
Máster en Cloud Apps
Desarrollo y despliegue de aplicaciones en la nube

Curso académico 2020/2021

Trabajo de Fin de Máster

# Twitter Scheduler

Autor: David Rojo Antona
Tutor: Francisco Gortázar Bellas

# CONTENIDO

# FIGURAS

# 1. RESUMEN

El presente Trabajo Fin de Máster consiste en la implementación de una aplicación usando el modelo de desarrollo de Trunk-based development con una serie de nuevas funcionalidades aplicando las técnicas de diseño de Feature toggles y Branch by abstraction.

El código del proyecto está alojado en un repositorio en GitHub, en el que después de cada commit, a través de un entorno de integración y despliegue continuo, se actualiza la versión en producción de la aplicación alojada en la plataforma Heroku[1], por tanto, se ha seguido fielmente el modelo de Trunk-based development.

La aplicación está implementada en Java 11, usando Spring Boot[2] y siguiendo una arquitectura hexagonal aplicando Domain Driven Design.

[1] https://www.heroku.com/
[2] https://spring.io/projects/spring-boot

# 2. OBJETIVOS

- Desarrollar una aplicación de principio a fin utilizando la técnica de Trunk-based development, apoyándome en los contenidos vistos en las diferentes asignaturas del Master CloudApps.
- Utilización de la técnica de Feature toogles para la implementación de nuevas funcionalidades.
- Proporcionar un entorno de integración y despliegue continuo para permitir automatizar el testeo, la construcción y el despliegue de la aplicación después de cada commit que se realice.
- Mantener la aplicación en producción funcionando en Heroku en todo momento.

# 3. INTRODUCCIÓN

Se ha tomado como modelo de desarrollo **Trunk-based development (TBD)**, dicho modelo aplica las siguientes bases:

- Todo el desarrollo se hace en *Trunk* [3].
- Trunk siempre está listo para versionar y desplegar en producción.
- Se pueden tener ramas, pero son de muy corta duración
- El código que no está listo todavía para ser usado se oculta mediante Feature toogles[4].
- Soluciona problemas de refactoring con la técnica de Branch by abstraction.
- El commit es revisado por el equipo, aunque este paso se puede saltar si se ha desarrollado aplicando pair programming[5], la manera habitual es a través de una pull request contra `main`.

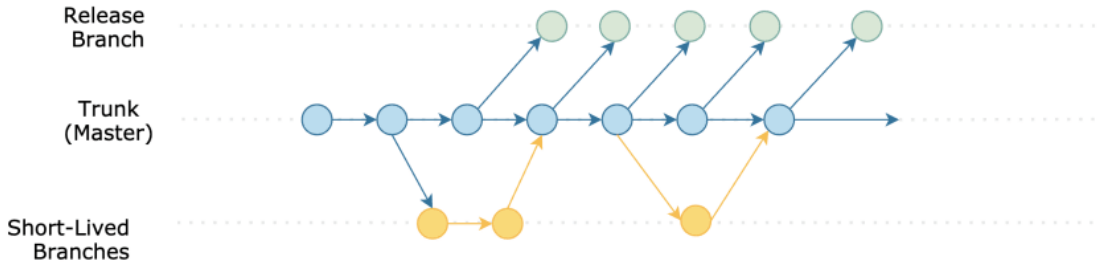


*Figura 1 - Flujo de trabajo de Trunk-based development*

Las **Feature toggles** son una técnica que permite activar o desactivar una funcionalidad de la aplicación, lo que permite ocultar una nueva funcionalidad hasta completarla, sin tener que esperar a que esté finalizada para poder incluir en `main` los cambios que se van haciendo. Esto permite:

- Los cambios son integrados rápidamente, evitando integraciones más costosas y complicadas ya que si es una tarea muy grande, llevará bastante tiempo implementarla y el código en `main` puede haber cambiado mucho.
- Los cambios son examinados por el resto del equipo al validar cada commit, poco a poco, sin tener que esperar a que toda la funcionalidad está terminada, lo que implicaría un mayor esfuerzo en su análisis y verificación.

El uso de Feature toggles puede ser aplicado en los siguientes casos de uso:

- Releases con funcionalidades incompletas: se puede hacer en cualquier momento, aunque haya funcionalidades incompletas.
- Canary releases: desplegar funcionalidades activadas para un subconjunto de usuarios, examinar su nivel de satisfacción y evaluar si se activan o no para el resto.
- Lean experiments: desplegar funcionalidades experimentales para usuarios concretos.
- Dark launches: desplegar una funcionalidad oculta para los usuarios y probar el correcto funcionamiento a nivel interno antes de hacerla visible.
- Incremental rollouts: ir poco a poco abriendo el número de usuarios que ven una funcionalidad a medida que se comprueba que funciona correctamente.
- Early access: acceso a una funcionalidad a un grupo de usuarios que han solicitado probarla.

[3] `main` en git
[4] También llamadas feature flags
[5] Programación por parejas

La técnica de **Branch by abstraction** permite realizar grandes modificaciones (o modificaciones que tardan bastante tiempo en realizarse) sin impactar en la capacidad de seguir introduciendo nuevas funcionalidades en la rama `main` y permitiendo la generación de nuevas releases. Se divide el cambio en las siguientes fases:

- Abstracción del código que se desea modificar.
- Inclusión de la nueva implementación.
- Activación del código nuevo.
- Eliminación de la abstracción.

Para poner en uso todo lo descrito anteriormente se ha elegido una aplicación Maven implementada en Java con SpringBoot siguiendo una arquitectura hexagonal aplicando Domain Driven Design. La aplicación permite la planificación de publicaciones en la red social Twitter y se han implementado dos nuevas funcionalidades aplicando Feature toggles y Branch by abstraction.

# 4. DESARROLLO DEL TRABAJO FIN DE MÁSTER

Para la realización de este proyecto, después de analizar y pensar la mejor manera de hacerlo, se ha decidido dividir el mismo en tres fases separadas y que serán explicadas en detalle en las siguientes secciones.

Cada fase depende de la fase anterior, por lo que se han llevado a cabo de manera secuencial y hasta que no se ha completado la primera, no se ha pasado a la segunda y así sucesivamente:

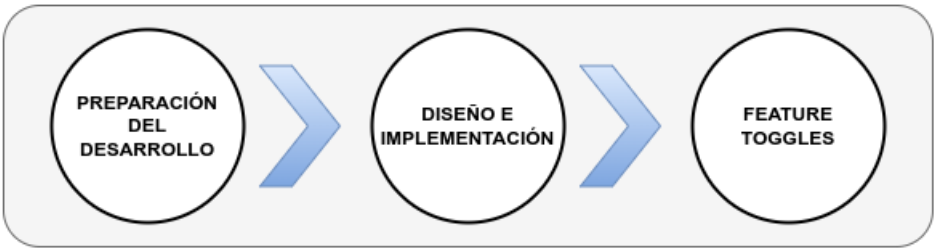


*Figura 2 - Fases de desarrollo del Trabajo Fin de Master*

## 4.1. PREPARACIÓN DEL DESARROLLO

El primer paso antes de comenzar con la implementación de la aplicación, era configurar adecuadamente y tener disponible un entorno de integración y despliegue continuo.

Se ha elegido GitHub como plataforma de alojamiento del repositorio de código del proyecto[6] ya que ofrece la herramienta integrada de GitHub actions que permite automatizar flujos de integración continua y despliegue continuo (CI/CD)[7] a través de workflows en formato YAML.

Para poder realizar las pruebas y comprobar que la integración y despliegue continuo funcionan correctamente, era necesario disponer de una aplicación con una mínima funcionalidad. Para ello se ha usado la utilidad Spring Initializr[8], para generar el esqueleto del proyecto Spring Boot y que posteriormente será usado para implementar la aplicación. Una vez generado, se ha incluido el módulo de Spring Actuator[9], el cual permite exponer una serie de endpoints de manera automática, entre ellos `health`, el cual informa del estado de la aplicación:

```
{"status":"UP"}
```

A nivel de **seguridad**, para que no sea pública y evitar el acceso indeseado a la aplicación se ha implementado a través de Spring Security[10] el acceso a los endpoints expuestos a través de basic authentication, usando un usuario y password concreto.

En el repositorio se dispone de una sola rama, `main`, en la que cada vez que se realiza un commit, se detecta y es ejecutado un workflow[11] por GitHub actions. En caso de que el worfklow termine satisfactoriamente, la release generada se despliega en la plataforma Heroku.

[6] https://github.com/MasterCloudApps-Projects/TwitterScheduler
[7] Acrónimos de los términos en inglés: Continuous integration y Continuous Deployment
[8] https://start.spring.io/
[9] https://docs.spring.io/spring-boot/docs/current/reference/html/actuator.html
[10] https://spring.io/projects/spring-security
[11] https://github.com/MasterCloudApps-Projects/TwitterScheduler/blob/main/.github/workflows/push-main.yml

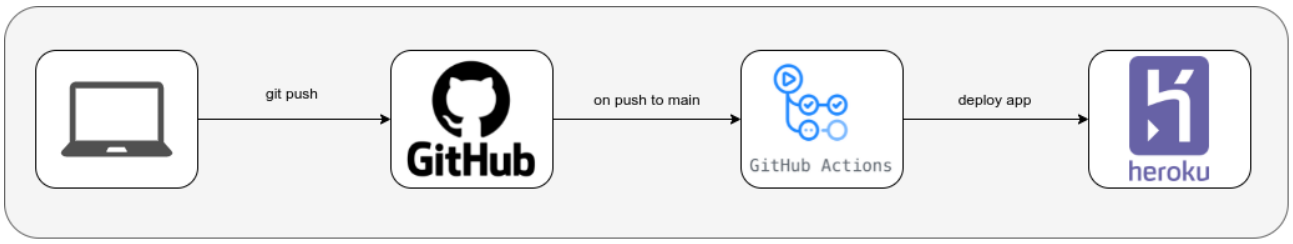


*Figura 3 - Integración y despliegue continuo en cada commit a `main`*

Para poder desplegar en Heroku, es necesario el registro en la plataforma, crear una aplicación (asignando un nombre único) y configurar las config vars necesarias como, por ejemplo, el usuario y password de la API.

Aunque Heroku ofrece diferentes opciones de pago por uso, todo el trabajo realizado en el presente proyecto, se ha hecho dentro de la **capa gratuita** de Heroku a la que se puede acceder a través de tener un usuario en dicha plataforma.

A continuación, se pueden ver las diferentes fases de las que se compone el workflow y que son ejecutadas después de cada commit hecho en `main`:

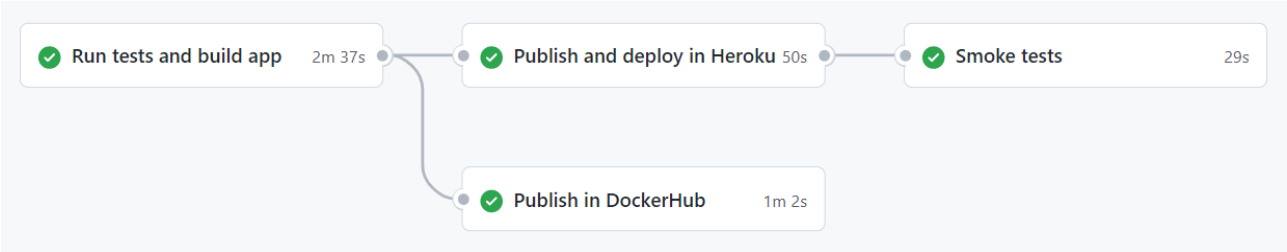


*Figura 4 - Fases del workflow de GitHub Actions*

1. **Run tests and build app:** son ejecutados los tests de la aplicación, tanto los unitarios como los de integración, en caso de ser ejecutados satisfactoriamente, se procede a construir la aplicación y se sube el archivo .jar generado, para que esté accesible para la siguiente fase.
2. **Publish in DockerHub:** se descarga el archivo .jar generado en la fase anterior, se construye la imagen Docker y se publica en el repositorio[12] de la aplicación creado en el servicio de registro de imágenes Docker Hub, con dos tags: la versión de la aplicación en el archivo pom.xml[13] y *latest*.
3. **Publish and deploy in Heroku:** se descarga el archivo .jar construido en la primera fase, se construye la imagen de Heroku y se procede al despliegue en Heroku del contenedor con la nueva release de la aplicación.
4. **Smoke tests:** una vez publicada la nueva versión en Heroku, se procede a comprobar que la aplicación funciona correctamente, para ello se realizan dos pruebas realizando consultas GET usando el comando curl al endpoint REST que determina si la aplicación funciona correctamente o no y analizando su respuesta, si la primera no es superada, la segunda consulta no se ejecuta:
    - Comprueba que el código http de la respuesta es 200 (OK).
    - Comprueba que la respuesta obtenida, tiene “UP” como valor para el atributo “status”.

Para poder ejecutar las fases indicadas en el workflow, es necesario almacenar credenciales de diferentes servicios, las cuales han sido guardadas como *Secrets* en el repositorio GitHub para que así estén protegidas.

[12] https://hub.docker.com/repository/docker/drojo/twitter-scheduler-tfm
[13] https://github.com/MasterCloudApps-Projects/TwitterScheduler/blob/main/pom.xml

## 4.2. DISEÑO E IMPLEMENTACIÓN INICIAL

Una vez listo el entorno de integración y despliegue continuo, el siguiente paso ha sido centrarse en el diseño e implementación de la aplicación que permite la planificación de publicación de tweets en la red social Twitter. Permite crear tweets pendientes con una fecha de publicación y dispone de una tarea programada, ejecutada con una frecuencia de tres minutos, que determina si hay tweets pendientes con fecha de publicación anterior a esa hora actual + 3 minutos (para abarcar hasta la siguiente ejecución de la tarea programada). En caso de que haya tweets pendientes, son publicados.

Se ha elegido que la aplicación tenga una **arquitectura hexagonal**[14] en la que hay tres conceptos principales:

- **Dominio** (o modelo): contiene la lógica del negocio y está situado en el centro, totalmente agnóstico de la tecnología que se use. Todas las dependencias son hacia el dominio y el dominio no tiene dependencias. Se mantiene en una jerarquía de paquetes separada.
- **Aplicación** (o puertos): encapsulan el dominio, son los casos de uso que tiene la aplicación. Por ejemplo: crear tweet pendiente.
- **Infraestructura** (o adaptadores): son los medios por los que el usuario puede realizar los casos de uso. Por ejemplo:
    - API REST para obtener lista de tweets publicados.
    - Base de datos PostgreSQL donde se almacenan los tweets publicados.

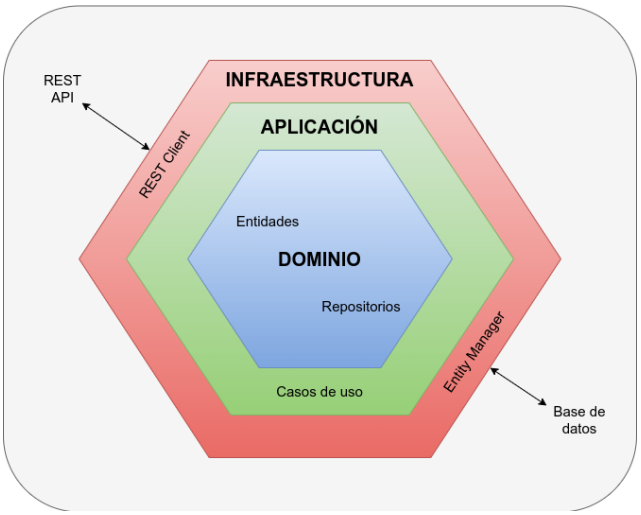


*Figura 5 - Arquitectura Hexagonal*

Se aplica el principio de inversión de dependencias lo que ayuda a mantener el dominio independiente del resto. Se utilizan interfaces para los puertos y el dominio depende de interfaces, no de implementaciones concretas, lo que facilita que se puedan realizar cambios en los adaptadores para cambiar la tecnología con la que se implementan los puertos.

Los casos de uso se definen mediante interfaces que son usadas por los controladores y servicios Spring.

Las dependencias del dominio hacia el exterior se modelan mediante puertos, en los que se define la interfaz de lo que espera el dominio (usando sólo dependencias del dominio o DTOs[15]) y se crean adapters (patrón adapter) que implementan esas interfaces a los objetos del modelo o DTOs agnósticas de ambos.

[14] También denominada Puertos y Adaptadores
[15] Data Transfer Object

Siguiendo un diseño **Domain Driven Design**, después de analizar el escenario, se ha determinado la existencia de dos Aggregate Root en el dominio de la aplicación:


*Figura 6 - Aggregate roots de la aplicación*

- **PendingTweet:** un tweet que aún no ha sido publicado, identificado por su clave autogenerada en la aplicación, de tipo numérico `pendingTweetId` y compuesto por los value objects:
  - `message:` cuerpo del tweet que será publicado. Si supera los 280 caracteres lanzará una excepción, indicando que supera el máximo permitido por Twitter.
  - `publicationDate:` fecha de publicación en la que será publicado el tweet.
  - `createdAt:` campo automático, indica en formato UTC cuando ha sido creado el PendingTweet
- **Tweet:** un tweet ya publicado en Twitter, identificado por su clave númerica tweetId, el cual es el identificador numérico generado por la red social y es usado como clave en la aplicación. Además, dispone de los siguientes value objects:
  - `message`: cuerpo del tweet publicado
  - `url`: url del tweet
  - `requestedPublicationTime`: fecha en formato UTC que indica cual fue el instante solicitado para que el tweet fuera publicado.
  - `publishedAt`: fecha en formato UTC que indica cual fue el instante en el que el tweet fue publicado.
  - `createdAt`: campo automático, indica en formato UTC cuando ha sido creado el Tweet.

El diseño de los aggregate root se ha simplificado a la hora de explicarlos en el presente documento, ya que he hecho una serie de clases base utilizadas[16] en la implementación de los aggregate root.

Para gestionar la aplicación se han creado dos perfiles[17] de Spring:

- standalone: usado para pruebas en local y ejecución de los tests de integración.
- pro: el perfil por defecto y el que se usa en producción.

Para implementar la **persistencia**, he elegido bases de datos relacionales dependiendo del perfil seleccionado:

- standalone: base de datos en memoria H2
- pro: base de datos Heroku Postgres[18], implementación de PostgreSQL ofrecida por Heroku. Es una base de datos en la nube

Se ha decidido incluir el uso de Flyway en base de datos para así garantizar una correcta migración de los datos, a medida que en la base de datos se van incluyendo nuevos cambios en la estructura de la misma.

[16] https://github.com/MasterCloudApps-Projects/TwitterScheduler/tree/main/src/main/java/com/mastercloudapps/twitterscheduler/domain/shared
[17] https://docs.spring.io/spring-boot/docs/current/reference/html/features.html#features.profiles
[18] https://www.heroku.com/postgres

Una vez que los pendingTweet son publicados en Twitter, se eliminan de la tabla de pendingTweets y son creados en la tabla de Tweets con la información necesaria.

Para realizar la publicación de los tweets en la red social Twitter se usa la librería twitter4j[19] y ha sido necesaria la creación de una aplicación en el Developer Portal[20] de Twitter y generar las keys y tokens necesarios (consumer key, consumer secret, access token y access token secret), los cuales son usados en la aplicación.

Se han incluido **tests** con Junit5 tanto de integración, en los que se utiliza la base de datos H2 en memoria, como unitarios, especialmente en el dominio, disponiendo de una suite compuesta de más de 130 tests.

Para operar con la aplicación, se ha definido un API REST con diferentes endpoints y se ha incluido en la aplicación la implementación de la especificación **OpenAPI**, lo que permite una documentación interactiva de la misma y acceder a los servicios de la API con un gasto mínimo de puesta en marcha.

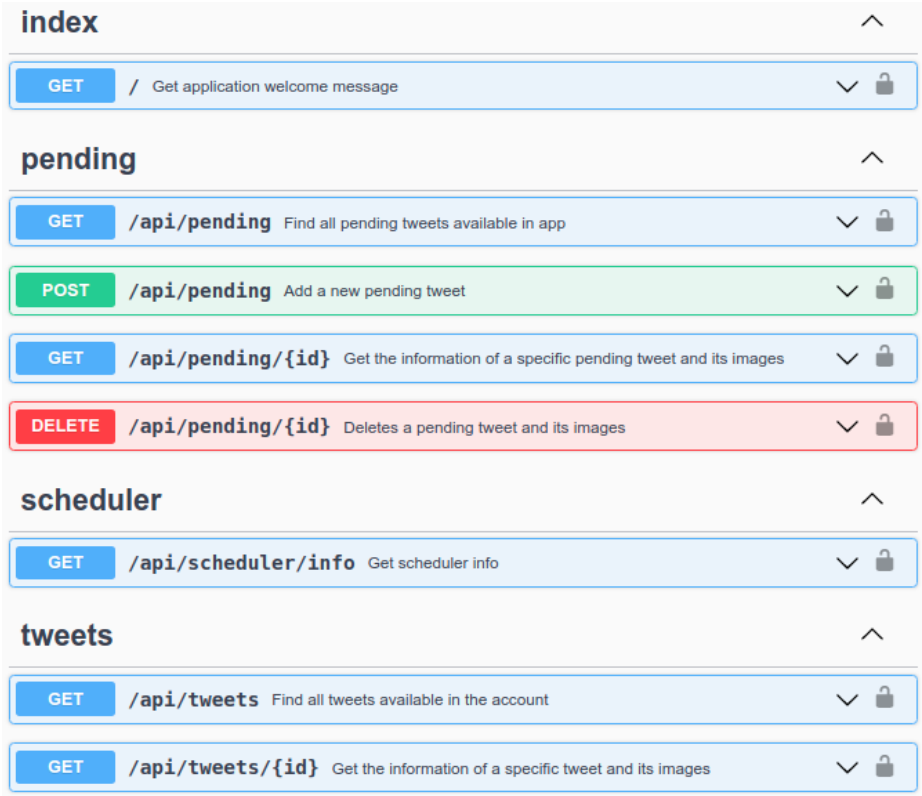


*Figura 7 - REST API expuesta por la aplicación siguiendo la especificación OpenAPI*

La API REST dispone de los siguientes endpoints:

- `GET /`: muestra un mensaje de bievenida a la aplicación.
- `GET /api/pending`: obtener todos los pendingTweet pendientes de publicar
- `POST /api/pending`: crear un nuevo pendingTweet
- `GET /api/pending/{id}`: obtener un pendingTweet por su ID.
- `DELETE /api/pending/{id}`: borrar un pendingTweet por su ID.
- `GET /api/scheduler/info`: obtener la información de la tarea programada que publica los tweets. Contiene si está activo o no (ya que se puede activar y desactivar), la espera inicial una vez arrancada la aplicación para realizar la primera ejecución de la tarea programada y la frecuencia de la tarea programada.

[19] https://twitter4j.org/en/index.html
[20] https://developer.twitter.com/en/portal/dashboard

- `GET /api/tweets`: obtener todos los tweets publicados por la aplicación en Twitter.
- `GET /api/tweets/{id}`: obtener un tweet por su ID.

Por tanto, con todo lo anteriormente descrito, a continuación, se muestra el diagrama de la aplicación desplegada en producción:

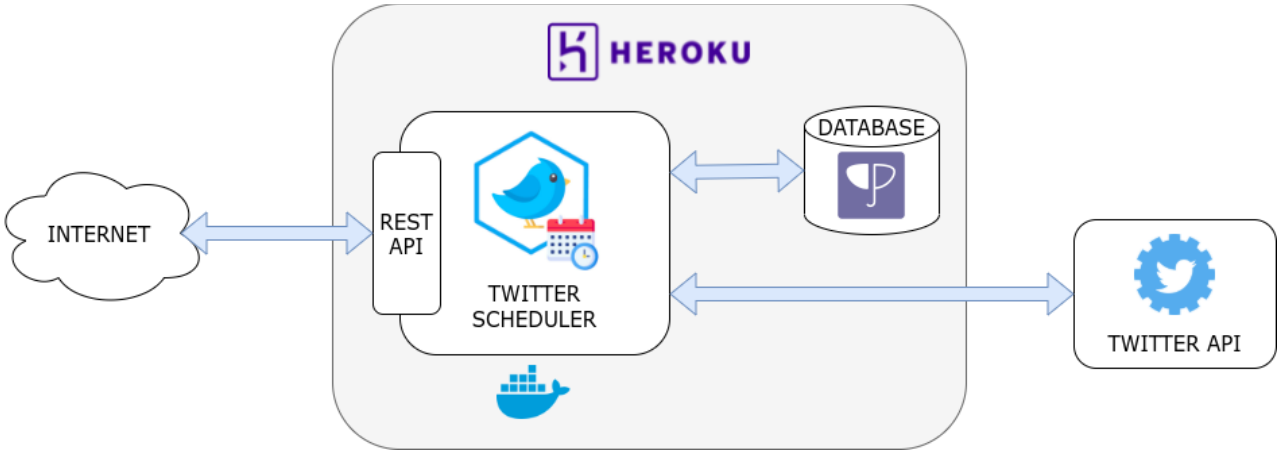


*Figura 8 - Aplicación desplegada en producción*

Los tweets son publicados en la cuenta **BlueOcean_TFM**[21]

## 4.3. FEATURE TOGGLES Y BRANCH BY ABSTRACTION

A continuación, con la aplicación ya en producción con la funcionalidad de poder publicar tweets dada una fecha exacta, se puede afrontar la tercera y última fase, la implementación de nuevas funcionalidades haciendo uso de las técnicas de Feature toggles y Branch by abstraction.

Para poder poner en práctica ambas técnicas, se ha usado la librería Togglz, integrada en la aplicación y que ofrece una interfaz web para activar/desactivar las Feature toggles:

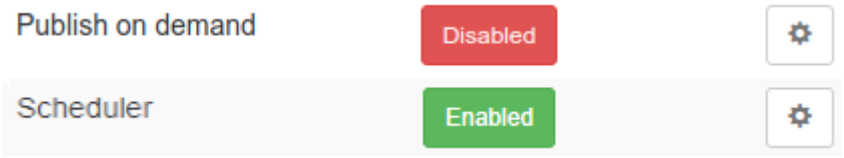


*Figura 9 - Estado de las feature toggles gestionadas por Togglz*

### 4.3.1. PUBLICACIÓN DE TWEETS PENDIENTES BAJO DEMANDA

La primera de las nuevas funcionalidades consiste en ofrecer la posibilidad de publicar de manera inmediata un tweet pendiente ya existente en la aplicación.

Esto implica cambios en el dominio ya que se necesita incluir en el aggregate root Tweet un nuevo value object, llamado `PublicationType` y que puede tomar dos valores:

- `SCHEDULED`: si el Tweet ha sido publicado mediante la tarea programada cuando ha sido la fecha indicada en su fecha de publicación.
- `ON_DEMAND`: si el Tweet ha sido publicado bajo demanda.

[21] https://twitter.com/BlueOcean_TFM

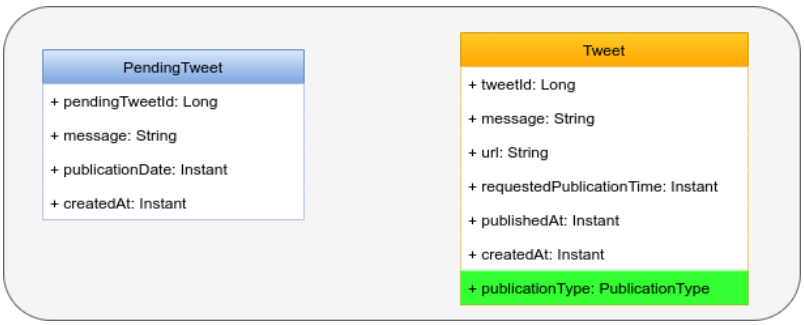


*Figura 10 - Cambios en el dominio para poder publicar tweets bajo demanda*

No solo hay que realizar cambios en el dominio, también es necesario modificar la base de datos para que se almacene esta información en una nueva columna de la tabla TWEET, por lo que es necesario crear un nuevo script de Flyway que haga ese cambio.

También es necesario la creación de un nuevo endpoint para poder ejecutar la publicación, en este caso se ha decidido que sea un `POST` al path `/api/pending/{id}/publish` y que devolverá el tweetId del Tweet publicado.

Desde la API REST se invocará al nuevo caso de uso y será el encargado de realizar la publicación del tweet, eliminarlo de los tweets pendientes y persistirlo en la tabla de tweets.

Se han seguido las técnicas de Feature toggle y Branch by abstraction, ocultando tras un toggle estos cambios, indicando que en caso de que se invocara al endpoint (que se ha ocultado de OpenAPI también) se devolviera un código http 405 (Method Not Allowed) implementando la funcionalidad en varios commits y siempre, después de cada uno de ellos, la aplicación se ha publicado correctamente en producción tal y como se puede ver en la siguiente imagen:

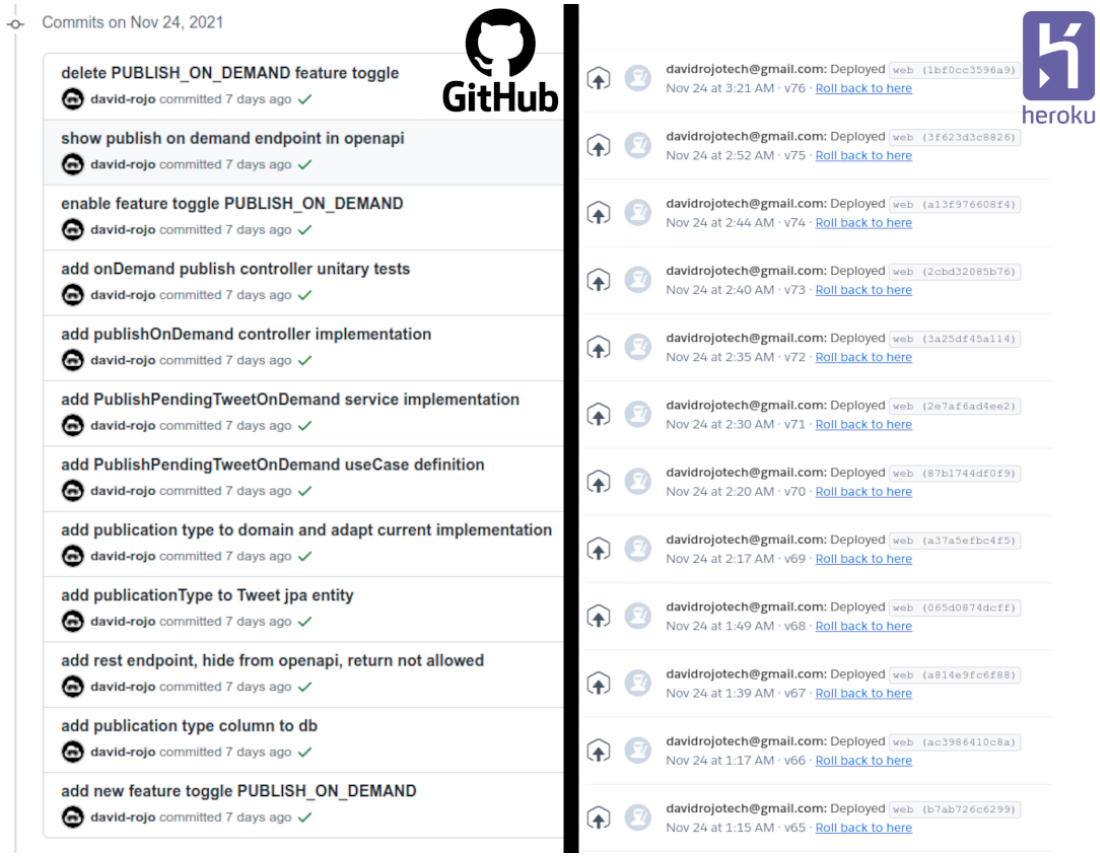


*Figura 11 - Relación de commits en GitHub y despliegues en Heroku*

Una vez completado el desarrollo se activa el toogle para que cuando se invoque al endpoint REST haga la publicación bajo demanda y una vez comprobado que funciona correctamente, se procede a eliminar el toggle y se habilita que el nuevo endpoint aparezca en la interfaz web de OpenAPI.

### 4.3.2. TWEETS CON IMÁGENES

La segunda de las nuevas funcionalidades consiste en ofrecer la posibilidad al usuario de que pueda incluir una o más imágenes en los tweets que va a publicar en la red social Twitter. Las imágenes tienen que estar disponibles a través de internet ya que, de cada imagen, se proporcionará su URL y será accedida para poderla publicar.

Este cambio impacta en mayor medida en el dominio, ya que incluye una nueva entity en cada aggregate root:

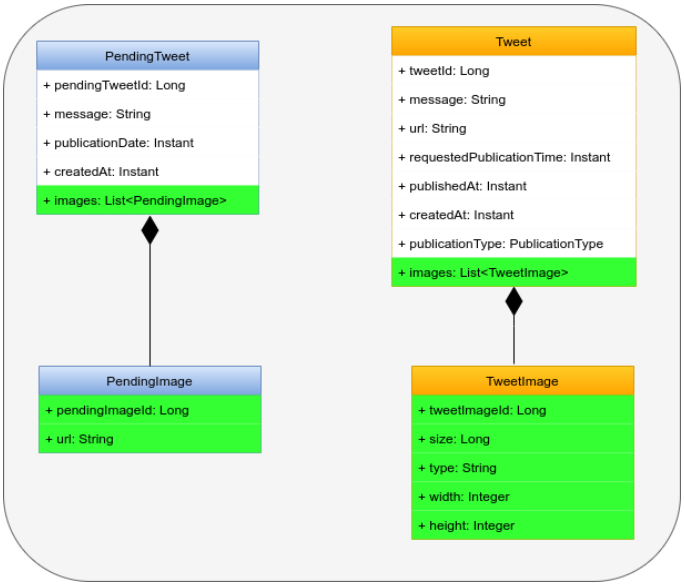


*Figura 12 - Cambios en el dominio para incluir los tweets con imágenes*

En PendingTweet, se incluye una lista (que puede estar vacía) de la entity **PendingImage** compuesta por:

- `pendingImageId`: clave autogenerada en la aplicación, de tipo numérico.
- `url`: URL en la que está accesible la imagen.

En Tweet, también se incluye una lista (que puede estar vacía) de la entity **TweetImage** compuesta por:

- `tweetImageId`: identificador numérico generado por la red social y es usado como clave en la aplicación.
- `size`: tamaño de la imagen en formato numérico.
- `type`: cadena de texto que representa el tipo de la imagen. Ejemplo: `image/jpeg.`
- `width`: ancho de la imagen en formato numérico.
- `height`: altura de la imagen en formato numérico.

Como en la anterior funcionalidad, no sólo basta con cambiar el dominio, sino que hay que aplicar cambios también en la base de datos, introduciendo dos nuevas tablas de imágenes PENDING_IMAGE y TWEET_IMAGE que están relacionadas de 1 a N con PENDING_TWEET y TWEET respectivamente, por lo que el diagrama de entidad relación de la base de datos queda de la siguiente manera:

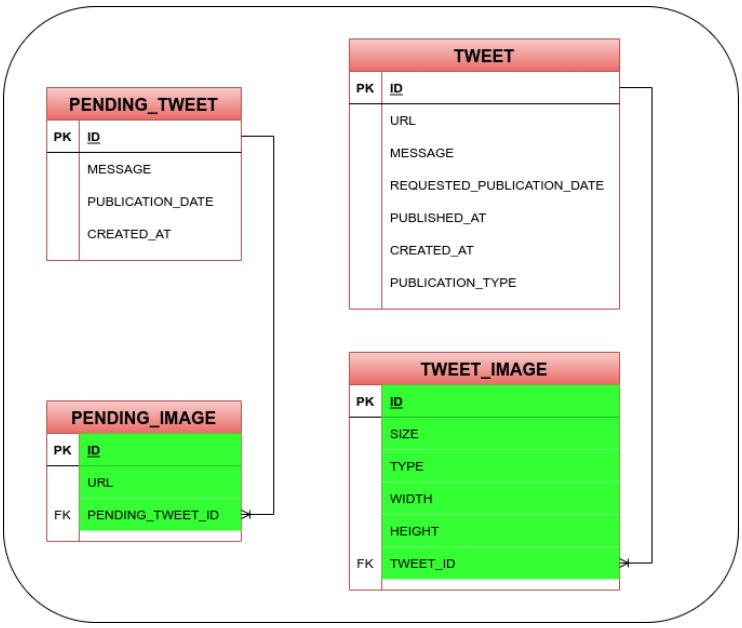


*Figura 13 - Cambios en base de datos para incluir los tweets con imágenes*

Nuevamente se han seguido las técnicas de Feature toggle y Branch by abstraction, ocultando tras un toggle estos cambios. En esta ocasión no hace falta crear un nuevo endpoint REST, ya que se usará el mismo endpoint que se utilizaba hasta ahora para crear PendingTweets, sin embargo, el cuerpo de la petición puede tener la lista de imágenes.

Se crea un nuevo toggle y se desactiva hasta que la implementación de la funcionalidad esté completa. El toggle es consultado en los siguientes puntos y si está desactivado, no se consulta si puede haber imágenes incluidas:

- Al crear un PendingTweet, al procesar la petición.
- Al devolver la respuesta de un PendingTweet creado.
- Al publicar un PendingTweet en Twitter en el servicio de publicación.
- Al devolver un Tweet ya publicado en Twitter.

A continuación, se incluye una imagen con el listado de commits que componen la implementación de esta funcionalidad y como se puede ver marcado con el recuadro rojo, hubo dos commits que fallaron y por tanto la aplicación no fue actualizada en producción.

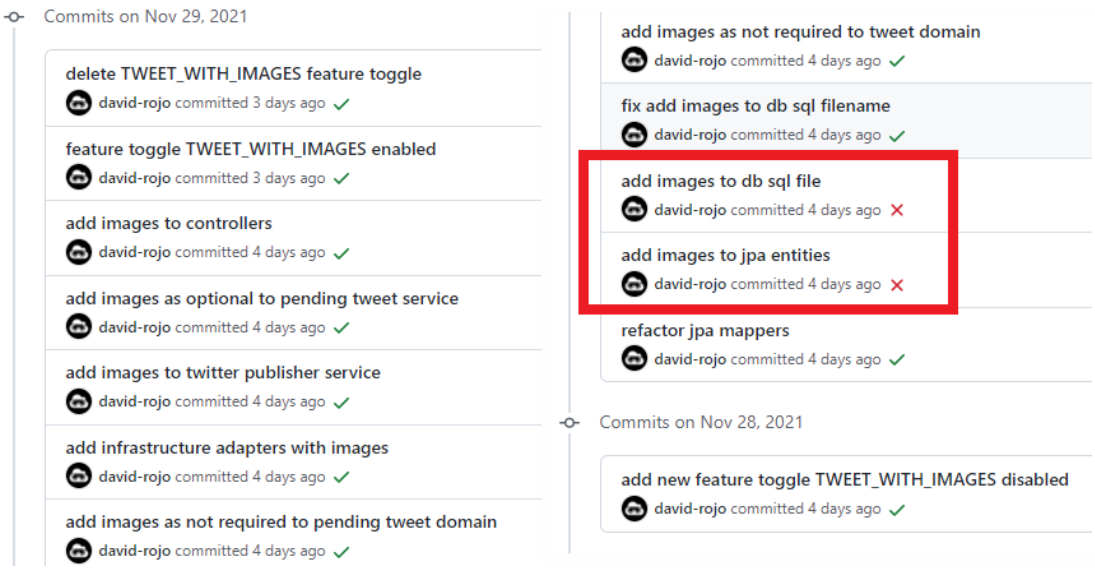


*Figura 14 - Commits para incluir los tweets con imágenes*

Los fallos fueron debidos a que primero no incluí en el commit el fichero de flyway para realizar los cambios necesarios en base de datos y el segundo fue, a que después de incluirlo, el nombre no era correcto ya que sigue el formato de `V<num_version>__nombre.sql` y había nombrado al fichero `V<num_version>_nombre.sql` (con un _ menos). Una vez corregido el error, el resto de commits fueron satisfactorios y los despliegues, correctos.

# 5. CONCLUSIONES

Considero que el hecho de tener disponible primero el entorno de integración y despliegue continuo con GitHub actions me ha ayudado mucho a realizar el proyecto mucho mas rápido que si hubiera invertido las fases, primero avanzando en la implementación y después haber configurado el entorno de integración y despliegue.

También me han sido de gran ayuda la gran cantidad de tests y de diferentes tipos (unitarios, integración y smoke test) para detectar errores y cuando se han producido saberlos identificar rápidamente. Los tests de primeras no se consideran necesarios, pero cuando ya estas acostumbrado a trabajar con ellos, cuando no los tienes parece que te falta algo y la velocidad de desarrollo y la calidad del código también se ven afectadas, por lo que considero que son indispensables en cualquier aplicación hoy en día.

El hecho de trabajar sobre la rama `main` directamente, es algo que, de primeras, personalmente me daba respeto ya que hasta ahora no había trabajado con este modelo de desarrollo y que cada vez que se hace un commit, se despliega una nueva versión.

Hasta tener una idea clara y no "ensuciar" el repositorio con commits, me abrí otro repositorio[22] en GitHub en mi cuenta personal con workflow de integración y despliegue continuo y también su aplicación Heroku paralela, digamos como entorno de "desarrollo" en el que fui haciendo pruebas y diversas ramas, que una vez validadas pasaba al repositorio de "producción".

Para poder aplicar satisfactoriamente Trunk-based development, es necesario disponer de una sólida infraestructura de desarrollo (el equipo de desarrollo tiene que ser capaz de poder ejecutar la aplicación localmente para así poder verificar la build), dominio de ciertas técnicas de diseño como Feature toggles o Branch by abstraction, así como técnicas avanzadas en el control de versiones, como cherry pick.

Ha sido mi primer contacto con Heroku y me ha parecido una plataforma muy interesante para realizar pequeñas pruebas de concepto, pero a nada que se quiera implementar algo profesional o que tenga cierta demanda de recursos, la capa gratuita se queda muy corta. En el desarrollo del presente proyecto, he tenido tres incidencias con Heroku:

1. Si la aplicación no está activa, Heroku la hiberna a los 30 minutos, aunque tengo la tarea programada de Spring para ejecutar el Scheduler, realizaba pruebas a publicar un tweet en una fecha concreta y no se publicaba. Pasada la fecha y al no verlo, tenía que acceder a algún endpoint de la aplicación, para que se arrancara de nuevo, entonces saltaba la tarea programada y el tweet era publicado.
2. Los primeros despliegues en Heroku, era tan sólo de la aplicación SpringBoot con los endpoints del Spring Actuator para comprobar que la aplicación se desplegaba satisfactoriamente y me respondía, ya tenía mensajes de error en los logs de la aplicación indicando que había excedido la quota de memoria. Estos errores han sido recurrentes desde entonces, pero no ha impedido que la aplicación funcionara.
3. La base de datos Heroku Postgres dentro de la capa gratuita, ofrece un máximo de 20 conexiones simultáneas y cuando hice la primera prueba de conectar la aplicación con la base de datos en Heroku, con sólo ejecutar los tests unitarios ya me dio error que había superado el máximo de conexiones permitidas. Lo pude arreglar limitando el pool de conexiones a base de datos mediante la propiedad:

[22] https://github.com/david-rojo/twitter-scheduler-dev

```
spring.datasource.hikari.maximum-pool-size=2
```

El uso de la arquitectura hexagonal y de Domain Driven Design, en una primera instancia es bastante tediosa y verbosa al añadir bastante complejidad, pero una vez superada esa primera parte menos amable y con el dominio claramente definido, el incluir nuevas funcionalidades y refactoring se simplifica en gran medida, ya que todo el código está muy separado. No quiero decir que sea algo trivial, pero considero que todo lo aplicado en el presente proyecto, ayuda a que un producto tenga una calidad óptima y sea sostenible en el tiempo.

Aún así, hay que analizar concienzudamente si este tipo de arquitectura es necesario o no, ya que en proyectos pequeños o de un alcance muy limitado, personalmente, considero que con una arquitectura de Modelo, Vista, Controlador (MVC) es mas que suficiente, al ser mas rápido de implementar y obtener resultados antes de cara al cliente.

Lo importante es tener conocimiento de todas las posibilidades que tenemos a nuestro alcance y saber elegir la idónea, la que se adapte mejor, para cada tipo de proyecto en los que participamos.

# 6. TRABAJOS FUTUROS

- Realizar el mismo Trabajo Fin de Máster, pero usando como lenguaje de programación Node para conseguir una implementación en JavaScript.
- Implementación de nuevas APIs para gestionar tweets: gRPC, GraphQL, colas (RabbitMQ/Kafka)
- Usar como base la aplicación actual y realizar una implementación del patrón Command Query Responsibility Seggregation (CQRS), aplicando diferentes modelos para leer y actualizar información, teniendo dos bases de datos independientes, por ejemplo, una base de datos relacional (MySQL, PostgreSQL) para actualizar la información y otra de tipo NoSQL (MongoDB, Apache Solr) para las lecturas de información.
- Almacenamiento de datos en la nube como AWS S3 o Google Cloud Storage
- Integración con otras redes sociales (Instagram, Facebook, TikTok)
- Permitir más tipos de contenidos en los tweets como, por ejemplo, videos.

# 7. BIBLIOGRAFÍA


- Eric Evans. *"Domain-Driven Design: Tackling Complexity in the Heart of Software"*. Addison Wesley, 2003
- Vaughn Vernon. *"Implementing Domain-Driven Design"*. Addison Wesley, 2013
- Robert C. Martin. *"Clean Architecture: A Craftsman's Guide to Software Structure and Design"*. Addison-Wesley, 2017
- Nicole Forsgren, Jez Humble, Gene Kim. *"Accelerate: The Science of Lean Software and Devops: Building and Scaling High Performing Technology Organizations"*. IT Revolution Press, 2018
- Paul M. Duvall, Steve Matyas, Andrew Glover. *"Continuous Integration: Improving Software Quality and Reducing Risk"*. Addison Wesley, 2007
- Jez Humble, David Farley. *"Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation"*. Addison Wesley, 2010
- Martin Fowler. *"UML Distilled: A Brief Guide to the Standard Object Modeling Language"*. Addison Wesley, 2003
- Sander Rossel. *"Continuous Integration, Delivery, and Deployment: Reliable and faster software releases with automating builds, tests, and deployment"*. Packt Publishing, 2017
- Adil Aijaz, Pato Echagüe. *"Managing Feature Flags: Deliver Software Faster in Small Increments"*. O'Reilly Media, Inc., 2018
- Ferdinando Santacroce. *"Git Essentials: Create, merge, and distribute code with Git, the most powerful and flexible versioning system available"*. Packt Publishing, 2017
- Craig Walls. *"Spring in Action"*. Manning Publications, 2019
- Craig Walls. *"Spring Boot in Action"*. Manning Publications, 2015
- John Carnell, Illary Sanchez. *"Spring Microservices in Action"*. Manning Publications, 2017
- Shekhar Gulati, Rahul Sharma. *"Java Unit Testing with JUnit 5: Test Driven Development with JUnit 5"*. Apress, 2017
- Regina Obe, Leo Hsu. *"PostgreSQL: Up and Running. A Practical Guide to the Advanced Open Source Database"*. O'Reilly Media, 2017
- Neil Middleton, Richard Schneeman. *"Heroku: Up and Running. Effortless Application Deployment and Scaling"*. O'Reilly Media, 2013
- Priscila Heller. *"Automating Workflows with GitHub Actions: Automate software development workflows and seamlessly deploy your applications using GitHub Actions"*. Packt Publishing, 2021
- Jeff Nickoloff, Stephen Kuenzli. *"Docker in Action"*. Manning Publications, 2019
- Sean P. Kane, Karl Matthias. *"Docker: Up & Running. Shipping Reliable Containers in Production"*. O'Reilly Media, 2018
- Harihara Subramanian, Pethuru Raj. *"Hands-On RESTful API Design Patterns and Best Practices"*. Packt Publishing, 2019
- Matthias Biehl. *"RESTful API Design: Best Practices in API Design with REST"*. API-University Press, 2016

# ANEXO 1: PETICIONES PARA CREAR TWEETS PENDIENTES

Hay que realizar un `POST` a `/api/pending` con el siguiente body:

- Sin imágenes:

```
{
  "message": "This is a test tweet without images",
  "publicationDate": "2021-12-17T10:00:00Z"
}
```

- Con imágenes:

```
{
  "message": "This is a test tweet with images",
  "publicationDate": "2021-12-17T09:00:00Z",
  "images": [
    {
      "url": "https://davidrojo.eu/images/tfm/1.jpg"
    },
    {
      "url": "https://davidrojo.eu/images/tfm/2.jpg"
    }
  ]
}
```

# ANEXO 2: RESPUESTA CUANDO SE CONSULTA UN TWEET YA PUBLICADO

Hay que realizar un `GET` a `/api/tweets/{id}` indicando el ID del Tweet:


```
{
    "id": 1465292923994611700,
    "message": "Pigmy seahorse",
    "url": "https://twitter.com/BlueOcean_TFM/status/1465292923994611715",
    "requestedPublicationDate": "11/29/21, 12:14 PM",
    "publishedAt": "11/29/21, 12:13 PM",
    "createdAt": "11/29/21, 12:13 PM",
    "publicationType": "SCHEDULED",
    "images": [
      {
        "id": 1465292922207801300,
        "size": 127798,
        "type": "image/jpeg",
        "width": 800,
        "height": 511
      }
    ]
  }
```