Facultad de Ciencias

API de HousingOn

HousingOn's API

Trabajo de fin de grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Joaquín García Benítez
Codirector: Ana Isabel Gómez Pérez
Codirector: Domingo Gómez Pérez

Julio de 2020

Agradecimientos

Me gustaría dedicar este proyecto a mi familia, que me ha estado apoyando a lo largo de este camino tanto en los momentos buenos como en los malos. Sobre todo este año, que han tenido que hacer un esfuerzo extra para que pudiera concentrarme con mis estudios en casa.

A mis amigos, que han estado animándome en todo momento a continuar con el grado hasta el final, me han hecho apreciar la suerte que tengo y han estado ahí para todo. Y ya que estos son los agradecimientos de un proyecto de Ingeniería Informática, me gustaría destacar a aquellos amigos que han participado junto a mí en eventos como el *hackathon* Hack2Progress, y a los que posteriormente han venido a recogerme para no tener que conducir hasta casa después de 24h programando.

También me gustaría agradecer a todo el equipo de Emancipia la oportunidad de formarme con ellos y dejarme demostrar lo que valgo, la confianza que han depositado en mí, mayor incluso que la mía propia. Gracias por dejarme formar parte de vuestra familia. En especial me gustaría agradecer a Alfonso Nicholls todos los detalles que ha tenido conmigo y todos los consejos que me ha proporcionado durante estos 4 años.

A todos los profesores mostrándose siempre atentos para resolver mis dudas. En especial al vicedecano Rafael Menéndez de Llano Rozas, por su entrega a la universidad, ayudando a organizar todos los eventos que le proponen o se le ocurren. Es una persona a la que puedes recurrir con cualquier tema o problema y que cuyo trato me hace sentirle como un compañero más que como profesor y vicedecano.

A mi tutor, Domingo Gómez Pérez, y a Ana Isabel Gómez Pérez, quiero agradecerles que hayan dedicado su tiempo a ayudarme con esta tarea y que hayan hecho un esfuerzo extra para que pudiera entregar el proyecto a tiempo.

Por último me gustaría felicitar a todos los compañeros que se gradúan este curso, para que quede en el recuerdo, ya que debido a las condiciones especiales causadas por el Covid-19 no hemos tenido ocasión de celebrarlo como es debido.

Gracias por su atención y sin mayor demora podemos dar paso al proyecto.

API de HousingOn

Resumen

palabras clave: BBDD, Flask, Back end, GraphQL, API, MariaDB, Docker

La empresa HousingOn, compañía que gestiona el alquiler de cientos de estudiantes universitarios, actualmente dispone de un conjunto de aplicaciones web para poder llevar el control del flujo que se han vuelto lentas y desactualizadas. Además, este conjunto de aplicaciones también se han visto afectadas por lo siguiente:

- Falta de estándares en desarrollo y mantenimiento, donde cada programador seguía su propio criterio. Carencia de documentación de las aplicaciones y complejidad del mantenimiento por no seguir las buenas prácticas para el desarrollo software
- Gran incremento de usuarios, además de mantener los usuarios existentes.
- Inconsistencia de datos debido a un diseño simple de la base de datos.
- Carencia de un sistema para obtener, actualizar e insertar datos desde aplicaciones para dispositivos móviles.

La base de este proyecto será la creación de una Interfaz de Programación de Aplicaciones, la cual permitirá conectar diferentes aplicaciones y servicios y permitir la comunicación de datos entre sí. Con este proyecto se pretende solventar todas estas carencias brindando a HousingOn de una aplicación estandarizada, documentada, con mejor rendimiento y más adaptada a las necesidades actuales y futuras de la empresa.

HousingOn's API

Abstract

keywords: API,GraphQL, Back end, DB, Flask, MariaDB, Docker

HousingOn is a company which manages accommodations for hundreds of university students. Currently this company has a set of applications that have become slow and outdated.

Furthermore, this set of applications has also been affected by the following issues:

- Lack of standards in development and maintenance, where the previous programmers followed their own criteria. Absence of documentation for the applications and high maintenance complexity, due to not adhering to good practices in software development.
- A large increase in users, in addition to maintain existing ones.
- Data inconsistency produced by a poor design of the database.
- Lack of a system to get, update and insert data from mobile device applications.

The starting point of this proyect is the creation of an application programming interface (API), that will connect different applicacions and services and will allow sharing the data between them. This project aims to solve all these shortcomings by providing HousingOn with a standardized, documented application, with improvements in the performance and adaptation to the current and future needs of the company.

Índice

1 Introducción

> «There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.» [Hoare, 1981, pp. 75-83]

HousingOn tiene actualmente un sistema muy grande para poder llevar el control del flujo completo del alquiler de un alojamiento, compuesto por las siguientes aplicaciones web:

- Un portal de publicación de alojamientos, en el que los clientes pueden consultar el precio, fotografías, la dirección y otras características del alojamiento.

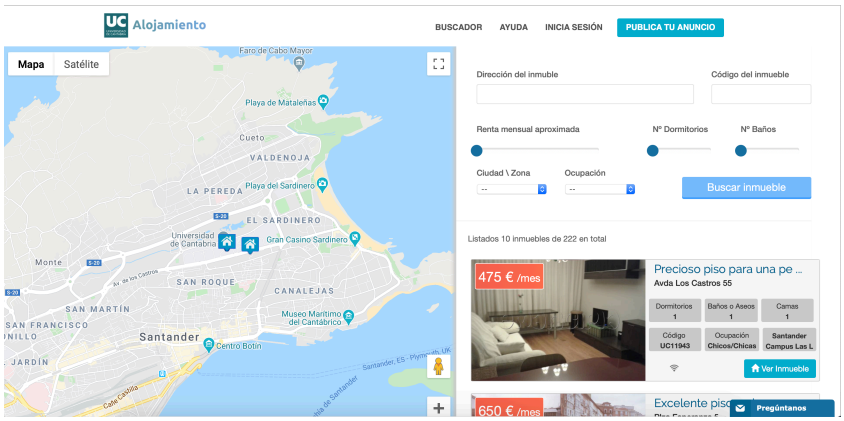

Ilustración 1: Captura de pantalla del portal de publicación de pisos.

- Un portal para propietarios e inquilinos, con el que pueden consultar los gastos, sus compañeros de alojamiento (en caso de ser inquilino en un piso compartido), los correos electrónicos de la empresa, pagar facturas, notificar problemas, ...

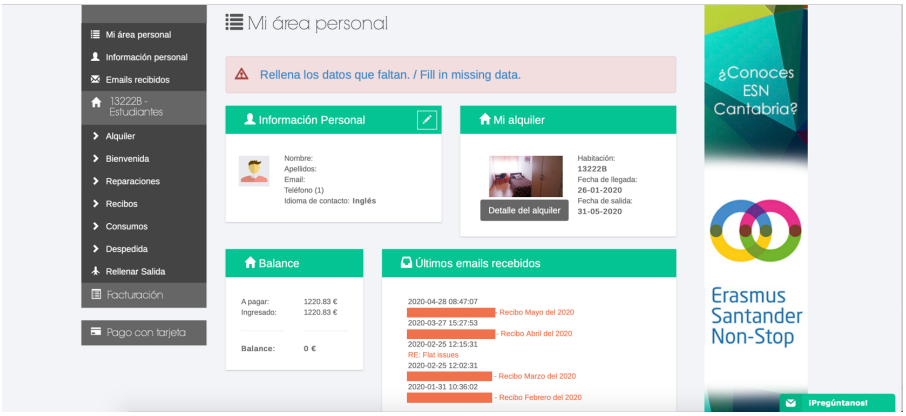

Ilustración 2: Captura de pantalla del área personal de un inquilino.

- Un portal de administración para poder administrar los alojamientos, las incidencias, facturas, inquilinos, llegadas, salidas, etc.

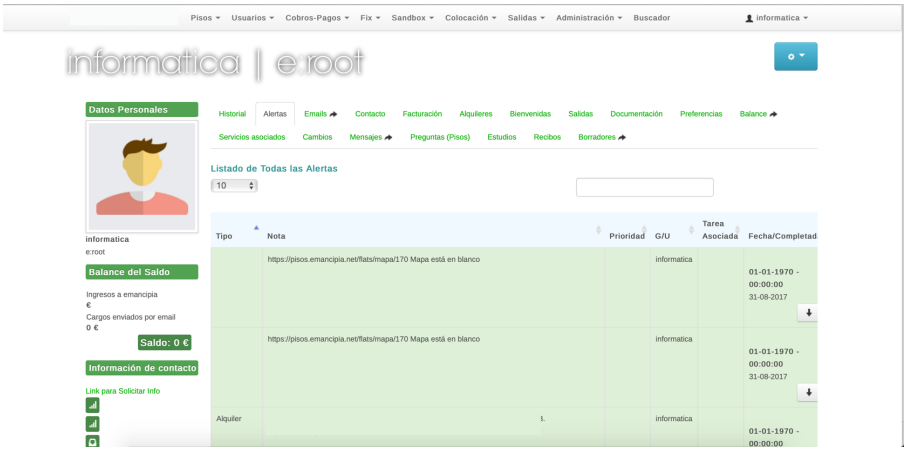

Ilustración 3: Captura de pantalla del portal de administración de la empresa.

El problema es que ese sistema está anticuado, es lento y muy difícil de mantener, además de hacer inviable la creación de una aplicación móvil, que es uno de los deseos de la empresa. Con la creación de este proyecto se tratará de solventar dichos problemas.

Estos portales están desarrollados usando el lenguaje de programación PHP en el lado servidor, en una versión ya antigua de este lenguaje, PHP 5, usando como *framework* principal CakePHP, en concreto, sus versiones ya desactualizadas 2.2 y 2.3.

El *framework* CakePHP está estructurado para desarrollar aplicaciones rápidamente y de forma segura mediante el uso del modelo Modelo-Vista-Controlador, que será explicado en el capítulo tres.

Para las vistas se usa el *framework* de CSS Bootstrap y para la funcionalidad en el lado cliente jQuery, una biblioteca de JavaScript que simplifica la manera de interactuar con documentos HTML.

Cada portal es una aplicación independiente, pero con una base de datos en común, dando lugar a muchas duplicidades de código, debido a que las funcionalidades similares en distintos portales han de ser trasladadas a cada portal.

Antes de poder comenzar a explicar cómo se ha llevado a cabo el proyecto, hay que explicar el concepto base del mismo, debido a que todo el trabajo rondará en torno a él.

El origen del término Interfaz de Programación de Aplicaciones, API por sus siglas en inglés Application Programming Interface, al parecer data de 1968. Es decir, se podría considerar su invención dentro de la tercera generación de computadores, o lo que es lo mismo, en la primera generación de computadores usando circuitos integrados. Se puede observar que el término lleva existiendo desde hace más de 50 años, y a lo largo de todos estos años han surgido interfaces de programación de aplicaciones de todo tipo que se podrían agrupar en:

- **Funciones en Sistemas Operativos**: Los Sistemas Operativos cuentan con todo tipo de interfaces disponibles, ya sea para poder crear las interfaces gráficas de los programas como para poder controlar las diferentes opciones de entradas de datos del usuario.
- **Basadas en clases**: Son interfaces que recogen las clases abstractas en lenguajes de programación orientada a objetos. Estas clases implementan todo tipo de funciones para poder crear nuestros programas.

- **Basadas en bibliotecas**: Las interfaces basadas en bibliotecas recogen la información sobre las mismas, recopilando un conjunto de funciones de forma que permiten ser importadas y utilizadas por diferentes programas.
- **Servicios web**: Este tipo de interfaz de programación de aplicaciones es la más reciente; con ella podemos intercambiar información entre un servicio web y una aplicación. Un servicio web es una tecnología que, a través de protocolos y estándares, permite el intercambio de datos entre aplicaciones de diversa índole.

La base del proyecto será la creación de una Interfaz de Programación de Aplicaciones, la cual permitirá conectar diferentes aplicaciones y servicios y permitir la comunicación de datos entre sí. La Interfaz de Programación de Aplicaciones otorgará una capa de abstracción de forma que mediante una serie de funcionalidades alojadas en la interfaz podremos consultar, modificar, y eliminar siguiendo un formato estándar de intercambio de datos. De esta forma, cualquier aplicación programada posteriormente, ya sea una web, una aplicación móvil, de escritorio o nuevos tipos de aplicaciones como programas para asistentes de voz, podrá hacer uso de la interfaz y manejar fácilmente los datos proporcionados por esta.

La API desarrollada para HousingOn pertenecerá al último grupo de la clasificación anterior. De esta forma, el objetivo principal será desarrollar un servicio web que permita llevar la gestión de los pisos de la compañía desde distintas aplicaciones.

Para ello se han creado dos APIs, una RESTful API que aporta la mantenibilidad y el bajo coste que la empresa necesita, y con la que se ha utilizado JSON como formato de archivo para el intercambio de datos, y como apuesta hacia futuro una API GraphQL, para que los clientes puedan personalizar su propia consulta o inserción de datos, favoreciendo la precisión de las consultas que pueden realizar a sus interfaces. Esta API, a fecha de presentación de este proyecto, requiere de trabajo para integrarse en el sistema completo. Estas tecnologías serán tratadas posteriormente.

Con la misión de obtener una Interfaz de Programación de Aplicaciones se han planificado los siguientes pasos que se describen en detalle en los siguientes capítulos de la memoria:

1. Definir los requisitos de la aplicación.
2. Crear una arquitectura del sistema.
3. Selección de las tecnologías a usar.
4. Desarrollo y testeo de la funcionalidad.
5. Montaje en servidor y lanzamiento.

2 Requisitos

> «The most difficult part of requirements gathering is not the act of recording what the user wants, it is the exploratory development activity of helping users figure out what they want.» [McConnell, 1998]

Según Sommerville [2016], antes de poder desarrollar la aplicación se tiene que comprobar que sea factible desarrollarla. Para ello, se debería de ser capaz de responder afirmativamente las siguientes tres preguntas:

«¿Contribuye el sistema a los objetivos generales de la organización?»

La respuesta a esta pregunta es claramente afirmativa, ya que los trabajadores de la empresa no serían capaces de manejar tantos alojamientos sin una herramienta que les facilitara el trabajo. Además, como se ha comentado anteriormente las aplicaciones que tienen actualmente están empezando a causar problemas, y también necesitan una herramienta que les permita poder compartir y acceder a los datos que generen entre distintos dispositivos y aplicaciones.

«¿Se puede realizar el proyecto a tiempo y con el presupuesto usando la tecnología actual?»

En este caso el proyecto puede ser realizado a tiempo y con el presupuesto, ya que la tecnología actual permite desarrollar APIs web de forma rápida y profesional, gracias al gran auge que han tenido estos años este tipo de servicios.

«¿Puede adaptarse el sistema con otros sistemas que se utilicen?»

En efecto, el sistema con el que cuentan actualmente les permite hacerlo, dado que el principal objetivo de este proyecto es adaptar este mismo sistema, obteniendo una API con la que se espera desarrollar las aplicaciones futuras que sustituyan a las que emplean actualmente.

2.1. Obtención de requisitos

Los requisitos de este proyecto se han recabado principalmente de tres formas:

- Entrevistas al personal de la empresa.
- Observación de la forma de trabajo del personal de la empresa.
- Análisis del código de la aplicación actual.

Con el fin de recabar información, todos los viernes y algunos miércoles se realizan entrevistas al personal de la empresa, en las que se aplican tanto entrevista abierta como entrevista cerrada.

Gracias a esta combinación se puede comenzar la entrevista de forma abierta, y analizar posteriormente las respuestas para recabar las preguntas necesarias para terminar de dar forma a los requisitos en una segunda entrevista cerrada.

Para llevar el control de las entrevistas se han creado de tres documentos de Google Drive:

- El primer documento contiene las cuestiones o dudas que aún necesitan respuesta. Se empleaba durante las entrevistas para poder llevar el control, y poder plantear distintos escenarios al entrevistado para poder comprender mejor el funcionamiento de los distintos flujos de trabajo. Estas entrevistas se hacían frente a un ordenador para poder señalar en la aplicación actual los fallos y carencias que encontraban, ayudando a la recogida de información.
- En el segundo documento se almacenan las dudas o cuestiones ya respondidas. Es de gran importancia, ya que en él se analizan las respuestas para obtener los requisitos. Si se consideraban completos, se pasaban al tercer documento; en caso contrario, se devolvían al primero para repasar las dudas que quedaran.
- El tercer documento consiste en el acuerdo final, y recoge los requisitos definidos para ser implementados. Este documento se mostró en la empresa para comprobar su adecuación antes de dar este proceso por terminado.

Este sistema ha permitido llevar un control bastante preciso sobre los requisitos. Antes de implementarlo, era usual que los requisitos que parecían establecidos cambiasen en otras entrevistas, complicando con ello el proceso de determinación de los requisitos.

2.2. Requisitos de la aplicación

A continuación se recogen los requisitos más notables de la aplicación. Se ha hecho una selección para este trabajo debido a la envergadura del sistema. Los requisitos de la aplicación según el nivel de detalle se pueden clasificar en dos:

- **Requisitos del usuario**: son sentencias elaboradas en un lenguaje natural, en las que se describen los servicios que se espera obtener, y restricciones que la aplicación debe tener.
- **Requisitos del sistema**: son los requerimientos orientados a la solución, en lugar de al problema. Es decir, es un requisito orientado a la lógica del software que va a tener que construirse para dar forma a los requisitos del usuario. Por tanto será más específico para la propuesta final del software que se va a desarrollar. Suelen ser comprobables y susceptibles de testeo, al contrario que los requisitos de usuario.

Los usuarios de la API serán otros desarrolladores, aunque para elaborarla correctamente habrá que fijarse en que el objetivo principal del sistema es dar servicio a un grupo más amplio de usuarios como administrativos, equipo de soporte, personal de reparaciones, inquilinos y propietarios de viviendas. Los requisitos de todos ellos se deberán tener en consideración para poder realizar una implementación satisfactoria.

- **Requisitos funcionales**: se corresponden con el funcionamiento de la aplicación; es decir, cómo se debe comportar en las diferentes situaciones, cómo son los datos de entrada...
- **Requisitos no-funcionales**: están relacionados con otras características que debe tener el sistema; por ejemplo, el rendimiento de la aplicación, requisitos de organización sobre cómo se deberá proceder a realizar ciertas acciones, cumplimiento de estándares...

2.2.1. Requisitos funcionales

La aplicación contará con los siguientes requisitos funcionales:

- **RF1** Se deberán recoger los datos de contacto de un posible inquilino, y el tipo de alojamiento que está buscando. (Véase el cuadro 3)

- **RF2** La aplicación deberá mostrar la información recogida por el registro de posibles inquilinos. (Véase el cuadro 4)
- **RF3** Se deberá poder asignar un usuario inquilino a un alojamiento, de forma que ambos queden relacionados. (Véase el cuadro 5)
- **RF4** La aplicación deberá permitir el registro de nuevos usuarios.
- **RF5** La aplicación deberá permitir que los usuarios inicien sesión en su cuenta.
- **RF6** Se podrán registrar nuevos alojamientos.
- **RF7** Será posible modificar la información de un alojamiento.
- **RF8** Será posible eliminar la información de un alojamiento.
- **RF9** Cualquier usuario podrá ver los alojamientos.
- **RF10** Se podrá filtrar la lista de alojamientos, según las fechas de disponibilidad, precio y tipo de alojamiento.
- **RF11** La aplicación deberá permitir las reservas de alojamiento.
- **RF12** Los usuarios de la aplicación podrán ser eliminados.
- **RF12** La aplicación deberá de permitir la modificación de información de usuarios.
- **RF14** Los usuarios de la aplicación deberán poderse listar.
- **RF15** El listado de usuarios podrá filtrarse por tipo de usuario, nombre, correo electrónico y DNI.

Cuadro 1: Requisitos funcionales de la aplicación

Código	**Descripción**
RF1	Registro de posibles inquilinos y intereses de alojamiento
RF2	Listado de información de posibles inquilinos e intereses de alojamiento
RF3	Relacionar un inquilino con un alojamiento
RF4	Inicio de sesión
RF5	Registro de usuarios
RF6	Creación de alojamientos
RF7	Modificación de alojamientos
RF8	Eliminación de alojamientos
RF9	Listado de alojamientos
RF10	Filtrar listado alojamientos
RF11	Reservar un alojamiento
RF12	Modificación de usuarios
RF13	Eliminación de usuarios
RF14	Listado de usuarios
RF15	Filtrar listado de usuarios

2.2.2. Requisitos no-funcionales

Los requisitos no funcionales de la aplicación son los siguientes:

- **RNF1** Mantenibilidad del software; esto implica que las herramientas a utilizar deben tener una amplia comunidad de usuarios y desarrolladores, de forma que sea fácil buscar respuesta a los problemas que puedan ocurrir.
- **RNF2** El proyecto deberá desarrollarse y mantenerse con el menor coste económico posible.
- **RNF3** El software usado en este sistema debe ser fácil de utilizar por cualquier desarrollador que necesite comprender y mantener la aplicación.
- **RNF4** Al tratarse de una startup, quieren apostar por el uso de tecnologías disruptivas.
- **RNF7** Los datos personales y confidenciales de los usuarios deberán de estar cifrados.
- **RNF6** La autenticación deberá ser realizada mediante el uso de tokens en la cabecera Authorization de la petición HTTPS.
- **RNF7** Las comunicaciones deberán realizarse mediante el uso del protocolo seguro de transferencia de hipertexto HTTPS.
- **RNF8** La aplicación deberá ser duplicativa, permitiendo realizar copias de la misma con distintos parámetros y/o configuraciones.

Cuadro 2: Requisitos no-funcionales de la aplicación

Código	**Descripción**	**Categoría**
RNF1	Mantenimiento del sistema	Mantenibilidad
RNF2	Desarrollo y mantenimiento de bajo coste económico	Económico
RNF3	Fácil comprensión de la aplicación	Mantenibilidad
RNF4	Uso de tecnologías disruptivas	Organizacional
RNF5	Cifrado de datos personales y confidenciales de usuarios	Regulador
RNF6	Autenticación mediante token a través de la cabecera Authorization	Seguridad
RNF7	Las comunicaciones se realizarán mediante el protocolo HTTPS	Seguridad
RNF8	La aplicación deberá ser duplicativa	Producto

3 Arquitectura del sistema

> *«Architecture starts when you carefully put two bricks together. There it begins.»* [van der Rohe, 28/June/1959]

Diseñar una correcta arquitectura otorgará una buena base para el posterior desarrollo de la aplicación.

Según Bass et al. [2012], diseñar y documentar la arquitectura del software tiene tres ventajas:

1. La arquitectura es una presentación de alto nivel del sistema y, por tanto, puede ser usada como centro de discusión por las partes interesadas.

2. El análisis de la arquitectura del sistema en una fase temprana del proyecto puede tener un gran impacto en si el sistema puede cumplir o no los requisitos críticos, como el rendimiento, fiabilidad y mantenibilidad.

3. Un modelo de arquitectura es una descripción manejable y compacta de cómo un sistema está organizado y cómo los componentes interactúan entre sí. La arquitectura de sistema es a menudo la misma para sistemas con requisitos similares y por ello pueden soportar una reutilización de software a gran escala.

Es decir, gracias a las decisiones tomadas a continuación se estará ayudando a la comprensión de la aplicación por las partes interesadas; por ejemplo, por otros desarrolladores en la empresa. Además se podrá comprobar si el sistema cumple o no con los requisitos críticos de la aplicación y, por último, se obtendrá un breve esquema de cómo los componentes se conectan entre sí.

La arquitectura y estructura deben estar en relación con los requisitos no-funcionales de la aplicación, donde en este caso el requisito crítico de la aplicación es la mantenibilidad del sistema, seguido por la seguridad, el rendimiento y la disponibilidad.

Como se ha mencionado anteriormente la arquitectura de sistema es a menudo la misma para requisitos similares, por ello existen una serie de patrones de arquitectura entre los que se podrán escoger los adecuados para este proyecto.

Debido a que se está construyendo una API que soporte servicios web, la solución óptima debería permitir intercambiar datos independientemente de su representación. Además, debe permitir soportar la presentación de los datos de distintas formas. Por ello la opción ganadora en este caso será el Modelo-Vista-Controlador, teniendo en cuenta que habrá que combinarla con aspectos de otros modelos como la arquitectura de repositorio y la arquitectura cliente servidor, que complementen al Modelo-Vista-Controlador (cuyas siglas MVC serán usadas de ahora en adelante) de forma que sea posible cumplir los requisitos del proyecto.

3.1. Posibles modelos de arquitectura

Se pasa a continuación a analizar los posibles modelos y por qué han sido descartados o aceptados para este proyecto:

3.1.1. Modelo-Vista-Controlador (MVC)

El modelo MVC separa la presentación y interacción del sistema de datos. Como su propio nombre indica, este modelo está dividido en tres componentes:

- *Modelo*: El modelo es el encargado de manejar el sistema de gestión de base de datos y de las operaciones asociadas a esos datos. Por ejemplo, si se quiere guardar un usuario en la base de datos, el modelo deberá de tener una función capaz de guardar datos de usuario, y una representación de qué es un usuario.
- *Vista*: La vista se podría definir de forma coloquial como "lo que el usuario ve", es decir, es la representación de los datos obtenidos por el modelo. La mayor parte del tiempo este componente se encargará de transformar los datos en información útil para el usuario. Por ejemplo una vista podría estar compuesta por los recuadros de texto que permiten introducir los datos de registro del usuario.
- *Controlador*: El controlador es el encargado de gestionar la interacción de los usuarios con los datos; es el componente que interactúa de intermediario entre el Modelo y la Vista. Dicho en otras palabras, es el encargado de aportar funcionalidad a las vistas, de forma que el usuario pueda intercambiar información con el sistema. Por ejemplo cuando el usuario termine de introducir los datos de registro, podrá pulsar un botón de envío de dicha información. Ese botón será detectado por el controlador que obtendrá los datos de la vista, los validará y los enviará al modelo para que este los guarde en la base de datos.

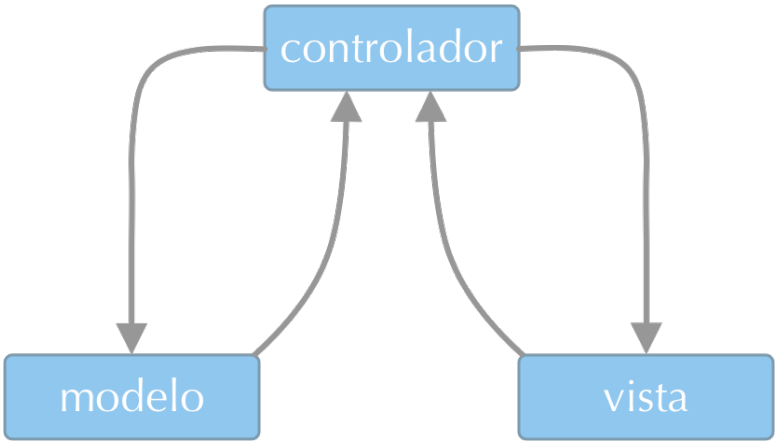

Ilustración 4: Flujo de trabajo de la arquitectura Modelo-Vista-Controlador

Gracias a estos tres componentes bien definidos, cuando se necesite actualizar alguno de ellos, por ejemplo cuando se necesite cambiar la interfaz gráfica que percibe el usuario (la vista), se podrá modificar únicamente el componente *Vista*, actualizándola, y seguir manteniendo la funcionalidad tal y como estaba, ya que para actualizar dicha interfaz, no se necesita cambiar el modelo y el controlador. Esto mismo puede ser aplicado con los otros dos componentes.

3.1.2. Modelo de Arquitectura de Capas

Esta arquitectura consiste en tener un núcleo al que se le van añadiendo capas. Idealmente cada capa debería de tener acceso a las funcionalidades de la capa inmediatamente inferior, y debería de proporcionar funcionalidad a la capa inmediatamente superior. Es una arquitectura muy utilizada cuando se quieren añadir nuevas funcionalidades en sistemas ya existentes, ya que en proyectos con con numerosos equipos esta arquitectura permite asignar un equipo a cada capa.

Otra ventaja es que este sistema permite reemplazar capas enteras, siempre que la interfaz sea mantenida.

Como desventaja principal está que proveer correctamente de una separación entre capas es muchas veces difícil, por lo que no es complicado ver arquitecturas por capas donde una capa tiene que acceder a una capa distinta a la inmediatamente inferior, y proporcionar funcionalidad a otras capas superiores además de la inmediatamente superior.

Este tipo de arquitectura es utilizada en grandes sistemas, como Sistemas Operativos, sistemas autónomos o diseño de sistemas de comunicaciones, donde lo normal es partir de una base sólida, como un sistema anterior, y actualizar las capas necesarias además de añadir las nuevas.

La API que se está desarrollando es nueva, por lo que no se tiene un sistema del que partir. Además, a lo sumo podrían diferenciarse dos o tres niveles:

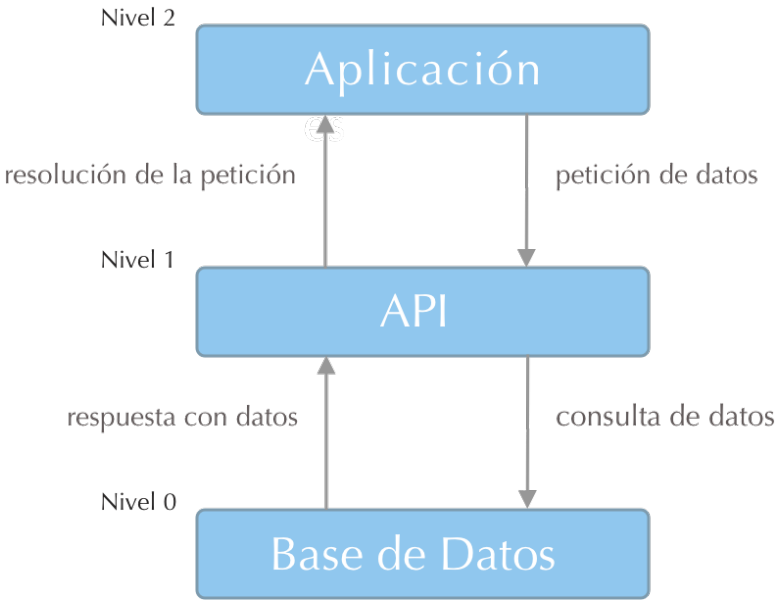

Ilustración 5: Esquema de arquitectura de capas

En este modelo, la capa 3 pertenecería a desarrollos posteriores fuera del alcance de este trabajo.
Aunque se podría considerar que la aplicación podría corresponderse con este modelo se ha decidido seguir optando por el MVC, desechando este modelo por no contemplar la premisa de un sistema ya existente.

3.1.3. Modelo de Arquitectura de Repositorio

Este proyecto también podría considerarse como un modelo de Arquitectura de Repositorio, ya que este modelo se basa en un conjunto de componentes interactuando, que pueden compartir datos.

La idea de este tipo de arquitectura es que todos los datos del sistema se encuentran en un núcleo central que es accesible a todos los componentes del sistema. Los componentes, además, no interactúan directamente entre ellos, sino que para intercambiar datos deberán de enviarlos al núcleo, y este enviarlo al correspondiente sistema.

Como se puede comprobar, la definición de esta arquitectura se asemeja con la del sistema API que se quiere montar. Además, se ha de tener en cuenta que este tipo de arquitectura se suele emplear cuando la información que maneje el sistema tenga que perdurar en el tiempo, como es el caso de la información que será tratada en HousingOn.

Gracias a que los componentes no actúan directamente entre ellos en esta arquitectura, sería posible cambiarlos sin inmutar el funcionamiento del resto. Esto es una clara ventaja para la empresa en un futuro, ya que tendrían posibilidad de cambiar por ejemplo una posible app para móvil, sin por ello tener que volver a adaptar sus páginas web.

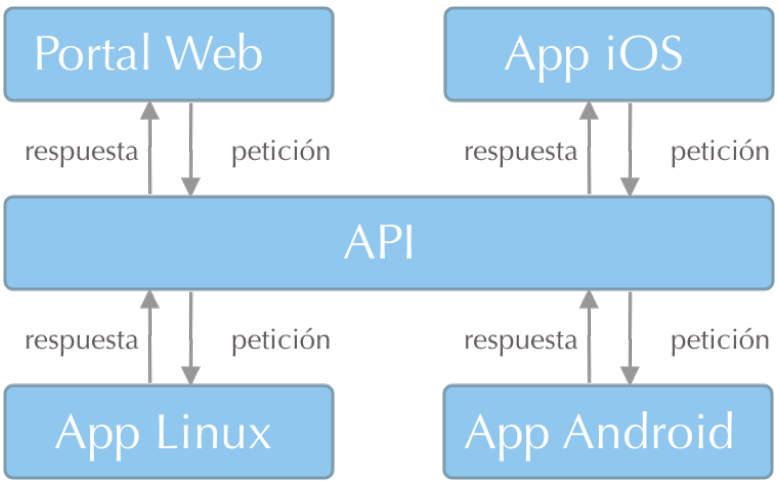

Ilustración 6: Esquema de arquitectura de Repositorio

Como contrapartida, este sistema tiene una clara desventaja, y es que en caso de fallar el repositorio, este afectará a todos los sistemas. Además, en caso de querer mitigar parte de este problema, o en caso de que el volumen de uso del servicio aumentase, organizar este repositorio en un sistema distribuido también es una tarea cuya complejidad se debe a los numerosos procesos de coordinación y sincronización que caracterizan a estos sistemas, además de las complicaciones derivadas de detectar fallos en las comunicaciones que puedan ocurrir.

3.1.4. Arquitectura Cliente-Servidor

Dada la naturaleza del proyecto, se puede deducir que la API a crear también tiene cierta correspondencia con este modelo.

Esta arquitectura se basa en un sistema distribuido en el que hay uno o más servidores que dan servicios a una serie de clientes. Este modelo es utilizado cuando una base de datos compartida debe ser accedida desde distintos puntos de acceso. Además, en caso de crecimiento, los servidores podrían ser replicados para dar servicio a todos los clientes.

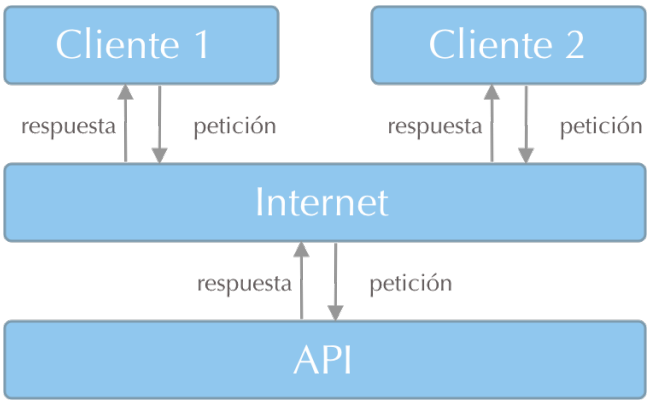

Ilustración 7: Esquema de arquitectura de Cliente-Servidor

La principal desventaja es que cada servicio es un posible punto de fallo, y es susceptible de ataques de denegación de servicios (ataques DoS por sus siglas en inglés, *denial-of-service*).

3.1.5. Arquitectura de Flujo de Datos

Esta arquitectura se basa en la idea de tuberías y filtros, donde cada componente del sistema se correspondería con una tubería, por la que se reciben los datos y se les aplican las funciones necesarias para procesar los datos. La principal ventaja de este tipo de arquitectura es que es fácil de comprender, además la idea del flujo pasando por las tuberías se corresponde con la estructura de muchos procesos de negocios, lo que facilita la comprensión por todas las partes interesadas.

Debido a las características de este modelo, el formato para la transferencia de datos debe estar en concordancia entre los distintos filtros y tuberías, de forma que todas las transformaciones que se realicen deberán analizar la entrada y adaptarla, y adaptar la salida. Esto genera un sobrecoste en las operaciones del sistema que puede afectar al rendimiento.

En el caso de la API de HousingOn este tipo de arquitectura sería inviable para ser usada por todo el proyecto, aunque habría muchos componentes que se podrían tratar con ella.

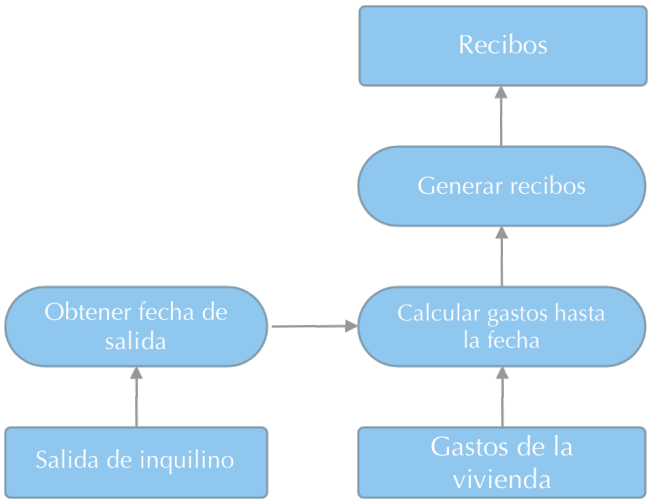

Ilustración 8: Flujo para generar recibo por salida de inquilino

3.2. Arquitectura de la aplicación

Además de estos patrones de arquitectura, se puede hablar de la arquitectura de aplicación, que nos permitirá clasificar el tipo de aplicación que se está desarrollando.

Según Sommerville [2016], se pueden englobar todas las aplicaciones en dos tipos de categorías:

1. Aplicaciones de procesamiento de transacciones: Este tipo de aplicación se centra en dar funcionalidad a una base de datos, es decir, aplicaciones en las que los usuarios pedirán y actualizarán datos. Con esta definición se puede englobar a la mayoría de aplicaciones del mercado, ya que se podrían incluir sistemas de información (blogs, periódicos, portales de empresas), redes sociales, reservas, bancos, compras online...

2. Sistemas de procesamiento de lenguajes: este tipo de sistemas engloba aquellos en los que el usuario, para conseguir su objetivo, deberá expresarse mediante lenguajes formales. Por ejemplo, los navegadores web deben interpretar HTML para mostrar una página; por lo tanto un programador deberá crear la página en HTML si quiere que el navegador lo interprete. También se puede pensar en un intérprete de Python, que tendrá como dato de entrada un programa de Python.

Con estas definiciones se podría decir que la API que se está construyendo tiene muchas características del primer grupo, ya que la API se dedicará a dar funcionalidad a la base de datos, permitiendo la actualización y petición de datos, aunque también se pretende incorporar características del segundo grupo, ya que las salidas de la API serán principalmente en formato JSON, dejando la posibilidad de que se pueda crear un analizador sintáctico para el usuario final.

Tras describir los distintos modelos en los que se podía basar este proyecto, se procede a mostrar el esquema de directorios donde se puede apreciar cómo se ha organizado el proyecto, y ver las similitudes tomadas con la arquitectura Modelo-Vista-Controlador:

```
housingon-api
  app
    controllers
      api
       welcome.py
      ...
       user.py
       __init__.py
      modules
       users
         mutations
           LoginUser.py
           ModifyUser.py
           ModifyUsers.py
           RegisterUser.py
           __init__.py
         queries
           Me.py
           __init__.py
         users.py
       welcomes
          welcomes.py
         __init__.py
    models
      Agreement.py
      ...
      Welcome.py
      __init__.py
    __init__.py
    email.py
    errors.py
    schema.py
    utils.py
  logs
    housingon.log
  .env
  Dockerfile
  README.md
  boot.sh
  config.py
  docker-compose.yml
  manage.py
  requirements.txt
  tests.py
```

Como se puede ver en el esquema de directorios, en los subdirectorios de *app* se encuentran sus componentes, dos términos ya conocidos: *controllers* y *models*. Faltaría un tercero, *views*, que no existe en el proyecto, puesto que la API es la encargada de enviar y recibir los datos mediante los controladores. La vista se encuentra en las aplicaciones clientes de la API.

Si se quiere trabajar con la funcionalidad de la aplicación se debe acudir al controlador correspondiente. En cambio, si se quiere modificar la estructura de la base de datos, ya sea para crear nuevas tablas o editar alguna de las anteriores, se deberá hacer en los modelos.

A la lógica de esta organización se debe añadir que los archivos internos de los directorios están nombrados para permitir una rápida identificación de los componentes. El modelo de bienvenidas se llama **Welcome.py**, por ejemplo. Los controladores de ambas APIs, tanto la RESTful API como la GraphQL, están en el directorio *api* y el directorio *modules*, respectivamente. Esos nombres se han escogido por ser nombres de directorios comunes en estos tipos de API.

Como se puede ver, el directorio *app* tiene cinco componentes además de las previamente mencionadas (__init__.py, email.py, errors.py, schema.py, utils.py), que constituyen funcionalidades internas de la API. Dotan a la API de capacidades como enviar correos electrónicos, lanzar errores personalizados o convertir los datos recibidos mediante GraphQL en un diccionario de Python. Los archivos están en esa ruta para poder ser empleados por la aplicación desde distintos controladores sin necesidad de repetir código, aumentando así su reusabilidad.

Por último, fuera del directorio *app* se encuentran otros archivos de configuración, un archivo para realizar tests (tests.py), dos archivos de configuración para Docker (Dockerfile y docker-compose.yml), *scripts*, un archivo con los requerimientos de librerías a utilizar (requirements.txt), etc. Estos archivos no involucran directamente la propia funcionalidad de la aplicación, y se encuentran en esa ruta para su fácil acceso, en caso de que se quiera, por ejemplo, cambiar la configuración de las contraseñas o el puerto de ejecución.

Se ha adoptado este sistema, además, porque refleja correctamente el servicio cliente-servidor y, como este sistema es susceptible a diversos ataques, tener las configuraciones en la raíz del proyecto facilita el cambio momentáneo de contraseñas, ganando tiempo en la caída de los servicios en caso de enfrentarse a un parón ocasionado en el sistema.

Por otra parte es más fácil llevar el control de los permisos de lectura y escritura del Sistema Operativo sobre los componentes del proyecto, si los archivos a los que no deberían de acceder los clientes están a un nivel superior de los que puedan acceder.

Este esquema de directorios se ha desarrollado desde cero, diseñado específicamente para la aplicación. Esto se debe a que, como se menciona en el capítulo cuatro, la aplicación se ha desarrollado en Flask, un *framework* que otorga mucha libertad a sus usuarios a la hora de organizar las aplicaciones.

El único requisito de Flask es que, para su uso en un proyecto, se ha de crear un archivo en la raíz que contenga las siguientes líneas:

```
from flask import Flask

app = Flask(__name__)
```

Una vez creado ese fichero en la raíz e introducidas las líneas mencionadas, el resto de ficheros y directorios podrán ser creados por el desarrollador con fines organizativos.

4 Tecnologías usadas

«If all you have is a hammer, everything looks like a nail» [Maslow, 1966]

Tratando de evitar la situación descrita anteriormente por Abraham Maslow, aparte de los conocimientos sobre los requisitos del proyecto y arquitectura de la aplicación será necesario contar con una variedad de herramientas que faciliten su desarrollo.

A continuación se encuentran listadas por orden alfabético las herramientas utilizadas en este proyecto:

Alembic: es una herramienta de migración de datos creada por el autor de SQLAlchemy (tecnología que se menciona posteriormente). Con esta herramienta se puede llevar el control de qué cambios se han de realizar en la base de datos, en caso de alterar los modelos de la aplicación. Esta tecnología se usa en el proyecto porque se quieren evitar problemas de independencia lógica de los datos, es decir, que si se edita alguna columna en la base de datos la información de esta columna se siga preservando correctamente. La intención con todo ello es cumplir un requisito de la empresa: evitar la perdida de información y aumentar las opciones de adaptabilidad de la empresa a nuevos cambios.

Docker: software libre de código abierto que proporciona una capa de abstracción adicional y virtualización para aplicaciones en distintos sistemas operativos.

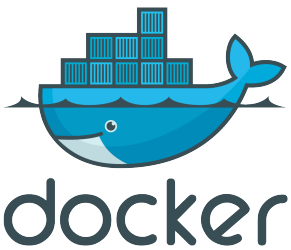

Ilustración 9: Logotipo de Docker.

Según Wikipedia, Docker utiliza "el soporte del kernel Linux para los espacios de nombres aísla la vista que tiene una aplicación de su entorno operativo, incluyendo árboles de proceso, red, ID de usuario y sistemas de archivos montados, mientras que los cgroups del kernel proporcionan aislamiento de recursos, incluyendo la CPU, la memoria, el bloque de E/S y de la red".

Los contenedores que se creen con el software de Docker serán aprovechados con diversas funciones, ya que, gracias a ellos, se podrá replicar la aplicación para distintas empresas. Además, se podrá escalar el número de servidores que atienden peticiones en caso de un gran aumento, gracias al uso de servicios de administración de contenedores en *cloud* como el de Azure [1]

[1] https://azure.microsoft.com/es-es/services/kubernetes-service/docker/

Permite compartir el proyecto con otros desarrolladores sin que estos tengan que invertir tiempo en ajustar el proyecto en sus equipos, ya que, teniendo Docker, solo necesitarán ejecutar el contenedor para empezar a trabajar.

Actualmente, la empresa sigue probando distintos servicios de alojamiento web, así como API y alojamiento en la nube, para encontrar el que más les convenza para establecerse. Por lo tanto, la portabilidad de la aplicación es un requisito no funcional importante para la empresa y justifica la elección de Docker en el proyecto.

Flask: es un *framework* ligero para Python que permite la creación de aplicaciones web rápidamente.

Ilustración 10: Logotipo de Flask.

Al elegir Python como lenguaje de programación para elaborar la API, había cuatro *frameworks* importantes entre los que escoger: Django, Flask, Pyramid y Bottle. De esos cuatro, dos fueron descartados inmediatamente por tener comunidades minoritarias en comparación con las otras dos; por lo tanto, aún había que escoger entre dos importantes *frameworks* de creación de aplicaciones web: Django y Flask. Para poder escoger el adecuado entre estos dos se buscó información y se realizó una comparación.

Django resultó ser un *framework* más completo. Cuenta con muchísimas funcionalidades, una estructura predefinida para el desarrollo de la aplicación, un sistema de internacionalización y localización, sistemas de seguridad integrados, un sistema para el uso de mapas geográficos...

Aunque estos parecen en un principio motivos para decantarse por Django, eran en realidad detrimentos, puesto que lo convierten en un *framework* más pesado y lento que Flask. Además, al ser un sistema cerrado, favorece la creación de aplicaciones web clásicas, pero dificulta la creación de APIs, al contrario que Flask.

Por ello, Flask era mejor opción. Cuenta también con una documentación más simple, y una curva de aprendizaje menor que la de Django. Ambas características encajan con los requisitos no funcionales de la empresa, ya que futuros desarrolladores deberán estudiar las tecnologías escogidas por su cuenta antes de comenzar a trabajar en la plataforma.

Git: creado por Linus Torvalds (también conocido por ser el desarrollador del kernel Linux), junto con Subversion (SVN) es uno de los principales sistemas de versiones que existen en la actualidad. Git es multiplataforma, y fácil de instalar en los S.O. que no lo traen de serie. Este software de control cuenta con una sintaxis básica pero no por ello carente de funcionalidades.

Además, el usuario se adapta a él con facilidad, ya que existen numerosas interfaces que le ayudan a llevar el control de los cambios de una forma gráfica.

Ilustración 11: Logotipo de git.

Gracias a esta herramienta se ha podido avanzar y evitar errores en todo momento, ya que en caso de error, existía el respaldo de un *commit* anterior al que poder retroceder. Además, aunque no se ha dado el caso, con Git se hubieran podido crear ramas para poder programar distintas funcionalidades sin que una afectase a la otra. Es una herramienta que, combinada con un repositorio en red (como el que se menciona a continuación), se vuelve extremadamente útil para el desarrollo en equipos, donde cada equipo puede llevar sus cambios en su propia rama hasta que sean funcionales y juntar las ramas al terminar, recopilando todas las funcionalidades en una única versión final.

Sin un software de control de versiones, el tiempo que se requeriría para solventar los errores introducidos durante el desarrollo colaborativo llevarían periodos de tiempo más elevados que los actuales. Un ejemplo donde el uso de git tiene un impacto crítico es el kernel de Linux.

En este caso se ha utilizado git en vez de Subversion u otros software de control de versiones, porque ya se contaba con experiencia previa con esta herramienta. La mayoría de software de control de versiones son similares en cuanto a características para la gran mayoría de usuarios, incluso las instrucciones son parecidas, por lo que realmente usar una o otra no supone una gran diferencia. Si bien es cierto que a nivel experto se pueden encontrar, por ejemplo en el manejo de Hooks, no es normal que en un proyecto como este se necesite usar ese tipo de herramientas avanzadas.

Otro punto a favor para el uso de git es la cantidad de herramientas e integraciones que existen de este sistema, desde sistemas operativos que ya traen de serie git integrado con ellos, hasta servicios en la nube que permiten subir la aplicación a través del comando de git, *git push*. Como ya se ha mencionado anteriormente, uno de los requisitos no funcionales de la aplicación es la portabilidad, y gracias a git el proyecto puede ser fácilmente portado de un sistema a otro, aunque a diferencia de Docker habrá que reconfigurarlo para el nuevo sistema.

Además, al usar git, se puede hacer uso de una gran plataforma como GitHub, para poder almacenar el repositorio y llevar el control sobre él (abrir y cerrar incidencias, asignar elementos a un usuario, etc.), de forma que el trabajo en el proyecto pueda ser repartido y coordinado.

GitHub: es una plataforma web que permite alojar repositorios Git. Actualmente la empresa propietaria de esta plataforma es Microsoft, y aunque la idea principal de GitHub es publicar tus repositorios de forma pública, también existe la opción de hacerlo de forma privada. La interfaz web aparenta ser básica pero es muy completa, contando con muchísimas funciones, como herramientas para poder colaborar, tener un perfil de usuario, e incluso realizar cambios en los archivos de un proyecto desde ella.

Ilustración 12: Logotipo de GitHub. Este logo es muy conocido por su mascota Octocat.

Al igual que git, se decidió emplear esta herramienta debido a la experiencia previa con ella. Además, tras hablar con el cliente, se decidió que era preferible alojar los repositorios en un servicio externo antes que en el propio servidor de la empresa, ya que en caso de fallo del mismo sería más complicado recuperarlo. De esta forma se refuerza el requisito no funcional que tiene la empresa sobre

la mantenibilidad del código y se reduce la amenaza ante una posible perdida del código del proyecto, mejorando así el sistema de gestión de riesgos.

Como ya se ha adelantado en el apartado de git, GitHub además permitirá coordinar el proyecto entre otros posibles programadores.

Gmail: es el servicio de correo electrónico ofrecido por Google. Su beta fue lanzada en 2004 y la versión final en 2009. En 2012 consiguió superar en número de usuarios a Microsoft con su servicio de correo electrónico Outlook y desde entonces es el servicio de correo electrónico más popular.

Ilustración 13: Logotipo de Gmail.

HousingOn lleva usando sus servicios desde la fundación de la empresa. Por ello, se ha utilizado Gmail para este proyecto, tanto para la comunicación con la empresa como para el servicio de envío de correos electrónicos de la aplicación. Se ha escogido solo por un requisito no funcional organizativo, ya que es un procedimiento existente.

GraphQL: creado por Facebook y de código libre, GraphQL es un lenguaje de consulta de datos utilizado para manipular APIs. Este lenguaje permite a los clientes de la API definir sus propias estructuras de datos, y el servidor adaptará sus respuestas a estas estructuras. Para poder usar este lenguaje a su vez hay que instalar en la API un servidor de GraphQL y en los clientes un cliente de GraphQL.

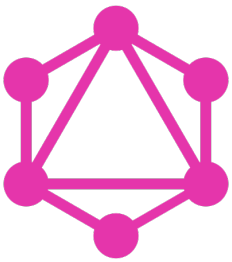

Ilustración 14: Logotipo de GraphQL.

Este lenguaje podría ser el próximo estándar en cuanto a APIs web se refiere debido a sus grandes ventajas. Ahora mismo es costoso en tiempo desarrollar APIs con este lenguaje, puesto que los *frameworks* y la información que se pueden encontrar están algo limitados. Actualmente, se está desarrollando una RESTful API, pero el empleo de la innovadora tecnología de GraphQL podría conseguirle a la empresa una ventaja respecto a sus competidores en un futuro.

HousingOn es de la opinión que una buena interfaz gráfica ayuda a aumentar los beneficios de la web. Para ello es necesario crear test A/B e ir modificando y ajustando la interfaz gráfica. Poder ajustar también los datos que se usan en cada vista cambiando simplemente la consulta puede agilizar considerablemente la realización de estas mejoras.

Graphene-Python: es la principal librería para implementar la API GraphQL en Python. Aunque la librería lleva poco tiempo existiendo (la primera versión publicada es de octubre de 2015), el proyecto ya va por la segunda versión y están preparando la tercera.

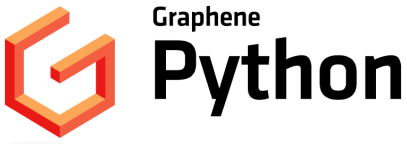

Ilustración 15: Logotipo de Graphene.

Graphene está dividido en 3 integraciones, según el backend que estemos usando:

- Graphene-Django: para los usuarios que quieran implementar GraphQL en Django. Es importante recordar que Django tenía un sistema bastante cerrado a diferencia de Flask, por eso Graphene cuenta con una versión única para Django que facilita la implementación.
- Graphene-SQLAlchemy: esta es la integración que usarán la mayoría de usuarios que quieran implementar Graphene en Python, puesto que SQLAlchemy, utilidad que usa el proyecto y por tanto está descrita posteriormente, se usa en la mayoría de proyectos de Python que requieren conexiones a bases de datos.
- Graphene-GAE: la última de las integraciones está creada para poder conectar con los servicios en la nube de Google, en concreto para su servicio de Google App Engine, que permite la ejecución de estas aplicaciones en la infraestructura de Google.

El uso de Graphene facilita la inclusión del servidor GraphQL en el sistema. Con el uso de Graphene se podrá programar la funcionalidad de la aplicación con menor complicación, y el mantenimiento del sistema será más sencillo para la empresa, alineándose esta circunstancia con los requisitos no funcionales.

Gunicorn: servidor HTTP para aplicaciones web desarrolladas en Python, soporta nativamente WSGI, web2py, Django y Paster, lo que hace que sea ideal para este proyecto Flask, ya que ambos están basados en la especificación WSGI[2].

Ilustración 16: Logotipo de Gunicorn.

Gunicorn entre otras tareas se encargará de manejar cada hilo de ejecución de la aplicación, según las peticiones recibidas.

La empresa usa actualmente Apache, porque la aplicación actual está basada en PHP. No obstante, este servidor de páginas web no es una solución factible para este proyecto, ya que aunque Apache soporta la ejecución de aplicaciones Python, el módulo que permitía dicha ejecución ha sido discontinuado, por lo que no cumple ciertos requisitos de la empresa, como que el proyecto pueda seguir siendo mantenido y actualizado.

El resto de posibles servidores de aplicaciones web en Python fueron descartadas por el bajo número de usuarios en su comunidad en comparación con gunicorn, como uWSGI, que las comparativas de rendimiento lo sitúan como un servidor más rápido que gunicorn, pero la comunidad llega a ser entre

[2]Web Server Gateway Interface

dos y tres veces menor que la del servidor escogido. Ocean, una gran empresa de servicios en la nube hace un resumen las virtudes y desventajas de los distintos servidores para Python.

Por todo esto el que más se ajusta a los requisitos no funcionales de la empresa es gunicorn, con un servidor más sencillo de configurar y con una tecnología más testada debido a su mayor comunidad.

JSON: es un formato de texto sencillo utilizado esencialmente para el intercambio de datos entre aplicaciones. Surge como alternativa a XML. La principal ventaja de JSON con respecto a XML es la facilidad que ofrece para implementar analizadores sintácticos, lo que permite la extracción e inserción de datos en este formato de una forma rápida y precisa.

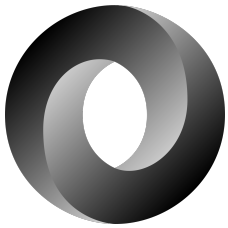

Ilustración 17: Logotipo de JSON.

JSON será el formato escogido para devolver datos mediante el sistema RESTful API.

Este formato ha sido escogido frente a XML por contar con las siguientes ventajas:

- El formato es sumamente simple, al contrario que en XML, que resulta bastante complejo. Esto produce una facilidad para la comprehensión del documento así como una facilidad para construir analizadores.
- La tecnología de hoy en día es capaz de procesar JSON mucho más rápido que XML, ya que la mayoría de lenguajes son capaces de hacer una traducción directa de JSON a una estructura de datos de tipo diccionario.
- El tamaño de un archivo JSON es mucho menor que la misma información en un archivo XML. En concreto puede llegar a ser un 30 % menor, lo cual optimiza los intercambios de información. Esto es ideal si los futuros usuarios utilizan conexiones lentas de internet.

MariaDB: es un fork de MySQL, por lo tanto, al igual que este, es un sistema gestor de bases de datos con licencia GPL. Esta herramienta surge tras la compra de MySQL por parte de Oracle, con la intención de privatizarlo y vender licencias. MariaDB es gratuito y gracias a su gran comunidad, ha mejorado tanto que se ha terminado convirtiendo en uno de los mejores sistemas de bases de datos relacionales.

Ilustración 18: Logotipo de MariaDB.

Por el tipo de requisitos el proyecto debía de contar con una base de datos relacional, debido a la conexión entre datos que debían de tener, las transacciones que se debían realizar... por tanto las

bases de datos no relacionales quedaban descartadas. Como otro de los requisitos no funcionales es realizar un proyecto de bajo coste monetario, con la intención de no elevar los costes de la empresa, se necesitaba de una herramienta a poder ser gratuita.

Como además ya se había utilizado MySQL anteriormente en la empresa, el cambio a MariaDB no suponía un gran coste, ya que al ser un fork siguen asemejándose en gran medida, y esto concuerda de nuevo con los requisitos organizativos de la empresa.

Passlib: librería con funciones hash para Python, cuenta con un total de 30 algoritmos de hash para contraseñas. Esta librería será la encargada de proporcionar la función de encriptado y comprobación de contraseñas, de forma que la base de datos cuente con todas las contraseñas de usuario correctamente encriptadas. Gracias a esta librería se intenta solventar una necesidad externa y es que, según el documento REGLAMENTO (UE) 2016/679 del Parlamento Europeo [2016][83], las contraseñas de usuario deben de estar cifradas en la base de datos para garantizar un nivel de seguridad adecuado para evitar «*la comunicación o acceso no autorizados a dichos datos, susceptibles en particular de ocasionar daños y perjuicios físicos, materiales o inmateriales*».

Python: es un lenguaje de programación interpretado, multiplataforma y orientado a objetos. Actualmente es uno de los lenguajes más populares en programación y en estos momentos se encuentra en la versión 3 aunque el lenguaje surgió en 1991. Este hecho, y que Python insiste en la importancia de la legibilidad del código, hace de este lenguaje el ideal para personas que se están iniciando en la programación, incluyendo profesionales en distintos campos como la física o las matemáticas, que saben que no necesitarán actualizar sus conocimientos cada año, porque las versiones de este son longevas.

Ilustración 19: Logotipo de Python.

Este lenguaje se ha escogido por una preferencia personal. Habría sido posible escoger otros, como PHP, `C#`, Java o Javascript, pero se decidió programar en Python porque apenas requiere de preparación previa en el entorno de desarrollo, y es un lenguaje de código libre con licencia GNU, por lo que estará disponible para toda la comunidad mientras sea mantenido.

Al resultar un lenguaje legible, ayuda a que otros trabajadores puedan entender el funcionamiento de la aplicación, y adaptarse a ella rápidamente. En otros lenguajes, como Java, se hubiera vuelto un proceso mucho más lento.

Si bien es cierto que Python no es el lenguaje más rápido con respecto al tiempo de ejecución, se ha considerado que esto no influye realmente en la decisión, ya que hay otros aspectos que pueden producir mayores variaciones en la velocidad de ejecución, como puede ser la localización del servidor con respecto al cliente. Además, la programación en Python es fácilmente comprensible, y eso facilita su mantenimiento.

En un futuro a HousingOn le encantaría poder aplicar business intelligence, y para ello hay grandes herramientas de análisis de datos programadas para ser usadas en este lenguaje, como Pandas. Además, es fácil encontrar mucha información en internet sobre este tema en relación con Python, ya que este lenguaje está entre los favoritos de los matemáticos.

SQLAlchemy: ORM (Object-Relational mapping) de código abierto creado para su uso en Python. Como se mencionaba anteriormente en el apartado de graphene-python, este ORM es el más usado en la comunidad de Python para bases de datos relacionales.

Ilustración 20: Logotipo de SQLALchemy.

Este ORM será el encargado de convertir las clases y objetos de Python en tablas y datos. A grosso modo, SQLAlchemy relaciona los modelos creados en Python en la arquitectura Modelo-Vista-Controlador con las tablas de la base de datos.

Gracias a esta herramienta, y a Alembic, mencionada anteriormente, se pueden crear modelos en la aplicación, y generar la base de datos a partir de estos. Ello permite mejorar el mantenimiento de la aplicación, además de producir un ahorro en el coste temporal de desarrollo. También se pueden realizar cambios en los modelos, que ajustan automáticamente la base de datos, reduciendo el riesgo de errores y evitando perjudicar las estructuras y los datos almacenados en la base de datos.

Virtual Environment: esta funcionalidad de Python permite aislar los paquetes y versiones de una aplicación de forma que por cada *virtual environment* se pueda usar una versión de Python distinta, o instalar una versión concreta de los paquetes en cada proyecto. Gracias a este sistema posteriormente se puede extraer un archivo con todos los paquetes utilizados en esa aplicación, sin que por ello se sumen paquetes de la librería estándar de Python o de otros proyectos, paquetes que no han sido utilizados o de versiones que no tienen por qué corresponderse con la instalada en el proyecto. Todo ello permite cumplir el requisito de mantenimiento, ya que se puede saber en todo momento qué software está instalado y en qué versión se encuentra.

Si anteriormente no se han usado *virtual environments* en Python, el comienzo puede resultar arduo, ya que ha de crearse un proyecto, y recordar activar el entorno cada vez que se utilice ese proyecto. A la larga, sin embargo, se agradecen sus ventajas. Sin él, la portabilidad del proyecto a otros sistemas aumentaría en dificultad, ya que se tendría que mantener un registro de los paquetes y versiones empleados en el proyecto para poder instalarlos manualmente en el servidor posteriormente.

Visual Studio Code: (abreviado VSC) es un editor de codigo abierto creado por Microsoft y multiplataforma fue lanzado en 2015 y hoy en día es uno de los editores de código más usados del mundo. Como curiosidad, el editor está basado en tecnologías web, por lo que, aunque el consumo de recursos es más elevado, la herramienta destaca por su agilidad, diseño y capacidad de adaptarse a todo tipo de desarrollo. Cuenta con una gran ventaja, que es la posibilidad de instalar complementos, para ajustar este editor a las necesidades del usuario.

Ilustración 21: Logotipo de Visual Studio Code.

Se puede emplear este editor si, por ejemplo, se está desarrollando una interfaz gráfica de una web, o para hacer debugging de programas en Python. Para este segundo cometido existen herramientas más específicas, como PyCharm o Visual Studio, pero suelen ser más pesadas, consumiendo mucho tiempo y recursos. Para un proyecto pequeño, la diferencia de tiempo entre el inicio de cualquiera de estas dos herramientas y el inicio de Visual Studio Code es considerable, llegando a haber diferencias de minutos, en función del hardware empleado.

Si se sigue la configuración recomendada al instalar la herramienta, se obtienen ventajas como arrancar el programa desde una shell, escribiendo únicamente *code* y la ruta en la que iniciar el espacio de trabajo.

También tiene atajos de teclado, que favorece la agilidad de programación, cuenta con un manejo de git integrado, *intellisense* para ayudar con la escritura de código, resaltado de sintaxis errónea y otra serie de herramientas que suelen ser comunes en entornos de desarrollo, pero son más difíciles de encontrar en editores de código.

Con él se ha podido manejar todo el desarrollo de la API, realizar el Debugging, llevar el control de versiones, ejecutar las imágenes de Docker, usar la Shell para la ejecución de comandos...

Como se ha mencionado anteriormente, esta herramienta puede resultar más rápida que otras algo más específicas, ahorrando en coste temporal de desarrollo. Además, es gratuita, y por tanto ayuda a la empresa con su requisito no funcional de mantener el proyecto con el menor coste posible.

5 Desarrollo y Testing

«Quality is a product of a conflict between programmers and testers» [Bugayenko, 2018]

5.1. Desarrollo

De acuerdo con Sommerville [2016], en grandes proyectos de software no es común ceñirse a un solo método de desarrollo. En este caso, el método mayoritario ha sido el desarrollo en cascada. Es uno de los métodos más antiguos, y como curiosidad es uno de los más odiados por los desarrolladores, debido a la gran cantidad de documentación que produce.

El desarrollo del software se divide en una serie de fases: definición de requisitos, sistema y diseño de software, implementación y test unitarios, integración y test del sistema y, por último, operación y mantenimiento.

El proyecto comienza por la definición de requisitos, y no se puede pasar a la siguiente fase antes de haber acabado la anterior. Al finalizar una fase, se debe elaborar la documentación necesaria para seguir adelante con el proyecto. La idea base de este modelo consiste en conseguir recorrer todas las fases sin encontrar problemas. Si se encuentran, se ha de retroceder las fases necesarias para solucionarlos, reescribiendo la documentación de cada una de ellas para incluir los cambios y las soluciones implementadas.

Una de sus mayores ventajas es que el software desarrollado puede ser de mayor calidad y seguridad, debido a la cantidad de documentación generada en el proceso.

Uno de los mayores inconvenientes es que la estructura del proceso implica un mayor tiempo de desarrollo con respecto a otros métodos.

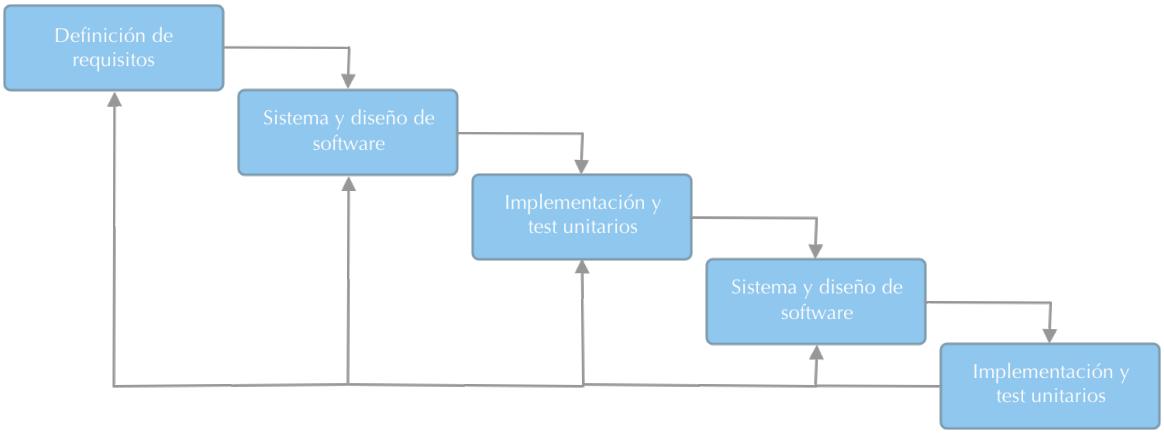

Ilustración 22: Modelo de desarrollo en cascada.

Además de este método de desarrollo, este proyecto también comparte algunas características con otros. Del desarrollo incremental se puede destacar que, aunque hay uno inicial fijado, el desarrollo posterior puede depender del progreso y prioridades del cliente. Del desarrollo de integración y configuración se puede destacar que se reutiliza parte del software.

Para poder empezar el proyecto, se necesitaba realizar una configuración mínima funcional de la aplicación, de forma que se pudiera comprobar que todas las características que se fuesen a usar posteriormente estuvieran correctamente funcionando. Entre los elementos de la configuración mínima se encontraban la posibilidad de establecer conexión con la base de datos, la posibilidad de poder realizar *tests*, poder realizar envíos de emails, poder crear algún hilo de ejecución para que tareas más pesadas, como mandar un email, pudiesen ejecutarse en un segundo plano sin detener la ejecución de otras funciones en el hilo principal, poder crear una instancia *Docker* con el proyecto funcionando en él y otra serie de funciones minoritarias.

Conseguir finalizar esta configuración mínima llevó dos semanas. Aunque en muchos casos solamente era necesario añadir un par de líneas al proyecto, era necesario saber que realizar llamadas a funciones, ajustar su configuración y compaginar las herramientas para que desde el archivo *Dockerfile* (introducido en 3.2) se pudiera definir el correo o la base de datos a usar daba lugar a gran cantidad de errores.

Lo más complicado durante el desarrollo fue el análisis de la base de datos actual, que carece de relaciones y estandarización. Además, las tablas contenían numerosos campos que jamás habían sido utilizados y que carecían de sentido en la aplicación. La base de datos actual tiene 91 tablas con algunas de las tablas conteniendo hasta ciento doce campos. Además, solo había una relación en toda la base de datos. Esto producía que la base de datos fuese muy compleja de mantener.

Crear la nueva base de datos ha llevado mucho tiempo, debido a la estandarización del nombre de tablas y campos, y a la eliminación de campos innecesarios y creación de nuevas relaciones. Esta tarea puede resultar ardua, ya que los resultados no se ven hasta haberla completado. Al finalizar, se puede crear una imagen que muestre las tablas y sus relaciones, como se puede observar en la ilustración 23. Crear una imagen así de la base de datos de partida, sin embargo, ha resultado imposible, pues no se ha encontrado un programa capaz de producir un gráfico oportuno para este documento. La nueva base de datos sí puede representarse porque se ha reducido tanto el número de tablas como el tamaño de las mismas, contando con treinta tablas, de las cuales las más grandes tienen unos veinte campos.

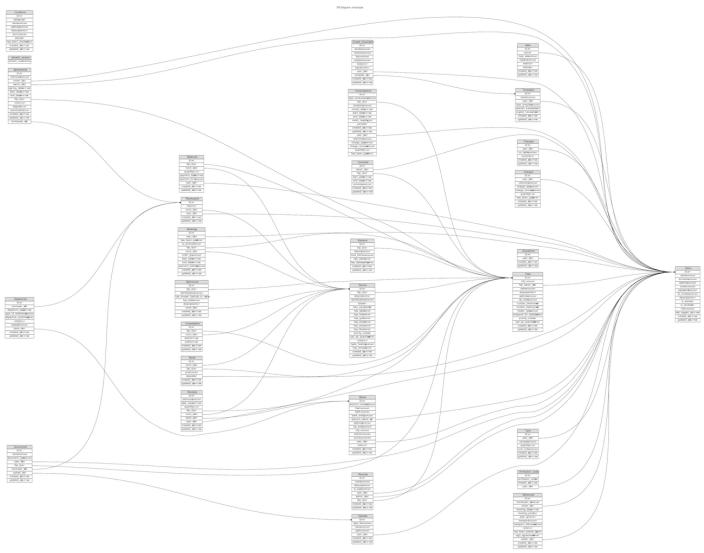

Ilustración 23: Nueva Base de Datos creada para la API

Para el desarrollo de la base de datos, como ya se ha mencionado anteriormente, primero se ha procedido a la creación de los modelos en Python, es decir, con el editor Visual Studio Code se han ido creando nuevos archivos con el nombre del modelo.

Gracias a SQLAlchemy, crear un modelo es tan simple como crear una clase de Python describiendo cada atributo de la clase con una clase de SQLAlchemy.

Una vez el modelo se ha completado (ejemplo de modelo A.3), se ha procedido a utilizar Alembic para guardar los cambios de la base de datos. Tras revisar los resultados de los cambios generados por Alembic, si estos eran correctos se procedía a utilizar Alembic para aplicar los cambios en la base de datos. Si los cambios no eran correctos, se procedía a corregirlos, para posteriormente repetir la comprobación hasta que estos fueran correctos, y poder aplicar los cambios en la base de datos utilizando Alembic.

Tras cada modelo desarrollado se ha realizado un test que realiza la comprobación de ese modelo. Ese test consiste en la creación de un objeto Python de ese modelo (hay que recordar que gracias a SQLAlchemy los modelos son clases de Python); al modelo se le pasan algunos valores para los atributos, y posteriormente se realiza un intento de guardado en la base de datos. Si al guardar no se producen errores, la creación del modelo es exitosa, por lo que se puede proceder al siguiente.

A veces estos pasos no se han podido realizar así, puesto que hay modelos que necesitan relacionarse con otros modelos que puede que aún no hayan sido desarrollados. Por ello, en este caso se han comentado los atributos que relacionan el modelo desarrollado con el que aún no ha sido desarrollado, pasando el test por primera vez con dichas relaciones comentadas, y tras la creación del modelo restante se ha procedido a realizar de nuevo el test, esta vez con las relaciones descomentadas. Además con cada cambio se han actualizando de nuevo con Alembic dichos cambios en la base de datos.

Como la empresa no tenía preferencia por qué API desarrollar primero, se decidió comenzar por la más innovadora: la API GraphQL. Esta no fue una buena decisión, puesto que cada paso dado en esta API llevaba días. Además, resultaba bastante desmoralizador no poder comprobar los resultados del esfuerzo hasta que ese paso estuviera terminado. Tratando de relacionar modelos entre sí, se encontró un fallo en Graphene-Python, con la interpretación de alguna de las nuevas relaciones, ya que esta tecnología no está preparada para un uso intensivo.

Para tratar de enmendar el error y poder avanzar de nuevo, se comenzó el desarrollo de la RESTful API, que beneficia tanto al proyecto como a la implementación de los clientes de la API, puesto que para la mayoría de lenguajes, las herramientas de GraphQL siguen siendo algo primitivas y costosas de implementar.

Realizar la RESTful API fue más satisfactorio, ya que se creaba funcionalidad más rápidamente, y los avances eran visibles visualmente, ya que se podían realizar peticiones de prueba al servidor.

Para el desarrollo de la RESTful API se ha procedido a crear un archivo dentro del directorio *api* (introducido en 3.2) por cada modelo que necesite intercambiar datos mediante el uso de la API.

Los archivos creados forman los controladores del Modelo-Vista-Controlador por lo que en ellos se han desarrollado las diversas funcionalidades recogidas en los requisitos.

Como la funcionalidad generada consiste en el intercambio de información a través de peticiones, tras desarrollar cada función se procedía a un testeo manual, en el que se realizaba la petición correspondiente, para comprobar si se devolvían y/o recibían los datos correctamente.

Es decir, si se quería por ejemplo comprobar que se estuviera devolviendo correctamente la información de los usuarios, se usaba la URL que se había generado en el controlador para acceder y revisar que los datos de usuario estaban siendo correctamente retornados en formato JSON.

Cada vez que un modelo o controlador ha sido generado o editado, se ha procedido a guardar los cambios con git y a subir dichos cambios a la plataforma de GitHub para su clasificación en el repositorio.

Implementar la imagen en el servidor de la empresa resultó ser una tarea sencilla, gracias a las dos semanas que se dedicaron a la configuración. Para que la aplicación funcionase correctamente se instaló Docker, y se subió la imagen funcional del proyecto. Con un comando la aplicación se ejecutaba en el servidor, con los puertos y conexiones correspondientes y en modo producción.

5.2. Testing

Para comprobar que el código funcione tanto en el momento de creación como posteriormente según se van añadiendo nuevas funcionalidades, se ha de usar alguna herramienta que permita llevar el control de la funcionalidad. Para ello en este proyecto se ha usado la herramienta *unittest*, un paquete incluido en Python que permite la creación de tests de una forma simple.

Aunque en la arquitectura se diseñó cada modelo, como Welcome.py (introducido en 3.2), en un archivo separado, para los tests de los modelos se ha recopilado todos ellos en un único archivo tests.py, ya que como se ha mencionado anteriormente la idea de estos tests es comprobar si las funciones nuevas y las anteriormente programadas funcionan correctamente. Por ello, como con cada nueva función implementada hay que probar el resto, tiene sentido tenerlos recopilados.

A continuación se analizan dos ejemplos de casos sencillos de test realizados para la revisión del funcionamiento:

- **User**: algo que se debería comprobar siempre es si las contraseñas de los usuarios están siendo encriptadas. Para ello, se ha creado una clase con tres funciones, las dos primeras repetidas para todos los tests. La primera función, *setUp*, establece una conexión con una base de datos SQLite (base de datos empleada durante parte del desarrollo, antes de cambiar a MariaDB), y crea la base de datos. La segunda función, *tearDown*, se encarga de borrar los datos y cerrar la conexión.

  Por último, la función *test_password_hashing* comprueba la encriptación de la contraseña. Para ello, crea un objeto usuario usando el modelo y recibe como parámetros el email de ejemplo (*me@email.com*) y la contraseña (*canYouGuess?*), que encripta en el proceso de creación del objeto. A continuación intenta guardar el usuario en la base de datos, y posteriormente comprueba que la contraseña de usuario no es *canYouGuess?*. Si la contraseña no coincide, se entiende que la encriptación ha sido satisfactoria. Para comprobarlo, se emplea una función de verificación de contraseña para comprobar que, a pesar de estar encriptada, la contraseña original sigue siendo *canYouGuess?*. En caso afirmativo, se realiza una última comprobación: que la función de verificación de la contraseña no confirme una contraseña errónea (por ejemplo, *password123*), como correcta.

  Si esa función no da errores, querrá decir que se ha creado un usuario con contraseña cifrada y se ha guardado, y además se ha comprobado que la contraseña guardada es la establecida por el usuario.

```
class UserModelCase(unittest.TestCase):
    def setUp(self):
        app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite://'
        db.create_all()

    def tearDown(self):
        db.session.remove()
        db.drop_all()

    def test_password_hashing(self):
        u = User(email='me@email.com',password="canYouGuess?")
        db.session.add(u)
        self.assertIsNot(u.password, 'canYouGuess?')
        self.assertTrue(u._verify_password("canYouGuess?"))
        self.assertFalse(u._verify_password('password123'))
```

- **Unavailable**: este modelo se encarga de llevar el control de disponibilidad de pisos y habitaciones, cambiando por mantenimiento, ocupación u otros motivos. Los objetos de esta clase no deben de tener una fecha de fin de "no disponibilidad" más temprana que la fecha de inicio. En este ejemplo se encuentran de nuevo tres funciones: las dos primeras, que se repiten, y la tercera, *test_initial_unavailable*. No es necesario guardar el dato en la base de datos, ni relacionar los objetos con los alojamientos afectados; se pueden comprobar únicamente las fechas.

  Para ello, es necesario tener en cuenta tres escenarios: que la fecha de inicio sea posterior a la de fin, que sean iguales, y que la fecha de inicio sea anterior a la de fin. Se crean tres objetos, cada uno con un caso, y se comprueba que el primer objeto devuelva un error. En el segundo caso, como en los requisitos se establece que es posible ocupar un alojamiento un día, no se devuelve error. El tercer caso es el correcto.

```
class UnavailableModelCase(unittest.TestCase):
    def setUp(self):
        app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite://'
        db.create_all()

    def tearDown(self):
        db.session.remove()
        db.drop_all()

    def test_inital_unavailable(self):

        u1 = Unavailable(start=datetime.now(),end=(datetime.now()
        -timedelta(days=1)))
        u2 = Unavailable(start=datetime.now(),end=datetime.now())
        u3 = Unavailable(start=datetime.now(),end=(datetime.now()
        +timedelta(days=364)))

        self.assertFalse(u1.start_date_before_end())
        self.assertTrue(u2.start_date_before_end())
        self.assertTrue(u3.start_date_before_end())
```

Como se han de comprobar todos los tests, el programa muestra por consola todos los tests realizados, con un *ok* en cada uno si funcionaron. Al final muestra información sobre el tiempo de ejecución y el número de tests ejecutados. Además, como la ejecución de tests es acumulativa, al final muestra un *OK* si funcionaron todos los tests, para saber sin necesidad de recorrer la lista si ha habido errores.

test con fallos

```
test_inital_agreement
(__main__.AgreementModelCase) ... ok
test_inital_ask
(__main__.AskModelCase) ... ok
test_inital_balance
(__main__.BalanceModelCase) ... ok
                ...
test_password_hashing
(__main__.UserModelCase) ... FAIL
test_inital_welcome
(__main__.WelcomeModelCase) ... ok

======================================
FAIL: test_password_hashing
(__main__.UserModelCase)
--------------------------------------
Traceback (most recent call last):

File "tests.py", line 47,
in test_password_hashing
self.assertTrue(u._verify_password("canYouGues"))
AssertionError: False is not true

--------------------------------------
Ran 28 tests in 10.988s

FAILED (failures=1)
```

test sin fallos

```
test_inital_agreement
(__main__.AgreementModelCase) ... ok
test_inital_ask
(__main__.AskModelCase) ... ok
test_inital_balance
(__main__.BalanceModelCase) ... ok
                ...
test_password_hashing
(__main__.UserModelCase) ... ok
test_inital_welcome
(__main__.WelcomeModelCase) ... ok
--------------------------------------
Ran 28 tests in 11.728s

OK
```

6 Montaje en servidor y lanzamiento

> «After launching the first version of Facebook for a few thousand users, we would discuss how this should be built for the world. It wasn't even a thought that maybe it could be us. We always thought it would be someone else doing it.» Atribuida a Mark Zuckerberg.

Mark Zuckerberg lanzó Facebook inicialmente a un grupo pequeño de usuarios, antes de planificar los siguientes pasos. Lo mismo debe hacer HousingOn con esta API.

6.1. Montaje en el servidor

Para poder subir la aplicación a internet, lo primero que se necesita es un espacio donde alojarla. Por el tipo de aplicación que se ha desarrollado, la variedad de sitios y formatos de alojamiento es inmensa. A continuación se comentan los que mayor probabilidad tienen de ser empleados por la empresa.

La primera distinción que se ha de realizar es si el proyecto debería de ser alojado en un servidor físico propio o escoger una opción de alojamiento en otra empresa.

6.1.1. Servidor propio

La principal ventaja de tener un servidor propio es la privacidad que este otorga, ya que los datos estarán siempre bajo control de la empresa. Lo mismo pasa con las copias de seguridad; con un servidor propio, la empresa puede poner en marcha su propio sistema privado de copias.

Además podrá escoger el software que quiera utilizar, y diseñar el servidor completamente a medida, con el hardware deseado, de forma que sea óptimo para la empresa.

Sin embargo, este tipo de servidor también cuenta con una serie de desventajas:

Los costes iniciales son elevados en comparación con un servicio de alojamiento externo. Puede que a la larga los costes de un servidor propio terminen siendo menores que los de un alojamiento externo, pero en muchas ocasiones esto no ocurrirá.

Hay que tener en cuenta que al montar un servidor físico es necesario comprar todos los componentes, sumando además los costes del tiempo dedicado a escogerlos; la empresa debe de asignarle un espacio, con unas condiciones de refrigeración y aislamiento de sonido óptimas, todo ello suponiendo que el tamaño físico del servidor no sea mucho más grande que un ordenador de sobremesa, ya que si no probablemente la empresa deba de ajustar la instalación eléctrica para soportar el consumo del servidor. Por último habrá que mejorar la conexión de red de la empresa para que esta no resulte ser un cuello de botella en la recepción y envío de peticiones.

Tras preparar todo lo anterior, se puede proceder a su montaje, lo que también sumará un coste de tiempo. Una vez se encuentre todo montado se podrá proceder a la instalación del sistema operativo y del software necesario, y la configuración de ambos para que el servidor pueda empezar a ejecutar la aplicación. Una vez se tengan software y hardware configurados, se podría subir y ejecutar la aplicación.

Todo esto forma parte del coste inicial, pero a lo largo del tiempo hay que añadir costes de mantenimiento, actualizaciones, costes de electricidad e internet.

Si se tiene en cuenta que actualmente la empresa no cuenta con empleados suficientes para mantener el servidor, el gasto de este tipo de servidor es prácticamente inasequible para una startup.

Por lo tanto este tipo de alojamiento queda descartado.

6.1.2. Alojamiento externo

En este caso la empresa pierde la privacidad de sus datos, ya que pasan a pertenecer a la empresa a la que le alquila el alojamiento, pero a cambio se ahorra el desembolso en costes iniciales.

En cuanto a alojamiento externo se refiere existen muchos posibles servicios a escoger además de empresas de alojamiento. A continuación se listan posibles opciones para el alojamiento de la aplicación:

- **Alojamiento web**: también conocido como alojamiento compartido, es la opción más sencilla. La empresa ofrece al usuario acceso a una interfaz en la que puede administrar los archivos de la aplicación, y algunas configuraciones sencillas del servidor. En concreto, aquellas relacionadas con las directrices de su aplicación, o con el uso de algunos módulos del servidor. Es la opción más rápida de configurar, siempre y cuando su software sea adaptable con el del servidor de la empresa.

  Como desventaja se puede señalar que, al tratarse de un alojamiento compartido, los recursos totales disponibles han de dividirse entre todos los clientes que hayan contratado servicios de alojamiento web. Como la aplicación del cliente siempre dependerá de los recursos que provea la empresa de alojamiento, en caso de crecimiento pueden dejar de ser suficientes. Tampoco se podrá ajustar el servidor a la aplicación, ya que las configuraciones que se pueden hacer son básicas. Por último, la empresa contratada debería tener sus servidores situados cerca de los usuarios de la aplicación del cliente, porque en otro caso las latencias pueden volverse un problema.

- **Servidor Virtual Privado**: más reconocido por sus siglas en inglés VPS (Virtual Private Server), es una opción algo más compleja. En este caso se contrata un hardware, es decir, almacenamiento, capacidad de procesamiento, etc. Esto, en realidad, es ficticio, ya que lo que la empresa hace es generar instancias de servidores en un servidor mayor, alquilando cada una de esas estancias a un cliente distinto. Normalmente, las empresas que ofrecen estos servicios permiten al cliente escoger el sistema operativo a instalar de una lista.

  A diferencia del alojamiento web, una vez que todo esté listo, aquí se deberá acceder virtualmente al servidor para realizar la configuración correspondiente a la aplicación. Por lo tanto, esta parte es común con tener un servidor propio.

  Los VPS requieren una inversión de tiempo mayor que el alojamiento web al principio, pero la única limitación que ejercen es el hardware que el cliente ha alquilado, y habitualmente esto puede ser sujeto de mejoras en caso de crecimiento de la aplicación. Otra diferencia que existe entre los VPS y tener un servidor propio es que se pueden contratar copias de seguridad externas. De este modo, si el VPS falla, se puede contratar un nuevo servicio e instalar la última copia de seguridad en él, solucionando el problema en cuestión de minutos.

Este servicio tiende a ser más caro que el alojamiento web. Sin embargo, pueden encontrarse opciones alrededor de tres euros mensuales, un gasto considerablemente inferior al de un servidor propio. Se pueden contratar servidores estándar o servidores dedicados; estos últimos son más rápidos, y por tanto más caros de alquilar. También es importante contratar un servidor cercano a los usuarios de la aplicación, para disminuir las latencias.

- **Web Apps en Cloud**: esta opción es similar a los alojamientos web. Está pensada para subir la aplicación al servidor y que comience a funcionar automáticamente. En la práctica, sin embargo, esto no es del todo cierto, ya que es probable que la aplicación necesite adaptarse a este tipo de servicios. Es muy popular en sistemas como Azure, Google Cloud o Amazon Web Services.

  Si se utiliza este servicio, se cuenta con la posibilidad de que la aplicación se distribuya entre los distintos servidores de la empresa que ofrece el servicio. De este modo, el cliente siempre es atendido por su servidor más cercano, solucionando el problema de latencias. Un gran atractivo para muchas empresas es que en estos servicios solo se paga lo que se consume.

  La principal desventaja de estos servicios es el coste, ya que suele ser muy superior al de los VPS. Además, calcular los costes es complicado, ya que se necesita estimar de antemano cuántos recursos se van a consumir.

- **Contenedores como servicios en Cloud**: esta opción es idéntica a la anterior, con la salvedad de que en lugar de manejar la aplicación se manejan contenedores. En este caso, el contenedor Docker que se ha creado durante el desarrollo se utiliza para evitar problemas de configuración. Este sistema es muy rápido, con opciones de escalabilidad, replicables bajo demanda y de pago por uso. Como en los anteriores, suelen ser más caros que los VPS.

Tras el previo análisis de estás opciones y sabiendo que la empresa contaba ya con un VPS, la opción tomada para el lanzamiento es este servicio. No obstante la aplicación está preparada para utilizar "contenedores como servicios" en un *cloud*, en caso de aumento considerable de los clientes. La primera opción fue descartada por los problemas de latencia que puede ocasionar y por la falta de configuraciones que tienen este tipo de servicios. En cambio la tercera opción fue descartada porque, teniendo ya un contenedor creado, es más sencillo implementar la cuarta opción teniendo ambas las mismas ventajas y desventajas.

6.2. Lanzamiento

En cuanto al lanzamiento, la aplicación será lanzada en una primera etapa tan solo para el uso de desarrolladores de HousingOn, es decir para que otros programadores pueden empezar a desarrollar sus apps en torno a ella. La intención de esta primera etapa es que en caso de descubrir fallos puedan ser corregidos antes de que puedan afectar a más usuarios. Además es un buen momento para realizar optimizaciones y análisis, para comprobar que la API pueda escalar correctamente.

Una vez los desarrolladores hayan creado sus aplicaciones comenzará la segunda fase. En ella, los trabajadores de HousingOn deberán utilizar las aplicaciones desarrolladas usando la API, para analizar el funcionamiento de ambas. En caso de haber algún fallo con la interfaz, este podría ser detectado y corregido antes de pasar a la siguiente fase. Los cambios en la API en esta etapa pueden afectar a las aplicaciones. Debido a ello, en función del cambio necesario, se deberá contemplar si es mejor realizarlo, o generar una nueva versión de la API para que futuras aplicaciones cuenten con ese error corregido.

En la tercera fase, los clientes de HousingOn (inquilinos, propietarios y personal de mantenimiento) podrán empezar a usar las aplicaciones. En esta fase se podrá realizar otro análisis para comprobar

la escalabilidad de la aplicación, buscar los cuellos de botella y corregir posibles errores que no se hubieran detectado anteriormente.

La última fase consiste en replicar el sistema para que otras empresas de alquileres puedan emplearlo. Esta fase es importante, ya que al replicar podrían producirse fallos en los ajustes que tendrían que ser corregidos.

Con estas cuatro etapas se pretende conseguir el despliegue total de la aplicación, de forma controlada y detectando los posibles fallos importantes al comienzo.

7 Conclusiones

> «It's important in life to conclude things properly. Only then can you let go. Otherwise you are left with words you should have said but never did, and your heart is heavy with remorse.» [Martel, 2001]

El objetivo de este proyecto era realizar una nueva aplicación para la empresa que sustituyese la funcionalidad de la actual.

El proyecto debía solucionar todos los problemas de concepto de la aplicación actual de gestión de la empresa HousingOn, ya que de otra forma la compañía vería mermado el rendimiento de su negocio debido a la ineficacia de esta.

Actualmente su aplicación está basada en web, utilizando varios portales desarrollados usando el lenguaje de programación PHP 5, que es una versión ya antigua de este lenguaje, así como otras tecnologías que causaban un difícil mantenimiento y escalabilidad de la aplicación.

Por estas razones, no podían desarrollar otros proyectos que tienen en mente, como por ejemplo poder realizar la revisión de salida de los inmuebles desde una aplicación móvil.

La creación de la RESTful API ofrece una base a los desarrolladores para construir una nueva aplicación que pueda cubrir las necesidades actuales de la empresa, así como también ofrece flexibilidad para las posibles necesidades futuras. Por la naturaleza de la empresa se ha favorecido el uso de proyectos de software de código abierto, en todo caso seleccionando los que se consideraban suficientemente maduros respecto a su adopción por un gran número de usuarios.

Por otro lado, se han estudiado distintas alternativas para su despliegue, y con esto en mente, se ha elegido la tecnología de contenedores para que la portabilidad sea más sencilla y permita su crecimiento con el número de usuarios. Finalmente, su creación deja abierto el uso de los datos generados para su uso más avanzado con el uso de conectores que permitan el uso de técnicas de minería de datos o de sistemas expertos para el apoyo a la toma de decisiones.

7.1. Conclusiones

El proyecto ha presentado una serie de dificultades iniciales debido a que la aplicación actual es inmensa, porque llevan años desarrollándola. Esto produce que la API tenga que ser enorme desde un inicio, todo un reto debido al tiempo que se le ha dedicado a la aplicación.

Además todos los conocimientos han sido obtenidos por el método de auto-aprendizaje, por lo que el avance ha sido más lento que si se hubiera tenido a quién consultar las dudas que iban surgiendo.

A tono personal, me gustaría mencionar que la realización de esta memoria me ha servido para darme cuenta de la importancia de planificar previamente los temas a tratar, para poder estructurar

el documento conforme a los requisitos. Además, tener que explicar conocimientos adquiridos durante el grado a la vez que tener que diseñar mis propios esquemas de forma que sean entendibles y con un patrón de diseño similar, me ayuda a apreciar el esfuerzo que requiere realizar documentos técnicos.

Ser capaz de llevar a cabo un proyecto de tal magnitud con ilusión me hace darme cuenta de la capacidad que tiene el ser humano para afrontar grandes retos.

He estado encantado de poder aplicar todos mis conocimientos para ir resolviendo las diferentes dificultades. Me gusta que la API haya sido desarrollada en Python, ya que me parece un lenguaje con muchísimo potencial y del cual me gustaría especializarme. Además, me llena de orgullo saber que mi aplicación será usada por la empresa en un futuro cercano, y que por lo tanto una creación mía está siendo útil, ayudando a la empresa a llevar el control sobre los inmuebles.

Lo más curioso de este proyecto es que he terminado adquiriendo un conocimiento general bastante extenso sobre el funcionamiento de toda la empresa; he aprendido sobre problemas frecuentes que suelen ocurrir en inmuebles de estudiantes, recomendaciones en la compra de electrodomésticos para pisos de alquiler, incluso para poderme hacer una idea aproximada de como son las distintas zonas de Santander en cuanto a inmuebles se refiere.

7.2. Futuro del sistema

El siguiente paso sería poder automatizar el *testing* de los *controllers*, ya que como se ha mencionado anteriormente, en el momento de redactar este trabajo se realizan de forma manual.

Este sistema se podría desarrollar con la tecnología que se usa para el *testing* de los modelos, añadiendo un paquete que permita realizar y obtener peticiones del servidor.

También sería útil contar con herramientas que ayuden con la generación de la documentación de la API, como Swagger, una herramienta libre que está disponible para multitud de lenguajes de programación, entre ellos Python.

En cuanto a la RESTful API, el siguiente paso a realizar sería poder terminar toda su funcionalidad y mejorarla con el *feedback* que den los desarrolladores de las aplicaciones que hagan uso de esta API.

Actualmente la API GraphQL requiere un esfuerzo adicional para poder seguir creando funcionalidad para la empresa, por lo que esta tarea debería de ser continuada hasta alcanzar su completitud.

Además la API no es muy útil para HousingOn si no se crean aplicaciones posteriores basadas en ésta, de forma que puedan manejar la información de una forma visual.

En un futuro cercano la empresa debería continuar creando una interfaz web, para poder sustituir los portales web que tiene actualmente. Posteriormente podrían empezar con el proyecto de la aplicación móvil, que llevan tiempo queriendo desarrollar, pero que por impedimentos de su actual sistema la aplicación no podía ser desarrollada.

A Anexo

A.1. UML casos de uso

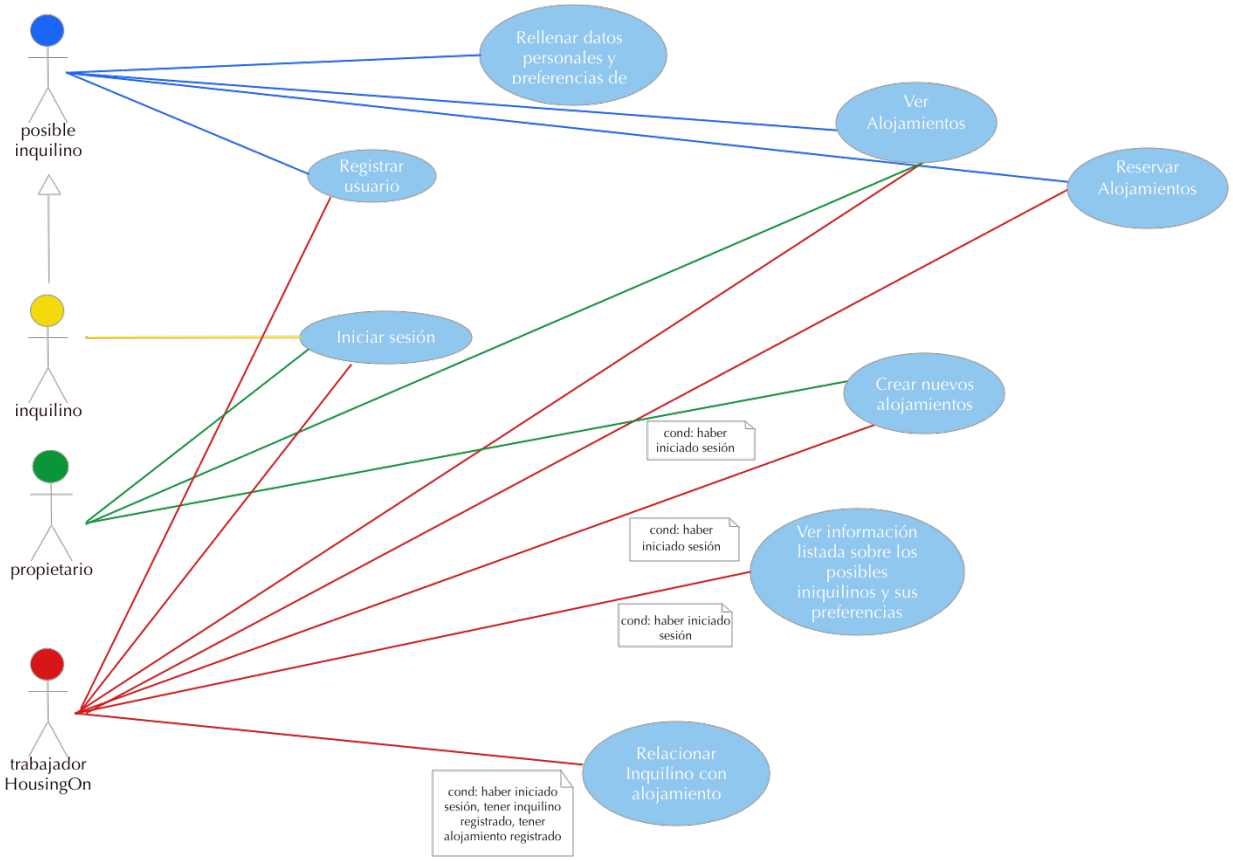

Ilustración 24: UML con casos de uso mencionados en los requisitos

A.2. Especificaciones casos de uso

Cuadro 3: RF1. Registro de posibles inquilinos y intereses de alojamiento

Función	Este registro deberá recoger los datos de contacto de un usuario, y el tipo de alojamiento que está buscando.
Descripción	Este registro sirve como primera toma de contacto entre un posible inquilino y la empresa. Es importante saber qué tipo de alojamiento está buscando, si querrá compartirlo o será privado, por qué zona o ciudad, y el presupuesto que tenía pensado invertir en el alojamiento además de datos como la edad, el nombre y el email del usuario.
Entradas	Tipo de alojamiento: compartido, privado y otros, compartir con: solo chicos, solo chicas, amig@s o indiferente, fecha aproximada de llegada, fecha aproximada de salida, presupuesto mensual, zona/ciudad, idioma de contacto, género, rango de edad, otra información que el usuario quiera añadir; datos de contacto: nombre, email, teléfono y preferencia de contacto.
Fuente	Esta información se recibirá en una petición HTTPS mediante el método POST en la API desde un cliente.
Salidas	Informará al cliente sobre si los datos han sido correctamente almacenados.
Destino	Cliente de la API que ha usado la función.
Acción	Se debe comprobar que los datos recibidos no existen actualmente en la base de datos y si los datos recibidos se encuentran entre los posibles valores. Por ejemplo, que el tipo de alojamiento se encuentre entre los tres casos expuestos, que la fecha de llegada sea inferior a la de salida, y que ambas sean mayores al día en el que se cumplimenta, que el presupuesto sea mayor o igual que el mínimo establecido... Una vez realizadas todas estas comprobaciones, si todo es valido, se deberá proceder al guardado en la base de datos y devolver al cliente que los datos se han almacenado correctamente. En caso de que algún dato no sea correcto se deberá devolver que los datos no se han podido guardar correctamente debido a un error en los mismos, y en caso de fallo en la base de datos, se deberá informar que los datos han fallado al intentar guardarlo en la base de datos.
Requerimientos	Los datos deben de enviarse desde el cliente en formato JSON, con los campos obligatorios, como el email, cumplimentados.
Precondición	Ninguna.
Postcondición	Ninguna.
Efectos secundarios	Ninguno.

Cuadro 4: RF2. Listado de información de posibles inquilinos e intereses de alojamiento

Función	Deberá de mostrarse la información recogida por el *Registro de posibles inquilinos*.
Descripción	Esta función deberá de recopilar los datos de los posibles inquilinos, ordenados por fecha, del más reciente al más antiguo, para su retorno.
Entradas	Se recibirá una petición HTTPS mediante el método GET. La petición puede contener un número de página y la cantidad de resultados que deben de ser devueltos.
Fuente	Esta petición se recibirá en la API desde un cliente.
Salidas	Devolverá la información de los posibles inquilinos, según las preferencias de la petición, ajustándose a los criterios de paginación y número de resultados, si son especificados.
Destino	Cliente de la API que realizó la petición.
Acción	Se debe comprobar que los usuarios están autenticados para poder devolver los datos; en caso de no estarlo se devolverá una respuesta en formato JSON informando de este requisito al cliente. En caso de estarlo se deberá obtener de la petición la paginación y el número de resultados que deben ser devueltos. Si no se recibe un página pero se recibe un número de resultados, se considerará que el número de página es la primera; en caso de recibir un número de página pero no recibir un número de resultados, se devolverá el número de resultados fijados en la configuración de la aplicación; en caso de recibir ambos datos, se calculará los resultados a devolver según los criterios recibidos, y en caso de no recibir ninguno deberá de devolverse todos los datos de posibles inquilinos Todos los resultados deberán de ser retornados en formato JSON.
Requerimientos	El usuario del cliente de la API debe estar autenticado como personal de la empresa.
Precondición	Debe existir información de posibles inquilinos.
Postcondición	Ninguna.
Efectos secundarios	Ninguno.

Cuadro 5: RF3. Relacionar un inquilino con un alojamiento

Función	Relacionar a un inquilino con un alojamiento.
Descripción	Esta función debe crear una relación entre un usuario inquilino y un alojamiento, de forma que este pase a estar ocupado por el usuario. Además, según el criterio de la empresa, se informará mediante un email al propietario del día de llegada del nuevo inquilino. Mientras el alojamiento esté alquilado se deberá indicar que no está disponible. El inquilino deberá recibir una confirmación mediante email, avisando de que se le ha ocupado un alojamiento, indicando los datos del mismo.
Entradas	Se recibirá una petición HTTPS mediante el método POST. La petición contendrá el identificador del alojamiento y el identificador del usuario en formato JSON.
Fuente	Esta petición se recibirá en la API desde un cliente.
Salidas	Se retornará un documento JSON, además de los emails enviados al inquilino y al propietario. En caso de éxito, ese JSON contendrá la relación entre cliente y alojamiento, además de la fecha en que ha sido creada y/o actualizada. En caso de error, el documento contendrá un aviso para el cliente del fallo en la relación.
Destino	Cliente de la API que realizó la petición.
Acción	Se debe comprobar que el usuario está autenticado como personal de la empresa. Se obtendrán los identificadores de usuario y alojamiento y se creará la relación entre ambos, guardándose en la tabla Roomusers de la base de datos. A su vez, se deberá consultar las fechas de llegada y salida del usuario en la base de datos para poder realizar un cambio en la disponibilidad del alojamiento. Se procederá posteriormente al envío de dos emails, informando al propietario de la llegada del inquilino y al inquilino del comienzo de su contrato de alquiler. También se responderá al cliente que realizó la petición con un JSON con la relación entre inquilino y alojamiento, incluyendo las fechas de creación y actualización.
Requerimientos	El usuario del cliente de la API debe estar autenticado como personal de la empresa para poder recibir la información.
Precondición	Debe existir información sobre el alojamiento y el inquilino para poder crear la relación.
Postcondición	Ninguna.
Efectos secundarios	Se modificará el estado del alojamiento a no disponible durante la duración de la instancia del inquilino.

A.3. Ejemplo de modelo desarrollado en Python

Welcome.py

```
import datetime

from app import app,db
from app.utils import PaginatedAPIMixin
from flask import url_for
from .Roomuser import Roomuser
from .User import User


class Welcome(PaginatedAPIMixin,db.Model):
    __tablename__ = 'Welcomes'

    id = db.Column(db.Integer, primary_key=True, nullable=False)
    roomuser_id = db.Column(db.Integer,db.ForeignKey('Roomusers.id'))
    meeting_date = db.Column(db.DateTime, index=True,
    default=datetime.datetime.now())
    meeting_point= db.Column(db.Text(100))
    pick_up = db.Column(db.Boolean, default=False)
    transport = db.Column(db.String(50), nullable=False, default=0)
    transport_reference = db.Column(db.String(80))
    note = db.Column(db.Text)
    has_been_picked_up = db.Column(db.Boolean, default=False)
    sign_agreement = db.Column(db.Boolean, default=False)
    picker_id = db.Column(db.Integer, db.ForeignKey('Users.id'))
    created_at = db.Column(db.DateTime, default=datetime.datetime.now())
    updated_at = db.Column(db.DateTime, default=datetime.datetime.now(),
    onupdate=datetime.datetime.now())

    roomuser = db.relationship("Roomuser", foreign_keys = [roomuser_id],
    backref = "welcome_roomuser" )
    picker = db.relationship("User", foreign_keys = [picker_id],
    backref = "welcome_picker")
```

Glosario

API La Interfaz de Programación de Aplicaciones o API de las siglas en inglés *Application Programming Interface* es una capa de abstracción de forma que un conjunto de funcionalidades, rutinas y procedimientos de una aplicación o servicio pueden ser usadas en otros programas o aplicaciones. Por ejemplo, OpenGL permite la representación de gráficos 2D y 3D además de ser multilenguaje y multiplataforma, por lo que un programa podrá hacer uso de esta API para representar elementos gráficamente. . 1–3, 5, 9, 11–14, 16, 18, 20, 25, 29, 33, 35, 37, 38, 40–42

app Es una abreviatura del inglés *application*, es decir una aplicación, o conjunto de funciones que conforman una utilidad para el usuario. Por ejemplo, una aplicación de mensajería le permitirá al usuario enviar y recibir mensajes. . 1, 12

backend Término empleado para designar la parte del software utilizada de forma habitual por programadores y administradores, y que no suele estar al alcance del usuario final. También se emplea para referirse a la parte del servidor, en arquitecturas cliente-servidor. Se suele emplear en relación al acceso a la base de datos. . 1, 21

base de datos Conjunto de datos con un contexto en común y clasificados de forma sistemática con la intención de ser utilizados posteriormente. . 1

business intelligence Inteligencia empresarial o inteligencia de negocios en castellano, hace referencia al conjunto de estrategias, aplicaciones, datos, etc., que puede ser analizado por la empresa para poder obtener conocimiento que permita la toma de decisiones de la empresa. . 1, 23

contenedor También conocido como *"virtualización a nivel de sistema operativo"*, es una unidad de software estándar que envuelve todo el código y sus dependencias de una aplicación, de forma que la aplicación pueda ser ejecutada de forma eficaz y rápida en cualquier entorno. Por ejemplo, con un contenedor software se puede trasladar la aplicación desde un ordenador de desarrollo a un servidor en producción sin necesidad de configuraciones extra, incluso aunque cada ordenador este siendo ejecutado con un S.O. distinto. . 1, 35

debugging Conocido en castellano como proceso de depuración de programas, es el proceso de hallar fallos o problemas y resolverlos con el objetivo de lograr el correcto funcionamiento de la ejecución del programa. . 1, 25

entrevista abierta En este caso la entrevista no tiene unas cuestiones predefinidas, sino que se tienen unos o varios temas base de los que van surgiendo dudas y posibles respuestas. . 1, 5

entrevista cerrada Este tipo de entrevista está relacionado con los cuestionarios, pues las preguntas que se realizan en este tipo de entrevista están predefinidas. . 1, 5

fork Su término en castellano es bifurcación, y es usado en el contexto de git, para referirse a la realización de una copia exacta de un repositorio, de forma que cada copia pueda seguir un proceso de desarrollo distinto. . 1, 22, 23

funciones hash También conocidas como funciones resumen, son funciones que tienen como entrada un conjunto de elementos, normalmente cadenas de texto, y lo convierten en un rango de salida finito, normalmente cadenas de longitud fija. . 1, 23

Google Drive Servicio de alojamiento de archivos en la nube creado por Google. Además de almacenar archivos, puedes crear archivos propios de Google, como los documentos y presentaciones de Google. . 1, 6

hilo de ejecución Cada hilo de ejecución es una tarea que se puede realizar a la par que otra y que comparten los mismos recursos, y gracias a ello cualquier hilo puede modificar dichos recursos. Este término está estrechamente relacionado con la programación en paralelo, ya que los hilos y procesos son la base de esta. . 1, 21, 28

imagen Una imagen es un conjunto de archivos y configuraciones sin estado de la que se podrán instanciar contenedores para la ejecución de la aplicación. Por ejemplo, una imagen podría corresponderse con el código de un programa en Python sin ejecutar, mientras que un contenedor se correspondería con la ejecución de ese programa Python. . 1, 30

paginación Técnica que permite manejar la cantidad de información que ha de enviarse a la interfaz gráfica. De esta forma se gana en velocidad de transferencia y en la representación de los datos. Se conoce como paginación porque la idea original consistía en mostrar un número de resultados determinado en la interfaz gráfica, cargando nuevos resultados en páginas posteriores. Hoy en día son más comunes los sistemas de paginación como *cargar más*, que añade nuevos resultados bajo los ya mostrados, o el *scroll infinito*, que muestra los datos conforme se desciende en la página. Un ejemplo de este último sistema de paginación podría ser Twitter. Como se puede observar, estos sistemas hacen referencia a la vista de un sistema Modelo-Vista-Controlador, ya que para el controlador estos tres sistemas funcionan de la misma manera. El controlador solamente debe conocer cuántos resultados debe devolver para la nueva página, y cuáles ha devuelto anteriormente. Por ejemplo, si debe devolver una página con diez resultados, y se encuentra en la página número cinco, deberá saber que ha devuelto previamente los resultados uno a cuarenta, y devolver por tanto los resultados cuarenta y uno a cincuenta. . 1, 41

RESTful API RESTful es una arquitectura de software con semejanzas al modelo HTTP. REST es un acrónimo de *REpresentational State Transfer*. Al ser un API, su función será recibir datos provenientes de una aplicación y procesarlos mediante la correspondiente función devolviendo al final los resultados al programa que envió los datos originalmente. . 1, 3, 16, 22, 29, 37, 38

shell Es una interfaz que permite el acceso a los servicios del sistema operativo. Las *shells* pueden ser accedidas tanto por linea de comandos como por interfaces gráficas, y se conocen por este nombre por ser la capa más externa del sistema operativo. . 1, 25

startup Una empresa emergente de reciente creación, creada sobre una base tecnológica, innovadora y supuestamente con una elevada capacidad de rápido crecimiento. . 1, 8, 34

test A/B El término test A/B se utiliza en el ámbito del Marketing Digital y la Analítica web para describir experimentos aleatorios con dos variantes, A y B, una de control y otra variante. Otra forma de referirse generalmente a los test A/B es con el término *split test*, aunque este último método se aplica cuando se realizan experimentos con más de dos variantes. . 1, 20

VPS Son las siglas de *Virtual Private Server* (Servidor Virtual Privado). Es una manera de particionar un servidor en diversos servidores inferiores en recursos, conocidos como VPS, ya que son particiones de servidor pero pueden actuar como un servidor independiente. . 1, 34

Referencias

L. Bass, P. Clements y R. Kazman. Software architecture in practice (third edit., p. 624), 2012.

Y. Bugayenko. *Code Ahead: Volume 1*. CreateSpace Independent Publishing Platform, 2018.

C. A. R. Hoare. The 1980 ac m turing award lecture. *Communications*, 1981.

Y. Martel. *Life of Pi*. Knopf Canada, 2001.

A. H. Maslow. The psychology of science. 1966.

S. McConnell. *Software project survival guide*. Pearson Education, 1998.

D. Ocean. Comparison of web servers for python based web applications. https://www.digitalocean.com/community/tutorials/a-comparison-of-web-servers-for-python-based-web-applications

C. d. l. U. E. Parlamento Europeo. REGLAMENTO (UE) 2016/679. *Offic. J. Europ. Union*, pages 1–88, 2016.

I. Sommerville. *Software Engineering GE*. Pearson Australia Pty Limited, 2016.

M. van der Rohe. New york herald tribune, 28/June/1959.

Wikipedia. Docker (software). https://es.wikipedia.org/wiki/Docker_(software)