



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

Curso académico 2020/2021

Trabajo de Fin de Máster

Aplicación de planificación de redes  
sociales con TBD y CI/CD

Autores:

Andrea Colina Fernández  
Juan Manuel Guijarro Sánchez

Tutor:

Francisco de Asis Gortázar Bellas



# Índice

<b>Índice</b>	<b>2</b>
<b>Resumen</b>	<b>3</b>
<b>Introducción y objetivos</b>	<b>4</b>
Introducción	4
Objetivos	4
<b>Tecnologías</b>	<b>6</b>
Base de datos	6
Gestión de las imágenes	7
Comunicación con Twitter	7
Comunicación con Instagram	8
Inclusión de HTTPS	8
Swagger	8
Dockerfile	9
GitHub Actions	9
Heroku	9
<b>Trunk Based Development</b>	<b>12</b>
Características	12
Técnicas de desarrollo aplicadas a TBD:	13
Feature Toggles	13
Branch by Abstraction	14
<b>Integración y Entrega Continua</b>	<b>16</b>
Github Actions	16
Testeo automatizado	17
<b>Conclusiones y futuros trabajos</b>	<b>18</b>
<b>Bibliografía</b>	<b>21</b>

## Resumen

Este proyecto de fin de Máster se basa en aprender a trabajar siguiendo una de las metodologías de Git estudiadas durante el curso, Trunk Based Development. Para ello, hemos trabajado en el desarrollo de una aplicación que permite la interacción con diferentes redes sociales.

Esta estrategia de Git, TBD, tiene una rama principal donde se integran todos los cambios. Dichos cambios se realizan en ramas de corta duración, 1 o 2 días como máximo. Estas ramas salen de la rama principal, contienen commits con pequeños cambios y se integran en ella una vez se haya terminado el trabajo mediante Pull Request.

Las técnicas de Integración y Despliegue Continuo usadas para el despliegue de la aplicación en producción, unidas al testeo automatizado, y junto con el trabajo en pareja, somos dos desarrolladores trabajando de manera simultánea en el proyecto, nos ha hecho prestar especial atención a mantener siempre en la rama principal una versión estable y funcional del código.

Mediante la aplicación el usuario puede agregar contenido a las redes sociales de Twitter e Instagram, así como consultar e interacciones con el contenido ya existente. Para añadir valor al producto, se incorpora un programador para subir contenido a las plataformas, de manera que el usuario puede dejar preparado el post para que se suba a la plataforma en un momento exacto.

# Introducción y objetivos

## Introducción

TBD-SOCIAL-NETWORK-PLANNER se plantea como una aplicación para aprender a trabajar bajo la estrategia de Git de TBD. Durante el máster se han estudiado otras estrategias de trabajo con Git, como GitFlow, pero a nosotros nos llamó la atención especialmente Trunk Based Development. Nos planteamos este proyecto como un reto, ya que nos impone cambiar por completo el sistema de trabajo, el cual venimos utilizando tanto en el máster como en nuestros respectivos trabajos. Romper con la rutina y adentrarnos en una estrategia nueva nos va a permitir ver el desarrollo desde otra perspectiva, poder comparar con lo que ya conocemos y establecer una serie de pros y contras de esta estrategia.

También cabe señalar que el hecho de interaccionar con las redes sociales más utilizadas del momento le da un carácter muy atractivo al proyecto. Tenemos una infinidad de posibilidades de interacción con las mismas, así que nuestros límites están muy lejos. Por otro lado, nos atrae la idea de desarrollar una API en la que podamos aplicar los conocimientos y técnicas usadas en el máster. Entre ellas queremos incluir la evolución de base de datos o la integración de algún servicio que nos proporciona Amazon Web Services. También nos gustaría poder desplegar la aplicación en alguna plataforma que nos ofrezca una capa gratuita y se adapte a nuestras necesidades.

## Objetivos

El objetivo principal de este proyecto de fin de máster es, por un lado, conocer de primera mano las bases del Trunk Base Development (TBD) y por otro lado, conseguir aplicar varias de las tecnologías/herramientas aprendidas a lo largo del curso como Flyway con una BBDD PostgreSQL, integración y entrega continua con GitHub Actions, servicio de almacenamiento con S3 de Amazon Web Service y Heroku todo ello desarrollado en un proyecto Java con Spring.

Como objetivos específicos podemos señalar:

- Aprender a trabajar en equipo aplicando la estrategia de Trunk Based Development.
- Trabajar siguiendo las pautas que marca TBD: realizar todo el desarrollo sobre una rama principal e integrar en ella ramas de corta duración con commits pequeños, entre otras.
- Aprender a trabajar con las Actions de GitHub y a configurarlas según las necesidades del proyecto.
- Desplegar la aplicación de producción en Heroku, utilizando herramientas de CI/CD. De esta manera podremos tener una versión siempre levantada sobre la que poder testear los cambios gracias a la entrega continua.

- Implementar código nuevo con sus respectivos test unitarios y de integración, si procede. Creemos que es muy importante habituarnos a realizar test del software, ya que en el ámbito de trabajo, por falta de tiempo, suelen caer en el olvido.
- Aplicar Flyway para la evolución de la base de datos según vayamos incluyendo nuevas funcionalidades.
- Uso e integración de AWS para el almacenamiento de recursos.
- Investigar las APIs de terceros, que en este caso son: API de Twitter y API de Facebook, que es la que usa Instagram, e integrarlas en nuestra aplicación. Estudiar cómo se puede realizar la autenticación de terceros para que un usuario se pueda autenticar a través de nuestra aplicación y postear en su cuenta.
- Elegir una base de datos relacional que se encontrará alojada en Heroku.
- Configurar los diferentes entornos: desarrollo, producción, ...

# Tecnologías

Pasamos a detallar el stack tecnológico utilizado para el desarrollo del proyecto:

Spring es un framework con muchas características que nos ayuda a crear aplicaciones web para Java de una manera muy sencilla. Junto con Maven nos permite añadir el conjunto de dependencias necesario para poder construir nuestra aplicación.

## Base de datos

Para el desarrollo del proyecto optamos por una base de datos PostgreSQL. Queríamos utilizar alguna de las opciones que nos brindaba la plataforma de Heroku (estaba disponible tanto MySQL como PostgreSQL) y puesto que la de MySQL era la que más habíamos utilizado nos decidimos por seleccionar la de PostgreSQL. Sin embargo, para este tipo de proyectos en los que no se tiene una gran base de datos, no se necesitan consultas complejas y lo que se busca es la rapidez, la mejor opción habría sido MySQL.

Para gestionar el versionado de la BBDD utilizamos la herramienta Flyway, que nos aporta de una manera sencilla la información acerca de los cambios que se han hecho sobre el modelo de la base de datos mediante una tabla que indica el script que se hubiera ejecutado y la fecha de ejecución del mismo.

Por otro lado, decidimos seleccionar una BBDD H2 para el desarrollo en local, ya que es una base de datos que se levanta en memoria a la vez que arranca la aplicación y vimos que era una buena opción para este caso. Dicho esto, nos encontramos con un problema al ejecutar los scripts de migración en producción, ya que el perfil que tenemos para levantar la aplicación en local utiliza, como hemos comentado antes, una BBDD H2, y la sintaxis de esta, difiere en algunos aspectos respecto a PostgreSQL.

En los primeros scripts que fueron de creación de tablas no hubo problemas, pero cuando tuvimos que modificar la tabla para añadir unas constraints, al arrancar la aplicación en producción daba errores de sintaxis. Cuando se incluye la dependencia de Flyway, la aplicación habilita o deshabilita la migración de la base de datos en función de una propiedad 'spring.flyway.enabled'. Por lo tanto, decidimos crear otro perfil en Spring para poder probar en local estas migraciones de la base de datos antes de subir a producción. El resultado fue un perfil 'flyway' el cual requiere de una BBDD postgresQL levantada en local, que tiene activada la propiedad que hemos mencionado.

De esta forma tenemos un perfil para levantar rápidamente la aplicación y hacer las pruebas de lo que es la lógica de la aplicación con la base de datos H2 y otro para hacer las pruebas de los scripts de migración de la base de datos que es algo más lento y no es necesario cuando no se ha realizado ninguna modificación sobre el modelo de datos.

## Gestión de las imágenes

Tanto Twitter como Instagram permite la subida de imágenes en los posts. La primera red social que añadimos fue Twitter. Esta permite la subida de imágenes desde un archivo que se le pasa directamente desde la request y era opcional. Al no tener que hacer nada más con la imagen, decidimos guardar exclusivamente el nombre de la imagen para tener un control sobre si una publicación tenía o no una imagen asociada.

Posteriormente añadimos Instagram, la cual requiere obligatoriamente de una imagen para realizar la publicación y que esta imagen esté alojada en un servidor público ya que al crear el contenedor de la imagen que se va a subir se hace un curl de la url pasada en la petición y si no fuera pública daría un error. Fue aquí donde tuvimos que pensar la forma en la que poder almacenar las imágenes en el servidor y nos decidimos a usar la plataforma S3 de AWS.

Creamos una interfaz para abstraer la subida de imágenes y creamos la implementación con el código de S3.

- Commit del repositorio: [e3cb16d834e99b86cc4e0e3fb5d20f77c3d11f70](https://github.com/3cb16d834e99b86cc4e0e3fb5d20f77c3d11f70)

De esta forma procesamos la imagen que recibimos en la request con el servicio de S3, devolvemos el link de la imagen, y se la pasamos al servicio que postea en Instagram, guarda la información del post y le asocia la imagen que anteriormente hemos subido a S3.

## Comunicación con Twitter

Para la comunicación con la API de Twitter elegimos utilizar la librería Twitter4J. Al principio estuvimos intentando configurar las llamadas conforme a la documentación oficial pero debido a algunos problemas con la autenticación, decidimos decantarnos por el uso de esta librería. Dedicamos unos días a investigar acerca de dicha librería y nos pareció bastante fiable por la cantidad de información y cantidad de gente que la utilizaba y estable. Lo que hicimos a la hora de integrarla en nuestro proyecto es hacer otra capa para abstraer la librería por si en el futuro ya no diera soporte poder cambiar la implementación de una manera segura y rápida.

Finalmente la parte de la autenticación queda de la siguiente manera:

- Llamamos al endpoint GET /api/v1/twitter/auth
- Este reinicia el objeto de Twitter que hubiera anteriormente y nos devuelve una url para que hagamos login con la cuenta que queremos utilizar
- Al hacer el login se ejecuta el callback que hemos definido en la configuración de la cuenta de Twitter developer, que en nuestro caso llama a otro endpoint que tenemos declarado: /api/v1/twitter/auth/callback
- Es éste último el que asigna los valores de los tokens necesarios para poder interactuar con la API de manera externa

## Comunicación con Instagram

Al contrario que nos ha pasado con Twitter, para Instagram no encontramos ninguna librería que nos facilitara la comunicación con la API por lo que tuvimos que crearnos un servicio cliente para abstraer esta comunicación. En la implementación llamamos a los endpoints que hemos extraído de la documentación oficial mediante el uso de RestTemplate.

En cuanto a la autenticación de manera externa, comentar que es un aspecto contemplado como mejora a futuro, ya que por el momento se hace en dos pasos, debido a que no conseguimos configurar la aplicación (nos faltaron una serie de requisitos por implementar) de la misma manera que hicimos con Twitter y nos bloquea el callback. Hay que llamar al endpoint de autenticación, hacer el login con un dispositivo en el cual hay que acceder a la url y meter el código que nos devuelve la petición. Una vez hecho esto habría que llamar al endpoint de callback para finalizar la autenticación.

## Inclusión de HTTPS

Debido a los requerimientos de la API de Facebook, que es la que utiliza Instagram, nos vimos obligados a configurar el método seguro de HTTP. Ya que para hacer las pruebas del callback en local obligatoriamente teníamos que tener configurado el HTTPS. A pesar de haber configurado el HTTPS seguimos sin poder ejecutar la autenticación en un paso.

## Swagger

Para la documentación de la API hemos utilizado la herramienta Swagger, se importa de manera sencilla como dependencia Maven y posteriormente añadimos la información deseada mediante el uso de anotaciones.

<b>instagram-controller</b> Instagram Controller	<b>tweet-controller</b> Tweet Controller
<b>resource-controller</b> Resource Controller	<b>POST</b> /api/v1/twitter/tweet Post new tweet.
<b>tweet-controller</b> Tweet Controller	<b>GET</b> /api/v1/twitter/tweet/{id} Get tweet details by id.
<b>twitter-actions-controller</b> Twitter Actions Controller	<b>DELETE</b> /api/v1/twitter/tweet/{id} Delete tweet by id.
<b>twitter-authentication-controller</b> Twitter Authentica	<b>POST</b> /api/v1/twitter/tweet/schedule Schedule new tweet to post it at concrete time.
	<b>GET</b> /api/v1/twitter/tweets Get all tweets of account.

Interfaz gráfica de Swagger. Diferentes controladores. Vista de detalle de los endpoints del Twitter Controller.

Después de haber implementado todo lo referente a Twitter, nos planteamos utilizar el enfoque de API-First para el desarrollo de la parte de Instagram, este enfoque tiene claras ventajas cuando la API va a ser consumida por un cliente externo, o simplemente para



obligarnos a hacer un primer análisis antes de ponernos a implementar la lógica sin saber que va a recibir y devolver nuestro endpoint. Pero, dado que ya habíamos empezado y que no teníamos parte cliente que consumiera nuestra API, decidimos seguir con el enfoque tradicional, eso sí, analizando bien todos los requisitos antes de comenzar el desarrollo

## CI/CD

Vamos a hablar ahora de cómo hemos configurado las distintas componentes del proyecto para hacer posible la integración y entrega continua. Dichas partes son tres que vamos a detallar a continuación:

### A. Dockerfile

En cuanto a la generación de la imagen, para poderla subir tanto a DockerHub como a Heroku Registry, hemos generado un fichero Dockerfile en el cual indicamos la imagen padre que vamos a utilizar para generar nuestra propia imagen, el jar de la aplicación que tiene que copiar y el comando que tiene que ejecutar la imagen al arrancarla:

- Fichero del repositorio: [Dockerfile](#)

### B. GitHub Actions

Por otro lado nos encontramos las GitHub Actions, podemos configurarlas en función de las necesidades que tengamos. En nuestro caso, puesto que estamos trabajando con TBD y queremos asegurarnos que nuestra rama principal siempre sea estable, tenemos dos actions configuradas, una que se ejecuta al hacer una pull request sobre main, que lanza los tests y ejecuta el build, y otra cuando se ejecuta cuando se hace un push a main que además de los jobs anteriores, publica la imagen tanto en DockerHub como en Heroku Registry como hemos comentado antes, y posteriormente despliega la aplicación en Heroku.

Estas actions se ejecutan en máquinas de ubuntu en las cuales configuramos java, construimos la imagen, nos logueamos en docker para subir la imagen, instalamos el cliente de Heroku y realizamos el despliegue de la aplicación.

### C. Heroku

Es la plataforma como servicio (PAAS) que hemos seleccionado para alojar nuestra aplicación en la nube debido a las facilidades que proporciona, ya que tiene una documentación oficial muy clara y muchos usuarios utilizan dicha plataforma, por lo que nos ha sido muy fácil solventar las dudas que nos han ido surgiendo.

Una de las ventajas que tiene Heroku es su gran variedad de add-ons, entre los cuales podemos encontrar bases de datos, sistemas de monitoreo, sistemas de logs, etc.

Personal > ais-tbd-social-networks ☆ Open app More

Overview Resources Deploy Metrics Activity Access Settings

Installed add-ons \$0.00/month [Configure Add-ons](#)

Heroku Postgres Hobby Dev postgresql-tetrahedral-49592

Dyno formation \$0.00/month [Configure Dynos](#)

This app is using free dynos

web java -jar /opt/webapp.jar ON

Latest activity [All Activity](#)

- heroku-postgresql: Update DATABASE by heroku-postgresql Yesterday at 1:47 PM · v104
- andreajuanmaurjc@gmail.com: Deployed web (13587f5942c8) Dec 2 at 8:30 PM · v103
- andreajuanmaurjc@gmail.com: Deployed web (11fc8e669d3f) Nov 30 at 1:20 AM · v102
- andreajuanmaurjc@gmail.com: Deployed web (d56a4c253c2c) Nov 28 at 11:30 PM · v101

Pantalla principal de nuestra aplicación en Heroku. Se muestran los add-ons, la actividad reciente y dynos.

Nosotros solo necesitamos tener una base de datos desplegada en la nube, por lo que seleccionamos Heroku Postgres en su versión gratuita la cual cubría nuestras necesidades.

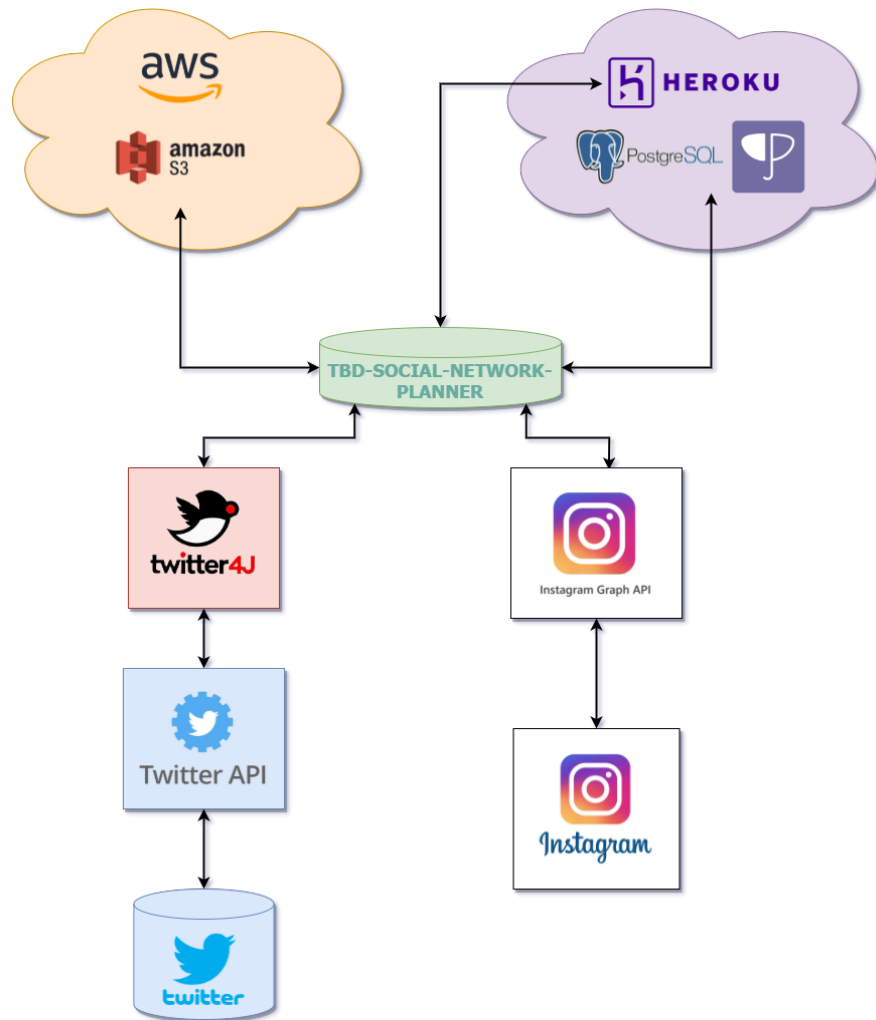
Por otro lado tiene una interfaz muy intuitiva y permite familiarizarse muy rápido con el entorno. Hemos utilizado el sistema de configuración de variables que proporciona para almacenar toda la información que debe ser privada y debe quedar fuera de los ficheros de configuración. En el entorno de pruebas informamos o mockeamos las propiedades que sean necesarias para el correcto funcionamiento, pero en el perfil de 'prod' que es el que se ejecuta en Heroku, ninguna de estas variables está informada.

Config Vars [Hide Config Vars](#)

ACCESS_TOKEN	14[REDACTED]SI	<a href="#">✎</a> <a href="#">✕</a>
ACCESS_TOKEN_SECRET	kt[REDACTED]k:	<a href="#">✎</a> <a href="#">✕</a>
AWS_ACCESS_KEY_ID	AK[REDACTED]	<a href="#">✎</a> <a href="#">✕</a>
AWS_S3_BUCKET_NAME	tbd-social-network	<a href="#">✎</a> <a href="#">✕</a>

Algunas variables de configuración almacenadas en Heroku.

Como resumen a este capítulo aportamos un esquema donde se muestra una visión general de la aplicación, las tecnologías aplicadas en su desarrollo así como las comunicaciones que se establecen con las APIs de terceros.



# Trunk Based Development

El principal objetivo del proyecto es aprender a trabajar siguiendo la estrategia de Trunk Based Development. Esta estrategia permite la entrega continua de código realizando múltiples subidas a la rama principal durante varias veces al día. La versión de la rama principal debe ser estable y no se debe romper nunca la build del código durante las subidas. A continuación hablaremos de los puntos comunes o características de esta estrategia.

## Características

1. **La rama principal:** TBD se caracteriza por tener una rama principal de larga duración denominada 'trunk'. Sobre esta rama se integran todos los cambios. En TBD no existen ramas secundarias como 'develop' o 'qa', sino que el código reside en la rama principal y todos los cambios se vuelcan directamente sobre ella. En nuestro caso particular, esa rama principal se llama 'main'.
2. **Ramas secundarias:** A partir de 'trunk' se sacan las ramas sobre las que trabajan los desarrolladores. Estas ramas se caracterizan por su carácter efímero. Lo ideal es que se haga commit sobre 'trunk' de manera frecuente, así las versiones de las ramas no se distancian en el tiempo de la última versión de trunk, evitando posibles conflictos. La duración de las ramas de desarrollo no es superior a uno o dos días.
3. **Commits:** Se trabaja realizando commits pequeños en ramas secundarias. Todo commit debe contener código funcional y debidamente testeado, para evitar posibles errores en la subida. Esta metodología cumple con el principio de la entrega continua, ya que los desarrolladores realizan subidas de manera periódica sobre la rama principal.
4. **Organización del trabajo:** Dependiendo del tamaño del equipo se pueden realizar commits directamente sobre trunk o bien si el tamaño del equipo crece se debe realizar el desarrollo en ramas secundarias de corta duración y mergearlas a la rama principal mediante Pull Request. De esta manera se evitan posibles conflictos con otras funcionalidades que se estén desarrollando de manera simultánea, ya que estamos entregando código constantemente.
5. **Integración y entrega continua:** Esta metodología favorece el uso de técnicas de integración y entrega continua. Se ejecutan una serie de procesos sobre el código antes de su subida a producción que nos garantizan el correcto funcionamiento del mismo, así como el despliegue de la nueva versión en producción una vez subido el código a la rama principal.

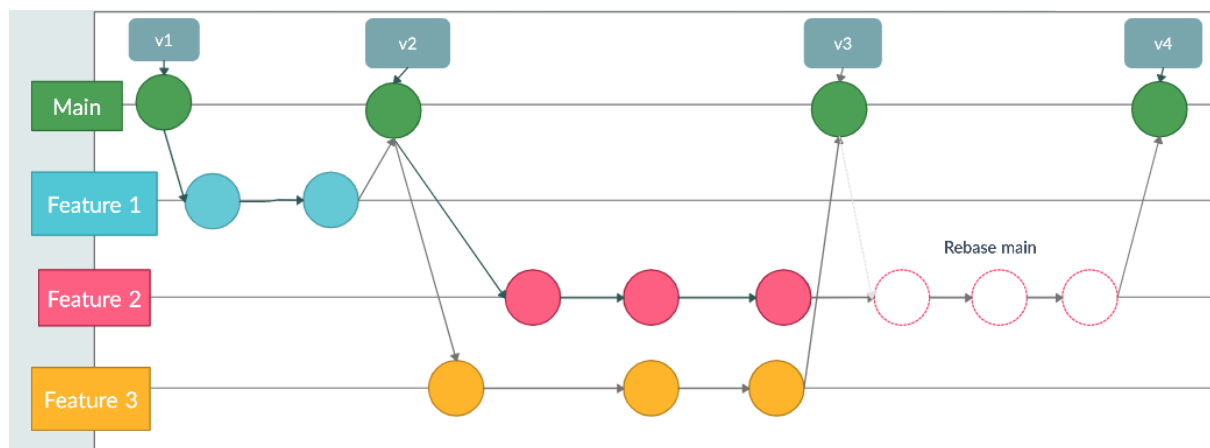
Durante el desarrollo de nuestra aplicación hemos usado una rama principal llamada *main*. Todas las ramas de trabajo han salido a partir de esta y han sido de corta duración. La vida de las ramas se ha visto directamente afectada por el tiempo de desarrollo de la

funcionalidad así como la disponibilidad del desarrollador para dedicarle tiempo a la misma. A veces dicha duración ha sido de horas y otras veces de un par de días. Se ha intentado trabajar de forma que se evitasen los conflictos, aplicando buenas prácticas como realizar *rebase* de la rama main cuando la rama secundaria se ha visto superada por la rama principal.

En cada commit se incluyen pocos cambios, siempre que sea posible. Nuestra idea es dejar un repositorio en la medida de lo posible “didáctico”, por lo que hemos evitado el uso de *squash* para agrupar todos los commits de una rama en uno solo.

Para subir el código a producción hemos usado Pull Request de rama de trabajo sobre main. Remarcar que no hemos incluido como requisito en la Pull Request la revisión del código ni un aprobado por parte de otro compañero para poder subir el código, aunque pensamos que es lo ideal y lo más lógico. La disponibilidad y horarios de los desarrolladores implicados nos ha impedido una mejor organización, ya que el trabajo se demoraba en complejidad y tiempo.

Se han aplicado técnicas de Integración y Entrega Continua utilizando las Actions de Github.



Esquema de ramas cortas con múltiples commits usando TBD. Uso de rebase.

## Técnicas de desarrollo aplicadas a TBD:

### A. Feature Toggles

También conocidos como Feature Flags, es una técnica usada para desarrollar código sin cambiar el funcionamiento de la aplicación de manera directa. Por ejemplo, se puede habilitar una funcionalidad para realizar el testeo en producción de manera ocasional, o habilitarla y deshabilitarla según las necesidades del momento. Con el tiempo se han ido desarrollando librerías que permiten y facilitan el uso de esta técnica. En nuestro caso hemos usado FF4J, para Java. Básicamente consiste en incluir un toggle sobre el cuál protegeremos nuestro código nuevo. Dependiendo del valor de ese toggle (activado/desactivado) se realizará la ejecución del código nuevo o se ejecutará el código existente. Por ejemplo, se activará ese toggle para realizar el testeo oportuno de la funcionalidad en producción pero para los usuarios de la aplicación permanecerá desactivado. El valor de estos toggles se

controla a través de una [interfaz gráfica](#) y tienen un valor por defecto asignado en código.

A la hora de realizar el desarrollo hemos utilizado feature toggles para incluir nuevas funcionalidades de manera segura desplegando primero en producción a modo de testeo. A continuación se expone brevemente un caso práctico que se puede encontrar en el código de la aplicación:

1. Primero se crea el feature flag, que utilizaremos posteriormente para proteger el nuevo código. Como podemos apreciar su estado por defecto es desactivado. De esta manera, la aplicación arrancará con dicha funcionalidad desactivada, y se activará de manera manual por el usuario.
  - Commit del repositorio: [d5ff7feb07be4befe9e0e55a4ea62e9c1c0beafe](#)
2. Actualizamos el endpoint protegiendo la llamada al nuevo servicio por el feature flag. Para todo usuario que no tenga el flag activado el endpoint nuevo se encuentra en estado de desarrollo, por tanto, no podrá hacer uso de la nueva funcionalidad. Mientras que si activamos el flag podremos probar el código nuevo en el entorno de producción.
  - Commit del repositorio: [88d3c79518c68ceef9b74a88b4b762be7ad8e6e0](#)
3. Por último, cuando la funcionalidad se ha testado en entornos de producción y está lista para su uso, se elimina el flag y el código que la protege.
  - Commit del repositorio: [318bddd0c425cc42201d15735f87db669ea5acf6](#)

## B. Branch by Abstraction

Durante el desarrollo del proyecto también hemos aplicado la técnica de *Branch by abstraction*. Esta técnica se caracteriza por abstraer la implementación de una funcionalidad a través de una interfaz que nos permita intercambiar la implementación de dicha funcionalidad, de manera que ambas coexistan en la aplicación de una forma segura.

En nuestra aplicación se realizó una primera fase de desarrollo de la funcionalidad que interacciona con Instagram. En esta primera fase se desarrolla en controlador y el servicio. En concreto se han utilizado llamadas REST a la API en cuestión, usando la clase de Java *RestTemplate*. Tanto las llamadas al cliente como la lógica de negocio necesaria para extraer la información requerida para obtener y realizar las operaciones pertinentes se encuentran alojadas en el servicio.

Para abstraer todo lo relacionado con la interacción con la otra API se decide crear una interfaz con las operaciones de Instagram que se usaran en el servicio: login, crear un post, obtener información de un post, ... etc. A continuación se crea la clase cliente que implementa dicha interfaz. Se transfiere el código referente a las llamadas REST a la API de Instagram a la nueva clase cliente, así como los correspondientes tests de los métodos implementados.

Una vez que tenemos abstraída la interacción con el cliente en la nueva clase cliente se procede a la inyección en el servicio. Para realizar este cambio hacemos uso nuevamente de los Feature Toggles. De esta manera ponemos en producción el nuevo cliente, que se encuentra protegido con el toggle. Podemos decir, que ambos coexisten de manera temporal en el Servicio. Realizamos las pruebas correspondientes a la nueva clase activando y desactivando el toggle.

Una vez que nos cercioramos de que la clase está correctamente implementada y que la aplicación no se ve afectada por la abstracción procedemos a la eliminación del código antiguo dejando únicamente las llamadas al cliente, junto con la lógica correspondiente en cada caso. Eliminamos también el toggle creado para esta abstracción y modificamos los tests del servicio, ya que ha habido modificaciones sobre el mismo.

Commits significativos en el proceso:

- Creación del toggle: [c265e7ae3b3b084e3fa7b6bb4ef78aaf51d5e4d4](#)
- Creación de la interfaz de cliente de Instagram: [1e14d01cd2aab52324293809d0c967cb5ed9d344](#)
- Transferencia del código del servicio al cliente: [2d1a9e223ac6b8a4afd1b8059c12041d70060784](#)
- Integración del cliente en el servicio, aplicando el uso del toggle que creamos anteriormente: [200db794475c0f90ede238584e62e6bfdb7f0315](#)
- Eliminación del flag y del código antiguo que se encuentra en el servicio: [18cdf41677da959ce3385ad0960abd282d23769e](#)
- Inclusión de nuevos tests: [b72875d65704ee5d256f6566d92a65ddf22251fa](#)

# Integración y Entrega Continua

La integración y entrega continua (CI/CD) va de la mano junto al Trunk Based Development, por tanto y como no podía ser de otra manera, hemos procurado incluir estas técnicas en nuestro proyecto. Para ello fijamos diferentes bases sobre las que trabajar o aplicar a nuestro trabajo en el día a día.


## 1. Github Actions

Como ya hemos mencionado anteriormente, son una herramienta ofrecida por GitHub, que nos permiten ejecutar diferentes acciones sobre nuestro código, desde realizar la build, ejecución de tests, despliegue de la aplicación...


*Workflow* es como se denomina al conjunto de trabajos que se ejecutan al ocurrir un evento en el repositorio de GitHub. Se han definido dos workflows para este proyecto.


[main-pr](#): El primer workflow se ejecutará cada vez que se realice una Pull Request de una rama secundaria sobre la rama principal, main.

Add more commits by pushing to the **update-readme** branch on **MasterCloudApps-Projects/tbd-social-networks-planner**.



✓ All checks have passed  
2 successful checks [Hide all checks](#)

✓  Main PR workflow / Run test (pull\_request) Successful in 1m [Details](#)

✓  Main PR workflow / Build Java application (pull\_request) Successful in 1m [Details](#)

✓ This branch has no conflicts with the base branch  
Merging can be performed automatically.

Merge pull request

You can also [open this in GitHub Desktop](#) or view [command line instructions](#).

Ejecución de los steps del workflow main-pr.yml antes de realizar el merge a main.

Este workflow contiene dos trabajos:

- Ejecutar todos los tests.
- Construir la build.

Consideramos estos dos trabajos como requisitos mínimos para poder integrar el código en la rama principal.

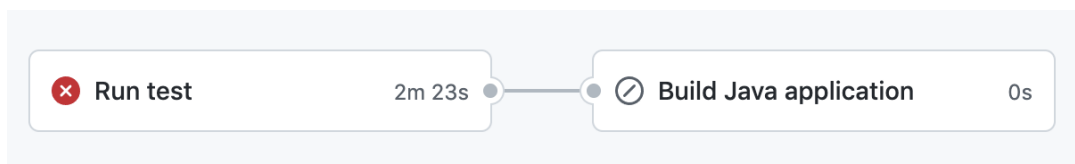
**main-pr.yml**  
on: pull\_request

```
graph LR; A[✓ Run test 1m 29s] --> B[✓ Build Java application 1m 49s]
```

Ejecución satisfactoria del workflow main-pr.yml.

16



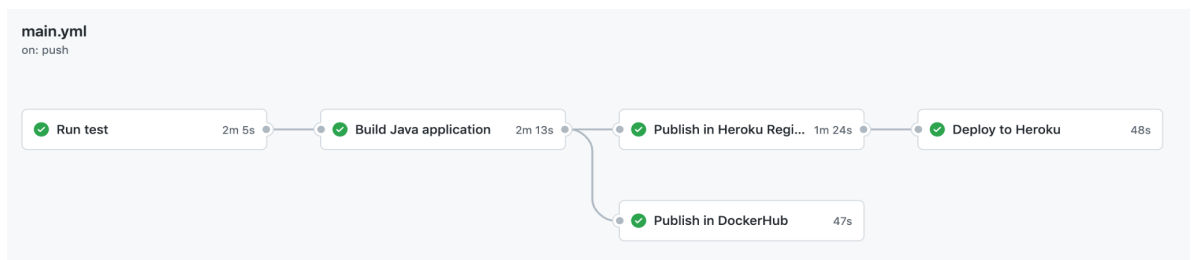


Fallo en la ejecución de los test en el workflow main-pr.yml

[main](#): Este workflow se ejecuta cada vez que se hace un commit sobre la rama principal, *main*. De manera que cada vez que se haga un commit de merge, tras una pull request se ejecutará este workflow. En este workflow se realizan varias comprobaciones acerca del estado del código, así como la publicación de la imagen correspondiente en DockerHub y el despliegue y publicación de la aplicación en Heroku.

Trabajos del workflow:

- Ejecución de los tests unitarios.
- Construir la build.
- Publicar en DockerHub
- Publicar en Heroku Registry
- Despliegue de la nueva versión de la aplicación en Heroku.



Ejecución satisfactoria del workflow main.yml con el correspondiente despliegue en Heroku y publicación en DockerHub.

## 2. Testeo automatizado

La entrega de código se hace con relativa frecuencia, ya que las ramas no deben ser de larga duración. Esto permite al desarrollador detectar los posibles conflictos con otras funcionalidades lo antes posible. Para trabajar con mayor seguridad, se realizan test del código nuevo. Nos pusimos como objetivo incluir test de las funcionalidades principales, es decir, de Instagram y Twitter. Para ello se han desarrollado test unitarios de los clientes y servicios, y test de integración de los controladores.

Se han desarrollado un total de 154 tests, entre test unitarios y de integración. Como resumen del reporte de Coverage de los test podemos señalar:

- A nivel de proyecto: 92% de coverage a nivel de clases, 67% de métodos y 76% de líneas.

- Funcionalidad de Instagram: 95% de coverage a nivel de clases, 68% de métodos y 80% de líneas.
- Funcionalidad de Twitter: 92% de coverage a nivel de clases, 72% de métodos y 83% de líneas.

Estos test se ejecutarán de manera automática, como hemos visto anteriormente, gracias a las Actions de GitHub. Por lo tanto, es importante mantener un alto nivel de coverage de la aplicación, lo que nos aporta seguridad a la hora de subir código nuevo.

## Conclusiones y futuros trabajos

Una vez finalizada esta parte de desarrollo de la aplicación, que para nada es un producto acabado, podemos sacar algunas conclusiones y reflexionar acerca de lo que hemos aprendido.

Trabajar durante unos meses aplicando Trunk Based Development, nos ha hecho cambiar por completo la manera de pensar, desde el momento que sacas la rama de trabajo hasta el momento en que subes el código a la rama principal. Durante todo este recorrido podemos hablar de un periodo de adaptación a la estrategia de Git TBD. Durante al menos unas dos semanas, se camina con pies de plomo, revisando constantemente los cambios, haciendo commits y pensando si los cambios se podrían haber dividido en commits más pequeños o quizás el commit contenga demasiada poca información. También hubo bastantes dudas acerca de cuándo integrar los Feature Toggles. Realmente no “veíamos la necesidad” porque nuestra mentalidad era: lo testamos en local y funciona, no vemos la necesidad de aplicar un feature toggle. Poco a poco, y en cierta medida obligándonos a usarlos, nos vamos dando cuenta de la importancia de incluir estos toggles y sus ventajas: esa sensación de ir sobre seguro a medida que se integran cambios gracias a ellos. Al final, realizar múltiples commits, utilizar los feature toggles o trabajar en ramas de corta duración se vuelve parte del día a día. Y no te planteas otra manera de trabajo.

Como en todos los casos, también hay aspectos no tan positivos que resaltar. Para empezar, hablar del tema de las versiones, tanto las imágenes como de la versión del código. Durante una primera fase, la de adaptación, no nos acordamos de subir la versión del código, ya que el código que se sube a main es una release, por tanto, las imágenes se sobrescriben durante todo este tiempo. Nos mentalizamos de subir la versión de la aplicación según correspondiera, pero tampoco funcionó. Podemos concluir que el tema de las versiones ha sido un poco caótico, ha pasado por diferentes fases y no hemos dado con una manera de trabajar que nos permitiese llevar un cierto orden respecto a ellas.

Por otra parte, señalar que se realizó una configuración inicial de las GitHub Actions que hemos usado durante todo el desarrollo. Esto nos ha permitido trabajar sobre seguro, ejecutando los test, realizar la build, etc. Como problema, en el workflow que se ejecuta sobre los commits en main se incluyeron las clases de test a ejecutar de manera manual, ya que no queríamos que se ejecutasen los tests de integración, porque conllevan más tiempo y ya han sido ejecutados en el workflow previo. Como el error humano existe, olvidamos durante cierto tiempo incluir las nuevas clases de tests que íbamos creando, suponiendo que se estaban ejecutando cuando no era así.

Acciones a tomar en un futuro:

- Respecto a las versiones: probablemente la opción más rápida y segura es usar la referencia del commit para etiquetar la versión de los artefactos. También nos planteamos alguna manera de automatizarlo en GitHub Actions pero no encontramos una manera de indicar si el número a incrementar era el menor, el mayor o el patch number.
- Respecto a incluir solo los test unitarios y no los de integración: investigar y revisar la manera de indicar en el GitHub Actions que no incluya los test

**\*\*IntegrationTest.class.** De manera que el desarrollador no tenga que añadir manualmente estas clases.

- Gestión de usuarios: habilitar la gestión de usuarios para que varias personas puedan estar a la vez usando la aplicación.
- Configurar el callback de Facebook: terminar de configurar el proyecto de Facebook para poder habilitar el callback una vez que el usuario da permisos a la aplicación. Este proceso es largo e imposible de completar en los plazos establecidos por los requerimientos de Facebook.
- Branch by Abstraction del cliente de Twitter: nos planteamos de cara a un futuro sustituir el cliente de Twitter, que en este caso usamos la librería Twitter4J por llamadas rest a la API, ya que usar esta librería nos crea una dependencia, que a la larga podemos evitar. Para ello habría que aplicar el patrón branch by abstraction sobre el cliente y proceder a realizar la nueva implementación, para posteriormente sustituir el cliente de la librería por el de llamadas Rest.

## Bibliografía

Fowler, M. (7 de enero de 2014) Branch by Abstraction.

<https://martinfowler.com/bliki/BranchByAbstraction.html>

Santacroce, F. (2015) Git Essentials. Editorial Packt.

Salesforce.com (2021) Java | Heroku Dev Center.

<https://devcenter.heroku.com/categories/java-support>

Red Gate Software Ltd. (1999 - 2020) Flyway Documentation.

<https://flywaydb.org/documentation/>

Macero. M. (8 de Noviembre de 2020) Learn Microservices with Spring Boot. Editorial Apress.

Hammant, P. (2017-2020). Trunk Based Development: Feature Flags.

<https://trunkbaseddevelopment.com/feature-flags/>

Hammant, P. (2017-2020). Trunk Based Development: Branch by Abstraction.

<https://trunkbaseddevelopment.com/branch-by-abstraction/>

Catamorphic Co. (2021) Feature Flag Guide.

<https://featureflags.io/>

Hammant, P. y Smith, S. (2017) Trunk-Based Development And Branch By Abstraction.

Valero Sanchez, J. C. (24 de Junio de 2021) Feature Flags with Spring.

<https://www.baeldung.com/spring-feature-flags>

Twitter, Inc. (2021) Twitter API Documentation.

<https://developer.twitter.com/en/products/twitter-api>

Meta. (2021) Instagram Graph API Documentation.

<https://developers.facebook.com/docs/instagram-api/>

Robert C. Martin (17 de septiembre de 2017) Clean Architecture.

Robert C. Martin (13 de marzo de 2011) The Clean Coder.

Baeldung. (23 de octubre de 2021) Setting Up Swagger 2 with a Spring REST API.

<https://www.baeldung.com/swagger-2-documentation-for-spring-rest-api>

Sarcar, V. (6 de diciembre de 2018) Java Design Patterns.

Vitale, T. (9 de julio de 2017) How to enable HTTPS in a Spring Boot Java application.

<https://www.thomasvitale.com/https-spring-boot-ssl-certificate/>

GitHub, Inc. (2021) Workflow syntax for GitHub Actions.

<https://docs.github.com/en/actions/learn-github-actions/workflow-syntax-for-github-actions>

Shazin Sadakath, M. (30 de Julio de 2018) Spring Boot 2.0 Projects. Editorial Packt.