

Δ chan

Proyecto fin de ciclo
Desarrollo de Aplicaciones Multiplataforma

Alfredo Rodríguez Gracia

21 de mayo de 2021

última revisión
6 de junio de 2021

Índice

1. Descripción del proyecto	2
1.1. Partes del proyecto	4
2. Ámbito de implantación	5
3. Recursos de <i>hardware</i> y <i>software</i>	6
3.1. Requisitos de desarrollo	6
3.2. Requisitos de despliegue	7
3.3. Requisitos de instalación por parte del usuario	8
4. Temporalización del desarrollo	8
5. Descripción de los datos base y resultados	10
5.1. Base de datos	10
5.2. <i>API</i>	12
6. Relación entre dispositivos y programa o rutinas	15
6.1. Visualización del contenido de un <i>board</i>	15
6.2. Creación de un nuevo <i>post</i> en un <i>thread</i> (<i>reply</i>)	18
Ejemplos de código	21
Referencias	22

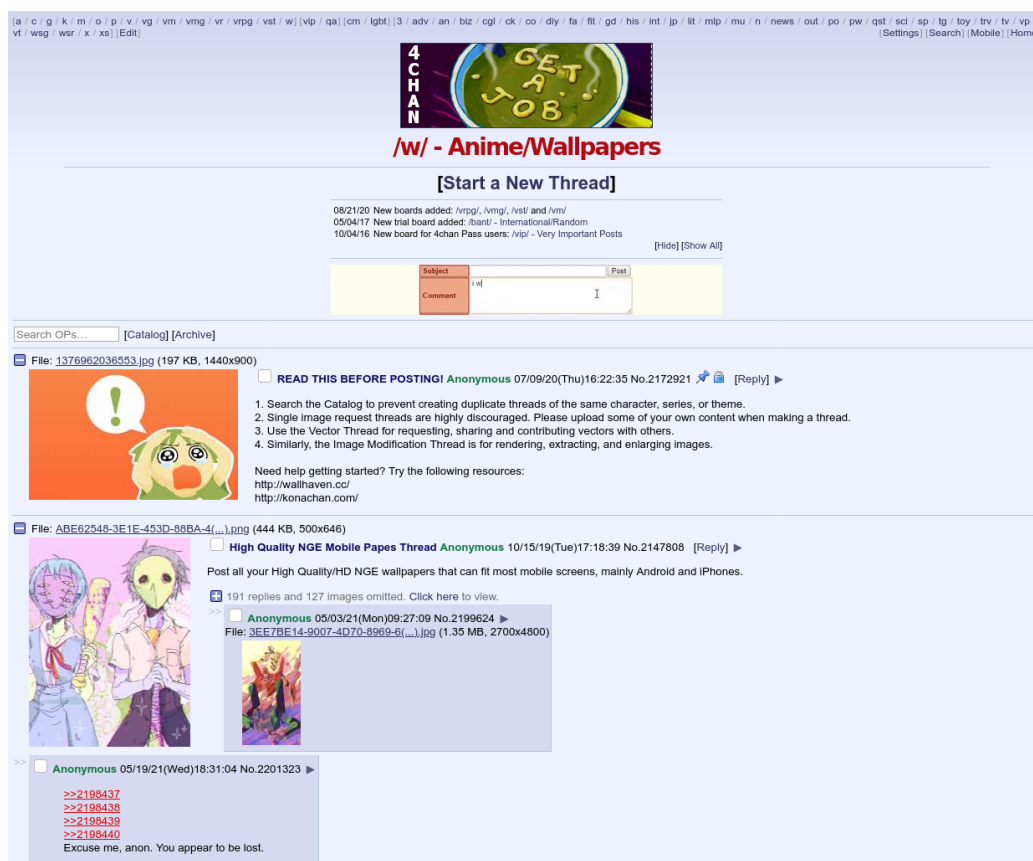


Obra bajo licencia de *Creative Commons* Reconocimiento 4.0 Internacional.

1. Descripción del proyecto

Δ chan (pronunciado como *dichan*) es un proyecto de tablón de imágenes [1], centrado en el anonimato y la libertad de expresión *on-line*, dónde los usuarios pueden subir imágenes y vídeos cortos para iniciar un debate. Está inspirado en otros tablonos existentes como *4chan* y *2channel*, sitios que, a pesar del enorme auge de las redes sociales, siguen siendo el refugio de muchos internautas hoy en día.

Figura 1: Vista del board */w/* de *4chan*



Un tablón de imágenes (también conocido por su nombre en inglés: *image-board*) es un tipo de página web anónima donde la publicación de imágenes y pequeños vídeos cobra una gran importancia. Los primeros tablonos de imágenes fueron creados en Japón a finales de los 90, y se basan en el concepto de los foros de texto. En términos generales ambos comparten la misma estructura, incluyendo la separación de los debates (*threads*) de diferentes temáticas en secciones, llamadas tablonos o *boards*. Sin embargo, los *threads* en

los *imageboards* pueden llegar a ser mucho más esporádicos que en los foros convencionales, donde el tiempo de vida de uno puede ser inferior a varias horas. Los tableros de imágenes más populares en occidente tienden a estar relacionados en su mayoría con la cultura japonesa, como son la temática del *anime* y *manga*. Sin embargo, en Japón son más populares y sus tópicos abarcan una gran variedad de temas.

El proyecto Δ chan intenta emular a estos tableros haciendo muy sencillo que cualquiera que lo desee pueda, incluso con muy pocos recursos, montar su propia instancia en un equipo. La estructura de la página es muy simple, consta principalmente de dos partes bien diferenciadas: la portada (*home*), donde se visualizará la lista de *boards* activos en la página; y los *boards* en sí, cada uno de su temática particular y limitado a nueve páginas de contenido. Cada página de un *board* contendrá cinco *threads* ordenados por fecha de actualización más reciente, es decir, en el primer puesto de la primera página se colocará el *thread* que ha recibido el último comentario y en el último puesto de la novena página estará el *thread* que ha pasado más tiempo sin comentarios. En el momento que un usuario decida abrir un nuevo *thread* ese último se borrará y el nuevo aparecerá en el primer puesto (en la figura 2 se describe el recorrido habitual que realizaría un usuario en la web). De esta forma se consigue ese dinamismo tan característico de los *imageboards* donde tienes la certeza de que lo primero que ves al entrar es de lo que se está hablando actualmente, es el tema del momento.

Figura 2: Recorrido del usuario para acceder al contenido que desea



La intención del proyecto mira hacia un futuro colaborativo, donde muchas personas puedan aportar sus opiniones y mejoras al mismo. Este es el motivo por el que se publica bajo la *GNU General Public License version 3* [2], para garantizar que forme parte del movimiento del *software libre* definido por Stallman [3]. El código fuente será accesible desde un repositorio *Git* de libre acceso, donde cualquier persona podrá proponer cambios a través de los procedimientos establecidos. De esta forma también es posible que surjan *forks* [4] donde, utilizando este proyecto como base, cualquiera puede cambiar radicalmente el propósito original y aportar un enfoque completamente diferente.

1.1. Partes del proyecto

El proyecto consta de tres partes muy bien diferenciadas: la **página web**, la cual constituye el *frontend*, desarrollada utilizando *HTML5*, *CSS3* y *ECMAScript 6* (*JavaScript*), es la encargada de lidiar cara a cara con el usuario final, mostrar los resultados de las llamadas a la *API* y distribuir esas respuestas adecuadamente en la interfaz gráfica (*GUI*); la **API**, una de las dos partes del *backend*, desarrollada utilizando el lenguaje *Python* utilizando la librería *Flask*, es la intermediaria entre la web y los datos almacenados, ofreciendo un estándar a la hora de consultar y modificar la información ofrecida por los usuarios; y la segunda parte del *backend* es la **base de datos** que, corriendo sobre un servidor *MariaDB*, almacenará toda la información de la aplicación en tablas relacionadas para poder acceder a ella de la manera más eficiente posible, garantizando siempre la integridad y la alta disponibilidad de los datos. Observando la figura 8 se puede apreciar de forma esquemática cómo se relacionan entre sí estas tres partes.

Figura 3: Vista de la portada de *2chan*



La gran ventaja de utilizar un *API* en el *backend* es que, de esta forma

se abre la posibilidad a la aparición de clientes programados por terceros que interactúen con los datos a través de un interfaz común para todos ellos. Haciendo muy portable el proyecto hacia nuevos entornos, como por ejemplo, en forma de app móvil, aplicación de escritorio u otras interfaces web. Asimismo, el hecho de independizar por completo el cliente del servidor permite que futuras versiones puedan prescindir de alguna de las partes con mucha facilidad y re-implementarla al gusto, facilitando al máximo la re-utilización del código.

2. Ámbito de implantación

Δ chan será administrado por una organización sin ánimo de lucro (*The Δ chan Foundation*), creada específicamente con la finalidad de asegurar la libertad, independencia y neutralidad del proyecto, así como mantenerlo ajeno a todo interés económico, político y personal que pueda derivar de una administración centrada en una única o un reducido número de personas. Imitando así modelos como los implementados en *Wikimedia* [5] o *The Internet Archive* [6]. Esta será la encargada tanto de la parte financiera, administrando las donaciones que se puedan recibir por parte de los usuarios, como la parte tecnológica, ofreciendo al proyecto de la infraestructura física y el mantenimiento necesario para asegurar su correcto funcionamiento.

Figura 4: Vista de lectura de un *thread* en *4chan*



El objetivo final pasa por establecer un lugar de debate e intercambio de opiniones abierto, libre y neutral; donde cada usuario pueda encontrar su rincón y hablar de lo que le apetezca con personas que comparten sus mismos gustos y aficiones. Y al mismo tiempo ofrecer espacios donde posiciones opuestas se enfrenten para que eventualmente se alcance un consenso común y beneficioso para ambas partes.

Este proyecto va dirigido a usuarios que les importan más las ideas – el debate en sí– y no las personas que sostienen esas ideas, gente de todas las edades que quiera compartir sus opiniones y experiencias con una gran comunidad de distribuida por todo el mundo. El anonimato hace que la gente se pueda expresar sin tapujos y sin esperar consecuencias en el ámbito personal, instigadas por lo que uno piensa.

3. Recursos de *hardware* y *software*

Puesto que Δ chan es un *aplicación web* existen tres escenarios a tratar con respecto a los requisitos de *hardware* y *software*: **desarrollo, despliegue e instalación por parte del usuario**. Inicialmente se pretende que en cualquier de ellos, estos recursos, sean lo más limitados y gratuitos posibles haciendo que el proyecto pueda ser accesible por cualquiera sin importar la calidad del *hardware* disponible.

3.1. Requisitos de desarrollo

Los requisitos para el desarrollo de este proyecto no son para nada exigentes, todas las partes se pueden montar e interconectar en un mismo equipo con, prácticamente, cualquier especificación de *hardware*. Unos requisitos mínimos podrían ser, un CPU de al menos cuatro núcleos y 2.0GHz de frecuencia, con al menos 4 GB de memoria *RAM*, 10 GB de espacio libre en el disco duro y, como requisito extra, es indispensable una conexión estable a internet en el puesto de trabajo, a ser posible de 50Mbps o superior. Alternativamente se recomienda lo siguiente: CPU con más de cuatro núcleos y frecuencia mínima cercana a 2.0GHz, 16Gb de *RAM* y 50Gb de espacio libre en *SSD*.

Además del *hardware*, los siguientes programas son necesarios para poder montar un entorno de desarrollo adecuado y funcional (todos ellos en sus versiones más recientes): un servidor de base de datos *MariaDB* o *MySQL*, para poder ejecutar consultas *SQL* contra él en el entorno local, que gracias a *Docker* [7] se podrá emular de la forma más similar como se realizarán una vez se haya desplegado toda la infraestructura; un servidor *HTTP*, también

“*dockerizado*”, como puede ser *NGINX* o *Apache2*; y, por último será necesario un gestor de versiones como *Git* encargado de gestionar las versiones y coordinar a las diferentes personas encargadas del desarrollo. Por otra parte, se recomiendan algunos programas opcionales que quedan a elección del desarrollador, como puede ser, un editor de código adaptado para las diferentes tecnologías y lenguajes utilizados en el proyecto (*Visual Studio Code OSS*, *Atom*, *Geany*...); un cliente de bases de datos como *DBeaver* o *MySQL Workbench*, para administrar de manera más sencilla la base de datos en las fases más tempranas del desarrollo. Y como apunte final sería aconsejable realizar todo el proceso de desarrollo sobre una distribución de *GUN/Linux* para simplificar al máximo la portabilidad al entorno del servidor cuando el proyecto se pase a fase de *producción*.

3.2. Requisitos de despliegue

El despliegue de la aplicación en el ámbito del servidor está pensado para realizarse en contenedores de *Docker* sobre un SO *GNU/Linux*, haciendo el entorno lo más ligero y portable posible, permitiendo una gran escalabilidad de cara al futuro. Al principio, el proyecto se podrá montar sobre una única máquina, y gracias a estar pensado para ser muy ligero, esta máquina no requerirá de unas características desmesuradas. Se podrá desplegar el proyecto en cualquier máquina virtual en la nube, por ejemplo en una instancia de *Amazon Web Services (AWS)*, *Google Cloud*, *Digital Ocean* o *Linode*; pero también en cualquier máquina virtual en un servidor propio. Además, si se prefiere, se puede instalar el servicio en una máquina física o distribuirlo por un entorno de varios servidores, en caso de prever grandes picos de tráfico. En cuanto al *hardware* (virtualizado o físico) mínimo se necesitan al menos las siguientes características: un CPU de al menos cuatro núcleos y 2.0GHz de frecuencia, memoria *RAM* de 8Gb o superior, 100Gb de espacio libre en disco duro y, sobre todo 100Mbps de conexión estable a internet. Pero, sin embargo, se recomienda: CPU con más de cuatro núcleos y frecuencia mínima cercana a 2.0GHz, 32Gb de *RAM*, 500Gb de espacio libre en *SSD* y conexión redundante y estable de al menos 600Mbps.

Tanto el diseño como el desarrollo de la base de datos se realizarán pensando en un servidor *MariaDB*, dejando así la puerta abierta a una mayor compatibilidad con *MySQL* y otras tecnologías similares. Estará pensada para ser lo más ligera posible, permitiendo que pueda ser alojada en el mismo equipo que el servidor *HTTP*. este se espera que sea un *NGINX* preferiblemente, o un *Apache2* como segunda opción, los dos excelentes servidores libres, utilizados tanto a nivel personal como profesional. Es importante hacer notar que las versiones de estos programas en el ámbito del servidor se

deben tomar con mucha cautela, y se suelen escoger las marcadas con el *tag LTS (Long Term Support)* para no sufrir cambios derivados de alguna futura actualización que puedan afectar al correcto funcionamiento de la aplicación.

3.3. Requisitos de instalación por parte del usuario

Al ser una aplicación web no requiere de una instalación, como tal, en el equipo del usuario final. Pero si que necesita de unos requisitos mínimos que vienen definidos por las tecnologías utilizadas en la parte del *frontend*. Por ejemplo, el usuario necesita tener instalado en su equipo por lo menos un navegador web que sea capaz de soportar código *ECMAScript 6* para que el *JavaScript* pueda funcionar sin ningún impedimento. Los navegadores más modernos, en sus versiones más actualizadas, con motores como *Gecko (Mozilla Firefox)* o *Webkit (Google Chrome, Opera, Brave...)* son perfectamente compatibles con esta reciente tecnología.

Con respecto a los requisitos *hardware*, no se requiere nada fuera de lo común, se considera que hoy en día todo dispositivo (*PC* o *smartphone*) posee las capacidades básicas para abrir un navegador, acceder a una página web y leer texto, visualizar imágenes o pequeños vídeos.

4. Temporalización del desarrollo

Lista de tareas a realizar y cuanto tiempo conlleva cada una¹.

- A. (2d²) *Setup* del entorno de desarrollo (*hardware* y *software*).
- B. (5d) Diseño de *mockups* para las pantallas de la web.
- C. (4d) Desarrollo y testeo de la base de datos.
- D. (7d) Desarrollo y testeo de los *endpoints* del *API*.
- E. (3d) Desarrollo y testeo de la conexión del *API* con la base de datos.
- F. (3d) Desarrollo y testeo de la estructura y funciones básicas de la web.
- G. (4d) Desarrollo y testeo de la conexión de la web con el *API*.
- H. (3d) Implementación de los estilos en la web.

¹Nótese que las tareas de diseño de la base de datos y definición de los *endpoints* de el *API* no se tienen en cuenta en la planificación temporal, porque ya están realizadas en este documento.

²Unidades de tiempo en días.

- I. (2d) Internacionalización de los textos de la web.
- J. (5d) Setup del entorno del servidor (*hardware* y *software*).
- K. (5d) Despliegue y publicación de la aplicación en el entorno del servidor.
- L. (7d) Redacción de la documentación del proyecto.

Figura 5: Diagrama PERT

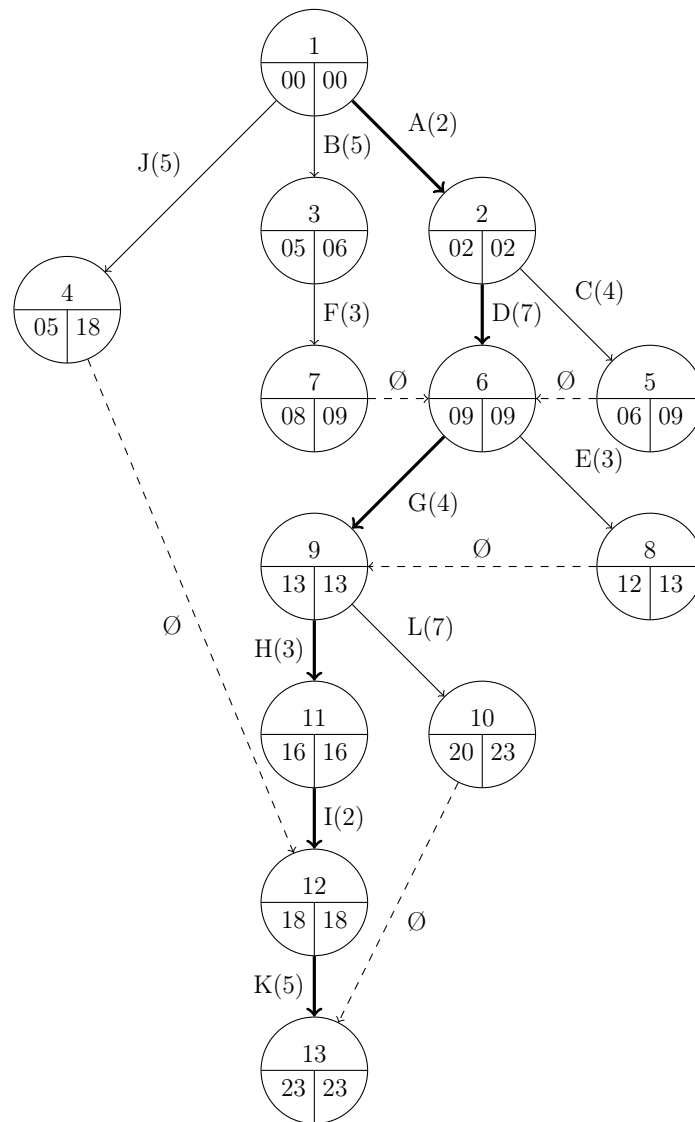
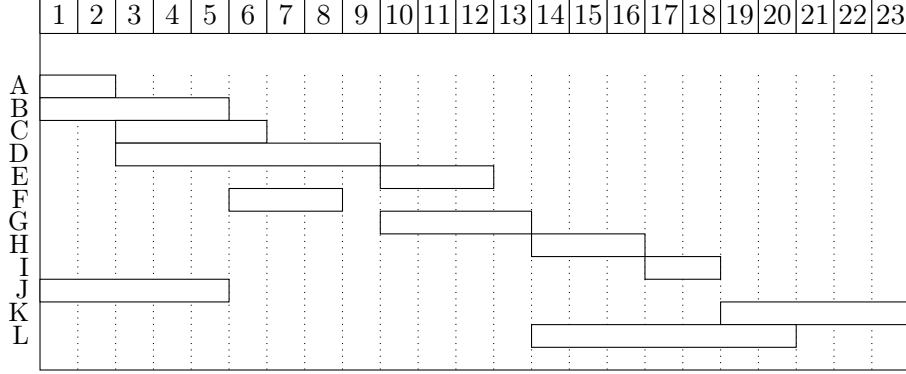


Figura 6: Diagrama de Gantt



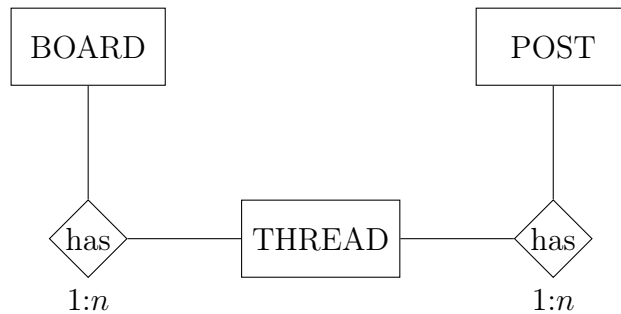
5. Descripción de los datos base y resultados

Las comunicaciones entre las diferentes partes del proyecto se realizarán “serializando” y “des-serializando” las clases de *Python* (en el *API*) y los “objetos” de *JavaScript* (en la web) a formato *JSON* para facilitar y estandarizar el transporte entre dichas partes. En cuanto a la base de datos la información se almacenará en tablas, siguiendo el modelo relacional de Codd [8].

5.1. Base de datos

El diseño de la base de datos, siempre enfocado hacia una mayor eficiencia y rapidez en la lectura y escritura de los datos, está dividida en tres tablas: *BOARD*, *THREAD* y *POST*. Este diseño intenta compartimentar al máximo las diferentes partes de la página con el mínimo acoplamiento entre las tablas para evitar hacer *joins* innecesarios, que supondrían una mayor carga para el servidor.

Figura 7: Diagrama Entidad-Relación de la base de datos de Δ chan



En el diagrama *ER* de la figura 7 no se han dibujado los atributos de las entidades con el fin de profundizar en ellos a continuación con el mayor detalle posible.

BOARD Cada uno de los tableros que dividen la página en diferentes temas.

- **slug** VARCHAR(4) PRIMARY KEY
- **name** VARCHAR(256) NOT NULL

THREAD Cada uno de los debates, o *hilos* de discusión, que se inician dentro de un tablón.

- **id** BIGINT PRIMARY KEY
- **subject** VARCHAR(256) DEFAULT '' NOT NULL
- **author** VARCHAR(50) DEFAULT 'Anonymous' NOT NULL
- **comment** VARCHAR(512) NOT NULL
- **fileurl** VARCHAR(512) DEFAULT NULL
- **published** DATETIME DEFAULT CURRENT_TIMESTAMP NOT NULL
- **sticky** BOOLEAN DEFAULT FALSE NOT NULL
- **closed** BOOLEAN DEFAULT FALSE NOT NULL
- **deleted** BOOLEAN DEFAULT FALSE NOT NULL
- **board** VARCHAR(4) FOREIGN KEY REF. BOARD(slug) NOT NULL

POST Cada comentario de un usuario dentro de un debate (*thread*), formado por un texto y una foto o pequeño vídeo opcional.

- **id** BIGINT PRIMARY KEY
- **author** VARCHAR(50) DEFAULT 'Anonymous' NOT NULL
- **comment** VARCHAR(512) DEFAULT 'Anonymous' NOT NULL

- **fileurl** VARCHAR(512) DEFAULT NULL
- **published** DATETIME DEFAULT CURRENT_TIMESTAMP NOT NULL
- **deleted** BOOLEAN DEFAULT FALSE NOT NULL
- **thread** BIGINT FOREIGN KEY REF. THREAD(id)

5.2. *API*

Definición de los *endpoints* del *API* basada en el estándar *Open API* versión 3.0.3 [9].

- (GET) /board – Devuelve la lista de todos los *boards*.
 - Respuestas
 - 200 – application/json – Un array con los *boards* y sus atributos.

Ejemplo 1: Respuesta 200 *OK* en *JSON* del endpoint (GET) /board

```

1  [
2    { "slug": "a", "name": "Anime & Manga" },
3    { "slug": "b", "name": "Random" },
4    { "slug": "g", "name": "Technology" },
5    { "slug": "news", "name": "Current News" },
6    { "slug": "vr", "name": "Retro Games" },
7    { "slug": "wg", "name": "Wallpapers/General" },
8    { "slug": "x", "name": "Paranormal" }
9  ]

```

- (GET) /{slug} – Devuelve la lista de todos los *threads* dentro de un *board*, ordenados desde el más reciente al más antiguo.
 - Parámetros
 - **division** – number, query – Número de *threads* que quieres que aparezcan por página.
 - **page** – number, query – Número de página del *board* que quieres ver, en base a la división anterior.
 - Respuestas

- 200 – `application/json` – Un array con los *threads* y su primer y últimos cinco *posts* (a modo de resumen).
 - 400 – `application/json` – Cuando los *query params* tienen un formato erróneo.
 - 404 – `application/json` – Cuando solicitas un *slug* o una página que no existe.
- (POST) `/slug` – Permite publicar un nuevo *thread* dentro de un *board* dado.
 - *Request body (required)* – `application/json`
 - **subject** – string – El asunto del *thread*.
 - **author** – string – El nombre del autor del *thread*, si se deja en blanco aparecerá *Anonymous*.
 - **comment** (*required*) – string – El texto del primer *post* del *thread*.
 - **fileurl** – string – La *URL* del archivo adjunto al *thread*.
 - **slug** (*required*) – string – El *slug* del *board* dónde se va a publicar el *thread*.
 - Respuestas
 - 201 – `application/json` – El *thread* se ha publicado correctamente.
 - 400 – `application/json` – Cuando el *request body* tiene un formato erróneo.
 - 404 – `application/json` – Cuando se envía un *slug* que no existe.

Ejemplo 2: *Request body* en *JSON* de llamada a (POST) `/x`

```

1 {
2   "subject": "Alien Disclosure",
3   "comment": "How do the vessels move without any
               visible propulsion system?",
4   "fileurl": "https://dcdn.org/x/1622583515994s.jpg",
5   "slug": "x"
6 }
```

- (GET) `/slug/thread/{id}` – Devuelve la lista de todos los *posts* de un *thread* dado.

- Respuestas
 - 200 – `application/json` – Un array con los *posts* ordenados de más antiguo a más reciente.
 - 404 – `application/json` – Cuando solicitas un *slug* o un *thread* que no existe.
- (POST) `/[{slug}]/thread/{id}` – Permite publicar un nuevo *post* dentro de un *thread* dado.
 - *Request body (required)* – `application/json`
 - **author** – string – El nombre del autor del *post*, si se deja en blanco aparecerá *Anonymous*.
 - **comment** (*required*) – string – El texto del *post*.
 - **fileurl** – string – La *URL* del archivo adjunto al *post*.
 - **thread** (*required*) – number – El número del *thread* dónde se va a publicar el *post*.
- Respuestas
 - 201 – `application/json` – El *post* se ha publicado correctamente.
 - 400 – `application/json` – Cuando el *request body* tiene un formato erróneo.
 - 404 – `application/json` – Cuando se envía un *slug* o un *thread* que no existe.

Además de los *endpoints*, en el *API*, los datos obtenidos desde la base de datos se traducen en objetos *DTO* (*Data Transfer Object* [10]) para facilitar el manejo y la serilaización de la información que se va a devolver.

Ejemplo 3: Implementación del *DTO ThreadDTO* en el *API*

```

1 class ThreadDTO:
2     def __init__(self, id, subject, author, comment,
3         fileurl, published, sticky, closed):
4         self.id = id;
5         self.subject = subject;
6         self.author = author;
7         self.comment = comment;
8         self.fileurl = fileurl;
9         self.published = published.strftime('%c');
10        self.sticky = sticky;
11        self.closed = closed;
```

```

11 def to_JSON(self):
12     return json.dumps(self, default=lambda o: o.__dict__)

```

6. Relación entre dispositivos y programa o rutinas

En este apartado se intenta detallar cómo funcionarán todas las partes en conjunto, explicando a bajo nivel algunas acciones que se pueden realizar por parte del usuario. Describiendo los componentes *software* implicados, que partes están conectadas y cómo se realizan las comunicaciones entre ellos.

Figura 8: Relaciones entre las partes del proyecto entre sí y con el usuario



Es importante resaltar que los ejemplos de código que se muestran en esta sección son ejemplos *mockup* que solo cubren las funcionalidades básicas de la aplicación, lo que en inglés se llama *happy path*. El propósito de estos es dar una idea de cómo se pretende estructurar el código para que puedan ser tomados como referencia en el futuro desarrollo del proyecto.

6.1. Visualización del contenido de un *board*

Una de las tareas mas simples, pero fundamentales, del proyecto, que ofrece al usuario una herramienta básica para navegar por la aplicación. En este ejemplo vamos a suponer que somos un usuario que se encuentra en la portada de la web (*home screen*), donde visualiza la lista de *boards* que están disponibles, como se muestra en la figura 9, y quiere acceder al *board Technology*, cuyo *slug* es */g/*.

Figura 9: Lista de *boards* en la portada de la web

Boards				filter ▼
Japanese Culture	Interests	Creative	Other	
Anime & Manga	Comics & Cartoons	Papercraft & Origami	Business & Finance	
Anime/Cute	Technology	Photography	Travel	
Anime/Wallpapers	Television & Film	Food & Cooking	Fitness	
Mecha	Weapons	Literature	Paranormal	
Cosplay & EGL	Auto	Music	Advice	
Cute/Male	Animals & Nature	Fashion	LGBT	
Transportation	Traditional Games	3DCG	Pony	
Otaku Culture	Sports	Graphic Design	Current News	
Virtual YouTubers	Extreme Sports	Do-It-Yourself	Worksafe Requests	
Video Games	Professional Wrestling	Worksafe GIF	Very Important Posts	
Video Games	Science & Math	Quests		
Video Game Generals	History & Humanities			
Video Games/Multiplayer	International			
Video Games/Mobile	Outdoors			
Pokemon	Toys			
Retro Games				
Video Games/RPG				
Video Games/Strategy				

Una vez el usuario haga clic en el enlace el proceso empieza con una llamada de tipo *GET* al *API* desde el método `getThreads()` en cliente web.

Ejemplo 4: Implementación del método `getThreads()` en el cliente web

```
1 const getThreads = async (slug, callback) => {
2   const response = await fetch('https://dchan.org/${slug}');
3   const threadsJSON = await response.json();
4   callback(threadsJSON);
5 }
```

Esta llamada es interceptada por el *API* y la redirige al siguiente *endpoint*.

Ejemplo 5: Implementación del *endpoint* (GET) `/g` en el *API*

```
1 @app.route('/<slug>', methods=['GET'])
2 def get_threads(slug):
3     thread_list = []
4     mydb = get_mydb()
5     mycursor = mydb.cursor()
6     mycursor.execute(
7         "SELECT name FROM BOARD WHERE slug = '{}'.format(
8             slug)
9     )
10    board_name = mycursor.fetchone()[0]
11    mycursor.execute(
12        "SELECT * FROM THREAD WHERE board = '{}' ORDER BY
13            published DESC".format(slug)
14    )
15    myresult = mycursor.fetchall()
16    for col in myresult:
17        thread_list.append(Thread(*col[:-2]))
18    board = Board(slug, board_name, thread_list)
19    encoded_JSON = json.dumps(board.to_JSON())
20    decoded_JSON = json.loads(encoded_JSON)
21    mycursor.close()
22    mydb.close()
23    return app.response_class(
24        response = decoded_JSON,
25        status = 200,
26        mimetype = 'application/json'
```

El método `get_threads()` se activa en el código del *API* con una llamada como (GET) `/g`. Este realiza la consulta a la base de datos para obtener

la lista de *threads* que hay dentro del *board /g/* y, una vez procesado y serializado el resultado, retorna la siguiente respuesta.

Ejemplo 6: Respuesta en *JSON* de llamar a (GET) */g*

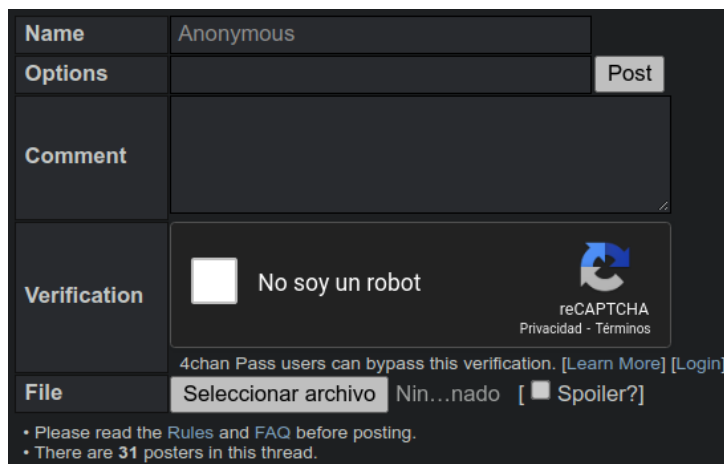
```
1  {
2    "slug": "g",
3    "name": "Technology",
4    "thread_list": [
5      {
6        "id": 81930017,
7        "subject": "/pcbg/ - PC Building General",
8        "author": "Anonymous",
9        "comment": ">UPGRADE & BUILD ADVICE\nPost build \n"
          "list\" or current specs including MONITOR\
          nConvient lister: https://pcpartpicker.com/\
          nProvide specific use cases (e.g. gaming,
10       "fileurl": "https://dcdn.org/g/1622853294379s.jpg",
11       "published": "Sat Jun  5 16:55:27 2021",
12       "sticky": 0,
13       "closed": 0
14     },
15     {
16       "id": 81931054,
17       "subject": "What is the most efficient way to heat
          water?",
18       "author": "Anonymous",
19       "comment": "Microfusion reactors?",
20       "fileurl": "https://dcdn.org/g/1622859370330.jpg",
21       "published": "Sat Jun  5 17:32:54 2021",
22       "sticky": 0,
23       "closed": 0
24     }
25   ]
26 }
```

Ahora que ya tenemos la respuesta en el cliente web, se ejecutará la función de *callback* en el método `getThreads()` del ejemplo 4, haciendo que se visualice en el cliente web la información solicitada, de forma similar a cómo se muestra en la figura 1.

6.2. Creación de un nuevo *post* en un *thread* (*reply*)

Supongamos que un usuario está leyendo su *thread* favorito y quiere aportar un lúcido comentario a la discusión. Lo único que tiene que hacer es pulsar sobre el botón que dice [Post a Reply] y se desplegará un formulario como el de la figura 10.

Figura 10: Ejemplo de formulario para agregar un nuevo *post* a un *thread*



The image shows a web form for posting a reply on 4chan. The form is titled 'Name' with the value 'Anonymous'. Below it is an 'Options' field. The main part of the form is a large text area labeled 'Comment'. Below the comment area is a 'Verification' section containing a checkbox labeled 'No soy un robot' and a reCAPTCHA logo. Below the verification section is a line of text: '4chan Pass users can bypass this verification. [Learn More] [Login]'. At the bottom of the form is a 'File' section with a button labeled 'Seleccionar archivo' and a checkbox labeled 'Nin...nado' with a 'Spoiler?' label. Below the file section is a small text area with two lines: '• Please read the Rules and FAQ before posting.' and '• There are 31 posters in this thread.'

Una vez se hayan rellenado todos los campos obligatorios del formulario, al pulsar sobre el botón de Post se ejecutará su método asociado para realizar la llamada al *API*.

Ejemplo 7: Implementación del método `postReply()` en el cliente web

```
1 const postReply = async (dataToSend, callback) => {
2   const { slug, threadId } = dataToSend;
3   const response = await fetch(`https://dchan.org/${slug}
4     }/thread/${threadId}`, {
5     method: 'POST',
6     body: JSON.stringify(dataToSend)
7   });
8   const responseJSON = await response.json();
9   callback(responseJSON);
10 }
```

Ejemplo 8: *Request body* que se manda a (POST) `/slug/thread/{id}`

```
1 {
2   "author": "Anonymous",
3   "comment": "me gustan los parterres",
4 }
```

```

4  "imageurl": "https://dcdn.org/g/1622865285989.png"
5  }

```

Una vez recibida la llamada en la *API* se ejecutará el método asociado, para procesar la información recibida y almacenarla correctamente en la base de datos.

Ejemplo 9: Implementación del *endpoint* (POST) `/slug/thread/{id}`

```

1  @app.route('/<slug>/thread/<thread_id>', methods=['POST',
2  ])
3  def post_reply(slug, thread_id):
4      formData = request.get_json()
5      mydb = get_mydb()
6      mycursor = mydb.cursor()
7      args = (
8          formData['author'],
9          formData['comment'],
10         formData['imageurl'],
11         thread_id
12     )
13     mycursor.callproc('insert_post', args)
14     mycursor.close()
15     mydb.close()
16     return app.response_class(
17         response = '{"info": "posted"}',
18         status = 201,
19         mimetype = 'application/json'
20     )

```

El método del *API* `post_reply()` hace uso del procedimiento almacenado `insert_post`, el cual facilita las cosas a la hora de insertar nuevos valores en la base de datos, pues ofrece automatismos como el incremento del `id`.

Ejemplo 10: Procedimiento almacenado para la inserción de un nuevo *post*

```

1  DELIMITER $$
2  DROP PROCEDURE IF EXISTS insert_post;$$
3  CREATE PROCEDURE insert_post(
4      IN new_author VARCHAR(50),
5      IN new_comment VARCHAR(512),
6      IN new_fileurl VARCHAR(512),
7      IN new_thread BIGINT
8  ) BEGIN
9      SET @last_id = 0;
10     SELECT id INTO @last_id FROM POST

```

```

11         ORDER BY id DESC LIMIT 1;
12     IF (@last_id IS NULL) THEN
13         SET @last_id = 0;
14     END IF;
15     INSERT
16         INTO POST (id, author, comment, fileurl, thread)
17         VALUES (@last_id + 1, new_author, new_comment,
18                 new_fileurl, new_thread);
19 END;$$
DELIMITER ;

```

Una vez el nuevo *post* queda registrado en la base de datos y asociado a un *thread*, el usuario será informado de que la publicación se ha realizado satisfactoriamente con un mensaje de felicitación. La próxima vez que se visite el *thread* (ejemplo en la figura 4) o se actualize la página, el usuario podrá ver reflejada su aportación a la conversación.

Ejemplos de código

1.	Respuesta 200 <i>OK</i> en <i>JSON</i> del endpoint (GET) <code>/board</code> . . .	12
2.	<i>Request body</i> en <i>JSON</i> de llamada a (POST) <code>/x</code>	13
3.	Implementación del <i>DTO ThreadDTO</i> en el <i>API</i>	14
4.	Implementación del método <code>getThreads()</code> en el cliente web .	16
5.	Implementación del <i>endpoint</i> (GET) <code>/g</code> en el <i>API</i>	16
6.	Respuesta en <i>JSON</i> de llamar a (GET) <code>/g</code>	17
7.	Implementación del método <code>postReply()</code> en el cliente web .	18
8.	<i>Request body</i> que se manda a (POST) <code>/slug/thread/{id}</code> .	18
9.	Implementación del <i>endpoint</i> (POST) <code>/slug/thread/{id}</code> .	19
10.	Procedimiento almacenado para la inserción de un nuevo <i>post</i>	19

Esta sección es una compilación todos los bloques de código que se han descrito a lo largo del documento, y **equivale al apartado 7** de la guía del proyecto.

Referencias

- [1] Wikipedia, *Imageboard*, n.d. dirección: <https://en.wikipedia.org/wiki/Imageboard> (visitado 21-05-2021).
- [2] F. S. F. Inc., *GNU General Public License version 3*, 2007. dirección: <https://www.gnu.org/licenses/gpl-3.0.html> (visitado 23-05-2021).
- [3] R. M. Stallman, «3. La definición del software libre,» en *Software libre para una sociedad libre*, Traficantes De Sueños, 2004. dirección: https://www.gnu.org/philosophy/fsfs/free_software2.es.pdf (visitado 23-05-2021).
- [4] Wikipedia, *Fork (software)*, n.d. dirección: [https://en.wikipedia.org/wiki/Fork_\(software_development\)](https://en.wikipedia.org/wiki/Fork_(software_development)) (visitado 21-05-2021).
- [5] T. W. Foundation, *About the Wikimedia Foundation*, n.d. dirección: <https://wikimediafoundation.org/about/> (visitado 29-05-2021).
- [6] T. I. Archive, *About the Internet Archive*, n.d. dirección: <https://archive.org/about/> (visitado 29-05-2021).
- [7] D. Inc., *Docker overview*, n.d. dirección: <https://docs.docker.com/get-started/overview/> (visitado 03-06-2021).
- [8] E. F. Codd, «A Relational Model of Data for Large Shared Data Banks,» *Communications of the ACM*, vol. 13, n.º 6, págs. 377-387, 1970. DOI: <https://dl.acm.org/doi/10.1145/362384.362685>.
- [9] S. Software, *OpenAPI Specification*, 20 de feb. de 2020. dirección: <https://swagger.io/specification/> (visitado 30-05-2021).
- [10] Wikipedia, *Data transfer object*, n.d. dirección: https://en.wikipedia.org/wiki/Data_transfer_object (visitado 05-06-2021).