2* y deberá ser adecuado. Además, la colaboración y uso no debe estar limitado por el lenguaje y debe estar abierto a ser localizado en diferentes idiomas. En este capítulo trataremos estos dos apartados al ser los más importantes.

### 9.1 Otros sistemas de Gestión de Tareas

---

En el capítulo 5: Ejecución en una Cola interna ya se ha explicado el proceso de desarrollo y creación de un autómata para los pasos que seguiría en Envío de una Tarea. En modo de ejemplo se había presentado una versión simplificada del autómata para un *job* local (clase *LocalJob*) y para este tipo de “trabajos” la definición es muy simple ya que este tipo de *jobs* no deben esperar su turno para ser procesados como en kahan.

Para mostrar la facilidad con la cual se puede extender *Pizarra* a otros sistemas de Colas se hará un repaso de como se ha definido el nuevo autómata para la clase *KahanJob* que extiende de *LocalJob*.

Para el sistema de Colas de kahan podemos ver el siguiente extracto donde se han creado nuevos métodos para obtener el ID con el que se ha desplegado la tarea y también la verificación de como si la tarea está todavía en la Cola o ejecutandose debemos esperar (*StepResult.WAIT*) y volver a verificar en un futuro.

```
...
def deploy(self):
    remote = RemoteClient.Instance()
    output = remote.execute_commands_in_directory(self.
        remote_directory, ['qsub script.sh'])
    kahan_queue_id = get_kahan_queue_id(output)
    if kahan_queue_id is None:
        return StepResult.NOK

    self.user_request.kahan_id = kahan_queue_id
    return StepResult.OK
```

---

<sup>1</sup>Sitio web de TORQUE: <https://adaptivecomputing.com/cherry-services/torque-resource-manager/>.

```

def check_queue(self):
    remote = RemoteClient.Instance()

    output = remote.execute_commands(['qstat |grep ' + self.
        user_request.kahan_id])
    request_status = get_kahan_queue_status(output)

    if request_status == RequestStatus.QUEUED or request_status
        == RequestStatus.RUNNING:
        if request_status != self.user_request.status:
            self.update_request(request_status)
        # requeue task
        return StepResult.WAIT
    ...

```

models\_jobs.py

Al comenzar la ejecución de los métodos *deploy* y *check\_queue* se obtiene una instancia de conexión con kahan; esto se hace utilizando un patrón de diseño llamado *Singleton* [6] para evitar abrir de múltiples conexiones *SSH* con el servidor frontal del *cluster*. El método *deploy* ejecuta el comando “*qsub script.sh*” para encolar el trabajo y obtiene el ID de la Cola. En cambio, *check\_queue* ejecuta el comando “*qstat*” para ver el estado del trabajo en la Cola y compara el estado, si sigue estando encolado (*QUEUED*) o ejecutandose (*RUNNING*) habrá que volver a revisar en un futuro.

Estos nuevos estados del autómata que difieren de la clase *LocalJob* se pueden ver en la definición de la clase *KahanJob*:

```

class KahanJob(LocalJob):
    def __init__(self, user_request):
        super().__init__(user_request)
        self.task_process = {
            RequestStatus.CREATED: {
                'f': self.start,
                'steps': {
                    StepResult.OK: RequestStatus.VERIFYING,
                }
            },
            RequestStatus.VERIFYING: {
                'f': self.verify,
                'steps': {
                    StepResult.OK: RequestStatus.COMPILING,
                    StepResult.NOK: RequestStatus.ERROR
                }
            },
            RequestStatus.COMPILING: {
                'f': self.compile,
                'steps': {
                    StepResult.OK: RequestStatus.DEPLOYING,
                    StepResult.NOK: RequestStatus.ERROR
                }
            },
            RequestStatus.DEPLOYING: {

```

```

        'f': self.deploy,
        'steps': {
            StepResult.OK: RequestStatus.QUEUED,
            StepResult.NOK: RequestStatus.ERROR
        }
    },
    ...

```

models\_jobs.py

## 9.2 Localización

*Pizarra* utiliza el Paquete *Babel*<sup>2</sup> para ofrecer los diferentes idiomas disponibles a los usuarios. La configuración de *Babel* se encuentra en el fichero **babel.cfg**<sup>3</sup> en la que se define los ficheros a escanear y extraer texto para traducir.

```

[python: app/**/*.py]
[jinja2: app/**/*.templates/**/*.html]
extensions=jinja2.ext.autoescape,jinja2.ext.with_

```

A continuación se muestra el “paso a paso” para añadir un nuevo lenguaje a la aplicación.

1. Con el siguiente comando se extrae el texto y mientras procesan los ficheros se van mostrando mensajes en la consola. Al finalizar el resultado lo tendremos disponible en el fichero **messages.pot**.

```

(env) $ pybabel extract -F babel.cfg -k _l -o messages.
      pot .
...
extracting messages from app/home/templates/macros/
      summaries.html (extensions="jinja2.ext.autoescape,
      jinja2.ext.with_")
extracting messages from app/home/templates/macros/teams
      .html (extensions="jinja2.ext.autoescape,jinja2.ext.
      with_")
writing PO template file to messages.pot

```

2. Una vez extraído el texto hay que generar el fichero de localización para el idioma que vayamos a traducir. Las localizaciones se encuentran en el directorio */app/translations*.

```

(env) $ pybabel init -i messages.pot -d app/translations
      -l es
creating catalog app/translations/es/LC_MESSAGES/
      messages.po based on messages.pot

```

3. El siguiente paso es comenzar a añadir las traducciones en las líneas identificadas con *msgstr*.

<sup>2</sup>Paquete *Babel*: <http://babel.pocoo.org/en/latest/>.

<sup>3</sup>Configuración de *Babel*: <https://github.com/nimar3/pizarra/blob/master/babel.cfg>.

```
$ cat app/translations/es/LC_MESSAGES/messages.po
...
#: app/account/routes.py:34
msgid "New Access Token has been generated!"
msgstr "Nuevo Token de acceso ha sido generado!"

#: app/account/routes.py:54
msgid "You joined Team {}"
msgstr "Te has unido al equipo {}"

#: app/account/routes.py:56
msgid "You are unable to join team {} because is full"
msgstr ""
...
```

4. Se compilan las traducciones.

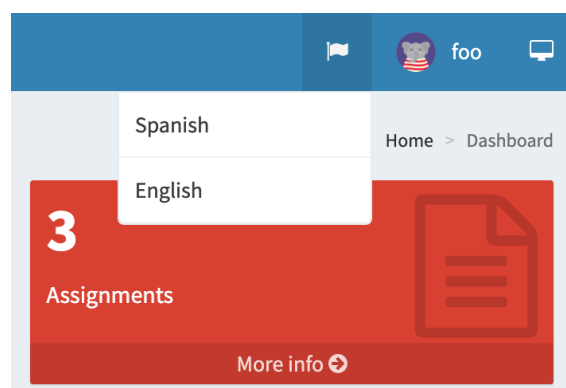
```
1 (env) $ pybabel compile -d app/translations
2 compiling catalog app/translations/es/LC_MESSAGES/messages.po to app/
  translations/es/LC_MESSAGES/messages.mo
```

5. Y se añade a la configuración de Pizarra el nuevo idioma disponible.

```
...
SUPPORTED_LANGUAGES = {'es': 'Spanish', 'en': 'English'
                        '}
...
```

config.py

Al reiniciar la aplicación tendremos en el menú se visualizará el nuevo idioma disponible y al seleccionarlo el lenguaje de la aplicación web se mantendrá durante la sesión que tengamos activa.



**Figura 9.1:** Selector de Idiomas en Pizarra.



---

---

## CAPÍTULO 10

# Conclusiones

---

...

### 10.1 Relación del trabajo desarrollado con los estudios cursados

---

?

### 10.2 C

---

competencias transversales

?



---

---

## CAPÍTULO 11

# Trabajos futuros

---

?



# Bibliografía

---

- [1] del Águila Cano, Isabel María. "Documentación de los requisitos". *Ingeniería de requisitos: Material didáctico. Cuaderno de teoría*. Almería, Editorial Universidad de Almería, 2019, pp 67-82. ISBN: 9788417261795
- [2] Amo, F. Alonso. Martínez Normand, Loïc A. Segovia Pérez, Francisco Javier. "El ciclo de vida clásico". *Introducción a la Ingeniería del software*. Madrid: Delta Publicaciones, 2005, pp. 77-91. ISBN: 8496477002
- [3] Bourne, John R. "Tools for Building Applications: The Model-View-Controller Paradigm and View Components". *Object-Oriented Engineering: Building Engineering Systems Using Smalltalk-80* United States of America: CRC Press, 1992. pp. 175-216 .ISBN: 9780256112108
- [4] Stouffer, Jack. "Creating Controllers with Blueprints". *Mastering Flask* Birmingham: Pack Publishing Ltd., 2015, pp. 57-62. ISBN: 9781784391928
- [5] Grinberg, Miguel "Deployment". *Flask Web Development: Developing Web Applications with Python* California: O'Reilly Media, Inc., 2014. ISBN: 9781491947616
- [6] Gamma, E. and Helm, R. and Johnson, R. and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software* Pearson Education, 1994. ISBN: 9780321700698
- [7] Junta de Andalucía. *Plantilla Especificación de Requisitos* [en línea] Consultado en 10 Mayo 2020. <http://www.juntadeandalucia.es/servicios/madeja/contenido/recurso/456>.
- [8] Jose E. Roman. *Clúster de Computación "Kahan"* [en línea] Consultado en 12 Abril 2020. <http://personales.upv.es/jroman/kahan.html>
- [9] Google. *Getting Started with Google Kubernetes Engine* [en línea] Consultado en 1 Junio 2020. <https://www.coursera.org/learn/google-kubernetes-engine>
- [10] Pallets *Standalone WSGI Containers* [en línea] Consultado en 20 Mayo 2020. <https://flask.palletsprojects.com/en/1.1.x/deploying/wsgi-standalone/>
- [11] Jacobs, Todd A. *Sharing Docker Containers across DevOps Environments* [en línea] Consultado en 2 Mayo 2020. <https://www.linuxjournal.com/content/sharing-docker-containers-across-devops-environments>
- [12] Grupo Trasgo *Tablón FAQ* [en línea] Consultado en 5 Marzo 2020. <http://frontendv.infor.uva.es/faq>
- [13] McIntyre, Spencer. *Rule Engine Documentation* [en línea] Consultado en 3 Junio 2020. <https://zerosteiner.github.io/rule-engine/>
- [14] Atlassian. *Comparing Workflows* [en línea] Consultado en 1 Marzo 2020. <https://www.atlassian.com/git/tutorials/comparing-workflows>



---

## APÉNDICE A

# Configuración del sistema

---

Para un correcto funcionamiento de Pizarra el sistema debe inicializarse siguiendo los pasos descritos a continuación. Aunque se dispone de un script que realiza el despliegue de forma automatizada hay algunos pasos que requieren de intervención manual.

### A.1 Inicialización

---

Antes de comenzar con la inicialización del sistema, debemos exportar las variables de entorno del SO que utiliza el script. Abrimos una consola y ejecutamos los siguientes comandos, donde *pizarra-id* es el ID de proyecto en Google Cloud Engine (GCE).

```
1 $ export PROJECT_ID=pizarra-id
2 $ export DB_USERNAME=pizarra
3 $ export DB_PASSWORD=pizarra
```

Si quisiéramos cambiar algún parámetro por defecto de Pizarra, debemos modificar el fichero **pizarra.yaml**<sup>1</sup> añadiendo los nuevos parámetros. En el siguiente extracto se han cambiado los puntos de penalización al obtener un *KO* y *TIMEWALL* en el envío de una Tarea y la carga de datos de ejemplo.

```
1 ...
2 spec:
3   containers:
4     - image: eu.gcr.io/pizarra-279100/pizarra
5       name: pizarra
6       env:
7         - name: TIMEWALL_PENALTY
8           value: "-20"
9         - name: "KO_PENALTY"
10          value: "-30"
11         - name: "IMPORT_SAMPLE_DATA"
12          value: "True"
13 ...
```

A continuación se ejecuta el script.

```
1 $ sh commands.sh
```

---

<sup>1</sup>Fichero YAML de Pizarra: <https://github.com/nimar3/pizarra/blob/master/gce/pizarra.yaml>.

Al terminar la ejecución deberíamos tener Pizarra con todos los componentes desplegados. Podemos obtener el estado de los contenedores y servicios con los siguientes comandos.

```
1 $ kubectl get pods
2 $ kubectl get services
```

## A.2 Parámetros

Al iniciar un Contenedor utilizamos variables de entorno del SO para configurar su funcionamiento. El Contenedor de Pizarra soporta estas variables para dar diferentes posibilidades en cómo queremos que se comporte el aplicativo.

Los mencionados a continuación con su descripción y opciones son los más relevantes en un despliegue. También se listan otros parámetros adicionales como referencia. Todos estos pueden ser consultados en el fichero de configuración<sup>2</sup>.

### ■ APP\_MODE

- Descripción: ejecución del aplicativo web o un worker
- Opciones
  - Pizarra: aplicativo web
  - Worker: worker
- Valor por defecto: *Pizarra*

### ■ CONFIG\_MODE

- Descripción: modo en el cual queremos que se ejecute Pizarra
- Opciones
  - Debug: modo Debug habilitado y Base de datos (BD) SQLite
  - Production: modo Debug deshabilitado y Base de datos (BD) Postgres
- Valor por defecto: *Debug*

### ■ REMOTE\_HOST

- Descripción: hostname o IP del sistema externo a conectarnos para las colas de kahan
- Valor por defecto: *kahan.dsic.upv.es*

### ■ REMOTE\_USER

- Descripción: usuario con el que nos conectaremos por SSH y SCP a kahan
- Valor por defecto: *pizarra*

### ■ REMOTE\_PATH

- Descripción: directorio remoto donde se copiarán los ficheros de cada ejecución en kahan
- Valor por defecto: */pizarra*

<sup>2</sup>Fichero de Configuración: <https://github.com/nimar3/pizarra/blob/master/config.py>.



**■ SSH\_FILE\_PATH**

- Descripción: ubicación de la clave privada para conectarnos por SSH y SCP a kahan
- Valor por defecto: *app/data/keys/id\_rsa*

**■ LOG\_LEVEL**

- Descripción: establece el nivel de mensajes que deben mostrarse al ejecutarse el aplicativo
- Opciones
  - CRITICAL
  - ERROR
  - WARNING
  - INFO
  - DEBUG
- Valor por defecto: *INFO*

**■ SECRET\_KEY**

- Descripción: clave secreta con la que se encriptarán las contraseñas, no se puede cambiar una vez desplegada la aplicación y por seguridad nunca debe utilizarse el valor por defecto
- Valor por defecto: *pizarra-app*

**■ IMPORT\_SAMPLE\_DATA**

- Descripción: borrado de información de la Base de datos e importación de data de ejemplo.
- Valor por defecto: *False*

**■ TIME\_BETWEEN\_REQUESTS**

- Descripción: tiempo mínimo en segundos que debe esperar un Alumno entre cada envío de Tareas
- Valor por defecto: *60*

**■ TEAM\_MAX\_SIZE**

- Descripción: tamaño máximo de Alumnos que puede tener un Equipo
- Valor por defecto: *3*

**■ REGISTRATION\_ENABLED**

- Descripción: habilita el registro de usuarios en el aplicativo
- Valor por defecto: *False*

**■ TIMEWALL\_PENALTY**

- Descripción: puntos de penalización al obtener un resultado de *TIMEWALL* en un envío de Tarea

- Valor por defecto: *-10*
- **KO\_PENALTY**
  - Descripción: puntos de penalización al obtener un resultado de *KO* en un envío de Tarea
  - Valor por defecto: *-15*
- **DEBUG**
  - Descripción: inicia el aplicativo en modo Debug, útil para identificar errores.
  - Valor por defecto: *False*

### Parámetros adicionales

- SQLALCHEMY\_DATABASE\_URI
- SQLALCHEMY\_TRACK\_MODIFICATIONS
- JSONIFY\_PRETTYPRINT\_REGULAR
- SUPPORTED\_LANGUAGES
- BABEL\_DEFAULT\_LOCALE
- BABEL\_DEFAULT\_TIMEZONE
- UPLOAD\_FOLDER
- FILE\_ALLOWED\_EXTENSIONS
- MAX\_CONTENT\_LENGTH
- TIMEWALL
- FORBIDDEN\_CODE
- RQ\_DASHBOARD\_REDIS\_URL
- QUEUES
- COMPILER

# Acrónimos

---

- API** Interfaz de programación de aplicaciones. 29, 51, 53
- BD** Base de datos. 15, 16, 19–21, 30, 48, 49, 51, 53
- CPA** Computación Paralela. 1, 7, 13, 51, 53
- CPU** Unidad central de procesamiento. 51, 53
- DSIC** Departamento de Sistemas Informáticos y Computación. 2, 23, 51, 53
- ETSINF** Escola Tècnica Superior d'Enginyeria Informàtica. 1, 51, 53
- FIFO** Primero en entrar, primero en salir. 51, 53
- GCE** Google Cloud Engine. 29–31, 47, 51, 53
- GiB** Gibibyte. 30, 51, 53
- GII** Grado de Ingeniería Informática. 1, 51, 53
- K8s** Kubernetes. 51, 53
- LPP** Lenguajes y Entornos de la Programación Paralela. 1, 2, 7, 13, 51, 53
- MVC** Modelo Vista Controlador. 18, 51, 53
- NFS** Sistema de archivos de red. 15, 27, 30, 51, 53
- RAM** Memoria de acceso aleatorio. 51, 53
- RF** Requisito funcional. 51, 53
- RNF** Requisito no funcional. 51, 53
- SCP** Secure Copy. 17, 48, 49, 51, 53
- SO** Sistema operativo. 27, 29, 35, 47, 48, 51, 53
- SSH** Secure Shell. 17, 48, 49, 51, 53
- TFG** Trabajo Final de Grado. 2, 7, 51, 53
- UPV** Universitat Politècnica de València. 13, 51, 53
- vCPUs** Unidades centrales de procesamiento virtuales. 51, 53



# Términos

---

- Administrador** Usuario con un rol que le permite administrar el sistema, normalmente será el profesor de un curso académico. 7–9, 51, 53
- Alumno** Estudiante que forma parte de un curso académico. 7–9, 21, 49, 51, 53
- Clase** Plantilla para la creación de objetos de datos según un modelo predeterminado. 51, 53
- Cola** Estructura de datos que sigue una filosofía FIFO. 15, 17, 19, 21, 23, 24, 37, 51, 53
- Contenedor** imagen ejecutable ligera y portátil que contiene el software y todas sus dependencias. 27, 30, 34, 48, 51, 53
- Entorno de producción** lugar donde se ejecuta el aplicativo para los usuarios finales. 51, 53
- Envío** Petición web que incluye el código fuente a compilar y ejecutar como propuesta de solución a una Tarea. 21, 51, 53
- Equipo** Grupo de estudiantes que pertenecen a una misma clase en un grupo académico. 7, 9, 49, 51, 53
- Equipo de Estudiantes** Grupo de uno o más estudiantes formado para resolver Tareas de forma conjunta. 51, 53
- formula** A mathematical expression. 51, 53
- GCD** Greatest Common Divisor. 51, 53
- Grupo** Agrupación de alumnos que pertenecen a un grupo académico. 7–9, 19, 21, 51, 53
- Grupo de Estudiantes** Clase a la que pertenece el estudiante para el año lectivo. 51, 53
- Insignia** Trofeo o Marcador que se obtiene al realizar cierta acción en la aplicación. 8, 17, 19, 51, 53
- latex** Is a mark up language specially suited for scientific documents. 51, 53
- LCM** Least Common Multiple. 51, 53
- mathematics** Mathematics is what mathematicians do. 51, 53
- Paquete** Carpeta que contiene varios módulos de Python. 15, 16, 19, 34, 35, 38, 51, 53
- Tarea** Problema a resolver asignado a un estudiante o equipo. 8, 15, 17, 19, 21, 31, 47, 49–51, 53