



UNIVERSIDAD DE MENDOZA

FACULTAD DE INGENIERÍA

INGENIERÍA EN INFORMÁTICA

TRABAJO FINAL

“Ingeniería de Software Aplicada a Cloud Computing”

Alumno: Facundo G. Martín

Legajo: 6008

Profesor asesor: Fabián Contigiani

Fecha: 29 de enero del 2020



UNIVERSIDAD DE MENDOZA FACULTAD DE INGENIERÍA

RESUMEN

En el desarrollo de aplicaciones es muy común que se haga foco en la escritura rápida del código con el fin de implementar nuevas funcionalidades lo antes posible. Al hacer esto se dejan de lado temas de igual o mayor importancia que comprenden todo lo que pasa alrededor del código y no al código en sí, como la metodología general del desarrollo, que comprende desde cómo se escribe el código a como se lo lleva a un entorno productivo, como se prueba el código y la aplicación o como se manejan los recursos de hardware, aun cuando este esté en un entorno Cloud. Y en un mundo donde las aplicaciones son cada vez más complejas, como por ejemplo aplicaciones distribuidas basadas en microservicios, lleva a que se tenga como consecuencia una arquitectura, y por ende un software, difícil de mantener, escalar y monitorear.

En este trabajo, tomando como base los Twelve-Factor App y las prácticas DevOps, se busca definir y aplicar una metodología, que comprenda desde el entorno de desarrollo de los programadores hasta la gestión del software en producción, identificando y explicando cada uno de los pasos intermedios. Que además cuente con Continuous Integration y Continuous Deployment (CI/CD) y que como paso final permita realizar el deployment de la aplicación en un cluster de Kubernetes.

Toda esta metodología se hará mediante el uso de tecnologías modernas y de código abierto, siguiendo las prácticas, métodos y documentación de los más grandes de la industria como Microsoft, Google, Netflix, etc.



UNIVERSIDAD DE MENDOZA FACULTAD DE INGENIERÍA

ÍNDICE

RESUMEN	1
ÍNDICE	2
CAPÍTULO 1: INTRODUCCIÓN	4
1.1. Estructura de la Tesis	4
1.2. Motivación	4
1.3. El Problema	5
1.4. Objetivos	7
1.5. Limitaciones y Alcances	7
CAPÍTULO 2: MARCO TEÓRICO	8
2.1. Arquitectura de Microservicios	8
2.2. Patrón BBF: Backend for Frontend	12
2.3. Beyond the Twelve-Factor App	14
CAPÍTULO 3: DESARROLLO DE LA METODOLOGÍA	20
3.1. Introducción	20
3.2. Proceso de trabajo	22
3.3. Diseño de la Arquitectura	25
3.4. Entorno de Desarrollo	30
3.5. Proceso de Continuous Integration/Continuous Delivery (CI/CD)	34
3.6. Proceso Iterativo e Incremental	38
CAPÍTULO 4: APLICACIÓN DE LA METODOLOGÍA	41
4.1. Introducción	41
4.2. Configurado del servidor de automatización (Jenkins)	41
4.3. Sprint 1: Desarrollo servidor OAuth2	44
4.4. Sprint 2: Creado de Fincas Microservice	62
4.5. Sprint 3: Creado del “WEB Gateway”	67
4.6. Sprint 4: Desarrollo de funcionalidad “Fincas”	72
4.7. Sprint n: Futuras iteraciones	87
CAPÍTULO 5: CONCLUSIONES	88
5.1. ¿Se cumplió con “Beyond The Twelve-Factor App”?	88



UNIVERSIDAD DE MENDOZA

FACULTAD DE INGENIERÍA

5.2. ¿Se cumplieron los objetivos?	92
5.3. Conclusiones finales	93
BIBLIOGRAFÍA	95



CAPÍTULO 1: INTRODUCCIÓN

1.1. Estructura de la Tesis

El presente trabajo de investigación está conformado por una parte teórica, en donde se busca identificar y explicar todos los elementos que componen un entorno de desarrollo de software que utilice tecnologías modernas y este enfocado al desarrollo de aplicaciones con arquitecturas de microservicios. A la vez tiene una parte práctica que se desarrolla en conjunto con la teoría, para poder entender cómo se aplican estos conceptos a un proyecto real.

Los elementos del desarrollo de software que se buscan explicar en este trabajo son:

1. Toma de requerimientos
2. Modo de trabajo
 - a. Metodologías ágiles
 - b. Buenas prácticas
3. Arquitectura del software
4. Entorno de desarrollo del software
 - a. Local (Desarrollo)
 - b. Cloud (Operaciones)
5. Proceso iterativo incremental

1.2. Motivación

La necesidad de desarrollar este trabajo esta impulsada por dos elementos principales. Uno de índole personal, donde se busca poder aplicar todo el aprendizaje realizado durante la carrera y además integrando nuevos conocimientos tales como Twelve-factor Apps, arquitecturas de microservicios, kubernetes, etc. Con el fin de poder comprender y trabajar en todo el stack de aplicaciones Cloud Native.

Por otro lado, el segundo elemento que motivo la elaboración del presente trabajo, es poder realizar un aporte a la informática y elaborar una metodología detalla que permita aplicar Ingeniería de Software para el desarrollo de aplicaciones en la nube. También se busca demostrar su aplicación con un ejemplo concreto, con el fin de que sea replicable a otros proyectos de similares características.



1.3. El Problema

Al querer desarrollar aplicaciones del tipo Cloud Native, existe un gran número de tecnologías y procedimientos para aplicar en cada una de las fases. Sin embargo, resulta de gran complejidad poder encontrar una metodología detalla que abarque todo el proceso, empezando desde 0 hasta tener un software deployado en un entorno cloud.

Esto lleva que al querer empezar a desarrollar una aplicación surjan un gran numero de dudas, desde preguntas simples como: ¿Que lenguaje utilizamos? o ¿Como manejamos el trabajo en equipo? a preguntas más complejas como podrían ser: ¿Cómo hacemos los deployments? ¿Cómo manejamos la integración continua? entre otras.

Este trabajo busca brindar *una* respuesta a esas y otras preguntas.

Por otro lado, se pueden identificar dos problemas comunes a la hora de desarrollar aplicaciones:

a. Falta de aplicación de buenas prácticas de ingeniería de software:

Si bien es cierto que, al querer trabajar teniendo en consideración prácticas como las propuestas por los twelve-factor apps y devops, se agrega mayor complejidad a todos los procesos y aumentan la cantidad de tiempo que se debe tener en cuenta para cumplir con estos requisitos (elaboración de diagramas, creación y mantenimientos de pipelines, versionado del código, test automáticos, etc.) los beneficios y las posibilidades de escalado/crecimiento que aportan son mucho mayor que las desventajas. Además, permiten desarrollar software de calidad.

Al no aplicar estas prácticas es mucho mayor el tiempo que se pierde reestructurando la arquitectura, el código, arreglando bugs, etc. al que hubiese sido necesario para aplicar buenas prácticas. Y no menor, la posibilidad de pérdida de clientes y confianza al desarrollar un software que no es de calidad.

b. Desarrollo de aplicaciones monolíticas:

Este problema depende de las necesidades del software. Muchas veces si los requerimientos del software a desarrollar no son muy complejos o grandes, una aplicación monolítica no representa un problema.



UNIVERSIDAD DE MENDOZA FACULTAD DE INGENIERÍA

De igual manera también, en algunos casos, conviene comenzar con aplicaciones monolíticas y migrar a microservicios según cambien las necesidades y complejidad del producto. Sin embargo, la implementación podría haber sido mejor y en menor tiempo si desde un principio se hubiese tenido en cuenta la posibilidad de incluir microservicios, por eso es muy importante un buen análisis previo al desarrollo de la aplicación para determinar qué tipo de arquitectura se adapta mejor.

El principal problema se encuentra en aplicaciones monolíticas que han crecido mucho y tienen cierto nivel de complejidad, como, por ejemplo, múltiples frontend, conexiones con diferentes sistemas y/o bases de datos, etc. En estos casos al tener un único paquete grande produce ciertas desventajas relacionadas con el primer problema:

1. Mayor tiempo y complejidad en los deployments de la aplicación. Lo que lleva a una mala integración continua y que la brecha que existe entre lo que está en producción y lo que está en un entorno de test sea mayor.
2. Dificultad para trabajar en paralelo. Al estar todo relacionado en un único paquete es más probable que se produzcan conflictos en el código o incluso que no sea posible que más de una persona trabaje en determinada funcionalidad del software por su relación con todo lo demás.
3. Si algo falla o queda fuera de servicio (ej: conexión a la base de datos) toda la aplicación quedará inaccesible debido a esa falla.



1.4. Objetivos

Objetivo General

- Definir y aplicar una metodología para el desarrollo de software que aplique prácticas de devops y “beyond the twelve-factor apps” con el fin de realizar CI/CD en kubernetes.

Objetivos Específicos

- Definir cada uno de los elementos que componen la metodología.
- Encontrar un proceso iterativo e incremental que permita trabajar en esta metodología.
- Aplicar la metodología y el proceso iterativo e incremental a un ejemplo concreto.

1.5. Limitaciones y Alcances

El foco de este trabajo está puesto en todo el proceso que define el desarrollo del software y no tanto en el software en sí. Dándole mayor importancia al modo de trabajo, entornos, la metodología y como se debe realizar la correcta implementación de esta, etc. No se busca desarrollar un software completo, ni explicar paso por paso como codificarlo. Si bien se explicarán las principales secciones del código utilizadas a modo de ejemplo, la mayor parte de este será generado automáticamente con la herramienta Jhipster.

El ejemplo se desarrolla utilizando java y angular, con todas herramientas de código libre (a excepción del clúster de kubernetes) con el fin de que sea replicable en cualquier entorno y lenguaje.



CAPÍTULO 2: MARCO TEÓRICO

2.1. Arquitectura de Microservicios

Introducción

La arquitectura basada en microservicios (y los microservicios en sí) nacen de la necesidad de modelar sistemas complejos y dinámicos. Requerimientos tales como continuous-delivery, automatización de la infraestructura, equipos autónomos y reducidos y sistemas escalables entre otros son algunos de los puntos que han llevado al surgimiento de este enfoque.

Si bien es algo que ya existe desde hace varios años¹, la popularidad de los microservicios se ha visto incrementada en la actualidad por las mejoras tecnológicas en virtualización y operación en entornos cloud que han permitido realizar una buena implementación de este enfoque.

¿Qué son los microservicios?

Sam Newman, autor de “Building Microservices” define a los microservicios como: “servicios *autónomos* y *pequeños* que trabajan en conjunto.”

Donde por pequeño se entiende que estos servicios deben resolver una función específica que está definida y delimitada por el modelo de dominio. Al diseñar una aplicación compleja el modelo de dominio general de toda la aplicación puede ser fragmentado en modelos más pequeños agrupando las funcionalidades y entidades por relación y dependencia que existen entre ellas, luego estos modelos se pueden aplicar en los diferentes microservicios. Mientras más pequeños sean, más se aprovechan las ventajas de esta arquitectura.

Por otro lado, la definición menciona que los servicios deben ser autónomos, esto hace referencia a que deben funcionar como entidades totalmente independientes. Deben poder ser deployados como si fueran

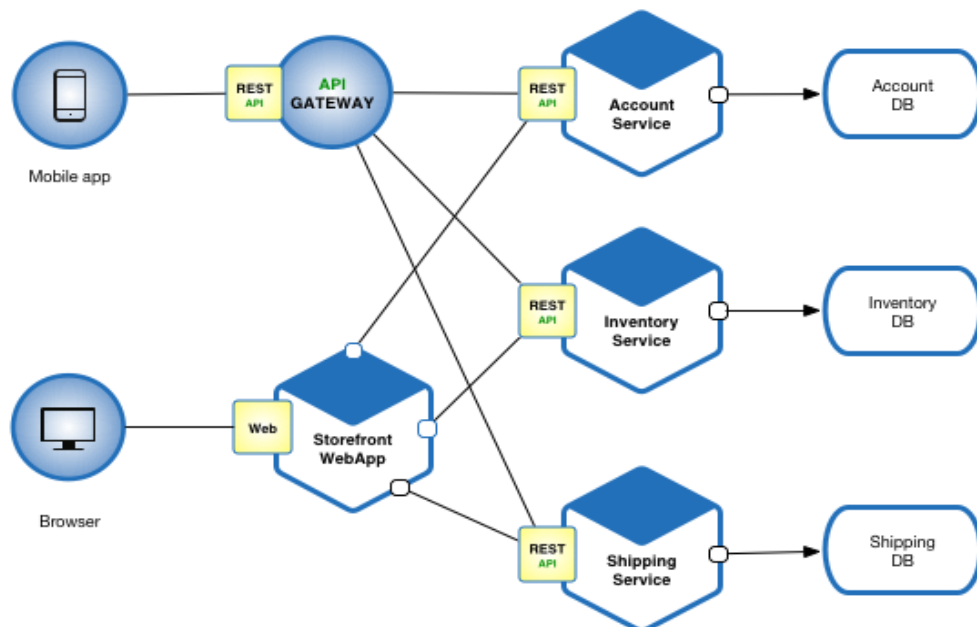
¹ En 2005 Peter Rodgers introdujo el término “Micro-Web-Services” durante una conferencia de cloud computing. En 2011 se utilizó el término “microservice” en un evento de arquitectura de software, haciendo alusión a la arquitectura que ha ido evolucionando hasta lo que hoy se conoce como tal. - <https://en.wikipedia.org/wiki/Microservices#History>

una aplicación independiente y la forma de comunicarse y acceder a estos es mediante las API definidas para esta “aplicación”.

La documentación de Microsoft para microservicios define ciertos criterios que estos deben cumplir para su correcta implementación:²

- Cada servicio tiene una única responsabilidad.
- No hay llamadas que generen mucha conversación entre servicios.
- Cada servicio es lo suficientemente pequeño como para que un equipo pequeño lo pueda generar trabajando de forma independiente.
- No hay interdependencias que requieran la implementación de dos o más servicios en sincronía.
- Los servicios no están estrechamente acoplados y pueden evolucionar independientemente.
- Los límites de servicio no causarán problemas con la coherencia de datos o la integridad. A veces es importante mantener la coherencia de datos colocando la funcionalidad en un único microservicio.

Ejemplo de una arquitectura de microservicios:



3

² <https://docs.microsoft.com/en-us/azure/architecture/microservices/model/microservice-boundaries>

³ https://microservices.io/i/Microservice_Architecture.png

Aplicación Monolítica vs. Microservicios⁴

Característica	Monolítico	Microservicio
Independencia de tecnologías	Se deben seleccionar tecnologías que sean capaces de cumplir con todas las necesidades y adaptar el software para que funcione con estas tecnologías.	Cada microservicio puede utilizar las tecnologías que mejor se adapten a las necesidades de este. Permite probar nuevas tecnologías en servicios de poco impacto.
Recuperación de fallos	Si algo falla, toda la aplicación falla. Es muy demandante, desde el punto de vista de los recursos, tener réplicas de la aplicación en caso de fallas.	Si algo falla, el problema se puede aislar fácilmente mientras el resto de la aplicación sigue funcionando. Se pueden tener réplicas de los microservicios más críticos. Hay que considerar las fallas debidas al networking entre servicios.
Escalabilidad	En caso de necesidad, se debe escalar toda la aplicación en conjunto, haciendo que los requerimientos de hardware sean mayores. O tener cuellos de botellas al tener alguna limitación en el hardware.	Se pueden tener múltiples instancias de los microservicios con mayor demanda, incluso con tecnologías como kubernetes es posible automatizar esto.
Facilidad de deployment	Para lanzar un nuevo cambio es necesario deployar toda la aplicación, lo que dificulta la integración continua.	Se pueden realizar cambios en un microservicio y deployarlos independientemente del resto de la aplicación. Lo que lleva a deployments más rápidos, livianos y seguidos permitiendo, además, que se mas fácil hacer un rollback en caso de que algo salga mal.

⁴ Cuadro elaborado en base a: Sam Newman - "Building Microservices" (2015) [p.19 - p.27]



UNIVERSIDAD DE MENDOZA FACULTAD DE INGENIERÍA

Estructura organizacional	En aplicaciones complejas es necesario asignar equipos grandes para que trabajen todos en un mismo código, lo que causa menor productividad al dificultar la sincronización.	Se pueden asignar equipos pequeños para que trabajen sobre un microservicio permitiendo una mayor especialización y productividad.
Reusabilidad	Ante la posibilidad de cambios en los requerimientos es muy complejo adaptar el software y que no afecte a otras partes de este.	Resulta más simple hacer microservicios reutilizables y, de ser necesario, adaptarlos a nuevas necesidades.
Optimización para reemplazos	Es muy complejo modificar o reemplazar partes críticas del código por el impacto que esto pueda llegar a tener en el resto de la aplicación.	Los microservicios pueden ser fácilmente reemplazados, borrados o modificados.

Si bien se observa que los microservicios tienen múltiples ventajas frente a una arquitectura monolítica, traen consigo una complejidad extra, ya sea por el networking entre estos, administración de los repositorios y pipelines, etc. Por lo que resulta indispensable realizar un análisis de las necesidades del software y determinar qué tipo de arquitectura se adapta mejor.



2.2. Patrón BBF: Backend for Frontend

Introducción

El patrón BBF es una manera de implementar una arquitectura de microservicios, el cual será utilizada para el desarrollo del presente trabajo.

Este patrón fue desarrollado y publicado por un exingeniero de SoundCloud, Phil Calçado, en 2015. Surge como una evolución de la arquitectura de su aplicación. En primera instancia migraron de una arquitectura monolítica a una de microservicios para, principalmente, disminuir los tiempos necesarios para publicar nuevos cambios al mercado.

Con el tiempo, principalmente impulsado por el incremento de la demanda de las aplicaciones Mobile (iOS y Android), surge el problema de que sus API endpoints eran muy genéricas, lo que hacía que fueran necesarias muchas llamadas al backend, según la plataforma, para obtener los datos que necesitaba la interfaz. Otro problema importante era que no podían ofrecer al usuario el tipo de experiencia que ellos querían, ya que sus API tenían que ser lo suficientemente genéricas para ser utilizadas por aplicaciones de terceros.

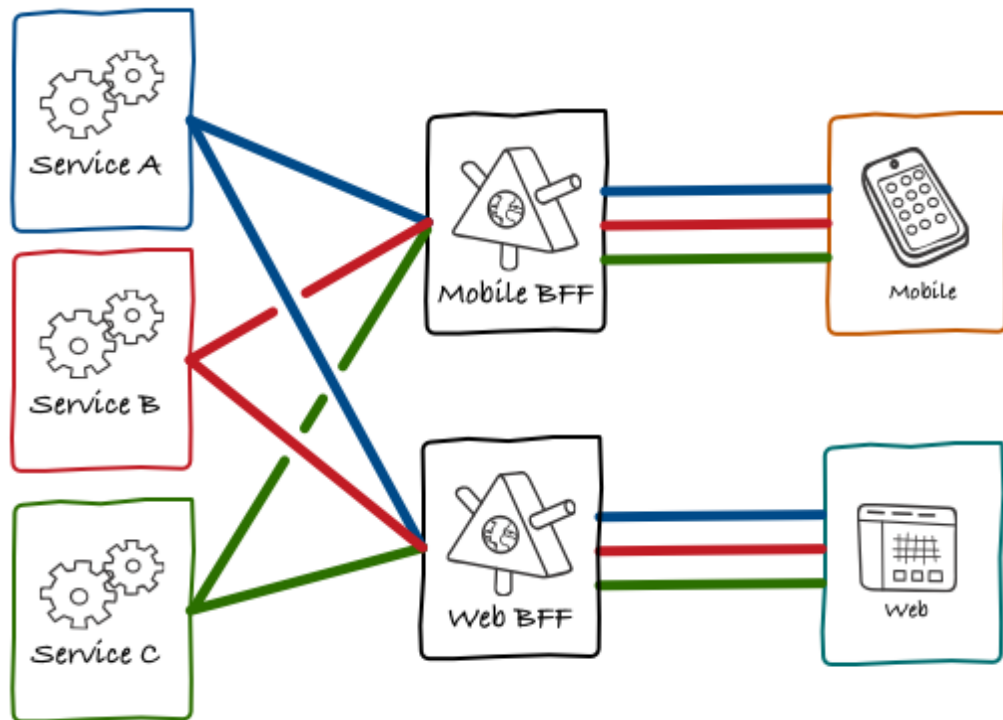
Con la implementación del patrón BFF lograron resolver estos problemas y, además, disminuir el tráfico generado en la comunicación entre backend y frontend. Un ejemplo es que pasaron de 159 llamadas al backend a 3 para generar la página del perfil.⁵

Definición

Básicamente lo que permite este patrón es que en lugar de tener un único punto central que sirva de conexión a todos los frontend y de ahí, mediante un proxy, acceder a los diferentes servicios, se pueda, como indica su nombre, tener un backend específico para cada frontend (desktop, Mobile, etc.). Esto permite que cada backend esté optimizado para cumplir con las necesidades de las interfaces, lo que genera, como resultado, ventajas muy similares a las planteadas para microservicios

⁵ https://philcalcado.com/2015/09/18/the_back_end_for_front_end_pattern_bff.html

(reducción del tráfico, facilidad para trabajar en equipos reducidos, menor complejidad en el código, etc.).^{6 7}



8

⁶ <https://docs.microsoft.com/en-us/azure/architecture/patterns/backends-for-frontends>

⁷ <https://samnewman.io/patterns/architectural/bff/>

⁸ <https://philcalcado.com/img/2015-09-back-end-for-front-end-pattern/sc-bff-1.png>



2.3. Beyond the Twelve-Factor App

Introducción

Los “Twelve-Factor App” es una metodología redactada por desarrolladores de *Heroku*⁹. Surge con la motivación de brindar soluciones a problemas comunes en el desarrollo de aplicaciones modernas del tipo SaaS (Software as a Service), utilizado para las Web Apps. El principal objetivo de esta metodología es poder construir aplicaciones que funcionen en entornos cloud.¹⁰

La idea es que se puedan aplicar a cualquier tipo de aplicación, independientemente del lenguaje y de los backing services. Sin embargo, el autor del libro “*Beyond the 12-Factor App*”, Kevin Hoffman, cuestiona que son muy específicos para la arquitectura que propone *Heroku*, por lo que él, tomando como base los originales, redacta su propia definición de estos, además de agregar tres “factor app” más:¹¹ (En cursiva los que son propios de Kevin Hoffman)

1. Un código base.
2. *API first*.
3. Manejo de dependencias.
4. *Diseñar*, construir, desplegar, ejecutar.
5. Configuraciones, *credenciales y código*.
6. Logs.
7. Desechabilidad.
8. Backing services.
9. Paridad de entornos.
10. Procesos administrativos.
11. Asignación de puertos.
12. Procesos sin estado.
13. Concurrencia.
14. *Telemetría*.
15. *Autenticación y autorización*.

Para la redacción de este capítulo y el desarrollo del trabajo se tomará la definición propuesta por Kevin Hoffman.

⁹ Plataforma como servicio (PaaS) que permite a los desarrolladores crear, ejecutar y operar aplicaciones completamente en la nube. - Traducción de la página oficial de *Heroku*.

¹⁰ Documentación oficial de “*The Twelve-Factor App*” (<https://www.12factor.net/>)

¹¹ Kevin Hoffman - “*Beyond the 12-Factor App*” - prefacio



UNIVERSIDAD DE MENDOZA FACULTAD DE INGENIERÍA

I. Un código base

Un código base es un repositorio, o conjunto de repositorios que comparten el mismo origen (root), donde se almacena el código fuente.

Las aplicaciones *cloud-native*¹² siempre deben estar compuestas por un único código base, el cual debe ser trackeado con algún controlador de versiones. Y, además, un código base debe servir a una única aplicación.

II. API First

Se debe considerar a las API como elementos de primera clase del proceso de desarrollo de software. Es decir, que desde que se comienza a desarrollar la aplicación, se debe hacer desde el punto de vista de que lo que se está construyendo son API destinadas a ser consumidas por aplicaciones y servicios.

Este enfoque ayuda a tener una clara idea de cómo será el funcionamiento de la API, e incluso testearla, antes de desarrollarla. Esto permite una mejor comunicación con los usuarios de estas y adaptarlas, en caso de ser necesario, sin mucho esfuerzo. Por otro lado, también facilita el trabajo en paralelo, ya que, los desarrolladores pueden saber cómo será el funcionamiento de la API aun cuando está todavía no ha sido codificada o implementada completamente.

Existen diferentes estándares y herramientas para la aplicación de este enfoque.

III. Manejo de dependencias

Las aplicaciones no pueden depender de un servidor que les suministra todo lo que necesitan, sino que, deben incluir todas sus dependencias dentro de un mismo *build artifact*. Ejemplo de administradores de dependencias podrían ser “Maven” o “Gradle” para Java, “NuGet” para .NET o “Node Package Manager” para angular. Estas herramientas permiten declarar dependencias y que la herramienta se encargue de cumplirlas.

¹² Enfoque de desarrollo de software que utiliza cloud computing para construir y ejecutar aplicaciones escalables en entornos cloud. -

<https://github.com/cncf/toc/blob/master/DEFINITION.md>



UNIVERSIDAD DE MENDOZA FACULTAD DE INGENIERÍA

IV. Diseñar, construir, desplegar, ejecutar

Los twelve-factor app solo hablan de “construir, desplegar y ejecutar”, Hoffman agrega un paso más que es el diseño.

Se lo define como la lista de pasos de un proceso iterativo para llevar el código de un *único código base* a ejecución, generalmente llevado a cabo dentro de los pipelines de CI/CD.

1. Diseño: Se deben realizar pequeños diseños para las funcionalidades que se vayan a implementar, teniendo en cuenta que estos pueden ir cambiando al ser un proceso iterativo. El principal objetivo es que el desarrollador pueda saber qué dependencias serán necesarias y cómo se relacionarán con el resto de la aplicación
2. Construir (Build): Es el paso en donde el código base se compila en un binario (JAR, EXE, ZIP, etc.). Todas las dependencias definidas en el diseño son empaquetadas dentro de un build artifact.
3. Desplegar (Deployment): En las aplicaciones cloud-native los deployments son llevados a cabo subiendo (pushing) la salida del Build (Build Artifact) al entorno cloud en el que se esté trabajando. Estos deployments deben ser únicos y deben tener un ID único que permita identificarlos.
4. Ejecutar: En este paso es en donde finalmente se ejecuta la aplicación en el entorno cloud y debe ser llevado a cabo por el proveedor de cloud, por lo cual, varía según cual sea este. Se debe tener en consideración que los desarrolladores también deben ser capaces de ejecutar la aplicación de forma local y que sea lo más parecido posible a como sería en la cloud.

V. Configuraciones, credenciales y código

Los 12 factor originales solo hablan de “configuraciones”, las cuales deben ser almacenadas dentro del entorno. Hoffman propone dos más “credenciales y código” y define que estos tres elementos deben estar claramente separados unos de los otros.

1. Configuraciones: URLs, strings de conexión a las bases de datos, credenciales de APIs externas, etc. Las configuraciones *no* contienen información interna que sea parte de la aplicación, es decir que estas varían según los deployments (test, staging, producción, etc.).



UNIVERSIDAD DE MENDOZA FACULTAD DE INGENIERÍA

2. Credenciales: Información sensible que nunca debe ser conocida por terceros. (ej.: credenciales de acceso a un repositorio de GIT)

Tanto las configuraciones como las credenciales nunca deben estar en el código base y una buena forma de lograr esto es inyectando esta información como variables de entorno durante los deployments.

VI. Logs

La aplicación *no* debe almacenar, procesar y analizar los logs si no que solo debe encargarse de emitir eventos por las salidas estándar (stdout, stderr) con la información que sea relevante. Luego herramientas externas o el mismo cloud provider se encargarán de escuchar estos eventos y hacer con esta información lo que sea necesario.

VII. Desechabilidad (Disposability)

Los procesos de la aplicación deben ser capaces de detenerse y ejecutarse rápidamente, esto permite disminuir los tiempos de downtime y la posibilidad de datos corruptos. Por ejemplo, si hay mucha carga a la aplicación y es necesario generar más instancias, estas deben poder arrancar rápidamente y luego detenidas cuando ya no sean necesarias.

VIII. Backing Service

Los *Backing Service* son cualquier servicio que la aplicación necesita para su funcionamiento (ej.: OAuth2 service, SMTP service, Cache service, etc). En un entorno cloud, el almacenamiento de archivos en disco, también debe ser considerado como un backing service.

La forma en que la aplicación debe tratar a los backing service es declarando su necesidad para este, y la configuración que permite esta vinculación debe hacerse mediante configuración externa (V Factor App). Además, la aplicación debe ser capaz de acoplar y desacoplar estos servicios en tiempo de ejecución, sin necesidad de hacer un nuevo deployment.

IX. Paridad de entornos

Los entornos de desarrollo (Test, Staging, Pre-Producción, etc.) deben ser lo más parecidos al entorno de producción en tiempo, personas y recursos.



UNIVERSIDAD DE MENDOZA FACULTAD DE INGENIERÍA

X. Procesos administrativos

Los Twelve-Factor originales sugieren el uso de procesos administrativos, por ejemplo, para migraciones de bases de datos o ejecución de scripts de un solo uso. Hoffman, sin embargo, desalienta el uso de estos procesos, ya que, en un entorno cloud, esto puede traer diferentes problemas. Por ejemplo, al tener múltiples instancias de la aplicación corriendo en simultáneo, si se ejecutase un script que modifique la base de datos (after migration) podría corromperla o duplicar datos.

Para solucionar esto Hoffman propone utilizar un microservicio destinado a esto, proporcionar endpoints REST para correr los procesos necesarios o utilizar procesos administrativos proporcionados por el mismo Cloud Provider.

XI. Asignación de puertos

Siempre debe existir una relación 1:1 entre las aplicaciones y los servidores de estas. Esto permite exponer el puerto de estas aplicaciones y luego, en una capa superior, dejar que el cloud provider se encargue de manejar el enrutamiento y el mapeo de los nombres de host a los puertos internos de la aplicación.

XII. Procesos sin estado (Stateless Processes)

Los procesos de la aplicación no deben tener estados que perduren más allá de una llamada y en caso de ser necesario tener estados que perduren en el tiempo estos deben ser almacenados fuera de la aplicación en backing services, por ejemplo, en un base de datos o un administrador de caché.

XIII. Concurrencia

La aplicación debe ser capaz de crecer horizontalmente, es decir que, en caso de necesidad, en lugar de aumentar CPU, RAM, o cualquier otro recurso y hacer un único gran proceso, se debe poder generar más instancias del proceso requerido y distribuir la carga entre todas estas instancias.

XIV. Telemetría

Al tener aplicaciones funcionando en un entorno cloud, resulta muy complejo comprender con exactitud cómo se comportan y en qué estado están estas aplicaciones, por lo que resulta necesario contar con herramientas que permitan examinar estos elementos (ej.: Uptime,



UNIVERSIDAD DE MENDOZA FACULTAD DE INGENIERÍA

páginas más visitadas, cantidad de llamadas HTTP en determinado momento, etc.).

XV. Autenticación y Autorización

La seguridad debe ser uno de los pilares de la aplicación. Todos los endpoints de la aplicación deben contar con controles de acceso de forma de que siempre se pueda conocer quién es y qué permisos (roles) tiene el usuario de determinado endpoint, y en caso de ser necesario, denegar el acceso.



CAPÍTULO 3: DESARROLLO DE LA METODOLOGÍA

3.1. Introducción

En el presente capítulo se desarrolla la metodología que aplique todos los conceptos que se han redactado en los capítulos anteriores, es decir, una metodología que tome como base los factor app de “Beyond The Twelve-Factor App”, un arquitectura de microservicios y, mediante ci/cd, realice el deployment de nuestras aplicaciones en un entorno cloud, específicamente, en kubernetes.

Finalmente, como parte de la metodología, se define y aplica, mediante un ejemplo, un proceso iterativo e incremental que permita trabajar con esta.

Los principales componentes son:

1. Proceso de trabajo - Scrum y buenas prácticas
2. Diseño de la Arquitectura del software - Microservices (BFF), Docker, Kubernetes.
3. Entorno de desarrollo - Entorno Local y entorno Cloud
4. Proceso de CI/CD - Integración continua y deployments automáticos en kubernetes.
5. Proceso iterativo e incremental - Adaptable a un sprint

Si bien los conceptos teóricos ya han sido explicados en el marco teórico, en este capítulo se analiza porque es conveniente y cómo aplicar cada uno de estos. Para lograr esto se desarrollará, a lo largo del presente capítulo, una aplicación “FincasApp” que sirva a modo a modo de ejemplo y permite analizar cómo se implementa la metodología desarrollada.

Los requerimientos para *FincasApp* son:

Fincas App:

Se quiere desarrollar una aplicación WEB para productores del sector agrícola que les permita a los usuarios tener un registro de sus campos y productos, así como también la posibilidad de comercializar con estos entre los usuarios de la App. También se debe contar con un servicio de fletes que les permita a los usuarios realizar pedidos de envíos entre ellos, donde uno genera el pedido y otro lo toma para llevarlo a cabo. La aplicación, además, debe proporcionar información del clima para quien lo desee.



UNIVERSIDAD DE MENDOZA FACULTAD DE INGENIERÍA

Debe contar con una versión completa para navegadores desktop y una versión reducida para dispositivos Mobile con la que se pueda acceder a la funcionalidad de fletes.



3.2. Proceso de trabajo

Introducción

El primer paso de este proceso es definir un proceso de trabajo con el cual se puedan realizar las tareas que demande el desarrollo de la aplicación.

Debe ser capaz de proporcionar la organización necesaria para poder trabajar de forma eficiente en un equipo interdisciplinario, brindar vías de comunicación fluidas, rápida adaptabilidad a los cambios y brindar la ayuda necesaria para lograr con los plazos establecidos para los cumplimientos de las tareas, ya sean de desarrollo, investigación, administración o de cualquier otro tipo que esté relacionado con la app.

Una buena metodología que permite lograr todo eso es SCRUM, que será la que se aplicará para el desarrollo de FincasApp.

SCRUM define las pautas generales de todo el proceso, sin embargo, también es recomendable contar con ciertas reglas o pautas más específicas para la parte de desarrollo. Estas pautas comúnmente se las conocen como “Buenas prácticas” y si bien hay muchas y van en el desarrollar aplicarlas o no, se pueden definir algunas que se deban aplicar de forma obligatoria al desarrollo, ya sea por control manual o forzando su aplicación mediante software, como podría ser el uso de un pipeline que controla el “test coverage”.

La aplicación de estas reglas de buenas prácticas permite, entre muchas ventajas, tener un código mucho más limpio, consistente, entendible, flexible a cambios y como principal ventaja: que el código no dependa de una persona del equipo, sino que cualquiera pueda trabajar en él.

SCRUM

Partiendo de lo que sugiere la metodología de SCRUM se toman los requerimientos del proyecto (FincasApp), obtenidos de un supuesto cliente, y se elabora el backlog del producto. Luego se tomarán elementos de este backlog que serán desglosadas en tareas que se resolverán a lo largo de diferentes sprints.

Todo el proceso actual de definir modo de trabajo, arquitectura, iteraciones, etc. puede ser considerado un *Sprint 0*, el cual no aporta valor de negocio, pero sí facilita la aplicación de futuros sprints.



UNIVERSIDAD DE MENDOZA FACULTAD DE INGENIERÍA

Backlog del producto

Una de las primeras tareas que propone SCRUM es la elaboración del backlog del producto, partiendo de los [requerimientos del sistema](#), de donde luego se desprenden las tareas que serán realizadas en cada sprint.

Un backlog del producto inicial¹³ para *FincasApp* podría ser:

	Order	ID	Title	State	Effort
+	1	34	Definir una infraestructura	● To Do	5
	2	35	Desarrollar Servicio OAuth	● To Do	2
	3	36	Desarrollar modulo de clima	● To Do	2
	4	37	Desarrollar modulo de delivery	● To Do	3
	5	38	Desarrollar modulo de seguimiento fincas	● To Do	4
	6	39	Desarrollar modulo de compra-venta	● To Do	4
	7	40	Desarrollar gateway para administradores	● To Do	2
	8	43	Diseñar Interfaz de usuario	● To Do	4
	9	41	Desarrollar gateway WEB	● To Do	3
	10	42	Desarrollar gateway Mobile	● To Do	3

Los criterios en base a los cuales se pueden ordenar estos elementos varían según la necesidad del proyecto, en este caso se utilizó el estado y el esfuerzo.

Para el esfuerzo se definió una unidad numérica abstracta que va de 1 a 5 que permita realizar una estimación inicial del esfuerzo¹⁴ requerido para cumplir con ese backlog ítem. Esta unidad resulta útil a la hora de definir que backlogs ítems se tomarán para determinando sprint, ya que, se puede definir una cantidad de esfuerzo máxima y asignar los ítems que “entren” en ese sprint.

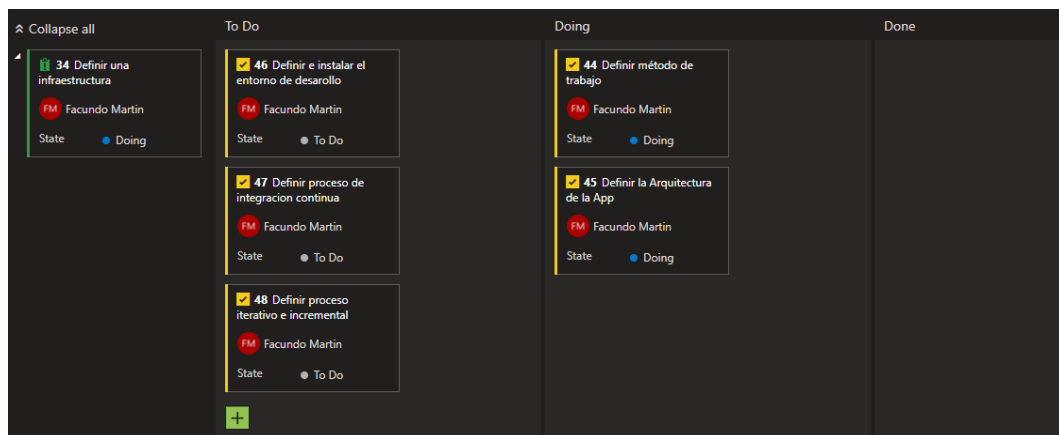
Sprint

Como ya se mencionó, para cada Sprint, conforme a la capacidad del equipo, se tomarán ítems del backlog del producto en base a su unidad de “esfuerzo”. Luego se definirán las tareas que se realizarán para ese ítem.

¹³ Se debe tener en cuenta que, en una situación real, conforme vayan pasando los sprints y vayan surgiendo nuevos requerimientos de las reuniones de fin de sprint, el backlog irá sufriendo modificaciones.

¹⁴ Existen diversas formas de realizar esta estimación. Una buena técnica para esto es “Planning poker” - https://en.wikipedia.org/wiki/Planning_poker

En las últimas secciones de este capítulo se define un proceso iterativo e incremental que será aplicado dentro de cada sprint y agregue valor al software. Pero de momento, como también se mencionó previamente, se trabaja en un sprint 0 el cual se basa en el backlog ítem “34: Definir una infraestructura” el cual tiene en cuenta estas primeras etapas (elección del método de trabajo, buenas prácticas, definición de arquitectura, etc.). Las tareas para este sprint serán:



Tanto para el backlog como para el seguimiento de las tareas se utilizó la herramienta de Azure Boards¹⁵ que permite hacer un buen seguimiento del proyecto y que es fácil de implementar para SCRUM, pero existen múltiples herramientas que cumplen el mismo fin como JIRA, Trello o incluso una planilla de Excel/Hoja de cálculo de Google.

Buena práctica: Tests-Driven Development

Los test son una parte fundamental de la aplicación y pueden ser llevados a cabo por personas (ejecución de test-cases o tests exploratorios) o por software (unit tests, integration tests, end to end tests). Si bien ambos son necesarios, los segundos permiten testear mayor parte de la aplicación en forma automática y en menor tiempo. Además, pueden ser ejecutados cuando se está desarrollando la aplicación, lo que les da a los desarrolladores una mayor confianza en el código.

Para las fases de desarrollo de FincasApp se utilizará la técnica de Test-Driven Development, lo que establece como regla que primero se escriben los test para determinada funcionalidad y luego el código que pase esos tests.

¹⁵ <https://azure.microsoft.com/es-es/services/devops/boards/>

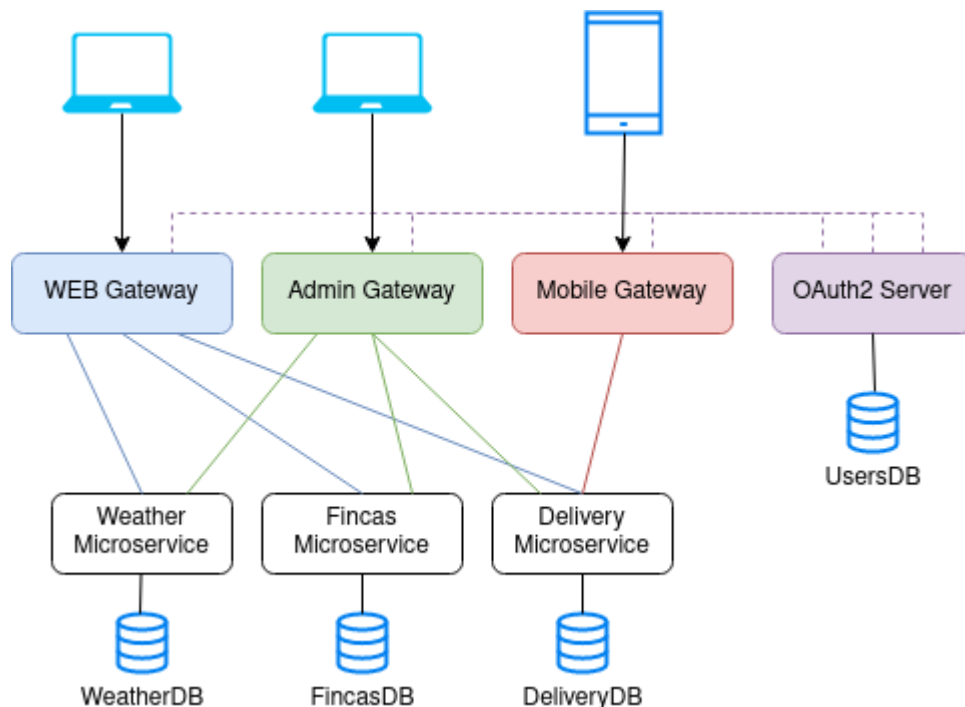
3.3. Diseño de la Arquitectura

Microservicios

La metodología planteada a lo largo de este trabajo está basada en una arquitectura de microservicios, también podría ser aplicable a una aplicación monolítica, sin embargo, todo el potencial y beneficios de esta forma de trabajo se ve reflejada al trabajar con este tipo de arquitectura.

Por otro lado, al trabajar con microservicios y las herramientas relacionadas para su aplicación eficiente (docker, kubernetes), le agrega una complejidad extra a todo el proyecto por lo que se debe analizar si esta complejidad agregada es menor que los beneficios que aporta esta arquitectura.

Analizando los requerimientos de *FincasApp* se pueden distinguir claramente la necesidad de tres microservicios, uno para la administración de las fincas, otro para el módulo de delivery y uno más para el clima. El diagrama inicial de microservicios, aplicando el patrón de [backend for frontends](#), es el siguiente:



Además de los microservicios mencionados previamente, hay uno más "OAuth2 Server" que, como su nombre indica, será un servidor propio que

mediante el protocolo de OAuth2 permita la autenticación a la aplicación, además de manejar una base de datos con los usuarios y roles de estos.

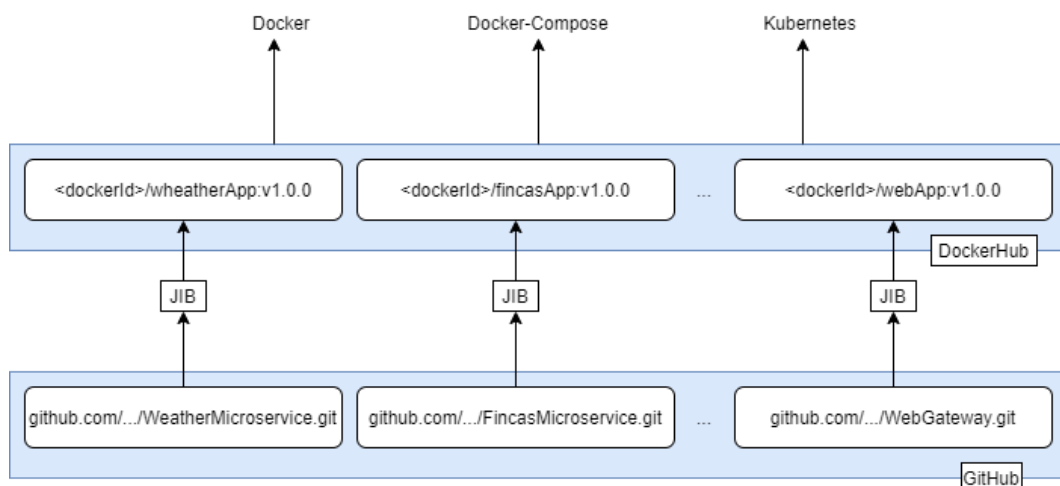
Por otro lado, se observan tres gateways, WEB, Admin y Mobile, que en el fondo también funcionan como microservicios, y es en donde están las interfaces de usuarios y se hacen las llamadas a los microservicios según sea necesario.

Docker

Para contener a los diferentes elementos de la aplicación se utiliza Docker, donde cada microservicio, gateway, base de datos u otros *backing service* necesarios están contenidos dentro de un docker image. Esto le permite a la aplicación poder ejecutarse en cualquier lugar que sea capaz de correr contenedores dockers. Además, permite aprovechar la ventaja de los microservicios permitiendo hacer un despliegue distribuido de la aplicación.

Por otro lado, con docker registry y un correcto taggeado de las imágenes se puede tener un historial de versiones de la aplicación y elegir deployar cualquiera de estas en los entornos correspondientes (test, staging, producción, etc.), así como también la posibilidad de hacer un rollback en caso de que algo no funcione.

La aplicación de dockers a *FincasApp* es de la siguiente manera:



Se toma el código base de los diferentes repositorios de GitHub, uno para cada microservicio y todos bajo la misma raíz, y se crea la imagen de



UNIVERSIDAD DE MENDOZA FACULTAD DE INGENIERÍA

docker utilizando JIB¹⁶. Lo cual resulta muy útil a la hora de hacer builds en algún servidor, que incluso podría estar corriendo dentro de un docker.

Posteriormente estas imágenes son taggeadas con el número de versión correspondiente, obtenidas del pom.xml, y subidas al docker registry, que en este caso se utiliza docker hub.

Por último, estas imágenes pueden ser ejecutadas mediante docker o docker-compose para un desarrollo/testeo local o con kubernetes para llevarlas a producción o a un entorno test en la nube.

Kubernetes

La mejor manera para hacer el deployment de la aplicación contenida en dockers es mediante kubernetes. Esto permite beneficios tales como: el escalado automático de la aplicación según el consumo, facilidad de despliegue, deployments con cero downtime y/o probar una nueva versión con un número reducido de clientes y de a poco ir mudando todo el tráfico a esta nueva versión si todo va bien o revertir el cambio si falla.

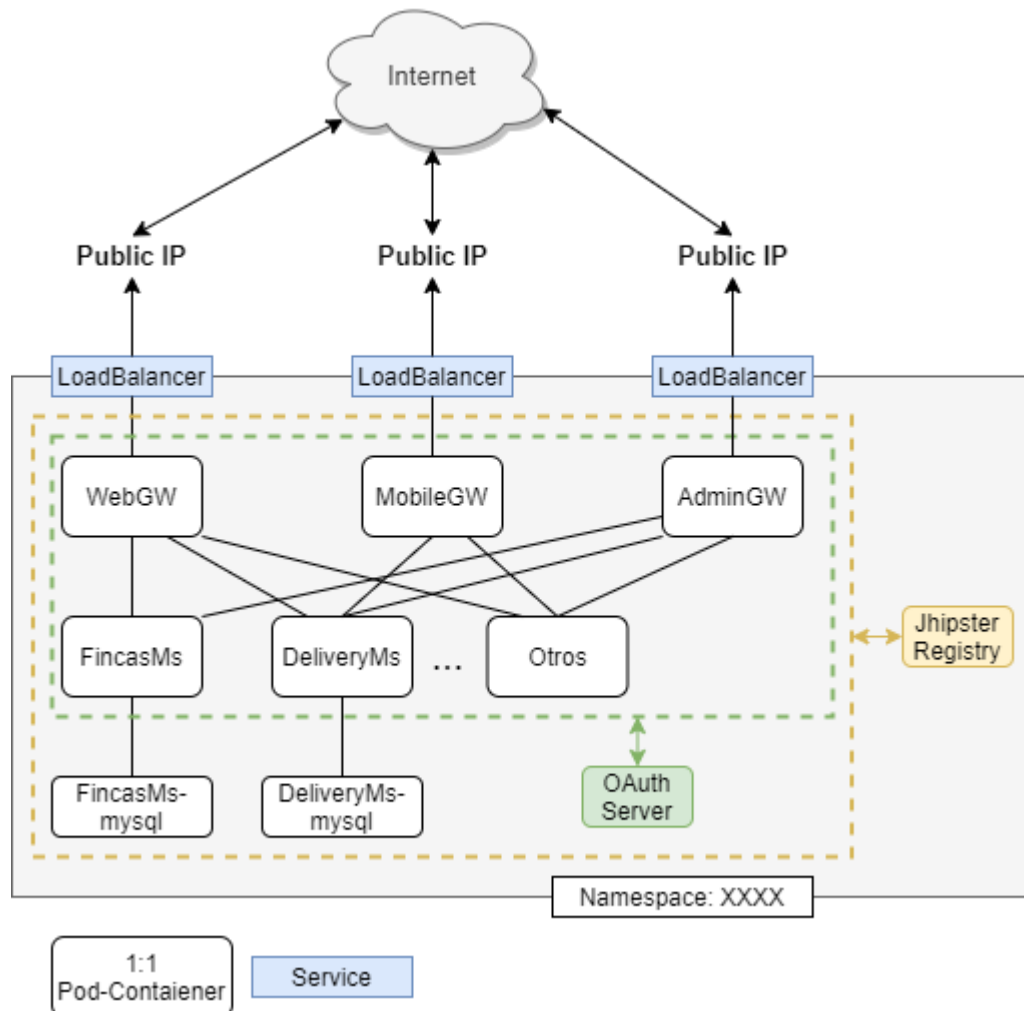
Para *FincasApp* se utilizan dos clusters de kubernetes en Digitalocean, uno para los entornos test y staging, que estarán divididos mediante separación lógica (namespaces) y otro únicamente para el entorno de producción.

Para el deployment de una nueva versión con cero downtime se creará un pod (o más según sea necesario) en el que se desplegará la nueva versión y luego se destruirán los antiguos.

Por último, para exponer las direcciones IP de nuestra aplicación a la red se hará mediante el uso de servicios del tipo Load Balancers.

¹⁶ Plugin de maven que permite el creado de imágenes de aplicaciones JAVA sin la necesidad de tener docker instalado.

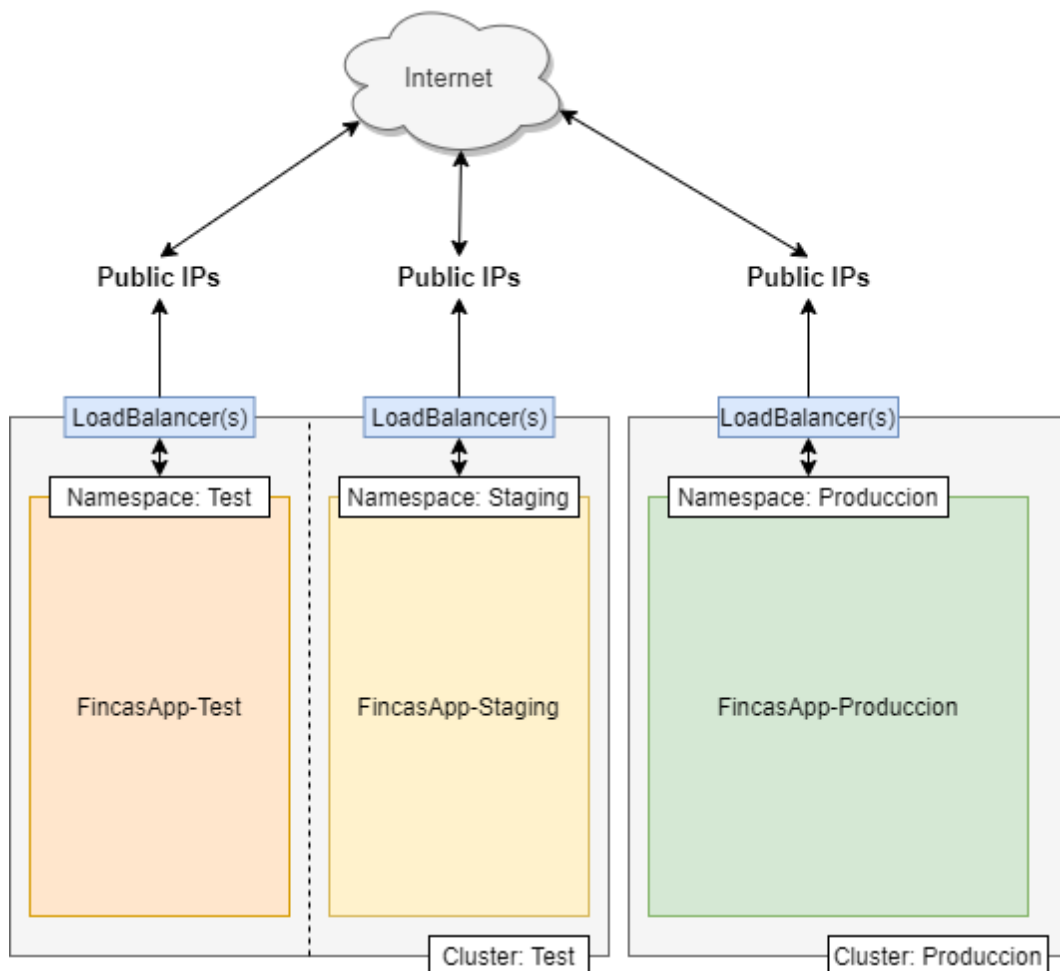
La vista de *uno* de estos namespaces dentro de los clusters sería:¹⁷



Además de los microservicios mencionados previamente, en el gráfico se puede apreciar la existencia de uno nuevo llamado “Jhipster Registry”, este microservicio se crea automáticamente mediante la herramienta de jhipster y es necesario para trabajar con arquitecturas de este tipo. Por un lado, permite que todos los microservicios se registren y puedan identificar a los otros que existen en la red. Y por otro, permite analizar el estado de estos.

¹⁷ Se supone una relación 1:1 entre pods y containers. Tampoco se muestran en el gráfico los servicios del tipo ClusterIP que son los utilizados para la comunicación interna dentro del cluster.

La vista de la infraestructura completa que corre dentro de kubernetes seria:



Donde dentro de cada Namespace se encuentra lo diagramado en la imagen anterior. También se debe considerar que por cada namespace existen múltiples LoadBalancer y por ende múltiples IP públicas, uno por cada gateway, como se mostraba en el gráfico anterior.

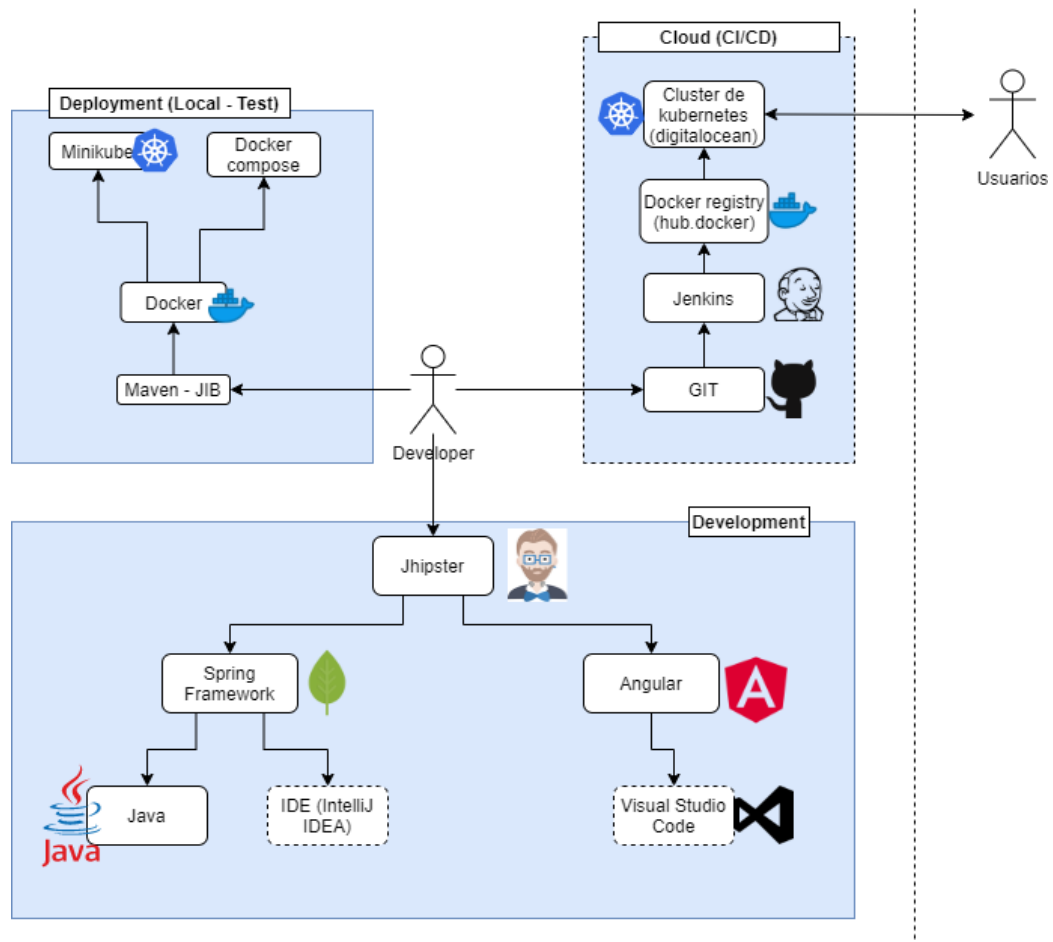
3.4. Entorno de Desarrollo

Introducción

Es importante contar con un entorno de desarrollo que permita trabajar con la arquitectura definida hasta ahora. Una vez armado y configurado este entorno es posible crear una imagen docker para asegurarse de que es el mismo en cada pc y además facilitar el proceso de integración de nuevos miembros al equipo, cambio de ordenadores o incluso hacer el desarrollo totalmente cloud.

Este entorno debe ser capaz de simular localmente lo mismo que existe en producción. Y además debe ser capaz de poder trabajar en todo el stack del proyecto, desde escribir código hasta la configuración del cluster de kubernetes.

En este caso se ha optado por armar un entorno de desarrollo con la siguiente estructura/herramientas:





UNIVERSIDAD DE MENDOZA FACULTAD DE INGENIERÍA

Como se puede observar en la imagen el entorno estará dividido en tres grandes grupos: Development (local), Deployment (local) y la parte cloud donde se realiza CI/CD.

Development (local):

Es donde el desarrollador realiza todos los cambios en relación al código del software, ya sea implementar alguna mejora, arreglo de bugs, optimización, etc. Para esto se optó por las siguientes herramientas:

1. Jhipster: Generador de aplicaciones gratuito y de código abierto que se utiliza para desarrollar rápidamente aplicaciones web modernas y microservicios utilizando Angular o React y Spring Framework.¹⁸ Para este trabajo se optó por Jhipster, ya que ofrece una buena solución tanto para la parte de desarrollo (creado de entidades, relaciones, microservicios, seguridad, etc) como así también para la parte de deployment (integración con git, build de imágenes, creación de archivos de configuración para kubernetes, etc). Como framework full stack otras alternativas podrían haber sido: ASP.net core para C#, Django para Python, etc.
2. Spring Framework: Es el framework Java en el cual se apoya Jhipster para todo el generado y desarrollo del backend.
3. Angular: Es el framework que utiliza Jhipster para el generado y desarrollo del frontend. Utiliza typescript como lenguaje y permite crear y mantener aplicaciones web de una sola página. Otra alternativa podría haber sido React, para la cual jhipster cuenta con soporte nativo.
4. IntelliJ - Visual Studio Code: Son los IDE utilizados para el desarrollo de la App. La elección del IDE queda principalmente a gusto del desarrollo, sin embargo, es recomendable que todo el equipo trabaje con el mismo IDE.

Deployment (local):

Es donde el desarrollador puede hacer un deploy de su código de forma local y obtener el mismo resultado que si lo estuviera ejecutando en los servidores ubicados en la nube. Esto le permite realizar pruebas y saber si sus cambios funcionarán en producción. Para esto las siguientes herramientas fueron elegidas:

¹⁸ <https://www.jhipster.tech/>



UNIVERSIDAD DE MENDOZA FACULTAD DE INGENIERÍA

1. JIB: Es un plugin disponible para Maven que permite generar una imagen docker de una aplicación en Java sin necesidad de contar con un *Docker daemon* local, así como también la subida de esta imagen a un Docker registry. Lo cual es una gran ventaja a la hora de construir las imágenes en un servidor que no cuenta con dockers.
2. Docker: permite la ejecución de imágenes de contenedores desligándose de todas las dependencias que tenga la aplicación o servicio contenida en esa imagen. Lo cual asegura que la aplicación funcionará exactamente igual en el entorno de desarrollo del programador como en el entorno de producción.
3. Minikube: herramienta que permite la ejecución de kubernetes de forma local, creando un cluster de kubernetes de un solo nodo en una máquina virtual.
4. Docker-compose: Alternativa más liviana a minikube, ya que permite correr las imágenes sin la necesidad de crear un cluster de kubernetes, con la desventaja de que no simula exactamente el mismo comportamiento que la aplicación en producción.

Cloud (CI/CD):

Es donde se encuentran todas las tecnologías que permiten la integración continua del código y el deployment automático de este a los servidores cloud (digitalOcean para este caso). Para esto se eligió:

1. GIT (GitHub): Git es un software de control de versiones que permite configurar un repositorio donde se encuentra todo el código fuente y tener un registro de todos los cambios que han ido sucediendo. También facilita la coordinación del trabajo en paralelo sobre un mismo código. Particularmente se utilizará GitHub como plataforma para almacenar el código. Ya sea para que los desarrolladores puedan compartir el código y obtener la última versión de este, o como para tomarlo como fuente a la hora de hacer los deployments automáticos.
2. Jenkins: Es un software de automatización que permite la creación y ejecución de pipelines.
3. Docker Hub: Docker Hub es un servicio proporcionado por Docker Inc para buscar, almacenar y compartir imágenes de contenedores



UNIVERSIDAD DE MENDOZA FACULTAD DE INGENIERÍA

con todo el equipo. Funciona como un repositorio para todas las imágenes docker de nuestro proyecto.

4. Cluster de kubernetes: Es donde se hará el deployment de la aplicación partiendo de todas las imágenes subidas en docker hub y de los archivos de configuración .yaml encontrados en github. Esto se hará mediante kubectl. Para este caso se utilizará digitalocean, otras alternativas podrían ser AWS, azure, google cloud, otras.



3.5. Proceso de Continuous Integration/Continuous Delivery (CI/CD)

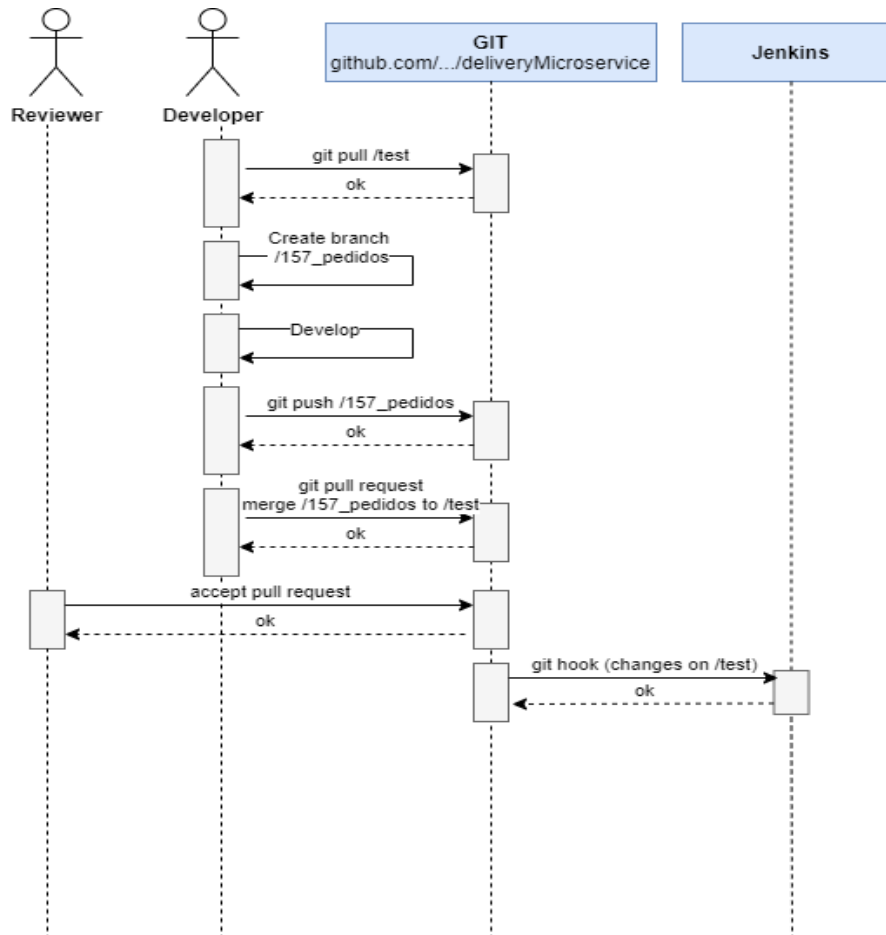
Introducción

La forma en que se realiza CI/CD es mediante la creación de pipelines en jenkins, una por cada servicio dentro de un entorno. Estos pipelines se ejecutan de forma automática al detectar cambios en determinados branches de los respectivos repositorios de git para cada servicio. Cada uno de estos repositorios contiene un jenkinsfile que configura el pipeline para crear, taggear y subir la imagen a docker hub mediante JIB y luego aplicar el cambio en el correcto cluster/namespace de kubernetes.

Diagrama de secuencia de un incremento en el código

Imaginemos que un desarrollador está trabajando en la funcionalidad de “carga de pedidos” del microservicio “Delivery Microservice” el proceso sería el siguiente:

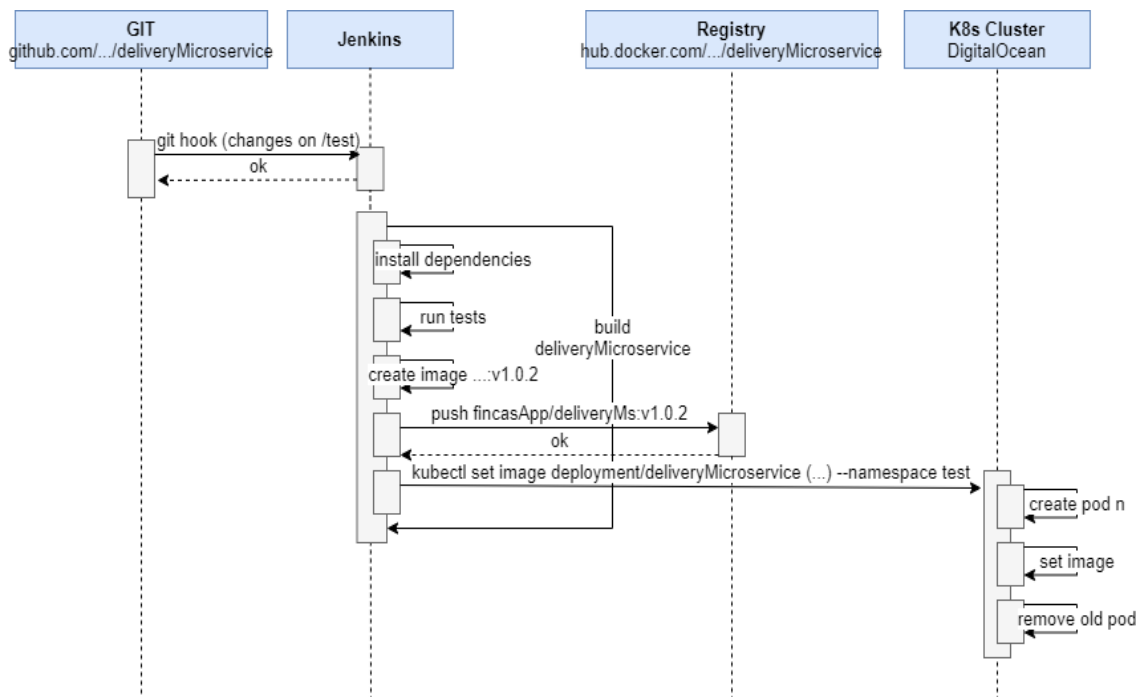
1 - Develop: Es el proceso que hace el desarrollador para incluir alguna nueva funcionalidad, arreglar algún bug, etc



1. El desarrollador toma un ticket del board para su desarrollo, en este caso: "Implementar carga de pedidos".
2. Se descarga (pull) el último código desde github del branch/entorno en el que esté trabajando, en este caso test.
3. Crea un nuevo branch con el ID del ticket (157) y el requerimiento (pedidos) como nombre del branch para que en el futuro sea fácil de identificar en caso de ser necesario.
4. Desarrolla todo el código que sea necesario - más adelante se define y aplica este proceso en detalle.
5. Sube su código al repositorio GIT en el branch correspondiente (git push 157_pedidos).
6. Crea un pull request para llevar su código al branch test.

7. Los revisores que haya designado el desarrollador aprueban el pull request y se mezcla el código. O en caso de que se rechace el pull request se vuelve al paso 4.
8. Git emite un evento (Git hooks) para notificar a nuestro servidor con Jenkins de que ha habido un cambio en el branch /test.
9. Jenkins recibe la notificación de git y arranca el proceso de integración continua.

2 - Integración Continua y deployment: Es el proceso automático que se lleva a cabo para realizar la integración continua (ejecutar los test y deployar una nueva versión en el entorno correspondiente)



1. Jenkins recibe la notificación de git luego de que ha habido un cambio en alguno de los branches para los cuales se han definido los hooks, estos podrían ser /test, /staging, /master o cualquier otro entorno que se haya definido.
2. Jenkins ejecuta el pipeline correspondiente al microservicio que haya sido actualizado (determinado por cuál fue el repositorio que emitió el hook):
 - a. Descarga el código fuente de git (git push) junto con el `jenkinsfile`.



UNIVERSIDAD DE MENDOZA FACULTAD DE INGENIERÍA

- b. Instala todas las dependencias necesarias (Java, maven dependencies, node dependencies, etc)
 - c. Ejecuta los test de unidad del backend y del frontend en caso de que exista (los microservicios no tienen frontend ya que estos se encuentran en los gateways).
 - d. Compila la aplicación y crea una imagen docker utilizando JIB con el siguiente nombre: <nombre-app>/<nombre-ms> y además se taggea con el número de versión y entorno, en el ejemplo: FincasApp/DeliveryMicroService:v1.0.2-test.
 - e. Sube la imagen al registry de docker en docker.hub.
 - f. Realiza el deployment de la nueva imagen en el cluster de kubernetes en digitalOcean
- 3. El cluster en digitalOcean realiza el deployment de la nueva imagen:
 - a. Descarga la versión de la imagen del registry de docker según la versión que recibe desde jenkins.
 - b. Crea un nuevo pod (o la cantidad que sea necesaria según la cantidad de recursos que necesite utilizar para el deploy de la imagen).
 - c. Se realiza el deployment de la imagen en ese nuevo pod.
 - d. Se redirecciona el tráfico a los pods que contienen la nueva versión y se eliminan los antiguos pods.

3.6. Proceso Iterativo e Incremental

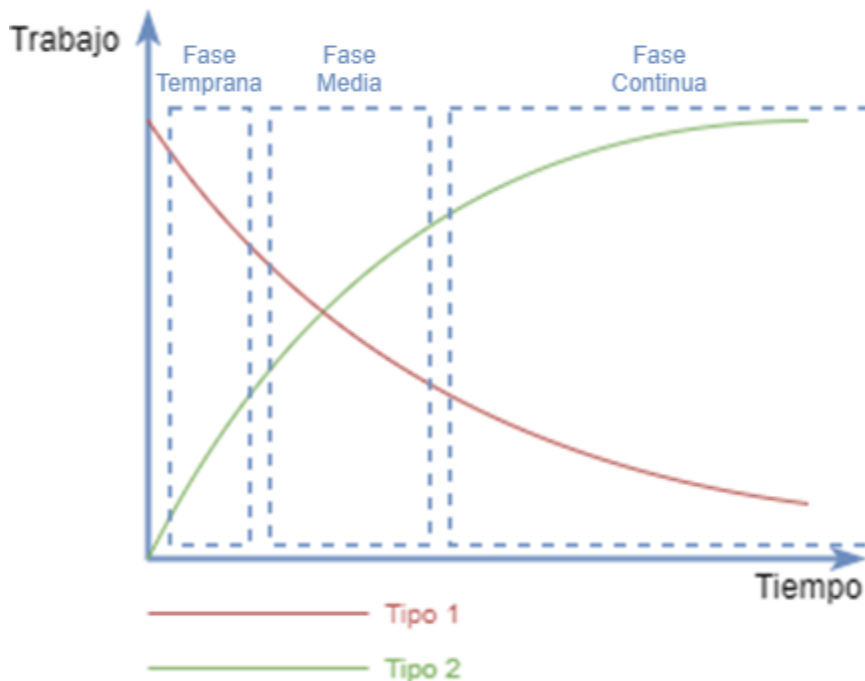
Introducción

Una vez que se ha definido el modo de trabajo, la arquitectura de la aplicación, el entorno y el proceso para la integración continua lo próximo es encontrar un proceso que sea iterable y que permita ir agregando valor al software de forma incremental.

Este proceso puede ser de dos maneras según si se quiere integrar/crear un nuevo microservicio, iteraciones del tipo 1, o si se desea desarrollar alguna nueva funcionalidad o reparación de bugs en un microservicio existente, iteración del tipo 2.

Las iteraciones del tipo 1 resultan mucho más complejas que las otras, ya que, en ellas se debe realizar todo el trabajo de automatización. Como contrapartida hace que las iteraciones del tipo 2 sean mucho más sencillas, ya que, la mayor parte del proceso será automático.

En fases tempranas del desarrollo se ha de suponer que habrá muchas iteraciones del tipo 1 y que con el tiempo casi no sean necesarias, enfocándonos únicamente en iteraciones del tipo 1.





UNIVERSIDAD DE MENDOZA FACULTAD DE INGENIERÍA

Como se puede observar en el gráfico, es posible distinguir tres fases del desarrollo:

1. Fase 1 o “Fase temprana”: En este punto, al no existir ningún microservicio, se trabajará puramente en el desarrollo de estos y los procesos de automatización necesarios. Por lo tanto, se trabajará casi exclusivamente en iteraciones del tipo 1.
2. Fase 2 o “Fase media”: En este punto ya existen ciertos microservicios, por lo tanto, algunos equipos pueden trabajar en el desarrollo de funcionalidades para estos y otros continuar con la integración de los nuevos.
3. Fase 3 o “Fase continua”: Finalmente, en este punto ya estarán desarrollado la mayoría de los microservicios necesarios y sus pipelines, por lo que la mayoría de la capacidad de trabajo estará destinada al desarrollo de funcionalidades de estos (iteraciones del tipo 2), dejando de lado la parte “Ops” para mantenimiento y eventualmente algún nuevo microservicio. Esta fase continuará en el tiempo tanto como lo haga el software que se esté desarrollando.

Iteración tipo 1: Integración de un nuevo microservicio

Este tipo de iteración será la que se deba realizar a la hora de incluir un nuevo microservicio donde, aparte del desarrollo de este, es necesario realizar la configuración necesaria para la integración continua. Los pasos son los siguientes:

1. Realizar diagramas: Tanto de la arquitectura (diagrama de microservicios) como del microservicio en si (diagramas de clase, secuencia, etc.).
2. Crear el microservicio y aplicar el schema inicial, así como también el CRUD básico para acceder a los datos en caso de ser necesario.
3. Crear un nuevo repositorio git que tenga la misma raíz que todos los microservicios de la aplicación y realizar un commit inicial.
4. Crear los archivos de configuración para los pipelines, en este caso un “jenkinsfile” y subirlos al repositorio.
5. Creado del pipeline en un servidor de automatización, para esto se utiliza Jenkins
6. Deployment en k8s
7. Deployment automatico
8. Creado de entornos de pruebas (Test, staging, etc.)



UNIVERSIDAD DE MENDOZA FACULTAD DE INGENIERÍA

Iteración tipo 2: Trabajar sobre un microservicio existente

Este es el tipo de iteración que se debe llevar a cabo a la hora de trabajar en una nueva funcionalidad, optimización o arreglos de bugs sobre un microservicio existente. Los pasos son los siguientes:

1. Realizar diagramas (diagrama de clase, secuencia, etc.).
2. Desarrollo de funcionalidad:
 - a. Descarga de repositorio.
 - b. Aplicar cambios de schema de la base de datos (en caso de ser necesario).
 - c. Diseño/mock de APIs (Api first development).
 - d. Escribir Unit tests (Test driven development).
 - e. Implementar nuevas funciones.
 - f. Subir al repositorio.
3. Integración continua (proceso automático)
4. Test



CAPÍTULO 4: APLICACIÓN DE LA METODOLOGÍA

4.1. Introducción

Si bien en el capítulo anterior ya se fue implementando algunas partes de la metodología, es el presente capítulo donde finalmente se trabaja en los sprints que agregan valor al producto. Al igual que en el capítulo anterior, se trabajará en el desarrollo de la aplicación *FincasApp*.

Al ser este un trabajo de investigación y no el desarrollo de un software real y acorde a los alcances y objetivos planteados en el capítulo 1, se han considerado las siguientes simplificaciones a la infraestructura:

1. **Entornos:** En lugar de tener 3 entornos (Test, Staging y Producción) dividido en 2 cluster se trabajará con un único cluster y 2 entornos en él (Test y Producción) con una división lógica - namespaces. Esto es así por la complejidad que conlleva, además de que para crear más entornos el procedimiento es el mismo que se realiza para crear Test y Producción.
2. **Git poll:** Al no contar con un servidor de automatización (Jenkins) en la nube, no es posible configurar *Git Hooks* que permitan escuchar a nuevos cambios en los repositorios, debido a que la IP del servidor es local. Por lo tanto, se utilizarán Git poll, en donde nuestro servidor pregunta a GIT si hay nuevos cambios en el repositorio cada cierto tiempo. De todas formas, siempre que se pueda es recomendable utilizar Git Hook, ya que el otro tipo de configuración resulta en muchas llamadas y procesamiento innecesario.

4.2. Configurado del servidor de automatización (Jenkins)

Una vez que se ha instalado el entorno y se cuenta con un servidor de Jenkins funcionando, antes de empezar a desarrollar la aplicación, lo próximo es configurarlo con las credenciales de acceso para Github, Dockerhub, y el cluster de kubernetes. Las credenciales de github y docker hub se almacenan como variables de entorno para que puedan ser utilizadas durante el Build. Y el acceso al cluster se configura en el mismo servidor para utilizarlo en los deployments.



UNIVERSIDAD DE MENDOZA FACULTAD DE INGENIERÍA

Credenciales de GitHub y Docker Hub

1. Se ingresa a la sección de “credentials” del servidor de jenkins:
`https://<url-server-jenkins>/credentials/`
2. En el dominio global se agrega una nueva credencial:

Stores scoped to Jenkins



3. Se ingresan los datos de acceso a Docker Hub:

Scope	Global (Jenkins, nodes, items, all child items, etc)
Username	fincasapp
Password
ID	docker-credential
Description	docker hub credentials

Save

Donde el ID es la forma en que luego se accedera a esta variable de entorno durante el build.



UNIVERSIDAD DE MENDOZA FACULTAD DE INGENIERÍA

4. De la misma manera se configuran los datos de acceso para GitHub:

Scope	Global (Jenkins, nodes, items, all child items, etc)
Username	fincaspp
Password
ID	5daad9fd-2b90-4e04-9df9-0c211266a17b
Description	Github

[Save](#)

Cualquier otra credencial, datos de accesos, tokens, API, secretos, etc que sean necesarios se deben configurar de la misma manera.

Configuración de acceso al cluster

También es necesario configurar *kubectll* en el servidor de jenkins para que este tenga acceso al cluster de kubernetes. En el caso de DigitalOcean una vez creado el cluster se puede descargar el archivo de configuración y colocarlo en el directorio de ~/.kube, que donde se almacenan las configuraciones de kubernetes, para tener acceso al cluster.

Una vez configurado con el comando “\$ kubectll get nodes”, desde alguna terminal dentro del servidor, se pueden visualizar todos los nodos del cluster:

```
jenkins@3fdd5b8b616f:/$ kubectll get nodes
NAME                                STATUS    ROLES    AGE    VERSION
pool-2nflx1llv-3skw2              Ready    <none>    15m    v1.19.3
pool-2nflx1llv-3skwl              Ready    <none>    15m    v1.19.3
pool-2nflx1llv-3skwp              Ready    <none>    15m    v1.19.3
```

4.3. Sprint 1: Desarrollo servidor OAuth2

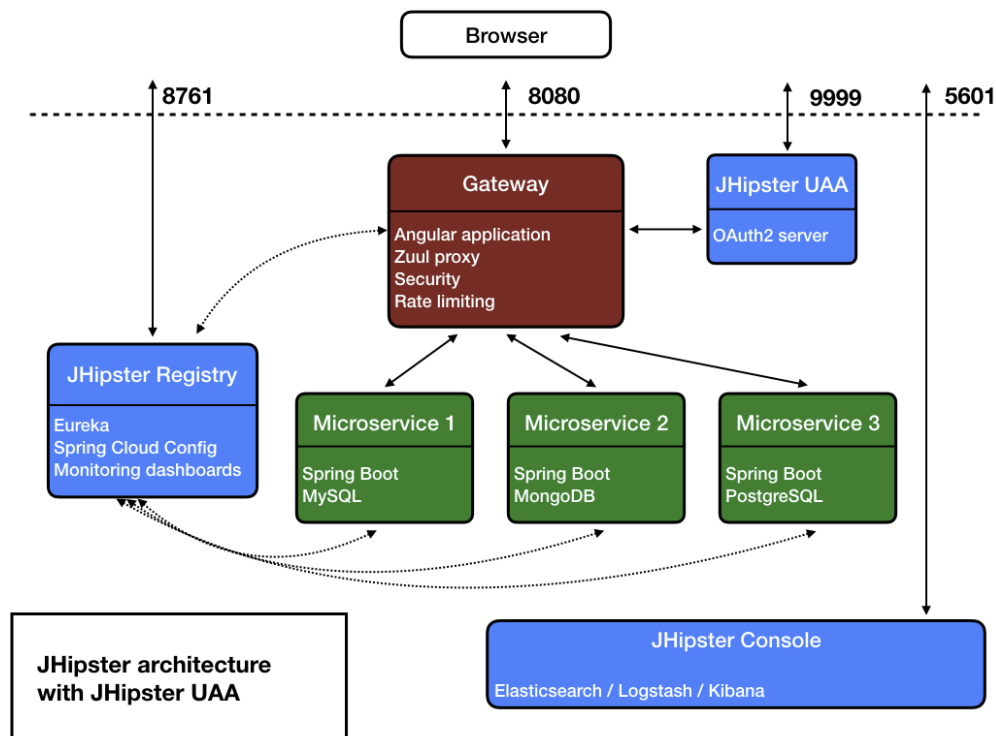
Introducción

En esta primera iteración se trabajará sobre el requerimiento de desarrollar el servicio de OAuth, ya que será necesario para almacenar los usuario y permitir el login y registro de estos en todos los otros servicios de la aplicación. Al no existir este microservicio esta iteración será del tipo “[Integración de un nuevo microservicio](#)”.

JHipster permite la creación automática de un servicio que utiliza el protocolo OAuth2 llamado JHipster UAA (User Account and Authentication server) que se integra de manera muy simple con el resto de la aplicación. Además Jhipster UAA, al igual que los otros microservicios, se conecta a JHipster Registry lo que le permite ser descubierto por otros microservicios. Por lo tanto el desarrollo de este servicio se hará de esa forma.

1. Diagramas

Diagrama de ejemplo de una arquitectura utilizando JHipster UAA:



19

¹⁹ <https://www.jhipster.tech/using-uaa/>



UNIVERSIDAD DE MENDOZA FACULTAD DE INGENIERÍA

2. Creado del Microservicio

El proceso para crear un servicio de JHipster UAA es el siguiente:

- a. Se crea una carpeta con el siguiente formato de nombre
“<nombreApp>-<nombreMs>” en este caso seria “fincasApp-uaa”:

```
$ mkdir fincasApp-uaa
```

- b. Dentro de esa carpeta se ejecuta el comando “jhipster” y se selecciona como tipo de aplicación a “JHipster UAA server”:

```
? Which *type* of application would you like to create?  
Monolithic application (recommended for simple projects)  
Microservice application  
Microservice gateway  
> JHipster UAA server
```

- c. Se completan el resto de las preguntas, se utilizó la opción por defecto en la mayoría de los casos. Para el nombre de la aplicación se utilizó: “fincasAppUaa”, base de datos: SQL y el nombre del paquete Java: “ar.edu.um.fincasapp.uaa”.

De esta manera queda creado el servicio de UAA y además JHipster genera los End Points necesarios para realizar el CRUD de usuarios y la generación y control de tokens, lo que luego permitirá la comunicación con los otros servicios de la aplicación. Por otro lado también genera test básicos para los controller (web.rest), los servicios y los repository.

Para poder testearlo localmente es necesario contar un JHipster Registry y una base de datos SQL. Para no tener que instalar, configurar, y correr estas herramientas de forma manual, se puede hacer con docker-compose. De esta se pueden obtener desde un repositorio las imágenes que sean necesarias (MySQL y jhipster registry) y del entorno local las que se están desarrollando para la aplicación (de momento solo sera fincasAppUaa) para luego ejecutarlas en un docker.

Las imágenes que se utilizaran en el docker-compose son:

1. Jhipster Registry (Imagen jhipster/jhipster-registry:v6.3.0).
2. Base de datos MySQL (Imagen mysql:8.0.20).
3. fincasAppUaa (Imagen local).

Las primeras dos imágenes jhipster-registry y mysql serán descargadas (pull) desde docker hub, en cambio la de fincasAppUaa debe ser creada,

para esto se realiza un proceso similar al que luego se llevará a cabo de forma automática en el build utilizando maven y el plugin JIB.

Dentro de la carpeta fincasApp-uaa se ejecuta:

```
$ ./mvnw -ntp -Pprod verify jib:dockerBuild
```

Este comando hará el build de la aplicación, ejecutará los tests y finalmente con el plugin de maven “Jib” creará la imagen en el entorno local. Si el build no devuelve ningún error se puede ver la imagen con el comando “\$ docker images:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
fincasappuaa	latest	f70d20283958	6 days ago	323MB

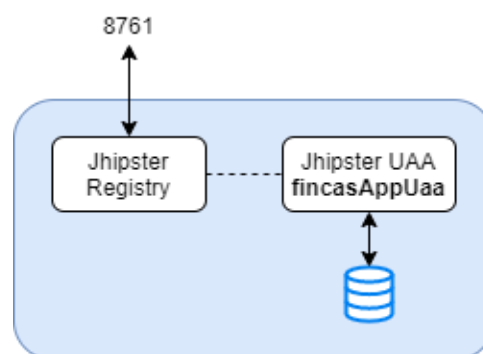
Para crear el docker-compose.yml se puede hacer de forma manual o de forma automática con jhipster. Esto resulta muy útil ya que cuando la aplicación crezca se encarga de generar todos los manifiestos para todos los microservicios y gateways de la App y sus dependencias (bases de datos, registry, etc).

Para esto se crea una nueva carpeta al mismo nivel que la de fincasAppUaa, obteniendo la siguiente estructura:

```
fincasApp
|-- fincasAppUaa
|-- docker-compose
```

Luego dentro de la carpeta docker-compose se ejecuta el siguiente comando: “\$ jhipster docker-compose”. Esto genera el docker-compose.yml y sus dependencias (registry.yml) con todas las configuraciones necesarias. Finalmente, con “\$ docker-compose up -d” podemos ejecutar toda nuestra infraestructura localmente.

De momento la infraestructura, bastante simple, se ve de la siguiente manera:





UNIVERSIDAD DE MENDOZA FACULTAD DE INGENIERÍA

En el gráfico se observa que solo se expone el puerto del jhipster registry. El de fincasAppUaa no se expone, ya que toda la comunicación entre los microservicios/gateways y UAA es interna.

Una vez que la aplicación está corriendo puede ser testeada con swagger en la siguiente url: <http://localhost:8761/admin/docs> donde 8761 es el puerto del container que está corriendo el jhipster-registry.

3. Creado del repositorio en GitHub

Una vez creado el microservicio y testado su funcionamiento de forma local, lo próximo sería subirlo a algún repositorio, en este caso github. Esto permite poder hacer el control de versiones y que esté disponible para todo el equipo de desarrollo.

Para *FincasApp* se definió una estructura para los nombres de todos los repositorios vinculados a la app para que sea fácil identificarlos y que compartan la misma raíz. El formato es el siguiente: “FincasApp-<nombre-del-microservicio>”, por lo tanto, para el microservicio actual el repositorio se llamará “FincasApp-uaa”.

Los pasos para el creado del repositorio son:

1. Se crea un nuevo repositorio vacío desde la aplicación de github con el formato de nombre definido: “FincasApp-uaa”
2. Se Inicializa el repositorio local:
 - a. Ingresar a la carpeta del microservicio:
`$ cd ~/<ruta>/fincasApp/fincasAppUaa`
 - b. Inicializar un repositorio de git:
`$ git init`
3. Se agrega el repositorio remoto de github:
`$ git remote add origin https://github.com/<usuario>/FincasApp-uaa.git`
4. Se sube el código al repositorio al branch master:
`$ git push -u origin master`

4. Archivos de configuración del Pipeline

Para configurar los pipelines, dependiendo del servidor de automatización que se use, se pueden configurar en el mismo servidor o que reciba la configuración de un archivo que podría ser un YAML, o en el caso de jenkins, un jenkinsfile. Para fincasApp se optó por la segunda alternativa, ya que esto permite tener la configuración dentro del repositorio y poder hacer un seguimiento de los cambios en la configuración de los pipelines.

Por lo tanto, será necesario crear un archivo *jenkinsfile* y luego subirlo al repositorio para posteriormente poder realizar el build partiendo de este archivo. El procedimiento es el siguiente:

1. Ir al a carpeta de fincasAppUaa:
\$ cd ~/<ruta>/fincasApp/fincasAppUaa
2. Se genera un archivo nuevo con el nombre "jenkinsfile":
\$ touch jenkinsfile
3. Se edita este archivo con el siguiente código:

```

1 #!/usr/bin/env groovy
2
3 node
4     stage('checkout') {
5         checkout scm
6     }
7
8     stage('check java') {
9         sh "java -version"
10    }
11
12    stage('clean') {
13        sh "chmod +x mvnw"
14        sh "./mvnw -ntp clean -P-frontend"
15    }
16    stage('nohttp') {
17        sh "./mvnw -ntp checkstyle:check"
18    }
19
20    stage('backend tests') {
21        try {
22            sh "./mvnw -ntp verify -P-frontend"
23        } catch(err) {
24            throw err
25        } finally {
26            junit '**/target/test-results/**/TEST-*.xml'
27        }
28    }
29
30    stage('packaging') {
31        sh "./mvnw -ntp verify -P-frontend -Pprod -DskipTests"
32        archiveArtifacts artifacts: '**/target/*.jar', fingerprint: true
33    }
34
35    def dockerImage
36    stage('publish docker') {
37        sh "./mvnw -ntp jib:build"
38    }
39

```

Descarga del código desde el repositorio

Verifica que java este instalado

Le da permisos de ejecución al script de maven "mvnw"

Borra cualquier otro build viejo que podría haber quedado

Verifica que el código este formateado correctamente

Ejecuta los test del backend y guarda los resultados "/test-results/"

Hace un build de la aplicación y guarda el resultado como un ejecutable de JAVA (.jar)

Arma un imagen de docker

Como se observa en la imagen, se ha definido una secuencia de stages que se llevará a cabo durante el build de la aplicación, estos son:

1. Descarga del código base
2. Verificar que Java esté instalado
3. Dar permisos de ejecución al script de maven y eliminar cualquier build viejo
4. Verificar que el código cumpla con las reglas de formato



UNIVERSIDAD DE MENDOZA FACULTAD DE INGENIERÍA

5. Ejecutar los tests
6. Hacer el build de la aplicación, para el cual la salida será un .JAR
7. Crear la imagen Docker

Si cualquiera de estos stages fallara, se interrumpe todo el build y se puede ver el error dentro de jenkins.

El último stage (#7) solo genera una imagen local pero no la sube a ningún lado, por lo que es necesario modificarlo y analizarlo en mayor detalle. Este stage, además de crear la imagen, debe ser capaz de “pushearla” al repositorio de docker, docker hub en este caso. Para esto se utilizarán las credenciales de docker hub que se configuraron previamente como variables de entorno. Por lo que el stage quedará como:

```
35     def dockerImage
36     stage('publish docker') {
37         withCredentials(
38             [usernamePassword(
39                 credentialsId: 'docker-credential',
40                 passwordVariable: 'DOCKER_REGISTRY_PWD',
41                 usernameVariable: 'DOCKER_REGISTRY_USER')]
42         ) {
43             sh "./mvnw -ntp jib:build"
44         }
45     }
46 }
```

Donde “credentialsId” es la misma que se definió al crear la credencial en jenkins. Y DOCKER_REGISTRY_PWD y DOCKER_REGISTRY_USER son los nombres de las variables de entorno.

Con esto ya se tiene acceso a las credenciales de docker hub durante el build, sin embargo, aún no se están utilizando. Como se observa en el stage, y como se ha explicado previamente, para el build se utiliza JIB y, como en todas las aplicaciones maven, la configuración de este se encuentra en el pom.xml:

```

565     <plugin>
566         <groupId>com.google.cloud.tools</groupId>
567         <artifactId>jib-maven-plugin</artifactId>
568         <version>${jib-maven-plugin.version}</version>
569         <configuration>
570             <from>
571                 <image>adoptopenjdk:11-jre-hotspot</image>
572             </from>
573             <to>
574                 <image>fincasappuaa:latest</image>
575             </to>
576         </configuration>
577     </plugin>
578 </plugins>

```

Y se modifica para que quede de la siguiente forma:

```

<plugin>
  <groupId>com.google.cloud.tools</groupId>
  <artifactId>jib-maven-plugin</artifactId>
  <version>${jib-maven-plugin.version}</version>
  <configuration>
    <from>
      <image>adoptopenjdk:11-jre-hotspot</image>
    </from>
    <to>      DockerHubId/nombreApp
      <image>facu077/fincasappuaa</image>
      <tags>
        <tag>${project.version}</tag>
        <tag>latest</tag>
      </tags>
      <auth>      Variables de entorno del build
        <username>${DOCKER_REGISTRY_USER}</username>
        <password>${DOCKER_REGISTRY_PWD}</password>
      </auth>
    </to>
  </configuration>
</plugin>

```



UNIVERSIDAD DE MENDOZA FACULTAD DE INGENIERÍA

Donde:

1. En `<image></image>` se indica el nombre del repositorio de dockerhub con el siguiente formato: “nombre-usuario-dockerhub”/”nombre-del-repositorio”.²⁰
2. En `<tags></tags>` se indica cómo se taguean las imágenes que se subirán al repositorio para lo cual se utilizó la versión de la app definida en el pom.xml y también, como buena práctica, se utilizó el tag “latest” para siempre poder identificar la última version, aun cuando no se conozca el número de esta.
3. En `<auth></auth>` se indican cuáles son las credenciales, mediante las variables de entorno, para poder ingresar a docker hub.

Hecho esto, solo queda subir los cambios al repositorio:

1. Se trackean todos los cambios locales:
\$ git add -A
2. Se realizamos el commit con un mensaje:
\$ git commit -m “pipeline setup”
3. Se realizamos el push:
\$ git push

5. Creado del Pipeline


Creado el archivo de configuración, el próximo paso es crear el pipeline que lo utilice, en el caso de Jenkins el procedimiento es el siguiente:

- A. En la página principal de Jenkins se selecciona la opción “new item” y luego “pipeline” y se lo nombra como “uaa build”:


²⁰ Para este caso se utilizó un repositorio personal, por lo que tiene el nombre “facu077” pero lo normal sería que tenga el nombre de la aplicación o de la empresa.

Enter an item name


» Required field

**Freestyle project**

This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.

**Pipeline**

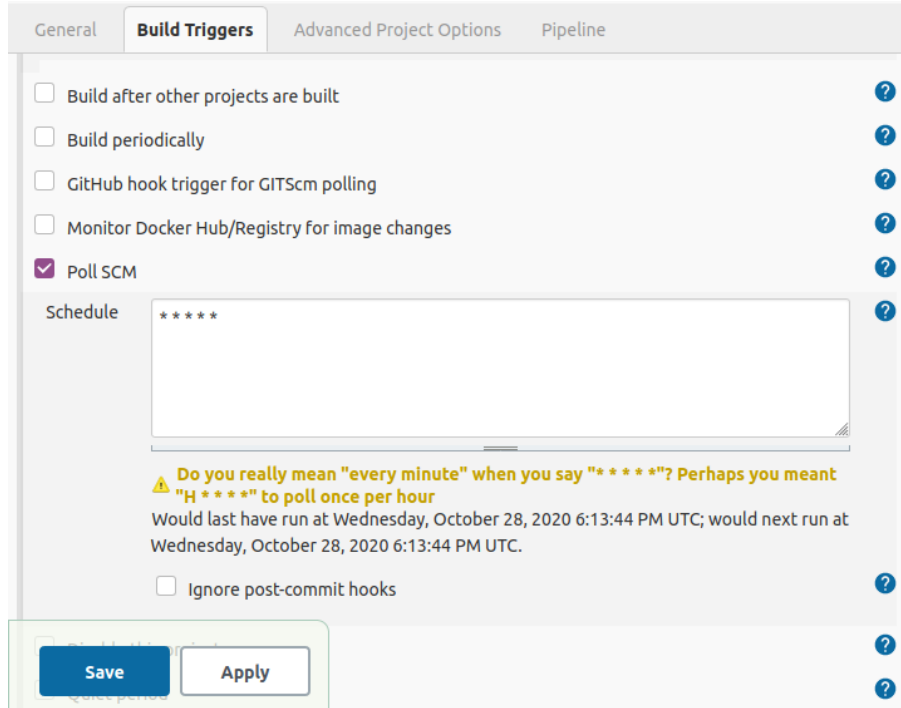
Orchestrates long-running activities that can span multiple build agents. Suitable for building pipelines (formerly known as workflows) and/or organizing complex activities that do not easily fit in free-style job type.

**Multi-configuration project**

Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.

OK

- B. Luego, en la sección de “build triggers”, como indica su nombre, es donde se configuran cuáles serán los eventos que harán que se ejecute el build, permitiendo la integración continua. Se recomienda utilizar “GitHub hook”, al ejecutar el build, si Jenkins tiene configurado el acceso a Github, se creará automáticamente este git hook. Sin embargo, por los motivos que se explicaron al [inicio de este capítulo](#), se utilizará “git poll”:



General **Build Triggers** Advanced Project Options Pipeline

☐ Build after other projects are built

☐ Build periodically

☐ GitHub hook trigger for GITScm polling

☐ Monitor Docker Hub/Registry for image changes

☒ Poll SCM

Schedule *****

⚠ Do you really mean "every minute" when you say "*****"? Perhaps you meant "H*****" to poll once per hour

Would last have run at Wednesday, October 28, 2020 6:13:44 PM UTC; would next run at Wednesday, October 28, 2020 6:13:44 PM UTC.

☐ Ignore post-commit hooks

Save Apply

Esta configuración hará que el servidor de Jenkins verifique cada 1 minuto si hay algún cambio en el repositorio al que apunta el pipeline, y de ser así, ejecutar el build.

- C. La sección de "Pipeline" se configura de la siguiente manera:
- Definition:** Pipeline script from SCM - Le indicamos que el build se debe hacer desde un script (jenkinsfile) ubicado en un repositorio.
 - SCM:** GIT - Indicamos que el repositorio es GIT
 - Se configura la URL del repositorio en github y se seleccionan las credenciales configuradas previamente para esto.
 - Branches to build:** */master - Al ser el build de producción se realizará sobre el branch master.

- D. Se guarda esta configuración y luego de un minuto un build debería correrse de forma automática con todos los steps que se definieron previamente en el Jenkinsfile, dando como resultado final la subida de la imagen de *fincasAppUaa* al Docker Hub:

Stage View

	checkout	check java	clean	nohttp	backend tests	packaging	publish docker
Average stage times: (Average full run time: ~5min 59s)	7s	636ms	2s	2s	1min 6s	8s	4min 20s
#1 Oct 28 15:22 No Changes	7s	636ms	2s	2s	1min 6s	8s	4min 20s

E. Y en el dashboard de Docker Hub se puede visualizar la imagen subida con los tags que fueron indicados en el pom.xml:

Repositories > [facu077 / fincasappuaa](#) > Using 0 of 1 private repositories. [Get more](#)

General **Tags** Builds Timeline Collaborators

Webhooks Settings

☐ Action

 Sort by

☐

TAG

[latest](#) docker pull facu077/fincasappuaa:latest

Last pushed a month ago by [facu077](#)

DIGEST	OS/ARCH	LAST PULL	COMPRESSED S...
203d649b41da	linux/amd64	5 minutes ago	166.74 MB

☐

TAG

[0.0.1](#) docker pull facu077/fincasappuaa:0.0.1

Last pushed 2 months ago by [facu077](#)

DIGEST	OS/ARCH	LAST PULL	COMPRESSED S...
1bd4df4853ad	linux/amd64	a month ago	166.74 MB

6. Deployment en Kubernetes

Hasta el momento se tiene un pipeline en Jenkins que permite la integración continua realizando un build del microservicio y guardando el resultado, como imagen Docker, en Docker Hub.



UNIVERSIDAD DE MENDOZA FACULTAD DE INGENIERÍA

A continuación, se indica el procedimiento para realizar el paso final en el cual se deploya esta imagen dentro de un cluster de kubernetes. Este procedimiento puede variar según cual sea el proveedor cloud (DigitalOcean en este caso), sin embargo, desde un punto de vista general, el procedimiento debería ser muy similar.

Suponiendo que ya se ha creado el cluster de kubernetes y ha sido configurado como se especificó en [el punto 4.2. de este capítulo](#), antes de poder agregar un step al build que realice el deployment de forma automática, es necesario aplicar de forma manual un manifest con el deployment dentro del cluster y namespace correspondiente.

Para crear los manifest con los deployments y los services de kubernetes se utilizará JHipster. El procedimiento es el siguiente:

1. Se crea una nueva carpeta al mismo nivel que las anteriores llamada “fincasApp-k8s”, obteniendo la siguiente estructura:

```
fincasApp
|-- fincasApp-uaa
|-- docker-compose
|-- fincasApp-k8s
```

2. Dentro de la carpeta recién creada, se ejecuta el comando “\$ jhipster kubernetes”. Esto genera los deployments para cada uno de los microservicios y sus correspondientes services, que serán del tipo LoadBalancer si se trata de un gateway y del tipo ClusterIP para cualquier otro recurso que solo se utilice internamente (Base de datos, JHipster Registry, microservicios, etc). También creará un bash “kubectl-apply.sh” que al ejecutarlo se encarga de ejecutar “kubectl apply -f” para cada uno de los deployments en el orden adecuado:

Orden de ejecución	Deployments	Tipo de Service	Puerto por defecto
1	JHipster Registry	ClusterIP	8761
2	Base de datos	ClusterIP	3306
3	Microservicios	ClusterIP	8080
4	UAA Server	ClusterIP	80
5	Gateways	LoadBalancer	8080

- El primer deployment se debe realizar de forma manual, luego se lo actualizará mediante el pipeline en Jenkins. Para aplicar los deployments en el cluster de kubernetes que esté configurado, en este caso el de DigitalOcean, se ejecuta el bash con la opción -f para utilizar kubectl:
\$./kubectl-apply.sh -f
- Finalmente, se puede observar que si el deployment se realizó con éxito con los siguientes comandos:

```
facundo@facundo-desktop:~/Desktop/fincasApp/fincasApp-uaa$ kubectl get deployments
NAME                READY   UP-TO-DATE   AVAILABLE   AGE
fincasappuaa        1/1     1             1           31m
fincasappuaa-mysql  1/1     1             1           31m
facundo@facundo-desktop:~/Desktop/fincasApp/fincasApp-uaa$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
fincasappuaa-5656988c97-l9cxk      1/1     Running   0           31m
fincasappuaa-mysql-58f8c78c77-fkgtg 1/1     Running   0           31m
jhipster-registry-0                1/1     Running   0           31m
jhipster-registry-1                1/1     Running   0           30m
facundo@facundo-desktop:~/Desktop/fincasApp/fincasApp-uaa$ kubectl get services
NAME                TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)    AGE
fincasappuaa        ClusterIP   10.245.77.224 <none>        80/TCP      31m
fincasappuaa-mysql  ClusterIP   10.245.205.171 <none>        3306/TCP    31m
jhipster-registry   ClusterIP   None          <none>        8761/TCP    31m
kubernetes           ClusterIP   10.245.0.1    <none>        443/TCP     70m
```



UNIVERSIDAD DE MENDOZA FACULTAD DE INGENIERÍA

Nótese que el jhipster registry no tiene ninguna IP asignada, esto es porque por defecto se crea como un headless service. Por lo tanto, para poder acceder al registry, y con eso a los logs de la aplicación, se puede crear otro servicio del tipo LoadBalancer que permita el acceso a este:

1. Creamos el service:
`$ kubectl expose service jhipster-registry --type=LoadBalancer --name=exposed-registry`
2. Inspeccionamos el nuevo service, que luego de un tiempo debería tener una IP asignada por el LoadBalancer de DigitalOcean:
`$ kubectl get svc exposed-registry`

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
exposed-registry	LoadBalancer	10.245.48.92	157.230.199.146	8761:31629/TCP	5m44s

Ahora en <http://157.230.199.146:8761/> se puede acceder al registry y analizar el estado de salud de los microservicios, logs, métricas, etc:

Registry v6.3.0

Home Eureka Configuration Administration

JHipster Registry v6.3.0

Refresh now disabled

System Status	
Environment	JHipster-Environment
Data Center	JHipster-DataCenter
Current Time	2020-12-16T21:17:04 +0000
System Uptime	00:11
Below Renew Threshold	false

Instances Registered		
App	Instance ID	Status
FINCASMS	fincasms:8b067f2bdcba7e70a3f833d10303d8ba	UP
JHIPSTER-REGISTRY	jhipsterRegistry:169173e1d756438cea25c6c53d38bde4	UP
JHIPSTER-REGISTRY	jhipsterRegistry:823d1817ecdc5a81939cf9f7813930c3	UP
WEBGATEWAY	webgateway:f4372e317a844c5013d4c4a8a2eae7e6	UP

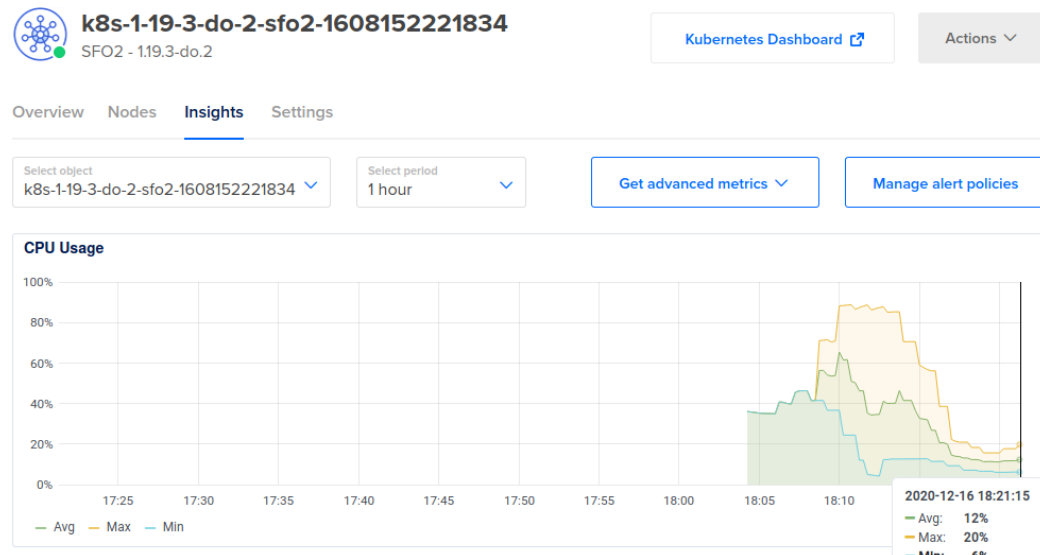
21

²¹ En la imagen se observa que ya están creados los microservicios FINCASMS y WEBGATEWAY porque fue obtenida en una fase más avanzada del desarrollo.



UNIVERSIDAD DE MENDOZA FACULTAD DE INGENIERÍA

También es posible acceder a distintas métricas sobre la carga, el tráfico, uso de CPU, uso de memoria, etc. así como también configurar alertas desde el dashboard de DigitalOcean:



7. Deployment automáticos

Por último, se agrega un nuevo step al pipeline encargado de setear la nueva imagen al deployment. Para esto editamos el Jenkinsfile dentro de fincasApp-uaa y agregamos este nuevo step:

```
def pomVersion = readMavenPom().version
stage( name: 'Deploy' ) {
    sh script: "kubectl set image deployment/fincasappuaa \
        fincasappuaa-app=facu077/fincasappuaa:${pomVersion} --namespace=default"
}
```

Guardamos en una variable la version obtenida desde el pom.xml

Indicamos la nueva imagen a setear en el deployment

De esta forma luego de haber creado y subido la imagen a docker hub le indicamos al cluster que reemplace la imagen que tiene el deployment “fincasappuaa” con una nueva que recibe el tag de la versión por variable de entorno, que es la misma que utilizamos en el push de la imagen.

Así mismo también se evita tener down time, ya que para el deployment se crean nuevos pods en los cuales se setea la nueva imagen y una vez que están listos se borran los antiguos.

Finalmente hacemos un push al repositorio de github y un nuevo build debería arrancar. Y a partir de este momento con cada actualización en el branch master se debería deployar la última versión en el cluster de kubernetes.

Es importante tener en cuenta que cada vez que se hace un merge/push al branch master se debe actualizar el número de versión el pom.xml de forma manual para que las imágenes se tagueen correctamente.

8. Creado del entorno TEST (u otros)

El procedimiento para crear otros entornos, en este caso uno para TEST, es muy similar a lo realizado hasta el momento:

1. Se crea un nuevo namespace en el cluster con el comando “\$ kubectl create namespace test”:

```
facundo@facundo-desktop:~/Downloads$ kubectl create namespace test
namespace/test created
facundo@facundo-desktop:~/Downloads$ kubectl get namespaces
NAME                STATUS    AGE
default              Active    8m35s
kube-node-lease      Active    8m36s
kube-public          Active    8m36s
kube-system          Active    8m36s
test                 Active    14s
```

2. Se crea el deployment con jhipster pero esta vez se especifica que el namespace sera “test” y que la imagen que utilice el deployment debe ser la que está taggeada como test:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: fincasappuaa
  namespace: test
spec:
  replicas: 1
  selector: ...
  template:
    metadata: ...
    spec:
      affinity: ...
      initContainers: ...
      containers:
      - name: fincasappuaa-app
        image: facu077/fincasappuaa:test
        env:
```

3. Posteriormente se debe crear un nuevo branch con el nombre test en el repositorio de git:
 - a. Se crea el branch: \$ git branch test

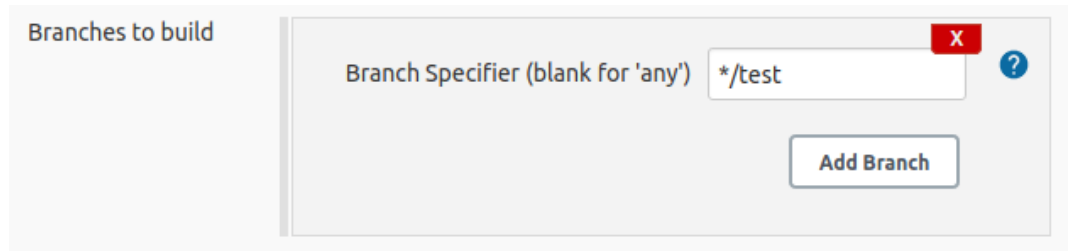
- b. Se selecciona: \$ git checkout test
- c. Se modifica la versión en el pom.xml agregando "<version>-test" para poder identificarla como tal:

```
<version>0.0.1-TEST</version>
```

- d. También en el pom.xml, en la configuración de JIB, cambiamos el tag "latest" por "test" con el fin de que "latest" quede solo reservado para la versión estable y que con test se pueda identificar la última versión de prueba.
- e. Por último se debe modificar el step final del jenkinsfile para que realice el deployment en un namespace con el nombre "test":

```
def pomVersion = readMavenPom().version
stage('Deploy') {
    sh "kubectl set image deployment/fincasappvaa \
        fincasappvaa-app=facu077/fincasappvaa:${pomVersion} --namespace=test"
}
```

- f. Se realiza el commit (\$ git commit -m "initial commit") y el push ("\$ git push -u origin test") a github.
4. Se crea el build en jenkins con el nombre "uaa build TEST" a partir del build creado anteriormente, modificando únicamente el branch que se utilizara para el build y trigger de este:



5. Después de un minuto un build debería iniciar y setear los deployments para el namespace test con las imágenes creadas para este entorno durante el mismo build:

```
facundo@facundo-desktop:~/Desktop/fincasApp/fincasApp-k8s-test$ kubectl get pods --namespace=test
```

NAME	READY	STATUS	RESTARTS	AGE
fincasappvaa-54767fb798-lf745	1/1	Running	0	5m5s
fincasappvaa-mysql-58f8c78c77-gcrdb	1/1	Running	0	5m5s
jhipster-registry-0	1/1	Running	0	5m7s
jhipster-registry-1	1/1	Running	0	4m38s

De esta forma tenemos un entorno para producción y otro para test para el microservicio de UAA ambos con integración continua que se ejecutará



UNIVERSIDAD DE MENDOZA FACULTAD DE INGENIERÍA

cada vez que se suba algún cambio a git en alguno de estos branch. El branch “master” debería ser únicamente modificado al realizar un merge desde “test” a “master”.

4.4. Sprint 2: Creado de Fincas Microservice

Introducción

En este sprint se creará el primer microservicio propio de la aplicación, y probablemente el más importante de la aplicación. Se trata del microservicio “Fincas” que es el que se encarga de toda la gestión de las fincas y sus elementos relacionados (vehiculos, maquinas, campos, etc).

Al igual que en el anterior Sprint se trata de una iteración del tipo 1 y el proceso es similar, salvo por algunas diferencias propias del desarrollo de este microservicio que se detallaran a continuación.

1. Diagramas

Diagrama de clases (General)

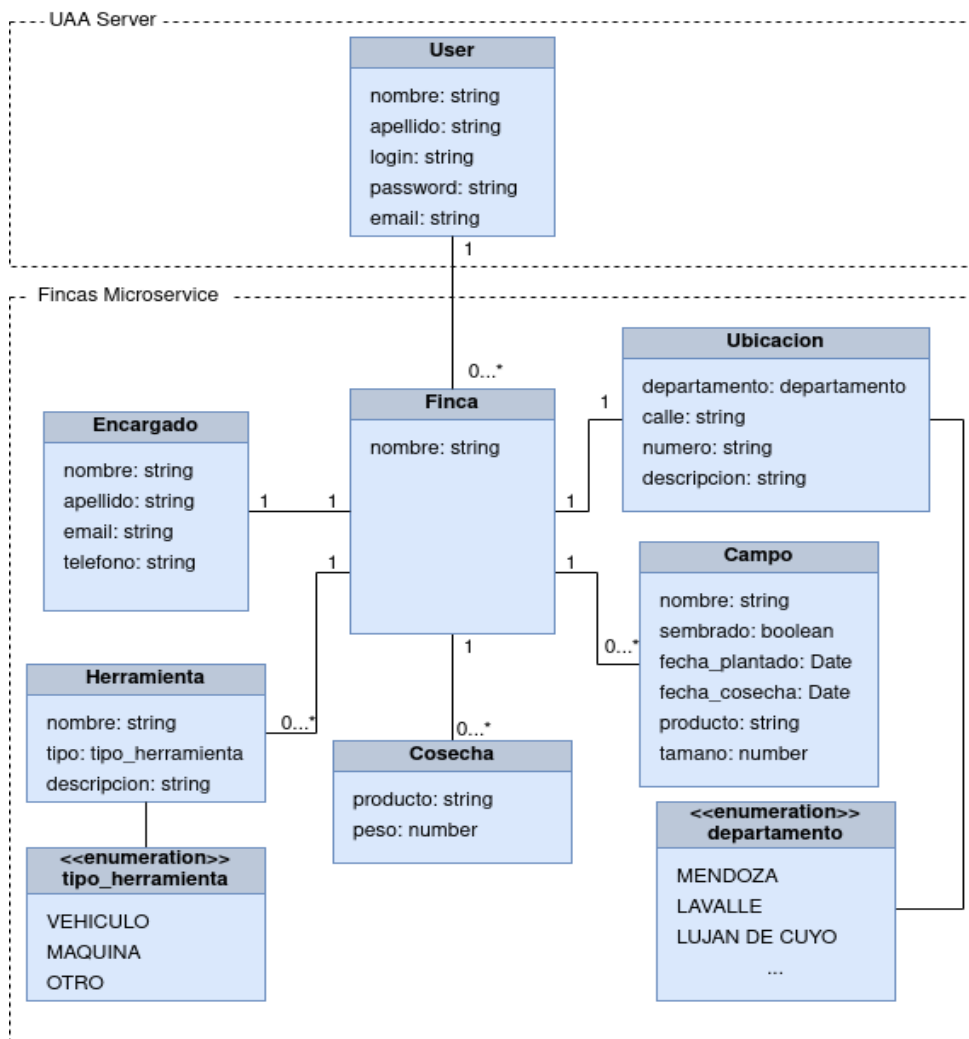
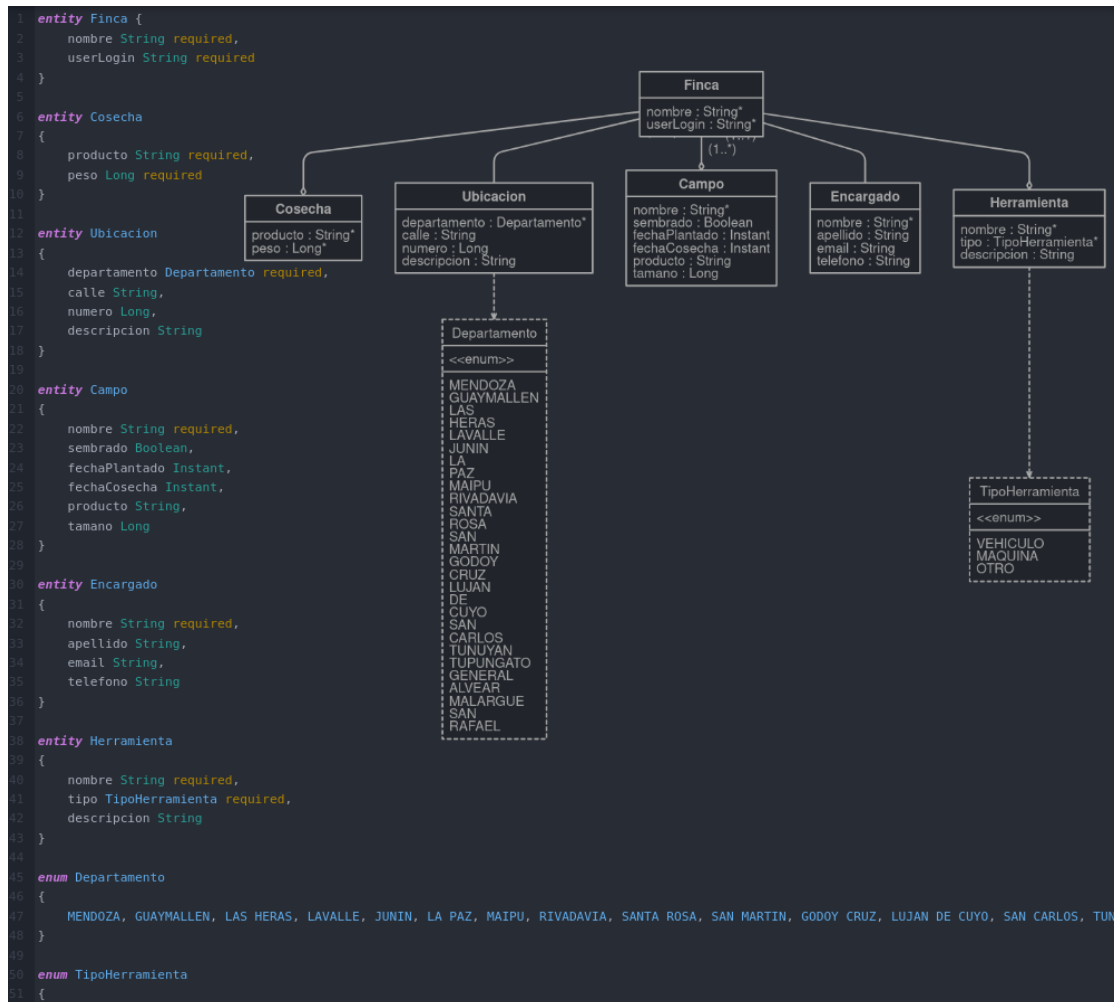


Diagrama de entidades (jdl-studio - propio de JHipster)



Similar al diagrama de clases, pero en un formato que jhipster pueda importar. La mayor diferencia se da en la clase “Finca”, que tiene un campo extra “userLogin” para almacenar el id del usuario, ya que la arquitectura de jhipster no permite una relación directa con la entidad “User” que se encuentra el servicio de UAA.

2. Creado del Microservicio

Para la creación del microservicio se utilizará jhipster y la herramienta jdl-studio, que permite armar el schema inicial e importarlo a un proyecto de jhipster. Jhipster se encargará de crear las clases, servicios, controladores y endpoints para realizar el CRUD de cada una de las entidades.

1. Se crea una nueva carpeta en el nivel más alto de la estructura de carpetas definidas con el nombre de “finca-microservice”:



UNIVERSIDAD DE MENDOZA FACULTAD DE INGENIERÍA

```
fincasApp  
|-- fincasApp-uaa  
|-- docker-compose  
|-- fincasApp-k8s  
|-- fincas-microservice
```

2. Dentro de esta carpeta se genera el microservicio con jhipster, que al igual que con el servidor de OAuth, se generan automáticamente los servicios, controladores, repositorios y configuraciones básicas (base de datos, conexión al Jhipster registry y autenticación con fincasApp-uaa) de la App.
3. Se importa al proyecto el archivo .jdl, generado previamente a partir del diagrama de entidades, con el comando:
"\$ jhipster import-jdl fincas.jdl"
Es recomendable guardar este archivo dentro de la carpeta de fincas-microservice por si hay que hacer alguna modificación en el futuro.

Finalmente, se lo puede testear localmente con docker-compose generando el archivo para esto desde Jhipster como se explicó anteriormente.

3. Creado del repositorio en GitHub

Al igual que con FincasAppUaa se crea el repositorio en git con el nombre "FincasApp-fincasMs" y se realiza el push inicial con el microservicio recién creado.

4. Archivos de configuración del Pipeline

Se crea el Jenkinsfile con "\$ jhipster ci-cd" y, al igual que antes, se hacen las modificaciones necesarias en el Jenkinsfile y el pom.xml.

5. Creado del Pipeline

Se crea un nuevo build en Jenkins "build fincasMs" donde la única diferencia con respecto al build de fincasAppUaa es que este debe utilizar el repositorio de git creado para este microservicio.



UNIVERSIDAD DE MENDOZA FACULTAD DE INGENIERÍA

Build ejecutado en jenkins:

	checkout	check java	clean	nohttp	backend tests	packaging	publish docker
Average stage times: (Average full run time: ~7min 35s)	27s	1s	10s	2s	2min 12s	8s	4min 21s
#1 Nov 05 20:30 No Changes	27s	1s	10s	2s	2min 12s	8s	4min 21s

Resultado en Docker Hub:

facu077 / fincasms Updated 5 minutes ago	Not Scanned	☆ 0	↓ 2	Public
facu077 / fincasappuaa Updated 3 days ago	Not Scanned	☆ 0	↓ 41	Public

6. Deployment en Kubernetes

Se generan los archivos de configuración para kubernetes con “\$ jhipster kubernetes” en fincasApp-k8s que esta vez, además de crear el deployment para fincasApp-uaa, lo hará para fincasMs.

Se agregan de forma manual, por única vez, los nuevos deployments con el bash “\$./kubectl-apply.sh -f”.

Resultado en el cluster:

```
facundo@Facundo-desktop:~/Desktop/fincasApp/fincasApp-k8s$ kubectl get all
```

NAME	READY	STATUS	RESTARTS	AGE
pod/fincasappuaa-5656988c97-nrx8p	1/1	Running	0	4m
pod/fincasappuaa-mysql-58f8c78c77-wq9qs	1/1	Running	0	3m59s
pod/fincasms-896f98c66-k6mmf	1/1	Running	0	4m3s
pod/fincasms-mysql-75cc5c9d69-87l8m	1/1	Running	0	4m2s
pod/jhipster-registry-0	1/1	Running	0	4m4s
pod/jhipster-registry-1	1/1	Running	0	3m50s

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
service/fincasappuaa	ClusterIP	10.245.35.150	<none>	80/TCP	4m
service/fincasappuaa-mysql	ClusterIP	10.245.124.175	<none>	3306/TCP	4m
service/fincasms	ClusterIP	10.245.138.209	<none>	8090/TCP	4m2s
service/fincasms-mysql	ClusterIP	10.245.117.233	<none>	3306/TCP	4m3s
service/jhipster-registry	ClusterIP	None	<none>	8761/TCP	4m5s
service/kubernetes	ClusterIP	10.245.0.1	<none>	443/TCP	13m

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
deployment.apps/fincasappuaa	1/1	1	1	4m1s
deployment.apps/fincasappuaa-mysql	1/1	1	1	4m
deployment.apps/fincasms	1/1	1	1	4m4s
deployment.apps/fincasms-mysql	1/1	1	1	4m3s



UNIVERSIDAD DE MENDOZA FACULTAD DE INGENIERÍA

7. Deployment automáticos

Por último se agrega un nuevo step en el build (jenkisfile) que está dentro de `./fincas-microservice` para que actualice la imagen del deployment cada vez que se realice un nuevo build.

```
def pomVersion = readMavenPom().version
stage('Deploy') {
    sh "kubectl set image deployment/fincasms fincasms-app=facu077/fincasms:${pomVersion} --namespace=default"
}
```

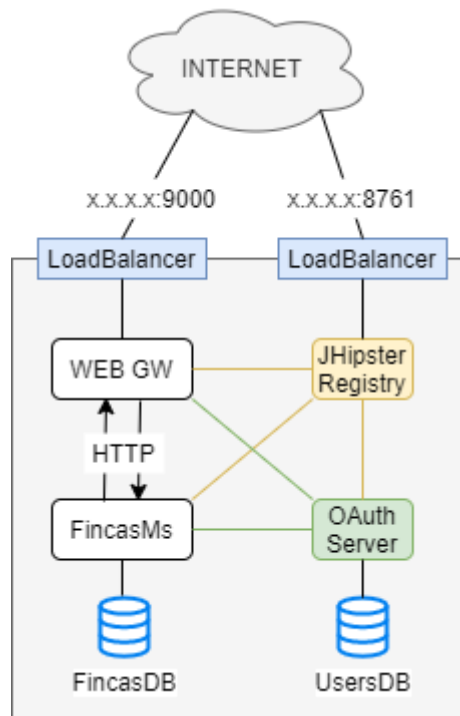
8. Creado del entorno TEST (u otros)

Mismo procedimiento que para fincasAppUaa.

4.5. Sprint 3: Creado del “WEB Gateway”

1. Diagramas

Arquitectura de la aplicación con el WEB Gateway:



2. Creado del Microservicio

Este microservicio, al tratarse de un gateway, estará compuesto más que nada por frontend (angular) y un backend (java) muy pequeño que permita la autenticación con el microservicio de UAA. Aquí es donde se configurarán las interfaces para la WEB y las llamadas a los diferentes backends/microservices mediante HTTP.

Por otro lado, a diferencia de FincasMs, es necesario que el gateway exponga su dirección IP a internet para poder acceder a la aplicación, para esto, dentro del cluster de k8s, se utilizará un service del tipo LoadBalancer que tendrá un IP pública y hará el ruteo al gateway.

Para crear el gateway con Jhipster se crea una nueva carpeta llamada web-gateway y dentro se ejecuta el comando “jhipster” seleccionando como tipo de aplicación: “gateway”, nombre: “webGateway” y sin base de datos. Esto generará un nuevo microservicio como se describió



UNIVERSIDAD DE MENDOZA FACULTAD DE INGENIERÍA

anteriormente y, que además, en el momento de su ejecución se registre en el “jhipster registry”.

Hasta el momento se ha creado un microservicio que se comunica con “fincasAppUaa” y un frontend sencillo que permite realizar el login y creación de usuarios. Pero todavía no existe ningún tipo de comunicación hacia “fincasMs”. Para eso es necesario tener el mismo modelo de datos de las entidades de fincasMs en el frontend y los services.ts necesarios que realicen las llamadas HTTP a los endpoints de “fincasMs”.

Para crear las entidades y servicios en el frontend se puede hacer de forma manual o, al igual que como se hizo con “fincasMs”, de forma automática con jhipster importando un “.jdl”. El proceso automático es similar, la única diferencia será que en el .jdl se debe indicar a que microservicio pertenecen las entidades de la siguiente manera:

“microservice <ENTITIES> with <MICROSERVICE_APP_NAME>”

Por lo que en este caso quedaría como:

“microservice * with fincasMs”

De esta forma se genera la estructura de datos para las clases (models) en el frontend y la estructura necesaria de angular (route, module, service, component y html) para generar pantallas sencillas que permiten realizar el CRUD de estas entidades.

De momento el comportamiento es similar al de una aplicación monolítica, donde fincasMs es el backend y web-gateway el frontend. La principal diferencia está en la comunicación con el jhipster-registry y el servidor de OAuth. En el startup del gateway (o cualquier otro microservicio), al registrarse en el jhipster registry, también se obtiene la URL donde están registrados todos los microservicios y se almacena en una constante. De esta forma se pueden obtener la URL de los diferentes endpoints, por ejemplo, para la entidad fincas de fincasMs sería:

resourceUrl = SERVER_API_URL + 'services/fincasms/api/fincas'

Una llamada del tipo HTTP GET a esa URL debería devolver todas las fincas.

3. Creado del repositorio en GitHub

Se crea un nuevo repositorio llamado “fincasApp-webGW” y se realiza el commit + push inicial.



UNIVERSIDAD DE MENDOZA FACULTAD DE INGENIERÍA

4. Archivos de configuración del Pipeline

Se genera el Jenkinsfile base con “\$ jhipster ci-cd” y se le aplican las modificaciones necesarias, así como también en el pom.xml.

La única diferencia de este pipeline es que tendrá un step adicional en cual se controlan los test del frontend:

```
stage('frontend tests') {  
  try {  
    sh './mvnw -ntp com.github.eirslett:frontend-maven-plugin:npm -Dfrontend.npm.arguments=\'run test\''  
  } catch(err) {  
    throw err  
  } finally {  
    junit '**/target/test-results/**/TEST-*.xml'  
  }  
}
```

5. Creado del Pipeline

Se crea un nuevo build en Jenkins “build webGW” que utilice el repositorio de git creado para este gateway.

Build ejecutado en Jenkins:

	checkout	check java	clean	nohttp	install tools	npm install	backend tests	frontend tests	packaging	publish docker
Average stage times: (Average full run time: ~9min 24s)	13s	654ms	2s	2s	5s	1min 3s	1min 14s	38s	1min 34s	4min 16s
#2 Nov 09 16:23 No Changes	13s	654ms	2s	2s	5s	1min 3s	1min 14s	38s	1min 34s	4min 16s

DockerHub:

facu077 / webgateway Updated a minute ago	Not Scanned	☆ 0	↓ 1	Public
facu077 / fincasms Updated 4 days ago	Not Scanned	☆ 0	↓ 12	Public
facu077 / fincasappuaa Updated 7 days ago	Not Scanned	☆ 0	↓ 42	Public

6. Deployment en Kubernetes

Se ejecuta el comando “\$ jhipster kubernetes” dentro del directorio fincasApp-k8s para que también genere el deployment del gateway, que a diferencia de los demás, el service de este deployment será del tipo LoadBalancer por las razones ya mencionadas.



UNIVERSIDAD DE MENDOZA FACULTAD DE INGENIERÍA

Se agregan los nuevos deployments al cluster con ejecutando el bash:

“\$./kubectl-apply.sh -f”.

Cluster:

```
facundo@facundo-desktop:~/Desktop/fincasApp/fincasApp-k8s$ kubectl get all
```

NAME	READY	STATUS	RESTARTS	AGE
pod/fincasappuaa-5656988c97-77724	1/1	Running	0	3m5s
pod/fincasappuaa-mysql-58f8c78c77-gbpwz	1/1	Running	0	3m4s
pod/fincasms-896f98c66-b4zsx	1/1	Running	0	3m7s
pod/fincasms-mysql-75cc5c9d69-xggdn	1/1	Running	0	3m7s
pod/jhipster-registry-0	1/1	Running	0	3m9s
pod/jhipster-registry-1	1/1	Running	0	2m53s
pod/webgateway-5b469fd4d6-kqwmb	1/1	Running	0	3m2s

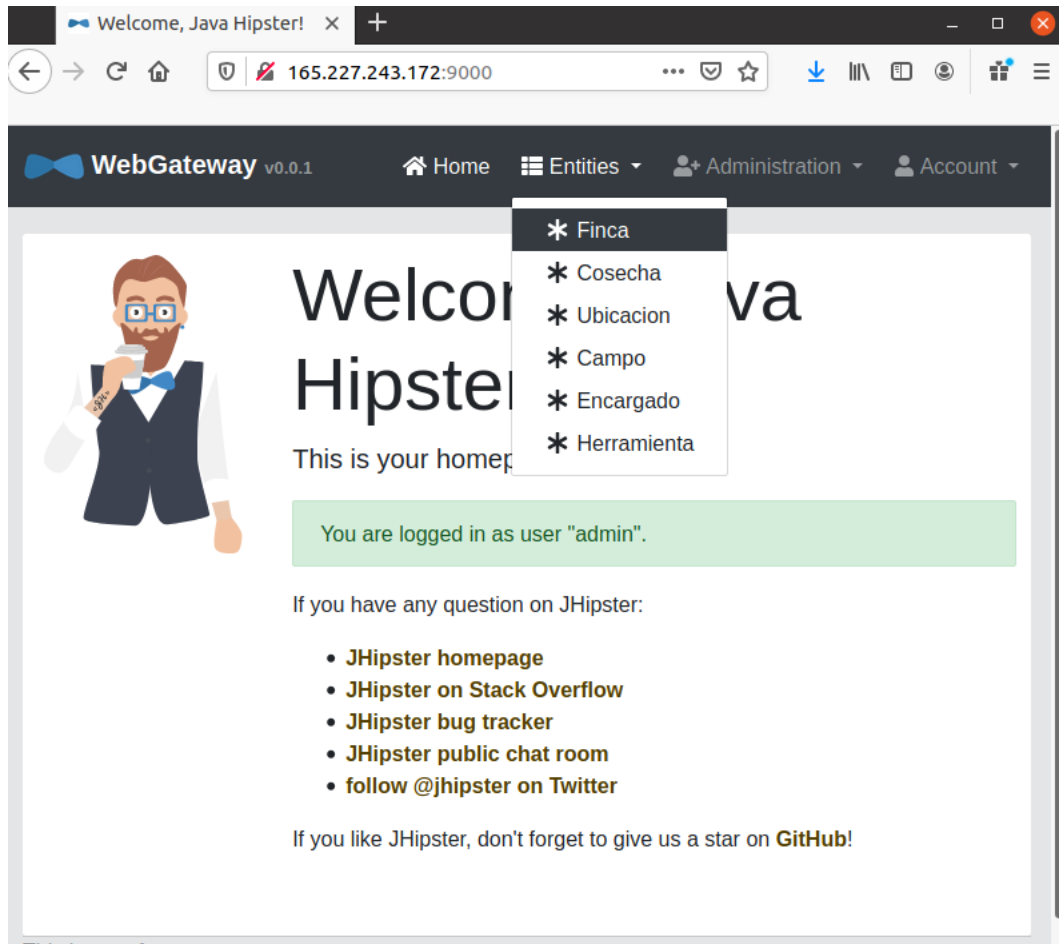
NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
service/fincasappuaa	ClusterIP	10.245.225.186	<none>	80/TCP	3m5s
service/fincasappuaa-mysql	ClusterIP	10.245.245.16	<none>	3306/TCP	3m5s
service/fincasms	ClusterIP	10.245.104.37	<none>	8090/TCP	3m7s
service/fincasms-mysql	ClusterIP	10.245.157.164	<none>	3306/TCP	3m8s
service/jhipster-registry	ClusterIP	None	<none>	8761/TCP	3m10s
service/kubernetes	ClusterIP	10.245.0.1	<none>	443/TCP	16m
service/webgateway	LoadBalancer	10.245.116.97	165.227.243.172	9000:31430/TCP	3m3s

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
deployment.apps/fincasappuaa	1/1	1	1	3m6s
deployment.apps/fincasappuaa-mysql	1/1	1	1	3m5s
deployment.apps/fincasms	1/1	1	1	3m8s
deployment.apps/fincasms-mysql	1/1	1	1	3m8s
deployment.apps/webgateway	1/1	1	1	3m3s



UNIVERSIDAD DE MENDOZA FACULTAD DE INGENIERÍA

Web Gateway:



7. Deployment automáticos

Se agrega un nuevo step en el build (jenkisfile) para que actualice la imagen del deployment cuando se realiza un nuevo build:

```
def pomVersion = readMavenPom().version
stage('Deploy') {
    sh "kubectl set image deployment/webgateway webgateway-app=facu077/webgateway:${pomVersion} --namespace=default"
}
```

8. Creado del entorno TEST (u otros)

Mismo procedimiento que para fincasAppUaa.



4.6. Sprint 4: Desarrollo de funcionalidad “Fincas”

Introducción

En el punto actual ya se cuenta con una arquitectura y una aplicación funcional donde un equipo podría estar trabajando en el creado de los microservicios faltantes (iteraciones tipo 1) y, en paralelo, otro equipo diferente en el desarrollo de funcionalidades de los existentes (iteraciones tipo 2) que a su vez se podría dividir, sin ningún problema, en un equipo que trabaje en el backend (FincasMs) y otro en el frontend (WebGW), también en forma simultánea.

En esta iteración se trabajará en el desarrollo de una de las funcionalidades especificadas en los requerimientos para el microservicio de fincas, por lo que será una iteración del tipo 2. Primero se trabajará el backend y luego el frontend, en forma secuencial. Sin embargo, como se mencionó previamente, esto podría ser trabajado en paralelo.

La función que se busca desarrollar es la de dar a los usuarios la posibilidad de crear fincas y listarlas.

Para el backend se implementarán los siguientes endpoints:

1. Cargar fincas del usuario actual.
2. Crear/Editar finca para el usuario actual.

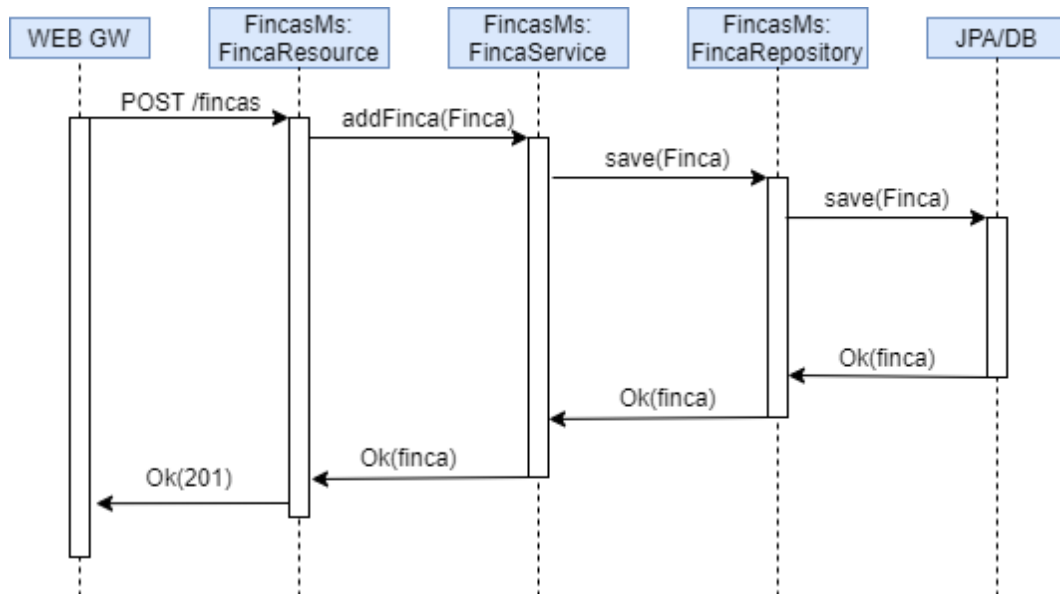
Por otro lado, en el frontend se implementarán las siguientes páginas:

1. Dashboard/front page que muestre bienvenida y todas las fincas del usuario.
2. Página de creación/edición de fincas.

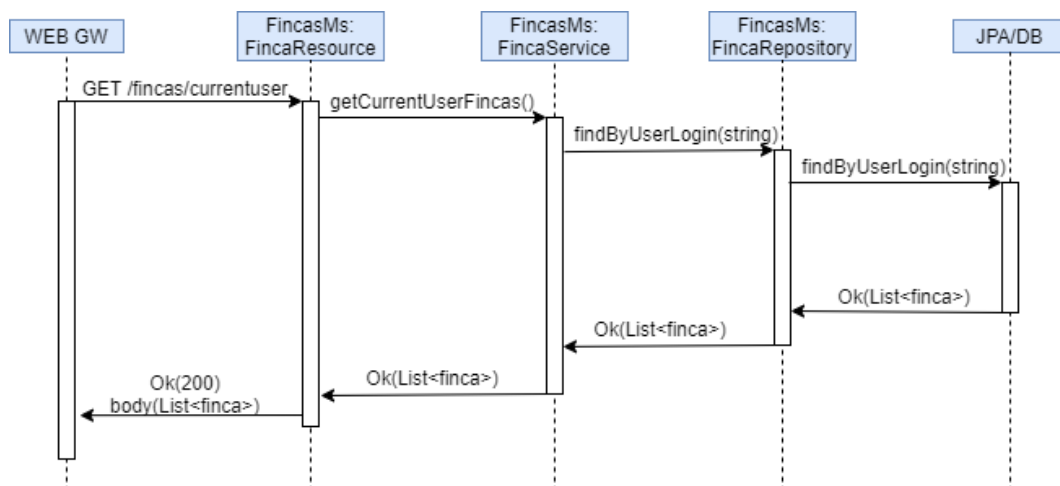
BACKEND (FincasMs)

1. Diagramas

CreateFinca()



GetCurrentUserFincas()



2. Desarrollo de funcionalidad

a. Descarga de repositorio

Primero se selecciona el branch de test (git checkout test) del repositorio de FincasApp-fincasMs y se lo descarga (git pull). Luego

se crea un branch con el número de tarea y el nombre de la funcionalidad a desarrollar.

b. Aplicar cambios de schema de la base de datos (en caso de ser necesario).

En este caso no es necesario. Si así lo fuera se debería modificar el .jdl e importarlo nuevamente al proyecto.

c. Diseño/mock de APIs (Api first development).

POST /api/fincas

Parámetros del Request: (objeto de finca)

```
{
  "campos": [
    {
      "fechaCosecha": "2020-12-30T21:18:56.506Z",
      "fechaPlantado": "2020-12-30T21:18:56.506Z",
      "id": 0,
      "nombre": "string",
      "producto": "string",
      "sembrado": true,
      "tamano": 0
    }
  ],
  "cosechas": [
    {
      "id": 0,
      "peso": 0,
      "producto": "string"
    }
  ],
  "encargado": {
    "apellido": "string",
    "email": "string",
    "id": 0,
    "nombre": "string",
    "telefono": "string"
  },
  "herramientas": [
    {
      "descripcion": "string",
      "id": 0,
      "nombre": "string",
      "tipo": "VEHICULO"
    }
  ],
  "id": 0,
  "nombre": "string",
  "ubicacion": {
    "calle": "string",
    "departamento": "MENDOZA",
    "descripcion": "string",
    "id": 0,
    "numero": 0
  },
  "userLogin": "string"
}
```

Responses:

Código	Descripción
201	Created(finca)
401	Unauthorized
403	Forbidden
404	Not Found

En el controller de finca (en el caso de Jhipster el controller se llama "FincaResource.java") escribimos la API que toma los datos desde un POST y si todo sale bien devuelve la finca creada, que de momento puede ser un mock, en el body de un mensaje HTML y un status 201 en el header. Si falla devolvemos el código de error en el header.

FincaResource.java

```
@PostMapping("/fincas")
public ResponseEntity<Finca> createFinca(@Valid @RequestBody Finca finca) throws URISyntaxException {
    log.debug("REST request to save Finca : {}", finca);
    Finca result = this.fincasService.addFinca(finca);
    return ResponseEntity.created(new URI( str: "/api/fincas/" + result.getId()))
        .headers(HeaderUtil.createEntityCreationAlert(
            applicationName,
            enableTranslation: false,
            ENTITY_NAME,
            result.getId().toString()))
        .body(result);
}
```



UNIVERSIDAD DE MENDOZA FACULTAD DE INGENIERÍA

Prueba con CURL:

```
curl -X POST "http://localhost:8090/api/fincas" -H "accept: */*" -H "Content-Type: application/json" -d "{\"campos\": [{\"fechaCosecha\": \"2020-12-30T21:18:56.506Z\", \"fechaPlantado\": \"2020-12-30T21:18:56.506Z\", \"id\": 0, \"nombre\": \"string\", \"producto\": \"string\", \"sembrado\": true, \"tamano\": 0}], \"cosechas\": [{ \"id\": 0, \"peso\": 0, \"producto\": \"string\" } ], \"encargado\": { \"apellido\": \"string\", \"email\": \"string\", \"id\": 0, \"nombre\": \"string\", \"telefono\": \"string\" }, \"herramientas\": [{ \"descripcion\": \"string\", \"id\": 0, \"nombre\": \"string\", \"tipo\": \"VEHICULO\" } ], \"id\": 0, \"nombre\": \"string\", \"ubicacion\": { \"calle\": \"string\", \"departamento\": \"MENDOZA\", \"descripcion\": \"string\", \"id\": 0, \"número\": 0 }, \"userLogin\": \"string\"}"
```

d. Escribir Unit tests (Test driven development).

Antes de poder escribir los unit test es necesario al menos tener declarada la función que será probada, idealmente en una interfaz que luego será implementada en el service que corresponda. Por lo tanto, el procedimiento será el siguiente:

1. Se crea un `fincasService.java` donde irá toda la business logic y la comunicación entre el controller y el repository.
2. En el service se crea un nuevo método donde irá la lógica encargada de crear una nueva finca para el usuario actual. (aún sin implementar)

```
@Service
@Transactional
public class FincasService {

    private static final String ENTITY_NAME = "fincasMsFinca";

    private final Logger log = LoggerFactory.getLogger(FincasService.class);
    private final FincaRepository fincaRepository;

    public FincasService(FincaRepository fincaRepository) { this.fincaRepository = fincaRepository; }

    public Finca addFinca(Finca finca)
    {
        throw new NotImplementedException();
    }
}
```

3. En una carpeta “test” en la raíz del microservicio y que internamente tenga la misma estructura que este, se crea una nueva clase java “FincasServiceIT” donde escribimos los test de integración que deberá pasar el método para que funcione correctamente.

```
@Test
@Transactional
public void addFinca()
{
    int databaseSizeBeforeCreate = fincaRepository.findAll().size();
    // Create the Finca
    Finca finca = fincasService.addFinca(this.finca);
    // Validate the Finca in the database
    List<Finca> fincaList = fincaRepository.findAll();
    assertThat(fincaList).hasSize(databaseSizeBeforeCreate + 1);
    Finca testFinca = fincaList.get(fincaList.size() - 1);
    // Asserts
    assertThat(testFinca.getNombre()).isEqualTo(DEFAULT_NOMBRE);
    assertThat(testFinca.getUserLogin()).isEqualTo(DEFAULT_USER_LOGIN);
    assertThat(finca).isEqualTo(testFinca);
}
```

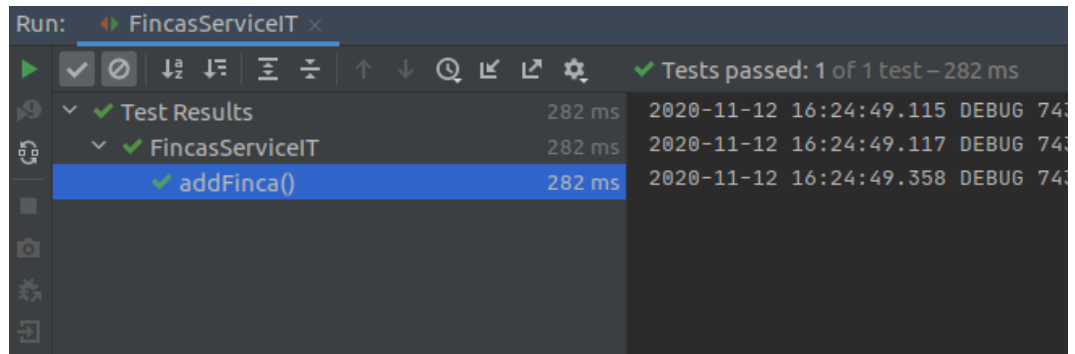
e. Implementar nuevas funciones.

Una vez desarrollado los test, lo próximo es implementar las funciones que pasen esos tests.

En fincasService.java:

```
public Finca addFinca(Finca finca)
{
    String currentLogin = SecurityUtils.getCurrentUserLogin().orElse("");
    if (currentLogin.equals(""))
    {
        throw new AuthorizationServiceException("Unauthorized");
    }
    if (finca.getId() != null)
    {
        throw new BadRequestAlertException("A new finca cannot already have an ID", ENTITY_NAME, "idexists");
    }
    finca.setUserLogin(currentLogin);
    Finca result = fincaRepository.save(finca);
    return result;
}
```

Luego se verifica que la función pase los tests:



f. Subir al repositorio

Finalmente se hace el commit y el push al branch creado para esta funcionalidad para luego hacer el pull request al branch test.

3. Integración continua (proceso automático)

Una vez aceptado el pull request debería activarse de forma automática el build en jenkins del entorno test del microservicio FincasMs que da como resultado una nueva versión de test ejecutándose en el cluster de DigitalOcean.

4. Test

Finalmente se prueba la aplicación con los test cases en el entorno test y, de ser necesario, se realizan las correcciones que hagan falta. Una vez que se hayan pasado todos los tests se puede hacer el merge de test a master, lo que activa el build automático y genera una nueva versión en producción de la aplicación.



UNIVERSIDAD DE MENDOZA FACULTAD DE INGENIERÍA

GetCurrentUserFincas()

Para el desarrollo de GetCurrentUserFincas() el proceso es el mismo, dando como resultado el siguiente código:

GET /api/fincas/currentuser

Parámetros del Request: vacío

Responses:

Código	Descripción
200	Ok(finca ²² [])
401	Unauthorized
403	Forbidden
404	Not Found

Prueba con CURL:

```
curl -X GET "https://localhost:8090/api/fincas/currentuser" -H "accept: */*"
```

FincasResource:

```
@GetMapping("/fincas/currentuser")
public ResponseEntity<List<Finca>> getCurrentUserFincas() {
    log.debug("REST request to get all of currentUser Fincas");
    List<Finca> fincas = this.fincasService.getCurrentUserFincas();
    return ResponseEntity.ok().body(fincas);
}
```

²² Arreglo del objeto "finca". Se puede ver el objeto completo en la definición de la API de POST /api/finca



UNIVERSIDAD DE MENDOZA FACULTAD DE INGENIERÍA

FincasServiceIT:

```
@Test
@Transactional
public void getCurrentUserFincas()
{
    // Initialize the database and add some fincas
    fincaRepository.flush();
    Finca fincaOne = new Finca()
        .nombre("Don Miguel")
        .userLogin("user");
    fincaRepository.save(fincaOne);
    Finca fincaTwo = new Finca()
        .nombre("El Rancho")
        .userLogin("user");
    fincaRepository.save(fincaTwo);
    Finca fincaThree = new Finca()
        .nombre("El Ramal")
        .userLogin("OtherUser");
    fincaRepository.save(fincaThree);
    // Get the fincas for current user ("user")
    List<Finca> fincas = fincasService.getCurrentUserFincas();
    // Asserts
    assertThat(fincas.size()).isEqualTo(2);
    assertThat(fincas.contains(fincaOne)).isTrue();
    assertThat(fincas.contains(fincaTwo)).isTrue();
    assertThat(fincas.contains(fincaThree)).isFalse();
}
```

FincasService:

```
public List<Finca> getCurrentUserFincas()
{
    String currentLogin = SecurityUtils.getCurrentUserLogin().orElse("other: ");
    if (currentLogin.equals(""))
    {
        throw new AuthorizationServiceException("Unauthorized");
    }
    List<Finca> page = fincaRepository.findByUserLogin(currentLogin);
    return page;
}
```



UNIVERSIDAD DE MENDOZA FACULTAD DE INGENIERÍA

Las conexiones a la base de datos se manejan con JPA, por lo que además en el repository de finca se agregó un método para buscar por login de usuario:

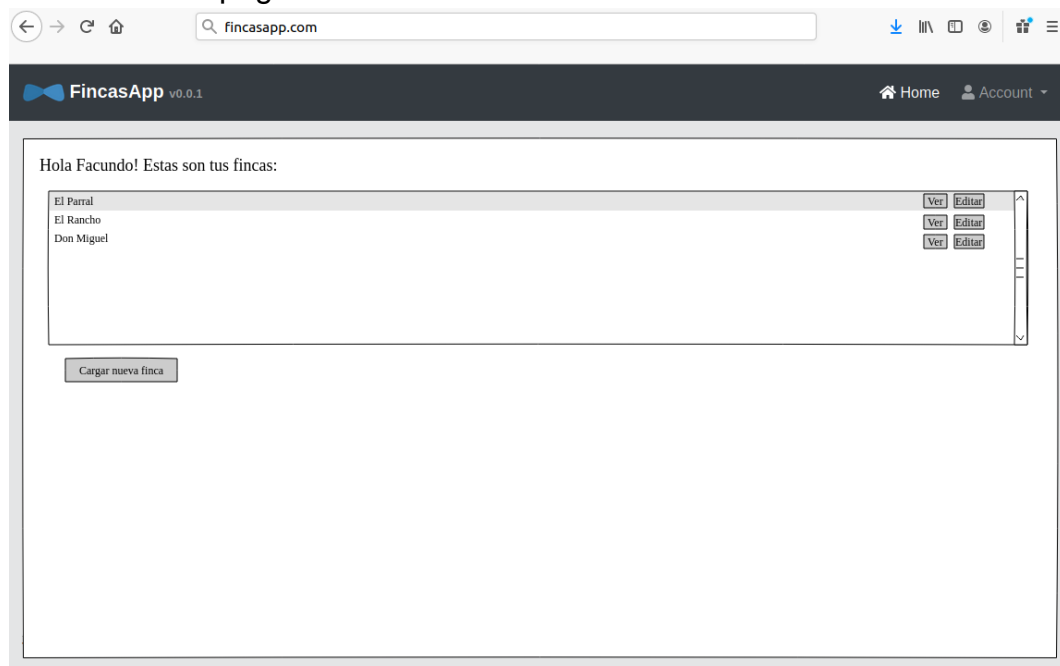
```
@Repository
public interface FincaRepository extends JpaRepository<Finca, Long> {
    List<Finca> findByUserLogin(String userLogin);
}
```

Tanto como en `GetCurrentUserFincas()` y `CreateFinca()` por seguridad *no* se envía el login/id desde el frontend. Siempre se utiliza el usuario actual que se obtiene desde el UAA service.

FRONTEND (WebGW)

1. Diagramas

Mock del front page:





UNIVERSIDAD DE MENDOZA FACULTAD DE INGENIERÍA

Mock de la creación/modificación de fincas:

← → ↻ 🏠 🔍 fincasapp.com ⬇️ 📄 📱 🌐 🛒 ☰

FincasApp v0.0.1 Home Account ▾

Crear o modificar Finca

Nombre

Cancel Save

2. Desarrollo de funcionalidad

a. Descarga de repositorio

Al igual que como se hizo con la parte del backend primero se descarga el branch test del repositorio del WebGw y luego se crea un nuevo branch en el que se desarrollará la funcionalidad.

b. Aplicar cambios de schema (o componentes para frontend)

Al trabajar con angular el equivalente al schema del backend vendrían a ser los componentes que son:

- Service - Encargado de la comunicación con los microservicios mediante API REST
- Component.ts - Encargado de manejar toda la lógica de la interacción con el usuario
- Component.html - UI
- Component.spec.ts - Unit tests
- Module - Donde se declaran todos los componentes
- Route - Donde se declaran las URL de acceso a los componentes



UNIVERSIDAD DE MENDOZA FACULTAD DE INGENIERÍA

El orden y la estructura de estos componentes pueden variar según la aplicación y las preferencias en el estilo de estructura. En este caso se utiliza la estructura propuesta por Jhipster donde por cada entidad se genera una carpeta con todos estos archivos.

c. Diseño/mock de API's

En este punto, debido a la estructura secuencial del trabajo, las API ya han sido implementadas, pero con la definición de las API que se hizo previamente ya se podría haber empezado a trabajar aun cuando estas no hubiesen estado implementadas.

Como se explicó previamente, para lograr la comunicación con los diferentes microservicios Jhipster utiliza Jhipster Registry y un proxy en los gateways. Cada vez que un microservicio se registra en el registry, el o los gateways pueden acceder a estos mediante la ruta: <url-del-server>/services/<nombre-del-microservicio>/api/<entidad>. Por ejemplo, para acceder a la API definida previamente para hacer un GET de las fincas del usuario sería: <http://localhost:9000/services/fincasms/api/fincas/currentuser>. Donde la URL del servidor es localhost si se trabaja local y sino se inyecta en una variable de entorno en el startup de la aplicación.

Por lo tanto en finca.service.ts del frontend vamos a tener un método que pida todas las fincas desde el fincasMs:

```
export class FinsaService {
  public resourceUrl = SERVER_API_URL + 'services/fincasms/api/fincas';

  constructor(protected http: HttpClient) {}

  getCurrentUserFincas(): Observable<EntityArrayResponseType>
  {
    return this.http.get<IFinca[]>(`${this.resourceUrl}/currentuser`, { observe: 'response' });
  }
}
```

d. Escribir Unit tests

Se escriben los unit test que prueben las funcionalidades en finca.service.spec.ts y home.component.spec.ts.

e. Implementar nuevas funciones

Para cargar las fincas del usuario se llamará al método creado en el finca.service.ts en el init del home.component.ts (si hay algún usuario logeado) y cada vez que cambie el usuario:



UNIVERSIDAD DE MENDOZA FACULTAD DE INGENIERÍA

home.component.ts:

```
ngOnInit(): void {  
  this.authSubscription = this.accountService.getAuthenticationState().subscribe(account =>  
    {  
      this.account = account;  
      if (this.account)  
      {  
        this.loadCurrentUserFincas();  
      }  
    });  
  if (this.account)  
  {  
    this.loadCurrentUserFincas();  
  }  
}
```

loadCurrentUserFincas():

```
loadCurrentUserFincas(): void  
{  
  this.fincaService  
    .getCurrentUserFincas()  
    .subscribe(  
      (res: HttpResponse<IFinca[]>) => {this.fincas = res.body}  
    );  
}
```

Finalmente se dibuja la tabla en la UI utilizando bootstrap.

home.component.html:

```
<h3 class="display-6">Hola, {{account?.login}} estas son tus fincas: </h3>
<div>
  <table class="table" *ngIf="fincas">
    <thead>
      <tr>
        <th>#</th>
        <th>Nombre</th>
      </tr>
    </thead>
    <tbody>
      <tr *ngFor="let finca of fincas; let i = index">
        <td>{{i+1}}</td>
        <td>{{finca.nombre}}</td>
        <td class="text-right">
          <button type="button" class="btn btn-info">Editar</button>
          <button type="button" class="btn btn-danger">Borrar</button>
        </td>
      </tr>
    </tbody>
  </table>
  <button type="button" class="btn btn-primary" [routerLink]="['/finca/new']">
    Crear nueva finca
  </button>
</div>
```

web gateway:



WebGateway vDEV

dev

Hola, facundo estas son tus fincas:

#	Nombre	
1	Don Miguel	Editar Borrar
2	El Rancho	Editar Borrar

Crear nueva finca

f. Subir al repositorio



UNIVERSIDAD DE MENDOZA FACULTAD DE INGENIERÍA

Se suben los cambios locales al branch creado para esta funcionalidad y se realiza el pull request a test.

3. Integración continua

Mismo proceso que para la parte del backend.

4. Test

Mismo proceso que para la parte del backend.

Para la función de crear nuevas fincas se realiza el mismo procedimiento.



4.7. Sprint n: Futuras iteraciones

Para seguir con el desarrollo de la aplicación el proceso sería similar a los Sprints anteriores, teniendo en cuenta que se trabajará en una iteración del tipo 1 si se quiere integrar un nuevo microservicio, o del tipo 2 si se quiere trabajar/modificar uno existente. La mayor diferencia entre los Sprints va a estar en la programación que depende de la funcionalidad o microservicio que se quiere desarrollar, pero el proceso en su conjunto debería ser siempre el mismo.

Por otro lado, por motivos de simplicidad, dentro de los sprints realizados en este trabajo siempre se trabajó sobre una única iteración dentro de cada sprint, sin embargo, lo lógico sería que, al trabajar en un equipo, se puedan trabajar en múltiples funcionales dentro de un mismo sprint, aprovechando uno de los beneficios de trabajar con microservicios.



CAPÍTULO 5: CONCLUSIONES

5.1. ¿Se cumplió con “Beyond The Twelve-Factor App”?

A continuación, se analiza cada uno de los *Factors App* descritos por “Beyond The Twelve-Factor App” y se los compara con el trabajo realizado para analizar si se cumplió con cada uno de ellos:

I. Un código base

Este punto se cumple ya que al utilizar git y github (como herramienta cloud) y contando con una única raíz para los repositorios se logra tener todo el código en un mismo lugar y realizar un control de versiones sobre este.

Nota: por motivos de simplicidad, se utilizó una cuenta particular de github (<https://github.com/facu077/FincasApp-<microservicio x>>), siendo la raíz “facu077/FincasApp”. Sin embargo, en un caso real se utilizaría una cuenta administrativa con el nombre del software o de la empresa, por ejemplo: <https://github.com/fincasapp/<microservicio x>> quedando reservada únicamente para los repositorios de la aplicación.

II. API first

En el desarrollo del backend del [Sprint 4 \(Desarrollo de funcionalidad “Fincas”\)](#) se puede observar una aplicación de esta técnica que debe ser tomada en cuenta en el desarrollo de las subsecuentes funcionalidades.

El proceso que permitió cumplir este punto fue:

1. Se realizó el diagrama de flujo que permite tener una idea básica de cómo sería esta API.
2. Se definió la URL y que método de HTML se utilizará.
3. Se definieron los parámetros, en formato JSON, que serían utilizados en el Request y en el Response de la llamada, así como también los diferentes tipos de código de la respuesta.
4. Se realizó una definición de la API en el código, aún sin implementar, con el fin de que ya se pueda probar.
5. Se probó la API con CURL.

III. Manejo de dependencias

Para el manejo de las dependencias se utilizó Maven para el backend, como por ejemplo se puede observar en la configuración del plugin de JIB



UNIVERSIDAD DE MENDOZA FACULTAD DE INGENIERÍA

en diferentes sprints en el [capítulo 4](#). Y para el Frontend se utiliza Node Package Manager. Ambas herramientas se encargan de suministrar todas las dependencias que hayan sido declaradas como necesarias para la aplicación.

IV. Diseñar, construir, desplegar, ejecutar

Si se observa la iteración propuesta definida como “[Iteración del tipo 1](#)” en el capítulo 3 (y sus implementaciones en el capítulo 4), que es en donde se arman nuevos microservicios y donde también se construye el proceso de CI/CD para cada uno de estos, se puede observar que, en esencia, es una secuencia de pasos similar a la propuesta por este factor app:

1. Se realizan los diseños necesarios (*Diseño*)
2. Se desarrolla el microservicio (pasos intermedios que no están directamente relacionado con este factor app)
3. Luego se arma un pipeline en Jenkins que:
 - a. Toma el código del repositorio de Github y lo compila (*Build*).
 - b. Se sube la salida del build (docker image) a Docker Hub con un nombre único (número de versión y entorno) que permita su identificación (*Deployment*).
 - c. Se le indica al cluster de Kubernetes que reemplace la imagen de su deployment con la nueva según cual sea el entorno (*Ejecución*)

V. Configuraciones, credenciales y código

El cumplimiento de este factor app se puede observar a la hora de configurar Jenkins, donde las credenciales de Github y Docker Hub son almacenadas como variables de entorno que luego se utilizan durante el build (ej: [Sprint 1](#) - Paso 4: Archivos de configuración del Pipeline).

Las configuraciones de acceso a las bases de datos, entre otras, son manejadas internamente por JHipster, pero en caso de hacerlo manual el proceso es similar: Se define una variable de entorno con el connection string a la base de datos y luego se la inyecta durante el build.

VI. Logs

Como se puede observar en la implementación de los métodos del microservicio “fincasMs” en el [sprint 4](#) en caso de producirse un error se arroja una excepción según cual sea el tipo de error (unauthorized, BadRequest, etc), la salida de estos errores es capturada por el JHipster Registry, ya que todos los microservicios están registrados en este



UNIVERSIDAD DE MENDOZA FACULTAD DE INGENIERÍA

servicio. Luego en el dashboard del registry se pueden visualizar todos estos errores (logs) junto con otra información y métricas.

De igual manera el cluster de kubernetes es monitoreado por DigitalOcean y permite acceder a los logs de errores ante cualquier fallo.

VII. Desechabilidad

Al trabajar con kubernetes y una arquitectura de microservicios se logra cumplir con este punto, ya que en cualquier momento se pueden lanzar más réplicas de un microservicio, deployar una nueva versión en un pod y eliminar cualquier otro que esté fallando o contenga una versión vieja.

VIII. Backing services

Mediante el “Jhipster Registry” es posible cambiar cualquiera de los backing services (ej: OAuth2 Server) en cualquier momento e incluso registrar nuevos que podrían llegar a ser necesarios para los microservicios. Y en caso de que falle el “Jhipster Registry” los microservicios entrarán en un loop intentando conectarse a este hasta que sea posible.

IX. Paridad de entornos

Mediante el proceso de CI/CD y pipelines planteados en la metodología se logra cumplir con este punto sin problema. Además, al tener un namespace definido dentro del cluster de kubernetes para cada entorno, se pueden analizar rápidamente los recursos, versiones y servicios corriendo dentro de cada entorno.

X. Procesos administrativos

En ningún momento se planteó ningún proceso administrativo y como se pudo ver los dos tipos de iteraciones planteadas permiten la flexibilidad suficiente para trabajar en los diferentes aspectos como podría ser crear y/o modificar microservicios y trabajar tanto en backend como frontend. Aspectos como migraciones de base de datos escapan del alcance de esta tesis.

XI. Asignación de puertos

Al tener un pod por cada microservicio y un loadbalancer por cada gateway que se quiera exponer a internet no hay problemas con la asignación de puertos, por lo que todos los servidores web podrían tener el puerto 80 o el 8080 que son los que se utilizan por defecto para este



UNIVERSIDAD DE MENDOZA FACULTAD DE INGENIERÍA

fin. Sin embargo, a cada microservicio se le asignó puertos diferentes (fincasMs: 8090, webGw: 9000) para que, en caso de trabajar en un entorno local, por ejemplo, mediante docker compose, no haya conflictos entre los puertos, ya que en esos casos la IP es la misma.

XII. Procesos sin estado

Todos los procesos que se han utilizado son procesos sin estados. Por ejemplo, en la llamada para obtener las fincas del usuario actual o crear una finca, donde se deben conocer los datos del usuario, se obtiene de una llamada al “FincasAppUaa” y luego, en el caso de crear una finca, se almacena este dato en la base de datos, pero nunca en el “FincasMs”.

XIII. Concurrencia

Como ya se explicó, mediante el cluster de kubernetes es posible crear tantas réplicas como sea necesaria de un único microservicio según la demanda que tenga, así como también eliminar réplicas cuando haya baja demanda.

XIV. Telemetría

Este punto se logra gracias al monitoreo del cluster que se realiza desde DigitalOcean y así como también al monitorio del “JHipster Registry”.

XV. Autenticación y autorización

Mediante el FincasAppUaa, que cuenta con el servidor de OAuth2, se controla el acceso a todos los microservicios y los roles. Y a su vez los endpoints verifican que exista un usuario actual y que tenga permiso para realizar determinada acción, en caso negativo se arroja una excepción del tipo “Unauthorized” y se devuelve un 401 como respuesta a la llamada.



5.2. ¿Se cumplieron los objetivos?

De igual manera que con el punto anterior, se analizará cómo se logró cumplir con cada objetivo del trabajo.

Objetivo General: Definir y aplicar una metodología de desarrollo de software que aplique prácticas devops y “beyond the twelve-factor apps” con el fin de realizar ci/cd en kubernetes.

El cumplimiento de este objetivo se puede ver reflejado en [5.1](#) y en el cumplimiento de los objetivos específicos.

Objetivos Específicos

- Definir cada uno de los elementos que componen la metodología.

Se logró identificar y explicar la aplicación de cada uno de los principales elementos que componen la metodología que permite cumplir con prácticas devops, los beyonds the twelve factor app, reglas de buenas prácticas generales y deployments en kubernetes.

Los elementos definidos que permitieron el logro de este objetivo fueron:

1. Modo de trabajo - Scrum y buenas prácticas.
2. Arquitectura del software - Microservices (BFF), Docker, Kubernetes.
3. Entorno de desarrollo - Entorno Local y entorno Cloud.
4. Proceso de CI/CD - Integración continua y deployments automáticos en kubernetes.
5. Proceso iterativo e incremental - Adaptable a un sprint.

- Encontrar un proceso iterativo e incremental que permita trabajar en esa metodología.

En la [última parte del capítulo 3](#) se lograron definir dos tipos de iteraciones, una para incluir nuevos microservicios, y la otra para trabajar sobre existentes y en ambos casos utilizando la metodología planteada y realizables dentro de un Sprint. También, en caso de trabajar en equipos, contemplan la posibilidad de realizar múltiples iteraciones en paralelo dentro de un mismo sprint, aprovechando uno de los beneficios de microservicios.



UNIVERSIDAD DE MENDOZA FACULTAD DE INGENIERÍA

- Aplicar la metodología y el proceso iterativo e incremental a un ejemplo concreto.

En paralelo a la definición de la metodología se la fue aplicando al desarrollo del software de ejemplo “FincasApp” y finalmente en el [capítulo 4](#) se le aplicaron los diferentes tipos de iteraciones. Si bien sólo se desarrolló una pequeña parte de “FincasApp”, ya que no era el objetivo de este trabajo, se desarrolló lo suficiente para poder ejemplificar los procesos planteados en este trabajo.

5.3. Conclusiones finales

Como se puede observar en los puntos anteriores se logró cumplir con los objetivos lo cual permite resolver los [problemas](#) que se habían planteado inicialmente. Esto es posible gracias a la metodología desarrollada a lo largo del trabajo que, por un lado, propone una manera de trabajar con microservicios y, por el otro, una forma de integrar este tipo de arquitectura a un proceso de integración continua con deployments en kubernetes. A su vez, mediante el desarrollo de la aplicación de ejemplo “FincasApp” ha quedado demostrado que es posible trabajar con esta metodología.

La metodología planteada debe ser considerada como una propuesta, ya que existen infinitas formas de desarrollar software y habrá algunas que se adopten mejor que otras a las necesidades de cierto producto. Sin embargo, se considera que es una buena base ya que, considera buenas prácticas como “Beyond the twelve factor app”, prácticas DevOps y Test Driven Development y se aplican mediante el uso de tecnologías modernas como son los microservicios, dockers, kubernetes y los pipelines de integración continua.

Por otro lado, como se pudo observar en la creación de nuevos microservicios, el proceso se hace un poco más complejo que en una aplicación monolítica. También se hace más pesado correr toda la aplicación en un entorno local ya que, se deben levantar múltiples aplicaciones/microservicios en simultáneo. Sin embargo, estas desventajas se ven contrarrestadas por las ventajas que permiten realizar los deployments en kubernetes de una manera muy simple, monitorear la aplicación y, de ser necesario, escalar de manera horizontal cuanto haga falta. También es posible llevarse toda la aplicación a cualquier otro proveedor de servicios cloud ya que, por la forma en que se manejan las imágenes y el versionado en Docker Hub (que a su vez podría ser



UNIVERSIDAD DE MENDOZA FACULTAD DE INGENIERÍA

reemplazado por cualquier otro docker registry) es posible realizar el deployment en cualquier otro cluster de kubernetes.



UNIVERSIDAD DE MENDOZA FACULTAD DE INGENIERÍA

BIBLIOGRAFÍA

- SAM NEWMAN - (2015) "Building Microservices"
- Sitio WEB de SAM NEWMAN - <https://samnewman.io/>
- THE BACKEND FOR FRONT END PATTERN
(https://philcalcado.com/2015/09/18/the_back_end_for_front_end_pattern_bff.html)
- MICROSOFT - documentacion de .NET,
(<https://docs.microsoft.com/en-us/dotnet/>)
- THE TWELVE-FACTOR APP - documentación oficial
(<https://www.12factor.net/>)
- KEVIN HOFFMAN - (2016) "Beyond The Twelve-Factor App"
- JHipster - documentación oficial - <https://www.jhipster.tech/>.
- Spring Boot - documentación oficial -
<https://spring.io/projects/spring-boot>.
- How To Set Up Continuous Integration Pipelines in Jenkins -
<https://www.digitalocean.com/community/tutorials/how-to-set-up-continuous-integration-pipelines-in-jenkins-on-ubuntu-16-04>.
- Kubernetes - documentación oficial -
<https://kubernetes.io/docs/home/>.