UNIVERSITAT POLITÈCNICA DE VALÈNCIA

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Politécnica Superior de Gandia

Sistema de gestión de traducciones de textos para la internacionalización de aplicaciones

Trabajo Fin de Grado

Grado en Tecnologías Interactivas

AUTOR/A: Esteve Peiro, Emilio

Tutor/a: Bataller Mascarell, Jordi

CURSO ACADÉMICO: 2022/2023

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

ESCOLA POLITÈCNICA SUPERIOR DE GANDIA

Grado en Tecnologías Interactivas

“Sistema de gestión de traducciones de textos para la internacionalización de aplicaciones”

TRABAJO FINAL DE GRADO

Autor/a:
Emilio Esteve Peiró

Tutor/a:
Jordi Bataller Mascarell

GANDÍA, 2023

1. Introducción..... 4
1.1 Presentación y objetivos..... 4
2. Entorno y estado del arte..... 6
2.1. Lokalise..... 7
2.2. Localazy..... 8
3. Análisis de requerimientos..... 9
3.1. Requisitos a nivel de API..... 9
3.2. Requisitos a nivel de infraestructura y desarrollo..... 10
4. Diseño de la aplicación..... 11
4.1. Diseño de entidades en base de datos..... 11
4.2. Diseño del servidor..... 12
4.3. Lógica de negocio..... 15
4.4. Diseño de la integración contínua del código..... 16
Formato..... 16
Estructura..... 16
Pruebas unitarias..... 17
4.5. Diseño de los despliegues continuos del servicio..... 18
Las modificaciones en base de datos que los desarrolladores han implementado en su código se lanzan una vez se despliegue el servicio..... 18
Diseño de la monitorización del servicio..... 19
5. Implementación..... 20
5.1. Herramientas de desarrollo..... 20
Ubuntu..... 20
NodeJS..... 20
Docker..... 20
Entorno de desarrollo integrado..... 20
Cliente de base de datos..... 21
Postman..... 21
Git..... 21
Fork..... 22
5.2. Estructura de la aplicación..... 23
5.3. Implementación del servicio..... 24
Aplicación..... 26
Dominio..... 26
Infraestructura..... 26
5.4. Implementación de la integración contínua y de los despliegues contínuos..... 28
Integración contínua..... 28
Despliegues contínuos..... 30
5.6. Problemas de implementación resueltos..... 33
Problemas con la programación del servicio..... 33
Problemas con la integración contínua y los despliegues contínuos..... 34
6. Manuales..... 35
6.1. Manual de instalación..... 35
6.2. Guía de uso..... 39

8. Conclusiones.....42
8.1. Evaluación.....43
8.1. Trabajo futuro.....44
9. Código.....45
10. Referencias de imágenes.....46
11. Bibliografía.....47

1. Introducción

1.1 Presentación y objetivos

Es evidente que, hoy en día, toda aplicación, videojuego o sistema informático debe estar dirigida a un público internacional y por ello, sus textos deben de aparecer en distintos lenguajes. Para facilitar estas tareas, es muy conveniente disponer de un sistema de traducciones automático. De esta forma, mediante la modificación del parámetro correspondiente en la configuración de la aplicación se conseguirá que los textos se muestren en el idioma seleccionado.

Las empresas encargadas de desarrollar este tipo de aplicaciones necesitan tener centralizadas las traducciones para que sus empleados las puedan consultar y modificar para poder insertarlas posteriormente en las aplicaciones que los programadores están desarrollando.

El objetivo principal que nos hemos propuesto en este proyecto es crear un sistema para la gestión de estas traducciones. Para ello, vamos a tener que resolver las siguientes cuestiones:

- Definir un formato que permita disponer de una estructura que facilite su manipulación y transmisión.
- Garantizar la seguridad de los usuarios de la aplicación soportando autenticación y autorización.
- Guardar un registro de las acciones que los usuarios realicen para permitir un posterior análisis o verificación.
- Permitir la importación y exportación de traducciones desde un formato neutral al que se utilice internamente.
- Posibilitar el despliegue de este servicio en servidores en la nube.

También nos hemos propuesto objetivos a nivel académico y profesional:

- Aprender sobre servicios de computación en la nube.
- Estudiar el despliegue de servicios mediante Google Cloud Platform (“Google Cloud Platform”).
- Instruirse en la contenerización de servicios mediante las tecnologías Docker (“Docker”) y Kubernetes (“Kubernetes”).
- Saber escalar y desescalar los servicios contenerizados en base a el uso que se esté ejerciendo de ellos.
- Averiguar cómo monitorizar los servicios para tener conocimiento sobre lo que está ocurriendo en ellos.
- Investigar la metodología DevOps (“DevOps”) para utilizarla convenientemente.
- Informarnos sobre buenas prácticas en la integración y entrega contínua de código y aplicarlo en el proyecto.
- Aplicar lo aprendido en nuestro entorno de trabajo.
- Transicionar de un puesto profesional de ingeniero backend a uno híbrido entre el anterior e ingeniero DevOps.

2. Entorno y estado del arte

La internacionalización de aplicaciones, juegos y cualquier tipo de servicio en internet es tanto responsabilidad de los programadores como de los diseñadores y traductores.

Por lo tanto es necesaria una plataforma a través de la que cada uno de estos roles puedan interaccionar de forma sencilla.

En una empresa lo suficientemente grande para tener estos roles contratados, la interacción ideal sería: el diseñador sube a la plataforma los textos que necesita traducir, los traductores traducen los textos y los programadores los utilizan en el servicio.

Existen ya varias aplicaciones para internacionalizar servicios, aunque todas ellas son excesivamente caras para lo que puede costar desarrollar y mantener el tipo de producto que nosotros podemos llegar a necesitar.

Nosotros hemos desarrollado este servicio para que cualquier empresa pueda tenerlo totalmente gratuito y desplegarlo en una plataforma cloud, ya que el código del despliegue es fácilmente manipulable y así puedan tenerlo en su propia infraestructura.

Esto permitirá que las empresas escalen sus costes tan solo cuando lo necesiten y paguen una parte muchísimo menor a lo que se les pide en otros servicios.

Además, el poder elegir dónde alojar el servicio les ayuda a mantener un nivel de latencia muy bajo si escogen bien, ya que pueden ponerlo tan cerca de los usuarios como se requiera.

Algunos de los servicios del sector más utilizados por empresas son:
- Lokalise (“Lokalise”)
- Localazy (“Localazy”)

2.1. Lokalise

Este servicio nos ofrece una librería para cada uno de los lenguajes más utilizados para desarrollar aplicaciones web y móviles. De esta forma el uso de su API se nos hará mucho más fácil de integrar en nuestro código y no tendremos que estar tan pendientes de la documentación, tanto para el desarrollo inicial como para su mantenimiento.

Tienen varios planes de precios, el que a nosotros nos interesa ver es el primero, ya que con el precio de este ya podríamos observar que la plataforma es bastante cara para lo que propone.

El pack “Starter” cuesta 140$ mensuales y tiene las siguientes funcionalidades:
- Proyectos ilimitados.
- Máximo 10 usuarios incluyendo el administrador.
- Editor de traducciones colaborativo.
- Posibilidad de asignación de tareas para los usuarios.
- Máximo 5000 claves de traducción.

De los requisitos anteriores lo que más costaría de mantener serían las 5000 claves de traducción, ya que serían 5000 elementos en una tabla o colección de una base de datos. Aunque realmente no es nada extraordinario, es más son muy pocos elementos, no es nada preocupante de almacenar.

Tan solo con la cuota del primer mes podríamos mantener el servicio durante aproximadamente un año.

Algunas de las empresas más grandes que usan este servicio son:
- Revolut
- Starbucks
- Hyundai
- Mastercard

2.2. Localazy

En cuanto a este servicio, contratando el pack “Team” (proporcional al elegido anteriormente en Lokalise) también tendremos acceso a SDKs y a su API para poder integrarnos fácilmente, como hemos explicado en el apartado anterior.

En este producto el límite de traducciones se cuentan por el número de palabras en total que tienen las claves a traducir.

Si comparamos este método de restricción con el del servicio anterior, observamos que si en el anterior la clave tuviera 20 palabras en su interior, llegaríamos al límite de 100.000 palabras soportadas por el plan, ya que 100.000 dividido entre 5.000 es igual a 20.

Las claves a traducir suele ser el texto literal que queremos que los traductores vean para que así puedan hacer su trabajo, por lo que no es de extrañar que nos encontremos con una media de 20 palabras por traducción.

Ante esto, podemos decir que la plataforma es de un precio bastante similar al de Localise, lo que nos llevaría a pensar exactamente lo mismo, que si nos ofrecen un sistema que podamos administrar nosotros mismos, no merecería la pena contratarla.

Algunas de las empresas más grandes que utilizan este servicio son:
- Github
- Gitlab
- Discord

3. Análisis de requerimientos

En este capítulo nos centraremos en los requerimientos del servicio tanto a nivel de API como a nivel de infraestructura y desarrollo, ya que será lo que desarrollaremos en este trabajo.

3.1. Requisitos a nivel de API

Al tratarse de un sistema de organización de traducciones, necesitaremos que cada empresa pueda registrarse en nuestro servicio.

La empresa deberá poder registrar a sus trabajadores, para que puedan tratar las traducciones que sean necesarias. Tan solo el usuario administrador podrá dar de alta a estos usuarios.

Con el objetivo de que se puedan organizar mejor, cada empresa podrá dividir sus traducciones en contextos, de esta forma los usuarios podrán filtrar las traducciones por un contexto particular en el cual estén especializados.

Cualquier acción llevada a cabo que afecte a una empresa o usuario registrado sólo podrá ser llevada por un usuario autenticado y autorizado. Lo que significa que el usuario deberá iniciar sesión en nuestro sistema y tener el permiso preciso para poder realizar la acción.

Los usuarios deberán poder descargar las traducciones de una empresa y un contexto determinado en un formato estandarizado. Además, estos podrán insertarlas y actualizarlas de forma masiva.

Ante cualquier incidente, el equipo técnico deberá ser capaz de obtener las transacciones llevadas a cabo por un usuario en las últimas dos semanas.

3.2. Requisitos a nivel de infraestructura y desarrollo

Una de las necesidades de cualquier servicio cuando se despliega en una infraestructura es que debe ser escalable, es decir, en el momento en el que empiecen a llegar muchas peticiones, este servicio deberá aumentar sus recursos para poder soportar el tráfico que le llega sin caerse.

En cuanto al desarrollo, al ser un código libre, que cualquiera puede acceder a él y descargarlo, tendremos que mantener un histórico de todas las funcionalidades que se van haciendo para que los demás programadores tengan constancia de lo que se ha implementado sin tener que acceder a ningún panel administrativo.

Además, este histórico tan solo deberá actualizarse una vez se publique una nueva versión del servicio; mientras no se haga una mezcla de código o una publicación, éste no se podrá al día, ya que si la funcionalidad finalmente se descartara sería confuso para los demás.

Toda publicación del servicio deberá ser automática, no tendremos que ejecutar manualmente ningún proceso que requiera de nuestra atención, ya que si no se automatiza se tiende a cometer errores y estas equivocaciones durante un proceso de este tipo suelen ser costosas de solventar.

4. Diseño de la aplicación

4.1. Diseño de entidades en base de datos

Todos los identificadores usados en esta aplicación son de tipo UUID versión 4, ya que es casi imposible que se repita, al menos en muchísimo tiempo.

Tendremos una entidad base (BaseEntity) de la cual extenderán las demás.

Ésta tendrá un identificador principal, una fecha de creación y una de actualización, la cuál deberá modificarse cada vez que la entidad varíe.

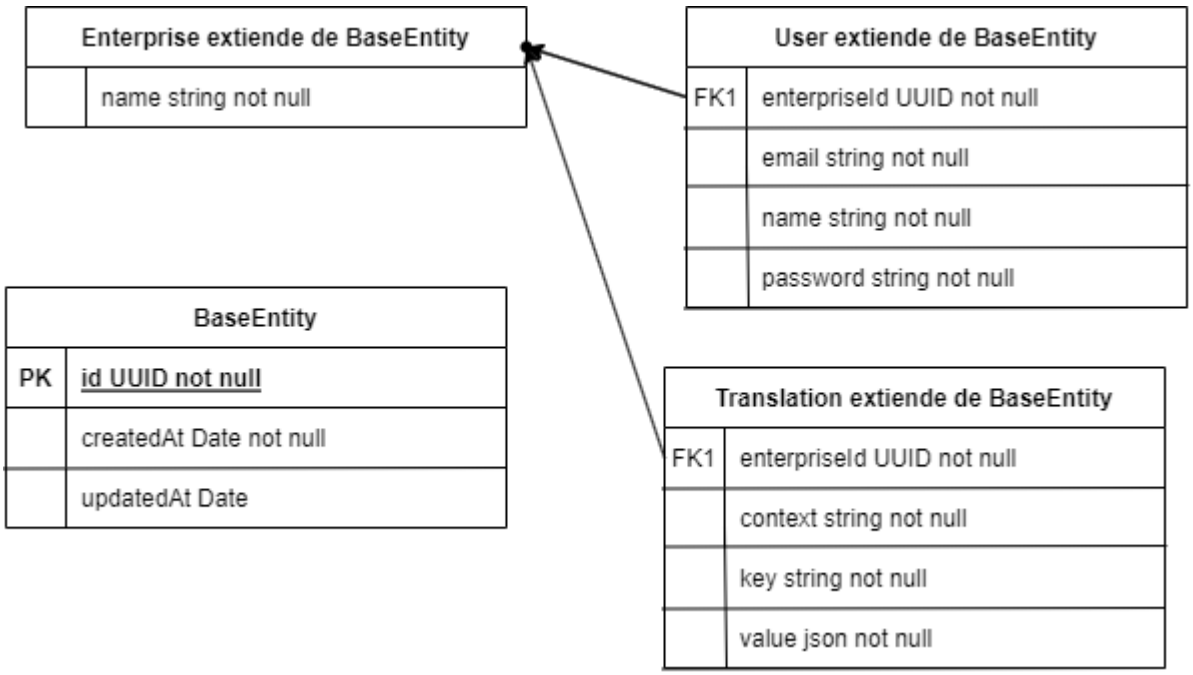

Figura 1: Diagrama de entidades

4.2. Diseño del servidor

Vamos a hacer un diseño tanto de las rutas HTTP de nuestro servidor como de los datos que necesitará.

En cuanto al módulo de empresa:
- Creación singular de una empresa -> POST /v1/enterprises.
- Actualización singular de una empresa -> PATCH /v1/enterprises/{enterpriseId}
- Listado de empresas -> GET /v1/enterprises
- Obtención unitaria de una empresa -> GET /v1/enterprises/{enterpriseId}

En el módulo de traducciones:
- Creación masiva de traducciones -> POST /v1/translations
- Actualización masiva de traducciones -> PATCH /v1/translations
- Borrado múltiple de traducciones -> DELETE /v1/translations

Y por último, el módulo de usuarios:
- Creación singular de un usuario -> POST /v1/users
- Actualización singular de un usuario -> PATCH /v1/users/{userId}
- Listado de usuarios -> GET /v1/users
- Obtención unitaria de un usuario -> GET /v1/users/{userId}
- Inicio de sesión de un usuario en una empresa -> POST /v1/enterprises/{enterpriseId}/users/login

En las rutas, todo lo que venga dentro de unas llaves será una variable que se deberá sustituir para que la ruta HTTP funcione. Por ejemplo, si quiero obtener la empresa con identificador 1, haré una petición a la ruta de obtención unitaria de la siguiente forma: GET /v1/enterprises/1.

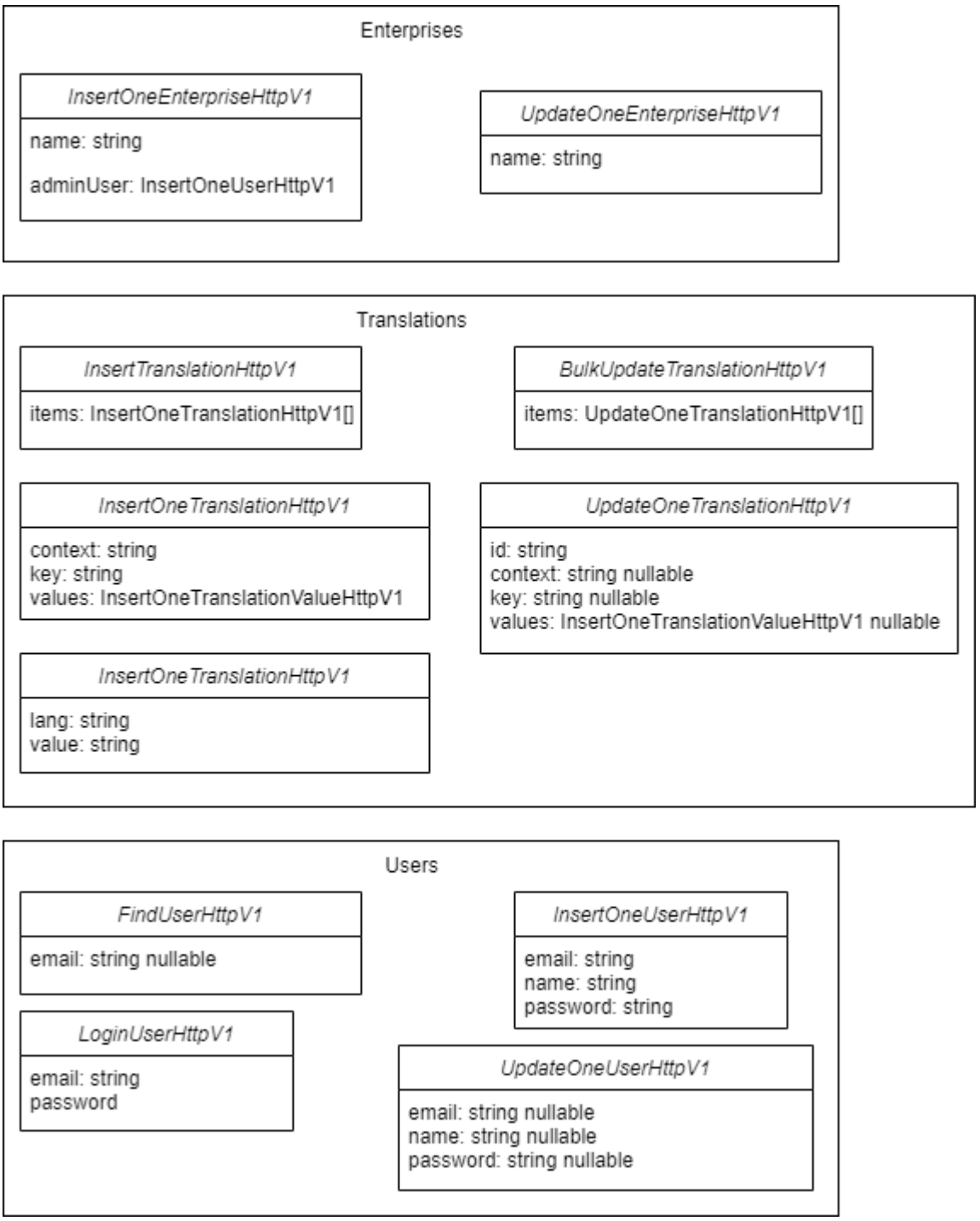

Figura 2: Modelos http de entrada y salida

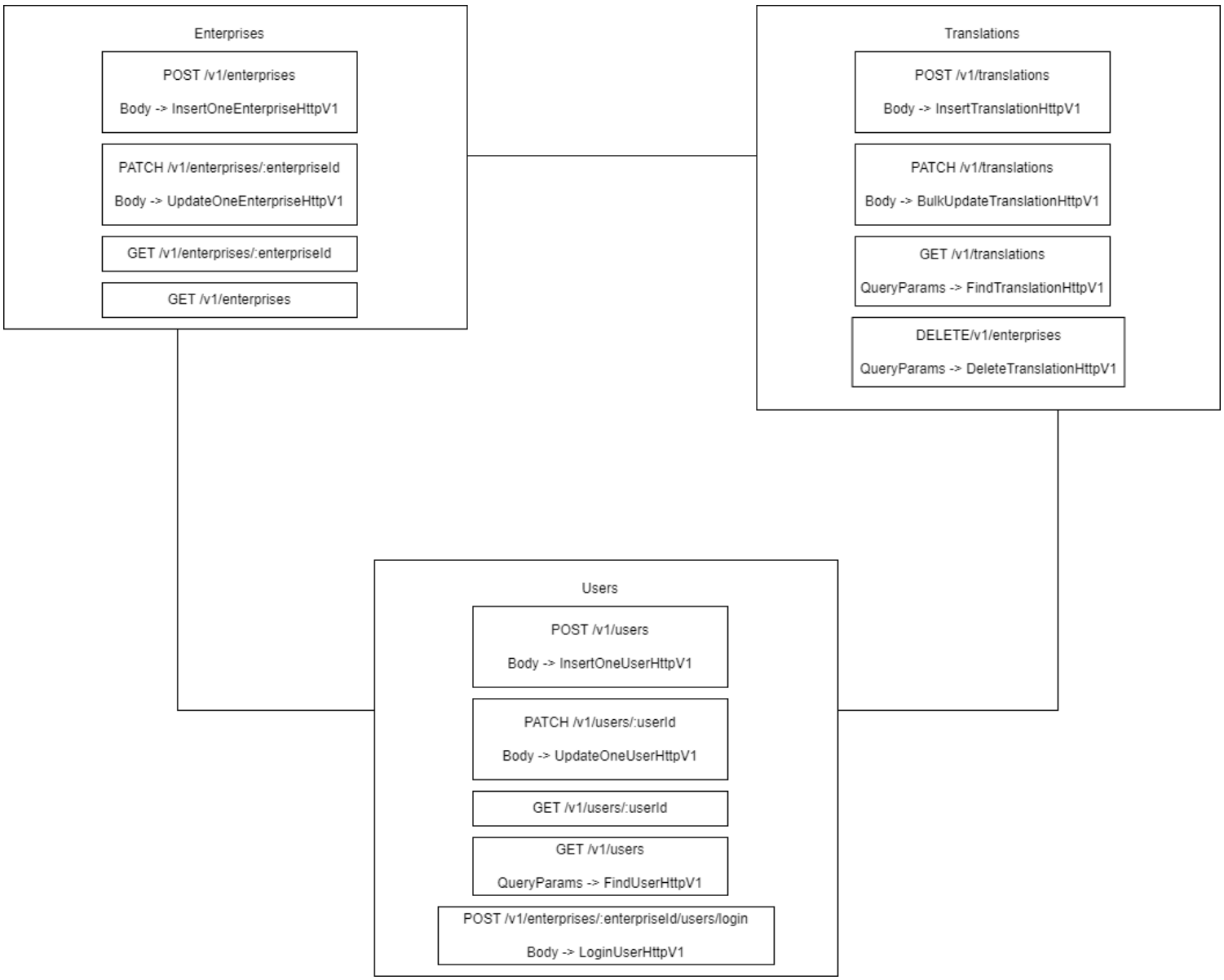

Figura 3: Diseño de rutas http del servidor

4.3. Lógica de negocio

En este capítulo vamos a describir la lógica de negocio de nuestra aplicación. Con ello describiremos qué acciones podrán realizar los usuarios en nuestro sistema, de esta forma lograremos el control que necesitamos para que todo funcione correctamente.

En cuanto a la lógica aplicada entre un usuario y los demás:

- Solo podrá listar u obtener usuarios de su misma empresa
- Podrá actualizar los datos que tenga sobre sí mismo, excepto si es administrador, que tendrá permiso para actualizar cualquier dato de cualquier usuario.

En lo referente a las traducciones:

- Serán únicas por empresa, clave y contexto.
- Un usuario será capaz de obtener todas las traducciones de su empresa.

Y respecto a las empresas, un usuario podrá actualizar los datos de su empresa si es su administrador.

4.4. Diseño de la integración contínua del código

A continuación se exponen todos los requisitos que deben cumplir los desarrolladores para que su código pueda mezclarse en la rama principal del proyecto (main).

Estos requisitos deberán ser probados tanto en un entorno local, antes de que se suba el código al repositorio, como en un entorno remoto aislado. Ya que el entorno local puede estar corrupto y no ejecutar los pasos de forma correcta, por lo que siempre se lanzarán en el entorno remoto para asegurarnos de que en una máquina nueva sin ninguna configuración adicional funciona perfectamente.

Formato

Tabulaciones de dos espacios.

Comillas simples para las variables de tipo texto siempre que se pueda.

Un espacio después de cada llave si se pretende escribir.

Punto y coma al final de cada sentencia

Coma al final de cada elemento de una lista.

Estructura

Definición de tipos en todas las declaraciones de variables

Definición consistente de los tipos de los argumentos de entrada de las funciones.

Complejidad de máximo 12 puntos en ramas lógicas.

Variables en un objeto ordenadas alfabéticamente.

Importaciones de librerías ordenadas alfabéticamente.

No pueden haber importaciones de librerías no utilizadas en un archivo.

Pruebas unitarias

Todos los archivos de este proyecto que implementen lógica de programación deben ser probados de forma unitaria.

En las pruebas de código unitarias existe una característica llamada cobertura. Ésta básicamente describe el porcentaje de líneas de un archivo por las que se ha pasado durante una prueba.

Se utiliza fundamentalmente para saber si todo ha sido probado correctamente y si no es así, qué líneas faltan por probar.

La cobertura de las pruebas de estos archivos deben ser del 100%, de forma que no se deje ninguna línea de código por revisar.

Aunque hay excepciones de cobertura, una aplicación bien programada nunca debe tener un 100% de líneas revisadas, porque eso significa que no hemos utilizado herencia o al menos, no correctamente.

En las clases que usemos herencia no tendremos ese 100% de cobertura, ya que no testeamos funcionalidades implementadas en la clase de la que hereda.

Cada vez que se ejecute un paso se subirán los archivos para que Gitlab analice esta cobertura y nos deje mezclar el código si no bajamos del porcentaje límite de líneas revisadas.

4.5. Diseño de los despliegues continuos del servicio

En este apartado vamos a describir cuales son los pasos necesarios para que se pueda cumplir correctamente con una filosofía de despliegues continuos.

La versión del proyecto es fundamental, ya que nos indica qué tipo de cambios ha ido sufriendo el proyecto, por lo que se deberá ir incrementando de forma automatizada dependiendo de las funcionalidades que se hayan implementado.

Este incremento se basa en los commits hechos por los desarrolladores que siguen un estándar para determinar si debe subir o no de versión, y, en caso de que sí, cuál ha de ser el incremento.

La versión de un proyecto viene determinada por tres números separados entre sí por puntos (1.2.3). El primer número indica la versión mayor, el segundo la menor y al tercero se le conoce como revisión.

Si se sube un commit que lleva una funcionalidad que solo arregla un error y se publica, incrementará en uno la revisión; si sube una nueva característica, uno el menor y si sube un cambio que rompe contrato con los clientes, subirá uno la mayor.

El cambio que predomina frente a los demás será el que más impacto tenga y será el que decidirá qué parte de la versión se incrementa.

Cuando se mezcla código en la rama principal del repositorio, tendremos un proceso automatizado para publicar una nueva versión del servicio y desplegarla en el entorno de producción.

Este proceso crea un puntero al último commit mezclado en el repositorio y despliega el código que hay en éste en el entorno.

Las modificaciones en base de datos que los desarrolladores han implementado en su código se lanzan una vez se despliegue el servicio.

Diseño de la monitorización del servicio

Este servicio de monitorización debe proporcionarnos métricas sobre:

- El tiempo tarda en responder cada petición.
- El tiempo de respuesta de los servicios internos usados en cada endpoint, como la base de datos.
- El contenido de la petición para poder analizarla en caso de error.
- Las transacciones de base de datos que se han hecho en cada petición.
- CPU y RAM utilizados por el servicio en todo momento, para poder analizarlos en caso de caída y saber lo que ha ocurrido.

Este sistema no debe ser dependiente de nuestra infraestructura, ya que si lo fuera, en caso de que ésta cayera, no seremos capaces de monitorizar y no sabremos lo que está ocurriendo.

Por esto, deberemos apoyarnos en un proveedor alternativo de servicios APM o asegurarnos de que el sistema implementado por el nuestro no sea dependiente de nuestra infraestructura.

En nuestro caso, Google Cloud Platform tiene un sistema de monitorización de recursos sobre máquinas ya implementado, por lo que lo utilizaremos.

Y para la monitorización de peticiones a nuestra API utilizaremos Elasticsearch APM.

5. Implementación

5.1. Herramientas de desarrollo

Ubuntu

Utilizamos ubuntu (“Ubuntu”) como sistema operativo, ya que tiene una mejor integración con Docker que windows.

NodeJS

NodeJS (“NodeJS”) es un entorno de ejecución que nos permitirá ejecutar Javascript, el lenguaje de programación con el que desarrollaremos nuestra API.

Docker

Docker es un sistema de contenerizado de aplicaciones. En nuestro caso nos permitirá contenerizar nuestra API y una base de datos, con el objetivo de optimizar el desarrollo en nuestro entorno local.

Entorno de desarrollo integrado

El IDE que utilizaremos para implementar este proyecto será Webstorm.

Este editor ha sido desarrollado por la Jetbrains y su objetivo es dar un entorno completo al programador para que pueda implementar cualquier funcionalidad orientada a la web.

Hemos instalado el los siguientes plugins en el IDE:

- Terraform (“Terraform”) para que nos ayude con la sintaxis del lenguaje
- Eslint (“Eslint”) para la estandarizar la estructura del código
- Prettier (“Prettier”) para estandarizar el formato del código
- One Dark Pro como tema para la visualización del entorno.

Cliente de base de datos

El cliente que utilizaremos será Robo3T (“Robo3T”), uno de los clientes más utilizados para MongoDB, nuestra base de datos.

Postman

Postman (“Postman”) es una plataforma que nos permite hacer la creación y el uso de APIs de forma más sencilla.

En nuestro caso nos ayuda a componer peticiones HTTP y HTTPs fácilmente.

Nos ayuda a automatizar pruebas de integración de nuestra API en caso de que quisiéramos hacerlas.

Nos permite tener grupos de trabajo entre desarrolladores y colecciones de peticiones en las que guardar variables y poder utilizarlas en estas colecciones.

Además, postman tiene integrado todos los sistemas de autenticación estándar.

Git

Git (“Git”) es el sistema de control de versiones que utilizamos en nuestro proyecto y el más usado y estandarizado a nivel global.

Nos permite tener ramificaciones de las versiones para que no afecten a la actual hasta que nosotros decidamos mezclar estas ramas.

También podemos analizar que partes del proyecto ha manipulado cada desarrollador y en qué instante se produjo.

Además, lo necesitamos para poder utilizar Gitlab como proveedor de repositorios cloud.

Fork

Fork (“Fork”) es la aplicación que usamos para utilizar git fácilmente y no a base de comandos en la consola.

Tiene integración con Gitlab (“Gitlab”), lo cuál nos facilita bastante el proceso de descarga de repositorios.

5.2. Estructura de la aplicación

Este proyecto ha sido desarrollado todo en un mismo repositorio y se subdivide en cuatro conjuntos.

Por una parte, tenemos la integración contínua y los despliegues contínuos.

Esta sección está codificada dentro de un archivo llamado gitlab-ci.yaml, situado en la raíz del repositorio. En él describimos la forma en la que se realizan los pasos que hemos diseñado anteriormente en el entorno remoto.

Además, para poder hacer las pruebas en local y cumplir con la parte de integración contínua en local, tenemos la carpeta .husky, la cual tiene dos archivos, uno llamado pre-commit.sh y otro llamado prepush.sh, los cuales se ejecutan antes de un commit o push al repositorio de código, respectivamente.

Durante la ejecución del último paso descrito en el archivo citado anteriormente, se crean recursos de infraestructura descritos en la carpeta infra.

En cuanto al código fuente de la aplicación, está en la carpeta src, en la raíz del repositorio.

La información sobre el repositorio la encontramos en dos archivos. Un README.md, que describe cómo ha de instalarse el servicio para poder ejecutarse en un entorno local y un Changelog.md, que contiene todos los cambios que se van dando en el servicio según se vayan sacando publicaciones de este.

5.3. Implementación del servicio

Ahora vamos a centrarnos en la parte de la arquitectura del código implementado en la aplicación.

La estructura del código de nuestra API se basará en una arquitectura hexagonal (“Arquitectura hexagonal”) orientada al dominio y el patrón CQRS (“Patrón CQRS”).

El patrón CQRS se define como una segregación entre operaciones de escritura y lectura de bases de datos y tiene como objetivo maximizar la velocidad, seguridad y escalabilidad de éstas.

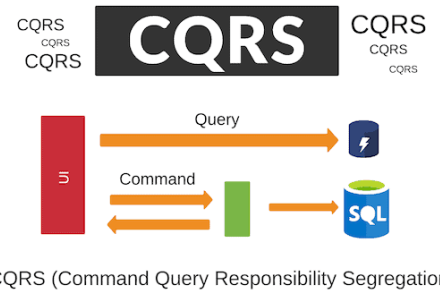

Figura 4: Diseño patrón CQRS

```
1 inheritor  6+ usages
export interface BaseEntityFindQuery extends IQuery {
    ids: UUID[] | undefined;
}
```

Figura 5: Ejemplo de interfaz de una query de dominio

La imagen anterior sería un ejemplo de contrato que deberían cumplir todas las operaciones de lectura masiva no paginada de una entidad que extendiera de nuestra entidad base.

Contaremos con un módulo común llamado "_shared", donde implementaremos funcionalidades compartidas entre las distintas capas de los diferentes módulos.

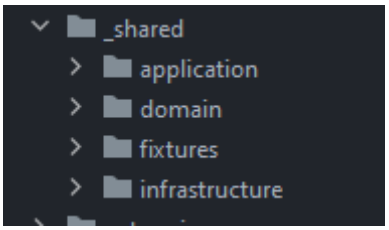

Figura 6: Distribución de la carpeta común entre módulos

En este definiremos nuestra entidad base y abstraeremos la máxima lógica posible con el objetivo de seguir un patrón D.R.Y. (no te repitas a tí mismo).

La arquitectura hexagonal describe tres capas, aplicación, dominio e infraestructura.

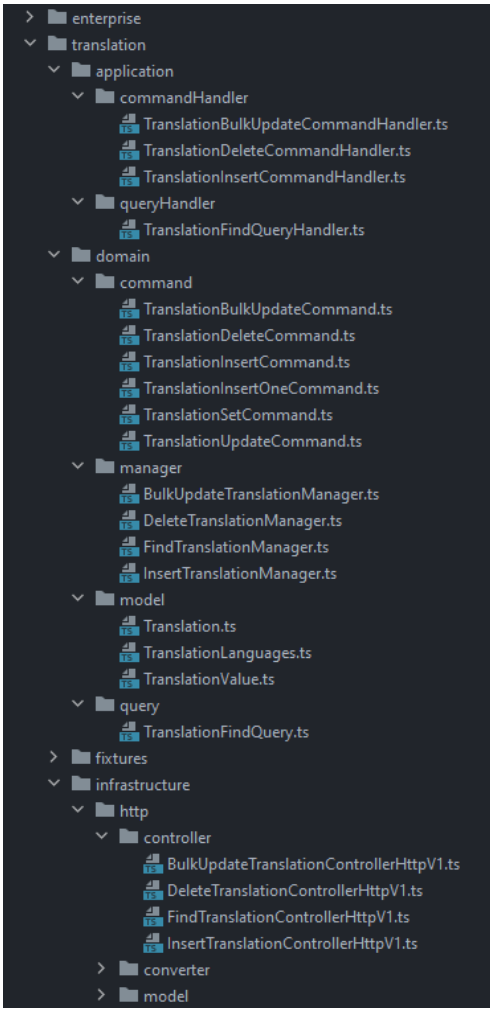

Figura 7: Ejemplo de distribución de archivos de un módulo

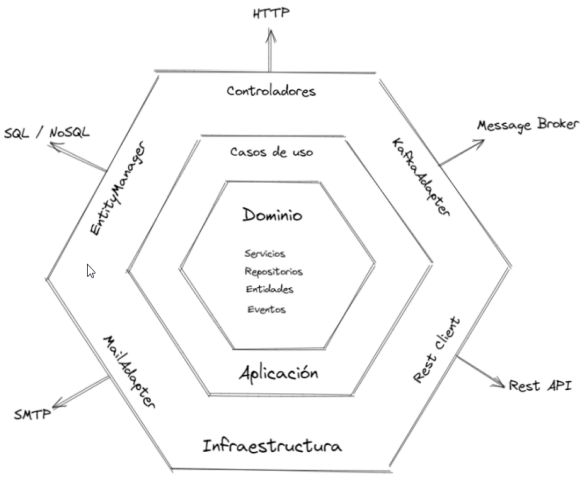

Figura 8: Arquitectura hexagonal

Aplicación

Entrelaza la capa de dominio e infraestructura mediante controladores de peticiones de lectura y/o escritura y eventos.

Dominio

Describe las entidades, peticiones, órdenes y eventos y contiene los contratos que deberán cumplir los adaptadores de la infraestructura cuando se implementen.

Infraestructura

Ejecuta las peticiones, órdenes y eventos implementando los contratos definidos en el dominio mediante los proveedores que sean necesarios en los diferentes casos.

Recibirá entidades y/o implementaciones de contratos de lectura y escritura del dominio y deberá convertir estos contratos a los especificados del proveedor que utilice para poder llevar a cabo la operación como el dominio indica.

En la siguiente imagen observamos una conversión del contrato de la operación de lectura mostrada en una de las imágenes anteriores en el contrato que espera recibir nuestro cliente de base de datos.

```
private convertToBaseEntityFindQueryMikroOrm(input: BaseEntityFindQuery): ObjectQuery<BaseEntityMikroOrm> {
    const output: ObjectQuery<BaseEntityMikroOrm> = {};

    if (input.ids !== undefined) {
        output.id = {
            $in: input.ids,
        };
    }

    return output;
}
```

Figura 9: Ejemplo de conversor de query de dominio a query de orm

5.4. Implementación de la integración contínua y de los despliegues contínuos

Al estar utilizando Gitlab como proveedor de repositorios de git en cloud, utilizaremos los llamados Gitlab Pipelines.

Este servicio ejecuta pasos en contenedores de aplicaciones dependiendo de la programación que tengamos en el “gitlab-ci.yaml”.

La imagen que ejecutan estos contenedores se especifica en cada uno de los pasos y el usuario que ejecuta el pipeline ha de tener acceso a él.

Integración contínua

En nuestro caso definiremos unos pasos a cumplir tanto en local como en remoto.

Lo primero que haremos tanto en remoto como en local será ejecutar el comando “npm install” para instalar las dependencias que necesitamos.

En el entorno local, husky, instala unos hooks que ejecutan scripts ante un commit y/o push.

En nuestro caso ante un commit ejecutamos el comando “npx lint-staged”, lo cual comprueba el formato y la estructura del código del que vayamos a hacer commit.

Y ante un push, ejecutamos el comando “npm run test”.

En caso de que cualquiera de los comandos falle no se ejecutará el commit o el push que se haya hecho.

Nuestro servicio necesita NodeJS para ejecutarse, por lo que en el entorno remoto, deberemos partir de una distribución alpine que ya lo tenga instalado.

En cuanto a los pasos que se han de cumplir en remoto, se ejecutarán una vez se haga un push al repositorio remoto y se abra una solicitud de mezcla de código.

Habrán tres pasos de comprobación que se ejecutarán en paralelo, uno que comprueba el formato, uno para la estructura del código y otro para los test unitarios.

La definición del formato del código vendrá dado por el archivo .prettierrc, que será el siguiente:

```
{
	"arrowParens": "always",
	"bracketSpacing": true,
	"endOfLine": "lf",
	"printWidth": 120,
	"quoteProps": "as-needed",
	"semi": true,
	"singleQuote": true,
	"tabWidth": 2,
	"trailingComma": "all",
	"useTabs": true
}
```

Las comillas que usaremos serán simples siempre que se pueda.

Utilizaremos tabulaciones siempre que se pueda.

Deberá haber un salto de línea al final de cada archivo.

Y al final de cada sentencia siempre habrá un punto y coma.

Despliegues contínuos

Los pasos del despliegue contínuo se podrán ejecutar una vez se mezcle código a la rama principal del repositorio.

Habrá un paso para crear un lanzamiento.

Se basará en la versión actual del servicio y en los commits hechos por los desarrolladores hasta el momento, que serán los que marcarán el incremento de la versión.

Estos commits deberán hacerse siguiendo el estándar de commits convencionales, los cuales indican si se está creando una nueva característica, arreglando otra o simplemente continuando con alguna funcionalidad sin querer que aparezca en el changelog.

Cuando finalice este paso, se ejecutará la construcción de la imagen que utilizarán los contenedores de nuestra aplicación en el proveedor cloud.

Después de éste, se comprobará el formato del código de infraestructura.

A continuación, se hará el plan de despliegue de la infraestructura.

Y para finalizar, se aplicará el plan hecho en el paso anterior.

Estos dos últimos pasos hay que dividirlos así porque de esta forma podemos ver el plan antes de ejecutarlo y si no nos convence el plan, podemos no ejecutarlo.

El archivo que define cómo va a ser el contenedor de la aplicación en kubernetes es el Dockerfile.

En éste indicamos la imagen de node de la que debe partir, un linux alpine con NodeJS versión 18.12.1.

Lo primero que haremos será instalar las dependencias de nuestro servicio.

A continuación construiremos la carpeta “dist”, la cual contiene el transpilado del código Typescript a Javascript.

Borramos las dependencias de desarrollo.

Finalmente, ejecutamos el archivo main.js.

Esta sería la definición de nuestro Dockerfile:

```
FROM node:18.12.1-alpine3.16

RUN mkdir /home/node/app && chown node:node /home/node/app

WORKDIR /home/node/app

USER node

COPY --chown=node:node . .

RUN npm ci --ignore-scripts

RUN npm run build

RUN npm prune --production

ENV TZ="$npm_package_config_TZ"

CMD [ "dist/main.js" ]
```

Además, tendremos un archivo ".dockerignore", el cual nos permite excluir todos los archivos y rutas que le indiquemos en él.

```
.git
.vscode
.env
.husky
.idea
.gitlab
data
coverage
dist
docs
docker
infra
firebase
node_modules
```

Los archivos de git son bastante pesados por lo que los ignoraremos.

Igual que los "node_modules" (dependencias), ya no solo porque sea una carpeta pesada, si no porque podrían no funcionar si utilizamos versiones de sistemas operativos distintos.

La carpeta dist, tampoco la subiremos, ya que cada docker debe construir su propia transpilación de código.

En la carpeta docs tendremos imágenes que tan solo serán informativas, por lo que tampoco será necesario subirla.

Y los demás archivos, son totalmente inútiles en cuanto a la ejecución del servicio, por lo que también los ignoraremos.

5.6. Problemas de implementación resueltos

En este apartado hablaremos de dos tipos de problemáticas a las que nos hemos enfrentado durante la implementación del servicio, las de programación de la API y las de implementación de la integración contínua y despliegues contínuos.

Problemas con la programación del servicio

Tuvimos dos problemas principales, la documentación del servicio y la elección de un framework http.

Necesitábamos una herramienta que automatice la generación de documentación, ya que no queríamos estar escribiendo comentarios o generando otros documentos que tan solo describen el comportamiento de nuestra API. Encontramos una herramienta llamada swagger (“Swagger”), la cuál hacía básicamente lo que queríamos pero se nos complicó la integración, debido a que inicialmente no entendimos cómo funcionaba el plugin de swagger que implementó el framework de NodeJS que utilizamos, NestJS (“NestJS”).

Swagger es una herramienta basada en el estándar de OpenAPI que nos ayuda a documentar nuestra API fácilmente.

El plugin se basaba en la recogida de metadatos de los objetos mediante decoradores, nosotros probamos una funcionalidad del plugin que permitía no llamar a estos decoradores, aunque después tuvimos que ponerlos, ya que no casaba bien con nuestra arquitectura.

Y en cuanto a la elección del framework http, teníamos dos opciones. Utilizar Fastify o Express.

Express es el framework más utilizado en el mundo entero, aunque eso no indica que sea el mejor, solo que es el que más tiempo ha estado en el mercado, por lo que todas las librerías suelen estar adaptados a éste.

En cambio Fastify nos pareció la mejor opción, ya que se ha estandarizado en la industria por ser el framework que más peticiones por minuto soporta, siendo capaz de soportar incluso seis veces más que Express según muchas comparaciones de expertos.

Además nuestro framework tiene una integración perfecta con Fastify, por lo que era muy fácil de utilizar.

Problemas con la integración contínua y los despliegues contínuos

Debido a que no teníamos experiencia con Gitlab, Google Cloud Platform, Kubernetes y Terraform, tuvimos bastantes problemas a la hora de implementar la integración contínua y los despliegues contínuos.

Hemos tenido que explorar los sistemas de caché que tienen los pipelines para poder ir pasando datos de paso en paso y no tener que recalcular o descargar artefactos generados anteriormente.

Gitlab, tiene su propio sistema de registro de contenedores, por lo que hemos tenido que aprender a subir las imágenes que necesitemos en nuestro sistema de despliegue contínuo para optimizar los pipelines lo máximo posible teniendo ya los sistemas preinstalados y que no haya que descargar en todos los pasos todo lo necesario cada vez que se ejecuten, tan solo se descargará una imagen que ya lo tendrá todo instalado.

En cuanto a Google Cloud Platform, hemos tenido que aprender a utilizar su registro de contenedores también, ya que necesitamos subir allí las imágenes de cada release para poder desplegarlas en dicho proveedor.

Además, cuando intentamos exponer una IP al exterior vimos que no era igual hacerlo en un cluster local que en Google Cloud, por lo que tuvimos que aprender a exponer un despliegue de contenedores de kubernetes mediante un servicio en el proveedor.

Y para finalizar, el problema con el que más tiempo estuvimos, fue el guardado del estado de la infraestructura de Terraform. Gitlab nos proporciona un sistema para guardar este estado y no nos funcionaba bien, cuando ya teníamos el estado guardado y lo queríamos volver a utilizar para la siguiente publicación del servicio nos salía un error en que nos decía que el estado estaba bloqueado, por lo que tuvimos que estudiar la documentación de Terraform y Gitlab para solucionar este problema.

6. Manuales

Todos los manuales necesarios los dejaremos en el repositorio de código, además de explicarlos aquí.

6.1. Manual de instalación

En esta sección explicaremos cómo instalar este proyecto en una distribución ubuntu, ya que es lo que nosotros usamos en nuestro día a día.

Primeramente instalaremos git para poder tener el control de versiones del proyecto mediante estos comandos

```
sudo apt update

sudo apt install git
```

Para comprobar que la instalación de git ha salido correctamente

```
git –version
```

A continuación instalaremos docker

```
sudo apt-get update

sudo apt-get install ca-certificates curl gnupg

sudo install -m 0755 -d /etc/apt/keyrings

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg
sudo chmod a+r /etc/apt/keyrings/docker.gpg

echo \
    "deb [arch="$(dpkg --print-architecture)" signed-by=/etc/apt/keyrings/docker.gpg] https://download.docker.com/linux/ubuntu \
  "$(. /etc/os-release && echo "$VERSION_CODENAME")" stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

sudo apt-get update

sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin
```

```
sudo groupadd docker

sudo usermod -aG docker $USER

newgrp docker
```

Para verificar que hemos instalado docker correctamente

```
docker run hello-world
```

Seguidamente instalaremos NodeJS

```
curl -fsSL https://deb.nodesource.com/setup_18.x | sudo -E bash - &&\
sudo apt-get install -y nodejs
```

Para verificar que hemos instalado NodeJS correctamente

```
node -v && npm -v
```

Para finalizar instalaremos las dependencias del proyecto mediante el comando

```
npm install
```

Y con esto ya tendremos instalado todo lo necesario para utilizar el proyecto.Copiaremos el archivo .env.example a un archivo .env, y si es necesario, sustituiremos las variables que existen en este documento por las que precisemos en nuestro sistema.

Escribiremos en la terminal “docker compose build” y cuando termine, “docker compose up”.

Con estos dos comandos descargamos las imágenes de Docker necesarias para correr el proyecto.

En este caso necesitaremos la imagen de mongodb y la de nodejs alpine con la versión correspondiente.

Este sería el archivo “docker-compose.yml”

```
version: '3.7'

services:
     mongo:
       image: mongo
       restart: always
       environment:
         AUTH: no

     api:
       build:
       context: .
       dockerfile: Dockerfile
       command: npm run start:dev
       volumes:
         - ./src:/home/node/app/src
       env_file:
       - .env
       depends_on:
       - mongo
       ports:
       - 3000:3000
       - 9228:9228
       - 9229:9229
```

En éste describimos que necesitamos dos servicios, uno de mongodb y otro basado en el Dockerfile descrito en el apartado de despliegues continuos.

El servicio de la api depende del servicio de mongodb, ya que sin él no podría funcionar.

En el momento en el que el comando “docker compose up” haya terminado, podremos probar nuestra aplicación. Lo haremos de la siguiente manera:

Abrimos un navegador cualquiera y en la url de éste ponemos “http://localhost:3000”.

Al acceder a esa url debería salirnos un “Ok” en la pantalla.

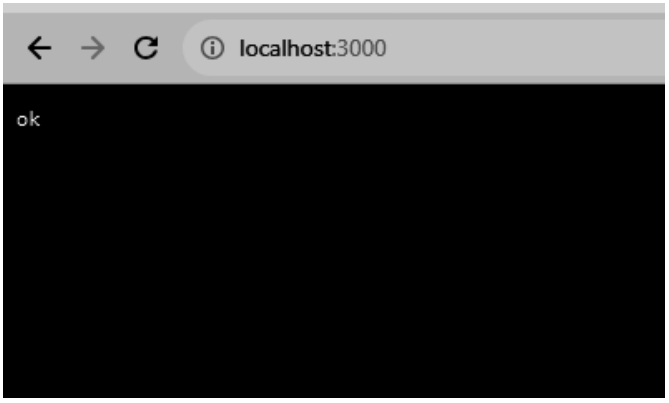

Figura 10: Demostración de funcionamiento de servidor local

6.2. Guía de uso

Gracias a las buenas prácticas de documentación desarrolladas en este proyecto, habremos desarrollado una web estática con ejemplos de todas las llamadas de la API.

Podremos utilizar esta web como una guía de uso.

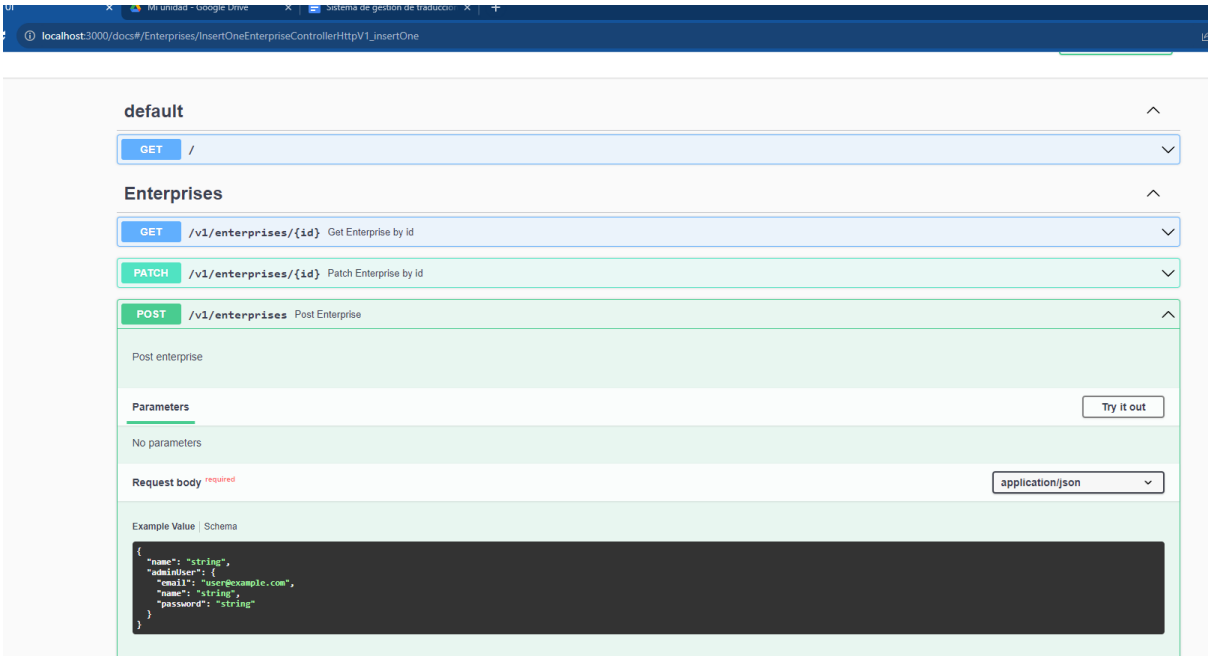

Figura 11: Web estática local de documentación sobre la API

En la imagen anterior observamos un ejemplo de petición HTTP con la cual podremos registrar una empresa en el sistema.

Esta web está basada en un archivo JSON que genera nuestro proyecto basándose en unos metadatos que nosotros generamos en el código. El JSON generado tiene un esquema basado en el estándar Open Api, el cual se utiliza en el desarrollo de APIs para documentar de forma pública una API con el objetivo de que los desarrolladores que vayan a utilizarla tengan una guía de uso.

Después de registrar la empresa en el sistema, ya tendremos un usuario automáticamente creado, el cual tendrá el email y la contraseña que hemos pasado en el cuerpo de la petición.

Utilizaremos este usuario para iniciar sesión en el sistema mediante otra petición HTTP.

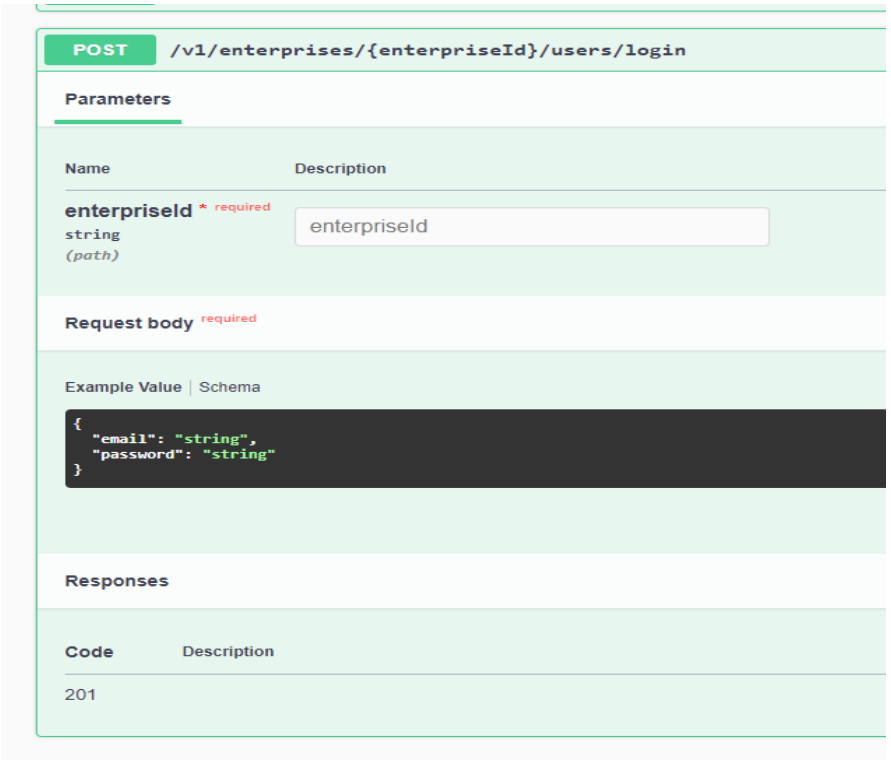

Figura 12: Petición de login a través de la web de documentación

Copiaremos el valor de la propiedad token de la respuesta que nos devuelva esta petición en caso de que el email y la contraseña fueran correctos, haremos click en el botón “Authorize” que hay en la parte superior de la página.

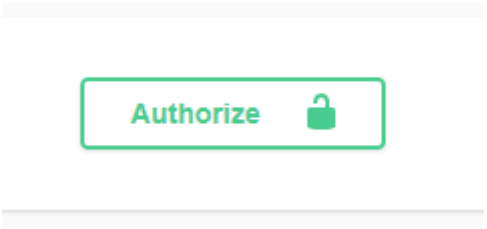

Figura 13: Botón para abrir la pantalla que cambia la autorización de las peticiones de la web de documentación

Y pegaremos el valor del token en el campo de texto que nos saldrá en la pantalla.

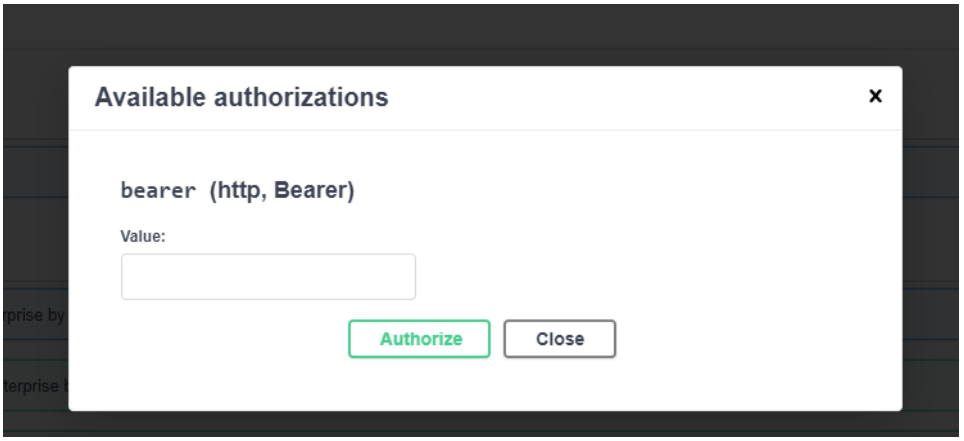

Figura 14: Pantalla que cambia la autorización de las peticiones de la web de documentación

Presionaremos el botón “Authorize” y las siguientes peticiones ya las tendremos autorizadas según el usuario con el que hayamos iniciado sesión.

8. Conclusiones

Se han cumplido todos y cada uno de los objetivos y requisitos que se propusieron.

La elección de Google Cloud Platform como proveedor cloud ha sido una muy buena decisión, ya que nos ofrecen 300$ para empezar a utilizarlo y así no tenemos que pagar nada.

Gitlab ha resultado ser otro de los grandes aciertos del proyecto, es una herramienta con un plan gratuito muy potente, el cuál tiene una capacidad más que suficiente para nuestro proyecto.

Al finalizar el proyecto hemos notado una mejora considerable en nuestras habilidades de programación. Sobre todo en lo referente a la construcción de infraestructuras para servicios en proveedores cloud.

La calidad del código es muy buena y no se ha dejado a penas deuda técnica, lo que nos facilitarán las futuras implementaciones de funcionalidades si queremos continuar el proyecto.

La arquitectura implementada es totalmente reusable para otros proyectos, lo que puede ayudarnos a empezar. Por lo que podemos montar un proyecto en Gitlab plantilla a partir de éste y así, utilizarlo fácilmente sin tener que ir copiando y pegando código de un repositorio a otro, simplemente presionaremos un botón y clonará el estado del repositorio plantilla en el que nosotros queramos.

Uno de los errores de este proyecto ha sido la elección del motor de base de datos, ya que no es necesario hacerlo mediante una base de datos no relacional, es más un modelo relacional hubiera sido más acertado, ya que realmente no hemos utilizado los elementos de las colecciones como documentos, sino más bien como elementos basados en columnas relacionadas con otras entidades.

8.1. Evaluación

Cuando empezamos este proyecto lo encaramos como un reto educacional y profesional, ya que todos los conocimientos que queríamos adquirir en durante él íbamos a poder aplicarlos en nuestra carrera.

Al analizar lo que hemos hecho, vemos que los conocimientos que deseábamos adquirir, principalmente sobre integración y despliegues contínuos, los tenemos.

Hemos sido capaces de resolver una gran cantidad de errores en este campo, darte tantos golpes con un mismo tema siempre te refuerza en él y ahora mismo podemos encontrar solución a muchos problemas que al principio del proyecto no sabríamos ni empezar a plantear.

Existen una grandísima cantidad de empresas que necesitan la automatización de despliegues llevada a cabo en este proyecto, ya que la gran mayoría se limitan a desplegar sus servicios de forma manual, ya sea mediante ir ejecutando proceso a proceso, copiando y pegando resultados de cada uno de ellos. En un entorno profesional ejecutar procesos de forma manual lleva a equivocaciones que cuestan tiempo y por lo tanto, dinero.

Por lo expuesto durante esta sección podemos decir que estamos muy contentos con el desempeño llevado a cabo y sobre todo, por el aprendizaje, el cuál podremos trasladar a nuestro entorno.

Además, gracias al desarrollo de este servicio, uno de nuestros objetivos profesionales descritos en la introducción de esta memoria, transicionar de un puesto profesional de ingeniero backend a uno híbrido entre el anterior e ingeniero DevOps, se ha cumplido.

8.1. Trabajo futuro

En este proyecto, nosotros tan solo hemos implementado pruebas unitarias de nuestro código, éstas verifican el comportamiento de cada una de las piezas probadas, lo que aporta mucho valor a nivel de producto, ya que tanto las refactorizaciones del código como las nuevas implementaciones serán mucho más seguras.

Pero para tener un nivel mucho mayor de seguridad deberíamos diseñar y desarrollar pruebas de integración automatizadas, las cuales probarán las funcionalidades completas. Por ejemplo, una prueba de integración de registro de empresa en nuestro sistema podría ser:

- Registramos la empresa
- Verificamos que existe listándola
- Iniciamos sesión con el usuario administrador pasado en la petición
- Todo lo anterior ha funcionado correctamente

Esto nos verificaría que toda la funcionalidad anterior se ha desarrollado bien y que no hay ningún fallo en el sistema a la hora de registrar empresas.

Estas pruebas las podríamos añadir a nuestros pasos de integración contínua de código, para una mayor seguridad.

Aunque las verificaciones siempre vendrán bien, lo que sin duda falta para que este proyecto sea viable es una aplicación con una interfaz que nos permita comunicarnos con la API sin enterarnos.

Esta aplicación aportaría mucha madurez al proyecto y dejaría paso a otra de las funcionalidades que nos gustaría implementar en un futuro, websockets.

Los websockets nos posibilitarán crear salas entre usuarios para que puedan interaccionar unos con otros y así evitar algunas condiciones de carrera que se podrían dar en la aplicación si varios usuarios estuvieran trabajando a la vez en las mismas traducciones.

Al terminar la aplicación y la implementación de las salas, ya tendríamos un producto que podríamos sacar al mercado.

9. Código

El código está subido en un repositorio público de Gitlab, por lo que cualquiera podrá acceder a él.

Pinchar aquí para entrar en el repositorio.

En él podremos encontrar todos los registros de las acciones llevadas a cabo por los desarrolladores durante el proyecto.

10. Referencias de imágenes

Figura 1: *Diagrama de entidades.*

Figura 2: *Modelos http de entrada y salida.*

Figura 3: *Diseño de rutas http del servidor.*

Figura 4: *Diseño patrón CQRS.*

Figura 5: *Ejemplo de interfaz de una query de dominio.*

Figura 6: *Distribución de la carpeta común entre módulos.*

Figura 7: *Ejemplo de distribución de archivos de un módulo.*

Figura 8: *Arquitectura hexagonal.*

Figura 9: *Ejemplo de conversor de query de dominio a query de ORM.*

Figura 10: *Demostración de funcionamiento de servidor local.*

Figura 11: *Web estática local de documentación sobre la API.*

Figura 12: *Petición de login a través de la web de documentación.*

Figura 13: *Botón para abrir la pantalla que cambia la autorización de las peticiones de la web de documentación.*

Figura 14: *Pantalla que cambia la autorización de las peticiones de la web de documentación.*

11. Bibliografía

"Arquitectura hexagonal." Accessed 5 Julio 2023.

"DevOps." *Wikipedia*, https://es.wikipedia.org/wiki/DevOps. Accessed 8 Mayo 2023.

"Docker." 2016, https://www.docker.com/. Accessed 08 Mayo 2023.

"Eslint." https://eslint.org/. Accessed 23 Mayo 2023.

"Fork." https://git-fork.com/. Accessed 28 Mayo 2023.

"Git." https://git-scm.com/. Accessed 28 Mayo 2023.

"Gitlab." https://about.gitlab.com/. Accessed 28 Mayo 2023.

"Google Cloud Platform." 2016, https://cloud.google.com/gcp/. Accessed 8 Mayo 2023.

"Kubernetes." *Kubernetes*, https://kubernetes.io/es/. Accessed 08 Mayo 2023.

"Localazy." *Content & Software localization on autopilot*, https://localazy.com/. Accessed 8 Mayo 2023.

"Lokalise." *Lokalise*, 2022, https://lokalise.com/. Accessed 28 05 2023.

"NestJS." https://nestjs.com/. Accessed 10 Junio 2023.

"NodeJS." https://nodejs.org/es. Accessed 20 Mayo 2023.

"Patrón CQRS." https://learn.microsoft.com/es-es/azure/architecture/patterns/cqrs. Accessed 8 Julio 2023.

"Postman." https://www.postman.com/. Accessed 28 Mayo 2023.

"Prettier." https://prettier.io/. Accessed 25 Mayo 2023.

"Robo3T." https://robomongo.org/. Accessed 28 Mayo 2023.

"Swagger." https://swagger.io/. Accessed 5 Junio 2023.

"Terraform." *Terraform*, https://www.terraform.io/. Accessed 25 Mayo 2023.

"Ubuntu." *Ubuntu: Enterprise Open Source and Linux*, https://ubuntu.com/. Accessed 20 Mayo 2023.